Expression Editor and Listen Node

Software Version: 1.80
Document Version: 1.00
Release date: 2020-08-21

TECHMAN ROBOT INC.

Contents

Revision History Table

Revision	Date	Revised Content
00	Aug, 2020	In addition to the previous software version, this version added: ● File functions ● Safety SI / SO of IO Parameterization ● Mode 11/12/13 Request read of TM Ethernet Slave ● More examples for TMSVR response messages ● GetToken () support to get last token by count is -1 ● RefCoordF3D and RefCoordT3D in FT ● The unit of X Y Z RX RY RZ in the math function ● Profinet functions ● Explanations of operations in progress by the types of the operands and modified: ● The unit of RX RY RZ in the motion functions such as PTP / Line / Move to ° ● The example description of modbus_read_string ● The description of TMSVR ● The example description of TMSVR binary transmission ● The modbus chapter and adjust it to the back of the motion command ● trans()/inversetrans()/applytrans() ● The note on Ethernet Slave 100Hz of no real-time guarantee

1. Expression

1.1 Types

Different data types of variable can be declared in Variables Manager.

byte	8bit integer	unsigned	0 to 255	significant digit 3
int	32bit integer	signed	-2147483648 to 2147483647	significant digit 10
float	32bit floating-point	signed	-3.40282e+038f to 3.40282e+038f	significant digit 7
double	64bit floating-point	signed	-1.79769e+308 to 1.79769e+308	significant digit 15
bool	boolean		true or false	
string	string			

For int type variable, both int16 and int32 are supported. The default type is int 32.

int16	16bit integer	signed	-32768 to 32767	significant digit 5
int32	32bit integer	signed	-2147483648 to 2147483647	significant digit 10

1.2 Variables and Constants

1. Variables

In the naming rule of variables, only the numbers, under line and the upper case and lower case English characters are supported.

Numbers 0123456789

Characters a-z, A-Z, _

Example

```
Int var_i = 0
string var_s = “ABC”
string var_s1 = “DEF”
string var_s2 = “123”
```

Without double quotation marks, strings will be taken as variables.

```
var_s = var_s1 + “ and “ + var_s2        // var_s = “DEF and 123”
                                        // var_s, var_s1, var_s2 are variable, and “ and “ is a string.
```

In addition to variables, the naming rule also applies to constants, numbers, strings, and Booleans except that string constants need to be enclosed in double quotes.

When a variable is generated in TMflow, a prefix is added based on the source. To use the variable for writing or reading, users must enter the full name including the prefix word. For the rules of adding prefixes, refer to the respective description in variable setting pages.

2. Numbers

- Decimal integer, decimal floating-point, binary, hexadecimal integer and scientific notation are supported.

Decimal integer	123
	-123
	+456
Decimal float	34.567
	-8.9
Binary	0b0000111
	0B1110000
Hexadecimal integer	0x123abc
	0X00456DEF
Scientific notation	3.4e5
	2.3E-4

- For binary and hexadecimal notation, there is no floating-point.
- The notation of number is not case sensitive.

  For example:

  0b0011 equals to 0B0011

  0xabcD equals to 0XABCD, 0xABCd, 0Xabcd etc.

  3.4e5 equals to 3.4E5

- The transforming between floating-point and byte array may cause discrepancy in value

  For example:

  float 5.317302E+030 ➔ float to byte[] {0x72,0x86,0x3A,0x42}

  byte[] {0x72,0x86,0x3A,0x43} ➔ byte[] to float 5.317302E+030

- Byte can only present unsigned numbers from 0 to 255. As a result, if negative number is assigned to

byte type variable directly or through calculation, only 8 bit unsigned value will be kept.

For example:

byte var_b = -100 // var_b = 156 // -100 is present as 0xFFFFFF9C by 16bit notation.

// Because byte can only keep 8 bit data, that is 0x9C (156), b will equals to 156

3. String

When inputting string constant, double quotation marks shall be placed in pairs around the string to avoid the recognition error of variable and string. ◦

For example

“Hello World! “

“Hello TM””5” (If “ is one of the character in the string, use two (“”) instead of one (”).

- Control character in double quotation mark are not supported.

  For example:

  “Hello World!\r\n” (the output would be **Hello World!\r\n** string)

- Without double quotation marks, the compiling will follows the rules below
  1. Numbers will be view as numbers
  2. The combination of numbers and characters will be view as variable as long as the variable does exist.
  3. If the variable does not exist, it will be compiled as string with warning message.
- The combination of string and variable
  1. Inside double quotation marks, variables will not be combined as variables

     For example:

     var_s = “TM5” // var_s = ”TM5”

     var_s1 = “Hi, s Robot” // var_s1 = ”Hi, s Robot”
  2. To input the combination of variables and strings, double quotation marks needs to be placed around the string, and plus sign (+) shall be used to link variables and numbers

     Example:

     var_s1 = “Hi, “ + var_s + “ Robot” // var_s1 = ”Hi, TM5 Robot”
  3. To be compatible with the old version software, the single quotation marks can be placed around the variables, but a warning message will be send out

     For example:

single quotation marks “Hi, ’s’ Robot” // var_s1 = ”Hi, TM5 Robot”

“Hi, ’x’ Robot” // var_s1 = ”Hi, ‘x’ Robot” // Because variable x does not exist, ‘x’ is viewed as string

4. Single quotation marks do not support element value retrieval with array indexes. The standard format with double quotation marks should be used.

   For example

   string[] var_ss = {“Techman”, “Robot”}

   “Hi, ‘var_s’ ‘ss[0]’ Robot” // var_s1 = ”Hi, TM5 ‘ss[0]’ Robot” // ‘ss[0]’ is invalid

   “Hi, “ + var_ s + “ “ + var_ss[0] + “ Robot” // var_s1 = ”Hi, TM5 Techman Robot”

5. Single quotation marks cannot be presented by ‘”. If users would like to input ’(variable name)’, The standard format with double quotation marks should be used.

   For example

   “Hi, ‘s’ Robot” // var_s1 = ”Hi, TM5 Robot”

   // If var_s1 = ”Hi, ‘s’ Robot” is what you want, please use the following syntax.

   “Hi, ‘” + “s” + “’ Robot” // var_s1 = “Hi, ‘s’ Robot”

- For control character, e.g. new line, please use Ctrl() command.

  For example

  var_s1 = “Hi, “ + Ctrl(“\r\n”) + s + “ Robot” or “Hi, “ + NewLine + var_s + “ Robot”

  Hi,

  TM5 Robot

- Reserved characters is similar to variables, no double quotation marks is needed. (But single quotation mark is not supported)

  1. empty empty string, equals to “”
  2. newline or NewLine new line, equals to Ctrl(“\r\n”) or Ctrl(0x0D0A)

4. Boolean

True or false value of logic.

Denote true value	true True	Denote false value	false False

The Boolean value is case sensitive. Misuses of capital letters such as TRue will be taken as a variable or a string.

1.3 Array

- Array is a set of data with the same data type. The initial value is assigned with {}, and every element remains the characteristic of its data type.

  For example

  ```
  int[] var_i = {0,1,2,3}                    // elements in number data type
  string[] var_s = {“ABC”, “DEF”, “GHI”} // elements in string data type
  bool[] var_bb = {true, false, true}        // elements in boolean data type
  ```

- By utilizing index, the value of specified element can be get, the index is start from 0

  For example

  index 0 1 2 3 4 5 6 7

  array  eight elements in total

  A[0] A[1] A[2] A[3] A[4] A[5] A[6] A[7]

  Valid index values [0] .. [7], an error will occur with invalid index number.

- Only one degree array is supported. The maximum index number is 2048.

- The array size may alter according to the return value of functions or assigned values. The maximum element number is 2048. This feature makes array meet the needs of different functions and applications in Network Node.

  For Example:

  ```
  string[] var_ss = {empty, empty, empty}          // The initial size of string array is 3 elements
  var_ss = String_Split(“A_B_C_D_F_G_H”, “_”)  // After splitting string, the string array has 7 elements
  var_len = Length(var_ss)                              // var_len = 7
  var_ss = String_Split(“A,B”, “,”)                 // After splitting string, the string array has 2 elements
  var_len = Length(var_ss)                              // var_len = 2
  ```

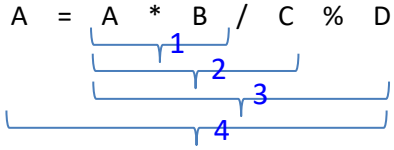
1.4 Operator Symbols

- The operator table is listed below.
- The calculation follows the precedence of operator first then the associativity.

  For example

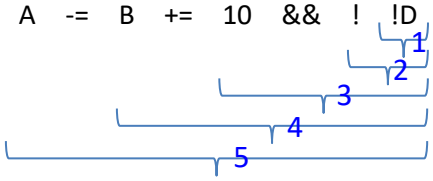
  left-to-right associativity

  A = A * B / C % D ➔ (A = (((A * B) / C) % D))

  right-to-left associativity

  A -= B += 10 && ! !D ➔ (A -= (B += (10 && (! (!D)))))

- The calculation will proceed by the type of the operand.

  1. When both values come as the integer type, the calculation will proceed by the integer type such as

     int var_a = 10

     int var_b = 3

     float var_c = var_a / var_b

     By the operator priority, the calculation goes / first and then =.

     var_a / var_b = 10 / 3 = 3 (both var_a and var_b are integers)

     var_c = 3 (The integer 3 assigns to the floating point number var_c)

  2. When one of the two values comes as the floating-point type, the calculation will proceed by the floating-point type such as

     int var_a = 10

     float var_b = 3

     float var_c = var_a / var_b

     var_a / var_b = 10 / 3 = 3.333333 (for var_b is a floating-point number)

     var_c = 3.333333 (the floating-point number 3.333333 assigns to var_c)

     var_c = var_a / 3

     var_a / 3 = 10 / 3 = 3 (both var_a and 3 are integers)

     var_c = 3

     var_c = var_a / 3.0

     var_a / 3.0 = 10 / 3.0 = 3.333333 (for 3.0 is a floating-point number)

     var_c = 3.333333

Precedence High to low	Operator	Name	Example	Requirement	associativity
17	++	Postfix increment	i++	Integer variable	left-to-right
	--	Postfix decrement	i--		
	()	Function call	int x = f()		
	[]	Allocate storage	array[4] = 2	Array variable	
16	++	Prefix increment	++i	Integer variable	right-to-left
	--	Prefix decrement	--i		
	+	Unary plus	int i = +1	Numeric variable, Constant	
	-	Unary minus	int i = -1		
	!	Logical negation (NOT)	if (!done) …	Boolean	
	~	Bitwise NOT	flag1 = ~flag2	Integer	
14	*	Multiplication	int i = 2 * 4	Numeric variable, Constant	left-to-right
	/	Division	float f = 10.0 / 3.0		
	%	Modulo (integer	int rem = 4 % 3		
13	+	Addition	int i = 2 + 3	Numeric variable, Constant	
	-	Subtraction	int i = 5 - 1		
12	<<	Bitwise left shift	int flags = 33 << 1	Integer variable, Constant	
	>>	Bitwise right shift	int flags = 33 >> 1		
11	<	Less than	if (i < 42) …	Numeric variable, Constant	
	<=	Less than or equal to	if (i <= 42) ...		
	>	Greater than	if (i > 42) …		
	>=	Greater than or equal to	if (i >= 42) ...		
10	==	Equal to	if (i == 42) ...		
	!=	Not equal to	if (I != 42) …		
9	&	Bitwise AND	flag1 = flag2 & 42	Integer variable, Constant	
8	^	Bitwise XOR	flag1 = flag2 ^ 42		
7	|	Bitwise OR	flag1 = flag2 | 42		
6	&&	Logical AND	if (conditionA && conditionB)		
5	||	Logical OR	if (conditionA || conditionB)		
4	c ? t : f	Ternary conditional	int i = a > b ? a : b		right-to-left
3	=	Basic assignment	int a = b		
	+=	Addition assignment	a += 3		
	-=	Subtraction assignment	b -= 4		

	*=	Multiplication assignment	a *= 5	Left side: Numeric variable Right side: Numeric	
	/=	Division assignment	a /= 2		
	%=	Modulo assignment	a %= 3		
	<<=	Bitwise left shift assignment	flags <<= 2	Left side: Integer variable Right side: Integer variable, Constant	
	>>=	Bitwise right shift assignment	flags >>= 2		
	&=	Bitwise AND assignment	flags &= new_flags		
	^=	Bitwise XOR assignment	flags ^= new_flags		
	|=	Bitwise OR assignment	flags |= new_flags		

1.5 Data type conversion

- Data types can be converted to each other and used in variables/constants or arrays.
- Conversions must be in the same format of the containers such as variable/constant conversions or array conversions. It is not permitted to convert a variable to an array or an array to a variable.

Native type	Conversion type	Example	Result
byte	int	int i = (int)100	i = 100
	float	float f = (float)100	f = 100
	double	double d = (double)100	d = 100
	bool	bool flag = (bool)0	flag = true (not null)
	string	string s = (string)100	s = “100”
int	byte	byte b = (byte)1000	b = 232
	float	float f = (float)1000	f = 1000
	double	double d = (double)1000	d = 1000
	bool	bool flag = (bool)1000	flag = true (not null)
	string	string s = (string)1000	s = “1000”
float	byte	byte b = (byte)1.23	b = 1
	int	int i = (int)1.23	i = 1
	double	double d = (double)1.23	d = 1.23
	bool	bool flag = (bool)1.23	flag = true
	string	string s = (string)1.23	s = “1.23”
double	byte	byte b = (byte)1.23	b = 1
	int	int i = (int)1.23	i = 1
	float	float f = (float)1.23	f = 1.23
	bool	bool flag = (bool)1.23	flag = true
	string	string s = (string)1.23	s = “1.23”
bool	byte	byte b = (byte)True	Error
	int	int i = (int)False	Error
	float	float f = (float)true	Error
	double	double d = (double)false	Error
	string	string s = (string)True	s = “True”
string	byte	byte b1 = (byte)”1.23” byte b2 = (byte)”XYZ”	1 Error
	int	int i = (int)”1.23”	1
	float	float f1 = (float)”1.23” float f2 = (float)”XYZ”	1.23 Error
	double	double d = (double)”1.23”	1.23
	bool	bool flag1 = (bool)”1.23” bool flag2 = (bool)””	flag1 = true (not null) flag2 = false (null)

- The conversion method of arrays is in accordance with the table above. The conversion is performed for each element in the array.

```
string[] var_ss = {“1.23”, “4.56”, “0.789”}

int[] var_i_array = (int[])var_ss          // var_i_array = {1, 4, 0}

float[] var_f_array = (float[])var_ss      // var_f_array = {1.23, 4.56, 0.789}
```

- Error messages will be returned should the conversions below occur.
  - Fail to convert to numeric correctly such as Booleans (true/false) or non-numeric strings (“XYZ”).

    int var_value = (int)true // Error

    int var_value = (int)”XYZ” // Error
  - Invalid floating-point numbers to convert to floats or doubles such as NaN or Infinity.

    string var_ dvalue = “1.79769e+308”

    float var_f = (float) var_dvalue // Error 1.79769e+308 is a valid double type and unable to convert to the float type.

1.6 Warning

A warning message will prompt, under the condition listed below.

- Double quotation marks does not placed around the string constant.

  string var_s = Hello // warningHello
- There is single quotation mark inside the string constant.

  string var_s0 = “World”

  string var_s1 = “Hello ‘s0’” // warning‘s0’
- When assigning float value to integer constant, some digits may get lost such as

  int var_i = 1.234// warning var_i = 1

  float var_f = 1.234

  var_i = var_f // warning var_i = 1
- When assigning value to variables with fewer digits, some digits may get lost such as

  byte var_b = 100

  int var_i = 1000

  float var_f = 1.234

  double var_d = 2.345

  var_b = var_i // warning var_b = 232// byte can contain values from 0 to 255

  var_f = var_d // warning var_f = 2.345
- When assigning string value to numeric variable, a conversion from string to number will be applied. If the conversion is executable, a warning message will prompt, or the project will be stopped by error such as

  int var_i = “1234” // warning var_i = 1234

  int var_j = “0x89AB” // warning var_j = 35243

```
int var_k = “0b1010”          // warning  var_k = 10
string var_s1 = 123           // warning var_s1 = 123 // Number to string
string var_s2 = “123”
int var_x = var_s2            // warning  // string to number
// The following code can be compiled with warning, but will be stopped by error when executing.
var_S2 = “XYZ”
var_x = var_s2                // warning // Stop executing by error
                              // var_s = “XYZ” cannot be converted to number
var_s2 = “”
var_x = var_s2                // warning // Stop executing by error
                              // var_s = “” cannot be converted to number
```

- String parameters are used as numeric parameters in functions such as

```
Ctrl(0x0A0B0C0D0E)    // warning  // 0x0A0B0C0D0E is not int type (over 32bit)
                                  // Because there is another syntax, Ctrl(string), the parameter would
                                  be applied to Ctrl(string)
```

Although the project can still be executed with a warning message, correcting all the errors in a warning message is highly recommended to eliminate unpredictable problems and prevent the project being stopped by errors.

- How to fix the error messages

1. Use double quotations with the string constants

```
string var_s = “Hello”
```

2. Use + to link the string constant and the string variable

```
string var_s0 = “World”

string var_s1 = “Hello ” + var_s0
```

3. Specify the type clearly for numerical conversions

```
float var_ f = 1.234
int var_i = (int) var_f    // Use (int) for type conversion, var_i = 1 while processing // It turns the number
                           in floating-point to an integer.
```

2. Functions

2.1 Byte_ToInt16()

Transform the first two bytes of the assigned byte array to integer, and returns in int type.

Syntax 1

```
int Byte_ToInt16(
    byte[],
    int,
    int
)
```

Parameters

`byte[]`	Byte array	
`int`	Byte array follows the Little Endian or Big Endian	
	`0`	Little Endian (Default)
	`1`	Big Endian
`int`	Transfer to signed int16 (Signed Number) or unsigned int16 (Unsigned Number)	
	`0`	signed int16 (Default)
	`1`	unsigned int16

Return

`int`	A signed or unsigned int16 formed by 2 bytes beginning at index [0]. Because only 2 bytes is needed, the index of byte array will be [0][1]. If the data is not long enough, it would be filled to 2 bytes before transforming.

Note

byte[] var_bb1 = {0x90, 0x01, 0x05}

byte[] var_bb2 = {0x01} // Cause var_bb2[] does not fill 2 bytes. It would be filled to 2 bytes before transforming.

var_value = **Byte_ToInt16**(var_bb1, 0, 0) // 0x0190 var_value = 400
var_value = **Byte_ToInt16**(var_bb1, 0, 1) // 0x0190 var_value = 400
var_value = **Byte_ToInt16**(var_bb1, 1, 0) // 0x9001 var_value = -28671
var_value = **Byte_ToInt16**(var_bb1, 1, 1) // 0x9001 var_value = 36865
var_value = **Byte_ToInt16**(var_bb2, 0, 0) // 0x0001 var_value = 1
var_value = **Byte_ToInt16**(var_bb2, 0, 1) // 0x0001 var_value = 1
var_value = **Byte_ToInt16**(var_bb2, 1, 0) // 0x0100 var_value = 256
var_value = **Byte_ToInt16**(var_bb2, 1, 1) // 0x0100 var_value = 256

Syntax 2

```
int Byte_ToInt16(
    byte[],
    int
)
```

Note

Similar to Syntax 1 with return value as signed int16

Byte_ToInt16(var_bb1, 0) => **Byte_ToInt16**(var_bb1, 0, 0)

Syntax 3

```
int Byte_ToInt16(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input and return value as signed int16

Byte_ToInt16(var_bb1) => **Byte_ToInt16**(var_bb1, 0)

2.2 Byte_ToInt32()

Transform the first four bytes of byte array to integer, and return in int type.

Syntax 1

```
int Byte_ToInt32(
    byte[],
    int
)
```

Parameters

`byte[]`	The input byte array
`int`	The input byte array follows Little Endian or Big Endian
`0`	Little Endian (Default)
`1`	Big Endian

Return

`int`	An unsigned int32 formed by 4 bytes beginning at index [0]. Because only 4 bytes is needed, the index of byte array will be [0][1][2][3]. If the data is not long enough, it would be filled to 4 bytes before transforming.

Note

byte[] var_bb1 = {0x01, 0x02, 0x03, 0x4F, 1}

byte[] var_bb2 = {0x01, 0x02, 0x03}

// Cause var_bb2[] does not fill 4 bytes. It would be filled to 4 bytes before transforming.

var_value = **Byte_ToInt32**(var_bb1, 0) // 0x4F030201 var_value = 1325597185

var_value = **Byte_ToInt32**(var_bb1, 1) // 0x0102034F var_value = 16909135

var_value = **Byte_ToInt32**(var_bb2, 0) // 0x00030201 var_value = 197121

var_value = **Byte_ToInt32**(var_bb2, 1) // 0x01020300 var_value = 16909056

Syntax 2

```
int Byte_ToInt32(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input

Byte_ToInt32(var_bb1) => **Byte_ToInt32**(var_bb1, 0)

2.3 Byte_ToFloat()

Transform the first four bytes of byte array to floating-point number, and return in floating-point type.

Syntax 1

```
float Byte_ToFloat(
    byte[],
    int
)
```

Parameters

byte[]	The input byte array
int	The input byte array follows Little Endian or Big Endian
0	Little Endian (Default)
1	Big Endian

Return

float	A floating-point number formed by 4 bytes beginning at index [0]. Because only 4 bytes is needed, the index of byte array will be [0][1][2][3]. If the data is not long enough, it would be filled to 4 bytes before transforming.

Note

byte[] var_bb1 = {0x01, 0x02, 0x03, 0x4F, 1}

byte[] var_bb2 = {0x01, 0x02, 0x03} // Cause bb2[] does not fill 4 bytes. It would be filled to 4 bytes before transforming.

var_value = **Byte_ToFloat**(var_bb1, 0) // 0x4F030201 var_value = 2.197947E+09

var_value = **Byte_ToFloat**(var_bb1, 1) // 0x0102034F var_value = 2.38796E-38

var_value = **Byte_ToFloat**(var_bb2, 0) // 0x00030201 var_value = 2.762254E-40

var_value = **Byte_ToFloat**(var_bb2, 1) // 0x01020300 var_value = 2.387938E-38

Syntax 2

```
float Byte_ToFloat(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input

Byte_ToFloat(var_bb1) => **Byte_ToFloat**(var_bb1, 0)

2.4 Byte_ToDouble()

Transform the first eight bytes of byte array to floating-point number, and return in double type.

Syntax 1

```
double Byte_ToDouble(
    byte[],
    int
)
```

Parameters

`byte[]`	The input byte array
`int`	The input byte array follows Little Endian or Big Endian
	`0` Little Endian (Default)
	`1` Big Endian

Return

`double`	A floating-point number formed by 8 bytes beginning at index [0]. Because only 8 bytes is needed, the index of byte array will be [0][1][2][3][4][5][6][7]. If the data is not long enough, it would be filled to 8 bytes before transforming.

Note

byte[] var_bb1 = {0x01, 0x02, 0x03, 0x4F, 1} // Cause bb1[] does not fill 8 bytes. It would be filled to 8 bytes before transforming.

byte[] var_bb2 = {0x01, 0x02, 0x03} // Cause bb1[] does not fill 8 bytes. It would be filled to 8 bytes before transforming.

var_value = **Byte_ToDouble**(var_bb1, 0) // 0x000000014F030201 var_value = 2.77692782029764E-314

var_value = **Byte_ToDouble**(var_bb1, 1) // 0x0102034F01000000 var_value = 8.20840179153173E-304

var_value = **Byte_ToDouble**(var_bb2, 0) // 0x0000000000030201 var_value = 9.73907141738724E-319

var_value = **Byte_ToDouble**(var_bb2, 1) // 0x0102030000000000 var_value = 8.20785244926136E-304

Syntax 2

```
double Byte_ToDouble(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input

Byte_ToDouble(var_bb1) => **Byte_ToDouble**(var_bb1, 0)

2.5 Byte_ToInt16Array()

Transform byte array to integer every 2 bytes, and return in int[] type.

Syntax 1

```
int[] Byte_ToInt16Array(
    byte[],
    int,
    int
)
```

Parameters

byte[]	The input byte array	
int	The input byte array follows Little Endian or Big Endian	
	0	Little Endian (Default)
	1	Big Endian
int	Transfer to signed int16 (Signed Number) or unsigned int16 (Unsigned Number)	
	0	signed int16 (Default)
	1	unsigned int16

Return

int[]	A integer array formed by every 2 bytes of byte array beginning at index [0]

Note

byte[] var_bb1 = {0x90, 0x01, 0x02, 0x03, 0x04} // When the remaining part does not fill 2 byte, it would be filled

to 2 bytes before transforming.

byte[] var_bb2 = {1, 2, 3, 4}

var_value = **Byte_ToInt16Array**(var_bb1, 0, 0) // {0x0190, 0x0302, 0x0004} var_value = {400, 770, 4}
var_value = **Byte_ToInt16Array**(var_bb1, 0, 1) // {0x0190, 0x0302, 0x0004} var_value = {400, 770, 4}
var_value = **Byte_ToInt16Array**(var_bb1, 1, 0) // {0x9001, 0x0203, 0x0400} var_value = {-28671, 515, 1024}
var_value = **Byte_ToInt16Array**(var_bb1, 1, 1) // {0x9001, 0x0203, 0x0400} var_value = {36865, 515, 1024}

var_value = **Byte_ToInt16Array**(var_bb2, 0, 0) // {0x0201, 0x0403} var_value = {513, 1027}
var_value = **Byte_ToInt16Array**(var_bb2, 0, 1) // {0x0201, 0x0403} var_value = {513, 1027}
var_value = **Byte_ToInt16Array**(var_bb2, 1, 0) // {0x0102, 0x0304} var_value = {258, 772}
var_value = **Byte_ToInt16Array**(var_bb2, 1, 1) // {0x0102, 0x0304} var_value = {258, 772}

Syntax 2

```
int[] Byte_ToInt16Array(
    byte[],
    int
)
```

Note

Similar to Syntax 1 with return value as signed int16

Byte_ToInt16Array(var_bb1, 0) => **Byte_ToInt16Array**(var_bb1, 0, 0)

Syntax 3

```
int[] Byte_ToInt16Array(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input and return value as signed int16

Byte_ToInt16Array(var_bb1) => **Byte_ToInt16Array**(var_bb1, 0)

2.6 Byte_ToInt32Array()

Transform byte array to integer every 4 bytes, and return in int[] type

Syntax 1

```
int[] Byte_ToInt32Array(
```

```
    byte[],
    int
)
```

Parameters

`byte[]`	The input byte array
`int`	The input byte array follows Little Endian or Big Endian
`0`	Little Endian (Default)
`1`	Big Endian

Return

`int[]`	A integer array formed by every 4 bytes of byte array beginning at index [0]

Note

```
byte[] var_bb1 = {0x01, 0x02, 0x03, 0x04, 0x05}  // When the remaining part does not fill 4 byte, it would be filled
                                                    to 4 bytes before transforming.
byte[] var_bb2 = {1, 2, 3, 4}

var_value = Byte_ToInt32Array(var_bb1, 0) // {0x04030201, 0x00000005} var_value = {67305985, 5}
var_value = Byte_ToInt32Array(var_bb1, 1) // {0x01020304, 0x05000000} var_value = {16909060, 83886080}
var_value = Byte_ToInt32Array(var_bb2, 0) // {0x04030201} var_value = {67305985}
var_value = Byte_ToInt32Array(var_bb2, 1) // {0x01020304} var_value = {16909060}
```

Syntax 2

```
int[] Byte_ToInt32Array(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input.

Byte_ToInt32Array(var_bb1) => **Byte_ToInt32Array**(var_bb1, 0)

2.7 Byte_ToFloatArray()

Transform byte array to integer every 4 bytes, and return in float[] type.

Syntax 1

```
float[] Byte_ToFloatArray(
    byte[],
    int
```

)

Parameters

`byte[]` The input byte array

`int` The input byte array follows Little Endian or Big Endian

`0` Little Endian (Default)

`1` Big Endian

Return

`float[]` A floating-point number array formed by every 4 bytes of byte array beginning at index [0]

Note

byte[] var_bb1 = {0x01, 0x02, 0x03, 0x04, 0x05}

// When the remaining part does not fill 4 byte, it would be filled to 4 bytes before transforming.

byte[] var_bb2 = {1, 2, 3, 4}

var_value = **Byte_ToFloatArray**(var_bb1, 0)

// {0x04030201, 0x00000005} var_value = {1.53999E-36, 7.006492E-45}

var_value = **Byte_ToFloatArray**(var_bb1, 1)

// {0x01020304, 0x05000000} var_value = {2.387939E-38, 6.018531E-36}

var_value = **Byte_ToFloatArray**(var_bb2, 0) // {0x04030201} var_value = {1.53999E-36}

var_value = **Byte_ToFloatArray**(var_bb2, 1) // {0x01020304} var_value = {2.387939E-38}

Syntax 2

```
float[] Byte_ToFloatArray(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input

Byte_ToFloatArray(var_bb1) => **Byte_ToFloatArray**(var_bb1, 0)

2.8 Byte_ToDoubleArray()

Transform byte array to double every 8 bytes, and return in double[] type.

Syntax 1

```
double[] Byte_ToDoubleArray(
    byte[],
    int
```

)

Parameters

`byte[]` The input byte array

`int` The input byte array follows Little Endian or Big Endian

`0` Little Endian (Default)

`1` Big Endian

Return

`double[]` A floating-point number array formed by every 8 bytes of byte array beginning at index [0]

Note

byte[] var_bb1 = {0x01, 0x02, 0x03, 0x04, 0x05} // When the remaining part does not fill 8 byte, it would be filled to 8 bytes before transforming.

byte[] var_bb2 = {1, 2, 3, 4} // When the remaining part does not fill 8 byte, it would be filled to 8 bytes before transforming.

var_value = **Byte_ToDoubleArray**(var_bb1, 0) // {0x0000000504030201} var_value = {1.06432325297744E-313}

var_value = **Byte_ToDoubleArray**(var_bb1, 1) // {0x0102030405000000} var_value = {8.20788039849233E-304}

var_value = **Byte_ToDoubleArray**(var_bb2, 0) // {0x0000000004030201} var_value = {3.32535749480063E-316}

var_value = **Byte_ToDoubleArray**(var_bb2, 1) // {0x0102030400000000} var_value = {8.2078802626846E-304}

Syntax 2

```
double[] Byte_ToDoubleArray(
    byte[]
)
```

Note

Similar to Syntax 1 with little endian input

Byte_ToDoubleArray(var_bb1) => **Byte_ToDoubleArray**(var_bb1, 0)

2.9 Byte_ToString()

Transform byte array to string

Syntax 1

```
string Byte_ToString(
    byte[],
    int
)
```

Parameters

`byte[]` The input byte array

`int` The character encoding rules applied to input byte array

`0` UTF8 (Default) (0x00 END)

`1` HEX BINARY

`2` ASCII (0x00 END)

Return

`string` String formed by byte array. The transformation begins from index [0].

Note

byte[] var_bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}

byte[] var_bb2 = {0x01, 0x54, 0x4D, 0x35, 0xE6, 0xA9, 0x9F, 0xE5, 0x99, 0xA8, 0xE4, 0xBA, 0xBA}

var_value = **Byte_ToString**(var_bb1, 0) // var_value = "123" (UTF8 stop at 0x00)

var_value = **Byte_ToString**(var_bb1, 1) // var_value = "313233004F01"

var_value = **Byte_ToString**(var_bb1, 2) // var_value = "123" (ASCII stop at 0x00)

var_value = **Byte_ToString**(var_bb2, 0) // var_value = "\u01TM5機器人" (UTF8)

var_value = **Byte_ToString**(var_bb2, 1) // var_value = "01544D35E6A99FE599A8E4BABA"

var_value = **Byte_ToString**(var_bb2, 2) // var_value = "\u01TM5?????????" (ASCII)

* \u01 represents the SOH control character, not the string value.

Syntax 2

```
string Byte_ToString(
    byte[]
)
```

Note

Similar to Syntax 1 with UTF8 character encoding rules

Byte_ToString(var_bb1) => **Byte_ToString**(var_bb1, 0)

2.10 Byte_Concat()

Concatenate two byte arrays, or concatenate one array with a byte value.

Syntax 1

```
byte[] Byte_Concat(
    byte[],
    byte
)
```

Parameters

`byte[]` The input byte array

`byte` The byte value concatenated after the byte array

Return

`byte[]` The byte array formed by the input byte array and byte value

Note

byte[] var_bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}

var_value = **Byte_Concat**(var_bb1, 12) // var_value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x0C}

Syntax 2

```
byte[] Byte_Concat(
    byte[],
    byte[]
)
```

Parameters

`byte[]` The input byte array1

`byte[]` The input byte array2, would be concatenated to the end of array1

Return

`byte[]` Byte array formed from concatenating input arrays.

Note

byte[] var_bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}

byte[] var_bb2 = {0x01, 0x02, 0x03}

var_value = **Byte_Concat**(var_bb1, var_bb2)
// var_value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02, 0x03}

Syntax 3

```
byte[] Byte_Concat(
    byte[],
    byte[],
    int
)
```

Parameters

`byte[]` The input byte array1

`byte[]` The input byte array2, would be concatenated after the end of array1

`int`	The number of element in array2 to be concatenated	
	0..the length of array2	Valid number
	<0	Invalid. Length of array2 will be applied instead.
	> the length of array2	Invalid. Length of array2 will be applied instead.

Return

`byte[]` Byte array formed from concatenating input arrays.

Note

```
byte[] var_bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}
byte[] var_bb2 = {0x01, 0x02, 0x03}

var_value = Byte_Concat(var_bb1, var_bb2, 2)  // var_value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02} //
                                              Concatenate only 2 elements from array2
var_value = Byte_Concat(var_bb1, var_ bb2, -1) // var_value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02, 0x03}
                                              // -1 is invalid value
var_value = Byte_Concat(var_bb1, var_bb2, 10) // var_value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02, 0x03}
                                              // 10 exceeds the array size

                                              // Length() can be utilized to acquire the array size
var_value = Byte_Concat(var_bb1, var_bb2, Length(var_bb2))
                                              // var_value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02, 0x03}
```

Syntax 4

```
byte[] Byte_Concat(
    byte[],
    int,
    int,
    byte[],
    int,
    int
)
```

Parameters

`byte[]`	The input byte array1	
`int`	The starting index of array1	
	0..(length of array1)-1	Valid
	<0	The starting index would be 0
	>=(length of array1)	The starting index would be the length of array2 (For index over the length of array2, an empty value would be captured)
`int`	The number of element in array1 to be concatenated	
	0.. (length of array1)	Valid

	<0	Invalid，length of array1 will be applied instead
	>(length of array1)	Invalid，length of array1 will be applied instead

If the total number of starting index and assigning elements exceeds the length of array1, the surplus index will be suspended.

`byte[]` The input byte array2，would be concatenated after the end of array1

`int` The starting index of array2

0.. (length of array2)-1	Valid
<0	The starting index would be 0
>=(length of array2)	The starting index would be the length of array2 (For index over the length of array2, an empty value would be captured)

`int` The number of element in array2 to be concatenated

0.. (length of array2)	Valid
<0	Invalid. Length of array2 will be applied instead.
>(length of array2)	Invalid. Length of array2 will be applied instead.

If the total number of starting index and assigning elements exceeds the length of array2, the surplus index will be suspended.

Return

`byte[]` Byte array formed from concatenating input arrays.

Note

byte[] var_bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}

byte[] var_bb2 = {0x01, 0x02, 0x03}

var_value = **Byte_Concat**(var_bb1, 1, 3, var_bb2, 1, 2) // var_value = {0x32, 0x33, 0x00, 0x02, 0x03}

var_value = **Byte_Concat**(var_bb1, -1, 3, var_bb2, 3, -1) // var_value = {0x31, 0x32, 0x33}

2.11 String_ToInteger()

Transform string to integer (int type)

Syntax 1

```
int String_ToInteger(
    string,
    int
)
```

Parameters

`string` The input string.

`int` The input string's notation is decimal, hexadecimal or binary

10	decimal or auto format detecting (Default)
16	hexadecimal
2	binary

String's notation

123	decimal
0x7F	hexadecimal
0b101	binary

Return

int	The integer value formed from input string. If notation is invalid, returns 0.

Note

```
var_value = String_ToInteger("1234", 10)        // var_value = 1234
var_value = String_ToInteger("1234", 16)        // var_value = 4660
var_value = String_ToInteger("1234", 2)         // var_value = 0      // Invalid binary format
var_value = String_ToInteger("1100", 2)         // var_value = 12
var_value = String_ToInteger("0x1234", 10)      // var_value = 4660   // Hexadecimal format by auto
                                                detecting
var_value = String_ToInteger("0x1234", 16)      // var_value = 4660
var_value = String_ToInteger("0x1234", 2)       // var_value = 0      // Invalid binary format
var_value = String_ToInteger("0b1100", 10)      // var_value = 12     // Binary format by auto detecting
var_value = String_ToInteger("0b1100", 16)      // var_value = 725248 // Valid Hexadecimal number
var_value = String_ToInteger("0b1100", 2)       // var_value = 12
var_value = String_ToInteger("+1234", 10)       // var_value = 1234
var_value = String_ToInteger("-1234", 10)       // var_value = -1234
var_value = String_ToInteger("-0x1234", 16)     // var_value = 0      // Invalid hex format
var_value = String_ToInteger("-0b1100", 2)      // var_value = 0      // Invalid binary format
```

Syntax 2

```
int String_ToInteger(
    string
)
```

Note

Similar to syntax1 with decimal format or auto format detection

String_ToInteger(str) => **String_ToInteger**(str, 10)

Syntax 3

```
int[] String_ToInteger(
    string[],
    int
)
```

Parameters

`string[]`	Input string array	
`int`	The notation of element in input string array is decimal, hexadecimal or binary	
	`10`	decimal or auto format detecting (Default)
	`16`	hexadecimal
	`2`	binary
	String's notation	
	123	decimal
	0x7F	hexadecimal
	0b101	binary
	* The notations of all the elements in a single array have to be identical	

Return

`int[]`	The integer array formed from input string array. If notation is invalid, returns 0.

Note

var_ss = {"12", "ab", "cc", "dd", "10"}

var_value = **String_ToInteger**(var_ss) // var_value = {12, 0, 0, 0, 10} // "ab","cc","dd" are invalid decimal numbers

var_value = **String_ToInteger**(var_ss, 2) // var_value = {0, 0, 0, 0, 2} // "12","ab","cc","dd" are invalid binary numbers

var_value = **String_ToInteger**(var_ss, 16) // var_value = {18, 171, 204, 221, 16}

var_value = **String_ToInteger**(var_ss, 10) // var_value = {12, 0, 0, 0, 10} // "ab","cc","dd" are invalid decimal numbers

2.12 String_ToFloat()

Transform string to floating-point (floating-point type)

Syntax 1

```
float String_ToFloat(
    string,
    int
)
```

Parameters

`string`	Input string	
`int`	Input string's notation is decimal, hexadecimal or binary format	
	`10`	decimal or auto format detecting (Default)
	`16`	hexadecimal
	`2`	binary

String's notation

123	decimal
0x7F	hexadecimal
0b101	binary

Return

`float` The floating-point number formed from input string. If notation is invalid, returns 0.

Note

```
var_value = String_ToFloat("12.34", 10)          // var_value = 12.34
var_value = String_ToFloat("12.34", 16)          // var_value = 0          // Invalid hexadecimal format
var_value = String_ToFloat("12.34", 2)           // var_value = 0          // Invalid binary format
var_value = String_ToFloat("11.00", 2)           // var_value = 0          // Invalid binary format
var_value = String_ToFloat("0x1234", 10)         // var_value = 6.530051E-42  // Hexadecimal format by auto detecting
var_value = String_ToFloat("0x1234", 16)         // var_value = 6.530051E-42
var_value = String_ToFloat("0x1234", 2)          // var_value = 0          // Invalid binary format
var_value = String_ToFloat("0b1100", 10)         // var_value = 1.681558E-44  // Binary format by auto detecting
var_value = String_ToFloat("0b1100", 16)         // var_value = 1.016289E-39  // Valid hexadecimal format
var_value = String_ToFloat("0b1100", 2)          // var_value = 1.681558E-44
var_value = String_ToFloat("+12.34", 10)         // var_value = 12.34
var_value = String_ToFloat("-12.34", 10)         // var_value = -12.34
var_value = String_ToFloat("-0x1234", 16)        // var_value = 0          // Invalid hex format
var_value = String_ToFloat("-0b1100", 2)         // var_value = 0          // Invalid format
```

Syntax 2

```
float String_ToFloat(
    string
)
```

Note

Similar to syntax1 with decimal format or auto format detection

String_ToFloat(str) => **String_ToFloat**(str, 10)

Syntax 3

```
float[] String_ToFloat(
    string[],
    int
)
```

Parameters

`string[]` Input string array

`int` The notation of elements in input string array is decimal, hexadecimal or binary

`10` decimal or auto format detecting (Default)

`16` hexadecimal

`2` binary

String's notation

123 decimal

0x7F hexadecimal

0b101 binary

* The notation of all the elements in a single array have to be identical

Return

`float[]` The floating-point number array formed from input string array. If notation is invalid, returns 0.

Note

var_ss = {"12.345", "ab", "cc", "dd", "10.111"}

var_value = **String_ToFloat**(var_ss) // var_value = {12.345, 0, 0, 0, 10.111}

var_value = **String_ToFloat**(var_ss, 2) // var_value = {0, 0, 0, 0, 0}

var_value = **String_ToFloat**(var_ss, 16) // var_value = {0, 2.39622E-43, 2.858649E-43, 3.09687E-43, 0}

var_value = **String_ToFloat**(var_ss, 10) // var_value = {12.345, 0, 0, 0, 10.111}

2.13 String_ToDouble()

Transform string to floating-point number (double type)

Syntax 1

```
double String_ToDouble(
    string,
    int
)
```

Parameters

`string` Input string

`int` Input string's notation is decimal, hexadecimal or binary format

`10` decimal or auto format detecting (Default)

`16` hexadecimal

`2` binary

String's notation

123 decimal

0x7F hexadecimal

0b101 binary

Return

`double` The floating-point number formed from input string. If notation is invalid, returns 0.

Note

```
var_value = String_ToDouble("12.34", 10)      // var_value = 12.34
var_value = String_ToDouble("12.34", 16)      // var_value = 0          // Invalid hexadecimal format
var_value = String_ToDouble("12.34", 2)       // var_value = 0          // Invalid binary format
var_value = String_ToDouble("11.00", 2)       // var_value = 0          // Invalid binary format
var_value = String_ToDouble("0x1234", 10)     // var_value = 2.30234590962021E-320   // Hexadecimal format by auto detecting
var_value = String_ToDouble("0x1234", 16)     // var_value = 2.30234590962021E-320
var_value = String_ToDouble("0x1234", 2)      // var_value = 0          // Invalid binary format
var_value = String_ToDouble("0b1100", 10)     // var_value = 5.92878775009496E-323   // Binary format by auto detecting
var_value = String_ToDouble("0b1100", 16)     // var_value = 3.58320121515072E-318   // Valid hexadecimal format
var_value = String_ToDouble("0b1100", 2)      // var_value = 5.92878775009496E-323
var_value = String_ToDouble("+12.34", 10)     // var_value = 12.34
var_value = String_ToDouble("-12.34", 10)     // var_value = -12.34
var_value = String_ToDouble("-0x1234", 16)    // var_value = 0          // Invalid hex format
var_value = String_ToDouble("-0b1100", 2)     // var_value = 0          // Invalid binary format
```

Syntax 2

```
double String_ToDouble(
    string
)
```

Note

Similar to syntax1 with decimal format or auto format detection

String_ToDouble(str) => **String_ToDouble**(str, 10)

Syntax 3

```
double[] String_ToDouble(
    string[],
    int
)
```

Parameters

`string[]` Input string array

`int` The notation of elements in input string array is decimal, hexadecimal or binary

10	decimal or auto format detecting (Default)
16	hexadecimal
2	binary

String's notation

123	decimal
0x7F	hexadecimal
0b101	binary

* The notation of all the elements in a single array has to be identical

Return

double[]	The floating-point number array formed from input string array. If notation is invalid, returns 0.

Note

var_ss = {"12.345", "ab", "cc", "dd", "10.111"}

var_value = **String_ToDouble**(var_ss) // var_value = {12.345, 0, 0, 0, 10.111}

var_value = **String_ToDouble**(var_ss, 2) // var_value = {0, 0, 0, 0, 0}

var_value = **String_ToDouble**(var_ss, 16) // var_value = {0, 8.44852254388532E-322, 1.00789391751614E-321, 1.09188507730915E-321, 0}

var_value = **String_ToDouble**(var_ss, 10) // var_value = {12.345, 0, 0, 0, 10.111}

2.14 String_ToByte()

Transform string to byte array

Syntax 1

```
byte[] String_ToByte(
    string,
    int
)
```

Parameters

string	Input string
int	The character encoding rules applied to input string
0	UTF8 (Default)
1	HEX BINARY // Stop at invalid Hex value
2	ASCII

Return

byte[]	The byte array formed from input string

Note

var_value = **String_ToByte**("12345", 0) // var_value = {0x31, 0x32, 0x33, 0x34, 0x35}

var_value = **String_ToByte**(“12345”, 1) // var_value = {0x12, 0x34, 0x50} // the insufficient part will be filled with 0

var_value = **String_ToByte**(“12345”, 2) // var_value = {0x31, 0x32, 0x33, 0x34, 0x35}

var_value = **String_ToByte**(“0x12345”, 0) // var_value = {0x30, 0x78, 0x31, 0x32, 0x33, 0x34, 0x35}

var_value = **String_ToByte**(“0x12345”, 1) // var_value = {0x00} // Only 0 be transformed, cause x is an invalid Hex value

var_value = **String_ToByte**(“0x12345”, 2) // var_value = {0x30, 0x78, 0x31, 0x32, 0x33, 0x34, 0x35}

var_value = **String_ToByte**(“TM5機器人”, 0) // var_value = {0x54, 0x4D, 0x35, 0xE6, 0xA9, 0x9F, 0xE5, 0x99, 0xA8, 0xE4, 0xBA, 0xBA}

var_value = **String_ToByte**(“TM5機器人”, 1) // var_value = {0x00} // T is an invalid Hex value

var_value = **String_ToByte**(“TM5機器人”, 2) // var_value = {0x54, 0x4D, 0x35, 0x3F, 0x3F, 0x3F}

var_value = **String_ToByte**(“0123456”, 1) // var_value = {0x01, 0x23, 0x45, 0x60}

var_value = **String_ToByte**(“01234G5”, 1) // var_value = {0x01, 0x23, 0x40} // G is an invalid Hex value

Syntax 2

```
byte[] String_ToByte(
    string
)
```

Note

Similar to syntax1 with UTF8 format

String_ToByte(str) => **String_ToByte**(str, 0)

2.15 String_IndexOf()

Report the zero-based index of the first occurrence of a specified string

Syntax 1

```
int String_IndexOf(
    string,
    string
)
```

Parameters

`string` Input string

`string` The specified string to be searched. The zero-based index of the first occurrence is to be found.

Return

`int`	`0..(Length of string)-1`	If the specified string is found, returns the index number
	`-1`	Not found
	`0`	The specified string is “” or empty

Note

var_value = **String_IndexOf**(“012314”, “1”) // var_value = 1

var_value = **String_IndexOf**(“012314”, “”) // var_value = 0

var_value = **String_IndexOf**(“012314”, empty) // var_value = 0

var_value = **String_IndexOf**(“012314”, “d”) // var_value = -1

var_value = **String_IndexOf**(“”, “d”) // var_value = -1

2.16 String_LastIndexOf()

Report the zero-based index position of the last occurrence of a specified string

Syntax 1

```
int String_LastIndexOf(
    string,
    string
)
```

Parameters

`string`	Input string
`string`	The specified string to be searched. The zero-based index of the last occurrence is to be found.

Return

`int`	`0..(Length of string)-1`	If the specified string is found, returns the index number
	`-1`	Not found
	`0`	The specified string is “” or empty

Note

var_value = **String_LastIndexOf**(“012314”, “1”) // var_value = 4

var_value = **String_LastIndexOf**(“012314”, “”) // var_value = 5

var_value = **String_LastIndexOf**(“012314”, empty) // var_value = 5

var_value = **String_LastIndexOf**(“012314”, “d”) // var_value = -1

var_value = **String_LastIndexOf**(“”, “d”) // var_value = -1

2.17 String_Substring()

Retrieve a substring from input string

Syntax 1

```
string String_Substring(
    string,
    int,
```

```
    int
)
```

Parameters

`string`	Input string
`int`	The starting character position of sub string (0 .. (length of input string)-1)
`int`	The length of substring

Return

`string`	Substring If starting character position <0, returns empty string If starting character position >= length of input string, returns empty string If length of substring <0, the substring ends at the last character of the input string If the sum of starting character position and length of substring exceeds the length of input string, the substring ends at the last character of the input string

Note

```
var_value = String_Substring("0x12345", 2, 4)          // var_value = "1234"
var_value = String_Substring("0x12345", -1, 4)         // var_value = ""
var_value = String_Substring("0x12345", 7, 4)          // var_value = ""
var_value = String_Substring("0x12345", 2, -1)         // var_value = "12345"
var_value = String_Substring("0x12345", 2, 100)        // var_value = "12345"
```

Syntax 2

```
string String_Substring(
    string,
    int
)
```

Note

Similar to syntax1 with the substring ends at the last character of the input string

String_Substring(str, 2) => **String_Substring**(str, 2, maxlen)

Syntax 3

```
string String_Substring(
    string,
    string,
    int
)
```

Parameters

`string` Input string

`string` The target string to be searched, the substring will start at its position, if it is found

`int` The length of substring

Return

`string` Substring

If the target string is empty, the substring start at index zero

If the target string is not found, returns empty string

If length of substring <0, the substring ends at the last character of the input string

If the sum of starting character position and length of substring exceeds the length of input string, the substring ends at the last character of the input string

Note

This syntax is the same as **String_Substring**(str, **String_IndexOf**(str1), int)

var_value = **String_Substring**("0x12345", "1", 4) // var_value = "1234"

var_value = **String_Substring**("0x12345", "", 4) // var_value = "0x12"

var_value = **String_Substring**("0x12345", "7", 4) // var_value = ""

var_value = **String_Substring**("0x12345", "1", -1) // var_value = "12345"

var_value = **String_Substring**("0x12345", "1", 100) // var_value = "12345"

Syntax 4

```
string String_Substring(
    string,
    string
)
```

Note

Similar to Syntax 3 with the substring ends at the last character of the input string

String_Substring(str, "1") => **String_Substring**(str, "1", maxlen)

Syntax 5

```
string String_Substring(
    string,
    string,
    string,
    int
)
```

Parameters

`string` Input string

`string` Prefix. The leading element of the substring

`string` Suffix. The trailing element of the substring

`int` The number of occurrence

Return

`string` Substring

If prefix and suffix are empty string, returns input string

If the number of occurrence<=0, returns empty string

Note

```
var_value = String_Substring("0x12345", "", "", 0)          // var_value = "0x12345"
var_value = String_Substring("0x12345", "1", "4", 1)        // var_value = "1234"
var_value = String_Substring("0x12345", "1", "4", 2)        // var_value = ""
var_value = String_Substring("0x12345", "1", "4", 0)        // var_value = ""
var_value = String_Substring("0x123450x12-345", "1", "4", 1)  // var_value = "1234"
var_value = String_Substring("0x123450x12-345", "1", "4", 2)  // var_value = "12-34"
var_value = String_Substring("0x123450x12-345", "1", "4", 3)  // var_value = ""
var_value = String_Substring("0x12345122", "1", "", 1)      // var_value = "12345122"   // All the character after prefix
var_value = String_Substring("0x12345122", "1", "", 2)      // var_value = "122"
var_value = String_Substring("0x12345122", "1", "", 4)      // var_value = ""
var_value = String_Substring("0x12345433", "", "4", 1)      // var_value = "0x123454"   // All the character before suffix
var_value = String_Substring("0x12345433", "", "4", 2)      // var_value = "0x1234"
```

Syntax 6

```
string String_Substring(
    string,
    string,
    string
)
```

Note

Similar to Syntax 5 with the substring start at the first occurrence

String_Substring(str, prefix, suffix) => **String_Substring**(str, prefix, suffix, 1)

2.18 String_Split()

Split the string using specified separator.

Syntax 1

```
string[] String_Split(
    string,
```

```
    string,
    int
)
```

Parameters

`string`	Input string
`string`	Separator (String)
`int`	Format
`0`	Split and keep the empty strings
`1`	Split and eliminate the empty strings
`2`	Split with the elements inside double quotation mark skipped, and keep the empty strings
`3`	Split with the elements inside double quotation mark skipped, and eliminate the empty strings

Return

`string[]`	Split substring If input string is empty, return substring have only one element. [0] = empty If separator is empty, return substring have only one element. [0] = Input string

Note

```
var_value = String_Split("0x112345", "1", 0)      // var_value = {"0x", "", "2345"}
var_value = String_Split("0x112345", "", 0)       // var_value = {"0x112345"}
var_value = String_Split("0x112345", "1", 1)      // var_value = {"0x", "2345"}
var_s1 = "123, ""456,67"",89"
var_value = String_Split(var_s1, ",", 0)          // var_value = {"123", """456", "67""", "89"}   // length = 4
var_value = String_Split(var_s1, ",", 2)          // var_value = {"123", """456,67""", "89"}      // length = 3
```

Syntax 2

```
string[] String_Split(
    string,
    string
)
```

Note

Similar to Syntax1 with splitting and keeping the empty strings

String_Split(str, separator) => **String_Split**(str, separator, 0)

2.19 String_Replace()

Return a new string in which all occurrences of a specified string in the input string are replaced with another specified string

Syntax 1

```
string String_Replace(
    string,
    string,
    string
)
```

Parameters

`string` Input string

`string` Old value, the string to be replaced

`string` New value, the string to replace all occurrences of old value

Return

`string` The string formed by replacing the old value with new value in input value. If the old value is empty, returns the input string

Note

var_value = **String_Replace**(“0x112345”, “1”, “2”) // var_value = “0x222345”

var_value = **String_Replace**(“0x112345”, “”, “2”) // var_value = “0x112345”

var_value = **String_Replace**(“0x112345”, “1”, “”) // var_value = ”0x2345”

2.20 String_Trim()

Return a new string in which all leading and trailing occurrences of specified characters or white-space characters from the input string are removed

Syntax 1

```
string String_Trim(
    string
)
```

Parameters

`string` Input string

Return

`string` String formed by removing all leading and trailing occurrences of white-space characters

Note

var_value = **String_Trim**(“0x112345 ”) // var_value = “0x112345”

var_value = **String_Trim**(“ 0x112345”) // var_value = “0x112345”

var_value = **String_Trim**(“ 0x112345 ”) // var_value = ”0x112345”

White-space characters

\u0020	\u1680	\u2000	\u2001	\u2002	\u2003	\u2004
\u2005	\u2006	\u2007	\u2008	\u2009	\u200A	\u202F
\u205F	\u3000					
\u2028						
\u2029						
\u0009	\u000A	\u000B	\u000C	\u000D	\u0085	\u00A0
\u200B	\uFEFF					

Syntax 2

```
string String_Trim(
    string,
    string
)
```

Parameters

`string` Input string

`string` Specified characters to be removed from leading occurrences

Return

`string` String formed by removing all leading occurrences of specified characters

Syntax 3

```
string String_Trim(
    string,
    string,
    string
)
```

Parameters

`string` Input string

`string` Specified characters to be removed from leading occurrences

`string` Specified characters to be removed from trailing occurrences

Return

`string` String formed by removing all leading and trailing occurrences of the specified characters

Note

var_string s1 = “Hello Hello World Hello World”

var_string s2 = “HelloHelloWorldHelloWorld”

var_value = **String_Trim**(var_s1, “Hello”) // var_value = “ Hello World Hello World”

var_value = **String_Trim**(var_s1, “World”) // var_value = “Hello Hello World Hello World”

var_value = **String_Trim**(var_s1, “”, “Hello”) // var_value = ”Hello Hello World Hello World”

var_value = **String_Trim**(var_s1, “”, “World”) // var_value = ”Hello Hello World Hello ”

var_value = **String_Trim**(var_s1, “Hello”, “World”) // var_value = ” Hello World Hello ”

var_value = **String_Trim**(var_s2, “Hello”) // var_value = ”WorldHelloWorld”

var_value = **String_Trim**(var_s2, “World”) // var_value = ”HelloHelloWorldHelloWorld”

var_value = **String_Trim**(var_s2, “”, “Hello”) // var_value = ”HelloHelloWorldHelloWorld”

var_value = **String_Trim**(var_s2, “”, “World”) // var_value = ”HelloHelloWorldHello”

var_value = **String_Trim**(var_s2, “Hello”, “World”) // var_value = ”WorldHello”

2.21 String_ToLower()

Change all the characters in a string to lower case

Syntax 1

```
string String_ToLower(
    string
)
```

Parameters

`string` Input string

Return

`string` The string formed by converting all the English character into lower case. Non-English character will be remained the same.

Note

var_value = **String_ToLower**(“0x11Acz34”) // var_value = “0x11acz34”

2.22 String_ToUpper()

Change all the characters in a string to upper case

Syntax 1

```
string String_ToUpper(
    string
)
```

Parameters

`string` Input string

Return

`string` The string formed by converting all the English character into upper case. Non-English character will remain the same.

Note

var_value = **String_ToUpper**("0x11Acz34") // var_value = "0X11ACZ34"

2.23 Array_Append()

Add new data as the elements in the end of the array.

Syntax 1

```
?[] Array_Append(
    ?[],
    ? or ?[]
)
```

Parameters

`?[]` Parameter 1, the array to be appended. Available types: byte, int, float, double, bool, and string.

`? or ?[]` Parameter 2, the data or the array to add. The type must be the same with the type of the array to be appended.

*Both parameters must go with the same type.

Return

`?[]` The new array with the parameter 2 elements appended to the parameter 1.

Note

? byte[] var_ n1 = {100, 200, 30}

byte[] var_n2 = {40, 50, 60}

var_n3 = **Array_Append**(var_n1, var_n2) // var_n3 = {100, 200, 30, *40, 50, 60*}

var_n1 = **Array_Append**(var_n1, 100) // var_n1 = {100, 200, 30, *100*}

var_n1 = **Array_Append**(var_n1, var_n3) // var_n1 = {100, 200, 30, 100, *100, 200, 30, 40, 50, 60*}

? float[] var_n1 = {1.1, 2.2, 3.3}

float[] var_n2 = {0.4, 0.5}

var_n3 = **Array_Append**(var_n1, var_n2) // var_n3 = {1.1, 2.2, 3.3, *0.4, 0.5*}

var_n4 = **Array_Append**(var_n3, 5.678) // var_n4 = {1.1, 2.2, 3.3, 0.4, 0.5, *5.678*}

```
?    string[] var_n1 = {“123”, “ABC”, “456”, “DEF”}
     string[] var_n2 = {“ABC”, “123”, “XYZ”}
     var_n3 = Array_Append(var_n1, var_n2)          // var_n3 = {“123”, “ABC”, “456”, “DEF”, “ABC”, “123”, “XYZ”}
     var_n4 = Array_Append(var_n2, “Hello World”)   // var_n4 = {“ABC”, “123”, “XYZ”, “Hello World”}
```

2.24 Array_Insert()

Insert data as the elements in the array.

Syntax1

```
?[] Array_Insert(
    ?[],
    int,
    ? or ?[]
)
```

Parameters

`?[]`	Parameter 1, the array to be inserted. Available types: byte, int, float, double, bool, and string.	
`int`	The index starting address of the parameter 1.	
	0	The length of the array 1 - 1	Legal value
	>=	The length of the array 1	Legal value, and will insert the value in the end of the parameter 1.
	< 0		Illegal value, the project will stop by error.
`? or ?[]`	Parameter 2, the data or the array to insert. The type must be the same with the type of the array to be appended. * Both parameters must go with the same type.	

Return

`?[]`	The new array with the parameter 2 elements inserted to the index starting address of the parameter 1.

Note

```
?    int[] var_n1 = {100, 200, 30}
     int[] var_n2 = {40, 50, 60}
     var_n3 = Array_Insert(var_n1, 0, var_n2)       // var_n3 = {40, 50, 60, 100, 200, 30}
                                                    // Insert to the index 0
     var_n4 = Array_Insert(var_n1, 2, var_n2)       // var_n4 = {100, 200, 40, 50, 60, 30}
                                                    // Insert to the index 2
```

```
var_n5 = Array_Insert(var_n1, -1, var_n2)        // var_n5 = {}
                                                 // The project will stop by error. Illegal index to start with
```

```
?    double[] var_n1 = {1.4, 2.6, 3.9}
     double[] var_n2 = {0.5, 0.7}
     var_n3 = Array_Insert(var_n1, 1, var_n2)    // var_n3 = {1.4, 0.5, 0.7, 2.6, 3.9}
     var_n4 = Array_Insert(var_n3, 4, 1.2345)    // var_n4 = {1.4, 0.5, 0.7, 2.6, 1.2345, 3.9}
     var_n5 = Array_Insert(var_n3, 100, 9)       // var_n5 = {1.4, 0.5, 0.7, 2.6, 3.9, 9}
                                                 // Out of the index. The value will insert in the end of the array.
```

2.25 Array_Remove()

Delete data as the elements in the array.

Syntax1

```
?[] Array_Remove(
    ?[],
    int,
    int
)
```

Parameters

?[]	Parameter 1, the array to be inserted. Available types: byte, int, float, double, bool, and string.	
int	The index starting address of the parameter 1 to remove.	
	0	The length of the parameter 1 - 1	Legal value
	>=	The length of the parameter 1	Illegal value, the project will stop by error.
	< 0		Illegal value, the project will stop by error.
int	The number of the elements to remove	
	> 0	The number of the elements to remove from the index starting address or until the end of the array.	
	< 0	The number will be 0 and no element will be removed.	

Return

?[]	The new array with elements removed after the index staring address.

Syntax2

```
?[] Array_Remove(
    ?[],
```

```
    int
)
```

Note

Same as syntax 1. The default number of the elements to remove is 1.

```
?   int[] var_n1 = {100, 200, 30, 40, 50, 60}
    var_n3 = Array_Remove(var_n1, -1)          // var_n3 = {}
                                               // The project will stop by error. Illegal value to start with.
    var_n4 = Array_Remove(var_n1, 100)         // var_n4 = {}
                                               // The project will stop by error. Illegal value to start with.
    var_n5 = Array_Remove(var_n1, 0)           // var_n5 = {200, 30, 40, 50, 60}    // Remove index 0
    var_n6 = Array_Remove(var_n1, 1, 2)        // var_n6 = {100, 40, 50, 60}
                                               // Remove 2 elements from index 1
    var_n7 = Array_Remove(var_n1, 1, 100)      // var_n7 = {100}
                                               // Remove 100 elements from index 1 (remove to the end of the
                                               array)
    var_n8 = Array_Remove(var_n1, Length(var_n1)-1)   // var_n8 = {100, 200, 30, 40, 50}
                                                      // Remove from the last of index
    var_n9 = Array_Remove(var_n1, Length(var_n1))     // var_n9 = {}
                                                      // The project will stop by error. Illegal value to start with.
```

2.26 Array_Equals()

Determine whether the specified two arrays are identical.

Syntax 1

```
bool Array_Equals(
    ?[],
    ?[]
)
```

Parameters

?[]	Input array1 (Data type can be byte, int, float, double, bool, string)
?[]	Input array2 (Data type can be byte, int, float, double, bool, string)
	* The data type of array1 and array2 must be identical.

Return

bool	Two arrays are identical or not?	
	true	two arrays are identical
	false	two arrays are not identical

Syntax 2

```
bool Array_Equals(
    ?[],vv
    int,
    ?[],
    int,
    int
)
```

Parameters

`?[]`	Input array1 (Data type can be byte, int, float, double, bool, string)
`int`	The starting index of array1 (0 .. (length of arry1)-1)
`?[]`	Input array2 (Data type can be byte, int, float, double, bool, string)
`int`	The starting index of array2 (0 .. (length of arry2)-1)
`int`	The number of elements to be compared (0: return true)
	* The data type of array1 and array2 must be identical.

Return

`bool`	The assigned elements in two arrays are identical or not?	
	`true`	identical
	`false`	not identical (or parameters are not valid)

Note

```
? byte[] var_n1 = {100, 200, 30}
  byte[] var_n2 = {100, 200, 30}
  Array_Equals(var_n1, var_n2)                        // true
  Array_Equals(var_n1, 0, var_n2, 0, 3)               // true
  Array_Equals(var_n1, 0, var_n2, 0, Length(var_n2))  // true

? int[] var_n1 = {1000, 2000, 3000}
  int[] var_n2 = {1000, 2000, 3000, 4000}
  Array_Equals(var_n1, var_n2)                        // false
  Array_Equals(var_n1, 0, var_n2, 0, Length(var_n2))  // false    // compare 4 elements
  Array_Equals(var_n1, 0, var_n2, 0, 3)               // true
? float[] var_n1 = {1.1, 2.2, 3.3}
  float[] var_n2 = {1.1, 2.2}
  Array_Equals(var_n1, var_n2)                        // false
  Array_Equals(var_n1, 0, var_n2, 0, Length(var_n2))  // true     // compare 2 elements
```

```
    Array_Equals(var_n1, 0, var_n2, 0, Length(var_n1))    // false
?   double[] var_n1 = {100, 200, 300, 3.3, 2.2, 1.1}
    double[]var_ n2 = {100, 200, 400, 3.3, 2.2, 4.4}
    Array_Equals(var_n1, var_n2)                          // false
    Array_Equals(var_n1, 0, var_n2, 0, Length(var_n2))    // false
    Array_Equals(var_n1, 0, var_n2, 0, 2)                 // true
    Array_Equals(var_n1, 3, var_n2, 3, 2)                 // true
?   bool[] var_n1 = {true, false, true, true, true}
    bool[] var_n2 = {true, false, true, false, true}
    Array_Equals(var_n1, var_n2)                          // false
    Array_Equals(var_n1, 0, var_n2, 0, -1)                // false
    Array_Equals(var_n1, 0, var_n2, 0, 0)                 // true     // compare 0 element
?   string[] var_n1 = {“123”, “ABC”, “456”, “DEF”}
    string[] var_n2 = {“123”, “ABC”, “456”, “DEF”}
    Array_Equals(var_n1, var_n2)                          // true
    Array_Equals(var_n1, -1, var_n2, 0, 4)                // false    // Invalid starting index
```

2.27 Array_IndexOf()

Search for the specified element and returns the index of its first occurrence in the input array

Syntax 1

```
int Array_IndexOf(
    ?[],
    ?
)
```

Parameters

`?[]`	input array (Data type can be byte, int, float, double, bool, string)
`?`	The target element to search (The data type needs to be the same as the input array `?[]`, but not an array

Return

`int`	`0..(length of input array)-1`	If the element is found , returns the index value
	`-1`	No element found

Note

```
?   byte[] var_n = {100, 200, 30}
    var_value = Array_IndexOf(var_n, 200)          // 1
```

```
    var_value = Array_IndexOf(var_n, 2000)          // error // 2000 is not byte data
?   int[] var_n = {1000, 2000, 3000}
    var_value = Array_IndexOf(var_n, 200)           // -1
?   float[] var_n = {1.1, 2.2, 3.3}
    var_value = Array_IndexOf(var_n, 1.1)           // 0
?   double[] var_n = {100, 200, 300, 3.3, 2.2, 1.1}
    var_value = Array_IndexOf(var_n, 1.1)           // 5
?   bool[] var_n = {true, false, true, true, true}
    var_value = Array_IndexOf(var_n, true)          // 0
?   string[] var_n = {"123", "ABC", "456", "DEF"}
    var_value = Array_IndexOf(var_n, "456")         // 2
```

2.28 Array_LastIndexOf()

Search for the specified element and returns the index of the last occurrence within the entire Array.

Syntax 1

```
int Array_LastIndexOf(
    ?[],
    ?
)
```

Parameters

?[]	input array (Data type can be byte, int, float, double, bool, string)
?	The target element to search (The data type needs to be the same as the input array ?[], but not an array

Return

int	0..(length of input array)-1	If the element is found , returns the index value
	-1	No element found

Note

```
?   byte[] var_n = {100, 200, 30}
    var_value = Array_LastIndexOf(var_n, 200)       // 1
    var_value = Array_LastIndexOf(var_n, 2000)      // error // 2000 is not byte data
?   int[] var_n = {1000, 2000, 3000}
    var_value = Array_LastIndexOf(var_n, 200)       // -1
?   float[] var_n = {1.1, 2.2, 3.3}
    var_value = Array_LastIndexOf(var_n, 1.1)       // 0
```

```
?    double[] var_n = {100, 200, 300, 3.3, 2.2, 1.1}
     var_value = Array_LastIndexOf(var_n, 1.1)          // 5
?    bool[] var_n = {true, false, true, true, true}
     var_value = Array_LastIndexOf(var_n, true)         // 4
?    string[]var_ n = {“123”, “ABC”, “456”, “DEF”}
     var_value = Array_LastIndexOf(var_n, “456”)        // 2
```

2.29 Array_Reverse()

Reverse the sequence of the elements in the array

Syntax 1

```
? [ ] Array_Reverse (
    ? [ ]
)
```

Parameters

? [ ]	input array (Data type can be byte, int, float, double, bool, string)

Return

? [ ]	The reversed array

Note

```
?    byte[] var_n = {100, 200, 30}
     var_n = Array_Reverse(var_n)          // var_n = {30, 200, 100}
?    int[] n = {1000, 2000, 3000}
     var_n = Array_Reverse(var_n)          // var_n = {3000, 2000, 1000}
?    float[] var_n = {1.1, 2.2, 3.3}
     var_n = Array_Reverse(var_n)          // var_n = {3.3, 2.2, 1.1}
?    double[] var_n = {100, 200, 300, 3.3, 2.2, 1.1}
     var_n = Array_Reverse(var_n)          // var_n = {1.1, 2.2, 3.3, 300, 200, 100}
?    bool[] var_n = {true, false, true, true, true}
     var_n = Array_Reverse(var_n)          // var_n = {true, true, true, false, true}
?    string[] var_n = {“123”, “ABC”, “456”, “DEF”}
     var_n = Array_Reverse(var_n)          // var_n = {“DEF”, “456”, “ABC”, “123”}
```

Syntax 2

```
? [ ] Array_Reverse (
    ? [ ],
```

int

)

Parameters

?[] input array (Data type can be byte, int, float, double, bool, string)

int the number of elements to be viewed as a section to be reversed

2 2 elements as a section

4 4 elements as a section

8 8 elements as a section

* The sequence of the elements in the same section will be reversed, but the sequence of the sections will remain the same

Return

?[] The reversed array

Note

? byte[] var_n = {100, 200, 30}

var_n = **Array_Reverse**(var_n, 2) // var_n = {200, 100, 30}
// 2 elements as a section, that is {100,200}{30}

var_n = **Array_Reverse**(var_n, 4) // var_n = {30, 200, 100}
// 4 elements as a section, that is {100,200,30}

var_n = **Array_Reverse**(var_n, 8) // var_n = {30, 200, 100}

? int[] var_n = {100, 200, 300, 400}

var_n = **Array_Reverse**(var_n, 2) // var_n = {200, 100, 400, 300}
// 2 elements as a section, that is {100,200}{300,400}

var_n = **Array_Reverse**(var_n, 4) // var_n = {400, 300, 200, 100}
// 4 elements as a section, that is {100,200,300,400}

var_n = **Array_Reverse**(var_n, 8) // var_n = {400, 300, 200, 100}

? float[] var_n = {1.1, 2.2, 3.3, 4.4, 5.5}

var_n = **Array_Reverse**(var_n, 2) // var_n = {2.2, 1.1, 4.4, 3.3, 5.5}
// 2 elements as a section, that is {1.1,2.2}{3.3,4.4}{5.5}

var_n = **Array_Reverse**(var_n, 4) // var_n = {4.4, 3.3, 2.2, 1.1, 5.5}
// 4 elements as a section, that is {1.1,2.2,3.3,4.4}{5.5}

var_n = **Array_Reverse**(var_n, 8) // var_n = {5.5, 4.4, 3.3, 2.2, 1.1}

? double[] var_n = {100, 200, 300, 400, 4.4, 3.3, 2.2, 1.1, 50, 60, 70, 80}

var_n = **Array_Reverse**(var_n, 2) // var_n = {200, 100, 400, 300, 3.3, 4.4, 1.1, 2.2, 60, 50, 80, 70}

var_n = **Array_Reverse**(var_n, 4) // var_n = {400, 300, 200, 100, 1.1, 2.2, 3.3, 4.4, 80, 70, 60, 50}

var_n = **Array_Reverse**(var_n, 8) // var_n = {1.1, 2.2, 3.3, 4.4, 400, 300, 200, 100, 80, 70, 60, 50}

? bool[] var_n = {true, false, true, true, true, false, true, false}

var_n = **Array_Reverse**(var_n, 2) // var_n = {false, true, true, true, false, true, false, true }

```
var_n = Array_Reverse(var_n,  4)   // var_n = {true, true, false, true, false, true, false, true}
var_n = Array_Reverse(var_n,  8)   // var_n = {false, true, false, true, true, true, false, true}
? string[] var_n = {“123”, “ABC”, “456”, “DEF”, “000”, “111”}
var_n = Array_Reverse(var_n,  2)   // var_n = {“ABC”, “123”, “DEF”, “456”, “111”, “000”}
var_n = Array_Reverse(var_n,  4)   // var_n = {“DEF”, “456”, “ABC”, “123”, “111”, “000”}
var_n = Array_Reverse(var_n,  8)   // var_n = {“111”, “000”, “DEF”, “456”, “ABC”, “123”}
```

2.30 Array_Sort()

Sort the elements in a array

Syntax 1

```
?[] Array_Sort(
    ?[],
    int
)
```

Parameters

?[]	input array (Data type can be byte, int, float, double, bool, string)
int	Sorting direction
0	Ascending Order (Default)
1	Descending Order

Return

?[]	The array after sorting

Syntax 2

```
?[] Array_Sort(
    ?[]
)
```

Note

Similar to Syntax1 with sorting direction as ascending order

Array_Sort(array[]) => **Array_Sort**(array[], 0)

```
? int[] var_n =  {1000, 2000, 3000}
var_n =  Array_Sort(var_n )          // var_n =  {1000, 2000, 3000}
? double[] var_n =  {100, 200, 300, 3.3, 2.2, 1.1}
var_n =  Array_Sort(var_n , 1)       // var_n =  {300, 200, 100, 3.3, 2.2, 1.1}
? bool[] var_n =  {true, false, true, true, true}
```

```
    var_n =  Array_Sort(var_n , 1)        // var_n =  {true, true, true, true, false}
?   string[] var_n =  {“123”, “ABC”, “456”, “DEF”}
    var_n =  Array_Sort(var_n )           // var_n =  {“123”, “456”, “ABC”, “DEF”}
```

2.31 Array_SubElements()

Retrieve the sub-elements from input array

Syntax 1

```
?[] Array_SubElements(
    ?[],
    int,
    int
)
```

Parameters

?[]	Input array (Data type can be byte, int, float, double, bool, string)
int	The starting index of sub-elements. (0 .. (length of array)-1)
int	The number of element in sub-elements

Return

?[]	The sub-elements from input arrays If starting index <0, sub-elements equals to empty array If starting index >= length of input array, sub-elements equals to empty array If sub-element number <0, sub-elements starts at starting index to the last element of input array If the sum of starting index and the number of element exceeds the length of the input array, sub-elements starts at starting index to the last element of input array

Syntax 2

```
?[] Array_SubElements(
    ?[],
    int
)
```

Note

Similar to Syntax 1, but the sub-elements starts at starting index to the last element of input array

Array_SubElements(array[], 2) => **Array_SubElements**(array[], 2, maxlen)

```
?   byte[] var_n =  {100, 200, 30}
    var_n1 =  Array_SubElements(var_n1 , 0)       // var_n1 =  {100, 200, 30}
```

```
var_n1 =  Array_SubElements(var_n1 , -1)     // var_n1 =  {}
var_n1 =  Array_SubElements(var_n1 , 0, 3)   // var_n1 =  {100, 200, 30}
var_n1 =  Array_SubElements(var_n1 , 1, 3)   // var_n1 =  {200, 30}
var_n1 =  Array_SubElements(var_n1 , 2)      // var_n1 =  {30}
var_n1 =  Array_SubElements(var_n1 , 3, 3)   // var_n1 =  {}
? int[] var_n =  {1000, 2000, 3000}
var_n1 =  Array_SubElements(var_n1 , 0)      // var_n1 =  {1000, 2000, 3000}
var_n1 =  Array_SubElements(var_n1 , -1)     // var_n1 =  {}
var_n1 =  Array_SubElements(var_n1 , 1, 3)   // var_n1 =  {2000, 3000}
var_n1 =  Array_SubElements(var_n1 , 2)      // var_n1 =  {3000}
? float[] var_n =  {1.1, 2.2, 3.3}
var_n1 =  Array_SubElements(var_n1 , 0)      // var_n1 =  {1.1, 2.2, 3.3}
var_n1 =  Array_SubElements(var_n1 , -1)     // var_n1 =  {}
var_n1 =  Array_SubElements(var_n1 , 1, 3)   // var_n1 =  {2.2, 3.3}
var_n1 =  Array_SubElements(var_n1 , 2)      // var_n1 =  {3.3}
? double[] var_n =  {100, 200, 3.3, 2.2, 1.1}
var_n1 =  Array_SubElements(var_n1 , 0)      // var_n1 =  {100, 200, 3.3, 2.2, 1.1}
var_n1 =  Array_SubElements(var_n1 , -1)     // var_n1 =  {}
var_n1 =  Array_SubElements(var_n1 , 1, 3)   // var_n1 =  {200, 3.3, 2.2}
var_n1 =  Array_SubElements(var_n1 , 2)      // var_n1 =  {3.3, 2.2, 1.1}
? bool[] var_n =  {true, false, true, true, true}
var_n1 =  Array_SubElements(var_n1 , 0)      // var_n1 =  {true, false, true, true, true}
var_n1 =  Array_SubElements(var_n1 , -1)     // var_n1 =  {}
var_n1 =  Array_SubElements(var_n1 , 1, 3)   // var_n1 =  {false, true, true}
var_n1 =  Array_SubElements(var_n1 , 2)      // var_n1 =  {true, true, true}
? string[] var_n =  {“123”, “ABC”, “456”, “DEF”}
var_n1 =  Array_SubElements(var_n1 , 0)      // var_n1 =  {“123”, “ABC”, “456”, “DEF”}
var_n1 =  Array_SubElements(var_n1 , -1)     // var_n1 =  {}
var_n1 =  Array_SubElements(var_n1 , 1, 3)   // var_n1 =  {“ABC”, “456”, “DEF”}
var_n1 =  Array_SubElements(var_n1 , 2)      // var_n1 =  {“456”, “DEF”}
```

2.32 ValueReverse()

Reverse the sequence of byte units inside input data (int 2 bytes or 4 bytes, float 4 bytes, double 8 bytes); or reverse the sequence of character of string.

Syntax 1

```
int ValueReverse(
```

```
    int,
    int
)
```

Parameters

`int` Input value

`int` The input value follows int32 or int16 format

`0` int32 (Default)

`1` int16. If the data does not meets int16 format, int32 will be applied instead.

`2` int16. Forced to apply int16 format. For int32 data input, there could be some bytes missing

Return

`int` The value formed from reversing the sequence of byte units inside the input value. For Int32 data, reverse with 4 bytes. For int16 data, reverse with 2 bytes.

Note

```
int var_i = 10000
var_value = ValueReverse(var_i, 0)   // 10000=0x00002710 → 0x10270000   // var_value = 270991360
var_value = ValueReverse(var_i, 1)   // 10000=0x2710     → 0x1027       // var_value = 4135
var_i = 100000                       // int32 value
var_value = ValueReverse(var_i, 0)   // 100000=0x000186A0 → 0xA0860100 // var_value = -1601830656
var_value = ValueReverse(var_i, 1)   // 100000=0x000186A0 → 0xA0860100 // var_value = -1601830656
var_value = ValueReverse(var_i, 2)   // 100000=0x000086A0 → 0xA0860000 // var_value = -24442
```

Syntax 2

```
int ValueReverse(
    int
)
```

Parameters

`int` Input value

Note

Similar to Syntax1 with int32 input format

ValueReverse(int) => **ValueReverse**(int, 0)

Syntax 3

```
float ValueReverse(
    float
)
```

Parameters

`float` Input value

Return

`float` The value formed from reversing the sequence of byte units inside the input value. For float data, reverse 4 bytes.

Note

float var_i = 40000

var_value = **ValueReverse**(var_i) // 40000=0x471C4000 → 0x00401C47 // var_value = 5.887616E-39

Syntax 4

```
double ValueReverse(
    double
)
```

Parameters

`double` Input value

Return

`double` The value formed from reversing the sequence of byte units inside the input value. For double data, reverse 8 bytes.

Note

double var_i = 80000

var_value = **ValueReverse**(var_i) // 80000=0x40F3880000000000 → 0x000000000088F340 // var_value = 4.43432217445369E-317

Syntax 5

```
string ValueReverse(
    string
)
```

Parameters

`string` Input string

Return

`string` The value formed from reversing the sequence of characters of input string.

Note

string var_i = “ABCDEF”

var_value = **ValueReverse**(var_i) // var_value = “FEDCBA”

Syntax 6

```
int[] ValueReverse(
    int[],
    int
)
```

Parameters

`int[]` Input array value

`int` The input value follows int32 or int16 format

`0` int32 (Default)

`1` int16. If the data does not meets int16 format, int32 will be applied instead.

`2` int16. Forced to apply int16 format. For int32 data input, there could be some bytes missing

Return

`int[]` The array formed from reversing the sequence of byte units inside every element of the input array.

Note

```
int[] var_i = {10000, 20000, 60000, 80000}
var_value = ValueReverse(var_i, 0)    // var_value = {270991360, 541982720, 1625948160, -2143813376}
var_value = ValueReverse(var_i, 1)    // var_value = {4135, 8270, 1625948160, -2143813376}
var_value = ValueReverse(var_i, 2)    // var_value = {4135, 8270, 24810, -32712}
```

Syntax 7

```
int[] ValueReverse(
    int[]
)
```

Parameters

`int[]` Input array value

Note

Similar to Syntax6 with input integer as int32

ValueReverse(int[]) => **ValueReverse**(int[], 0)

Syntax 8

```
float[] ValueReverse(
    float[]
)
```

Parameters

`float[]` Input array value

Return

`float[]` The array formed from reversing the sequence of byte units inside every element of the input array.

Note

```
float[] var_i = {10000, 20000}
var_value = ValueReverse(var_i)          // var_value = {5.887614E-39, 5.933532E-39}
```

Syntax 9

```
double[] ValueReverse(
    double[]
)
```

Parameters

`double[]` Input array value

Return

`double[]` The array formed from reversing the sequence of byte units inside every element of the input array.

Note

```
double[] var_i = {10000, 20000}
var_value = ValueReverse(var_i)          // var_value = {4.42825109579759E-317, 4.43027478868296E-317}
```

Syntax 10

```
string[] ValueReverse(
    string[]
)
```

Parameters

`string[]` Input string array

Return

`string[]` The string array formed from reversing the string inside every element of the input string array.

Note

```
string[] var_i = {“ABCDEFG”, “12345678”}
var_value = ValueReverse(var_i)          // var_value = {“GFEDCBA”, “87654321”}
```

2.33 GetBytes()

Convert arbitrary data type to byte array.

Syntax 1

```
byte[] GetBytes(
    ?,
    int
)
```

Parameters

`?`	The input data. Data type can be int, float, double, bool, string or array.
`int`	The input data as integers and floating points follows Little Endian or Big Endian
	`0` Little Endian (Default)
	`1` Big Endian
	The input data as string arrays separates with 0x00 0x00 for each element
	`0` Not separate with 0x00 0x00 (Default)
	`1` Separate with 0x00 0x00

Return

`byte[]`	The byte array formed by input data

Syntax 2

```
byte[] GetBytes(
    ?
)
```

Note

Same as syntax 1 with Little Endian or Big Endian defaults to 0 such as returns based on Little Endian

GetBytes(?) => **GetBytes**(?, 0)

```
?   byte var_n = 100
    var_value = GetBytes(var_n)          // var_value = {0x64}
    var_value = GetBytes(var_n, 0)       // var_value = {0x64}
    var_value = GetBytes(var_n, 1)       // var_value = {0x64}

?   byte[] var_n = {100, 200}            // Convert every element of the array to byte, 1 byte as a single unit.
    var_value = GetBytes(var_n)          // var_value = {0x64, 0xC8}
    var_value = GetBytes(var_n, 0)       // var_value = {0x64, 0xC8}
    var_value = GetBytes(var_n, 1)       // var_value = {0x64, 0xC8}

?   int
```

```
var_value = GetBytes(123456)          // var_value = {0x40, 0xE2, 0x01, 0x00}
var_value = GetBytes(123456, 0)       // var_value = {0x40, 0xE2, 0x01, 0x00}
var_value = GetBytes(0x123456, 0)     // var_value = {0x56, 0x34, 0x12, 0x00}
var_value = GetBytes(0x1234561, 1)    // var_value = {0x01, 0x23, 0x45, 0x61}
```

? `int[] var_n = {10000, 20000, 80000}`

```
// Convert every single element of the array to byte. For int32 data, works on 4 bytes sequentially.
var_value = GetBytes(var_n)
// var_value = {0x10, 0x27, 0x00, 0x00, 0x20, 0x4E, 0x00, 0x00, 0x80, 0x38, 0x01, 0x00}
var_value = GetBytes(var_n, 0)
// var_value = {0x10, 0x27, 0x00, 0x00, 0x20, 0x4E, 0x00, 0x00, 0x80, 0x38, 0x01, 0x00}
var_value = GetBytes(var_n, 1)
// var_value = {0x00, 0x00, 0x27, 0x10, 0x00, 0x00, 0x4E, 0x20, 0x00, 0x01, 0x38, 0x80}
```

? `float`

```
var_value = GetBytes(123.456, 0)      // var_value = {0x79, 0xE9, 0xF6, 0x42}
float var_n = -1.2345
var_value = GetBytes(var_n, 0)        // var_value = {0x19, 0x04, 0x9E, 0xBF}
var_value = GetBytes(var_n, 1)        // var_value = {0xBF, 0x9E, 0x04, 0x19}
```

? `float[] var_n = {1.23, 4.56, -7.89}`

```
// Convert every single element of the array to byte. For float data, works on 4 bytes sequentially.
var_value = GetBytes(var_n)
// var_value = {0xA4, 0x70, 0x9D, 0x3F, 0x85, 0xEB, 0x91, 0x40, 0xE1, 0x7A, 0xFC, 0xC0}
var_value = GetBytes(var_n, 0)
// var_value = {0xA4, 0x70, 0x9D, 0x3F, 0x85, 0xEB, 0x91, 0x40, 0xE1, 0x7A, 0xFC, 0xC0}
var_value = GetBytes(var_n, 1)
// var_value = {0x3F, 0x9D, 0x70, 0xA4, 0x40, 0x91, 0xEB, 0x85, 0xC0, 0xFC, 0x7A, 0xE1}
```

? `double var_n = -1.2345`

```
var_value = GetBytes(var_n, 0)  // var_value = {0x8D, 0x97, 0x6E, 0x12, 0x83, 0xC0, 0xF3, 0xBF}
var_value = GetBytes(var_n, 1)  // var_value = {0xBF, 0xF3, 0xC0, 0x83, 0x12, 0x6E, 0x97, 0x8D}
```

? `double[] var_n = {1.23, -7.89}`

```
// Convert every single element of the array to byte. For double data, works on 8 bytes sequentially.
var_value = GetBytes(var_n)
// var_value = {0xAE,0x47,0xE1,0x7A,0x14,0xAE,0xF3,0x3F,0x8F,0xC2,0xF5,0x28,0x5C,0x8F,0x1F,0xC0}
var_value = GetBytes(var_n, 0)
```

```
// var_value = {0xAE,0x47,0xE1,0x7A,0x14,0xAE,0xF3,0x3F,0x8F,0xC2,0xF5,0x28,0x5C,0x8F,0x1F,0xC0}
var_value = GetBytes(var_n, 1)
// var_value = {0x3F,0xF3,0xAE,0x14,0x7A,0xE1,0x47,0xAE,0xC0,0x1F,0x8F,0x5C,0x28,0xF5,0xC2,0x8F}
```

? `bool var_flag = true` // true is converted to 1；false is converted to 0

```
var_value = GetBytes(flag)        // var_value = {1}
var_value = GetBytes(flag, 0)
// var_value = {1}// Because bool is 1 byte, Endian Parameters are not sufficient.
var_value = GetBytes(flag, 1)     // var_value = {1}
```

? `bool[] var_flag = {true, false, true, false, false, true, true}`

```
var_value = GetBytes(flag)        // var_value = {1, 0, 1, 0, 0, 1, 1}
var_value = GetBytes(flag, 0)     // var_value = {1, 0, 1, 0, 0, 1, 1}
var_value = GetBytes(flag, 1)     // var_value = {1, 0, 1, 0, 0, 1, 1}
```

? `string var_n = “ABCDEFG”` // string to encode in UTF8

```
var_value = GetBytes(var_n)       // var_value = {0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47}
var_value = GetBytes(var_n, 0)
    // var_value = {0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47} // Endian Parameters not sufficient
var_value = GetBytes(var_n, 1)    // var_value = {0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47}
```

? `string[] var_n = {“ABC”, “DEF”, “達明機器人” }`

```
var_value = GetBytes(var_n)
// var_value = {0x41, 0x42, 0x43, 0x44, 0x45, 0x46,
                0xE9,0x81,0x94,0xE6,0x98,0x8E,0xE6,0xA9,0x9F,0xE5,0x99,0xA8,0xE4,0xBA,0xBA}
var_value = GetBytes(var_n, 1)
// var_value = {0x41, 0x42, 0x43, 0x00, 0x00, 0x44, 0x45, 0x46, 0x00, 0x00,
                0xE9,0x81,0x94,0xE6,0x98,0x8E,0xE6,0xA9,0x9F,0xE5,0x99,0xA8,0xE4,0xBA,0xBA}
```

*Conversion of string[] to byte[] without separation bytes will save the content completely, but it is unable to turn byte[] back to string[] effectively.

*It is effective to turn byte[] back to string[] by inserting separation bytes (2 consecutive 0x00s) between the elements in the array, but it is possible to find conversion errors if the value of the string come with 0x00 0x00.

Syntax 3

Convert integer (int type) to byte array.

```
byte[] GetBytes(
```

```
    int,
    int,
    int
)
```

Parameters

`int`	The input integer (int type)
`int`	The input value follows Little Endian or Big Endian `0` Little Endian (Default) `1` Big Endian
`int`	The input integer value's data type is int32 or int16 `0` int32 (Default) `1` int16. If the data does not meets int16 format, int32 will be applied instead. `2` int16. Forced to apply int16 format. For int32 data input, there could be some bytes missing.

Return

`byte[]`	The byte array formed by input integer. For int32 data, convert with 4 bytes. For int16 data, convert with 2 bytes.

Note

```
var_value = GetBytes(12345, 0, 0)          // var_value = {0x39, 0x30, 0x00, 0x00}
var_value = GetBytes(12345, 0, 1)          // var_value = {0x39, 0x30}
var_value = GetBytes(12345, 0, 2)          // var_value = {0x39, 0x30}
var_value = GetBytes(0x123456, 0, 0)       // var_value = {0x56, 0x34, 0x12, 0x00}
var_value = GetBytes(0x123456, 0, 1)       // var_value = {0x56, 0x34, 0x12, 0x00}
var_value = GetBytes(0x123456, 0, 2)       // var_value = {0x56, 0x34}    // bytes missing
var_value = GetBytes(0x1234561, 1, 0)      // var_value = {0x01, 0x23, 0x45, 0x61}
var_value = GetBytes(0x1234561, 1, 1)      // var_value = {0x01, 0x23, 0x45, 0x61}
var_value = GetBytes(0x1234561, 1, 2)      // var_value = {0x45, 0x61}    // bytes missing
```

Syntax 4

Convert the integer array (int[] type) to byte array

```
byte[] GetBytes(
    int[],
    int,
    int
)
```

Parameters

`int[]`	The input integer array (int[] type)
`int`	The input integer array follows Little Endian or Big Endian

0 Little Endian (Default)

1 Big Endian

`int` The input integer array's data type is int32 or int16

0 int32 (Default)

1 int16. If the data does not meets int16 format, int32 will be applied instead

2 int16. Forced to apply int16 format. For int32 data input, there could be some bytes missing.

Return

`byte[]` The byte array formed by input integer array. Every element is converted independently and forms an array. For int32 data, convert with 4 bytes. For int16 data, convert with 2 bytes.

Note

```
var_i ={10000, 20000, 80000}
var_value = GetBytes(var_i, 0, 0)
    // var_value = {0x10, 0x27, 0x00, 0x00, 0x20, 0x4E, 0x00, 0x00, 0x80, 0x38, 0x01, 0x00}
var_value = GetBytes(var_i, 0, 1)    // var_value = {0x10, 0x27, 0x20, 0x4E, 0x80, 0x38, 0x01, 0x00}
var_value = GetBytes(var_i, 0, 2)    // var_value = {0x10, 0x27, 0x20, 0x4E, 0x80, 0x38}   // bytes missing
var_value = GetBytes(var_i, 1, 0)
    // var_value = {0x00, 0x00, 0x27, 0x10, 0x00, 0x00, 0x4E, 0x20, 0x00, 0x01, 0x38, 0x80}
var_value = GetBytes(var_i, 1, 1)    // var_value = {0x27, 0x10, 0x4E, 0x20, 0x00, 0x01, 0x38, 0x80}
var_value = GetBytes(var_i, 1, 2)    // var_value = {0x27, 0x10, 0x4E, 0x20, 0x38, 0x80}   // bytes missing
```

2.34 GetString()

Convert arbitrary data type to string

Syntax 1

```
string GetString(
    ?,
    int,
    int
)
```

Parameters

`?` The input data. Data type can be int, float, double, bool, string or array.

`int` The output string's notation is decimal, hexadecimal or binary (Can be only applied to hexadecimal or binary number)

10 decimal

16 hexadecimal

2 binary

String's notation

123	decimal
0x7F	hexadecimal
0b101	binary

When the input value is a string array, the output string value is in standard string format or not.

0 or 10	Automatic detection. If the values in the string come with double quotations or commas, it converse to standard string format.
1	Mandatory conversion to standard string format
0ther	No conversion

`int` The output string format (Can be applied to hexadecimal or binary number only)

0	Fill up digits. Add prefix 0x or 0b, e.g. 0x0C or 0b00001100
1	Fill up digits. No prefix 0x or 0b, e.g. 0C or 00001100
2	Don't fill up digits. Add prefix 0x or 0b, e.g. 0xC or 0b1100
3	Don't fill up digits. No prefix 0x or 0b, e.g. C or 1100

Return

`string` String converted from input data. If the input data cannot be converted, returns empty string. If the input data is array, every element is converted respectively, and returned in "{ , , }" format

Syntax 2

```
string GetString(
    ?,
    int
)
```

Note

Similar to Syntax1 with filling up digits and adding prefix 0x or 0b.

GetString(?, 16) => **GetString**(?, 16, 0)

Syntax 3

```
string GetString(
    ?
)
```

Note

Same as syntax 1. The output string's notation defaults to 10 and the output string format defaults to 0.

GetString(?) => **GetString**(?, 10, 0)

GetString(?) => **GetString**(?, 0, 0) // supposed ? is a string array

? byte var_n = 123

var_value = **GetString**(var_n) // var_value = “123”

var_value = **GetString**(var_n, 10) // var_value = “123”

var_value = **GetString**(var_n, 16) // var_value = “0x7B”

var_value = **GetString**(var_n, 2) // var_value = “0b01111011”

var_value = **GetString**(var_n, 16, 3) // var_value = “7B”

var_value = **GetString**(var_n, 2, 2) // var_value = “0b1111011”

? byte[] var_n = {12, 34, 56}

var_value = **GetString**(var_n) // var_value = “{12,34,56}”

var_value = **GetString**(var_n, 10) // var_value = “{12,34,56}”

var_value = **GetString**(var_n, 16) // var_value = “{0x0C,0x22,0x38}”

var_value = **GetString**(var_n, 2) // var_value = “{0b00001100,0b00100010,0b00111000}”

var_value = **GetString**(var_n, 16, 3) // var_value = “{C,22,38}”

var_value = **GetString**(var_n, 2, 2) // var_value = “{0b1100,0b100010,0b111000}”

? int var_n = 1234

var_value = **GetString**(var_n) // var_value = “1234”

var_value = **GetString**(var_n, 10) // var_value = “1234”

var_value = **GetString**(var_n, 16) // var_value = “0x000004D2”

var_value = **GetString**(var_n, 2) // var_value = “0b00000000000000000000010011010010”

var_value = **GetString**(var_n, 16, 3) // var_value = “4D2”

var_value = **GetString**(var_n, 2, 2) // var_value = “0b10011010010”

? int[] var_n = {123, 345, -123, -456}

var_value = **GetString**(var_n) // var_value = “{123,345,-123,-456}”

var_value = **GetString**(var_n, 10) // var_value = “{123,345,-123,-456}”

var_value = **GetString**(var_n, 16) // var_value = “{0x0000007B,0x00000159,0xFFFFFF85,0xFFFFFE38}”

var_value = **GetString**(var_n, 2) // var_value = “{0b00000000000000000000000001111011,
0b00000000000000000000000101011001,
0b11111111111111111111111110000101,
0b11111111111111111111111000111000}”

var_value = **GetString**(var_n, 16, 3) // var_value = “{7B,159,FFFFFF85,FFFFFE38}”

var_value = **GetString**(var_n, 2, 2) // var_value = “{0b1111011,
0b101011001,
0b11111111111111111111111110000101,

0b11111111111111111111111000111000}”

? float var_n = 12.34

var_value = **GetString**(var_n) // var_value = “12.34”

var_value = **GetString**(var_n, 10) // var_value = “12.34”

var_value = **GetString**(var_n, 16) // var_value = “0x414570A4”

var_value = **GetString**(var_n, 2) // var_value = “0b01000001010001010111000010100100”

var_value = **GetString**(var_n, 16, 3) // var_value = “414570A4”

var_value = **GetString**(var_n, 2, 2) // var_value = “0b10000010100010101110000101001 00”

? float[] var_n = {123.4, 345.6, -123.4, -456.7}

var_value = **GetString**(var_n) // var_value = “{123.4,345.6,-123.4,-456.7}”

var_value = **GetString**(var_n, 10) // var_value = “{123.4,345.6,-123.4,-456.7}”

var_value = **GetString**(var_n, 16) // var_value = “{0x42F6CCCD,0x43ACCCCD,0xC2F6CCCD,0xC3E4599A}”

var_value = **GetString**(var_n, 16, 3) // var_value = “{42F6CCCD,43ACCCCD,C2F6CCCD,C3E4599A}”

? double var_n = 12.34

var_value = **GetString**(var_n) // var_value = “12.34”

var_value = **GetString**(var_n, 10) // var_value = “12.34”

var_value = **GetString**(var_n, 16) // var_value = “0x4028AE147AE147AE”

var_value = **GetString**(var_n, 16, 3) // var_value = “4028AE147AE147AE”

? double[] var_n = {123.45, 345.67, -123.48, -456.79}

var_value = **GetString**(var_n) // var_value = “{123.45,345.67,-123.48,-456.79}”

var_value = **GetString**(var_n, 10) // var_value = “{123.45,345.67,-123.48,-456.79}”

var_value = **GetString**(var_n, 16) // var_value = “{0x405EDCCCCCCCCCCD,0x40759AB851EB851F,
0xC05EDEB851EB851F,0xC07C8CA3D70A3D71}”

var_value = **GetString**(var_n, 16, 3) // var_value = “{405EDCCCCCCCCCCD,40759AB851EB851F,
C05EDEB851EB851F,C07C8CA3D70A3D71}”

? bool var_n = true

var_value = **GetString**(var_n) // var_value = “true”

var_value = **GetString**(var_n, 16) // var_value = “true”

var_value = **GetString**(var_n, 2) // var_value = “true”

var_value = **GetString**(var_n, 16, 3) // var_value = “true”

? bool[] var_n = {true, false, true, false, false, true}

var_value = **GetString**(var_n) // var_value = “{true,false,true,false,false,true}”

var_value = **GetString**(var_n, 16) // var_value = “{true,false,true,false,false,true}”

var_value = **GetString**(var_n, 2) // var_value = “{true,false,true,false,false,true}”

var_value = **GetString**(var_n, 16, 3) // var_value = “{true,false,true,false,false,true}”

? string var_n = “1234567890”

var_value = **GetString**(var_n) // var_value = “1234567890”

var_value = **GetString**(var_n, 16) // var_value = “1234567890”

var_value = **GetString**(var_n, 2) // var_value = “1234567890”

var_value = **GetString**(var_n, 16, 3) // var_value = “1234567890”

? string[] var_n = {“123.45”, “345.67”, “-12""3.48”, “-45A6.79”}

var_value = **GetString**(var_n) // var_value = “{123.45,345.67,-12""3.48,-45A6.79}”

var_value = **GetString**(var_n, 16) // var_value = “{123.45,345.67,-12""3.48,-45A6.79}”

var_value = **GetString**(var_n, 2) // var_value = “{123.45,345.67,-12""3.48,-45A6.79}” // -12""3.48 displayed as -12"3.48

var_value = **GetString**(var_n, 16, 3) // var_value = “{123.45,345.67,-12""3.48,-45A6.79}”

//use automatic detection as the default

Syntax 4

```
string GetString(
    ?,
    string,
    int,
    int
)
```

Parameters

? The input data. Data type can be int, float, double, bool, string or array.

string Separator for output string (Only effective to array input)

int The output string’s notation is decimal, hexadecimal or binary (Can be only applied to hexadecimal or binary number)

10 decimal

16 hexadecimal

2 binary

String’s notation

123 decimal

0x7F hexadecimal

0b101 binary

When the input value is a string array, the output string value is in standard string format or not.

0 or 10 Automatic detection. If the values in the string come with double quotations or

separation symbols, it converse to standard string format.

1 Mandatory conversion to standard string format

0ther No conversion

int The output string format (Can be only applied to hexadecimal or binary number)

0 Fill up digits. Add prefix 0x or 0b, e.g. 0x0C or 0b00001100

1 Fill up digits. No prefix 0x or 0b, e.g. 0C or 00001100

2 Don't fill up digits. Add prefix 0x or 0b, e.g. 0xC or 0b1100

3 Don't fill up digits. No prefix 0x or 0b, e.g. C or 1100

Return

string String converted from input data. If the input data cannot be converted, returns empty string. If the input data is array, every element is converted respectively, and returned as a string with the assigned separator

Syntax 5

```
string GetString(
    ?,
    string,
    int
)
```

Note

Same as Syntax 4 with filling up digits and adding prefix 0x or 0b

GetString(?, str, 16) => **GetString**(?, str, 16, 0)

Syntax 6

```
string GetString(
    ?,
    string
)
```

Note

Same as Syntax 4. The output string's notation defaults to 10 and the output string format defaults to 0.

GetString(?) => **GetString**(?, 10, 0)

GetString(?) => **GetString**(?, 0, 0) // supposed ? is a string array

```
?    byte var_n = 123
     var_value = GetString(var_n)              // var_value = "123"
     var_value = GetString(var_n, ";", 10)     // var_value = "123"
     var_value = GetString(var_n, "-", 16)     // var_value = "0x7B"
```

var_value = **GetString**(var_n, “#”, 2) // var_value = “0b01111011”

var_value = **GetString**(var_n, “,”, 16, 3) // var_value = “7B”

var_value = **GetString**(var_n, “,”, 2, 2) // var_value = “0b1111011”

* Separator is effective to array input only.

? byte[] var_n = {12, 34, 56}

var_value = **GetString**(var_n, “-“) // var_value = “12-34-56”

var_value = **GetString**(var_n, Ctrl(“\r\n”), 10) // var_value = “12\u0D0A34\u0D0A56”

var_value = **GetString**(var_n, newline, 16) // var_value = “0x0C\u0D0A0x22\u0D0A0x38”

var_value = **GetString**(var_n, NewLine, 2) // var_value = “0b00001100\u0D0A0b00100010\u0D0A0b00111000”

var_value = **GetString**(var_n, “-“, 16, 3) // var_value = “C-22-38”

var_value = **GetString**(var_n, “-“, 2, 2) // var_value = “0b1100-0b100010-0b111000”

* \u0D0A is Newline control character, not string value.

? string[] var_n = {“123.45”, “345.67”, “-12””3.48”, “-45A6.79”}

var_value = **GetString**(var_n, “-“) // var_value = “123.45-345.67-"-12""3.48"-"-45A6.79””

var_value = **GetString**(var_n, “-“, 1) // var_value = “"123.45"-"345.67"-"-12""3.48"-"-45A6.79””

var_value = **GetString**(var_n, “-“, 2) // var_value = “123.45-345.67--12"3.48--45A6.79”

// Troubled for identifying the separation symbols and the negative signs.

Syntax 7

```
string GetString(
    ?,
    string,
    string,
    int,
    int
)
```

Parameters

?	The input data. Data type can be int, float, double, bool, string or array.
string	The index of the output string for array input. (Only effective to ? as array type data) * Support numeric format strings
string	Separator for output string (Only effective to array input)
int	The output string’s notation is decimal, hexadecimal or binary (Can be only applied to hexadecimal or binary number) 10 decimal

16 hexadecimal

2 binary

String's notation

123 decimal

0x7F hexadecimal

0b101 binary

When the input value is a string array, the output string value is in standard string format or not.

0 or 10 Automatic detection. If the values in the string come with double quotations or separation symbols, it converse to standard string format.

1 Mandatory conversion to standard string format

0ther No conversion

int The output string format (Can be only applied to hexadecimal or binary number)

0 Fill up digits. Add prefix 0x or 0b, e.g. 0x0C or 0b00001100

1 Fill up digits. No prefix 0x or 0b, e.g. 0C or 00001100

2 Don't fill up digits. Add prefix 0x or 0b, e.g. 0xC or 0b1100

3 Don't fill up digits. No prefix 0x or 0b, e.g. C or 1100

Return

string Converse the value to the string to return. If unable to converse, it returns an empty string. If the type is array, elements in the array will be conversed to strings with prefixes of the element index value format string separated by separation symbols to return.
There will be no braces.

Syntax 8

```
string GetString(
    ?,
    string,
    string,
    int
)
```

Note

Similar to Syntax7 with filling up digits and adding prefix.

GetString(?, str, str, 16) => **GetString**(?, str, str, 16, 0)

Syntax 9

```
string GetString(
    ?,
    string,
    string
```

)

Note

Similar to Syntax7 with decimal output, with filling up digits and adding prefix.

GetString(?, str, str) => **GetString**(?, str, str, 10, 0)

```
? byte var_n = 123
var_value = GetString(var_n)                    // var_value = "123"
var_value = GetString(var_n, "[0]=", ";", 10)   // var_value = "123"
var_value = GetString(var_n, "[0]=", "-", 16)   // var_value = "0x7B"
var_value = GetString(var_n, "[0]=", "#", 2)    // var_value = "0b01111011"
```

* Index and sepapator are only effective to array input.

```
? byte[] var_n = {12, 34, 56}
var_value = GetString(var_n, "[0]=", "-")                  // var_value = "[0]=12-[1]=34-[2]=56"
var_value = GetString(var_n, "[0]=", Ctrl("\r\n"), 10)     // var_value = "[0]=12\u0D0A[1]=34\u0D0A[2]=56"
var_value = GetString(var_n, "[0]=", newline, 16)          // var_value =
                                                            "[0]=0x0C\u0D0A[1]=0x22\u0D0A[2]=0x38"
var_value = GetString(var_n, "[0]=", "-", 16, 3)           // var_value = "[0]=C-[1]=22-[2]=38"
var_value = GetString(var_n, "[0]=", "-", 2, 2)            // var_value = "[0]=0b1100-[1]=0b100010-
                                                            [2]=0b111000"
```

* "[0]=" Support numeric format strings

* \u0D0A is Newline control character, not string value.

2.35 GetToken()

Retrieve a substring from input string, or the sub-array from the input byte[] array

Syntax 1

```
string GetToken(
    string,
    string,
    string,
    int,
    int
)
```

Parameters

`string`	Input string
`string`	Prefix. The leading element of the substring
`string`	Suffix. The trailing element of the substring
`int`	The number of the matched substring to retrieve

	>=1	Retrieve the n[th] matched substring
	-1	Retrieve the last matched substring
int	Remove the prefix and suffix or not	
	0	Reserve prefix and suffix (Default)
	1	Remove prefix and suffix

Return

string	String formed by part of the input string If the prefix and suffix are empty strings, returns the input string If the number of the matched substrings <=0 or larger than the number of the totol matached substrings, returns empty string

Syntax 2

```
string GetToken(
    string,
    string,
    string,
    int
)
```

Note

Similar to Syntax1 with reserving prefix and suffix.

GetToken(str,str,str,1) => **GetToken**(str,str,str,1,0)

Syntax 3

```
string GetToken(
    string,
    string,
    string
)
```

Note

Similar to Syntax1 with returning the first occurrence, and reserving prefix and suffix.

GetToken(str,str,str) => **GetToken**(str,str,str,1,0)

```
string var_n = "$abcd$1234$ABCD$"
var_value = GetToken(var_n, "", "", 0)          // var_value = "$abcd$1234$ABCD$"
var_value = GetToken(var_n, "$", "$")           // var_value = "$abcd$"
var_value = GetToken(var_n, "$", "$", 0)        // var_value = ""
var_value = GetToken(var_n, "$", "$", 1)        // var_value = "$abcd$"
var_value = GetToken(var_n, "$", "$", 2)        // var_value = "$ABCD$"
var_value = GetToken(var_n, "$", "$", 3)        // var_value = ""
```

var_value = **GetToken**(var_n, “$”, “$”, -1, 1) // var_value = “ABCD”

var_value = **GetToken**(var_n, “$”, “$”, 1, 1) // var_value = “abcd”

var_value = **GetToken**(var_n, “$”, “$”, 2, 1) // var_value = “ABCD”

var_value = **GetToken**(var_n, “$”, “”, 1) // var_value = “$abcd”

var_value = **GetToken**(var_n, “$”, “”, 2) // var_value = “$1234”

var_value = **GetToken**(var_n, “$”, “”, 3) // var_value = “$ABCD”

var_value = **GetToken**(var_n, “$”, “”, 4) // var_value = “$”

var_value = **GetToken**(var_n, “”, “$”, 1) // var_value = “$”

var_value = **GetToken**(var_n, “”, “$”, 2) // var_value = “abcd$”

var_value = **GetToken**(var_n, “”, “$”, 3) // var_value = “1234$”

var_value = **GetToken**(var_n, “”, “$”, 4) // var_value = “ABCD$”

string var_n = “$abcd$1234$ABCD$” + Ctrl(“\r\n”) + “56\r\n78$”

var_value = **GetToken**(var_n, “$”, Ctrl(“\r\n”), 1) // var_value = “$abcd$1234$ABCD$\u0D0A”

var_value = **GetToken**(var_n, “$”, newline, 2) // var_value = “”

var_value = **GetToken**(var_n, “$”, NewLine, 1, 1) // var_value = “abcd1234ABCD$”

// Remove prefix and suffix

var_value = **GetToken**(var_n, Ctrl(“\r\n”), “$”, 1) // var_value = “\u0D0A56\r\n78$”

var_value = **GetToken**(var_n, newline, “$”, 2) // var_value = “”

var_value = **GetToken**(var_n, NewLine, “$”, 1, 1) // var_value = “56\r\n78”

* \u0D0A is Newline control character, not string value.

Syntax 4

```
string GetToken(
    string,
    byte[],
    byte[],
    int,
    int
)
```

Parameters

`string`	Input string	
`byte[]`	Prefix. The leading element of the substring, byte[] type	
`byte[]`	Suffix. The trailing element of the substring, byte[] type	
`int`	The number of the matched substtring to retrieve	
	`>=1`	Retrieve the n^{th} matched substring
	`-1`	Retrieve the last matched substring
`int`	Remove prefix and suffix or not	
	`0`	Reserve prefix and suffix (Default)
	`1`	Remove prefix and suffix

Return

`string` String formed by part of the input string

If the prefix and suffix are empty strings, returns the input string

If the number of the matched substrings <=0 or larger than the number of the totol matached substrings, returns empty string

Syntax 5

```
string GetToken(
    string,
    byte[],
    byte[],
    int
)
```

Note

Similar to Syntax4 with reserving prefix and suffix

GetToken(str,byte[],byte[],1) => **GetToken**(str,byte[],byte[],1,0)

Syntax 6

```
string GetToken(
    string,
    byte[],
    byte[]
)
```

Note

Similar to Syntax 4 with the first occurrence and reserving prefix and suffix

GetToken(str,byte[],byte[]) => **GetToken**(str,byte[],byte[],1,0)

```
string var_n = "$abcd$1234$ABCD$"
byte[] var_bb0 = {}, var_bb1 = {0x24}  // 0x24 is $
var_value = GetToken(var_n, var_bb0, var_bb0, 0)      // var_value = "$abcd$1234$ABCD$"
var_value = GetToken(var_n, var_bb1, var_bb1)         // var_value = "$abcd$"
var_value = GetToken(var_n, var_bb1, var_bb1, 0)      // var_value = ""
var_value = GetToken(var_n, var_bb1, var_bb1, 1)      // var_value = "$abcd$"
var_value = GetToken(var_n, var_bb1, var_bb1, 2)      // var_value = "$ABCD$"
var_value = GetToken(var_n, var_bb1, var_bb1, 3)      // var_value = ""
var_value = GetToken(var_n, var_bb1, var_bb1, 1, 1)   // var_value = "abcd"
var_value = GetToken(var_n, var_bb1, var_bb1, 2, 1)   // var_value = "ABCD"
var_value = GetToken(var_n, var_bb1, var_bb0, 1)      // var_value = "$abcd"
var_value = GetToken(var_n, var_bb1, var_bb0, 2)      // var_value = "$1234"
```

```
var_value = GetToken(var_n, var_bb1, var_bb0, 3)     // var_value = "$ABCD"
var_value = GetToken(var_n, var_bb1, var_bb0, 4)     // var_value = "$"
var_value = GetToken(var_n, var_bb0, var_bb1, 1)     // var_value = "$"
var_value = GetToken(var_n, var_bb0, var_bb1, 2)     // var_value = "abcd$"
var_value = GetToken(var_n, var_bb0, var_bb1, 3)     // var_value = "1234$"
var_value = GetToken(var_n, var_bb0, var_bb1, 4)     // var_value = "ABCD$"

string var_n = "$abcd$1234$ABCD$" + Ctrl("\r\n") + "56\r\n78$"
byte[] var_bb0 = {0x0D,0x0A}, var_bb1 = {0x24}       // 0x24 is $ // 0x0D,0x0A is \u0D0A
var_value = GetToken(var_n, var_bb1, var_bb0, 1)     // var_value = "$abcd$1234$ABCD$\u0D0A"
var_value = GetToken(var_n, var_bb1, var_bb0, 2)     // var_value = ""
var_value = GetToken(var_n, var_bb1, var_bb0, 1, 1)  // var_value = "abcd$1234$ABCD$"
                                                     //  去除前置與後置
var_value = GetToken(var_n, var_bb0, var_bb1, 1)     // var_value = "\u0D0A56\r\n78$"
var_value = GetToken(var_n, var_bb0, var_bb1, 2)     // var_value = ""
var_value = GetToken(var_n, var_bb0, var_bb1, 1, 1)  // var_value = "56\r\n78"
```

* \u0D0A is the Newline control character, not the string content.

Syntax 7

```
byte[] GetToken(
    byte[],
    string,
    string,
    int,
    int
)
```

Parameters

`byte[]`	The input byte[]
`string`	Prefix. The leading element of the output byte[], byte[] type
`string`	Suffix. The trailing element of the output byte[], byte[] type
`int`	The number of the matched substring to retrieve
`>=1`	Retrieve the n^{th} matched substring
`-1`	Retrieve the last matched substring
`int`	Remove prefix and suffix or not
`0`	Reserve prefix and suffix (Default)
`1`	Remove prefix and suffix

Return

`byte[]`	The byte[] formed from part of the input byte[]
	If the prefix and suffix are empty, returns the input array
	If the number of the matched substrings <=0 or larger than the number of the totol matached

substrings, returns empty array

Syntax 8

```
byte[] GetToken(
    byte[],
    string,
    string,
    int
)
```

Note

Similar to Syntax7 with reserving prefix and suffix

GetToken(byte[],str,str,1) => **GetToken**(byte[],str,str,1,0)

Syntax 9

```
byte[] GetToken(
    byte[],
    string,
    string
)
```

Note

Similar to Syntax7 with returning the first occurrence, and reserving prefix and suffix.

GetToken(byte[],str,str) => **GetToken**(byte[],str,str,1,0)

```
string var_s ="$abcd$1234$ABCD$"
byte[] var_n = GetBytes(var_s)
var_value = GetToken(var_n, "", "", 0)
        // var_value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}
var_value = GetToken(var_n, "$", "$")       // var_value = {0x24,0x61,0x62,0x63,0x64,0x24}
var_value = GetToken(var_n, "$", "$", 0)    // var_value = {}
var_value = GetToken(var_n, "$", "$", 1)    // var_value = {0x24,0x61,0x62,0x63,0x64,0x24}
var_value = GetToken(var_n, "$", "$", 2)    // var_value = {0x24,0x41,0x42,0x43,0x44,0x24}
var_value = GetToken(var_n, "$", "$", 1, 1) // var_value = {0x61,0x62,0x63,0x64}
var_value = GetToken(var_n, "$", "$", 2, 1) // var_value = {0x41,0x42,0x43,0x44}
var_value = GetToken(var_n, "$", "", 1)     // var_value = {0x24,0x61,0x62,0x63,0x64}
var_value = GetToken(var_n, "$", "", 2)     // var_value = {0x24,0x31,0x32,0x33,0x34}
var_value = GetToken(var_n, "$", "", 3)     // var_value = {0x24,0x41,0x42,0x43,0x44}
var_value = GetToken(var_n, "$", "", 4)     // var_value = {0x24}
var_value = GetToken(var_n, "", "$", 1)     // var_value = {0x24}
var_value = GetToken(var_n, "", "$", 2)     // var_value = {0x61,0x62,0x63,0x64,0x24}
var_value = GetToken(var_n, "", "$", 3)     // var_value = {0x31,0x32,0x33,0x34,0x24}
```

```
var_value = GetToken(var_n, "", "$", 4)          // var_value = {0x41,0x42,0x43,0x44,0x24}
string var_var_s ="$abcd$1234$ABCD$" + Ctrl("\r\n") + "56\r\n78$"
byte[] var_n = GetBytes(var_s)
var_value = GetToken(var_n, "$", Ctrl("\r\n"), 1)
    // var_value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24,0x0D,0x0A}
var_value = GetToken(var_n, "$", Ctrl("\r\n"), 1, 1)
    // var_value = {0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}
    // Removing prefix and suffix
var_value = GetToken(var_n, Ctrl("\r\n"), "$", 1)
    // var_value = {0x0D,0x0A,0x35,0x36,0x5C,0x72,0x5C,0x6E,0x37,0x38,0x24}
var_value = GetToken(var_n, Ctrl("\r\n"), "$", 1, 1)
    // var_value = {0x35,0x36,0x5C,0x72,0x5C,0x6E,0x37,0x38}
```

Syntax 10

```
byte[] GetToken(
    byte[],
    byte[],
    byte[],
    int,
    int
)
```

Parameters

`byte[]`	The input byte[] array	
`byte[]`	Prefix. The leading element of the output byte[]	
`byte[]`	Suffix. The trailing element of the output byte[]	
`int`	The number of the matched substring to retrieve	
	`>=1`	Retrieve the n^{th} matched substring
	`-1`	Retrieve the last matched substring
`int`	Remove prefix and suffix or not	
	`0`	Reserve prefix and suffix (Default)
	`1`	Remove prefix and suffix

Return

`byte[]` The byte[] formed from part of the input byte[]

If the prefix and suffix are empty, returns the input array

If the number of the matched substrings <=0 or larger than the number of totol matached substrings, returns empty array

Syntax 11

```
byte[] GetToken(
```

```
    byte[],
    byte[],
    byte[],
    int
)
```

Note

Similar to Syntax10 with reserving the prefix and suffix

GetToken(byte[],byte[],byte[],1) => **GetToken**(byte[],byte[],byte[],1,0)

Syntax 12

```
byte[] GetToken(
    byte[],
    byte[],
    byte[]
)
```

Note

Similar to Syntax10 with returning the first occurrence, and reserving prefix and suffix.

GetToken(byte[],byte[],byte[]) => **GetToken**(byte[],byte[],byte[],1,0)

```
string var_s =“$abcd$1234$ABCD$”
byte[] var_n = GetBytes(var_s)
byte[] var_bb0 = {}, var_bb1 = {0x24}  // 0x24 is $
var_value = GetToken(var_n, var_bb0, var_bb0, 0)
            // var_value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}
var_value = GetToken(var_n, var_bb1, var_bb1)        // var_value = {0x24,0x61,0x62,0x63,0x64,0x24}
var_value = GetToken(var_n, var_bb1, var_bb1, 0)     // var_value = {}
var_value = GetToken(var_n, var_bb1, var_bb1, 1)     // var_value = {0x24,0x61,0x62,0x63,0x64,0x24}
var_value = GetToken(var_n, var_bb1, var_bb1, 2)     // var_value = {0x24,0x41,0x42,0x43,0x44,0x24}
var_value = GetToken(var_n, var_bb1, var_bb1, 1, 1)  // var_value = {0x61,0x62,0x63,0x64}
var_value = GetToken(var_n, var_bb1, var_bb1, 2, 1)  // var_value = {0x41,0x42,0x43,0x44}
var_value = GetToken(var_n, var_bb1, var_bb0, 1)     // var_value = {0x24,0x61,0x62,0x63,0x64}
var_value = GetToken(var_n, var_bb1, var_bb0, 2)     // var_value = {0x24,0x31,0x32,0x33,0x34}
var_value = GetToken(var_n, var_bb1, var_bb0, 3)     // var_value = {0x24,0x41,0x42,0x43,0x44}
var_value = GetToken(var_n, var_bb0, var_bb1, 1)     // var_value = {0x24}
var_value = GetToken(var_n, var_bb0, var_bb1, 2)     // var_value = {0x61,0x62,0x63,0x64,0x24}
var_value = GetToken(var_n, var_bb0, var_bb1, 3)     // var_value = {0x31,0x32,0x33,0x34,0x24}
string var_s =“$abcd$1234$ABCD$” + Ctrl(“\r\n”) + “56\r\n78$”
byte[] var_n = GetBytes(var_s)
```

```
byte[] var_bb0 = {0x0D,0x0A}, var_bb1 = {0x24}
var_value = GetToken(var_n, var_bb1, var_bb0, 1)
    // var_value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24,0x0D,0x0A}
var_value = GetToken(var_n, var_bb1, var_bb0, 1, 1)
    // var_value = {0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}
    // Remove prefix and suffix
var_value = GetToken(var_n, var_bb0, var_bb1, 1)
    // var_value = {0x0D,0x0A,0x35,0x36,0x5C,0x72,0x5C,0x6E,0x37,0x38,0x24}
var_value = GetToken(var_n, var_bb0, var_bb1, 1, 1)
    // var_value = {0x35,0x36,0x5C,0x72,0x5C,0x6E,0x37,0x38}
```

2.36 GetAllTokens()

Retrieve all the substrings from input string, which meets the given condition

Syntax 1

```
string[] GetAllTokens(
    string,
    string,
    string,
    int
)
```

Parameters

`string`	Input string
`string`	Prefix. The leading element of the substring
`string`	Suffix. The trailing element of the substring
`int`	Remove prefix and suffix or not
`0`	Reserve prefix and suffix (Default)
`1`	Remove prefix and suffix

Return

`string[]`	String array formed from retrieving all the substrings from input string If the prefix and suffix are empty, returns the input array

Syntax 2

```
string[] GetAllTokens(
    string,
    string,
    string
)
```

Note

Similar to Syntax1 with reserving prefix and suffix

GetAllTokens(str,str,str) => **GetAllTokens**(str,str,str,0)

```
string var_n = "$abcd$1234$ABCD$"
var_value = GetAllTokens(var_n, "", "")          // var_value = {"$abcd$1234$ABCD$"}
var_value = GetAllTokens(var_n, "$", "$")        // var_value = {"$abcd$", "$ABCD$"}
var_value = GetAllTokens(var_n, "$", "$", 1)     // var_value = {"abcd", "ABCD"}
var_value = GetAllTokens(var_n, "$", "")         // var_value = {"$abcd", "$1234", "$ABCD", "$"}
var_value = GetAllTokens(var_n, "", "$", 1)      // var_value = {"", "abcd", "1234", "ABCD"}
```

2.37 GetNow()

Get the current system time

Syntax 1

```
string GetNow(
    string
)
```

Parameters

`string` The date and time format strings defining the text representation of a date and time value. The definition of each specifier is listed below. The strings not included will remains the same.

`d`	The day of the month, from 1 through 31.
`dd`	The day of the month, from 01 through 31.
`ddd`	The abbreviated name of the day of the week.
`dddd`	The full name of the day of the week.
`f`	The tenths of a second in a date and time value.
`ff`	The hundredths of a second in a date and time value.
`fff`	The milliseconds in a date and time value.
`ffff`	The ten thousandths of a second in a date and time value.
`h`	The hour, using a 12-hour clock from 1 to 12.
`hh`	The hour, using a 12-hour clock from 01 to 12.
`H`	The hour, using a 24-hour clock from 0 to 23.
`HH`	The hour, using a 24-hour clock from 00 to 23.
`m`	The minute, from 0 through 59.
`mm`	The minute, from 00 through 59.
`M`	The month, from 1 through 12.
`MM`	The month, from 01 through 12.
`MMM`	The abbreviated name of the month.

MMMM	The full name of the month.
s	The second, from 0 through 59.
ss	The second, from 00 through 59.
t	The first character of the AM/PM designator.
tt	The AM/PM designator.
y	The year, from 0 to 99.
yy	The year, from 00 to 99.
yyyy	The year as a four-digit number.
/	The date separator.

Return

`string` Current date and time. If there is errors in format setting, the default format will be applied as MM/dd/yyyy HH:mm:ss.

Note

```
var_value = GetNow("MM/dd/yyyy HH:mm:ss")          // var_value = 08/15/2017 13:40:30
var_value = GetNow("yyyy/MM/dd HH:mm:ss.ffff")     // var_value = 2017/08/15 13:40:30.123
var_value = GetNow("yyyy-MM-dd hh:mm:ss tt")       // var_value = 2017-08-15 01:40:30 PM
```

Syntax 2

```
string GetNow(
)
```

Parameters

`void` No format defined. Default format "MM/dd/yyyy HH:mm:ss" will be applied

Return

`string` Current date and time.

Note

```
var_value = GetNow()          // var_value = 08/15/2017 13:40:30
```

2.38 GetNowStamp()

Get the total run time or difference in total run time

Syntax 1

```
int GetNowStamp(
)
```

Parameters

`void` No parameter

Return

`int` The total run time of the current project in ms. The upper limit is 2147483647 ms

< 0 Over flow, invalid total run time

Note

var_value = **GetNowStamp**() // var_value = 2147483647

… others …

var_value = **GetNowStamp**() // var_value = -1 // Over flow

Syntax 2

```
double GetNowStamp(
    bool
)
```

Parameters

bool Use double format to record project's total run time or not?

true Use double type, the upper limit is 9223372036854775807 ms

false Use int32 type, the upper limit is 2147483647 ms

Return

double The total run time of the current project

< 0 Over flow. Invalid total run time.

Note

var_value = **GetNowStamp**(false) // var_value = 2147483647

… others …

var_value = **GetNowStamp**(false) // var_value = -1 // Over flow

var_value = **GetNowStamp**(true) // var_value = 3147483647

Syntax 3

```
int GetNowStamp(
    int
)
```

Parameters

int Previous recorded run time in ms

Return

int The difference between the current run time and the input run time in ms.

Run time difference = current run time – input run time

< 0 Invalid run time difference, caused by input run time larger than current run time, or over flow.

Note

var_value = **GetNowStamp**() // var_value = 2147483546

… others … (After 100ms)

diff = **GetNowStamp**(var_value) // diff = 100

```
... others ... (After 200ms)
diff = GetNowStamp(var_value) // diff = -1          // Value is over 2147483647
```

Syntax 4

```
double GetNowStamp(
    double
)
```

Parameters

`double` Previous recorded run time in ms

Return

`double` The difference between the current run time and the input run time in ms.

Run time difference = current run time – input run time

`< 0` Invalid run time difference, caused by input run time larger than current run time, or over flow.

Note

```
var_value = GetNowStamp()          // var_value = 2147483546
... others ... (After 100ms)
diff = GetNowStamp(var_value) // diff = 100
... others ... (After 200ms)
diff = GetNowStamp(var_value) // diff = 200
```

Syntax 5

```
bool GetNowStamp(
    int,
    int
)
```

Parameters

`int` Previous recorded run time in ms

`int` The expected run time difference

Return

`bool` The time difference between current run time and input run time is larger than the expected run time difference or not.

`true` (Current run time – input run time) >= expected run time

`Or` Time difference smaller than zero or over flow

`false` (Current run time – input run time) < expected run time

Note

```
var_value = GetNowStamp()          // var_value = 41730494
... others ... (After 60ms)
```

```
var_flag = GetNowStamp(var_value, 100)  // diff = 60        // var_flag = false
... others ... (After 60ms)
var_flag = GetNowStamp(var_value, 100)  // diff = 120       // var_flag = true
```

Syntax 6

```
bool GetNowStamp(
    double,
    double
)
```

Parameters

`double` Previous recorded run time in ms

`double` The expected run time difference

Return

`bool` The time difference between current run time and input run time is larger than the expected run time difference or not.

`true` (Current run time – input run time) >= expected run time

`Or` Time difference smaller than zero or over flow

`false` (Current run time – input run time) < expected run time

Note

```
var_value = GetNowStamp()                // var_value = 41730494
... others ... (After 60ms)
var_flag = GetNowStamp(var_value, 100)  // diff = 60        // var_flag = false
... others ... (After 60ms)
var_flag = GetNowStamp(var_value, 100)  // diff = 120       // var_flag = true
```

2.39 Length()

Acquire the number of byte of input data, length of string or length of array (number of elements in array)

Syntax 1

```
int Length(
    ?
)
```

Parameters

`?` The input data. The available data types are integer, floating-point, boolean, string, or array.

Return

`int`	Length of data For input as integer, floating-point number, and boolean, returns the number of byte. For input as string, returns the length of string. For input as array, returns the number of element in array

Note

? byte var_n = 100

var_value = **Length**(var_n) // var_value = 1

var_value = **Length**(100) // var_value = 1

? int var_n = 400

var_value = **Length**(var_n) // var_value = 4

var_value = **Length**(400) // var_value = 4

? float var_n = 1.234

var_value = **Length**(var_n) // var_value = 4

var_value = **Length**(1.234) // var_value = 4

? double var_n = 1.234

var_value = **Length**(var_n) // var_value = 8

var_value = **Length**(1.234) // var_value = 4

// float // Numbers would be stored as the smaller data type first.

? bool var_n = true

var_value = **Length**(var_n) // var_value = 1

var_value = **Length**(false) // var_value = 1

? string var_n = “A””BC”

var_value = **Length**(var_n) // var_value = 4

// The string is A”BC. Two double quotation marks represent “ in string

var_value = **Length**(“”) // var_value = 0

var_value = **Length**(“123”) // var_value = 3

var_value = **Length**(empty) // var_value = 0

? byte[] var_n = {100, 200, 30}

var_value = **Length**(var_n) // var_value = 3

? int[] var_n = {}

var_value = **Length**(var_n) // var_value = 0

var_n = {400, 500, 600}

var_value = **Length**(var_n) // var_value = 3

? float[] var_n = {1.234}

var_value = **Length**(var_n) // var_value = 1

? double[] var_n = {1.234, 200, -100, +300}

var_value = **Length**(var_n) // var_value = 4

? bool[] var_n = {true, false, true, true, true, true, false}

var_value = **Length**(var_n) // var_value = 7

? string[] var_n = {“A””BC”, “123”, “456”, “ABC”}

var_value = **Length**(var_n) // var_value = 4

2.40 Ctrl()

Change the integer or string to control characters

Syntax 1

```
string Ctrl(
    int
)
```

Parameters

`int` The input integer, which follows the Big Endian format. 4 characters could be transformed at most. 0x00 will not be transformed.

Return

`string` The string formed by input integer (contains the control character)

Note

var_b = 0x0D0A

var_value = **Ctrl**(var_b) // var_value = \r\n

var_value = **Ctrl**(0x0D0A) // var_value = \r\n

var_value = **Ctrl**(0x0D000A09) // var_value = \r\n\t // 0x00 will not be transformed

var_value = **Ctrl**(0x0D300A09) // var_value = \r0\n\t // 0x30 is transformed to 0

var_value = **Ctrl**(0x00) // var_value = “” // empty string does not equal to NULL. For NULL, the code is Ctrl(“\0”)

Syntax 2

```
string Ctrl(
    string
)
```

Parameters

`string` Input string. The following rules will be applied. For string not on the list, it will remain the same.

`\0`	0x00 null
`\a`	0x07 bell
`\b`	0x08 backspace

\t	0x09 horizontal tab
\r	0x0D carriage return
\v	0x0B vertical tab
\f	0x0C form feed
\n	0x0A line feed

Return

`string` The string formed by input integer (contains the control character)

Note

var_b = "\r\n"

var_value = **Ctrl**(var_b) // var_value = \r\n

var_value = **Ctrl**("\r\n") // var_value = \r\n

var_value = **Ctrl**("\r\n\t") // var_value = \r\n\t

var_value = **Ctrl**("\r0\n\t") // var_value = \r0\n\t

var_value = **Ctrl**("\0") // var_value = \0 // NULL

Syntax 3

```
string Ctrl(
    byte[]
)
```

Parameters

`byte[]` The input byte array, the transfer will start from index [0] to the end of the array. (0x00 will be transferred also)

Return

`string` The string formed by input integer (contains the control character)

Note

byte[] var_bb1 = {0xFF,0x55,0x31,0x32,0x33,0x00,0x35,0x36,0x0D,0x0A}

var_value = **Ctrl**(var_bb1) // var_value = U123 56\r\n

byte[] var_bb2 = {}

var_value = **Ctrl**(var_bb2) // var_value = ""

2.41 XOR8()

Utilize XOR 8 bits algorithm to computes the checksum

Syntax 1

```
byte XOR8(
    byte[],
```

```
    int,
    int
)
```

Parameters

`byte[]`	The input byte array
`int`	The starting index
	0..(array size-1) Valid
	<0 Invalid. Returns the initial value 0
	>=array size Invalid. Returns the initial value 0
`int`	The number of elements to be computed.
	If the number of elements <0, the calculation ends at the last element of the array
	If the sum of starting index and number of element exceeds the array size, the calculation ends at the last element of the array.

Return

`byte`	Checksum.

Note

```
byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
var_value = XOR8(var_bb1,0,Length(var_bb1))   // var_value = 0x6F
var_value = XOR8(var_bb1,0,-1)                // var_value = 0x6F
var_value = XOR8(var_bb1,1,-1)                // var_value = 0x7F
var_value = XOR8(var_bb1,-1,-1)               // var_value = 0
```

Syntax 2

```
byte XOR8(
    byte[],
    int
)
```

Note

Similar to Syntax1 with computing to the last element of the array

XOR8(byte[], int) => **XOR8**(byte[], int, Length(byte[]))

Syntax 3

```
byte XOR8(
    byte[]
)
```

Note

Similar to Syntax1 with computing all the elements of the array

XOR8(byte[]) => **XOR8**(byte[], 0, Length(byte[]))

byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

var_value = **XOR8**(var_bb1,0,Length(var_bb1)) // var_value = 0x6F

var_value = **XOR8**(var_bb1,0) // var_value = 0x6F

var_value = **XOR8**(var_bb1) // var_value = 0x6F

var_bb1 = **Byte_Concat**(var_bb1, **XOR**(var_bb1))

// var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x6F}

2.42 SUM8()

Utilize SUM 8 bits algorithm to computes the checksum

Syntax 1

```
byte SUM8(
    byte[],
    int,
    int
)
```

Parameters

`byte[]` The input byte array

`int` The starting index

0..array size-1 Valid

<0 Invalid. Returns the initial value 0

>=array size Invalid. Returns the initial value 0

`int` The number of elements to be computed.

If the number of elements <0, the calculation ends at the last element of the array

If the sum of starting index and number of element exceeds the array size, the calculation ends at the last element of the array.

Return

`byte` Checksum.

Note

byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

var_value = **SUM8**(var_bb1,0,Length(var_bb1)) // var_value = 0x6D

var_value = **SUM8**(var_bb1,0,-1) // var_value = 0x6D

var_value = **SUM8**(var_bb1,1,-1) // var_value = 0x5D

var_value = **SUM8**(var_bb1,-1,-1) // var_value = 0

Syntax 2

```
byte SUM8(
    byte[],
    int
)
```

Note

Similar to Syntax1 with computing to the last element of the array

SUM8(byte[], int) => **SUM8**(byte[], int, Length(byte[]))

Syntax 3

```
byte SUM8(
    byte[]
)
```

Note

Similar to Syntax1 with computing all the elements of the array

SUM8(byte[]) => **SUM8**(byte[], 0, Length(byte[]))

```
byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
var_value = SUM8(var_bb1,0,Length(var_bb1))        // var_value = 0x6D
var_value = SUM8(var_bb1,0)                        // var_value = 0x6D
var_value = SUM8(var_bb1)                          // var_value = 0x6D
var_bb1 = Byte_Concat(var_bb1, SUM8(var_bb1))      // var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF,
0xFF, 0x6D}
```

2.43 SUM16()

Utilize SUM 16 bits algorithm to computes the checksum

Syntax 1

```
byte[] SUM16(
    byte[],
    int,
    int
)
```

Parameters

byte[]	The input byte array	
int	The starting index	
	0..array size-1	Valid
	<0	Invalid. Returns the initial value 0

>=array size Invalid. Returns the initial value 0

`int` The number of elements to be computed.

If the number of elements <0, the calculation ends at the last element of the array

If the sum of starting index and number of element exceeds the array size, the calculation ends at the last element of the array.

Return

`byte[]` Checksum. The length is 16bits 2 bytes (The Checksum follows Big Endian)

Note

```
byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
var_value = SUM16(var_bb1,0,Length(var_bb1)) // var_value = {0x04, 0x6D}
var_value = SUM16(var_bb1,0,-1)              // var_value = {0x04, 0x6D}
var_value = SUM16(var_bb1,1,-1)              // var_value = {0x04, 0x5D}
var_value = SUM16(var_bb1,-1,-1)             // var_value = {0x00, 0x00}
```

Syntax 2

```
byte[] SUM16(
    byte[],
    int
)
```

Note

Similar to Syntax1 with computing to the last element of the array

SUM16(byte[], int) => **SUM16**(byte[], int, Length(byte[]))

Syntax 3

```
byte[] SUM16(
    byte[]
)
```

Note

Similar to Syntax1 with computing all the elements of the array

SUM16(byte[]) => **SUM16**(byte[], 0, Length(byte[]))

```
byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
var_value = SUM16(var_bb1,0,Length(var_bb1))      // var_value = {0x04, 0x6D}
var_value = SUM16(var_bb1,0)                      // var_value = {0x04, 0x6D}
var_value = SUM16(var_bb1)                        // var_value = {0x04, 0x6D}
var_bb1 = Byte_Concat(var_bb1, SUM16(var_bb1))    // var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF,
0x04, 0x6D}
```

2.44 SUM32()

Utilize SUM 32 bits algorithm to computes the checksum

Syntax 1

```
byte[] SUM32(
    byte[],
    int,
    int
)
```

Parameters

byte[]	The input byte array	
int	The starting index	
	0..array size-1	Valid
	<0	Invalid. Returns the initial value 0
	>=array size	Invalid. Returns the initial value 0
int	The number of elements to be computed.	
	If the number of elements <0, the calculation ends at the last element of the array	
	If the sum of starting index and number of element exceeds the array size, the calculation ends at the last element of the array.	

Return

byte[]	Checksum. The length is 32bits 4 bytes (The Checksum follows Big Endian)

Note

```
byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
var_value = SUM32(var_bb1,0,Length(var_bb1)) // var_value = {0x00, 0x00, 0x04, 0x6D}
var_value = SUM32(var_bb1,0,-1)              // var_value = {0x00, 0x00, 0x04, 0x6D}
var_value = SUM32(var_bb1,1,-1)              // var_value = {0x00, 0x00, 0x04, 0x5D}
var_value = SUM32(var_bb1,-1,-1)             // var_value = {0x00, 0x00, 0x00, 0x00}
```

Syntax 2

```
byte[] SUM32(
    byte[],
    int
)
```

Note

Similar to Syntax1 with computing to the last element of the array

SUM32(byte[], int) => **SUM32**(byte[], int, Length(byte[]))

Syntax 3

```
byte[] SUM32(
    byte[]
)
```

Note

Similar to Syntax1 with computing all the elements of the array

SUM32(byte[]) => **SUM32**(byte[], 0, Length(byte[]))

byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

var_value = **SUM32**(var_bb1,0,Length(var_bb1)) // var_value = {0x00, 0x00, 0x04, 0x6D}

var_value = **SUM32**(var_bb1,0) // var_value = {0x00, 0x00, 0x04, 0x6D}

var_value = **SUM32**(var_bb1) // var_value = {0x00, 0x00, 0x04, 0x6D}

var_bb1 = **Byte_Concat**(var_bb1, **SUM32**(var_bb1))

// var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x04, 0x6D}

2.45 CRC16()

Utilize CRC 16 bits algorithm to computes the checksum

Syntax 1

```
byte[] CRC16(
    int,
    byte[],
    int,
    int
)
```

Parameters

`int` CRC16 algorithm (Reference https://www.lammertbies.nl/comm/info/crc-calculation.html)

0	CRC16	// initial value 0x0000 // Polynomial 0xA001
1	CRC16 (Modbus)	// initial value 0xFFFF // Polynomial 0xA001
2	CRC16 (Sick)	// initial value 0x0000 // Polynomial 0x8005
3	CRC16-CCITT (0x1D0F)	// initial value 0x1D0F // Polynomial 0x1021
4	CRC16-CCITT (0xFFFF)	// initial value 0xFFFF // Polynomial 0x1021
5	CRC16-CCITT (XModem)	// initial value 0x0000 // Polynomial 0x1021
6	CRC16-CCITT (Kermit)	// initial value 0x0000 // Polynomial 0x8408
7	CRC16 Schunk Gripper	// initial value 0xFFFF // Polynomial 0x1021

`byte[]` The input byte array

`int` The starting index

0..array size-1	Valid
<0	Invalid. Returns the initial value
>=array size	Invalid. Returns the initial value

`int` The number of elements to be computed.

If the number of elements <0, the calculation ends at the last element of the array

If the sum of starting index and number of element exceeds the array size, the calculation ends at the last element of the array.

Return

`byte[]` Checksum. The length is 16bits 2 bytes (The checksum follows Big Endian)

Note

```
byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
var_value = CRC16(0, var_bb1,0,Length(var_bb1))        // var_value = {0x2D, 0xD4}
var_value = CRC16(0, var_bb1,0,-1)                     // var_value = {0x2D, 0xD4}
var_value = CRC16(0, var_bb1,1,-1)                     // var_value = {0xEC, 0xC5}
var_value = CRC16(0, var_bb1,-1,-1)                    // var_value = {0x00, 0x00}
var_value = CRC16(3, var_bb1,0,Length(var_bb1))        // var_value = {0x42, 0x12}
var_value = CRC16(4, var_bb1,0,Length(var_bb1))        // var_value = {0xAB, 0xAE}
```

Syntax 2

```
byte[] CRC16(
    int,
    byte[],
    int
)
```

Note

Similar to Syntax1 with computing to the last element of the array

CRC16(int, byte[], int) => **CRC16**(int, byte[], int, Length(byte[]))

Syntax 3

```
byte[] CRC16(
    int,
    byte[]
)
```

Note

Similar to Syntax1 with computing all the elements of the array

CRC16(int, byte[]) => **CRC16**(int, byte[], 0, Length(byte[]))

byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

```
var_value = CRC16(0, var_bb1,0,Length(var_bb1))          // var_value = {0x2D, 0xD4}
var_value = CRC16(0, var_bb1,0)                          // var_value = {0x2D, 0xD4}
var_value = CRC16(0, var_bb1)                            // var_value = {0x2D, 0xD4}
var_bb1 = Byte_Concat(var_bb1, CRC16(0, var_bb1))
        // var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x2D, 0xD4}
```

Syntax 4

```
byte[] CRC16(
    byte[],
    int,
    int
)
```

Note

Similar to Syntax1 with CRC16 algorithm as 0 CRC16

CRC16(byte[], int, int) => **CRC16**(0, byte[], int, int)

Syntax 5

```
byte[] CRC16(
    byte[],
    int
)
```

Note

Similar to Syntax1 with CRC16 algorithm as 0 CRC16 and computing to the last element of the array

CRC16(byte[], int) => **CRC16**(0, byte[], int, Length(byte[]))

Syntax 6

```
byte[] CRC16(
    byte[]
)
```

Note

Similar to Syntax1 with CRC16 algorithm as 0 CRC16 and computing all the elements of the array

CRC16(byte[]) => **CRC16**(0, byte[], 0, Length(byte[]))

2.46 CRC32()

Utilize CRC 32 bits algorithm to computes the checksum

Syntax 1

```
byte[] CRC32(
    byte[],
    int,
    int
)
```

Parameters

byte[]	The input byte array	
int	The starting index	
	0..array size-1	Valid
	<0	Invalid. Returns the initial value 0
	>=array size	Invalid. Returns the initial value 0
int	The number of elements to be computed.	
	If the number of elements <0, the calculation ends at the last element of the array	
	If the sum of starting index and number of element exceeds the array size, the calculation ends at the last element of the array.	

Return

byte[]	Checksum. The checksum length is 32bits 4 bytes (The checksum follows Big Endian)

Note

byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

```
var_value = CRC32(var_bb1,0,Length(var_bb1))  // var_value = {0x43, 0xD5, 0xB9, 0xF8}
var_value = CRC32(var_bb1,0,-1)               // var_value = {0x43, 0xD5, 0xB9, 0xF8}
var_value = CRC32(var_bb1,1,-1)               // var_value = {0x08, 0xA5, 0x5B, 0xEB}
var_value = CRC32(var_bb1,-1,-1)              // var_value = {0x00, 0x00, 0x00, 0x00}
```

Syntax 2

```
byte[] CRC32(
    byte[],
    int
)
```

Note

Similar to Syntax1 with computing to the last element of the array

CRC32(byte[], int) => **CRC32**(byte[], int, Length(byte[]))

Syntax 3

```
byte[] CRC32(
```

```
    byte[]
)
```

Note

Similar to Syntax1 with computing all the elements of the array

CRC32(byte[]) => **CRC32**(byte[], 0, Length(byte[]))

byte[] var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

var_value = **CRC32**(var_bb1,0,Length(var_bb1)) // var_value = {0x43, 0xD5, 0xB9, 0xF8}

var_value = **CRC32**(var_bb1,0) // var_value = {0x43, 0xD5, 0xB9, 0xF8}

var_value = **CRC32**(var_bb1) // var_value = {0x43, 0xD5, 0xB9, 0xF8}

var_bb1 = **Byte_Concat**(var_bb1, **CRC32**(var_bb1))

// var_bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x43, 0xD5, 0xB9, 0xF8}

2.47 ListenPacket()

Pack the string contents as the compatible protocol for the Listen Node (External Script Control Mode)

Syntax 1

```
string ListenPacket(
    string,
    string
)
```

Parameters

`string` User defined Header. For empty string, Default string “TMSCT” will be applied

`string` The data section in Listen Node communication format

Return

`string` Packed data (Including header, data length and check sum)

Note

string var_data1 = “1,var_i++”

string var_data2 = “Hello World”

var_value = **ListenPacket**(“TMSCT”, var_data1) // $TMSCT,9,1,var_i++,*06\r\n

var_value = **ListenPacket**(“”, var_data2)

// $TMSCT,11,Hello World,*51\r\n // Error for TMSCT

var_value = **ListenPacket**(“”, “2,Techman Robot”) // $TMSCT,15,2,Techman Robot,*57\r\n

var_value = **ListenPacket**(“TMSTA”, var_data2)

// $TMSTA,11,Hello World,*53\r\n // Error for TMSTA

var_value = **ListenPacket**(“TMSTA”, “00”) // $TMSTA,2,00,*41\r\n

Syntax 2

```
string ListenPacket(
    string
)
```

Parameters

`string` The data section in Listen Node communication format (With TMSCT header)

Return

`string` Packed data (Including header, data length and check sum)

Note

```
string var_data1 = "1,var_counter++"          // ScriptID, ScriptLanguage
var_value = ListenPacket(var_data1)           // $TMSCT,15,1,var_counter++,*26\r\n
```

2.48 ListenSend()

Send TMSTA, the communication protocol of Listen node, to the client devices connected to the Listen Server currently.

Syntax1

```
int ListenSend(
    string,
    int,
    ?
)
```

Parameters

`string` Target IP filtering such as 127.0.0.1 meaning to send to all client devices connecting from 127.0.0.1.

`int` TMSTA SubCmd numbering for sending self-defined data message only 90 .. 99

`?` The value to send. Available types: byte, int, float, double, bool, and string.

Numeric values will be conversed in Little Endian, and string values will be converse in UTF8.

Return

`int` Return the result

0	sent successfully
-1	error. Listen Server is not starting.
-2	error. SubCmd must be between 90 and 99.

Syntax2

```
int ListenSend(
    int,
    ?
)
```

Parameters

int	TMSTA SubCmd numbering for sending self-defined data message only	90 .. 99
?	The value to send. Available types: byte, int, float, double, bool, and string.	
	Numeric values will be conversed in Little Endian, and string values will be converse in UTF8.	

Return

int	Return the result	
	0	sent successfully
	-1	error. Listen Server is not starting.
	-2	error. SubCmd must be between 90 and 99.

Note

No target IP filtering will result in sending data messages to all connected client devices.

Note

```
string var_ip = “127.0.0.1”

byte var_b = 100
var_value = ListenSend(var_ip, 10, var_b)
    // send 0x64 to ipfilter “127.0.0.1”        // var_value = -2  // SubCmd must be between 90 and 99.
var_value = ListenSend(var_ip, 90, var_b)
    // send 0x64 to ipfilter “127.0.0.1”        // var_value = -1  // Supposedly Listen Server is not starting.
var_value = ListenSend(var_ip, 90, var_b)
    // send 0x64 to ipfilter “127.0.0.1”        // var_value = 0          // sent successfully
    // IP filtering 127.0.0.1 and send to the devices connected to Listen Server via the IP.
    // $TMSTA,4,90,d,*06                  // The value of 100 is conversed to 0x64.
var_value = ListenSend(var_ip, 90, 123456)
    // send 0x40 0xE2 0x01 0x00 to ipfilter “127.0.0.1”
    // $TMSTA,7,90,@, ,*C2
    // The value of 123456 is conversed to 0x40 0xE2 0x01 0x00 (int, Little Endian)
var_value = ListenSend(90, “123.456”)
    // send 0x31 0x32 0x33 0x2E 0x34 0x35 0x36
    // No target IP filtering will result in sending data messages to all connected client devices.
```

```
        // $TMSTA,10,90,123.456,*7E
        // The value of “123.456” is conversed to 0x31 0x32 0x33 0x2E 0x34 0x35 0x36 (string, UTF8).
    byte[] var_bb = {100, 200}
    var_value = ListenSend(90, var_bb)
        // send 0x64 0xC8
        // $TMSTA,5,90,d,*CF        // The value of {100, 200} is conversed to 0x64 0xC8
    string[] var_ss = {“T”, “M”, “達明機器人” }
    var_value = ListenSend(90, var_ss)
        // send 0x54 0x4D 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA
        // $TMSTA,20,90,TM達明機器人,*A1
```

2.49 VarSync()

Send the Variable object to TMManager (Robot Management System)

* When performing this function, the flow will not go on until the object is sent out successfully or the maximum retry times is reached.

Syntax 1

```
int VarSync(
    int,
    int,
    ?
)
```

Parameters

`int` The maximum times to retry

`<= 0` Keep retrying as error occurred.

`int` The time duration between two retries (millisecond)

`< 0` Invalid time duration. The default value, 1000ms, will be applied

`?` The string or string array. The name of variables to be sent.

Multiple items can be listed. If there are indefinite variables, they will be not be sent; other definite variables will be sent.

* The item is the name of the variable, not what the variable equals to such that var_i goes with “var_i”.

* If the variable is listed, the value of the variable will be used to send the matched object.

Return

`int` Sending times

> 0	Send success. The return value returns the sending times
0	Send failed
-1	TM Manager function is not enabled
-9	Invalid Parameters

Note

string var_s = “ABC”

string var_s1 = “var_s”

string[] var_ss = {“ABC”, “var_s”, “var_s1”}

var_value = **VarSync**(1, 1000, “var_s”) // Send var_s variable object

var_value = **VarSync**(2, 2000, var_s) // Send ABC variable object (Because the value of var_s is “ABC”)

var_value = **VarSync**(3, 2000, var_ss) // Send ABC, var_s, var_s1 variable object (From the value of var_ss string array)

var_value = **VarSync**(3, 2000, “var_ss”) // Send var_ss variable object

var_value = **VarSync**(4, 2000, “var_ss”, “var_s1”, “ABC”) // Send var_ss, var_s1, ABC variable object

Syntax 2

```
int VarSync(
    int,
    ?
)
```

Note

Same as Syntax 1 with the time between two retries defaults to 1000 ms.

VarSync(int, ?) => **VarSync**(int, 1000, ?)

Syntax 3

```
int VarSync(
    ?
)
```

Note

Same as Syntax 1 with the time between two retries defaults to 1000 ms without limit of times to retry

VarSync(?) => **VarSync**(0, 1000, ?)

3. Math Functions

3.1 abs()

Return the absolute value of the designate number

Syntax 1

```
int abs(
    int
)
```

Parameter

`int` Input number in integer

Return

`int` Return the absolute value of the input number in integer

Note

```
int var_i = 10
var_value = abs(var_i)      // 10
var_i = -10
var_value = abs(var_i)      // 10
```

Syntax 2

```
float abs(
    float
)
```

Parameter

`float` Input number in float

Return

`float` Return the absolute value of the input number in float

Note

```
float var_f = 10.1
var_value = abs(var_f)      // 10.1
var_f = -10.1
var_value = abs(var_f)      // 10.1
```

Syntax 3

```
double abs(
    double
```

)

Parameter

`double` Input number in double

Return

`double` Return the absolute value of the input number in double

Note

```
double var_d = 10.8
var_value = abs(var_d)      // 10.8
var_d = -10.8
var_value = abs(var_d)      // 10.8
```

3.2 pow()

Return the power of the designate base and exponent

Syntax 1

```
int pow(
    int,
    double
)
```

Parameter

`int` Input base in integer

`double` Exponent

Return

`int` Return the power in integer

Syntax 2

```
float pow(
    float,
    double
)
```

Parameter

`float` Input base in float

`double` Exponent

Return

`float` Return the power in float

Syntax 3

```
double pow(
    double,
    double
)
```

Parameter

`double` Input base in double

`double` Exponent

Return

`double` Return the power in double

Note

? `int` var_b = 100

```
var_value = pow(var_b, 2)  // 10000
var_value = pow(var_b, -2)// 0          // 0.0001, but int type
var_value = pow(var_b, 0.1)      // 1          // 1.5848931924611136, but int type
var_value = pow(var_b, 2.1)      // 15848     // 15848.931924611141, but int type
var_value = pow(var_b, -2.1)     // 0          // 0.000063095734448019293, but int type
```

? `float` var_b = -100

```
var_value = pow(var_b, 2)  // 10000
var_value = pow(var_b, -2)// 0.0001
var_value = pow(var_b, 0.2)      // Error // NaN
var_value = pow(var_b, 2.2)      // Error // NaN
var_value = pow(var_b, -2.2)     // Error // NaN
```

? `double` var_b = 100

```
var_value = pow(var_b, 2)        // 10000
var_value = pow(var_b, -2)// 0.0001
var_value = pow(var_b, 0.31)     // 4.16869383470335
var_value = pow(var_b, 2.31)     // 41686.9383470336
var_value = pow(var_b, -2.31)    // 0.0000239883291901949
```

3.3 sqrt()

Return the square root of the designate number

Syntax 1

```
float sqrt(
    float
)
```

Parameter

`float` Input number in float

Return

`float` Return the square root in float

Syntax 2

```
double sqrt(
    double
)
```

Parameter

`double` Input number in double

Return

`double` Return the square root in double

Note

```
var_value = sqrt(100)       // 10
var_value = sqrt(100.1234)      // 10.005
var_value = sqrt(0.1234)        // 0.3162278
var_value = sqrt(-100)          // Error // NaN
var_value = sqrt(-100.1234)     // Error // NaN
var_value = sqrt(-0.1234)       // Error // NaN
```

3.4 ceil()

Return a number rounded upward to its nearest integer.

Syntax 1

```
float ceil(
    float
)
```

Parameter

`float` input number in float

Return

`float` Return a number in float rounded upward to its nearest integer

Syntax 2

```
double ceil(
    double
)
```

Parameter

`double` input number in double

Return

`double` Return a number in double rounded upward to its nearest integer

Note

```
var_value = ceil(100)        // 100
var_value = ceil(100.1234) // 101
var_value = ceil(0.1234)          // 1
var_value = ceil(-100)            // -100
var_value = ceil(-100.1234)       // -100
var_value = ceil(-0.1234)         // 0
```

3.5 floor()

Return a number rounded downward to its nearest integer.

Syntax 1

```
float floor(
    float
)
```

Parameter

`float` input number in float

Return

`float` Return a number in float rounded downward to its nearest integer

Syntax 2

```
double floor(
```

```
    double
)
```

Parameter

`double` input number in double

Return

`double` Return a number in double rounded downward to its nearest integer

Note

```
var_value = floor(100)          // 100
var_value = floor(100.1234)     // 100
var_value = floor(0.1234)       // 0
var_value = floor(-100)         // -100
var_value = floor(-100.1234)    // -101
var_value = floor(-0.1234) // -1
```

3.6 round()

Return a number rounded to its nearest integer.

Syntax 1

```
float round(
    float,
    int
)
```

Parameter

`float`	input number in float	
`int`	digits after the returned decimal point (0 by default meaning the number is rounded to integer)	
	0 .. 15	valid values
	< 0	value invalid, will use 0 by default
	> 15	value invalid, will use 0 by default

Return

`float` Return a number in float rounded to its nearest integer.

Syntax 2

```
float round(
    float
)
```

Note

Same as syntax 1. Obtain 0 digit after the decimal point by default.

round(float) => **round**(float, 0)

Syntax 3

```
double round(
    double,
    int
)
```

Parameter

`double`	input number in double	
`int`	digits after the returned decimal point (0 by default meaning the number is rounded to integer)	
	0 .. 15	valid values
	< 0	value invalid, will use 0 by default
	> 15	value invalid, will use 0 by default

Return

`double`	Return a number in double rounded to its nearest integer.

Syntax 4

```
double round(
    double
)
```

Note

Same as syntax 3. Obtain 0 digit after the decimal point by default.

round(double) => **round**(double, 0)

```
var_value = round(100)                  // 100
var_value = round(100.456)              // 100
var_value = round(0.567)                // 1
var_value = round(-100)                 // -100
var_value = round(-100.456)             // -100
var_value = round(-0.567)          // -1
var_value = round(100.345, 1)           // 100.3
var_value = round(100.345, 2)           // 100.35
var_value = round(-100.345, 1)          // -100.3
var_value = round(-100.345, 2)          // -100.35
var_value = round(-100.345, 16)// -100
```

3.7 random()

Return a random number in float between 0 and 1 or in integer between the lower bound and the upper bound.

Syntax 1

```
float random(
)
```

Parameter

`void` No input value required.

Return

`float` Return a random number in float between 0 and 1.

Note

```
var_value = random()        // 0.9473762
var_value = random()        // 0.7764986
var_value = random()        // 0.9911129
```

Syntax 2

```
int random(
    int
)
```

Parameter

`int` The upper bound of the random number

Return

`int` Return a random number in integer between 0 and the upper bound

Note

```
var_value = random(10)      // 8
var_value = random(10)      // 1
var_value = random(10)      // 5
var_value = random(-10)     // 0   // The value of the upper bound must be larger than 0.
```

Syntax 3

```
int random(
    int,
    int
)
```

Parameter

`int` The lower bound of the random number

`int` The upper bound of the random number must be larger than the lower bound, or it will return the value of the lower bound in integer.

Return

`int` Return a random number in integer between the lower bound and the upper bound.

Note

```
var_value = random(5, 10) // 8
var_value = random(5, 10) // 8
var_value = random(5, 10) // 6
var_value = random(5, -1)  // 5   // The upper bound is smaller than the lower bound. Returned the value of the lower bound in integer.
```

3.8 d2r()

Convert the value of degree to radian

Syntax 1

```
float d2r(
    float
)
```

Parameter

`float` Input the value of degree in float

Return

`float` Return the value of radian in float

Syntax 2

```
double d2r(
    double
)
```

Parameter

`double` Input the value of degree in double

Return

`double` Return the value of radian in double

Note

```
var_value = d2r(1)          // 0.01745329
```

3.9 r2d()

Convert the value of degree to radian to degree

Syntax 1

```
float r2d(
    float
)
```

Parameter

`float` Input the value of radian in float

Return

`float` Return the value of degree in float

Syntax 2

```
double r2d(
    double
)
```

Parameter

`double` Input the value of radian in double

Return

`double` Return the value of degree in double

Note

```
var_value = r2d(1)          // 57.29578
```

3.10 sin()

Return the sine of the input value of degree

Syntax 1

```
float sin(
    float
)
```

Parameter

`float` Input the value of degree in float

Return

`float` Return the sine of the input value of degree in float

Syntax 2

```
double sin(
    double
)
```

Parameter

`double` Input the value of degree in double

Return

`double` Return the sine of the input value of degree in double

Note

```
var_value = sin(0)          // 0
var_value = sin(15)         // 0.258819
var_value = sin(30)         // 0.5
var_value = sin(60)         // 0.8660254
var_value = sin(90)         // 1
```

3.11 cos()

Return the cosine of the input value of degree

Syntax 1

```
float cos(
    float
)
```

Parameter

`float` Input the value of degree in float

Return

`float` Return the cosine of the input value of degree in float

Syntax 2

```
double cos(
    double
)
```

Parameter

`double` Input the value of degree in double

Return

`double` Return the cosine of the input value of degree in double

Note

```
var_value = cos(0)          // 1
var_value = cos(15)         // 0.9659258
var_value = cos(30)         // 0.8660254
var_value = cos(45)         // 0.7071068
var_value = cos(60)         // 0.5
```

3.12 tan()

Return the tangent of the input value of degree

Syntax 1

```
float tan(
    float
)
```

Parameter

`float` Input the value of degree in float

Return

`float` Return the tangent of the input value of degree in float

Syntax 2

```
double tan(
    double
)
```

Parameter

`double` Input the value of degree in double

Return

`double` Return the tangent of the input value of degree in double

Note

```
var_value = tan(0)          // 0
var_value = tan(15)         // 0.2679492
var_value = tan(30)         // 0.5773503
var_value = tan(45)         // 1
var_value = tan(60)         // 1.732051
```

3.13 asin()

Return the arcsine of the input value in degree

Syntax 1

```
float asin(
    float
)
```

Parameter

`float` Input the sine value in float between -1 and 1

Return

`float` Return the arcsine of the input value of degree in float

Syntax 2

```
double asin(
    double
)
```

Parameter

`double` Input the sine value in double between -1 and 1

Return

`double` Return the arcsine of the input value of degree in double

Note

```
var_value = asin(0)              // 0
var_value = asin(0.258819)       // 15
var_value = asin(0.5)        // 30
var_value = asin(0.8660254)      // 60
var_value = asin(1)              // 90
```

3.14 acos()

Return the arccosine of the input value in degree

Syntax 1

```
float acos(
    float
)
```

Parameter

`float` Input the cosine value in float between -1 and 1

Return

`float` Return the degree value in float

Syntax 2

```
double acos(
    double
)
```

Parameter

`double` Input the cosine value in double between -1 and 1

Return

`double` Return the degree value in double

Note

```
var_value = acos(1)              // 0
var_value = acos(0.9659258)      // 15
var_value = acos(0.8660254)      // 30
var_value = acos(0.7071068)      // 45
var_value = acos(0.5)        // 60
```

3.15 atan()

Return the arctangent of the input value in degree

Syntax 1

```
float atan(
    float
)
```

Parameter

`float` Input the arctangent value in float

Return

`float` Return the degree value in float

Syntax 2

```
double atan(
    double
)
```

Parameter

`double` Input the arctangent value in double

Return

`double` Return the degree value in double

Note

```
var_value = atan(0)          // 0
var_value = atan(0.2679492)  // 15
var_value = atan(0.5773503)  // 30
var_value = atan(1)          // 45
var_value = atan(1.732051)   // 60
```

3.16 atan2()

Return the arctangent of the quotient of it arguments

Syntax 1

```
float atan2(
    float,
    float
)
```

Parameter

`float` Input a number in float representing the Y coordinate

`float` Input a number in float representing the X coordinate

Return

`float` Return the degree value in float

Syntax 2

```
double atan2(
    double,
    double
)
```

Parameter

`double` Input a number in double representing the Y coordinate

`double` Input a number in double representing the X coordinate

Return

`double` Return the degree value in double

Note

```
var_value = atan2(2, 1)      // 63.43495
```

```
var_value = atan2(1, 1)          // 45
var_value = atan2(-1, -1)        // -135
var_value = atan2(4, -3)         // 126.8699
```

3.17 log()

Return the natural logarithm of the input value

Syntax 1

```
float log(
    float,
    double
)
```

Parameter

`float`	Input value in float
`double`	The base of the logarithm

Return

`float`	Return the logarithm of the input value and the base in float

Syntax 2

```
double log(
    double,
    double
)
```

Parameter

`double`	Input value in double
`double`	The base of the logarithm

Return

`double`	Return the logarithm of the input value and the base in double

Note

```
var_value = log(16, 2)           // 4
var_value = log(16, 8)           // 1.333333
var_value = log(16, 10)          // 1.20412
var_value = log(16, 16)          // 1
```

Syntax 3

```
float log(
    float
)
```

Parameter

`float` Input value in float

Return

`float` Return the natural logarithm of the input value and the base e in float

Syntax 4

```
double log(
    double
)
```

Parameter

`double` Input value in double

Return

`double` Return the natural logarithm of the input value and the base e in double

Note

```
var_value = log(16, 2)          // 4
var_value = log(16)             // 2.772589
var_value = log(2)              // 0.6931472
var_value = log(16)/log(2)  // 2.772589/ 0.6931472 = 4.000000288539
```

3.18 log10()

Return the logarithm of the input value with the base 10

Syntax 1

```
float log10(
    float
)
```

Parameter

`float` Input value in float

Return

`float` Return the logarithm of the input value with the base 10 in float

Syntax 2

```
double log10(
    double
)
```

Parameter

`double` Input value in double

Return

`double` Return the logarithm of the input value with the base 10 in double

Note

```
var_value = log(16, 10)        // 1.20412
var_value = log10(16)          // 1.20412
var_value = log(500, 10)       // 2.69897
var_value = log10(500)         // 2.69897
```

3.19 norm2()

Return the second norm of a specified vector.

Syntax 1

```
float norm2(
    float[]
)
```

Parameter

`float[]` A vector whose second norm (or called Euclidean norm, vector magnitude) is to be found.

Return

`float` the second norm (or called Euclidean norm, vector magnitude) of a specified vector

Note

$$\|v\| = \sqrt{\sum_{i=1}^{i=N} |v_i|^2}$$

```
float[] var_vector1 = {3,4}
float[] var_vector2 = {3,4,5}
float[] var_vector3 = {3,4,5,6,8}
var_value = norm2(var_vector1)      // 5
var_value = norm2(var_vector2)      // 7.071068
var_value = norm2(var_vector3)      // 12.24745
```

3.20 dist()

Return the distance between the two coordinates.

Syntax 1

```
float dist(
    float[],
    float[]
)
```

Parameter

`float[]`	The first coordinate	$\{X_{1(mm)} \quad Y_{1(mm)} \quad Z_{1(mm)} \quad RX_{1(°)} \quad RY_{1(°)} \quad RZ_{1(°)}\}$
`float[]`	The second coordinate	$\{X_{2(mm)} \quad Y_{2(mm)} \quad Z_{2(mm)} \quad RX_{2(°)} \quad RY_{2(°)} \quad RZ_{2(°)}\}$

Return

`float`	The distance between the two coordinates

Note

```
float[] var_c1 = {100,200,100,30,50,20}
float[] var_c2 = {100,100,100,50,50,10}
var_value = dist(var_c1, var_c2)        // 100
```

3.21 trans()

Return the displacement and rotation angle from one specified point to another point.

Syntax 1

```
float[] trans(
    float[],
    float[],
    bool
)
```

Parameter

`float[]`	First Point	$\{X_{1(mm)} \quad Y_{1(mm)} \quad Z_{1(mm)} \quad RX_{1(°)} \quad RY_{1(°)} \quad RZ_{1(°)}\}$
`float[]`	Second Point	$\{X_{2(mm)} \quad Y_{2(mm)} \quad Z_{2(mm)} \quad RX_{2(°)} \quad RY_{2(°)} \quad RZ_{2(°)}\}$
`bool`	The reference coordinate	
	`false`	Refer to the robot's base(default)
	`true`	Refer to the first point

Return

`float[]`	The displacement and rotation angle from first point to second point

$$\{X_{trans} \quad Y_{trans} \quad Z_{trans} \quad RX_{trans} \quad RY_{trans} \quad RZ_{trans}\}$$

Return an empty array if unable to calculate.

Syntax 2

```
float[] trans(
    float[],
    float[]
)
```

Note

Same as syntax 1. By default, the reference coordinate is set to false for the robot's base.

Original Transformation Matrix = $\begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$

$$R_n = \begin{bmatrix} cos(RZ_n)\,cos(RY_n) & -sin(RZ_n)cos(RX_n) + cos(RZ_n)sin(RY_n)sin(RX_n) & sin(RZ_n)sin(RX_n) + cos(RZ_n)sin(RY_n)cos(RX_n) \\ sin(RZ_n)cos(RY_n) & cos(RZ_n)cos(RX_n) + sin(RZ_n)sin(RY_n)sin(RX_n) & -cos(RZ_n)sin(RX_n) + sin(RZ_n)sin(RY_n)cos(RX_n) \\ -sin(RY_n) & cos(RY_n)sin(RX_n) & cos(RY_n)cos(RX_n) \end{bmatrix}$$

First Point = $\begin{bmatrix} R_1 & P_1 \\ 0 & 1 \end{bmatrix}$

Second Point = $\begin{bmatrix} R_2 & P_2 \\ 0 & 1 \end{bmatrix}$

If *reference coordinate* is `false` (reference coordinate is Robot Base)

$$P_{trans} = P_2 - P_1$$
$$R_{trans} = R_2 * {R_1}^{-1}$$

If *reference coordinate* is `true` (reference coordinate is First Point)

$$P_{trans} = {R_1}^{-1} * (P_2 - P_1)$$
$$R_{trans} = {R_1}^{-1} * R_2$$

```
float[] var_P1 = {100, -200, 300, 10, 20, 60}
float[] var_P2 = {-400, 200, 50, -20, 30, -45}

float[] var_trans_RB = trans(var_P1, var_P2)
    // {-500, 400, -250, -24.61587, -15.56518, -88.61369}
float[] var_trans_i = trans(var_P1, var_P2, true)
    // {176.101,588.3278,-308.8024,3.745925,23.13792,-92.46916}
```

3.22 inversetrans()

Return the displacement and rotation angle {x, y, z, rx, ry, rz} opposite to the input displacement and

rotation angle {x, y, z, rx, ry, rz}.

Syntax 1

```
float[] inversetrans(
    float[],
    bool
)
```

Parameter

`float[]` The input displacement and rotation angle $\{X_o \quad Y_o \quad Z_o \quad RX_o \quad RY_o \quad RZ_o\}$

`bool` The reference coordinate

`false` Refer to the robot's base(default)

`true` Refer to the input displacement and the rotation angle

Return

`float[]` The displacement and rotation angle $\{\mathrm{X_{inv}} \quad \mathrm{Y_{inv}} \quad \mathrm{Z_{inv}} \quad \mathrm{RX_{inv}} \quad \mathrm{RY_{inv}} \quad \mathrm{RZ_{inv}}\}$ opposite to the input displacement and rotation angle $\{\mathrm{X_o} \quad \mathrm{Y_o} \quad \mathrm{Z_o} \quad \mathrm{RX_o} \quad \mathrm{RY_o} \quad \mathrm{RZ_o}\}$

Return an empty array if unable to calculate.

Syntax 2

```
float[] inversetrans(
    float[]
)
```

Note

Same as syntax 1. By default, the reference coordinate is set to false for the robot's base.

Original Transformation Matrix = $\begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$

$$R_n = \begin{bmatrix} cos(RZ_n)\,cos(RY_n) & -sin(RZ_n)cos(RX_n) + cos(RZ_n)sin(RY_n)sin(RX_n) & sin(RZ_n)sin(RX_n) + cos(RZ_n)sin(RY_n)cos(RX_n) \\ sin(RZ_n)cos(RY_n) & cos(RZ_n)cos(RX_n) + sin(RZ_n)sin(RY_n)sin(RX_n) & -cos(RZ_n)sin(RX_n) + sin(RZ_n)sin(RY_n)cos(RX_n) \\ -sin(RY_n) & cos(RY_n)sin(RX_n) & cos(RY_n)cos(RX_n) \end{bmatrix}$$

Intial Point = $\begin{bmatrix} R_i & P_i \\ 0 & 1 \end{bmatrix}$

If *reference coordinate* is `false` (reference coordinate is Robot Base)

$$P_{inv} = -P_i$$
$$R_{inv} = {R_i}^{-1}$$

If *reference coordinate* is `true` (reference coordinate is the input {x, y, z, rx, ry, rz}, which is equivalent to inverse of Transformation Matrix)

$$P_{inv} = {R_i}^{-1} * (-\ P_i)$$
$$R_{inv} = {R_i}^{-1}$$

```
float[] var_P1 = {100, -200, 300, 10, 20, 60}

float[] var_inv_RB = inversetrans(var_P1)
    // {-100, 200, -300, 12.48313, -18.59011, -60.28337}
float[] var_inv_i = inversetrans(var_P1, true)
    // {218.381, 142.1322, -268.5297, 12.48313, -18.59011, -60.28337}
```

3.23 applytrans()

Return the terminal point computed by applied the displacement and rotation angle to the specified point.

Syntax 1

```
float[] applytrans(
    float[],
    float[],
    bool
)
```

Parameter

`float[]`	Initial point $\{X_i \quad Y_i \quad Z_i \quad RX_i \quad RY_i \quad RZ_i\}$
`float[]`	the displacement and rotation angle $\{X_o \quad Y_o \quad Z_o \quad RX_o \quad RY_o \quad RZ_o\}$
`bool`	The reference coordinate
	`false` Refer to the robot's base(default)
	`true` Refer to the initial point

Return

`float[]`	the terminal point $\{X_t \quad Y_t \quad Z_t \quad RX_t \quad RY_t \quad RZ_t\}$ computed by applied the displacement and rotation angle to the initial point Return an empty array if unable to calculate.

Syntax 2

```
float[] applytrans(
    float[],
    float[]
)
```

Note

Same as syntax 1. By default, the reference coordinate is set to false for the robot's base.

Original Transformation Matrix = $\begin{bmatrix} R & P \\ 0 & 1 \end{bmatrix}$

$$R_n = \begin{bmatrix} cos(RZ_n)\,cos(RY_n) & -sin(RZ_n)cos(RX_n) + cos(RZ_n)sin(RY_n)sin(RX_n) & sin(RZ_n)sin(RX_n) + cos(RZ_n)sin(RY_n)cos(RX_n) \\ sin(RZ_n)cos(RY_n) & cos(RZ_n)cos(RX_n) + sin(RZ_n)sin(RY_n)sin(RX_n) & -cos(RZ_n)sin(RX_n) + sin(RZ_n)sin(RY_n)cos(RX_n) \\ -sin(RY_n) & cos(RY_n)sin(RX_n) & cos(RY_n)cos(RX_n) \end{bmatrix}$$

Initial Point = $\begin{bmatrix} R_i & P_i \\ 0 & 1 \end{bmatrix}$

If *reference coordinate* is `false` (reference coordinate is Robot Base)

$$P_t = P_i + P_{trans}$$
$$R_t = R_{trans} * R_i$$

If *reference coordinate* is `true` (reference coordinate is Initial point)

$$P_t = P_i + R_i * P_{trans}$$
$$R_t = R_i * R_{trans}$$

```
float[] var_P1 = {100, -200, 300, 10, 20, 60}
float[] var_P2 = {-400, 200, 50, -20, 30, -45}

float[] var_trans_RB = trans(var_P1, var_P2)
    // {-500, 400, -250, -24.61587, -15.56518, -88.61369}
float[] var_trans_i = trans(var_P1, var_P2, true)
    // {176.101,588.3278,-308.8024,3.745925,23.13792,-92.46916}

float[] var_apply_RB = applytrans(var_P1, var_trans_RB)
    // {-400, 200, 50, -20, 30, -45.00001}
float[] var_apply_i = applytrans(var_P1, var_trans_i, true)
    // {-400, 200.0001, 49.99998, -20, 30, -45}

float[] var_inv_RB = inversetrans(var_P1)
    // {-100, 200, -300, 12.48313, -18.59011, -60.28337}
float[] var_inv_i = inversetrans(var_P1, true)
    // {218.381, 142.1322, -268.5297, 12.48313, -18.59011, -60.28337}

float[] var_apply_1 = applytrans(var_P1, var_inv_RB)
    // {0, 0, 0, 1.488326E-06, 4.389056E-06, -4.243126E-06}
float[] var_apply_2 = applytrans(var_P1, var_inv_i, true)
    // {-1.029038E-05, 7.456403E-05, 2.154642E-05, -4.266358E-06, 2.728826E-06, -3.719508E-06}
```

3.24 interpoint()

Return the interpolate point between two points according to the specified points and ratio

Syntax 1

```
float[] interpoint(
    float[],
    float[],
    float
)
```

Parameter

`float[]`	First Point $\{X_{1(mm)} \quad Y_{1(mm)} \quad Z_{1(mm)} \quad RX_{1(^\circ)} \quad RY_{1(^\circ)} \quad RZ_{1(^\circ)}\}$
`float[]`	Second Point $\{X_{2(mm)} \quad Y_{2(mm)} \quad Z_{2(mm)} \quad RX_{2(^\circ)} \quad RY_{2(^\circ)} \quad RZ_{2(^\circ)}\}$
`float`	Ratio

Return

`float[]`	the linear interpolate point $\{X_i \quad Y_i \quad Z_i \quad RX_i \quad RY_i \quad RZ_i\}$ between initial point and endpoint according to the ratio. Return an empty array if unable to calculate.

Note

$$\{X_i \quad Y_i \quad Z_i \quad RX_i \quad RY_i \quad RZ_i\} = (\{X_2 \quad Y_2 \quad Z_2 \quad RX_2 \quad RY_2 \quad RZ_2\} - \{X_1 \quad Y_1 \quad Z_1 \quad RX_1 \quad RY_1 \quad RZ_1\}) \times Ratio + \{X_1 \quad Y_1 \quad Z_1 \quad RX_1 \quad RY_1 \quad RZ_1\}$$

```
float[] var_P1 = {-388.3831,-199.8061,367.0702,177.4319,1.717448,-46.02005}
float[] var_P2 = {-436.9584,115.7343,371.4378,179.4419,-42.86601,-96.91867}
float[] var_interp = interpoint(var_P1, var_P2, 0.5)
    // {-412.6707,-42.0359,369.254,172.919,-20.6906,-69.3384}
```

3.25 changeref()

Return the new coordinate value described with the new coordinate system converted from the original coordinate value through the coordinate system conversion. In the process of the conversion, the physical position of the original point in the world of the coordinates will remain the same, the change takes effects on its descriptions of the reference coordinates and the corresponding coordinate values.

Syntax 1

```
float[] changeref(
    float[],
    float[],
    float[]
)
```

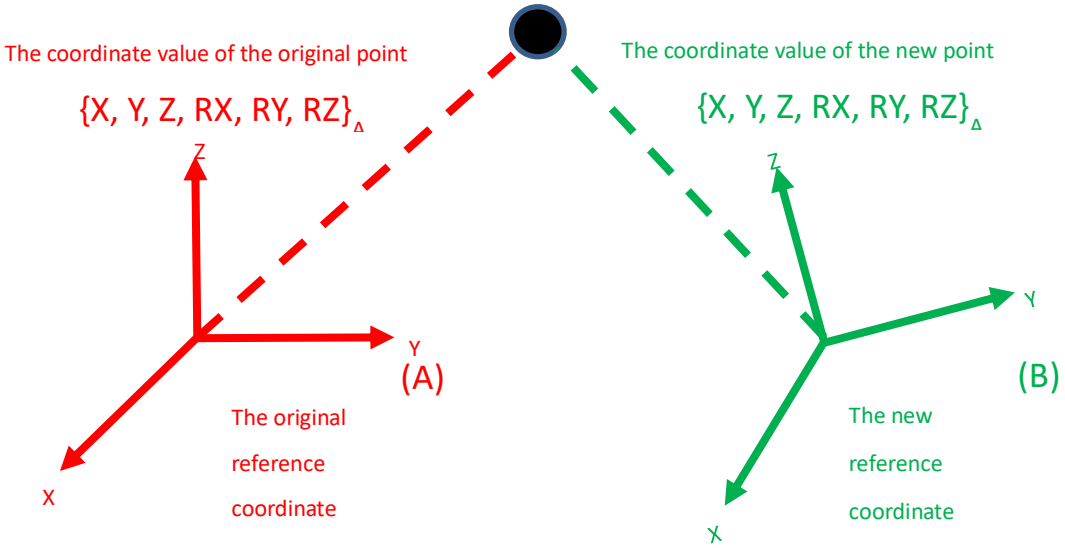
Parameter

`float[]`	The coordinate value of the original point $\{X_o \quad Y_o \quad Z_o \quad RX_o \quad RY_o \quad RZ_o\}_A$
`float[]`	The original reference coordinate system $\{X_{oa} \quad Y_{oa} \quad Z_{oa} \quad RX_{oa} \quad RY_{oa} \quad RZ_{oa}\}_A$
`float[]`	The new reference coordinate system $\{X_n \quad Y_n \quad Z_n \quad RX_n \quad RY_n \quad RZ_n\}_B$

Return

`float[]`	The coordinate value of the new point $\{X_{nb} \quad Y_{nb} \quad Z_{nb} \quad RX_{nb} \quad RY_{nb} \quad RZ_{nb}\}_B$ Return an empty array if unable to calculate.

Note



P1 = {-431.927, -140.6103, 368.7306, -179.288, -0.6893783, -105.8449}

RobotBase = {0, 0, 0, 0, 0, 0}

base1 = {-431.93, -140.61, 368.73, -57.70, -44.98, 33.62}

float[] var_f0 = changeref(Point[“P1”].Value, Base[“RobotBase”].Value, Base[“base1”].Value)

// var_f0 = {0.002052, 0.000020, -0.002272, 113.9423, 14.9346, -123.1989}

// Convert the value of “P1” in the coordinate system ”RobotBase” to the value of a point in the coordinate system ”base1”

Syntax 2

```
float[] changeref(
    float[],
    float[]
```

)

Parameter

`float[]` The coordinate value of the original point $\{X_o \quad Y_o \quad Z_o \quad RX_o \quad RY_o \quad RZ_o\}_A$

`float[]` The original reference coordinate system $\{X_{oa} \quad Y_{oa} \quad Z_{oa} \quad RX_{oa} \quad RY_{oa} \quad RZ_{oa}\}_A$

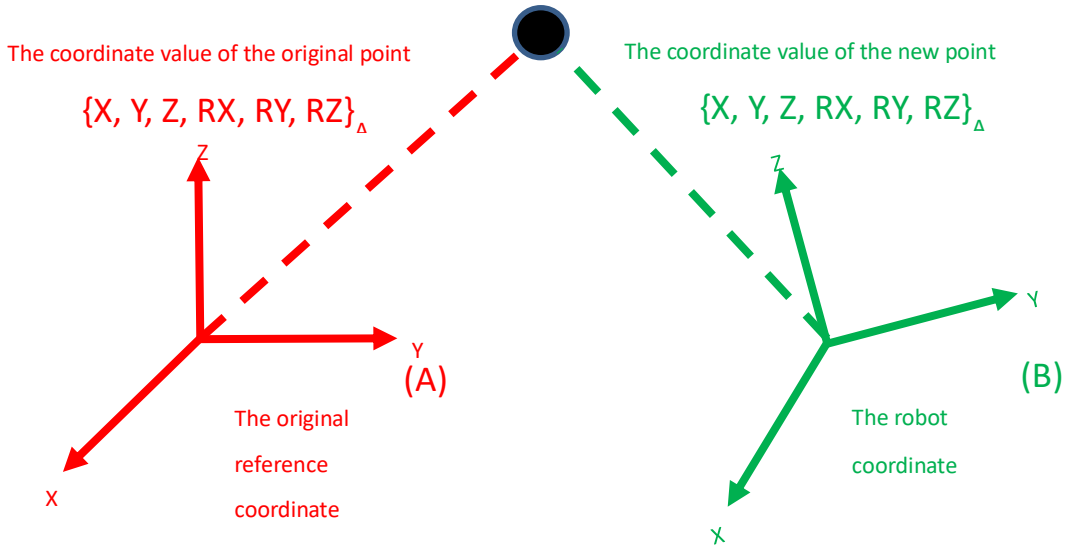
Return

`float[]` The coordinate value of the new point $\{X_{nr} \quad Y_{nr} \quad Z_{nr} \quad RX_{nr} \quad RY_{nr} \quad RZ_{nr}\}_R$

Return an empty array if unable to calculate.

Note

The usage is the same as Syntax1's except assuming the robot coordinate system $\{0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0\}_R$ as the default new reference coordinate system.



base1 = {-431.93, -140.61, 368.73, -57.70, -44.98, 33.62}

f0 = {0.002052, 0.000020, -0.002272, 113.9423, 14.9346, -123.1989}

float[] var_f1 = changeref(var_f, Base["base1"].Value)

// var_f1 = {-431.927, -140.6103, 368.7306, -179.288, -0.6893424, -105.8449}

4. File Functions

- The file functions are capable of operations related to file reading, writing, or inquiry.
- File path

1. Local path. Available in the directories named **TextFiles** or **XmlFiles** only.

FileName.txt	The directory default to .\TextFiles (File path the same as .\TextFiles\FileName.txt)
.\TextFiles\FileName.txt	The file is in the local directory named **TextFiles**.
.\XmlFiles\FileName.xml	The file is in the local directory named **XmlFiles**.
.\XmlFiles\folder\file	The subdirector is in the local directory named **XmlFiles**.
..\folder	Unavailable.
.\TextFiles\..\..\folder	Unavailable.

Void for absolute paths.

C:\file1	Void.
D:\folder\file2	Void.
\TextFiles\FileName.txt	Void.

2. External device path. Available to USB drives or SSD labelled **TMROBOT**.

\USB\TMROBOT	The root directory of the exteranl USB drive.

3. Remote path. Available with the Network service in TMflow.

\\127.0.0.1\shared	SMB / CIFS
ftp://127.0.0.1	FTP

- The path is not case sensitive. For example, the paths below all point to the same file.

  .\TextFiles\FileName.txt

  .\textfiles\fileName.txt

  .\Textfiles\Filename.TXT

- The path is avaiable for pointing to subdirectories such as:

  subfolder\file

  .\TextFiles\subfolder1\subfolder2\file

  .\XmlFiles\subfolder\file

  \USB\TMROBOT\subfolder\file

  \\127.0.0.1\shared\subfolder\file

- The maximum file size is limited to 2MB (2097152 Bytes).

- The type of the array to read or write depends on the definition of the array.

4.1 File_ReadBytes()

Read the file content and return in the type of byte[].

Syntax 1

```
byte[] File_ReadBytes(
    string
)
```

Parameter

`string` File path

Return

`byte[]` Return the file content in the type of byte[].

Note

```
.\TextFiles\SampleFile1.txt
1|  1Hello World!
2|  1Hello TM Robot!
```

byte[] var_bb1 = **File_ReadBytes**("sampleFile1.txt")

// {0x31,0x48,0x65,0x6C,0x6C,0x6F,0x20,0x57,0x6F,0x72,0x6C,0x64,0x21,0x0D,0x0A,

0x31,0x48,0x65,0x6C,0x6C,0x6F,0x20,0x54,0x4D,0x20,0x52,0x6F,0x62,0x6F,0x74,0x21}

byte[] var_bb2 = **File_ReadBytes**(".\TextFiles\SampleFile1.txt")

// {0x31,0x48,0x65,0x6C,0x6C,0x6F,0x20,0x57,0x6F,0x72,0x6C,0x64,0x21,0x0D,0x0A,

0x31,0x48,0x65,0x6C,0x6C,0x6F,0x20,0x54,0x4D,0x20,0x52,0x6F,0x62,0x6F,0x74,0x21}

byte[] var_bb3 = **File_ReadBytes**("C:\SampleFile1.txt")// Error. Void for absolute paths.

byte[] var_bb4 = **File_ReadBytes**(".\SampleFile1.txt") // Error. The file is in the local directory named TextFiles or XmlFiles.

byte[] var_bb5 = **File_ReadBytes**("SampleFileXX.txt") // Error. The file does not exist.

4.2 File_ReadText()

Read the file content and return in the type of string.

Syntax 1

```
string File_ReadText(
    string
```

)

Parameter

`string` File path

Return

`string` Return the file content in the type of string.

Note

```
.\TextFiles\SampleFile1.txt
1|  1Hello World!
2|  1Hello TM Robot!
```

string var_s1 = **File_ReadText**(“sampleFile1.txt”) // “1Hello World!\u0D0A1Hello TM Robot!”

string var_s2 = **File_ReadText**(“.\TextFiles\SampleFile1.txt”) // “1Hello World!\u0D0A1Hello TM Robot!”

* \u0D0A denotes a new line character but not a string value.

string var_s3 = **File_ReadText**(“C:\SampleFile1.txt”) // Error. Void for absolute paths.

string var_s4 = **File_ReadText**(“.\SampleFile1.txt”) // Error. The file is in the local directory named TextFiles or XmlFiles.

4.3 File_ReadLines()

Read the file content and return in the type of string separated by new line characters .

Syntax 1

```
string[] File_ReadLines(
    string
)
```

Parameter

`string` File path

Return

`string[]` Return the file content in the type of string separated by new line characters.

Syntax 2

```
string[] File_ReadLines(
    string,
    int,
    int
)
```

Parameter

`string`	File path
`int`	The number of the line to start to read
`int`	The amount of the lines to read

Return

`string[]`	Return the file content in the type of string separated by new line characters. If the number of the line to start to read <= 0, it returns an empty array. If the number of the line to start to read > the total number of lines, it returns an empty array. If the amount of the lines to read <= 0，it returns from the first line to the last line. If the amount of the lines to read > the total number of lines，it returns from the first line to star to read to the last line.

Syntax 3

```
string[] File_ReadLines(
    string,
    int
)
```

Note

Same as Syntax 2 with the parameter of amount of the lines to read defaults to 0 and returns to the last line. **File_ReadLines**(string,int,int) => **File_ReadLines**(string,int,0)

Note

```
.\TextFiles\SampleFile2.txt
1|  2Hello World!
2|  2Hello TM Robot!
3|  2Hi TM Robot!
```

```
string[] var_ss = {""}
var_ss = File_ReadLines("SampleFile2.txt")        // {"2Hello World!", "2Hello TM Robot!", "2Hi TM Robot!"}
var_ss = File_ReadLines("SampleFile2.txt", 1, 2)  // {"2Hello World!", "2Hello TM Robot!"}
var_ss = File_ReadLines("SampleFile2.txt", 2, 2)  // {"2Hello TM Robot!", "2Hi TM Robot!"}
var_ss = File_ReadLines("SampleFile2.txt", 3, 2)  // {"2Hi TM Robot!"}   // Tops the total number of lines. Returns to the last line.
var_ss = File_ReadLines("SampleFile2.txt", 0)     // {}          // empty array
var_ss = File_ReadLines("SampleFile2.txt", 4)     // {}          // empty array
int var_len = Length(var_ss)                      // 0
var_ss = File_ReadLines("SampleFile2.txt", 3, 0)  // {"2Hi TM Robot!"}   // Returns from line 3 to the last line.
```

```
.\TextFiles\SampleFile3.txt
1|
```

```
var_ss = File_ReadLines("SampleFile3.txt")      // {""}      // var_ss[0] = ""
var_len = Length(var_ss)                        // 1
```

4.4 File_NextLine()

Record the last read file path, and continue to read the next line of the file content or open the file to read.

Syntax1

```
string File_NextLine(
    string
)
```

Parameter

string	File path

Return

string	If the same as the the last read file path, it returns the next line of the file content. If different from the last read file path, it opens the file and returns the first line of the file content. If read the end of the file, it returns an empty string.

Syntax 2

```
string File_NextLine(
    string,
    bool
)
```

Parameter

string	File path
bool	Whether open the file to read or not
false	Try the file path. Continue to read the next line if the same. Open the file to read if different.
true	Open the file and read the first line.

Return

string	Whether open the file to read or not false If the same as the the last read file path, it returns the next line of the file content. If different from the last read file path, it opens the file and returns the first

line of the file content.

Whether open the file to read or not true

It opens the file and returns the first line of the file content.

If read the end of the file, it returns an empty string.

Syntax 3

```
string File_NextLine(
)
```

Parameter

void	No input value

Return

string	Return the next line of the file content in the last record to read or returns an empty string if not read.

Note

```
.\TextFiles\SampleFile4.txt
1|  4Hello World!
2|
3|  4Hello TM Robot!
```

```
.\TextFiles\SampleFile5.txt
1|  5Hello World!
2|  5Hello TM Robot!
3|  5Hi TM Robot!
```

```
string var_s = ""
var_s = File_NextLine()                          // ""                  // Not open the file to read.
var_s = File_NextLine("SampleFile4.txt")         // "4Hello World!"
var_s = File_NextLine("SampleFile4.txt")         // ""                  // Continue to read the next line.
var_s = File_NextLine("SampleFile5.txt")         // "5Hello World!"     // Different file path. Open the file to read.
var_s = File_NextLine("SampleFile4.txt")         // "4Hello World!"     // Different file path. Open the file to read.
var_s = File_NextLine("SampleFile4.txt")         // ""                  // Continue to read the next line
var_s = File_NextLine("SampleFile4.txt")         // "4Hello TM Robot!"
var_s = File_NextLine("SampleFile4.txt")         // ""                  // Continue to read the next line (to the EOF)
var_s = File_NextLine("SampleFile4.txt", true)   // "4Hello World!"     // Open the file to read the first line.
var_s = File_NextLine("SampleFile4.txt", true)   // "4Hello World!"     // Open the file to read the first line.
var_s = File_NextLine("SampleFile4.txt", false)  // ""                  // Continue to read the next line
var_s = File_NextLine()                          // "4Hello TM Robot!"
```

* To determine a blank line or the end of the file, use syntax 4 with the size of string[].

* Or, use File_NextEOF() to determine the end of the file.

Syntax 4

```
string[] File_NextLine(
    string,
    int
)
```

Parameter

string	File path
int	Parameters to read
0	Try the file path. Continue to read the next line if the same. Open the file to read if different.
1	Open the file and read the first line.
2	Open the file without reading. Returns a empty array.

Return

string[]	Return strings in the next line of the file content in an array. If the array size is 1, it denotes read the strings in the next line line. If the array size is 0, it denotes read the end of the file already.

Syntax 5

```
string[] File_NextLine(
    int
)
```

Parameter

int	Parameters to read
0	Try the file path. Continue to read the next line if the same. Open the file to read if different.
1	Open the file and read the first line.
2	Open the file without reading. Returns a empty array.

Return

string[]	Return strings in the next line of the file content in an array in the last record to read or an empty string array if not read. If the array size is 1, it denotes read the strings in the next line line. If the array size is 0, it denotes read the end of the file already.

Note

```
.\TextFiles\SampleFile4.txt
1|  4Hello World!
2|
3|  4Hello TM Robot!
```

```
.\TextFiles\SampleFile5.txt
1|  5Hello World!
2|  5Hello TM Robot!
3|  5Hi TM Robot!
```

```
string[] var_ss = {""}
var_ss = File_NextLine(0)                          // {}                          // Not open the file to read.
var_ss = File_NextLine("SampleFile4.txt", 0)       // {"4Hello World!"}
var_ss = File_NextLine("SampleFile4.txt", 0)       // {""}                        // Continue to read the next line.
var_ss = File_NextLine("SampleFile5.txt", 0)       // {"5Hello World!"}           // Different file path. Open the file to read.
var_ss = File_NextLine("SampleFile4.txt", 0)       // {"4Hello World!"}           // Different file path. Open the file to read.
var_ss = File_NextLine("SampleFile4.txt", 0)       // {""}                        // Continue to read the next line.
int var_len = Length(var_ss)                       // 1
var_ss = File_NextLine("SampleFile4.txt", 0)       // {"4Hello TM Robot!"}
var_ss = File_NextLine("SampleFile4.txt", 0)       // {}                          // Continue to read the next line (to the EOF)
var_len = Length(var_ss)                           // 0
var_ss = File_NextLine("SampleFile4.txt", 1)       // {"4Hello World!"}           //Open the file and read the first line.
var_ss = File_NextLine("SampleFile4.txt", 2)       // {}                          //Open the file without reading.
var_len = Length(var_ss)                           // 0
var_ss = File_NextLine("SampleFile4.txt")          // {"4Hello World!"}           // Continue to read the next line.
var_ss = File_NextLine(0)                          // {""}                        // Continue to read the next line.
var_ss = File_NextLine(0)                          // {"4Hello TM Robot!"}
var_ss = File_NextLine(0)                          // {}
```

4.5 File_NextEOF()

Try the last read file path for reading to the end of the file already.

Syntax 1

```
bool File_NextEOF(
)
```

Parameter

void	No input value but needs to use with File_NextLine()

Return

bool	Return true if not read.	
	false	Not read to the end of the file.
	true	Not open the file or read to the end of the file.

Note

```
.\TextFiles\SampleFile4.txt
1|  4Hello World!
2|
3|  4Hello TM Robot!
```

```
bool var_eof = File_NextEOF()                 // true      // Not open the file to read.
string var_s = ""
var_s = File_NextLine("SampleFile4.txt")      // "4Hello World!"
var_eof = File_NextEOF()                      // false
var_s = File_NextLine("SampleFile4.txt")      // ""
var_eof = File_NextEOF()                      // false
var_s = File_NextLine("SampleFile4.txt")      // 4Hello TM Robot!"
var_eof = File_NextEOF()                      // true
var_s = File_NextLine("SampleFile4.txt")      // ""

File_NextLine("SampleFile4.txt", 2)           // Open the file without reading.
var_eof = File_NextEOF()                      // false
```

4.6 File_WriteBytes()

Put the data content into an array in byte and write to a file.

Syntax 1

```
bool File_WriteBytes(
    string,
    ?,
    int,
    int,
    int
)
```

Parameter

string	File path	
?	Values to write. Eligible for integers, floating-point numbers, booleans, strings, or arrays. Values will be converted with Little Endian, and strings will be converted with UTF8.	
int	Overwrite the file or append to the file.	
	0	Over the file. If not existed, create a new file.
	1	Append to the file. If not existed, create a new file.
int	The starting index of the value to write (eligible for strings and arrays)	
	0 .. length-1	Legitimate value
	< 0	Illegitimate value. The starting index will be set to 0.
	>= length	Illegitimate value. The starting index will be set to 0.
int	The length of the value to write (eligible for strings and arrays)	
	<= 0	Write from the starting index to the end of the data.
	> 0	Write from the staring index for a specified nubmer of the length up to the data ends.

Return

bool	true	Write successfully.
	false	Write unsuccessfully.

Syntax 2

```
bool File_WriteBytes(
    string,
    ?,
    int,
    int
)
```

Note

Same as Syntax 1. Set the length of the value to write to 0 writing up to the data ends.

File_WriteBytes(string,?,int,int) => **File_WriteBytes**(string,?,int,int,0)

Syntax 3

```
bool File_WriteBytes(
    string,
    ?,
    int
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 writing up to the data ends.

File_WriteBytes(string,?,int) => **File_WriteBytes**(string,?,int,0,0)

Syntax 4

```
bool File_WriteBytes(
    string,
    ?
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 overwriting the file up to the data ends.

File_WriteBytes(string,?) => **File_WriteBytes**(string,?,0,0,0)

byte[] var_bb1 = {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08}

byte[] var_bb2 = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38}

byte[] var_bb3 = {}

bool var_flag = false

var_flag = **File_WriteBytes**("writebytes.txt", var_bb1) // Overwrite var_bb1 to the file

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	02	03	04	05	06	07	08							

var_flag = **File_WriteBytes**("writebytes.txt", var_bb2) // Overwrite var_bb2 to the file

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	30	31	32	33	34	35	36	37	38							

var_flag = **File_WriteBytes**("writebytes.txt", var_bb3) // Overwrite var_bb3 to the file

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000																

File_WriteBytes("writebytes.txt", var_bb1, 1) // Append var_bb1 to the file

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	02	03	04	05	06	07	08							

File_WriteBytes("writebytes.txt", var_bb2, 1) // Append var_bb2 to the file

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	02	03	04	05	06	07	08	30	31	32	33	34	35	36
00000010	37	38														

File_WriteBytes("writebytes.txt", var_bb1, 1, 3) // Append var_bb1 to the file starting from index 3 to the end.

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	02	03	04	05	06	07	08	30	31	32	33	34	35	36
00000010	37	38	03	04	05	06	07	08								

File_WriteBytes(“writebytes.txt”, var_bb2, 1, 3, 2) // Append var_bb2 to the file starting from index 3 for the length of 2

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	02	03	04	05	06	07	08	30	31	32	33	34	35	36
00000010	37	38	03	04	05	06	07	08	33	34						

File_WriteBytes(“writebytes.txt”, var_bb1, 1, -1)

// -1 illegitimate value // Append var_bb1 to the file starting from index 0 to the end.

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	02	03	04	05	06	07	08	30	31	32	33	34	35	36
00000010	37	38	03	04	05	06	07	08	33	34	00	01	02	03	04	05
00000020	06	07	08													

File_WriteBytes(“writebytes.txt”, var_bb2, 1, 0, 100)

// Append var_bb2 to the file starting from index 0 for the length of 100 or to the end.

writebytes.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	00	01	02	03	04	05	06	07	08	30	31	32	33	34	35	36
00000010	37	38	03	04	05	06	07	08	33	34	00	01	02	03	04	05
00000020	06	07	08	30	31	32	33	34	35	36	37	38				

? byte var_n1 = 100 // Convert the value with Little Endian.

File_WriteBytes(“writebytes2.txt”, var_n1, 1)

writebytes2.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	64															

? byte[] var_n2 = {100, 200} // Convert every value in the array with Little Endian one after another.

File_WriteBytes(“writebytes2.txt”, var_n2, 1)

writebytes2.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	64	64	C8													

? int var_n3 = 10000

File_WriteBytes(“writebytes2.txt”, var_n3, 1)

writebytes2.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	64	64	C8	10	27	00	00									

? int[] var_n4 = {10000, 20000, 80000}

File_WriteBytes(“writebytes2.txt”, var_n4, 1)

writebytes2.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	64	64	C8	10	27	00	00	10	27	00	00	20	4E	00	00	80
00000010	38	01	00													

? float var_n5 = 1.234

File_WriteBytes(“writebytes2.txt”, var_n5, 1)

writebytes2.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	64	64	C8	10	27	00	00	10	27	00	00	20	4E	00	00	80
00000010	38	01	00	B6	F3	9D	3F									

? float[] var_n6 = {1.23, 4.56, -7.89}

File_WriteBytes(“writebytes2.txt”, var_n6, 1)

writebytes2.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	64	64	C8	10	27	00	00	10	27	00	00	20	4E	00	00	80
00000010	38	01	00	B6	F3	9D	3F	A4	70	9D	3F	85	EB	91	40	E1
00000020	7A	FC	C0													

? double var_n7 = -1.2345

File_WriteBytes(“writebytes3.txt”, var_n7, 1)

writebytes3.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	8D	97	6E	12	83	C0	F3	BF								

? double[] var_n8 = {1.23, -7.89}

File_WriteBytes(“writebytes3.txt”, var_n8, 1)

writebytes3.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	8D	97	6E	12	83	C0	F3	BF	AE	47	E1	7A	14	AE	F3	3F
00000010	8F	C2	F5	28	5C	8F	1F	C0								

? bool var_n9 = true // Convert true to 1 and false to 0.

File_WriteBytes(“writebytes3.txt”, var_n9, 1)

writebytes3.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	8D	97	6E	12	83	C0	F3	BF	AE	47	E1	7A	14	AE	F3	3F
00000010	8F	C2	F5	28	5C	8F	1F	C0	01							

? bool[] var_n10 = {true, false, true, false, false, true, true}

File_WriteBytes(“writebytes3.txt”, var_n10, 1)

writebytes3.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	8D	97	6E	12	83	C0	F3	BF	AE	47	E1	7A	14	AE	F3	3F
00000010	8F	C2	F5	28	5C	8F	1F	C0	01	01	00	01	00	00	01	01

? string var_n11 = “ABCDEFG” // Convert the string into UTF8.

File_WriteBytes(“writebytes3.txt”, var_n11, 1)

writebytes3.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	8D	97	6E	12	83	C0	F3	BF	AE	47	E1	7A	14	AE	F3	3F
00000010	8F	C2	F5	28	5C	8F	1F	C0	01	01	00	01	00	00	01	01
00000020	41	42	43	44	45	46	47									

? string[] var_n12 = {“ABC”, “DEF”, “達明機器人” }

File_WriteBytes(“writebytes3.txt”, var_n12, 1)

writebytes3.txt

Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00000000	8D	97	6E	12	83	C0	F3	BF	AE	47	E1	7A	14	AE	F3	3F
00000010	8F	C2	F5	28	5C	8F	1F	C0	01	01	00	01	00	00	01	01
00000020	41	42	43	44	45	46	47	41	42	43	44	45	46	E9	81	94
00000030	E6	98	8E	E6	A9	9F	E5	99	A8	E4	BA	BA				

4.7 File_WriteText()

Put the data content in a string and write to a file.

Syntax 1

```
bool File_WriteText(
    string,
    ?,
    int,
    int,
    int
)
```

Parameter

string	File path	
?	Values to write. Eligible for integers, floating-point numbers, booleans, strings, or arrays.	
int	Overwrite the file or append to the file.	
	0	Over the file. If not existed, create a new file.
	1	Append to the file. If not existed, create a new file.
int	The starting index of the value to write (eligible for strings and arrays)	
	0 .. length -1	Legitimate value
	< 0	Illegitimate value. The starting index will be set to 0.
	>= length	Illegitimate value. The starting index will be set to 0.
int	The length of the value to write (eligible for strings and arrays)	
	<= 0	Write from the starting index to the end of the data.
	> 0	Write from the staring index for a specified nubmer of the length up to the data ends.

Return

bool	true	Write successfully.
	false	Write unsuccessfully.

Syntax 2

```
bool File_WriteText(
```

```
    string,
    ?,
    int,
    int
)
```

Note

Same as Syntax 1. Set the length of the value to write to 0 writing up to the data ends.

File_WriteText(string,?,int,int) => **File_WriteText**(string,?,int,int,0)

Syntax 3

```
bool File_WriteText(
    string,
    ?,
    int
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 writing up to the data ends.

File_WriteText(string,?,int) => **File_WriteText**(string,?,int,0,0)

Syntax 4

```
bool File_WriteText(
    string,
    ?
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 overwriting the file up to the data ends.

File_WriteText(string,?) => **File_WriteText**(string,?,0,0,0)

string var_s1 = “Hi TM Robot”

string var_s2 = “達明機器人”

bool var_flag = false

var_flag = **File_WriteText**(“writetext.txt”, var_s1) // Overwrite “Hi TM Robot” to the file

```
writetext.txt
1|  Hi TM Robot
```

var_flag = **File_WriteText**(“writetext.txt”, var_s2) // Overwrite “達明機器人” to the file

```
writetext.txt
1|  達明機器人
```

var_flag = **File_WriteText**(“writetext.txt”, var_s1, 1) // Append “Hi TM Robot” to the file

writetext.txt
1| 達明機器人Hi TM Robot

var_flag = **File_WriteText**(“writetext.txt”, var_s2, 1, 2, 3) // Append from index 2 for 3 character “機器人” to the file.

writetext.txt
1| 達明機器人Hi TM Robot機器人

? byte var_n1 = 100 // Convert the value in decimal to a string

File_WriteText(“writetext2.txt”, var_n1, 1)

writetext2.txt
1| 100

? int[] var_n4 = {10000, 20000, 80000} // The array uses the format {, ,}.

File_WriteText(“writetext2.txt”, var_n4, 1)

writetext2.txt
1| 100{10000,20000,80000}

// For other formats, use GetString() to convert to string and write.

? float var_n5 = 1.234

File_WriteText(“writetext2.txt”, var_n5, 1)

writetext2.txt
1| 100{10000,20000,80000}1.234

? double[] var_n8 = {1.23, -7.89}

File_WriteText(“writetext2.txt”, var_n8, 1)

writetext2.txt
1| 100{10000,20000,80000}1.234{1.23,-7.89}

? bool var_n9 = true

File_WriteText(“writetext2.txt”, var_n9, 1)

writetext2.txt
1| 100{10000,20000,80000}1.234{1.23,-7.89}true

? string var_n11 = “ABCDEFG”

File_WriteText(“writetext2.txt”, var_n11, 1)

writetext2.txt
1| 100{10000,20000,80000}1.234{1.23,-7.89}trueABCDEFG

? string[] var_n12 = {“ABC”, “DEF”, “達明機器人” }

File_WriteText(“writetext2.txt”, var_n12, 1)

writetext2.txt
1| 100{10000,20000,80000}1.234{1.23,-7.89}trueABCDEFG{ABC,DEF,達明機器人}

4.8 File_WriteLine()

Put the data content in a string with newline characters (0x0D 0x0A) in the end and write to a file.

Syntax 1

```
bool File_WriteLine(
    string,
    ?,
    int,
    int,
    int
)
```

Parameter

string	File path	
?	Values to write. Eligible for integers, floating-point numbers, booleans, strings, or arrays.	
int	Overwrite the file or append to the file.	
	0	Over the file. If not existed, create a new file.
	1	Append to the file. If not existed, create a new file.
int	The starting index of the value to write (eligible for strings and arrays)	
	0 .. length -1	Legitimate value
	< 0	Illegitimate value. The starting index will be set to 0.
	>= length	Illegitimate value. The starting index will be set to 0.
int	The length of the value to write (eligible for strings and arrays)	
	<= 0	Write from the starting index to the end of the data.
	> 0	Write from the staring index for a specified nubmer of the length up to the data ends.

Return

bool	true	Write successfully.
	false	Write unsuccessfully.

Syntax 2

```
bool File_WriteLine(
    string,
    ?,
    int,
    int
)
```

Note

Same as Syntax 1. Set the length of the value to write to 0 writing up to the data ends.

File_WriteLine(string,?,int,int) => **File_WriteLine**(string,?,int,int,0)

Syntax 3

```
bool File_WriteLine(
    string,
    ?,
    int
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 writing up to the data ends.

File_WriteLine(string,?,int) => **File_WriteLine**(string,?,int,0,0)

Syntax 4

```
bool File_WriteLine(
    string,
    ?
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 overwriting the file up to the data ends.

File_WriteLine(string,?) => **File_WriteLine**(string,?,0,0,0)

string var_s1 = “Hi TM Robot”

string var_s2 = “達明機器人”

bool var_flag = false

var_flag = **File_WriteLine**(“writeline.txt”, var_s2) // Overwrite “達明機器人\u0D0A” to the file

writeline.txt
1| 達明機器人
2|

var_flag = **File_WriteLine**(“writeline.txt”, var_s1) // Overwrite “Hi TM Robot\u0D0A” to the file

writeline.txt
1| Hi TM Robot
2|

var_flag = **File_WriteLine**(“writeline.txt”, var_s2, 1) // Append “達明機器人\u0D0A” to the file

writeline.txt
1| Hi TM Robot
2| 達明機器人
3|

var_flag = **File_WriteLine**(“writeline.txt”, var_s1, 1, 3) // Append “TM Robot\u0D0A” from the starting index 3 to the end to the file.

```
writeline.txt
1|  Hi TM Robot
2|  達明機器人
3|  TM Robot
4|
```

? byte[] var_n2 = {100, 200} // The array uses the format {, ,}.

File_WriteLine(“writeline2.txt”, var_n2, 1)

```
writeline2.txt
1|  {100,200}
2|
```

// For other formats, use GetString() to convert to string and write.

? int var_n3 = 10000 // Convert the value in decimal to a string.

File_WriteLine(“writeline2.txt”, var_n3, 1)

```
writeline2.txt
1|  {100,200}
2|  10000
3|
```

? float[] var_n6 = {1.23, 4.56, -7.89}

File_WriteLine(“writeline2.txt”, var_n6, 1)

```
writeline2.txt
1|  {100,200}
2|  10000
3|  {1.23,4.56,-7.89}
4|
```

? bool var_n9 = true

File_WriteLine(“writeline2.txt”, var_n9, 1)

```
writeline2.txt
1|  {100,200}
2|  10000
3|  {1.23,4.56,-7.89}
4|  true
5|
```

? string var_n11 = “ABCDEFG”

File_WriteLine(“writeline2.txt”, var_n11, 1)

```
writeline2.txt
1|  {100,200}
2|  10000
3|  {1.23,4.56,-7.89}
4|  true
5|  ABCDEFG
6|
```

```
?    string[] var_n12 = {“ABC”, “DEF”, “達明機器人” }

     File_WriteLine(“writeline2.txt”, var_n12, 1)
```

```
writeline2.txt
1|  {100,200}
2|  10000
3|  {1.23,4.56,-7.89}
4|  true
5|  ABCDEFG
6|  {ABC,DEF,達明機器人}
7|
```

4.9 File_WriteLines()

Put the data content in a string array with newline characters (0x0D 0x0A) in the end and write to a file.

Syntax 1

```
bool File_WriteLines(
    string,
    ?,
    int,
    int,
    int
)
```

Parameter

string	File path	
?	Values to write. Eligible for integers, floating-point numbers, booleans, strings, or arrays.	
int	Overwrite the file or append to the file.	
	0	Over the file. If not existed, create a new file.
	1	Append to the file. If not existed, create a new file.
int	The starting index of the value to write (eligible for strings and arrays)	
	0 .. length-1	Legitimate value
	< 0	Illegitimate value. The starting index will be set to 0.
	>= length	Illegitimate value. The starting index will be set to 0.
int	The length of the value to write (eligible for strings and arrays)	
	<= 0	Write from the starting index to the end of the data.
	> 0	Write from the staring index for a specified nubmer of the length up to the data ends.

Return

bool	true	Write successfully.
	false	Write unsuccessfully.

Syntax 2

```
bool File_WriteLines(
    string,
    ?,
    int,
    int
)
```

Note

Same as Syntax 1. Set the length of the value to write to 0 writing up to the data ends.

File_WriteLines(string,?,int,int) => **File_WriteLines**(string,?,int,int,0)

Syntax 3

```
bool File_WriteLines(
    string,
    ?,
    int
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 writing up to the data ends.

File_WriteLines(string,?,int) => **File_WriteLines**(string,?,int,0,0)

Syntax 4

```
bool File_WriteLines(
    string,
    ?
)
```

Note

Same as Syntax 1. Set the starting index and the length of the value to write to 0 overwriting the file up to the data ends.

File_WriteLines(string,?) => **File_WriteLines**(string,?,0,0,0)

* File_WriteText() : convert the data values to write to a string without adding newline characters in the end of the string.
  File_WriteLine(): convert the data values to write to a string with adding newline characters in the end of the string.
  File_WriteLines(): convert the data values to write to a string array with adding newline characters in the end of each element of the array.

string var_ss1 = {“Hi TM Robot”, “達明機器人”}

bool var_flag = false

var_flag = **File_WriteLines**(“writelines.txt”, var_ss1) // Overwrite var_ss1 to the file

```
writelines.txt
1|  Hi TM Robot
2|  達明機器人
3|
```

var_flag = **File_WriteLines**(“writelines.txt”, var_ss1, 1, 1) // Append var_ss1 from the starting index 1 to the end to the file.

```
writelines.txt
1|  Hi TM Robot
2|  達明機器人
3|  Hi TM Robot
4|
```

? byte[] var_n2 = {100, 200}

File_WriteLines(“writelines2.txt”, var_n2, 1)

```
writelines2.txt
1|  100
2|  200
3|
```

? int var_n3 = 10000

File_WriteLines(“writelines2.txt”, var_n3, 1)

```
writelines2.txt
1|  100
2|  200
3|  10000
4|
```

? float[] var_n6 = {1.23, 4.56, -7.89}

File_WriteLines(“writelines2.txt”, var_n6, 1)

```
writelines2.txt
1|  100
2|  200
3|  10000
4|  1.23
5|  4.56
6|  -7.89
7|
```

? string var_n11 = “ABCDEFG”

File_WriteLines(“writelines2.txt”, var_n11, 1)

```
writelines2.txt
1|  100
2|  200
3|  10000
4|  1.23
5|  4.56
6|  -7.89
7|  ABCDEFG
8|
```

? string[] var_n12 = {“ABC”, “DEF”, “達明機器人” }

File_WriteLines(“writelines2.txt”, var_n12, 1)

writelines2.txt

```
 1| 100
 2| 200
 3| 10000
 4| 1.23
 5| 4.56
 6| -7.89
 7| ABCDEFG
 8| ABC
 9| DEF
10| 達明機器人
11|
```

4.10 File_Exists()

Check the file path for availability.

Syntax 1

```
bool File_Exists(
    string
)
```

Parameter

string		File path

Return

bool	true	File path available
	false	File path unavailable
	*Return false file path unavailable for Voided file path without errors.	

Note

bool var_exists = false

var_exists = **File_Exists**(“sampleFile1.txt”) // true

var_exists = **File_Exists**(“sampleFileX.txt”) // false // File path unavailable

var_exists = **File_Exists**(“C:\SampleFile1.txt”) // false // Void for absolute paths.

4.11 File_Length()

Check the file size.

Syntax 1

```
int File_Length(
```

```
    string
)
```

Parameter

`string`	File path

Return

`int`		In int32 data type. The maximum file size is limited to 2147483647 bytes.
	-1	File path unavailable.
	-2	Exceeded the maximum file size limit.

* Return -1 file path unavailable for void file path without errors.

Note

Int var_len = 0

var_len = **File_Length**(“sampleFile1.txt”) // 31

var_len = **File_Length**(“sampleFileX.txt”) // -1 // File path unavailable

var_len = **File_Length**(“C:\SampleFile1.txt”) // -1 // Void for absolute paths.

4.12 File_Delete()

Delet the file.

Syntax 1

```
bool File_Delete(
    string
    ...
)
```

Parameter

`string`	File path
`...`	Available for multiple strings.

Return

`bool`	true	Delete successfully. (Included unavailable or void file paths)
	false	Delete unsuccessfully. (Unable to delete the file for occupied)

Syntax 2

```
bool File_Delete(
    string[]
)
```

Parameter

`string[]`	File path

Return

bool	true	Delete successfully. (Included unavailable or void file paths)
	false	Delete unsuccessfully. (Unable to delete the file for occupied)

Note

```
bool var_flag = false
var_flag = File_Delete(“sampleFile1.txt”)          // true
var_flag = File_Delete(“sampleFileX.txt”)          // true      // File path unavailable
var_flag = File_Delete(“C:\SampleFile1.txt”)       // true      // Void for absolute paths.
var_flag = File_Delete(“sampleFile1.txt”, “sampleFileX.txt”)// Available for multiple file paths.
var_flag = File_Delete(“sampleFile1.txt”, “sampleFileX.txt”, “C:\SampleFile1.txt”)
                                                   // Available for multiple file paths.
string[] var_ss = {“sampleFile1.txt”, “sampleFileX.txt”, “C:\SampleFile1.txt””}
var_flag = File_Delete(var_ss)
var_flag = File_Delete(var_ss, “sampleFile2.txt”) // Error. Void for syntax.
```

4.13 File_Copy()

Copy the file.

Syntax 1

```
bool File_Copy(
    string,
    string,
    string
)
```

Parameter

string	Source file path
string	Target directory path
string	Target file name. Use the source file path equivalent for naming as the default if empty.

Return

bool	true	Copy successfully.
	false	Copy unsuccessfully.
		* Overwrite the target file if existed in the target path.

Syntax 2

```
bool File_Copy(
    string,
```

```
    string
)
```

Note

Same as Syntax 1. Set the target file name with an empty string and use the source file path equivalent for naming.

File_Copy(string,string) => **File_Copy**(string,string,"")

File_Copy("sampleFile1.txt", ".\TextFiles", "s1.txt") // copy .\TextFiles\sampleFile1.txt to .\TextFiles\s1.txt

File_Copy("sampleFile1.txt", ".\XmlFiles", "") // copy .\TextFiles\sampleFile1.txt to .\XmlFiles\sampleFile1.txt

File_Copy("sampleFile1.txt", "\USB\TMROBOT", "s1.txt") // copy .\TextFiles\sampleFile1.txt to USB\s1.txt

File_Copy("sampleFile1.txt", "\USB\TMROBOT") // copy .\TextFiles\sampleFile1.txt to USB\sampleFile1.txt

bool var_flag = false

var_flag = **File_Copy**("sampleFile1.txt", "C:\folder") // Error. Void for absolute paths.

var_flag = **File_Copy**("sampleFile1.txt", ".") // Error. Neither TextFiles nor XmlFiles is in the file path.

4.14 File_Replace()

Replace and overwrite the string in the file with a specified string.

Syntax 1

```
bool File_Replace(
    string,
    string,
    string
)
```

Parameter

string	File path
string	The string to be replaced
string	The string to replace

Return

bool	true	Success	1. The string to be replaced is empty. 2. The string to be replaced is absent. 3. The string to be replaced is found and overwritten in the file.
	false	Unsuccess	

Note

```
.\TextFiles\SampleFile6.txt
1|  6Hello World!
2|  6Hello TM Robot!
3|  6Hi TM Robot!
```

bool var_flag = false

var_flag = **File_Replace**(“SampleFile6.txt”, “Hello”, “HI”)

```
SampleFile6.txt
1|  6HI World!
2|  6HI TM Robot!
3|  6Hi TM Robot!
```

var_flag = **File_Replace**(“SampleFile6.txt”, “TM”, “Techman”)

```
SampleFile6.txt
1|  6HI World!
2|  6HI Techman Robot!
3|  6Hi Techman Robot!
```

var_flag = **File_Replace**(“SampleFile6.txt”, “6”, “”)

```
SampleFile6.txt
1|  HI World!
2|  HI Techman Robot!
3|  Hi Techman Robot!
```

4.15 File_GetToken()

Read the file by the string pattern and retrieve the substring in the string.

Syntax 1

```
string File_GetToken(
    string,
    string,
    string,
    int,
    int
)
```

Parameter

string	File path
string	The prefix of the string to retrieve
string	The suffix of the string to retrieve
int	Matching number
int	Whether to remove the prefix and the suffix or not
	0 Not remove the prefix and the suffix (default)

1 Remove the prefix and the suffix

Return

`string` Return the retrieved string.

Return the content of the string in the file if the prefix and the suffix are empty.

Return an empty string if matching number <=0.

Syntax 2

```
string File_GetToken(
    string,
    string,
    string,
    int
)
```

Note

Same as Syntax 1. Fill 0 for not removing the prefix and the suffix as the default.

File_GetToken(string,string,string,int) => **File_GetToken**(string,string,string,int,0)

Syntax 3

```
string File_GetToken(
    string,
    string,
    string
)
```

Note

Same as Syntax 1. Fill 1 for the matching and 0 for not removing the prefix and the suffix as the default.

File_GetToken(string,string,string) => **File_GetToken**(string,string,string,1,0)

.\TextFiles\SampleFile7.txt

```
1|  $Hello World!
2|  $Hello TM Robot!
3|  $Hi TM Robot!$
```

string var_n = “SampleFile7.txt”

string var_s = “”

var_s = **File_GetToken**(var_n, “”, “”, 0) // “$Hello World!\u0D0A$Hello TM Robot!\u0D0A$Hi TM Robot!$”

var_s = **File_GetToken**(var_n, “$”, “$”) // “$Hello World!\u0D0A$”

var_s = **File_GetToken**(var_n, “$”, “$”, 0) // “”

var_s = **File_GetToken**(var_n, “$”, “$”, 1) // “$Hello World!\u0D0A$”

var_s = **File_GetToken**(var_n, “$”, “$”, 2) // “$Hi TM Robot!$”

var_s = **File_GetToken**(var_n, “$”, “$”, 3) // “”

var_s = **File_GetToken**(var_n, “$”, “$”, 1, 1) // “Hello World!\u0D0A”

```
var_s = File_GetToken(var_n, "$", "$", 2, 1)              // "Hi TM Robot!"
var_s = File_GetToken(var_n, "$", "", 1)          // "$Hello World!\u0D0A"
var_s = File_GetToken(var_n, "$", "", 2)          // "$Hello TM Robot!\u0D0A"
var_s = File_GetToken(var_n, "$", "", 3)          // "$Hi TM Robot!"
var_s = File_GetToken(var_n, "$", "", 4)          // "$"
var_s = File_GetToken(var_n, "", "$", 1)          // "$"
var_s = File_GetToken(var_n, "", "$", 2)          // "Hello World!\u0D0A$"
var_s = File_GetToken(var_n, "", "$", 3)          // "Hello TM Robot!\u0D0A$"
var_s = File_GetToken(var_n, "", "$", 4)          // "Hi TM Robot!$"
var_s = File_GetToken(var_n, "$", Ctrl("\r\n"), 1) // "$Hello World!\u0D0A"
var_s = File_GetToken(var_n, "$", newline, 2)     // "$Hello TM Robot!\u0D0A"
var_s = File_GetToken(var_n, "$", NewLine, 1, 1) // "Hello World!"          // Remove the prefix and the suffix
var_s = File_GetToken(var_n, Ctrl("\r\n"), "$", 1) // "\u0D0A$"
var_s = File_GetToken(var_n, newline, "$", 2)     // "\u0D0A$"
var_s = File_GetToken(var_n, NewLine, "$", 1, 1) // ""
```

* \u0D0A denotes a new line character but not a string value.

Syntax 4

```
string File_GetToken(
    string,
    byte[],
    byte[],
    int,
    int
)
```

Parameter

`string`	File path
`byte[]`	The prefix of the string to retrieve in the byte array
`byte[]`	The suffix of the string to retrieve in the byte array
`int`	Matching number
`int`	Whether to remove the prefix and the suffix or not
	`0` Not remove the prefix and the suffix (default)
	`1` Remove the prefix and the suffix

Return

`string`	Return the retrieved string.
	Return the content of the string in the file if the prefix and the suffix are empty.
	Return an empty string if matching number <=0.

Syntax 5

```
string File_GetToken(
    string,
```

```
    byte[],
    byte[],
    int
)
```

Note

Same as Syntax 4. Fill 0 for not removing the prefix and the suffix as the default.

File_GetToken(string,byte[],byte[],int) => **File_GetToken**(string,byte[],byte[],int,0)

Syntax 6

```
string File_GetToken(
    string,
    byte[],
    byte[]
)
```

Note

Same as Syntax 4. Fill 1 for the matching and 0 for not removing the prefix and the suffix as the default.

File_GetToken(string,byte[],byte[]) => **File_GetToken**(string,byte[],byte[],1,0)

```
.\TextFiles\SampleFile8.txt
1|  $Hello World!
2|  Hello$ TM Robot!
3|  Hi$ TM Robot!$
```

string var_n = “SampleFile8.txt”, var_s = “”

byte[] var_bb0 = {}, var_bb1 = {0x24}, var_bb2 = {0x0D, 0x0A} // 0x24 is $ and 0x0D 0x0A is \u0D0A

var_s = **File_GetToken**(var_n, bb0, bb0, 0) // “$Hello World\u0D0AHello$ TM Robot!\u0D0AHi$ TM Robot!$”

var_s = **File_GetToken**(var_n, bb1, bb1) // “$Hello World\u0D0AHello$”

var_s = **File_GetToken**(var_n, bb1, bb1, 0) // “”

var_s = **File_GetToken**(var_n, bb1, bb1, 1) // “$Hello World\u0D0AHello$”

var_s = **File_GetToken**(var_n, bb1, bb1, 2) // “$ TM Robot!$”

var_s = **File_GetToken**(var_n, bb1, bb1, 3) // “”

var_s = **File_GetToken**(var_n, bb1, bb1, 1, 1) // “Hello World\u0D0AHello”

var_s = **File_GetToken**(var_n, bb1, bb1, 2, 1) // “ TM Robot!”

var_s = **File_GetToken**(var_n, bb1, bb0, 1) // “$Hello World\u0D0AHello”

var_s = **File_GetToken**(var_n, bb1, bb0, 2) // “$ TM Robot!\u0D0AHi”

var_s = **File_GetToken**(var_n, bb1, bb0, 3) // “$ TM Robot!”

var_s = **File_GetToken**(var_n, bb1, bb0, 4) // “$”

var_s = **File_GetToken**(var_n, bb0, bb1, 1) // “$”

var_s = **File_GetToken**(var_n, bb0, bb1, 2) // “Hello World\u0D0AHello$”

var_s = **File_GetToken**(var_n, bb0, bb1, 3) // “ TM Robot!\u0D0AHi$”

var_s = **File_GetToken**(var_n, bb0, bb1, 4) // “ TM Robot!$”

```
var_s = File_GetToken(var_n, bb1, bb2, 1)        // "$Hello World\u0D0A"
var_s = File_GetToken(var_n, bb1, bb2, 2)        // "$ TM Robot!\u0D0A"
var_s = File_GetToken(var_n, bb1, bb2, 1, 1)     // "Hello World"          // Remove the prefix and the suffix
var_s = File_GetToken(var_n, bb2, bb1, 1)        // "\u0D0AHello$"
var_s = File_GetToken(var_n, bb2, bb1, 2)        // "\u0D0AHi$"
var_s = File_GetToken(var_n, bb2, bb1, 1, 1)     // "Hello"
```

* \u0D0A denotes a new line character but not a string value.

4.16 File_GetAllTokens()

Read the file by the string pattern and retrieve all eligible substrings.

Syntax 1

```
string[] File_GetAllTokens(
    string,
    string,
    string,
    int
)
```

Parameter

string	File path
string	The prefix of the string to retrieve
string	The suffix of the string to retrieve
int	Whether to remove the prefix and the suffix or not
	0 Not remove the prefix and the suffix (default)
	1 Remove the prefix and the suffix

Return

string[]	Return the eligible string in an array.
	Return the content of the string in the file as a string array if the prefix and the suffix are empty.

Syntax 2

```
string[] File_GetAllTokens(
    string,
    string,
    string
)
```

Note

Same as Syntax 1. Fill 0 for not removing the prefix and the suffix as the default.

File_GetAllTokens(string,string,string) => **File_GetAllTokens**(string,string,string,0)

.\TextFiles\SampleFile7.txt

```
1|  $Hello World!
2|  $Hello TM Robot!
3|  $Hi TM Robot!$
```

string var_n = “SampleFile7.txt”

string[] var_ss = **File_GetAllTokens**(var_n, “”, “”)// {“$Hello World!\u0D0A$Hello TM Robot!\u0D0A$Hi TM Robot!$”}

var_ss = **File_GetAllTokens**(var_n, “$”, “$”) // {“$Hello World!\u0D0A$”, “$Hi TM Robot!$”}

var_ss = **File_GetAllTokens**(var_n, “$”, “$”, 1) // {“Hello World!\u0D0A”, “Hi TM Robot!”}

var_ss = **File_GetAllTokens**(var_n, “$”, “”, 1) // {“Hello World!\u0D0A”, “Hello TM Robot!\u0D0A”, “Hi TM Robot!”, “”}

5. Serial Port Functions

When the project starts running as going from the start node, it opens the serial port for connections and receives the data from the serial port consistently. For data received in the received buffer, users can use the function com_read to read data in the buffer. Once the project stops running, it closes the opened serial port and clears the received buffer.

The maximum capacity of the received buffer is 2 MB. If there is data coming to the received buffer and the received buffer is out of space, it removes the earliest data automatically for the latest data coming into the received buffer.

5.1 com_read()

Read data in the Serial Port received buffer and return an array byte[].

Syntax 1

```
byte[] com_read(
    string
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

Return

`byte[]` Return all the data content. If the content is empty, it returns byte[0].

Note

var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x20,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}

byte[] var_value = **com_read**(“spd”)

// var_value byte[] = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x20,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}

// var_ReceivedBuffer = {}

*This function reads all data in the received buffer and clears the received buffer.

Syntax 2

```
byte[] com_read(
    string,
    int,
    int
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

`int` The number of the elements to read (based on the length of byte[])

- `<= 0` Read all elements
- `> 0` Read a specified number of the elements (Data is available when the specified number fulfills.)

`int` The length of time to read in millisecond

- `<= 0` Read once only
- `> 0` Read many times until there is data or the time is up.

Return

`byte[]` Return the specified number of the elements with byte[]. If the elements is insufficient, it returns byte[0].

Syntax 3

```
byte[] com_read(
    string,
    int
)
```

Note

The syntax is the same as syntax 2. The default length of time to read is 0.

```
var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x20,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read("spd", 6)
  // var_value byte[] = {0x48,0x65,0x6C,0x6C,0x6F,0x2C}
  // var_ReceivedBuffer = {0x20,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read("spd", 100)
  // var_value byte[] = {}
  // Insufficient elements for no more than 100 elements in the received buffer and return byte[0].
  // var_ReceivedBuffer = {0x20,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read("spd", 0)
  // var_value byte[] = {0x20,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}    // Read all elements
  // var_ReceivedBuffer = {}
var_value = com_read("spd", 4, 100)
  // value byte[] = {}    // Insufficient elements for no more than 4 elements in the received buffer and return byte[0].
```

```
// But the length of time to read is set to 100 ms, the process stays in the function until there is data or the time is
up and exits the function.
// var_ReceivedBuffer = {0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38}      // Supposed it receives data after 50ms,
// var_ value byte[] = {0x31,0x32,0x33,0x34} // it reads 4 element and exits the function.
// var_ReceivedBuffer = {0x35,0x36,0x37,0x38}
```

Syntax 4

```
byte[] com_read(
    string,
    byte[] or string,
    byte[] or string,
    int,
    int
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

`byte[] or string`

Terms of the prefix to read. If the input is byte[0] or “”, an empty string, it means no prefix terms.

`byte[] or string`

Terms of the suffix to read. If the input is byte[0] or “”, an empty string, it means no suffix terms.

`int` To remove the prefix and the suffix from the read content or not

`0` Not remove the prefix and the suffix (default)

`1` Remove the prefix and the suffix

`int` The length of time to read in millisecond

`<= 0` Read once only

`> 0` Read many times until there is data or the time is up.

Return

`byte[]` Return with byte[] in the first matched terms of the prefix and the suffix.

It retrieves data with the content matches the first of all terms, and the rest will be reserved and not retrieved.

If there is no match, it returns byte[0].

Syntax 5

```
byte[] com_read(
    string,
    byte[] or string,
    byte[] or string,
    int
)
```

Note

The syntax is the same as syntax 4. The default length of time to read is 0.

Syntax 6

```
byte[] com_read(
    string,
    byte[] or string,
    byte[] or string
)
```

Note

The syntax is the same as syntax 4. The default is not to remove the prefix and the suffix from the read content and the length of time to read is 0.

```
var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read("spd", "He", newline)              // prefix "He", suffix \u0D0A
  // var_value byte[] = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A}   // Hello,\u0D0A
  //vVar_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}     // retrieve the first match and reserve the rest
var_value = com_read("spd", "", newline, 1)             // prefix "", suffix \u0D0A. Remove both the prefix and the
  suffix.
  // var_value byte[] = {0x57,0x6F,0x72,0x6C,0x64}          // World
  // var_ReceivedBuffer = {}
var_value = com_read("spd", "", newline, 1, 100)        // prefix "", suffix \u0D0A. Remove both the prefix and the
                                                          suffix. The length of time to read is 100ms.
  // var_ value byte[] = {}
  // No matched terms to read. Read byte[0]. Wait for 100 ms.
  // var_ReceivedBuffer = {}
  // var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A}
  // var_value byte[] = {0x48,0x65,0x6C,0x6C,0x6F,0x2C}
  // var_ReceivedBuffer = {}


var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read("spd", "lo", newline)        // prefix "lo", suffix \u0D0A
  // var_value byte[] = {0x6C,0x6F,0x2C,0x0D,0x0A}              // lo,\u0D0A
  // The data before the first matched term, {0x48,0x65,0x6C}, will be removed.
  // var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
  // retrieve the first match and reserve the rest
byte[] var_bb = {}
var_value = com_read("spd", var_bb, newline)              // prefix byte[0], suffix \u0D0A
  // var_ value byte[] = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}        // World\u0D0A
  // var_ReceivedBuffer = {}
var_value = com_read("spd", bb, newline, 0, 100)        // prefix byte[0], suffix \u0D0A, 100ms
```

```
// var_ value byte[] = {}
// No matched terms to read. Read byte[0]. Wait for 100 ms.
// var_ReceivedBuffer = {}
// var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A}
// var_ value byte[] = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A}
// var_ReceivedBuffer = {}
```

Syntax 7

```
byte[] com_read(
    string,
    byte[] or string,
    int,
    int
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

`byte[] or string`

Terms of the suffix to read. If the input is byte[0] or “”, an empty string, it means no suffix terms.

`int` To remove the prefix and the suffix from the read content or not

`0` Not remove the prefix and the suffix (default)

`1` Remove the prefix and the suffix

`int` The length of time to read in millisecond

`<= 0` Read once only

`> 0` Read many times until there is data or the time is up.

* No terms of the prefix to read.

Return

`byte[]` Return with byte[] in the first matched terms of the prefix and the suffix.

It retrieves data with the content matches the first of all terms, and the rest will be reserved and not retrieved.

If there is no match, it returns byte[0].

Syntax 8

```
byte[] com_read(
    string,
    byte[] or string,
    int
)
```

Note

The syntax is the same as syntax 7. The default length of time to read is 0.

Syntax 9

```
byte[] com_read(
```

```
    string,
    byte[] or string
)
```

Note

The syntax is the same as syntax 7. The default is not to remove the prefix and the suffix from the read content and the length of time to read is 0.

Note

```
var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read("spd", newline)       // suffix \u0D0A
   // var_ value byte[] = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A}   // Hello,\u0D0A
   // Var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
   // retrieve the first match and reserve the rest
var_value = com_read("spd", newline)       // suffix \u0D0A
   // var_ value byte[] = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}          // World\u0D0A
   // var_ReceivedBuffer = {}


var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read("spd", newline, 1)          // suffix \u0D0A
   // var_ value byte[] = {0x48,0x65,0x6C,0x6C,0x6F,0x2C}          // Hello,
   // var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
   // retrieve the first match and reserve the rest
var_value = com_read("spd", newline, 1)          // suffix \u0D0A
   // var_ value byte[] = {0x57,0x6F,0x72,0x6C,0x64}                // World
   // var_ReceivedBuffer = {}
var_value = com_read("spd", newline, 1, 100)    // suffix \u0D0A, 100ms
   // var_ value byte[] = {}                                        // No matched terms to read. Read byte[0]. Wait for 100 ms.
   // var_ReceivedBuffer = {}
   // var_ReceivedBuffer = {0x31,0x32,0x33,0x34,0x35,0x36,0x0D,0x0A}
   // var_ value byte[] = {0x31,0x32,0x33,0x34,0x35,0x36}
   // var_ReceivedBuffer = {}
```

5.2 com_read_string()

Read the data in the Serial Port buffer, and convert the data to a UTF8 string.

Syntax 1

```
string com_read_string(
    string
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

Return

`string` Return all the data content. If the content is empty, it returns an empty string.

Syntax 2

```
string com_read_string(
    string,
    int,
    int
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

`int` The number of characters to read (based on the number of characters of the string)

- `<= 0` Read all characters
- `> 0` Read a specified number of the characters (Data is available when the specified number fulfills.)

`int` The length of time to read in millisecond

- `<= 0` Read once only
- `> 0` Read many times until there is data or the time is up.

Return

`string` Returns the specified number of characters as a string. If the characters are insufficient, it returns an empty string.

Syntax 3

```
string com_read_string(
    string,
    int
)
```

Note

The syntax is the same as syntax 2. The default length of time to read is 0.

Syntax 4

```
string com_read_string(
    string,
    byte[] or string,
    byte[] or string,
```

```
    int,
    int
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

`byte[] or string`

Terms of the prefix to read. If the input is byte[0] or “”, an empty string, it means no prefix terms.

`byte[] or string`

Terms of the suffix to read. If the input is byte[0] or “”, an empty string, it means no suffix terms.

`int` To remove the prefix and the suffix from the read content or not

`0` Not remove the prefix and the suffix (default)

`1` Remove the prefix and the suffix

`int` The length of time to read in millisecond

`<= 0` Read once only

`> 0` Read many times until there is data or the time is up.

Return

`string` It retrieves data with the content matches the first of all terms

It retrieves data with the content matches the first of all terms, and the rest will be reserved and not retrieved.

If there is no match, it returns an empty string.

Syntax 5

```
string com_read_string(
    string,
    byte[] or string,
    byte[] or string,
    int
)
```

Note

The syntax is the same as syntax 4. The default length of time to read is 0.

Syntax 6

```
string com_read_string(
    string,
    byte[] or string,
    byte[] or string
)
```

Note

The syntax is the same as syntax 4. The default is not to remove the prefix and the suffix from the read content and the length of time to read is 0.

Syntax 7

```
string com_read_string(
    string,
    byte[] or string,
    int,
    int
)
```

Parameters

`string` The name of the device on the Serial Port (configure in the Serial Port Device)

`byte[] or string`

Terms of the suffix to read. If the input is byte[0] or "", an empty string, it means no suffix terms.

`int` To remove the prefix and the suffix from the read content or not

`0` Not remove the prefix and the suffix (default)

`1` Remove the prefix and the suffix

`int` The length of time to read in millisecond

`<= 0` Read once only

`> 0` Read many times until there is data or the time is up.

* No terms of the prefix to read.

Return

`string` Returns as a string with the first matched terms of the prefix and the suffix.

Retrieves data in the content matches to the first of all terms, and the rest will be reserved and not retrieved.

If there is no match, it returns an empty string.

Syntax 8

```
string com_read_string(
    string,
    byte[] or string,
    int
)
```

Note

The syntax is the same as syntax 7. The default length of time to read is 0.

Syntax 9

```
string com_read_string(
```

```
    string,
    byte[] or string
)
```

Note

The syntax is the same as syntax 7. The default is not to remove the prefix and the suffix from the read content and the length of time to read is 0.

Note

```
var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x20,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
string var_value = com_read_string(“spd”)
  // var_ value string = “Hello, World\u0D0A”
  // var_ReceivedBuffer = {}

var_ReceivedBuffer =
{0x54,0x4D,0xE9,0x81,0x94,0xE6,0x98,0x8E,0xE6,0xA9,0x9F,0xE5,0x99,0xA8,0xE4,0xBA,0xBA}
var_value = com_read_string(“spd”, 4)
  // var_ value string = “TM 達明” // {0x54,0x4D,0xE9,0x81,0x94,0xE6,0x98,0x8E}
  //Retrieve 4 characters based on the length of the string.
  // var _ReceivedBuffer = {0xE6,0xA9,0x9F,0xE5,0x99,0xA8,0xE4,0xBA,0xBA}
var_value = com_read_string(“spd”, 5, 100)
  // var_ value string = “”
  // Insufficient characters for no more than 5 characters in the received buffer based on the length of the string.
  Wait for 100 ms.
  // var_ReceivedBuffer = {0xE6,0xA9,0x9F,0xE5,0x99,0xA8,0xE4,0xBA,0xBA}
  // var_ReceivedBuffer = {0xE6,0xA9,0x9F,0xE5,0x99,0xA8,0xE4,0xBA,0xBA, 0x0D, 0x0A}
  // var_value string = “機器人\u0D0A”
  // var_ReceivedBuffer = {}

Var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read_string(“spd”, “He”, newline)   // prefix “He”, suffix \u0D0A
  // value string = “Hello,\u0D0A”
  // var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}      // retrieve the first match and reserve the rest
var_value = com_read_string(“spd”, “”, newline, 1)
  // prefix “”, suffix \u0D0A. Remove both the prefix and the suffix.
  // var_ value string = “World”
  // var_ReceivedBuffer = {}
var_value = com_read_string(“spd”, “”, newline, 1, 100)
  // var_ value string = “”                    // No matched terms to read. Read an empty string. Wait for 100 ms.
```

```
// var_ReceivedBuffer = {}
// var_ReceivedBuffer = {0xE6,0xA9,0x9F,0xE5,0x99,0xA8,0xE4,0xBA,0xBA, 0x0D, 0x0A}
// var_ value string = "機器人"

var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A,0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
var_value = com_read_string("spd", "lo", newline)     // prefix "lo", suffix \u0D0A
// var_ value string = "lo,\u0D0A"
// The data before the first matched term, "Hel", will be removed.
// var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A}
// retrieve the first match and reserve the rest
var_value = com_read_string("spd", newline, 1) // suffix \u0D0A
// var_ value string = "World"
// var_ReceivedBuffer = {}
var_ReceivedBuffer = {0x48,0x65,0x6C,0x6C,0x6F,0x2C,0x0D,0x0A,0x57,0x6F,0x72,0x6C,0x64}
var_value = com_read_string("spd", newline)           // suffix \u0D0A
// var_ value string = "Hello,\u0D0A"
// var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64}     // retrieve the first match and reserve the rest
var_value = com_read_string("spd", newline, 0) // suffix \u0D0A
// var_ value string = ""                              // No matched terms to read. Read an empty string.
// var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64}
var_value = com_read_string("spd", newline, 1, 100)  // suffix \u0D0A
// var_ value string = ""
// No matched terms to read. Read an empty string. Wait for 100 ms.
// var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64}
// var_ReceivedBuffer = {0x57,0x6F,0x72,0x6C,0x64,0x0D,0x0A,0x31,0x32,0x33,0x0D,0x0A}
// var_ value string = "World"
// var_ReceivedBuffer = {0x31,0x32,0x33,0x0D,0x0A}     // retrieve the first match and reserve the rest
```

5.3 com_write()

Write data to the Serial Port

Syntax 1

```
bool com_write(
    string,
    ?
)
```

Parameters

`string`	The name of the device on the Serial Port (configure in the Serial Port Device)
`?`	The value to write. Available types: int, float, , bool, string, and array. Numeric values will be conversed in Little Endian, and string values will be converse in UTF8.

Return

`bool`	`True`	write successfully	
	`False`	write unsuccessfully	1. The value to write is an empty string or an empty array. 2. Unable to send to the serial port correctly.

Syntax 2

```
bool com_write(
    string,
    ?,
    int,
    int
)
```

Parameters

`string`	The name of the device on the Serial Port (configure in the Serial Port Device)		
`?`	The value to write. Available types: int, float, , bool, string, and array. Numeric values will be conversed in Little Endian, and string values will be converse in UTF8.		
`int`	The starting index of the data to write. (valid for strings or arrays)		
	0	The length of the string - 1	Legal value
	< 0		Illegal value, and the starting index will be 0.
	>=	The length of the string	Illegal value, and the starting index will be 0.
`int`	The length of the data to write. (valid for strings or arrays)		
	<= 0		From the start of the index to the end of the data
	> 0		From the start of the index, write the specified length of the data until the data ends.

Return

`bool`	`True`	write successfully	
	`False`	write unsuccessfully	1. The value to write is an empty string or an empty array. 2. Unable to send to the serial port correctly.

Syntax 3

```
bool com_write(
    string,
    ?,
    int
```

)

Note

The syntax is the same as syntax 2. The default length of data to write is 0.

```
var_flag = com_write(“spd”, 100)              // write 0x64
var_flag = com_write(“spd”, 1000)             // write 0xE8 0x03 0x00 0x00 (int, Little Endian)
var_flag = com_write(“spd”, (float)1.234)     // write 0xB6 0xF3 0x9D 0x3F (float, Little Endian)
var_flag = com_write(“spd”, (double)123.456)
   // write 0x77 0xBE 0x9F 0x1A 0x2F 0xDD 0x5E 0x40 (double, Little Endian)
var_flag = com_write(“spd”, “Hello, World”+newline)
   // write 0x48 0x65 0x6C 0x6C 0x6F 0x2C 0x20 0x57 0x6F 0x72 0x6C 0x64 0x0D 0x0A (string, UTF8)
var_flag = com_write(“spd”, 1000, 1, 2)                  //Invalid in the value, the starting index, and the length
   // write 0xE8 0x03 0x00 0x00 (int, Little Endian)

byte[] var_bb = {100, 200}
var_flag = com_write(“spd”, var_bb)           // write 0x64 0xC8
var_flag = com_write(“spd”, var_bb, 1, 1)     // write 0xC8
// Array. Retrieve 1 element from the index 1. [1]=200
var_flag = com_write(“spd”, var_bb, -1, 1)    // write 0x64
// Array. Retrieve 1 element from the index 0. [0]=100
var_flag = com_write(“spd”, “達明機器人”, 2)
// String. Retrieve from the index 2 until the index ends. ”機器人”
// write 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA (string, UTF8)
string[] var_ss = {“TM”, “”, “達明機器人” }
var_flag = com_write(“spd”, var_ss)
   // write 0x54 0x4D 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA
var_flag = com_write(“spd”, Byte_Concat(GetBytes(var_ss), GetBytes(newline)))
   // write 0x54 0x4D 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA 0x0D 0x0A
var_flag = com_write(“spd”, var_ss, 2, 100)
   // Array. Retrieve 100 elements (to the end) from the index 2. [2]=達明機器人
   // write 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA
```

5.4 com_writeline()

Write data to the Serial Port and add line break symbols, 0x0D 0x0A, in the end of the data automatically

Syntax 1

```
bool com_writeline(
    string,
```

```
    ?
)
```

Parameters

`string`	The name of the device on the Serial Port (configure in the Serial Port Device)
`?`	The value to write. Available types: int, float, , bool, string, and array. Numeric values will be conversed in Little Endian, and string values will be converse in UTF8.

Return

`bool`	`True`	write successfully	
	`False`	write unsuccessfully	1. The value to write is an empty string or an empty array. 2. Unable to send to the serial port correctly.

Syntax 2

```
bool com_writeline(
    string,
    ?,
    int,
    int
)
```

Parameters

`string`	The name of the device on the Serial Port (configure in the Serial Port Device)		
`?`	The value to write. Available types: int, float, , bool, string, and array. Numeric values will be conversed in Little Endian, and string values will be converse in UTF8.		
`int`	The starting index of the data to write. (valid for strings or arrays)		
	0	The length of the string - 1	Legal value
	< 0		Illegal value, and the starting index will be 0.
	>=	The length of the string	Illegal value, and the starting index will be 0.
`int`	The length of the data to write. (valid for strings or arrays)		
	<= 0		From the start of the index to the end of the data
	> 0		From the start of the index, write the specified length of the data until the data ends.

Return

`bool`	`True`	write successfully	
	`False`	write unsuccessfully	1. The value to write is an empty string or an empty array. 2. Unable to send to the serial port correctly.

Syntax 3

```
bool com_writeline(
```

```
    string,
    ?,
    int
)
```

Note

The syntax is the same as syntax 2. The default length of data to write is 0.

```
var_flag = com_writeline("spd", 100)              // write 0x64 0x0D 0x0A
var_flag = com_writeline("spd", 1000)             // write 0xE8 0x03 0x00 0x00 0x0D 0x0A (int, Little Endian)
var_flag = com_writeline("spd", (float)1.234)     // write 0xB6 0xF3 0x9D 0x3F 0x0D 0x0A (float, Little Endian)
var_flag = com_writeline("spd", (double)123.456)
    // write 0x77 0xBE 0x9F 0x1A 0x2F 0xDD 0x5E 0x40 0x0D 0x0A (double, Little Endian)
var_flag = com_write("spd", "Hello, World"+newline)
    // write 0x48 0x65 0x6C 0x6C 0x6F 0x2C 0x20 0x57 0x6F 0x72 0x6C 0x64 0x0D 0x0A (string, UTF8)
var_flag = com_writeline("spd", "Hello, World")
    // write 0x48 0x65 0x6C 0x6C 0x6F 0x2C 0x20 0x57 0x6F 0x72 0x6C 0x64 0x0D 0x0A (string, UTF8)
var_flag = com_writeline("spd", 1000, 1, 2)       // Invalid in the value, the starting index, and the length
    // write 0xE8 0x03 0x00 0x00 0x0D 0x0A (int, Little Endian)

byte[] var_bb = {100, 200}
var_flag = com_writeline("spd", var_bb)           // write 0x64 0xC8 0x0D 0x0A
var_flag = com_writeline("spd", var_bb, 1, 1)     // write 0xC8 0x0D 0x0A
                                                  // Array. Retrieve 1 element from the index 1. [1]=200
var_flag = com_writeline("spd", var_bb, -1, 1)    // write 0x64 0x0D 0x0A
                                                  // Array. Retrieve 1 element from the index 0. [0]=100
var_flag = com_writeline("spd", "達明機器人", 2)
    // String. Retrieve from the index 2 until the index ends. "機器人"
    // write 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA 0x0D 0x0A (string, UTF8)
string[] var_ss = {"TM", "", "達明機器人" }
var_flag = com_writeline("spd", var_ss)
  // write 0x54 0x4D 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA 0x0D 0x0A
var_flag = com_write("spd", Byte_Concat(GetBytes(var_ss), GetBytes(newline)))
  // write 0x54 0x4D 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA 0x0D 0x0A
var_flag = com_writeline("spd", var_ss)
  // write 0x54 0x4D 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA 0x0D 0x0A
var_flag = com_writeline("spd", var_ss, 2, 100)
    // Array. Retrieve 100 elements (to the end) from the index 2. [2]=達明機器人
    // write 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8 0xE4 0xBA 0xBA 0x0D 0x0A
```

6. Parameterized objects

Using parameterized objects is the same as using user defined variables. Parameterized objects can be used without declarations to get or modify point data through the syntaxes in the project operations and make the robot go with more flexibility. The expression comes with 3 parts, item, index, and attribute, and the syntax is shown as below.

parameterized item[index].attribute

The supported parameterized items include:

1. Point
2. Base
3. TCP
4. VPoint
5. IO
6. Robot
7. FT

Definitions of the indexes and the attributes vary from parameterized items.

Take the reading and writing of the coordinate (attribute) of the Point (item) "P1" (index) as a example. The index is defined as the name of the point, and the attribute, as the data type of float (the same usage as the array's) with modes of reading and writing.

Value	float[]	R/W	point coordinate{X,Y,Z,RX,RY,RZ}

Read values

```
float[] var_f = Point["P1"].Value          // In the item Point, the index is defined as the name of the point and the data type of string.
float var_f1 = Point["P1"].Value[0]        // The x value of "P1" can be obtained solely
```

Write values

```
Point["P1"].Value = {0, 0, 90, 0, 90, 0}   // Replace to the coordinate of "P1" with {0,0,90,0,90,0}
Point["P1"].Value[2] = 120                 // or replace the z value of "P1" with 12 solely
```

6.1 Point

Syntax

```
Point[string].attribute
```

Item

```
Point
```

Index

`string` The name of the point in the point manager

Attribute

Name	Type	Mode	Description	Format
Value	float[]	R/W	The coordiniate of the point	{X, Y, Z, RX, RY, RZ}, Size = 6
Pose	int[]	R/W	The pose of the robot	{Config1, Config2, Config3}, Size = 3
Flange	float[]	R	The coordinate of the flange'scenter	{X, Y, Z, RX, RY, RZ}, Size = 6
BaseName	string	R	The name of the base	"Base Name"
TCPName	string	R	The name of the TCP	"TCP Name"
TeachValue	float[]	R	The original coordiniate of the teaching point	{X, Y, Z, RX, RY, RZ}, Size = 6
TeachPose	int[]	R	The original pose of the roboton the teaching point	{Config1, Config2, Config3}, Size = 3

Note

```
// Read values
float[] var_f = Point["P1"].Value          // Obatin the coordinate {X, Y, Z, RX, RY, RZ} of "P1"
float var_f1 = Point["P1"].Value[0]        // or retrieve the x value of "P1" solely
float var_f1 = Point["P1"].Value[6]        // Return error, exceeding the array's access range
string var_s =Point["P1"].BaseName         // var_s ="RobotBase"
// Write values
Point["P1"].Value = {0, 0, 90, 0, 90, 0}   // Replace the coordinate of "P1" with {0,0,90,0,90,0}
Point["P1"].Value[2] = 120                 // or replace the z value of "P1" with 120 solely
Point["P1"].Flange = {0, 0, 90, 0, 90, 0}  // Read only, invalid operation
Point["P1"].Value = {0, 0, 90, 0, 90}      // Return error, writing elements to the array do not match to 6
                                           (writing 5 elements)
Point["P1"].Pose = {1, 2, 4, 0}            // Return error, writing elements to the array do not match to 3
                                           (writing 4 elements)
```

6.2 Base

Syntax

```
Base[string].attribute

Base[string, int].attribute
```

Item

```
Base
```

Index

`string`	The name of the base in the base manager *The name of the base comes with the attribute of the mode in reading without writing only. "RobotBase"
`int`	The index of the base, available to assign with multiple bases built by vision one shot get all, ranging from 0 as the dafult to N.

Attribute

Name	Type	Mode	Description	Format
`Value`	float[]	R/W	The value of the base	{X, Y, Z, RX, RY, RZ}, Size = 6
`Type`	string	R	The type of the base	"R": Robot Base "V": Vision Base "C": Custom Base
`TeachValue`	float[]	R	Theoriginal teaching vale of the base	{X, Y, Z, RX, RY, RZ}, Size = 6

Note

```
// Read values
float[] var_f = Base[“RobotBase”].Value          // Obatin the base value {0,0,0,0,0,0} of the base “RobotBase”
float var_f1 = Base[“base1”].Value[0]            // or retrieve the x value of “base1” solely
string var_s =Base[“base1”].Type                 // var_s =“C”
var_s =Base[Point[“P1”].BaseName].Type           // var_s =“R”          // Given the type of “P1” is “RobotBase”
float[] var_f = Base[“vision_osga”,1].Value      // Obtain the 2nd value of the “vision_osga”
// Write values
Base[“RobotBase”].Value = {0, 0, 90, 0, 90, 0}   // Read only, invalid operation, because "RobotBase" is the system coordinate system
Base[“base1”].Value = {0, 90, 0, 0, 90, 0}       // Replace the value of “base1” with {0,90,0,0,90,0}
Base[“base1”].Value[4] = 120                     // or replace the RY value of “base1” with 120 solely
Base[“base1”].Value[6] = 120                     // Return error, exceeding the array’s access range
```

```
Base[“base1”].Type = “C”                    // Read only, invalid operation
Base[“base1”].Value = {0, 0, 90, 0, 90}     // Return error, writing elements to the array do not match to 6 (writing 5 elements)
Base[“base1”].Value = {0, 0, 90, 0, 90, 0, 100}  // Return error, writing elements to the array do not match to 6 (writing 7 elements)
```

6.3 TCP

Syntax

```
TCP[string].attribute
```

Item

```
TCP
```

Index

`string` The name of the TCP in the TCP list

*The name of the TCP comes with the attribute of the mode in reading without writing only.

"NOTOOL"

"HandCamera"

Attribute

Name	Type	Mode	Description	Format
Value	float[]	R/W	The value of the TCP	{X, Y, Z, RX, RY, RZ}, Size = 6
Mass	float	R/W	The value of mass	Mass in kg
MOI	float[]	R/W	The value of the Principal Moments of Inertia	{Ixx, Iyy, Izz}, Size = 3
MCF	float[]	R/W	The value of Mass center frame with principle axes w.r.t tool frame	{X, Y, Z, RX, RY, RZ}, Size = 6
TeachValue	float[]	R	The original value of the TCP	{X, Y, Z, RX, RY, RZ}, Size = 6
TeachMass	float	R	The original value of mass	Mass in kg
TeachMOI	float[]	R	The original value of the Principal Moments of Inertia	{Ixx, Iyy, Izz}, Size = 3
TeachMCF	float[]	R	The original value of Mass center frame with principle axes w.r.t tool frame	{X, Y, Z, RX, RY, RZ}, Size = 6

Note

```
// Read values
float[] var_f = TCP[“NOTOOL”].Value      // Obtain the value {0,0,0,0,0,0} of the TCP “NOTOOL”
float var_f1 = TCP[“NOTOOL”].Value[0]    // or retrieve the x value of “NOTOOL” solely
```

```
float var_mass = TCP[“T1”].Mass                    // var_mass = 2.0
float[] var_moi = TCP[“T1”].MOI                    // var_ moi = {0,0,0}
float[] var_mcf = TCP[“T1”].MCF                    // var_mcf = {0,0,0,0,0,0}
// Write values
TCP[“NOTOOL”].Value = {0, -10, 0, 0, 0, 0}         // Read only, invalid operation, because ”NOTOOL” is the system TCP
TCP[“T1”].Value = {0, -10, 0, 0, 0, 0}             // Replace the value of “T1”with {0,-10,0,0,0,0}
TCP[“T1”].Value[0] = 10                            // or replace the X value of “T1” with 10 solely
TCP[“T1”].Mass = 2.4                               // Replace the mass value of “T1” with 2.4 kg
TCP[“T1”].MOI = {0, 0, 0, 1, 2}                    // Return error, writing elements to the array do not match to 3 (writing 5 elements)
TCP[“T1”].MCF = {0, -20, 0, 0, 0, 0, 0}            // Return error, writing elements to the array do not match to 6 (writing 7 elements)
```

6.4 VPoint

Syntax

```
VPoint[string].attribute
```

Item

VPoint	Initial position of the vision job

Index

string	The name of the VPoint

Attribute

Name	Type	Mode	Description	Format
Value	float[]	R/W	The initial coordinate of VPoint	{X, Y, Z, RX, RY, RZ}, Size = 6
BaseName	string	R	The name of the VPoint	"Base Name"
TeachValue	float[]	R	The original job initial coordinate of VPoint	{X, Y, Z, RX, RY, RZ}, Size = 6

Note

```
// Read values
float[] var_f = VPoint[“Job1”].Value          // Obtain the initial coordinate  {X, Y, Z, RX, RY, RZ} of VPoint “Job1”
float var_f1 = VPoint[“Job1”].Value[0]        // or retrieve the x valueof “Job1”
float var_f1 = VPoint[“Job1”].Value[6]        // Return error, exceeding the array’s access range
string var_s = VPoint[“Job1”].BaseName        // var_s =“RobotBase”
// Write values
VPoint[“Job1”].Value = {0, 0, 90, 0, 90, 0}   // Replacethe initial coordinate of VPoint “Job1” with {0,0,90,0,90,0}
```

```
VPoint["Job1"].Value[2] = 120                    // or replace the Z value of "Job1" with 120 solely
VPoint["Job1"].BaseName = "base1"                // Read only, invalid operation
VPoint["Job1"].Value = {0, 0, 90, 0, 90}         // Return error, writing elements to the array do not match to
                                                 6 (writing 5 elements)
VPoint["Job1"].Value = {0, 0, 90, 0, 90, 0, 100} // Return error, writing elements to the array do not match to
                                                 6 (writing 7 elements)
```

6.5 IO

Syntax

```
IO[string].attribute
```

Item

```
IO
```

Index

string	The name of the control module
	ControlBox
	EndModule
	ExtModuleN (N = 0 .. n)
	Safety

Attribute

ControlBox / EndModule / ExtModuleN

Name	Type	Mode	Description	Format
DI	byte[]	R	Digital input	[0] = DI0 0: Low, 1: High [1] = DI1 [n] = DIn
DO	byte[]	R/W	Digital output	[0] = DO0 0: Low, 1: High [1] = DO1 [n] = DOn
AI	float[]	R	Analog input	-10.24V .. +10.24V (Voltage)
AO	float[]	R/W	Analog output	-10.00V .. + 10.00V (Voltage)
InstantDO	byte[]	R/W	Digital output (Instant Command)	[0] = DO0 0: Low, 1: High [1] = DO1 [n] = DOn
InstantAO	float[]	R/W	Analog output (Instant Command)	-10.00V .. + 10.00V (Voltage)

* The set of DI[n]/DO[n]/AI[n]/AO[n] vary from actual hardware device identification.

Safety

Name	Type	Mode	Description	Format
SI	byte[]	R	Safety function input	0: Low, 1: High SI[0] = SF1 User Connected ESTOP input SI[1] = SF3 User Connected External Safeguard Input SI[2] = SF9 User Connected External Safeguard Input for Human-Machine Safety Setting SI[3] = SF15 User Connected Enabling Device Input SI[4] = SF16 User Connected ESTOP Input without Robot ESTOP Output
SO	byte[]	R	Safety function output	0: Low, 1: High SO[0] = SF10 Robot ESTOP Output SO[1] = SF11 User Connected External Safeguard Output SO[2] = SF12 User Connected External Safeguard Output for Human-Machine Safety Settings SO[3] = SF13 Robot Internal Protective Stop Output SO[4] = SF14 Robot Encoder Standstill Output

The difference between Do and InstantDO

DO is the queue command with reservations in the main flow of the project. If a DO is after the robot motion function such as a point node with the mixture of trajectories, the DO will be operated after the point node is finished. If using an InstantDO command, the DO will be operated while the point node is on the run and without the wait for the point node finishes, and the result is the same as using DO in a thread page.

Note

```
// Read values
byte[] var_di = IO[“ControlBox”].DI          // Obtain the digital input status of ControlBox
int var_dilen = Length(di)                   // Obtain the amount of digital PINs with the size of the array
byte var_di0 = IO[“ControlBox”].DI[0]        // Obtain the status of ControlBox DI[0]
byte var_di32 = IO[“ControlBox”].DI[32]      // Return error, exceeding the array’s access range (given DI is an
                                             array with the length of 16 where the indexes start with 0 and end with 15.
float[] var_ai = IO[“ControlBox”].AI         // Obtain the analog input status of ControlBox
float[] var_ao = IO[“ControlBox”].AO         // Obtain the analog output status of ControlBox
byte si0 = IO[“Safety”].SI[0]                // Obtain the safety input status of Safety SI[0]
byte so4 = IO[“Safety”].SO[4]                // Obtain the safety output status of Safety SO[4]
byte si1 = IO[“ControlBox”].SI[1]            // Return error, ControlBox does not support SI attribute.
byte di2 = IO[“Safety”].DI[2]                // Return error, Safety does not support DI attribute.
```

```
// Write values
IO["ControlBox"].DI = {1,1,0,0}        // Read only, invalid operation
IO["ControlBox"].DI[0] = 0             // Read only, invalid operation
IO["ControlBox"].DO[2] = 1             // Set DO 2 to High
IO["ControlBox"].AO[0] = 3.3           // Set AO0 to 3.3V
IO["ControlBox"].DO = {1,1,0,0}        // Return error, elements to write mismatch to array's size (given DI is an
                                       array with the length of 16 which covers 16 elements)
IO["ControlBox"].InstantDO[0] = 1      // Set DO 0 to High (Instant Execution)
IO["Safety"].SI[0] = 0                 // Read only, invalid operation
IO["Safety"].SO[4] = 1                 // Read only, invalid operation
IO["ControlBox"].SO[1] = 1             // Return error, ControlBox does not support SO attribute.
IO["Safety"].DO[2] = 1                 // Return error, Safety does not support DO attribute.
```

6.6 Robot

Syntax

```
Robot[int].attribute
```

Item

```
Robot
```

Index

`int` The index of the robot fixed at 0

Attribute

Name	Type	Mode	Description	Format
CoordRobot	float[]	R	The TCP coordinate of the robot end point opposite to the RobotBaseof the robot	{X, Y, Z, RX, RY, RZ}, Size = 6
CoordBase	float[]	R	The TCP coordinate of the robot end point opposite to the current base fo the robot.	{X, Y, Z, RX, RY, RZ}, Size = 6
Joint	float[]	R	The current robot joint angle	{J1, J2, J3, J4, J5, J6}, Size = 6
BaseName	string	R	The name of the current base	"Base Name"
TCPName	string	R	The name of the current TCP	"TCP Name"
CameraLight	byte	R/W	The lighting of the robot's camera	0: Low (Off), 1: High (On)
TCPForce3D	float	R	The current TCP force as the composite force of the robot base x, y, and z.	N

TCPSpeed3D	float	R	The current TCP speed as a composite speed of the robot base x, y, and z.	mm/s

Note

```
// Read values
float[] var_rtool = Robot[0].CoordRobot     // Obtain the current TCP coordinate of the robot end point opposite to the RobotBaseof the robot
float[] var_ftool = Robot[0].CoordBase      // Obtain the current TCP coordinate of the robot end point opposite to the current base fo the robot.
float var_f = Robot[0]. CoordBase[0]        // or retrieve the X value of the current TCP coordinate of the robot end point opposite to the current base fo the robot solely.
var_f = Robot[0]. CoordBase[6]              // Return error, exceeding the array's access range
float[] var_joint = Robot[0].Joint          // Obtain the current robot joint angle
float var_j = Robot[0].Joint[0]             // or retrieve the current angle of the robot's 1st joint solely
string var_b = Robot[0].BaseName            // var_b = "RobotBase"
string var_t = Robot[0].TCPName             // var_t = "NOTOOL"
byte var_light = Robot[0].CameraLight       // var_light = 0 (OFF)
float var_tf3d = Robot[0].TCPForce3D        // var_tf3d = 1.234
float var_ts3d = Robot[0].TCPSpeed3D        // var_ts3d = 1.234
// Write values
Robot[0].CoordRobot = {0, 90, 0, 0, 0, 0}   // Read only, invalid operation
Robot[0].CoordBase = {0, 0, 90, 0, 90, 0}   // Read only, invalid operation
Robot[0].BaseName = "Base1"                 // Read only, invalid operation
Robot[0].TCPName = "Tool1"                  // Read only, invalid operation
Robot[0].CameraLight = 1                    // Turn on the lighting of the robot's camera
Robot[0].CameraLight = 0                    // Turn off the lighting of the robot's camera
Robot[0].TCPSpeed3D = 1.234                 // Read only, invalid operation
```

6.7 FT

Syntax

```
FT[string].attribute
```

Item

FT	Force Torque sensor status

Index

string	The name of F/T sensor in the F/T sensor list

Attribute

Name	Type	Mode	Description	Format
X	float	R	The strength value of the X axis	
Y	float	R	The strength value of the y axis	
Z	float	R	The strength value of the z axis	
TX	float	R	The torque value of the X axis	
TY	float	R	The torque value of the y axis	
TZ	float	R	The torque value of the z axis	
F3D	float	R	The XYZ force strength value	
T3D	float	R	The XYZ force torque value	
ForceValue	float[]	R	The XYZ force strength value array	{X, Y, Z}, Size = 3
TorqueValue	float[]	R	The XYZ force torque value array	{TX, TY, TZ}, Size = 3
RefCoordX	float	R	The X-axis strength value measured based on the reference coordinate system set in the node	
RefCoordY	float	R	The Y-axis strength value measured based on the reference coordinate system set in the node	
RefCoordZ	float	R	The Z-axis strength value measured based on the reference coordinate system set in the node	
RefCoordTX	float	R	The X-axis torque value measured based on the reference coordinate system set in the node	
RefCoordTY	float	R	The Y-axis torque value measured based on the reference coordinate system set in the node	
RefCoordTZ	float	R	The Z-axis torque value measured based on the reference coordinate system set in the node	
RefCoordF3D	float	R	The XYZ strength measured based on the reference coordinate system set in the node	
RefCoordT3D	float	R	The XYZ torque measured based on the reference coordinate system set in the node	
RefCoordForceValue	float[]	R	The XYZ strength value matrix measured based on the reference coordinate system set in the node	{RefCoordX, RefCoordY, RefCoordZ}, Size = 3

RefCoordTorqueValue	float[]	R	The XYZ torque value matrix measured based on the reference coordinate system set in the node	{RefCoordTX, RefCoordTY, RefCoordTZ}, Size = 3
Model	string	R	The Model name of the F/T sensor	
Zero	byte	R/W	Turn on or off F/T sensor offset	0: Zero OFF, 1: Zero ON

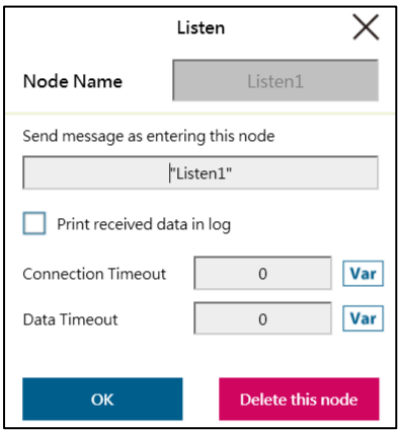
Note

```
// Read values
float var_x = FT[“fts1”].X                  // Obtain the current X-axis strength value of F/T sensor “fts1”
float var_tx = FT[“fts1”].TX                // Obtain the current X-axis torque value of F/T sensor “fts1”
float var_f3d = FT[“fts1”].F3D              // Obtain the current XYZ force value of F/T sensor “fts1”
float[] var_force = FT[“fts1”].ForceValue   // Obtain the current XYZ force strength value array of F/T sensor “fts1”
string var_mode = FT[“fts1”].Model          // Obtain the model name of F/T sensor “fts1”
// Write values
FT[“fts1”].Y = 3.14                         // Read only, invalid operation
FT[“fts1”].TY = 1.34                        // Read only, invalid operation
FT[“fts1”].T3D = 4.13                       // Read only, invalid operation
FT[“fts1”].TorqueValue = {1.1, 2.2, 3.3}    // Read only, invalid operation
FT[“fts1”].Zero = 1                         // Book the current offset of F/T sensor
```

7. External Script

7.1 Listen Node

Users can establish a socket TCPlistener (server site) in the listen node to connect to external devices and communicate based on the packet format. All features available in TM_Robot_Function can also be operated in the listen node.



- **Send Message:** When entering this node, it will initiate a message
- **Print Log:** Enable Communication Log (shown on the right)
- **Connection Timeout:** When entering this node, if more than the time (milliseconds) is not connected, it will be overtime.
  If <= 0, no timeout
- **Data Timeout:** When connected, the timeout will be exceeded when there is no communication packet
  If <= 0, no timeout

Socket TCPListener is started up after the project being executed, and closed as the project stopped. The IP and listen port will be shown on the Notice Log window on the right, after the Socket TCPListener is started up.

IP HMI → System → Network → IP Address

Port 5890

When entering the Listen Node, the flow will keep at Listen Node until either of the the two exit conditions is fulfilled.

Pass: *ScriptExit()* is executed or the project is stopped

Fail: 1. Connection Timeout

2. Data Timeout

3. Before the TCP Listener is started up, the flow has entered this Listen Node

The command received by listen node will be executed in order. If the command is not valid, an error message will be returned carrying the line number with errors. If the command is valid, it will be executed.

The command can be divided into two categories. The first category is commands which can be accomplished in instance, like assigning variable value. The second category is commands needs to be executed in sequence, like motion command and IO value assigning. The second category command will be placed in queue and executed in order.

7.2 ScriptExit()

Exit external script control mode.

Syntax 1

```
bool ScriptExit(
)
```

Parameters

`void` No parameter

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Exit the external script control mode and wait for the command to finish, and then quit the listen node and move on with the pass route.

* Execute via TMSCT communication packets

* Functions after ScriptExit() will not be executed such as

< $***TMSCT***,78,2,ChangeBase("RobotBase")\r\n

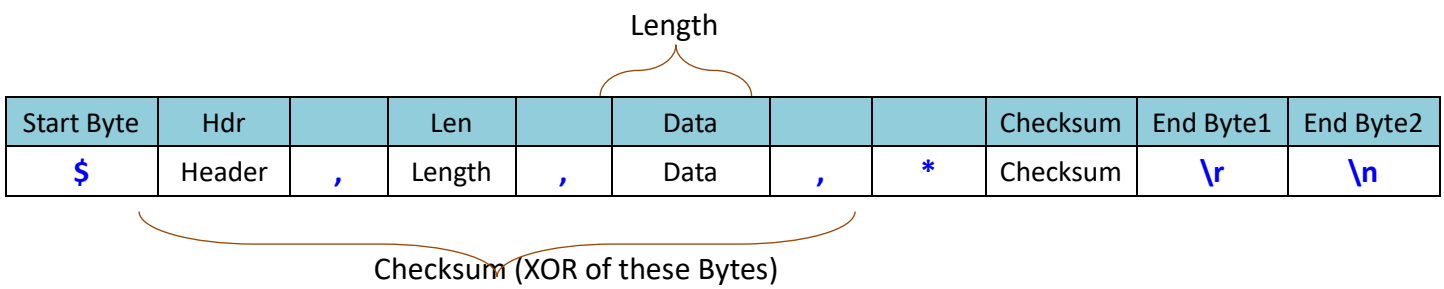
ChangeTCP("NOTOOL")\r\n

ScriptExit()\r\n // Exit the external script control mode.

ChangeLoad(10.1),*6C\r\n // ChangeLoad will not be executed.

* After exiting the script mode, it is required to wait for all the commands and the functions to complete executions until quitting the listen node and moving on with the pass route. At the time being of waiting for quitting the listen node, it is not in the external script control mode, so no more external commands will be accepted and CPEER error packets will be replied.

7.3 Communication Protocol



Start Byte	Hdr		Len		Data			Checksum	End Byte1	End Byte2
$	Header	,	Length	,	Data	,	*	Checksum	\r	\n

Name	Size	ASCII	HEX	Description
Start Byte	1	$	0x24	Start Byte for Communication

Header	*X*			Header for Communication
Separator	1	,	0x2C	Separator between Header and Length
Length	*Y*			Length of Data
Separator	1	,	0x2C	Separator between Length and Data
Data	*Z*			Communication Data
Separator	1	,	0x2C	Separator between Data and Checksum
Sign	1	*	0x2A	Begin Sign of Checksum
Checksum	*2*			Checksum of Communication
End Byte 1	1	\r	0x0D	
End Byte 2	1	\n	0x0A	End Byte of Communication

1. ***Header***

   Defines the purpose of the communication package. The data definition could be different with different Header.

   - *TMSCT* External Script
   - *TMSTA* Acquiring status or properties
   - *CPERR* Communication data error (E.g. Packet error, checksum error, header error, etc.)

2. ***Length***

   Length defines the length in UTF8 byte. It can be represented in decimal, hexadecimal or binary, the upper limit is int 32bits

   Example:

   ```
   $TMSCT,100,Data,*CS\r\n      // Decimal 100, that is the data length is 100 bytes
   $TMSCT,0x100,Data,*CS\r\n    // Hexadecimal 0x100, that is the data length is 256 bytes
   $TMSCT,0b100,Data,*CS\r\n    // Binary 0b100, that is the data length is 4 bytes
   $TMSCT,8,1,達明,*58\r\n      // The Data length  1,達明  is 8 bytes (UTF8)
   ```

3. ***Data***

   The content of the communication package. Arbitrary characters are supported (including 0x00 .. 0xFF in UTF8).

   The data length is defined in Length and the purpose is defined in Header

4. ***Checksum***

   The checksum of the communication package. The checksum is calculated with XOR(exclusive OR), and the range for checksum computation starts from $ to * ($ and * are excluded) as shown below:

   $TMSCT,100,Data,*CS\r\n

   Checksum = Byte[1] ^ Byte[2] … ^ Byte[N-6]

   The representation of checksum is fixed to 2 bytes in hexadecimal format (without 0x).

   For example:

$TMSCT,5,10,OK,*6D

CS = 0x54 ^ 0x4D ^ 0x53 ^ 0x43 ^ 0x54 ^ 0x2C ^ 0x35 ^ 0x2C ^ 0x31 ^ 0x30 ^ 0x2C ^ 0x4F ^ 0x4B ^ 0x2C = 0x6D

CS = 6D (0x36 0x44)

7.4 TMSCT

Start Byte	Hdr		Len		Data			Checksum	End Byte1	End Byte2
$	*TMSCT*	,	Length	,	Data	,	*	Checksum	**\r**	**\n**

ID		SCRIPT
Script ID	,	Script Language

TMSCT defines the communication package as External Script Language. In External Script Language, the data contains two parts and is separated by comma. One is ID and the other is SCRIPT

`ID` Script ID, can be arbitrary English character or number (a CPERR 04 error will be reported when encountering non-alphanumeric byte). The ID is used as specifying the target SCRIPT of return message.

`,` Separator

`SCRIPT` The content defined in Script Language. In a communication package, multi-line scripts can fit into the SCRIPT section with separator (0x0D 0x0A)

Note

TMSCT is available only when in the external script control mode, otherwise CPEER error packets will be replied.

Return (Robot→External Device)

1. When it enters Listen Node, the robot will send a message to all the connected device. The ID is set to 0.

   $***TMSCT***,9,0,Listen1,*4C\r\n

   9 The length of *0,Listen1* is 9 bytes

   0 The Script ID is 0

   , Separator

   Listen1 The message to send

2. The OK or ERROR message is replied according to the Script's content. For message with ;N, ;N represents the number of line with error or warning. After the message is received, robot will execute the message, then send back the return message, if the Script is valid. For invalid Script, the return

message will be sent back immediately without executed.

```
$TMSCT,4,1,OK,*5C\r\n                // Response to ID 1
                                     // OK means valid Script.
$TMSCT,8,2,OK;2;3,*52\r\n            // Response to ID 2
                                     // OK;2;3 means valid Script with warnings in line 2 and 3.
$TMSCT,13,3,ERROR;1;2;3,*3F\r\n      // Response to ID 3
                                     // ERROR;1;2;3 means invalid Script with errors in line 1, 2 and 3.
```

Receive (Robot←External Device)

1. When it enters the listen node, the robot will start to receive, check, and execute the external script. If the robot did not enter the listen node (not in the external script control mode), the Script received will be disposed and CPEER error packets will be replied.
2. The message from external device should define the Script ID as a ID used in messages replied by robot.

```
<   $TMSCT,25,1,ChangeBase("RobotBase"),*08\r\n      // Defined as ID 1
>   $TMSCT,4,1,OK,*5C\r\n                            // Response to ID 1
```

3. In a communication package, multi-line scripts can fit into the SCRIPT section with separator \r\n

```
<   $TMSCT,64,2,ChangeBase("RobotBase")\r\n
    ChangeTCP("NOTOOL")\r\n
    ChangeLoad(10.1),*68\r\n      // Three lines Script in a communication package (Lines are
    separated by \r\n)
>   $TMSCT,4,2,OK,*5F\r\n
```

4. In Listen Node, local variables are supported and valid before quitting the Listen Node.

```
<   $TMSCT,40,3,int var_i = 100\r\n
    var_i = 1000\r\n
    var_i++,*5A\r\n
>   $TMSCT,4,3,OK,*5E\r\n

<   $TMSCT,42,4,int var_i = 100\r\n
    var_i = 1000\r\n
    var_i++\r\n
    ,*58\r\n
>   $TMSCT,9,4,ERROR;1,*02\r\n      // Because int var_i has been declared, an error occurred.
```

5. In the listen node, it is possible to access or modify the project's variables, but no new variable can be declared since the variables created in the listen node are local variables.

7.5 TMSTA

Start Byte	Hdr		Len		Data			Checksum	End Byte1	End Byte2
$	*TMSTA*	**,**	Length	**,**	Data	**,**	*	Checksum	**\r**	**\n**

SubCmd		
SubCmd	…	…

(Based on SubCmd)

TMSTA defines the communication package as acquiring status or properties. The data section of the package contains different sub command (SubCMD). The package format could be different according to different SubCMD. The definitions are listed below.

`SubCmd`

`00` In external script control mode or not

`01` Complete the configured QueueTag numbering or not

`90..99` Date message to send (the format of data is self-definable)

Note

TMSTA could be executed without entering the Listen Node

SubCmd 00 In external script control mode or not

Format

Response (Robot→External Device)

SubCmd		Entry		Message
00	,	false	,	
00	,	true	,	message

Receive (Robot←External Device)

SubCmd
00

Response (Robot→External Device)

1. If not in external script control mode, it will reply false.

$***TMSTA***,9,00,false,,*37\r\n

9 Indicates the length of 00,false, is 9 bytes

00	Indicates SubCmd as 00
,	Separator
false	The flow has not entered Listen Node
,	Separator
	Empty string (Have not entered Listen Node)

2. If in external script control mode, it will reply true.

$***TMSTA***,15,00,true,Listen1,*79\r\n

15	Indicates the length of 00,true,Listen1 is 15 bytes
00	Indicates SubCmd as 00
,	Separator
true	The flow has entered the Listen Node
,	Separator
Listen1	The message to be sent as in Listen Node (It indicates the flow is in Listen1)

Receive (Robot←External Device)

1. Send to the robot from the external device

$***TMSTA***,2,00,*41\r\n

2	Indicates the length of 00 is 2 bytes.
00	Indicates the SubCmd is 00 whether in external script control mode or not.

SubCmd 01 Complete the configured QueueTag numbering or not

Format

Send (Robot→External Device)

SubCmd		Tag Number		Status
01	,	01 .. 15	,	true/false/none

Receive (Robot←External Device)

SubCmd		Tag Number
01	,	01 .. 15

Note

When inquiring with TMSTA 01, users can look up to the status of the last 4 tag numbers.

Send (Robot→External Device)

1. Send to the external device from the robot. Spontaneously sending after QueueTag numbering

completed.

$***TMSTA***,10,01,08,true,*6D\r\n

10		Indicates the length of 01,00,true is 10 bytes
01		Indicates SubCmd as 01 to send the status of Tag Number
,		Separation symbol
08		Tag Number 08
,		Separation symbol
true	*true*	Indicates Tag Number complete
	false	Indicates Tag Number incomplete
	none	Indicates Tag Number not existed

Receive (Robot←External Device)

1. Send from the external device to the robot. Users can look up to the status of the last 4 tag numbers.

   $***TMSTA***,5,01,15,*6F\r\n

5	Indicates the length of 01,88 is 5 bytes
01	Indicates SubCmd as 01 to send the status of Tag Number
,	Separation symbol
15	Tag Number 15

   > $***TMSTA***,10,01,15,none,*7D\r\n // TagNumber 15 not existed

2. Tag Number uses the value of integers between 1 and 15. If the value is invalid, it relies none for not existed.

   $***TMSTA***,5,01,88,*6B\r\n

   > $***TMSTA***,10,01,88,none,*79\r\n // TagNumber 88 not existed

SubCmd 90 .. 99 Send data message

Format

Send (Robot→External Device)

SubCmd		Data
90 .. 99	,	…

Receive (Robot←External Device)

None

Note

1. When sending with TMSTA 90 .. 99, users can use their self-defined formats.
2. Self-defined formats denote the formats are defined by both the project flow and the external device.

3. To enhance the flexibility of usages, users can various SubCmd of 90 .. 99 to define different formats to send such as

   SubCmd 90 defined as string；

   SubCmd 91 defined as float[]；

   SubCmd 92 defined as byte[]

   …

   and so on for the external device to analyze and resolve based on the SubCmd with different methods.

Send (Robot→External Device)

1. Send to the external device from the robot. When the external script executes the ListenSend() function, it will send data.

   string var_s = “Hello World”

   float[] var_f = {1,2,3,4}

   byte[] var_b = {0x10, 0x11, 0x12, 0x13}

   ListenSend(90, var_s)

   // the content of communication $TMSTA,14,90,Hello World,*73\r\n

   // 0x39,0x30,0x2C,0x48,0x65,0x6C,0x6C,0x6F,0x20,0x57,0x6F,0x72,0x6C,0x64

   ListenSend(91, var_f)

   // the content of communication $TMSTA,19,91,…,*60\r\n

   // 0x39,0x31,0x2C,0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40,0x00,0x00,0x80,0x40

   ListenSend(92, var_b)

   // the content of communication $TMSTA,7,92,…,*63\r\n

   // 0x39,0x32,0x2C,0x10,0x11,0x12,0x13

7.6 CPERR

Start Byte	Hdr		Len		Data			Checksum	End Byte1	End Byte2
$	*CPERR*	**,**	Length	**,**	Data	**,**	*****	Checksum	**\r**	**\n**

Error Code
Code (00 .. FF)

CPERR defines the communication package as sending the Communication Protocol Error. The data section is defined as Error Code.

`Error Code` Error code, presented in 2 bytes hexadecimal format (without 0x)

00	Packet correct. No error. (The return message usually reply to the content of packet instead of returning no error)
01	Packet Error.
02	Checksum Error.
03	Header Error.
04	Packet Data Error.
F1	Have not entered Listen Node

Note

Used by robot to response to external device

Response (Robot→External Device)

01 Packet Error

< $***TMSCT***,**-100**,1,ChangeBase("RobotBase"),*13\r\n // Length cannot be negative

> $***CPERR***,2,01,*49\r\n // CPERR Error Code 01

02 Checksum Error

< $***TMSCT***,25,1,ChangeBase("RobotBase"),***09**\r\n // 09 is not a correct Checksum

> $***CPERR***,2,02,*4A\r\n // CPERR Error Code 02

03 Header Error

< $***TMsct***,25,1,ChangeBase("RobotBase"),*28\r\n // TMsct is not a correct Header

> $***CPERR***,2,03,*4B\r\n // CPERR Error Code 03

04 Packet Data Error

< $***TMSTA***,4,XXXX,*47\r\n // There is no XXXX SubCmd under TMSTA

> $***CPERR***,2,04,*4C\r\n // CPERR Error Code 04

F1 No External Script Mode

< $***TMSCT***,25,1,ChangeBase("RobotBase"),*0D\r\n

// Suppose currently not in external script control mode

> $***CPERR***,2,F1,*3F\r\n // CPERR Error Code F1

8. Robot Motion Functions

Robot Motion Functions can only be performed with external scripts, meaning the project flow must be in the listen node and the $TMSCT header must be used. All the motion functions will be queued in the buffer and performed in sequence.

8.1 QueueTag()

Set robot motions with Queue Tag Numbers to denote the current robot motion in process. The status of each queue tag can be monitored using TMSTA SubCmd 01.

Syntax 1

```
bool QueueTag(
    int,
    int
)
```

Parameters

`int`	The tag number. Valid for integers between 1 and 15.	
`int`	Wait for the tagging to continue processing or not.	
	`0`	Not wait (default)
	`1`	Wait
	When the value is set to 1, the process stays in the function and waits for the tagging to complete and continue processing.	

Return

`bool`	Return `True` when tagged successfully. Return `False` when tagged unsuccessfully.

Syntax 2

```
bool QueueTag(
    int
)
```

Note

The syntax is the same as syntax 1. The default is not to wait for the tagging to continue processing.

QueueTag(int, int) => **QueueTag**(int, 0)

8.2 WaitQueueTag()

Wait for the Queue Tag Number of the robot motion to complete.

Syntax 1

```
int WaitQueueTag(
    int,
    int
)
```

Parameters

int	The tag number. Valid for integers between 1 and 15.	
int	Set the time to the timeout	
	<= 0	No timeout (default)
	> 0	Wait in milliseconds before timeout
	When the value is set to larger than 0, the process stays in the function until the tagging is completed, the tagging is not existed, or timeout, and then continues processing.	

Return

int	Return the result of waiting	
	1	The tagging is completed
	0	The tagging is incomplete or timeout
	-1	The tagging is not existed

Syntax 2

```
int WaitQueueTag(
    int
)
```

Note

The syntax is the same as syntax 1. The default is no timeout and required to wait for the tagging to complete (or not existed)

WaitQueueTag(int, int) => **WaitQueueTag**(int, 0)

- **Motion Function Queue Tag**

Motion function queue tags are used to cooperate with the robot motion functions. Since all motion functions are queued in the buffer and executed in order, use the cooperative queue tags, it is possible to know which motion function is in execution currently.

1. < $***TMSCT***,172,2,float[] targetP1= {0,0,90,0,90,0}\r\n
PTP("JPP",targetP1,10,200,0,false)\r\n
QueueTag(1)\r\n // QueueTag(1) not wait and continue processing
float[] targetP2 = {0,90,0,90,0,0}\r\n
PTP("JPP",targetP2,10,200,10,false)\r\n
QueueTag(2)\r\n // QueueTag(2) not wait and continue processing
,*49\r\n

When executed the script content, since QueueTag() did not wait, after execution, the process returned

> $***TMSCT***,4,2,OK,*5F\r\n

When robot motion executed PTP() targetP1, because of QueueTag(1), it will return

> $***TMSTA***,10,01,01,true,*64\r\n // TMSTA SubCmd 01, TagNumber 01, completed

When robot motion executed PTP() targetP2, because of QueueTag(2) , it will return

> $***TMSTA***,10,01,02,true,*67\r\n // TMSTA SubCmd 01, TagNumber 02, completed

2. < $***TMSCT***,174,2,float[] targetP1= {0,0,90,0,90,0}\r\n
PTP("JPP",targetP1,10,200,0,false)\r\n
QueueTag(3,1)\r\n // QueueTag(3) wait and stay in the function until the tagging completed
float[] targetP2 = {0,90,0,90,0,0}\r\n
PTP("JPP",targetP2,10,200,10,false)\r\n
QueueTag(4)\r\n // QueueTag(4) not wait and continue processing
,*56\r\n

When executed the script content, since QueueTag(3,1) is set to wait, after tagging completed, the process returned

> $***TMSTA***,10,01,03,true,*66\r\n // TMSTA SubCmd 01, TagNumber 03, completed

When QueueTag(3) completed, the process continues, since QueueTag(4) is not set to wait, after execution, the process returned

> $***TMSCT***,4,2,OK,*5F\r\n

When robot motion executed PTP() targetP2, because of QueueTag(4) , it will return

> $*TMSTA*,10,01,04,true,*61\r\n // TMSTA SubCmd 01, TagNumber 04, completed

* $TMSCT,4,2,OK is returned when the process executed the script. Therefore, if using QueueTag to wait or WaitQueueTag to wait, it will return after the execution as well.

8.3 StopAndClearBuffer()

Stop the motion of the robot and clear existing commands of the robot in the buffer.

Syntax

```
bool StopAndClearBuffer(
)
```

Parameter

`void` No input values required

Return

`bool` `True` Command accepted； `False` Command rejected

Note

StopAndClearBuffer()

8.4 Pause()

Pause the project and the motion of the robot other than non-paused threads and external script. Use Resume() or press the Play button on the robot stick to resume.

Syntax1

```
bool Pause(
)
```

Parameter

`void` No input values required

Return

`bool` `True` Command accepted； `False` Command rejected

Note

Pause()

8.5 Resume()

Resume the project and the motion of the robot.

Syntax1

```
bool Resume(
)
```

Parameter

`void` No input values required

Return

`bool` `True` Command accepted； `False` Command rejected

Note

Resume()

8.6 PTP()

Define and send PTP motion command into buffer for execution.

Syntax 1

```
bool PTP(
    string,
    float[],
    int,
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

`"J"` expressed in joint angles

`"C"` expressed in Cartesian coordinate

#2: Speed format:

`"P"` expressed as a percentage

#3: Blending format

`"P"` expressed as a percentage

`float[]` Motion target degree. If defined with joint angle, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)；If defined with Cartesian coordinate, it

includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable precise positioning

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Data format parameter includes: (1) `"JPP"`, (2) `"CPP"`

```
float[] var_targetP1= {0,0,90,0,90,0}          // Declare a float array to store the target coordinate
PTP("JPP", var_targetP1,10,200,0,false)        // Move to var_targetP1 with PTP, speed = 10%, time to top speed = 200ms.
```

Syntax 2

```
bool PTP(
    string,
    float[],
    int,
    int,
    int,
    bool,
    int[]
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

`"C"` expressed in Cartesian coordinate

#2: Speed format:

`"P"` expressed as a percentage

#3: Blending format

`"P"` expressed as a percentage

`float[]` Motion target. If defined with Cartesian coordinate, it includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable precise positioning

`int[]` The pose of robot : [Config1, Config2, Config3], please find more information in appendix

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Data format parameter includes: (1) `"CPP"`

```
float[] var_targetP1 = {417.50,-122.30,343.90,180.00,0.00,90.00}
        // Declare a float array to store the target coordinate.
float[] var_pose = {0,2,4}                          // Declare a float array to store pose.
PTP("CPP", var_targetP1,50,200,0,false, var_pose)   // Move to var_targetP1 with PTP, speed =
                                                    50%, time to top speed = 200ms.
```

Syntax 3

```
bool PTP(
    string,
    float, float, float, float, float, float,
    int,
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

`"J"` expressed in joint angles

`"C"` expressed in Cartesian coordinate

#2: Speed format:

`"P"` expressed as a percentage

#3: Blending format:

`"P"` expressed as a percentage

`float, float, float, float, float, float`

Motion target. If expressed in joint angles, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)；If expressed in Cartesian coordinate, it includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable precise positioning

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Data format parameter includes: (1) `"JPP"` and (2) `"CPP"`

PTP("JPP",0,0,90,0,90,0,35,200,0,false) // Move to joint angle 0,0,90,0,90,0 with PTP, speed = 35%, time to top speed = 200ms.

Syntax 4

```
bool PTP(
    string,
    float, float, float, float, float, float,
    int,
    int,
    int,
    bool,
    int, int, int
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

`"C"` expressed in Cartesian coordinate

#2: Speed format:

`"P"` expressed as a percentage

#3: Blending format:

`"P"` expressed as a percentage

`float, float, float, float, float, float`

Motion target. It includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable precise positioning

`int, int, int`

The pose of robot : Config1, Config2, Config3, please find more information in appendix

Return

`bool` True Command accepted； False Command rejected (format error)

Note

Data format parameter includes: (1) `"CPP"`

PTP("CPP",417.50,-122.30,343.90,180.00,0.00,90.00,10,200,0,false,0,2,4) // Move to coordinate 417.50,-122.30,343.90,180.00,0.00,90.00, with PTP, speed = 10%, time to top speed = 200ms, pose = 024.

8.7 Line()

Define and send Line motion command into buffer for execution.

Syntax 1

```
bool Line(
    string,
    float[],
    int,
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

`"C"` expressed in Cartesian coordinate

#2: Speed format:

`"P"` expressed as a percentage

`"A"` expressed in velocity (mm/s)

#3: Blending format:

`"P"` expressed as a percentage

`"R"` expressed in radius

`float[]` Motion target. It includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%) or in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%) or in radius (mm)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable precise positioning

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Data format parameter includes: (1) "CPP", (2) "CPR", (3) "CAP", and (4) "CAR"

```
float[] var_Point1 = {417.50,-122.30,343.90,180.00,0.00,90.00}
        // Declare a float array to store the target coordinate
Line("CAR", var_Point1,100,200,50,false)        // Move to var_Point1 with Line, speed = 100mm/s, time to top speed = 200ms, blending radius = 50mm
```

Syntax 2

```
bool Line(
    string,
    float, float, float, float, float, float,
    int,
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

"C" expressed in Cartesian coordinate

#2: Speed format:

"P" expressed as a percentage

"A" expressed in velocity (mm/s)

#3: Blending format:

"P" expressed as a percentage

"R" expressed in radius

`float, float, float, float, float, float`

Motion target. It includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%) or in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%) or in radius (mm)

bool Disable precise positioning

true Disable precise positioning

false Enable precise positioning

Return

bool True Command accepted; False Command rejected (format error)

Note

Data format parameter includes: (1) "CPP", (2) "CPR", (3) "CAP", and (4) "CAR"

Line("CAR", 417.50,-122.30,343.90,180.00,0.00,90.00,100,200,50,false) // Move to 417.50,-122.30,343.90,180.00,0.00,90.00 with Line, velocity = 100mm/s, time to top speed = 200ms, blending radius = 50mm

8.8 Circle()

Define and send Circle motion command into buffer for execution.

Syntax 1

```
bool Circle(
    string,
    float[],
    float[],
    int,
    int,
    int,
    int,
    bool
)
```

Parameters

string Definition of data format, combines three letters

#1: Motion target format:

"C" expressed in Cartesian coordinate

#2: Speed format:

"P" expressed as a percentage

"A" expressed in velocity (mm/s)

#3: Blending format:

"P" expressed as a percentage

float[] A point on arc. It includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`float[]` The end point of arc, it includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%) or in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`int` Arc angle(°), If non-zero value is given, the TCP will keep the same pose and move from current point to the assigned arc angle via the given point and end point on arc; If zero is given, the TCP will move from current point and pose to end point and pose via the point on arc with linear interpolation on pose.

`bool` Disable precise positioning

- `true` Disable precise positioning
- `false` Enable precise positioning

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Data format parameter includes: (1) `"CPP"` and (2) `"CAP"`

```
float[] var_PassP = {417.50,-122.30,343.90,180.00,0.00,90.00}   // Declare a float array to store the via point value
float[] var_EndP = {381.70,208.74,343.90,180.00,0.00,135.00}    // Declare a float array to store the end point value
Circle("CAP", var_PassP, var_EndP,100,200,50,270,false)         // Move on 270° arc, velocity = 100mm/s, time to top speed = 200ms, blending value = 50%
```

Syntax 2

```
bool Circle(
    string,
    float, float, float, float, float, float,
    float, float, float, float, float, float,
    int,
    int,
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

"C" expressed in Cartesian coordinate

#2: Speed format:

"P" expressed as a percentage

"A" expressed in velocity (mm/s)

#3: Blending format:

"P" expressed as a percentage

`float, float, float, float, float, float`

A point on arc. It includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`float, float, float, float, float, float`

The end point of arc. It includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed as a percentage (%) or in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`int` Arc angle(°), If non-zero value is given, the TCP will keep the same pose and move from current point to the assigned arc angle via the given point and end point on arc; If zero is given, the TCP will move from current point and pose to end point and pose via the point on arc with linear interpolation on pose.

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable precise positioning

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Data format parameter includes: (1) "CPP" and (2) "CAP"

Circle("CAP", 417.50,-122.30,343.90,180.00,0.00,90.00,
381.70,208.74,343.90,180.00,0.00,135.00,100,200,50,270,false)
// Move on 270° arc, velocity = 100mm/s, time to top speed = 200ms, blending value = 50%, via point = 417.50,-122.30,343.90,180.00,0.00,90.00, end point = 381.70,208.74,343.90,180.00,0.00,135

8.9 PLine()

Define and send PLine motion command into buffer for execution.

Syntax 1

```
bool PLine(
    string,
    float[],
    int,
    int,
    int
)
```

Parameters

`string`	Definition of data format, combines three letters #1: Motion target format: "J": expressed in joint angles "C": expressed in Cartesian coordinate #2: Speed format: "A": expressed in velocity (mm/s) #3: Blending format: "P": expressed as a percentage
`float[]`	Motion target. If expressed in joint angles, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)；If expressed in Cartesian coordinate, it includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)
`int`	The speed setting, expressed in velocity (mm/s)
`int`	The time interval to accelerate to top speed (ms)
`int`	Blending value, expressed as a percentage (%)

Return

`bool`	True Command accepted； False Command rejected (format error)

Note

Data format parameter includes: (1) "JAP" and (2) "CAP"

```
float[] targetP1 = {417.50,-122.30,343.90,180.00,0.00,90.00}    // Declare a float array to store the target coordinate
PLine("CAP",targetP1,100,200,50)                                // Move to targetP1 with PLine, velocity = 100mm/s, time to top speed = 200ms, blending value = 50%
```

Syntax 2

```
bool PLine(
    string,
    float, float, float, float, float, float,
    int,
    int,
    int
```

)

Parameters

`string` Definition of data format, combines three letters

#1: Motion target format:

"J": expressed in joint angles

"C": expressed in Cartesian coordinate

#2: Speed format:

"A": expressed in velocity (mm/s)

#3: Blending format:

"P": expressed as a percentage

`float, float, float, float, float, float,`

Motion target. If expressed in joint angles, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)；If expressed in Cartesian coordinate, it includes the Cartesian coordinate of tool center point: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°)

`int` The speed setting, expressed in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Data format parameter includes: (1) "JAP" and (2) "CAP"

PLine("CAP", 417.50,-122.30,343.90,180.00,0.00,90.00,100,200,50)

// Move to 417.50,-122.30,343.90,180.00,0.00,90.00 with PLine, velocity = 100mm/s, time to top speed = 200ms, Blending value = 50%

8.10 Move_PTP()

Define and send PTP relative motion commands for execution.

Syntax 1

```
bool Move_PTP(
    string,
    float[],
    int,
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format made of three letters

#1: Relative motion target format:

"C": expressed w.r.t. current base

"T": expressed w.r.t. tool coordinate

"J": expressed in joint angles

#2: Speed format:

"P": expressed as a percentage

#3: Blending format:

"P": expressed as a percentage

`float[]` relative motion parameters. If expressed in coordinate (w.r.t. current base or tool coordinate), it includes the tool end TCP relative motion value with respect to the specified coordinate: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°); If defined with joint angle, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)

`int` The speed setting, expressed as a percentage (%)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Motion command parameter includes: (1) "CPP", (2) "TPP" or (3) "JPP"

```
float[] var_relmove = {0,0,10,45,0,0}          // Declare a float array to store the relative motion target
Move_PTP("TPP", var_relmove,10,200,0,false)    // Move to relative motion target with PTP, velocity = 10%, time to top speed = 200ms
```

Syntax 2

```
bool Move_PTP(
    string,
    float, float, float, float, float, float,
    int,
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format, combines three letters

#1: Relative motion target format:

"C": expressed w.r.t. current base

"T": expressed w.r.t. tool coordinate

"J": expressed in joint angles

#2: Speed format:

"P": expressed as a percentage

#3: Blending format:

"P": expressed as a percentage

`float, float, float, float, float, float`

relative motion parameters. If expressed in coordinate (w.r.t. current base or tool coordinate), it includes the tool end TCP relative motion value with respect to the specified coordinate: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°); If defined with joint angle, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)

`int` The speed setting, expressed as a percentage (%)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Motion command parameter includes: (1) "CPP", (2) "TPP" and (3) "JPP"

Move_PTP("TPP",0,0,10,45,0,0,10,200,0,false) // Move 0,0,10,45,0,0, with respect to tool coordinate, with PTP, velocity = 10%, time to top speed = 200ms

8.11 Move_Line()

Define and send Line relative motion commands for execution.

Syntax 1

```
bool Move_Line(
    string,
    float[],
    int,
```

```
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format made of three letters

#1: Relative motion target format:

"C": expressed w.r.t. current base

"T": expressed w.r.t. tool coordinate

#2: Speed format:

"P": expressed as a percentage

"A": expressed in velocity (mm/s)

#3: Blending format:

"P": expressed as a percentage

"R": expressed in radius

`float[]` Relative motion parameter. It includes the tool end TCP relative motion value with respect to the specified coordinate (current base or tool coordinate): X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°).

`int` The speed setting, expressed as a percentage (%) or in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%) or in radius (mm)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Motion command parameter includes: (1) "CPP", (2) "CPR", (3) "CAP", (4) "CAR", (5) "TPP", (6) "TPR", (7) "TAP" and (8) "TAR"

```
float[] var_relmove = {0,0,10,45,0,0}          //Declare a float array to store the relative motion target
Move_Line("TAP", var_relmove,125,200,0,false)  // Move to relative motion target, with Line, velocity = 125mm/s, time to top speed = 200ms
```

Syntax 2

```
bool Move_Line(
    string,
    float, float, float, float, float, float,
    int,
```

```
    int,
    int,
    bool
)
```

Parameters

`string` Definition of data format made of three letters

#1: Relative motion target format:

"C": expressed w.r.t. current base

"T": expressed w.r.t. tool coordinate

#2: Speed format:

"P": expressed as a percentage

"A": expressed in velocity (mm/s)

#3: Blending format:

"P": expressed as a percentage

"R": expressed in radius

`float, float, float, float, float, float`

Relative motion parameter. It includes the tool end TCP relative motion value with respect to the specified coordinate (current base or tool coordinate): X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°).

`int` The speed setting, expressed as a percentage (%) or in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%) or in radius (mm)

`bool` Disable precise positioning

`true` Disable precise positioning

`false` Enable

Return

`bool` True Command accepted； False Command rejected (format error)

Note

Motion command parameter includes: (1) "CPP", (2) "CPR", (3) "CAP", (4) "CAR", (5) "TPP", (6) "TPR", (7) "TAP" and (8) "TAR"

```
Move_Line("TAP", 0,0,10,45,0,0,125,200,0,false)    // Move to relative motion target 0,0,10,45,0,0 with Line, velocity = 125mm/s, time to top speed = 200ms.
```

8.12 Move_PLine()

Define and send PLine relative motion commands for execution.

Syntax 1

```
bool Move_PLine(
    string,
    float[],
    int,
    int,
    int
)
```

Parameters

`string` Definition of data format made of three letters

#1: Relative motion target format:

"C": expressed w.r.t. current base

"T": expressed w.r.t. tool coordinate

"J": expressed in joint angles

#2: Speed format:

"A": expressed in velocity (mm/s)

#3: Blending format:

"P": expressed as a percentage

`float[]` If expressed in coordinate (w.r.t. current base or tool coordinate), it includes the tool end TCP relative motion value with respect to the specified coordinate: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°); If defined with joint angle, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)

`int` The speed setting, expressed in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

Motion command parameter includes: (1) "CAP", (2) "TAP" and (3) "JAP"

```
float[] var_target = {0,0,10,45,0,0}          // Declare a float array to store the relative motion target
Move_PLine("CAP", var_target,125,200,0)       //Move to relative motion target, with PLine, velocity = 125mm/s, time to top speed = 200ms.
```

Syntax 2

```
bool Move_PLine(
    string,
    float, float, float, float, float, float,
```

```
    int,
    int,
    int,
)
```

Parameters

`string` Definition of data format made of three letters

#1: Relative motion target format:

"C": expressed w.r.t. current base

"T": expressed w.r.t. tool coordinate

"J": expressed in joint angles

#2: Speed format:

"A": expressed in velocity (mm/s)

#3: Blending format:

"P": expressed as a percentage

`float, float, float, float, float, float`

Relative motion parameters. If expressed in coordinate (w.r.t. current base or tool coordinate), it includes the tool end TCP relative motion value with respect to the specified coordinate: X (mm), Y (mm), Z (mm), RX(°), RY(°), RZ(°); If defined with joint angle, it includes the angles of six joints: Joint1(°), Joint 2(°), Joint 3(°), Joint 4(°), Joint 5(°), Joint 6(°)

`int` The speed setting, expressed in velocity (mm/s)

`int` The time interval to accelerate to top speed (ms)

`int` Blending value, expressed as a percentage (%)

Return

`bool` True Command accepted； False Command rejected (format error)

Note

Motion command parameter includes: (1) "CAP", (2) "TAP" and (3) "JAP"

```
Move_PLine("CAP",0,0,10,45,0,0,125,200,0)        // Move 0,0,10,45,0,0, with PLine, velocity = 125mm/s, time to top speed = 200ms
```

8.13 ChangeBase()

Send the command of changing the base of the follow-up motions into buffer for execution.

Syntax 1

```
bool ChangeBase(
    string
)
```

Parameters

`string` Base Name

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

ChangeBase(”RobotBase”) // Change the base to ”RobotBase”, a base listed on the base list in TMflow.

Syntax 2

```
bool ChangeBase(
    float[]
)
```

Parameters

`float[]` Base parameters, combines X, Y, Z, RX, RY, RZ

Return

`True` Command accepted; `False` Command rejected (format error)

Note

```
float[] var_Base1 = {20,30,10,0,0,90}        // Declare a float array to store the base value
ChangeBase(var_Base1)                        // Change the base value to var_Base1
```

Syntax 3

```
bool ChangeBase(
    float, float, float, float, float, float
)
```

Parameters

`float, float, float, float, float, float`

Base parameters, combines X, Y, Z, RX, RY, RZ

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

ChangeBase(20,30,10,0,0,90) // Change the base value to {20,30,10,0,0,90}

8.14 ChangeTCP()

Send the command of changing the TCP of the follow-up motions into buffer for execution.

Syntax 1

```
bool ChangeTCP(
    string
)
```

Parameters

`string` TCP name

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

ChangeTCP(”NOTOOL”) // Change the TCP to ”NOTOOL”, a TCP listed on the base list in TMflow.

Syntax 2

```
bool ChangeTCP(
    float[]
)
```

Parameters

`float[]` TCP Parameter, combines X, Y, Z, RX, RY, RZ

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

```
float[] var_Tool1 = {0,0,150,0,0,90}        // Declare a float array to store the TCP value
ChangeTCP(var_Tool1)                        // Change the TCP value to var_Tool1
```

Syntax 3

```
bool ChangeTCP(
    float[],
    float
)
```

Parameters

`float[]` TCP Parameter, combines X, Y, Z, RX, RY, RZ

`float` Tool’s weight

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

```
float[] var_Tool1 = {0,0,150,0,0,90}        // Declare a float array to store the TCP value
ChangeTCP(var_Tool1,2)                      // Change the TCP value to var_Tool1 with weight = 2kg
```

Syntax 4

```
bool ChangeTCP(
    float[],
    float,
    float[]
)
```

Parameters

`float[]` TCP Parameter, combines X, Y, Z, RX, RY, RZ

`float` Tool's weight

`float[]` Tool's moment of inertia: (1)Ixx, (2)Iyy, (3)Izz and its frame of reference: (4)X, (5)Y, (6)Z, (7)RX, (8)RY, (9)RZ

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

```
float[] var_Tool1 = {0,0,150,0,0,90}                    // Declare a float array to store the TCP value
float[] var_COM1 = {2,0.5,0.5,0,0,-80,0,0,0}            // Declare a float array to store the moment of inertia and its frame of reference
ChangeTCP(var_Tool1,2, var_COM1)                        // Change the TCP value to var_Tool1 with weight = 2kg and moment of inertia to var_COM1
```

Syntax 5

```
bool ChangeTCP(
    float, float, float, float, float, float
)
```

Parameters

`float, float, float, float, float, float`

TCP Parameter, combines X, Y, Z, RX, RY, RZ

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

```
ChangeTCP(0,0,150,0,0,90)                               // Change the TCP value to {0,0,150,0,0,90}
```

Syntax 6

```
bool ChangeTCP(
    float, float, float, float, float, float,
    float
```

)

Parameters

`float, float, float, float, float, float`

TCP Parameter, combines X, Y, Z, RX, RY, RZ

`float` TCP weight

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

ChangeTCP(0,0,150,0,0,90,2) // Change the TCP value to {0,0,150,0,0,90}, weight = 2kg

Syntax 7

`bool` **ChangeTCP**(

`float, float, float, float, float, float,`

`float,`

`float, float, float, float, float, float, float, float, float`

)

Parameters

`float, float, float, float, float, float`

TCP Parameter, combines X, Y, Z, RX, RY, RZ

`float` Tool's weight

`float, float, float, float, float, float, float, float, float`

Tool's moment of inertia: (1)Ixx, (2)Iyy, (3)Izz and its frame of reference: (4)X, (5)Y, (6)Z, (7)RX, (8)RY, (9)RZ

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

ChangeTCP(0,0,150,0,0,90,2, 2,0.5,0.5,0,0,-80,0,0,0) // Change the TCP value to {0,0,150,0,0,90}, weight = 2kg, moment of inertia = {2,0.5,0.5} and frame of reference = {0,0,-80,0,0,0}

8.15 ChangeLoad()

Send the command of changing the payload value of the follow-up motions into buffer for execution.

Syntax 1

`bool` **ChangeLoad**(

`float`

)

Parameters

`float` Payload(kg)

Return

`bool` `True` Command accepted; `False` Command rejected (format error)

Note

ChangeLoad(5.3) //Set payload to 5.3kg

8.16 PVTEnter()

Set PVT mode to start with Joint/Cartesian command

Syntax1

```
bool PVTEnter(
    int
)
```

Parameter

`int` PVT Mode

`0` Joint

`1` Cartesian

Return

`bool` `True` Command accepted; `False` Command rejected

Note

Syntax2

```
bool PVTEnter(
)
```

Parameter

`void` No input values required. Use PVT mode with Joint Command by default.

Return

`bool` `True` Command accepted; `False` Command rejected

Note

PVTEnter(1)

8.17 PVTExit()

Set PVT mode motion to exit

Syntax1

```
bool PVTExit(
)
```

Parameter

void	No input values required

Return

bool	True Command accepted; False Command rejected

Note

PVTExit()

8.18 PVTPoint()

Set the PVT mode parameters of motion in position, velocity, and duration.

Syntax1

```
bool PVTPoint(
    float[],
    float[],
    float
)
```

Parameter

float[]	Target position {J1, J2, J3, J4, J5, J6} in PVT mode with Joint {X, Y, Z, RX, RY, RZ} in PVT mode with Cartesian
float[]	Target velocity {J1, J2, J3, J4, J5, J6}' in PVT mode with Joint {X, Y, Z, RX, RY, RZ}' in PVT mode with Cartesian
float	Duration (s)

Return

bool	True Command accepted; False Command rejected

Note

Syntax2

```
bool PVTPoint(
    float, float, float, float, float, float,
    float, float, float, float, float, float,
    float
)
```

Parameter

`float, float, float, float, float, float,` Target Position.
{J1, J2, J3, J4, J5, J6} in PVT mode with Joint
{X, Y, Z, RX, RY, RZ} in PVT mode with Cartesian

`float, float, float, float, float, float,` Target Velocity.
{J1, J2, J3, J4, J5, J6}' in PVT mode with Joint
{X, Y, Z, RX, RY, RZ}' in PVT mode with Cartesian

`float` Duration (s)

Return

`bool` `True` Command accepted; `False` Command rejected

Note

PVTEnter(1)
PVTPoint(467.5,-122.2,359.7,180,0,90,50,50,0,0,0,0,0.5)
PVTPoint(467.5,-72.2,359.7,180,0,90,-50,50,0,0,0,0,0.5)
PVTPoint(417.5,-72.2,359.7,180,0,90,0,0,0,0,0,0,0.5)
PVTPoint(417.5,-122.2,359.7,180,0,90,50,50,0,0,0,0,0.5)
PVTPoint(417.5,-122.2,359.7,180,0,60,50,50,0,0,0,0,3)
PVTPoint(417.5,-122.2,359.7,180,0,90,50,50,0,0,0,0,3)
PVTExit()

8.19 PVTPause()

Set PVT mode motion to pause

Syntax1

```
bool PVTPause(
)
```

Parameter

`void` No input values required

Return

`bool` `True` Command accepted; `False` Command rejected

Note

PVTPause()

8.20 PVTResume()

Set PVT mode motion to resume

Syntax1

```
bool PVTResume(
)
```

Parameter

`void` No input values required

Return

`bool` `True` Command accepted; `False` Command rejected

Note

PVTResume()

8.21 socket_send()

Establish the connection between the socket client and the remote server and send messages to the remote server devices.

*Applicable in the external script control mode.

Syntax 1

```
int socket_send(
    string,
    int,
    ?
    int,
    int,
)
```

Parameters

`string` The IP of the remote server

`int` The port number of the remote server

?	The value to send. Available types: int, float, , bool, string, and array. Numeric values will be conversed in Little Endian, and string values will be converse in UTF8
int	The maximum number of attempts to post
<= 0	Keep on trying after encountered errors until posted successfully.
int	The time in milliseconds to wait for the retry after encountered errors.
< 0	The invalid time to wait. Will set to the default: 1000ms

Return

int	Returns the number of attempts to post successfully.
> 0	Post successfully. The returned value shows the number of attempts to post successfully such as 1 indicate the post is success at the first attempt.
0	Failed to post.
-1	Unable to connect to the remote server
-2	Incorrect IP or Port number

Syntax 2

```
int socket_send(
    string,
    int,
    ?
)
```

Note

The syntax is the same as syntax 1. The default maximum number of times to post is 1, and the time to wait to retry is 1000 ms.

socket_send(string, int, ?) => **socket_send**(string, int, ?, 1, 1000)

Note

string var_ip = “127.0.0.1”

int var_port = 12345

byte var_b = 100

var_value = **socket_send**(var_ip, 99999, var_b) // send 0x64 // var_value = -2 // Post unsuccessfully. // Incorrect Port number

var_value = **socket_send**(var_ip, port, var_b) // send 0x64 // var_value = -1 // Post unsuccessfully. // Suppose the server is unavailable.

var_value = **socket_send**(var_ip, port, var_b, 0, 1000) // send 0x64 // var_value = 23 // Retry to post and successfully until the 23rd time.

var_value = **socket_send**(var_ip, var_port, 123456) // send 0x40 0xE2 0x01 0x00

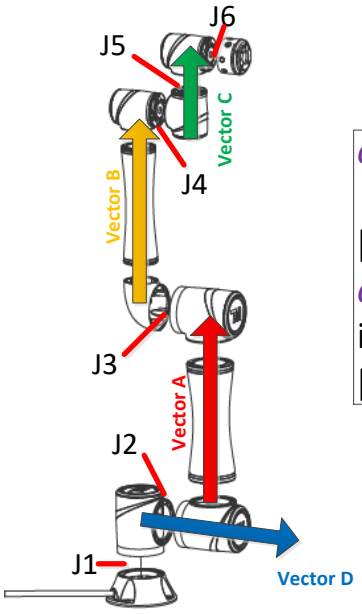
```
var_value = socket_send(var_ip, var_port, 123.456)    // send 0x79 0xE9 0xF6 0x42
double var_d = 123.456
var_value = socket_send(var_ip, var_port, var_d)       // send 0x77 0xBE 0x9F 0x1A 0x2F 0xDD 0x5E 0x40
var_value = socket_send(var_ip, var_port, “123.456”)// send 0x31 0x32 0x33 0x2E 0x34 0x35 0x36
var_value = socket_send(var_ip, var_port, true)        // send 0x01

byte[] var_bb = {100, 200}
var_value = socket_send(var_ip, var_port, var_bb)      // send 0x64 0xC8

string[] var_ss = {“ABC”, “DEF”, “達明機器人” }
var_value = socket_send(var_ip, var_port, var_ss)
            // send 0x41 0x42 0x43 0x44 0x45 0x46 0xE9 0x81 0x94 0xE6 0x98 0x8E 0xE6 0xA9 0x9F 0xE5 0x99 0xA8
            0xE4 0xBA 0xBA
```

*Use GetBytes() to retrieve the array byte[] before posting and post byte[] to the remote server device.

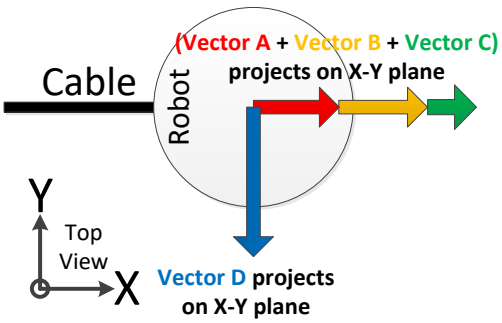
Pose Configuration Parameters: [Config1, Config2, Config3]



Config: config1, config2, config3

config1=0:
if [(Vector A + Vector B + Vector C) projects on X-Y plane] cross [Vector D projects on X-Y plane] is on negative-Z
config1=1:
if [(Vector A + Vector B + Vector C) projects on X-Y plane] cross [Vector D projects on X-Y plane] is on positive-Z

config2=2:
if (M=0 and J3 is positive) or (M=1 and J3 is negative)
config2=3:
if (M=0 and J3 is negative) or (M=1 and J3 is positive)



config3=4:
if (M=0 and J5 is positive) or (M=1 and J5 is negative)
config3=5:
if (M=0 and J5 is negative) or (M=1 and J5 is positive)

9. Modbus Functions

9.1 modbus_read()

Modbus TCP/RTU read function

Syntax 1 (TCP/RTU)

```
? modbus_read(
    string,
    string
)
```

Parameters

`string` TCP/RTU device name (Set in Modbus Device setting)

`string` The predefined parameters belong to TCP/RTU device (Set in Modbus Device setting)

Return

`?` The return data type is decided by the predefined parameters

Signal Type	Function Code	Type	Num Of Addr	Return data type
Digital Output	01	byte	1	byte (H: 1)(L: 0)
		bool	1	bool (H: true)(L: false)
Digital Input	02	byte	1	byte (H: 1)(L: 0)
		bool	1	bool (H: true)(L: false)
Register Output	03	byte	1	byte
		int16	1	int
		int32	2	int
		float	2	float
		double	4	double
		string	?	string
		bool	1	bool
Register Input	04	byte	1	byte
		int16	1	int
		int32	2	int
		float	2	float
		double	4	double
		string	?	string
		bool	1	bool

* According to the Little Endian (CD AB) or Big Endian (AB CD) setting, the int32, float, double data will transformed automatically.

* string will follows the UTF8 data format transformation (Stop at 0x00)

Note

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the default values are applied in Preset Setting

preset_800	DO	800	byte	
preset_7202	DI	7202	bool	
preset_9000	RO	9000	string	4
preset_7001	RI	7001	float	Big-Endian (AB CD)

```
var_value = modbus_read("TCP_1", "preset_800")   // var_value = 1              // DO 1 address = 1 bit
var_value = modbus_read("TCP_1", "preset_7202")  // var_value = true           // DI 1 address = 1 bit
var_value = modbus_read("TCP_1", "preset_9000")  // var_value = ab1234cd       // RO 4 address = 8 bytes size
var_value = modbus_read("TCP_1", "preset_7001")  // var_value = -314.1593      // RI 2 address = 4 bytes size (float)
```

Syntax 2 (TCP/RTU)

```
byte[] modbus_read(
    string,
    byte,
    string,
    int,
    int
)
```

Parameters

string	TCP/RTU Device Name (Set in Modbus Device setting)	
byte	Slave ID	
string	Read type	
DO	Digital Output	(Function Code : 01)
DI	Digital Input	(Function Code : 02)
RO	Register Output	(Function Code : 03)
RI	Register Input	(Function Code : 04)

`int`	Starting address
`int`	Data length

Return

`byte[]`	The returned byte array from modbus server *User defined modbus_read only follows Big-Endian (AB CD) format to read byte[]

Note

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the user defined values are applied to User Setting as

TCP device	0	DO	800	4
TCP device	0	DI	7202	3
TCP device	0	RO	9000	6
TCP device	0	RI	7001	12
TCP device	0	RI	7301	6

```
var_value = modbus_read("TCP_1", 0, "DO", 800, 4)
    // var_value = {0,0,0,0} // DO 4 address = 4 bit to byte array
var_value = modbus_read("TCP_1", 0, "DI", 7202, 3)
    // var_value = {1,0,0}          // DI 3 address = 3 bit to byte array
var_value = modbus_read("TCP_1", 0, "RO", 9000, 6)
    // var_value = {0x54,0x65,0x63,0x68,0x6D,0x61,0x6E,0xE9,0x81,0x94,0xE6,0x98}
    // RO 6 address = 12 bytes size
var_value = modbus_read("TCP_1", 0, "RI", 7001, 12)
    // var_value = {0x29,0x30,0x9F,0x4C,0xC3,0x7C,0x99,0x9A,0x44,0x5E,0xEC,0xCD,0x42,0xB4,0x00,0x00,
    0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}     // RI 12 address = 24 bytes size
var_value = modbus_read("TCP_1", 0, "RI", 7301, 6)
    // var_value = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x0F,0x00,0x0A,0x00,0x39}
    // RI 6 address = 12 bytes size
```

9.2 modbus_read_int16()

Modbus TCP/RTU read function, and transform modbus address data array to int16 array

Syntax 1 (TCP/RTU)

```
int[] modbus_read_int16(
    string,
    byte,
    string,
    int,
    int,
    int
)
```

Parameters

`string`	TCP/RTU Device Name (Set in Modbus Device setting)		
`byte`	Slave ID		
`string`	Read type		
	`DO`	Digital Output	(Function Code : 01)
	`DI`	Digital Input	(Function Code : 02)
	`RO`	Register Output	(Function Code : 03)
	`RI`	Register Input	(Function Code : 04)
`int`	Starting address		
`int`	Data length		
`int`	Follows Little Endian (CD AB) or Big Endian (AB CD) to transform the address data to int16 array. *Invalid Parameter. Only support int32, float, double		
	`0`	Little Endian	
	`1`	Big Endian (Default)	

Return

`int[]`	The returned int array from modbus server

Syntax 2 (TCP/RTU)

```
int[] modbus_read_int16(
    string,
    byte,
    string,
    int,
    int
)
```

Note

Similar to Syntax1 with Big Endian (AB CD) setting

modbus_read_int16(“TCP_1”, 0, “DI”, 7200, 2) => **modbus_read_int16**(“TCP_1”, 0, “DI”, 7200, 2, 1)

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the user defined values are applied to User Setting as

TCP device	0	DO	800	4
TCP device	0	DI	7202	3
TCP device	0	RO	9000	6
TCP device	0	RI	7001	12
TCP device	0	RI	7301	6

var_value = **modbus_read_int16**(“TCP_1”, 0, “DO”, 800, 4)

// byte[] = {0,0,0,0} to int16[] var_value = {0,0} // byte[0][1] , byte[2][3]

var_value = **modbus_read_int16**(“TCP_1”, 0, “DI”, 7202, 3)

// byte[] = {1,0,0} to int16[] var_value = {256,0}

// byte[0][1] , byte[2][3] // Fill up to [3] automatically

var_value = **modbus_read_int16**(“TCP_1”, 0, “RO”, 9000, 6)

// byte[] = {0x54,0x65,0x63,0x68,0x6D,0x61,0x6E,0xE9,0x81,0x94,0xE6,0x98}

// to int16[] var_value = {21605,25448,28001,28393,-32364,-6504}

var_value = **modbus_read_int16**(“TCP_1”, 0, “RI”, 7001, 12)

// byte[] = {0x29,0x30,0x9F,0x4C,0xC3,0x7C,0x99,0x9A,0x44,0x5E,0xEC,0xCD,0x42,0xB4,0x00,0x00,

0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// to int16[] var_value = {10544,-24756,-15492,-26214,17502,-4915,17076,0,-32768,0,0,0}

var_value = **modbus_read_int16**(“TCP_1”, 0, “RI”, 7301, 6)

// byte[] = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x0F,0x00,0x31,0x00,0x23}

// to int16[] var_value = {2018,5,18,15,49,35}

var_value = **modbus_read_int16**(“TCP_1”, 0, “RI”, 7301, 6, 0)

// byte[] = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x0F,0x00,0x31,0x00,0x23}

// to int16[] var_value = {2018,5,18,15,49,35}

9.3 modbus_read_int32()

Modbus TCP/RTU read function, and transform modbus address data array to int32 array

Syntax 1 (TCP/RTU)

```
int[] modbus_read_int32(
    string,
    byte,
    string,
    int,
    int,
    int
)
```

Parameters

`string`	TCP/RTU DEVICE NAME (Set in Modbus Device setting)		
`byte`	Slave ID		
`string`	Read type		
	`DO`	Digital Output	(Function Code : 01)
	`DI`	Digital Input	(Function Code : 02)
	`RO`	Register Output	(Function Code : 03)
	`RI`	Register Input	(Function Code : 04)
`int`	Starting address		
`int`	Data length		
`int`	Follows Little Endian (CD AB) or Big Endian (AB CD) to transform the address data to int32 array.		
	`0`	Little Endian	
	`1`	Big Endian (Default)	

Return

`int[]`	The returned int array from modbus server

Syntax 2 (TCP/RTU)

```
int[] modbus_read_int32(
    string,
    byte,
    string,
    int,
    int
)
```

Note

Similar to Syntax1 with Big Endian (AB CD) setting.

modbus_read_int32(“TCP_1”, 0, “DI”, 7200, 2) => **modbus_read_int32**(“TCP_1”, 0, “DI”, 7200, 2, 1)

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the user defined values are applied to User Setting as

TCP device	0	DO	800	4
TCP device	0	DI	7202	3
TCP device	0	RO	9000	6
TCP device	0	RI	7001	12
TCP device	0	RI	7301	6

var_value = **modbus_read_int32**(“TCP_1”, 0, “DO”, 800, 4)

// byte[] = {0,0,0,0} to int32[] var_value = {0} // byte[0][1][2][3]

var_value = **modbus_read_int32**(“TCP_1”, 0, “DI”, 7202, 3)

// byte[] = {1,0,0} to int32[] var_value = {16777216}// byte[0][1][2][3] // Fill up to [3] automatically.

var_value = **modbus_read_int32**(“TCP_1”, 0, “RO”, 9000, 6)

// byte[] = {0x54,0x65,0x63,0x68,0x6D,0x61,0x6E,0xE9,0x81,0x94,0xE6,0x98}

// to int32[] var_value = {1415930728,1835101929,-2120948072}

var_value = **modbus_read_int32**(“TCP_1”, 0, “RI”, 7001, 12)

// byte[] = {0x29,0x30,0x9F,0x4C,0xC3,0x7C,0x99,0x9A,0x44,0x5E,0xEC,0xCD,0x42,0xB4,0x00,0x00,
0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// to int32[] var_value = {691052364,-1015244390,1147071693,1119092736,-2147483648,0}

var_value = **modbus_read_int32**(“TCP_1”, 0, “RI”, 7301, 6)

// byte[] = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x0F,0x00,0x31,0x00,0x23}

// to int32[] var_value = {132251653,1179663,3211299}

var_value = **modbus_read_int32**(“TCP_1”, 0, “RI”, 7301, 6, 0) // byte[2][3][0][1]

// byte[] = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x0F,0x00,0x31,0x00,0x23}

// to int32[] value ={0x000507E2,0x000F0012,0x00230031} = {329698,983058,2293809}

9.4 modbus_read_float()

Modbus TCP/RTU read function, and transform modbus address data array to float array

Syntax 1 (TCP/RTU)

```
float[] modbus_read_float(
    string,
    byte,
    string,
    int,
    int,
    int
)
```

Parameters

`string`	TCP/RTU DEVICE NAME (Set in Modbus Device setting)		
`byte`	Slave ID		
`string`	Read type		
	`DO`	Digital Output	(Function Code : 01)
	`DI`	Digital Input	(Function Code : 02)
	`RO`	Register Output	(Function Code : 03)
	`RI`	Register Input	(Function Code : 04)
`int`	Starting address		
`int`	Data length		
`int`	Follows Little Endian (CD AB) or Big Endian (AB CD) to transform the address data to float array.		
	`0`	Little Endian	
	`1`	Big Endian (Default)	

Return

`float[]`	The returned float array from modbus server

Syntax 2 (TCP/RTU)

```
float[] modbus_read_float(
    string,
    byte,
    string,
    int,
    int
)
```

Note

Similar to Syntax1 with Big Endian (AB CD) setting.

modbus_read_float(“TCP_1”, 0, “DI”, 7200, 2) => **modbus_read_float**(“TCP_1”, 0, “DI”, 7200, 2, 1)

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the user defined values are applied to User Setting as

TCP device	0	DO	800	4
TCP device	0	DI	7202	3
TCP device	0	RO	9000	6
TCP device	0	RI	7001	12
TCP device	0	RI	7301	6

var_value = **modbus_read_float**(“TCP_1”, 0, “DO”, 800, 4)

// byte[] = {0,0,0,0} to float[] var_value = {0} // byte[0][1][2][3]

var_value = **modbus_read_float**(“TCP_1”, 0, “DI”, 7202, 3)

// byte[] = {1,0,0} to float[] var_value = {2.350989E-38} // byte[0][1][2][3]

// Fill up to [3] automatically.

var_value = **modbus_read_float**(“TCP_1”, 0, “RO”, 9000, 6)

// byte[] = {0x54,0x65,0x63,0x68,0x6D,0x61,0x6E,0xE9,0x81,0x94,0xE6,0x98}

// to float[] var_value = {3.940861E+12,4.360513E+27,-5.46975E-38}

var_value = **modbus_read_float**(“TCP_1”, 0, “RI”, 7001, 12)

// byte[] = {0x29,0x30,0x9F,0x4C,0xC3,0x7C,0x99,0x9A,0x44,0x5E,0xEC,0xCD,0x42,0xB4,0x00,0x00,

0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// to float[] var_value = {3.921802E-14,-252.6,891.7,90,0,0}

var_value = **modbus_read_float**(“TCP_1”, 0, “RI”, 7001, 12, 0) // byte[2][3][0][1]

// to float[] var_value = {0x9F4C2930,0x999AC37C,0xECCD445E,0x000042B4,0x00008000,0x00000000}

= {-4.323275E-20,-1.600218E-23,-1.985221E+27,2.392857E-41,4.591775E-41,0}

var_value = **modbus_read_float**(“TCP_1”, 0, “RI”, 7301, 6)

// byte[] = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x0F,0x00,0x3A,0x00,0x26}

// to float[] var_value = {3.400471E-34,1.65306E-39,5.326512E-39}

9.5 modbus_read_double()

Modbus TCP/RTU read function, and transform modbus address data array to double array

Syntax 1 (TCP/RTU)

```
double[] modbus_read_double(
    string,
    byte,
    string,
    int,
    int,
    int
)
```

Parameters

`string`	TCP/RTU DEVICE NAME (Set in Modbus Device setting)		
`byte`	Slave ID		
`string`	Read type		
	`DO`	Digital Output	(Function Code : 01)
	`DI`	Digital Input	(Function Code : 02)
	`RO`	Register Output	(Function Code : 03)
	`RI`	Register Input	(Function Code : 04)
`int`	Starting address		
`int`	Data length		
`int`	Follows Little Endian (CD AB) or Big Endian (AB CD) to transform the address data to double array.		
	`0`	Little Endian	
	`1`	Big Endian (Default)	

Return

`double[]`	The returned double array from modbus server

Syntax 2 (TCP/RTU)

```
double[] modbus_read_double(
    string,
    byte,
    string,
    int,
    int
)
```

Note

Similar to Syntax1 with Big Endian (AB CD) setting.

modbus_read_double(“TCP_1”, 0, “DI”, 7200, 2) => **modbus_read_double**(“TCP_1”, 0, “DI”, 7200, 2, 1)

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the user defined values are applied to User Setting as

TCP device	0	DO	800	4
TCP device	0	DI	7202	3
TCP device	0	RO	9000	6
TCP device	0	RI	7001	12
TCP device	0	RI	7301	6

var_value = **modbus_read_double**(“TCP_1”, 0, “DO”, 800, 4)

// byte[] = {0,0,0,0} to double[] var_value = {0} // byte[0][1][2][3][4][5][6][7]

var_value = **modbus_read_double**(“TCP_1”, 0, “DI”, 7202, 3)

// byte[] = {1,0,0} to double[] var_value = {7.2911220195564E-304} // byte[0][1][2][3][4][5][6][7]

var_value = **modbus_read_double**(“TCP_1”, 0, “RO”, 9000, 6)

// byte[] = {0x54,0x65,0x63,0x68,0x6D,0x61,0x6E,0xE9,0x81,0x94,0xE6,0x98}

// to double[] var_value = {3.65481260356117E+98,-4.87647898854073E-301}

var_value = **modbus_read_double**(“TCP_1”, 0, “RI”, 7001, 12)

// byte[] = {0x29,0x30,0x9F,0x4C,0xC3,0x7C,0x99,0x9A,0x44,0x5E,0xEC,0xCD,0x42,0xB4,0x00,0x00,
0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// to double[] var_value = {2.76472410615396E-110,2.2818627604613E+21,0}

var_value = **modbus_read_double**(“TCP_1”, 0, “RI”, 7001, 12, 0) // byte[6][7][4][5][2][3][0][1]

// to double[] var_value = {0x999AC37C9F4C2930,0x000042B4ECCD445E,0x0000000000008000}
= {-2.4604103205376E-185,3.62371629877526E-310,1.6189543082926E-319}

var_value = **modbus_read_double**(“TCP_1”, 0, “RI”, 7301, 6)

// byte[] = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x10,0x00,0x0B,0x00,0x29}

```
// to double[]      var_value = {1.06475148078395E-270,1.52982527955113E-308}
```

9.6 modbus_read_string()

Modbus TCP/RTU read function, and convert modbus address data array to string text in UTF8

Syntax 1 (TCP/RTU)

```
string modbus_read_string(
    string,
    byte,
    string,
    int,
    int,
    int
)
```

Parameters

`string`	TCP/RTU DEVICE NAME (Set in Modbus Device setting)		
`byte`	Slave ID		
`string`	Read type		
	`DO`	Digital Output	(Function Code : 01)
	`DI`	Digital Input	(Function Code : 02)
	`RO`	Register Output	(Function Code : 03)
	`RI`	Register Input	(Function Code : 04)
`int`	Starting address		
`int`	Data length		
`int`	Follows Little Endian (CD AB) or Big Endian (AB CD) to transform the address data to string. *Invalid Parameter. Only support int32, float, double. String follows UTF8 and is sequentially transferred according to address.		
	`0`	Little Endian	
	`1`	Big Endian (Default)	

Return

`string`	The returned UTF8 string from modbus server (Stop at 0x00)

Syntax 2 (TCP/RTU)

```
string modbus_read_string(
    string,
    byte,
    string,
```

```
    int,
    int
)
```

Note

Similar to Syntax1 with Big Endian (AB CD) setting.

modbus_read_string(“TCP_1”, 0, “RO”, 9000, 2) => **modbus_read_string**(“TCP_1”, 0, “RO”, 9000, 2, 1)

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the user defined values are applied to User Setting as

TCP device	0	RO	9000	12

modbus_write(“TCP_1”, 0, “RO”, 9000) = “1234 達明机器手臂”

```
        // Undefined numbers of addresses to write, the default value 0 denotes to write the complete data length
        of 22 bytes.
        // Write byte[] = {0x31,0x32,0x33,0x34,0xE9,0x81,0x94,0xE6,0x98,0x8E,
                0xE6,0x9C,0xBA,0xE5,0x99,0xA8,0xE6,0x89,0x8B,0xE8,0x87,0x82}
var_value = modbus_read_string(“TCP_1”, 0, “RO”, 9000, 3)
        // byte[] = {0x31,0x32,0x33,0x34,0xE9,0x81}    // RO 3 address = 6 bytes size
        // to string = 1234
var_value = modbus_read_string(“TCP_1”, 0, “RO”, 9000, 6)
        // byte[] = {0x31,0x32,0x33,0x34,0xE9,0x81,0x94,0xE6,0x98,0x8E,0xE6,0x9C}
        // to string = 1234 達明
var_value = modbus_read_string(“TCP_1”, 0, “RO”, 9000, 12)
        // byte[] = {0x31,0x32,0x33,0x34,0xE9,0x81,0x94,0xE6,0x98,0x8E,
                0xE6,0x9C,0xBA,0xE5,0x99,0xA8,0xE6,0x89,0x8B,0xE8,0x87,0x82, 0x41,0x42}
        // to string = 1234 達明机器手臂 AB    // UTF8 format convertion
        // The ending, 0x00, will not be included when writing data. When reading 12 addresses, it will read
        beyond the range.
modbus_write(“TCP_1”, 0, “RO”, 9000) = “1234”+Ctrl(“\0”)
        // Write byte[] = {0x31,0x32,0x33,0x34,0x00}    // Needs to write 3 Register address
```

var_value = **modbus_read_string**(“TCP_1”, 0, “RO”, 9000, 5)

// byte[] = {0x31,0x32,0x33,0x34,0x00,0x00, 0x94,0xE6,0x98,0x8E} // The last 4 values are the orginal data at those addresses

// to string = 1234 // UTF8 format conversion stops at 0x00

9.7 modbus_write()

Modbus TCP/RTU write function

Syntax 1 (TCP/RTU)

```
bool modbus_write(
    string,
    string,
    ?,
    int
)
```

Parameters

`string` TCP/RTU Device Name (Set in Modbus Device setting)

`string` TCP/RTU The predefined parameters belong to TCP/RTU device (Set in Modbus Device setting)

`?` The input data. The predefined parameters will be applied according to the table below.

Signal Type	Function Code	Type	Input type	Input value
Digital Output	05	byte	byte	(H: 1)(L: 0)
		bool	bool	(H: true)(L: false)
Register Output	06	byte	byte	
		bool	bool	
		int16	int	
Register Output	16	int32	int	
		float	float	
		double	double	
		string	string	

* int32, float, double will be transferred with Little Endian (CD AB) or Big Endian (AB CD) according to user’s setting.

* string will be transferred with UTF8 format

* Writing array value is not supported with predefined parameters. To write with the array value, user defined method should be applied (Syntax 3/4)

`int` The maximum number of addresses to be write, only effective to string type data

> 0	Valid address length. Write with defined address length	
<= 0	Invalid address length. Write all the data	

When this parameter is skipped (As shown in Syntax2), the predefined address length will be applied.

Return

bool	True	Write success	
	False	Write failed	1. If the input data ? is empty string or array 2. If an error occurred in Modbus communication

Syntax 2 (TCP/RTU)

```
bool modbus_write(
    string,
    string,
    ?,
)
```

Note

Similar to Syntax1 with predefined address length to write. If the predefined address length <= 0, it will write all the data.

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

If the user defined values are applied to User Setting as as

preset_800	DO	800	bool	
preset_9000	RO	9000	string	4

```
modbus_write("TCP_1", "preset_800", 1)          // write 1 (true)
modbus_write("TCP_1", "preset_800", 0)          // write 0 (false)
bool var_flag = true
modbus_write("TCP_1", "preset_800", var_flag) // write 1 (true)
modbus_write("TCP_1", "preset_800", false)      // write 0 (false)

string var_ss = "ABCDEFGHIJKLMNOPQRST"      // With no number of address, the predefined address length, 4,
                                             is applied. That is 4 RO = 8 bytes size can be written.
```

modbus_write(“TCP_1”, “preset_9000”, var_ss) // write ABCDEFGH // The exceeding part will be skipped

// With no number of address, the predefined address length, 4, is applied. That is 4 RO = 8 bytes size can be written.

modbus_write(“TCP_1”, “preset_9000”, “1234567”) // write 1234567\0 // Use 0x00 to fill up 4 address

// With address length = 0, write all the data. ”09AB123” nees 4 addresses.

modbus_write(“TCP_1”, “preset_9000”, “09AB123”, 0) // write 09AB123\0 // Use 0x00 to fill up 4 address

// With address length = 5, write data in 5 addresses. That is 5 RO = 10 bytes size can be wrote.

modbus_write(“TCP_1”, “preset_9000”, “09AB1234”, 5) // write 09AB1234 // The input data needs only 4 addresses.

Syntax 3 (TCP/RTU)

```
bool modbus_write(
    string,
    byte,
    string,
    int,
    ?,
    int
)
```

Parameters

`string` TCP/RTU DEVICE NAME (Set in Modbus Device setting)

`byte` Slave ID

`string` Write type

`DO` Digital Output (Function Code : 05/15)

`RO` Register Output (Function Code : 06/16)

`int` Starting address

`?` Input data

Signal Type	Function Code	Input ? type	Input value
Digital Output	05	byte	(H: 1)(L: 0)
		bool	(H: true)(L: false)
Digital Output	15	byte[]	(H: 1)(L: 0)
		bool[]	(H: true)(L: false)

Register Output	06	byte	
		bool	
Register Output	16	int	
		float	
		double	
		string	
		byte[]	
		int[]	
		float[]	
		double[]	
		string[]	
		bool[]	

*User defined modbus_write will follows Big-Endian (AB CD) format to write

* Here int means int32. For int16 type data, GetBytes() needs to be applied first to change int16 to byte[]

`int` The maximum number of addresses to be write, only effective to string type data

> 0 Valid address length. Write with defined address length

<= 0 Invalid address length. Write all the data

Return

`bool` `True` Write success

`False` Write failed 1. If the input data ? is empty string or array

2. If an error occurred in Modbus communication

Syntax 4 (TCP/RTU)

```
bool modbus_write(
    string,
    byte,
    string,
    int,
    ?
)
```

Note

Similar to Syntax3 with address length <= 0, it will write all the data.

modbus_write("TCP_1", 0, "RO", 9000, var_bb) => **modbus_write**("TCP_1", 0, "RO", 9000, var_bb, 0)

Modbus Address data size

Digital 1 address = 1 bit size

Register 1 address = 2 bytes size

If the user defined values are applied to User Setting as

TCP device	0	DO	800	4
TCP device	0	RO	9000	12

```
byte[] var_bb = {10, 20, 30}
modbus_write("TCP_1", 0, "DO", 800, var_ bb)        // write 1,1,1
                                                    // Zero value, write 0. Non-zero value, write 1.
modbus_write("TCP_1", 0, "DO", 800, var_bb, 2)      // write 1,1
                                                    // Address number = 2, only write 2 addresses.
modbus_write("TCP_1", 0, "DO", 800, true)           // write 1
int var_i = 10000
modbus_write("TCP_1", 0, "RO", 9000, var_i)         // write 0x00,0x00,0x27,0x10
                                                    // with int32 BigEndian (AB CD) default
var_bb = GetBytes(var_i, 0, 1)                      // var_bb = {0x10,0x27}
                                                    // transfer to int16 LittleEndian (CD AB)
modbus_write("TCP_1", 0, "RO", 9000, var_bb)        // write 0x10,0x27
string[] var_n = {"ABC", "12", "34"}
modbus_write("TCP_1", 0, "RO", 9000, var_n, 2)      // write ABC1
                                                    // Only 2 addresses available, the exceeding values cannot
                                                    be applied.
modbus_write("TCP_1", 0, "RO", 9000, var_n, 5)      // write ABC12340
                                                    // The data needs 4 addresses (0xAB 0xC1 0x23 0x40)
```

10. TM Ethernet Slave

Ethernet Slave comes with functions established with Socket TCP based on the framework of client/server connections. Once enabled, the robot establishes a Socket TCP Listener Serve to send the robot status and data to all of the connected clients or receive the contents from the clients to execute the respective instructions and update the respective information periodically and promptly without the real-time guarantee.

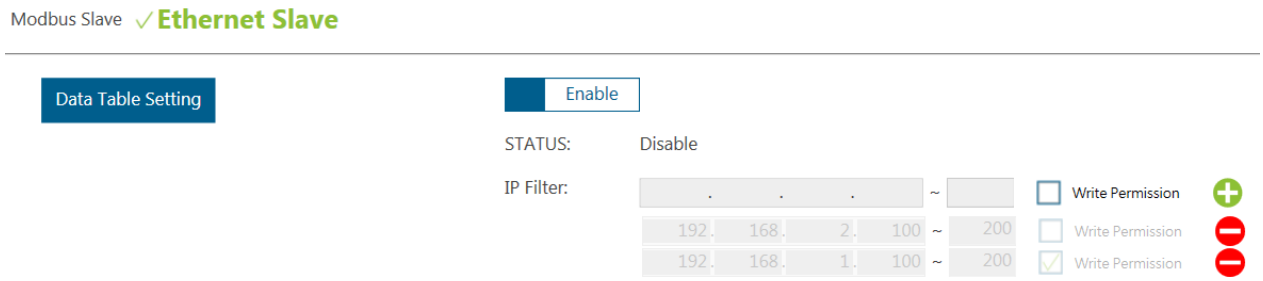
Like the Modbus Slave, the Ethernet Slave will automatically start on its own after power cycling if it was previously set to **Enable**. The established IP and Port will be shown in the Notice Log on the right.

IP TMflow → System → Network → IP Address

Port 5891

Frequency 100Hz (no real-time guarantee)

10.1 GUI Setting



Enable/Disable	Enable or disable Ethernet Slave
IP Filter	IP whitelist Sets ranges for eligible IP addresses that are allowed to connect to the Ethernet Slave. If no filters are set, all devices on the network can connect to the Ethernet Slave.
Write Permission	If checked, allows devices within the corresponding IP range to write to the Ethernet Server with TMSVR commands.

For example, setting IP Filter.

Group 1 192.168.2.100 ~ 200 denotes IP 192.168.2.100, 192.168.2.101, … , and 192.168.2.200 are available for connections.

Group 2 192.168.1.100 ~ 200 denotes IP 192.168.1.100, 192.168.1.101, … , and 192.168.1.200 are available for connections.

If the IP address of the client is not in the range of the IPs listed above, it rejects the client to connect.

Group 2 192.168.1.100 ~ 200 has write permission, so clients within the Group 2 IP range can send TMSVR commands and write to the Ethernet Slave. Group 1 192.168.2.100 ~ 200does not have write permission, so clients in Group 1 will receive errors when attempting to write to the Ethernet Slave.

10.2 svr_read()

Read the item value in the communication data table of Ethernet Slave in the Connection Tab of Robot Setting at the local host.

Syntax 1

```
? svr_read(
    string
)
```

Parameter

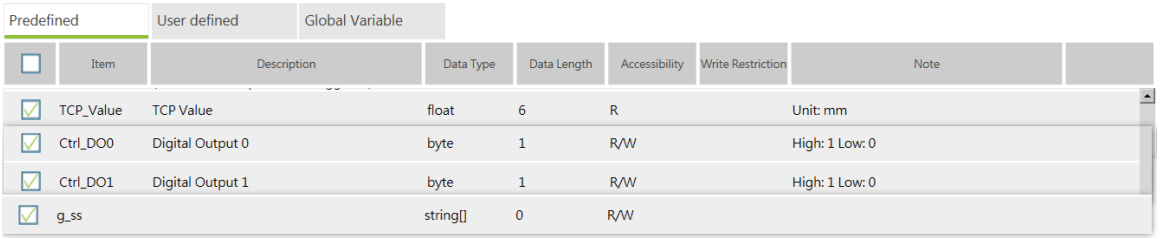
`string` Item name

Return

`?` Return value by set data type

Note

For example, checking with TCP_Value float[], Ctrl_DO0 byte, Ctrl_DO1 byte, and g_ss string[] as the communication data table.




```
float[] var_fval = svr_read("TCP_Value")        // var_fval = {1, 1, 1, 0.1, 0.2, 0.1}
byte var_b0 = svr_read("Ctrl_DO0")              // var_b0 = 0
byte var_b1 = svr_read("Ctrl_DO1")              // var_b1 = 1
string[] var_ss = svr_read("g_ss")              // var_ss = {"Hi","TM","Robot"}
                                                // Use g_ss for global variable naming
byte var_st = svr_read("Robot_Link")            // var_st = 0 (Robot disconnected) 1 (Robot connected)

float[] var_fval = svr_read("TCP_Value")        // Error. Suppose Ethernet Slave is not launched.
float[] var_fval = svr_read("TCP_Value1")       // Error. Item name TCP_Value1 does not exist.
float[] var_fval = svr_read("Coord_Base_Flange") // Error. Item name Coord_Base_Flange does not exist.
// Item name Coord_Base_Flange is predefined but does not exist for not checking as the communication data.
```

10.3 svr_write()

Write the item value into the communication data table of Ethernet Slave in the Connection Tab of Robot Setting at the local host.

Syntax 1

```
bool svr_write(
    string,
    ?
)
```

Parameters

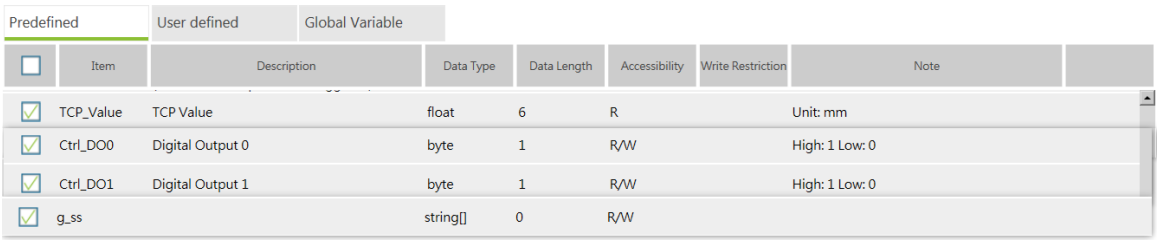
`string`	Item name
`?`	Item value

Return

`bool`	`True`	Write successfully	
	`False`	Write failed	Possible causes 1. Item name does not exist. 2. Unable to write the read-only item name. 3. Item value to write mismatched with item data type.

Note

For example, checking with TCP_Value float[], Ctrl_DO0 byte, Ctrl_DO1 byte, and g_ss string[] as the communication data table.



Predefined | User defined | Global Variable

	Item	Description	Data Type	Data Length	Accessibility	Write Restriction	Note
☑	TCP_Value	TCP Value	float	6	R		Unit: mm
☑	Ctrl_DO0	Digital Output 0	byte	1	R/W		High: 1 Low: 0
☑	Ctrl_DO1	Digital Output 1	byte	1	R/W		High: 1 Low: 0
☑	g_ss		string[]	0	R/W		

```
float[] var_tvalue = {1,2,3,0.1,0.2,0.3}
var_flag = svr_write("TCP_Value", var_tvalue)   // var_flag = false      read-only, invalid process (not an error)
var_flag = svr_write("Ctrl_DO0", 1)             // var_flag = true , Ctrl_DO0 = 1
var_flag = svr_write("Ctrl_DO1", 0)             // var_flag = true , Ctrl_DO1 = 0

var_flag = svr_write("TCP_Value", var_tvalue)   // Error. Suppose Ethernet Slave is not launched.
var_flag = svr_write("TCP_Value1", var_tvalue)  // Error. Item name TCP_Value1 does not exist.
var_flag = svr_write("Ctrl_DO0", "True")        // Error. Item name Ctrl_DO0 writes value as string (the data
                                                   type is set to byte)
```

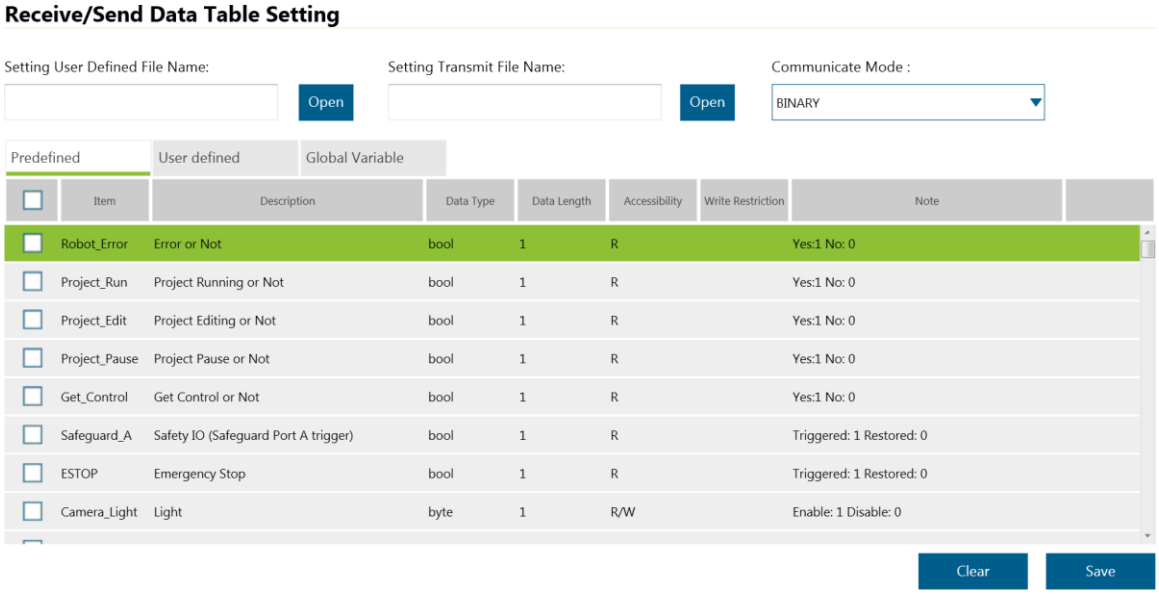
10.4 Data Table

Users can use the items listed the Data Table to customize the required data content as well as configure the commication protocol to transmit between the Ethernet Slave and clients, and save the settings as a communictaion file. When the Ethernet Slave is enabled, the data items in the communication file will be established with the relevant data content to the item to send to the connected clients periodically (no real-time guarantee). The types of the data format is defined by the settings in the commuicataion file. The client

can send data to the server with any type of the supported data formats.

In the protocol, the types of the supported data format are:

BINARY	Binary format, converse in Byte array (Little Endian / UTF8)
STRING	String format, similar to the external command format
JSON	JSON string format



Item ***Robot_Link*** is predefined in Ethernet Slave as a type of byte and the attribute of read only to denote whether to connect to the robot.

1. ***Predefined***

   Items and settings in this section are defined by TMflow, and the data content of the items is updated by TMflow. The defined items are the general statuses of the robot, such as the coordinates of the robot, the state of the project, the state of the electrical control box, or the IO related statuses, such as digital input / digital output, analog input / analog output.

2. ***User defined***

   Items and settings in this section are defined by TMflow users for project programs to read / write item data through the Expression Editor or for external users to read / write item data through the TMSVR commands over a TCP/IP connection. With the user defined tab, the project programs can work with external communication devices as a data exchange protocol. The item list in the user-defined tab can be saved as a custom-defined fil to be edited or exchanged data in the future.

3. ***Global Variable***

   In the global variable tab, the variable list created by the TMflow users provides a way to directly use the variable name for read / write operations in the project programming, and the external communication devices can read / write global variables with the communication protocol.

10.5 Communication Protocol

Length

Start Byte	Hdr		Len		Data			Checksum	End Byte1	End Byte2
$	Header	,	Length	,	Data	,	*	Checksum	\r	\n

Checksum (XOR of these Bytes)

Name	Size	ASCII	HEX	Description
Start Byte	1	$	0x24	Start Byte for Communication
Header	*X*			Header for Communication
Separator	1	,	0x2C	Separator between Header and Length
Length	*Y*			Length of Data
Separator	1	,	0x2C	Separator between Length and Data
Data	*Z*			Communication Data
Separator	1	,	0x2C	Separator between Data and Checksum
Sign	1	*	0x2A	Begin Sign of Checksum
Checksum	*2*			Checksum of Communication
End Byte 1	1	\r	0x0D	
End Byte 2	1	\n	0x0A	End Byte of Communication

*Using the same communication protocol with external commands.

1. Header

Defines the purpose of communication packets. Different headers come with different definitions of communication packets and data.

- *TMSVR* Defines the function of TM Ethernet Slave
- *CPERR* Defines the errors of the communication packets such as packer errors, checksum errors, header errors, and so on.

  *Using the same content definitions with external commands.

2. Length

The length indicates the length in the UTF8 bytes occupied by Data. Users can use decimal, hexadecimal, or binary format. The maximum length is 32 bits.

For example,

$TMSVR,100,Data,*CS\r\n // 100 in decimal indicates the data length is 100 bytes

$TMSVR,0x100,Data,*CS\r\n // 0x100 in hexadecimal indicates the data length is 256 bytes

$TMSVR,0b100,Data,*CS\r\n // 0b100 in binary indicates the data length is 4 bytes

$TMSVR,8,1,達明,*CS\r\n // indicates the length of Data, 1,達明, is 8 bytes (UTF8)

3. Data

The content of the communication packet can support any character (including 0x00 .. 0xFF and uses UTF8 encoding), and the data length determines by Length. The purpose and description defined in Data must be defined by the header.

4. Checksum

The checksum of the communication packet. The calculation method is XOR (exclusive OR). The calculation range is all Bytes between $ and * (excluding $ and *) as shown below.

$TMSVR,100,Data,*CS\r\n

Checksum = Byte[1] ^ Byte[2] … ^ Byte[N-6]

The checksum format is set to 2 bytes in hexadecimal (but not 0x), such as

$TMSVR,5,10,OK,*7E

CS = 0x54 ^ 0x4D ^ 0x53 ^ 0x56 ^ 0x52 ^ 0x2C ^ 0x35 ^ 0x2C ^ 0x31 ^ 0x30 ^ 0x2C ^ 0x4F ^ 0x4B ^ 0x2C = 0x7E

CS = 7E (0x37 0x45)

10.6 TMSVR

Start Byte	Hdr		Len		Data			Checksum	End Byte1	End Byte2
$	*TMSVR*	**,**	Length	**,**	Data	**,**	*	Checksum	**\r**	**\n**

ID		Mode		Content
Transaction ID	,	0/1/2/3/ 11/12/13	,	Item and Value

TMSVR is defined as the TM Ethernet Slave protocol. The Data section of the packet is further divided into three segments, ID (Transaction ID), Mode (Content Mode), and Content (Item and Value), separated with commas and described below.

ID The transaction number expressed in any alphanumeric characters. (Reports the CPERR 04 error

if a non-alphanumeric byte is encountered.) When used as a communication packet response, it is a transaction number that identifies which group of commands to respond.

`,` the symbol to separate

`Mode` The mode as the format of the data content

- `0` Indicates the server responds to the client command in string format.
- `1` Indicates the content data type in binary format
- `2` Indicates the content data type in string format
- `3` Indicates the content data type in JSON format
- `11` Indicates the content data type in binary format (Request read)
- `12` Indicates the content data type in string format (Request read)
- `13` Indicates the content data type in JSON format (Request read)
  - 1/2/3 are for client write to server and client read from server. The client read from server is that the server sends contents to the connected client periodically.
  - 11/12/13 are for client read from server with request read, which is the client sends read request for the item and the server responds with the item value to the client.

`,` the symbol to separate

`Content` The data content. Formatted by the mode definition.

Note

TMSVR command is for the client and the server to communicate in both directions. Under normal circumstances, the server will broadcast the data items from the Transmit and User Defined communication files to the connected clients periodically. When the server sends to the client, the data is sent to the client to read from the server with no response to the server required. When the client sends to the server, the data is received by the server from the client to write with response to the client required.

The transaction number, when the server sends data, cycles from 0 to 9 with each iteration. When the client sends data to the server, the transaction number can be in any alphanumeric characters customized at the client side. If the communication packet format is checked and correct , the server will reply the client with the command processing status by the transaction number in the packet.

When the client sends data to the server, the server checks whether all the criteria to write are correct before performing data writing. If there is any error with the write command, no data will be overwritten. The criteria to write for inspection are:

1. The validity of the mode as the format of the data content
2. The connected client's write permissions based on the IP Filter.
3. The data content matches to the mode.

4. The item to write exists in the Transmit or User Defined communication file.
5. The attribute of the item to write is not read only.
6. The robot is in the appropriate mode (M/A).
7. The written data matches the data type of each item.

When the Mode is 11/12/13, the request read method is used. The client sends an item request and the server responds to the item value. The requested items can be the items in the Predefined area without limited to check the periodical delivery, but in the Userdefined area and the GlobalVariable area, for custom definitions, it still need to check the periodical delivery to get the request.

When the client sends an item request, it is not limited to only one item but multiple items can be acquired at once. When the item request is success, the server will send the item value to the client based on the format matching to Mode 11/12/13. However, if the item request is failed such as the item name does not exist, the server will send the command processing status based on the Mode 0 format.

1. *Mode = 0 (the status the server responds to the client command processing)*

After the server receives and processes a write command from a client, it will respond with another TMSVR command with Mode 0. The details for Mode 0 are as follows.

Data

ID		Mode		Error Code		Error Description
Transaction ID	,	*0*	,	00 .. 07	,	

Transaction ID — Defined while the client sends the command for the server to respond with.

Mode — 0 for the server to respond to the client

Error Code — Error code definitions. Fixed as 2 bytes and in hexadecimal (but not 0x)

- 00 — Correct writing. No error.
- 01 — The communication format or mode is not supported. (Ex. Mode = 99)
- 02 — The connected client is not permitted to write. (IP filer without write permission)
- 03 — The communication format and the data content format are mismatched. (Ex. Mode = 3, but the data content is not in JSON format)
- 04 — Item to write or read does not exist.
- 05 — Unable to write to read-only items.
- 06 — Incorrect M/A mode while writing.
- 07 — Values to write mismatches with the configured type or the size.

Error Description — Error description, following the error code.

- 00 — OK

01 NotSupport

02 WritePermission

03 InvalidData

04 NotExist;XXX // ;XXX denotes which data item

05 ReadOnly;XXX

06 ModeError;XXX

07 ValueError;XXX

< $***TMSVR***,15,S0,*2*,Ctrl_DO0=1,*76\r\n // transcation ID S0, string format, set Ctrl_DO0=1

> $***TMSVR***,10,S0,*0*,00,OK,*18\r\n

// server responds transcation ID S0, mode 0, error code 00, correct writing

< $***TMSVR***,16,S1,*99*,Ctrl_DO0=1,*46\r\n // transcation ID S1, mode 99

> $***TMSVR***,18,S1,*0*,01,NotSupport,*0E\r\n

// server responds transcation ID S1, mode 0, error code 01, mode not support

< $***TMSVR***,15,S2,*2*,Ctrl_DO0=1,*74\r\n // transcation ID S2, string format, set Ctrl_DO0=1

> $***TMSVR***,23,S2,*0*,02,WritePermission,*6A\r\n

// server responds transcation ID S2, mode 0, error code 02, the connected client is not granted with write permission.

< $***TMSVR***,15,S3,*3*,Ctrl_DO0=1,*74\r\n // transcation ID S3, JSON format, set Ctrl_DO0=1

> $***TMSVR***,19,S3,*0*,03,InvalidData,*74\r\n

// server responds transcation ID S3, mode 0, error code 03, JSON format, data format (JSON) mistached with the content data format (STRING)

< $***TMSVR***,16,S4,*2*,Ctrl_DO32=1,*40\r\n // transcation ID S4, string format, set Ctrl_DO32=1

> $***TMSVR***,26,S4,*0*,04,NotExist;Ctrl_DO32,*58\r\n

// server responds transcation ID S4, mode 0, error ode 04, item Ctrl_DO32 does not exist.

< $***TMSVR***,17,S5,*2*,Robot_Link=1,*07\r\n // transcation ID S5, string format, set Robot_Link=1

> $***TMSVR***,27,S5,*0*,05,ReadOnly;Robot_Link,*1E\r\n

// server responds transcation ID S5, mode 0, error code 05, the item Robot_Link is read only.

Supposed the user defined Item: adata, Type: int, Size: 4, and Write: Auto.

< $***TMSVR***,20,S6,*2*,adata={1,2,3,4},*55\r\n // transcation ID S6, string format, set adata={1,2,3,4}

> $***TMSVR***,23,S6,*0*,06,ModeError;adata,*2D\r\n

// server responds transcation ID S6, mode 0, error code 06, M/A mode mismatched while writing (suppose it it Manual Mode while writing).

< $***TMSVR***,18,S7,*2*,adata={1,2,3},*47\r\n // transcation ID S7, string format, set adata={1,2,3}

> $***TMSVR***,24,S7,*0*,07,ValueError;adata,*42\r\n

// server responds transcation ID S7, mode 0, error code 07, writing values and data size or type mismatched. (the configured size is 4, but there is only 3 to write.)

2. *Mode = 1 BINARY*

The data content is transmitted in binary mode by converting the data item name with the Little Endian value and the value with UTF8 to a byte array accordingly. The format is shown as below.

Data

ID		Mode		Content
Transaction ID	,	*1*	,	Item and Value

Length of Item *2 bytes Little Endian*	***Item*** *UTF8*	***Length of Value*** *2 bytes Little Endian*	***Value*** *Little Endian / UTF8*	...

Length of Item	2 bytes in Little Endian, value from 0 to 65535 indicating the length of the item that follows
Item	item name
Length of Value	2 bytes in Little Endian), value from 0 to 65535 indicating the length of the data that follows
Value	data value

Check TCP_Value float[] and Ctrl_DO0 byte as the communication data and transmit in binary mode as an example.

Predefined | User defined | Global Variable

Item	Description	Data Type	Data Length	Accessibility	Write Restriction	Note
TCP_Value	TCP Value	float	6	R		Unit: mm
Ctrl_DO0	Digital Output 0	byte	1	R/W		High: 1 Low: 0

> 24 54 4D 53 56 52 2C // $TMSVR, // Header

36 39 2C // 69, // Length

30 2C *31* 2C // 0,1, // transaction ID 0, mode 1, binary

0A 00 *52 6F 62 6F 74 5F 4C 69 6E 6B* 01 00 *00* // Robot_Link=0 // The name occupied 10 bytes, the value, 1 byte

09 00 *54 43 50 5F 56 61 6C 75 65* 18 00 *00 00 80 3F 00 00 80 3F 00 00 80 3F CD CC CC 3D CD CC 4C 3E CD CC CC 3D* // TCP_Value={1,1,1,0.1,0.2,0.1} // The name occupied 9 bytes, the value, 24 bytes

08 00 *43 74 72 6C 5F 44 4F 30* 01 00 *00* // Ctrl_DO0=0 // The name occupied 8 bytes, the value, 1 byte

2C 2A 39 36 0D 0A // ,*96\r\n // Checksum

< 24 54 4D 53 56 52 2C // $TMSVR, // Header

31 38 2C // 18, // Length

54 31 2C *31* 2C // T1,1, // transaction ID T1, mode 1, binary

08 00 *43 74 72 6C 5F 44 4F 30* 01 00 *01* // Ctrl_DO0=1 // The name occupied 8 bytes, the value, 1 byte

2C 2A 37 41 0D 0A // ,*7A\r\n // Checksum

> $***TMSVR***,10,T1,*0*,00,OK,*1E\r\n // server responds to ID T1, mode 1, error code 00, correct writing

Once the data type of the item to send is string [], two bytes, *0x00 0x00*, are inserted between the string elements as the separators.

> 24 54 4D 53 56 52 2C // $TMSVR, // Header

39 30 2C // 90, // Length

30 2C *31* 2C // 0,1 // transaction ID 0, mode 1, binary

0A 00 *52 6F 62 6F 74 5F 4C 69 6E 6B* 01 00 *00* // Robot_Link=0 //The name occupied 10 bytes, the value, 1 byte

09 00 *54 43 50 5F 56 61 6C 75 65* 18 00 *00 00 80 3F 00 00 80 3F 00 00 80 3F CD CC CC 3D CD CC 4C 3E CD CC CC 3D* // TCP_Value={1,1,1,0.1,0.2,0.1} //The name occupied 9 bytes, the value, 24 bytes

08 00 *43 74 72 6C 5F 44 4F 30* 01 00 *01* // Ctrl_DO0=1 // The name occupied 8 bytes, the value, 1 byte

04 00 *67 5F 73 73* 0D 00 *48 69 00 00 54 4D 00 00 52 6F 62 6F 74*

// g_ss={“Hi”,“TM”,“Robot”} // The name occupied 4 bytes, the value, 13 bytes

2C 2A 44 43 0D 0A // ,*DC\r\n // Checksum

Also, if the data type of the item to receive is string [], when converting to a byte array, two bytes, 00 00, are inserted between the string elements as the separators.

< 24 54 4D 53 56 52 2C // $TMSVR, // Header

32 35 2C // 25, // Length

54 32 2C *31* 2C // T2,1, // transaction ID T2, mode 1, binary

04 00 *67 5F 73 73* 0C 00 *48 65 6C 6C 6F 00 00 57 6F 72 6C 64*

// g_ss={"Hello", "World"} // The name occupied 4 bytes, the value, 12 bytes

2C 2A 30 32 0D 0A // ,*02\r\n // Checksum

> $***TMSVR***,10,T2,*0*,00,OK,*1D\r\n //server responds to ID T2, mode 0, error code 00, correct writing

3. *Mode = 2 STRING*

The data content is transmitted as a string with the name and value of the data item in the Script string of an external command. The format is shown as below.

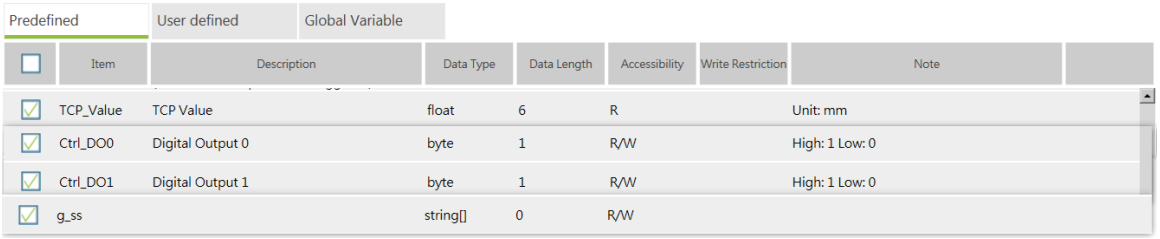
Data

ID		Mode		Content
Transaction ID	,	**2**	,	Item and Value

Item	***=***	***Value***	***\r\n***	***...***

Item	item name
=	equal
Value	data value
\r\n	symbol of carriage return as required if there is an item up next for separation.

Check TCP_Value float[] and Ctrl_DO0 byte, Ctrl_DO1 byte, g_ss string[] as the communication data and transmit in string mode as an example.



Item	Description	Data Type	Data Length	Accessibility	Write Restriction	Note
TCP_Value	TCP Value	float	6	R		Unit: mm
Ctrl_DO0	Digital Output 0	byte	1	R/W		High: 1 Low: 0
Ctrl_DO1	Digital Output 1	byte	1	R/W		High: 1 Low: 0
g_ss		string[]	0	R/W		

> $***TMSVR***,97,9,*2*,Robot_Link=0\r\n // Robot_Link=0 // transaction ID 9, mode 2, string

TCP_Value={1,1,1,0.1,0.2,0.1}\r\n // TCP_Value={1,1,1,0.1,0.2,0.1}

Ctrl_DO0=1\r\n // Ctrl_DO0=1

Ctrl_DO1=0\r\n // Ctrl_DO1=0

g_ss={“Hi”,”TM”,”Robot”},*77\r\n // g_ss={“Hi”,”TM”,”Robot”}

< $***TMSVR***,15,T2,*2*,Ctrl_DO0=0\r\n // set Ctrl_DO0=0// transaction ID T2, mode 2, string

Ctrl_DO1=1,*34\r\n // set Ctrl_DO1=1

> $***TMSVR***,10,T2,*0*,00,OK,*1D\r\n // server responds to ID T2, mode 0, error code 00, correct writing

4. Mode = 3 JSON

The data content is transmitted as a JSON string with the name and value of the data item serialized in the JSON format as shown below.

Data

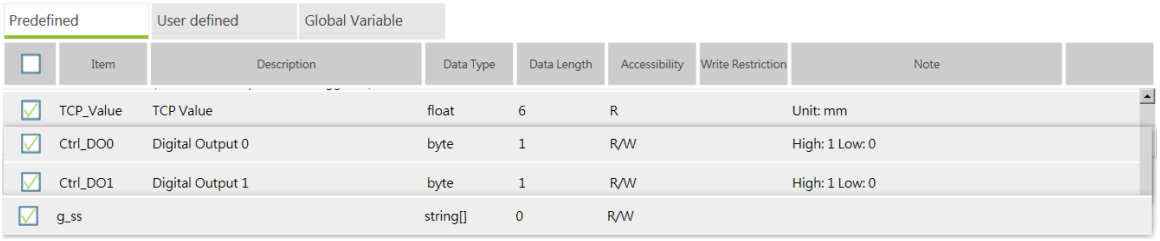
ID		Mode		Content
Transaction ID	,	*3*	,	Item and Value

Item	item name
Value	data value

```
public class TMSVRJsonData
{
    public string Item;
    public object Value;
}
```

*[] array is in use when it comes it multiple items.

Check TCP_Value float[] and Ctrl_DO0 byte, Ctrl_DO1 byte, g_ss string[] as the communication data and transmit in JSON mode as an example.



> $***TMSVR***,196,5,*3*,[{"Item":"Robot_Link","Value":0}, // Robot_Link=0

// transaction ID 5, mode 3, JSON

{"Item":"TCP_Value","Value":[1.0,1.0,1.0,0.1,0.2,0.1]}, // TCP_Value={1,1,1,0.1,0.2,0.1}

{"Item":"Ctrl_DO0","Value":0}, // Ctrl_DO0=0

{"Item":"Ctrl_DO1","Value":0}, // Ctrl_DO1=0

{"Item":"g_ss","Value":["Hi","TM","Robot"]}],*3A\r\n // g_ss={“Hi”,”TM”,”Robot”}

< $***TMSVR***,113,T9,*3*,[{"Item":"Ctrl_DO0","Value":1}, // Ctrl_DO0=1

{"Item":"Ctrl_DO1","Value":0}, // Ctrl_DO1=0

{"Item":"g_ss","Value":["Hello","TM","Robot"]}],*7C\r\n // g_ss={“Hello”,”TM”,”Robot”}

> $***TMSVR***,10,T9,*0*,00,OK,*16\r\n // server responds to ID T9, mode 0, error code 0, correct writing

5. Mode = 11 BINARY (Request read)

The data content is transmitted in binary mode by converting the data item name with the Little Endian value and the value with UTF8 to a byte array accordingly. The format is shown as below.

Data (client to server)

ID		Mode		Content
Transaction ID	,	*11*	,	Item

Length of Item	***Item***	...
2 bytes Little Endian	*UTF8*	

The difference of the read request from Mode = 1 is no value required.

Length of Item 2 bytes in Little Endian, value from 0 to 65535 indicating the length of the item that follows

Item Item name

Check TCP_Value float[] and Ctrl_DO0 byte as the communication data and transmit in binary mode as an example

Predefined | User defined | Global Variable

	Item	Description	Data Type	Data Length	Accessibility	Write Restriction	Note
☑	TCP_Value	TCP Value	float	6	R		Unit: mm
☑	Ctrl_DO0	Digital Output 0	byte	1	R/W		High: 1 Low: 0

server periodical delivery

> 24 54 4D 53 56 52 2C // $TMSVR, // Header

36 39 2C // 69, // Length

30 2C *31* 2C // 0,1, // transaction ID 0, mode 1, binary

0A 00 *52 6F 62 6F 74 5F 4C 69 6E 6B* 01 00 *00* // Robot_Link=0 // The name occupied 10 bytes, the value, 1 byte

09 00 *54 43 50 5F 56 61 6C 75 65* 18 00 *00 00 80 3F 00 00 80 3F 00 00 80 3F CD CC CC 3D CD CC 4C 3E CD CC CC 3D* // TCP_Value={1,1,1,0.1,0.2,0.1} // The name occupied 9 bytes, the value, 24 bytes

08 00 *43 74 72 6C 5F 44 4F 30* 01 00 *00* // Ctrl_DO0=0 // The name occupied 8 bytes, the value, 1 byte

2C 2A 39 36 0D 0A // ,*96\r\n // Checksum

client requested to read

< 24 54 4D 53 56 52 2C // $TMSVR, // Header

32 36 2C // 26, // Length

51 31 2C *31 31* 2C // Q1,11, // transaction ID Q1, mode 11 binary (Request read)

08 00 *43 74 72 6C 5F 44 4F 30* // Ctrl_DO0 // The name occupied 8 bytes

08 00 *54 43 50 5F 4D 61 73 73* // TCP_Mass // The name occupied 8 bytes

2C 2A 37 46 0D 0A // ,*7F\r\n // Checksum

server replied with the item value

> 24 54 4D 53 56 52 2C // $TMSVR, // Header

33 35 2C // 35, // Length

51 31 2C *31 31* 2C // Q1,11, // server responds to ID Q1, mode 11 binary

08 00 *43 74 72 6C 5F 44 4F 30* 01 00 *00*

// Ctrl_DO0=0 // The name occupied 8 bytes, the value, 1 byte

08 00 *54 43 50 5F 4D 61 73 73* 04 00 *00 00 00 00*

// TCP_Mass=0 // The name occupied 8 bytes, the value, 4 byte

2C 2A 37 38 0D 0A // ,*78\r\n // Checksum

* server replied the same content format as Mode = 1 BINARY

client requested to read

< 24 54 4D 53 56 52 2C // $TMSVR, // Header

32 36 2C // 26, // Length

51 32 2C *31 31* 2C // Q2,11, // transaction ID Q1, mode 11 binary (Request read)

08 00 *43 74 72 6C 5F 44 4F 30* // Ctrl_DO0 // The name occupied 8 bytes

08 00 *54 43 50 5F 4D 61 58 58* // TCP_MaXX // The name occupied 8 bytes

2C 2A 37 43 0D 0A // ,*7C\r\n // Checksum

server replied with the item value

> $***TMSVR***,25,Q2,*0*,04,NotExist;TCP_MaXX,*17\r\n

// server responds to ID Q2, mode 0, error code 04, item not existed

6. Mode = 12 STRING (Request read)

The data content is transmitted as a string with the name and value of the data item in the Script string of an external command. The format is shown as below.

Data (client to server)

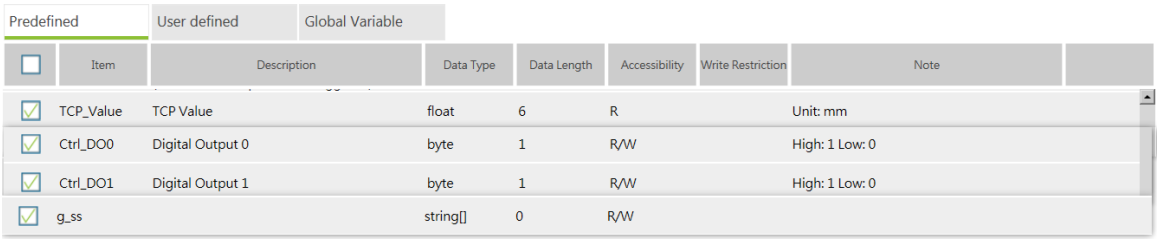
ID		Mode		Content
Transaction ID	,	***12***	,	Item and Value

Item	***\r\n***	...

No Value required.

Item	Item name
\r\n	The newline characters. Required only as a delimiter if the next item comes.

Check TCP_Value float[] and Ctrl_DO0 byte, Ctrl_DO1 byte, g_ss string[] as the communication data and transmit in STRING mode.



server periodical delivery

> $***TMSVR***,97,9,*2*,Robot_Link=0\r\n // Robot_Link=0 // transaction ID 9, mode 2

TCP_Value={1,1,1,0.1,0.2,0.1}\r\n // TCP_Value={1,1,1,0.1,0.2,0.1}

Ctrl_DO0=1\r\n // Ctrl_DO0=1

Ctrl_DO1=0\r\n // Ctrl_DO1=0

g_ss={“Hi”,”TM”,”Robot”},*77\r\n // g_ss={“Hi”,”TM”,”Robot”}

client requested to read

< $***TMSVR***,28,Q2,*12*,Robot_Link\r\n // Item Robot_Link

// transaction ID Q2, mode 12 JSON (Request read)

TCP_Mass,*0E\r\n // Item TCP_Mass

server replied with the item value

> $***TMSVR***,30,Q2,*12*,Robot_Link=0\r\n // server responds to ID Q2, mode 12

TCP_Mass=0,*09\r\n

* server replied the same content format as Mode = 2 STRING

7. *Mode = 13 JSON (Request read)*

The data content is transmitted as a JSON string with the name and value of the data item serialized in the JSON format as shown below.

Data (client to server)

ID		Mode		Content
Transaction ID	,	13	,	Item and Value

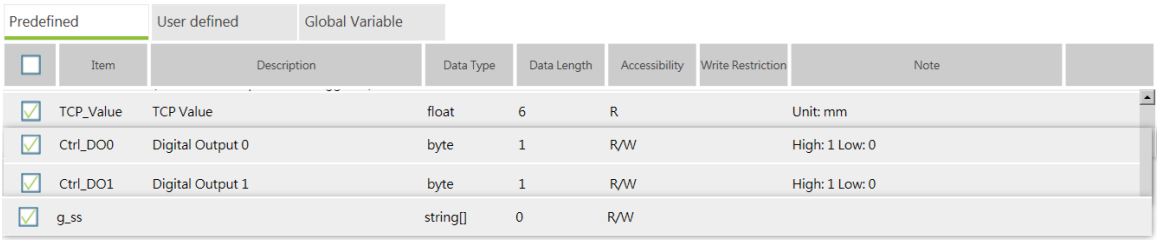
Item	Item name
Value	Data value

```
public class TMSVRJsonData
{
    public string Item;
    public object Value;
}
```

* [] array is in use when it comes it multiple items.

* Shared with Mode = 3 JSON for using the same class for serialization / deserialization, but the Value attribute may not exist

Check TCP_Value float[] and Ctrl_DO0 byte, Ctrl_DO1 byte, g_ss string[] as the communication data and transmit in JSON mode.



server periodical delivery

> $***TMSVR***,196,5,*3*,[{"Item":"Robot_Link","Value":0}, // Robot_Link=0 // transaction ID 5, mode 3
{"Item":"TCP_Value","Value":[1.0,1.0,1.0,0.1,0.2,0.1]}, // TCP_Value={1,1,1,0.1,0.2,0.1}
{"Item":"Ctrl_DO0","Value":0}, // Ctrl_DO0=0
{"Item":"Ctrl_DO1","Value":0}, // Ctrl_DO1=0
{"Item":"g_ss","Value":["Hi","TM","Robot"]}],*3A\r\n // g_ss={“Hi”,”TM”,”Robot”}

client requested to read

< $***TMSVR***,27,Q3,*13*,[{"Item":"TCP_Mass"}],*3C\r\n
// transaction ID Q3, mode 13 JSON (Request read)

server replied with the item value

> $***TMSVR***,39,Q3,*13*,[{"Item":"TCP_Mass","Value":0.0}],*40\r\n // server responds to ID Q3, mode 13

* server replied the same content format as Mode = 3 JSON

11. Profinet Functions

The robot communicates with external controllers via the Profinet communication protocol. In the mechanism of the Profinet communication protocol, the robot works as a Profinet IO device for external devices to read and write the robot data. Meanwhile, TMflow monitors the table of data receiving from external devices and the table of data sending to external devices with Profinet functions as well as changes the custom definition section in the table of data sending to external devices.

Communication Data Table

The data table is composed of the input data and the ouput data. Input Data Table is for external devices posting on the robot, and Output Data Table is for the robot sending to external devices. Both of the data tables come with System Definition Section and Custom Definition Section for data.

1. System Definition Section：Items and settings are defined by the robot, and the data contents are updated by the robot or external devices. The defined items are robot status relevant such as robot bases, project status, control box status, or input/output status relevant such as digital I/Os and analog I/Os. Users can use Profinet functions to read the input data table and the output data table in the system definition section.
2. Custom Definition Section：Items and settings are defined by users, and the data contents are updated by users or external devices. In the meantime of the project editing, users can use Profinet functions to read and write the output data table in the custom definition section or read input data table in the custom definition section as well as use the custom definition section as a data exchange register between the project and external devices.

Communication Data Table (at the robot's view)	Data Section	TMflow Profinet Function Permissions	External Device Permissions
Input Data Table	System Definition Section	Read	Write
	Custom Definition Section	Read	Write
Output Data Table	System Definition Section	Read	Read
	Custom Definition Section	Read/Write	Read

11.1 profinet_read_input()

Read the input table content.

Syntax 1

```
byte[] profinet_read_input(
    int,
    int
)
```

Paramaters

int	Starting address
int	The address length to read

Return

byte[]	Return data in a byte array.

Note

```
byte[] var_value = profinet_read_input(148, 16)
    // {0x30,0x31,0x30,0x36,0x30,0x31,0x31,0x32,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
```

Syntax 2

```
byte profinet_read_input(
    int,
)
```

Paramaters

int	Starting address

Return

byte	Return data in byte.

Note

```
byte var_value = profinet_read_input(148)
    // 0x30
```

11.2 profinet_read_input_int()

Read the input table content and convert the data to the 32-bit integer.

Syntax 1

```
int[] profinet_read_input_int(
```

```
    int,
    int,
    int
)
```

Paramaters

`int`	Starting address
`int`	The address length to read
`int`	The coversion of the read data to an int array based on Little Endian (DCBA) or Big Endian (ABCD).
	`0` Little-Endian
	`1` Big-Endian
	`2` Based on the configuration file.

Return

`int[]`	Return data in an integer array.

Note

```
int[] var_value = profinet_read_input_int(164, 12, 0)
    // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian)      to int[]
    // int[] = {0x00007FFF,0x0001869F,0xFFFF8000} (Little Endian)
    // int[] = {32767,99999,-32768}
int[] var_value = profinet_read_input_int(164, 11, 0)
    // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF} (Little Endian)     to int[]
    // int[] = {0x00007FFF,0x0001869F,0x00FF8000} (Little Endian)
    // int[] = {32767,99999,16744448}
int[] var_value = profinet_read_input_int(164, 10, 0)
    // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80} (Little Endian)    to int[]
    // int[] = {0x00007FFF,0x0001869F,0x00008000} (Little Endian)
    // int[] = {32767,99999,32768}
int[] var_value = profinet_read_input_int(164, 12, 1)
    // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian)      to int[]
    // int[] = {0xFF7F0000,0x9F860100,0x0080FFFF} (Big Endian)
    // int[] = {-8454144,-1618607872,8454143}
int[] var_value = profinet_read_input_int(164, 12, 2)
    // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian)      to int[]
    // int[] = {0x00007FFF,0x0001869F,0xFFFF8000} (Little Endian)
    // int[] = {32767,99999,-32768}
```

Syntax 2

```
int[] profinet_read_input_int(
    int,
    int
)
```

Note

Same as Syntax 1 with the parameter of the conversion of the read data defaults to 2.

* Convert the read data to an int array based on Little Endian (DCBA) or Big Endian (ABCD).

int[] var_value = **profinet_read_input_int**(164, 12, 2)

// byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian) to int[]

// int[] = {0x00007FFF,0x0001869F,0xFFFF8000} (Little Endian)

// int[] = {32767,99999,-32768}

Syntax 3

```
int profinet_read_input_int(
    int
)
```

Paramaters

`int` Starting address

* Convert the read data to an int array based on Little Endian (DCBA) or Big Endian (ABCD).

Return

`int` Return data in integer.

Note

int var_value = **profinet_read_input_int**(164)

// byte[] = {0xE4,0x07,0x00,0x00} (Little Endian) to int

// int = 0x000007E4 (Little Endian)

// int = 2020

int var_value = **profinet_read_input_int**(164)

// byte[] = {0x00,0x00,0x07,0xE4} (Big Endian) to int

// int = 0x000007E4 (Big Endian)

// int = 2020

11.3 profinet_read_input_float()

Read the input table content and convert the data to the 32-bit floating-point number.

Syntax 1

```
float[] profinet_read_input_float(
    int,
    int,
    int
)
```

Paramaters

`int`	Starting address
`int`	The address length to read
`int`	The conversion of the read data to a float array based on Little Endian (DCBA) or Big Endian (ABCD).
`0`	Little-Endian
`1`	Big-Endian
`2`	Based on the configuration file.

Return

`float[]` Return data in a floating-point number array.

Note

```
float[] var_value = profinet_read_input_float(284, 12, 0)
    // byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian)    to float[]
    // float[] = {0x3F800000,0x40000000,0x40400000} (Little Endian)
    // float[] = {1.0,2.0,3.0}
float[] var_value = profinet_read_input_float(284, 11, 0)
    // byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40} (Little Endian)    to float[]
    // float[] = {0x3F800000,0x40000000,0x00400000} (Little Endian)
    // float[] = {1.0,2.0,5.877472E-39}
float[] var_value = profinet_read_input_float(284, 10, 0)
    // byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00} (Little Endian)    to float[]
    // float[] = {0x3F800000,0x40000000,0x00000000} (Little Endian)
    // float[] = {1.0,2.0,0.0}
float[] var_value = profinet_read_input_float(284, 12, 1)
    // byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian)    to float[]
    // float[] = {0x0000803F,0x00000040,0x00004040} (Big Endian)
    // float[] = {4.600603E-41,8.96831E-44,2.304856E-41}
float[] var_value = profinet_read_input_float(284, 12, 2)
```

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian) to float[]

// float[] = {0x3F800000,0x40000000,0x40400000} (Little Endian)

// float[] = {1.0,2.0,3.0}

Syntax 2

```
float[] profinet_read_input_float(
    int,
    int
)
```

Note

Same as Syntax 1 with the parameter of the conversion of the read data defaults to 2.

* Convert the read data to a float array based on Little Endian (DCBA) or Big Endian (ABCD).

float[] var_value = **profinet_read_input_float**(284, 12, 2)

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian) to float[]

// float[] = {0x3F800000,0x40000000,0x40400000} (Little Endian)

// float[] = {1.0,2.0,3.0}

Syntax 3

```
float profinet_read_input_float(
    int
)
```

Paramaters

`int` Starting address

* Convert the read data to a float array based on Little Endian (DCBA) or Big Endian (ABCD).

Return

`float` Return data in floating-point

Note

float var_value = **profinet_read_input_float**(284)

// byte[] = {0x00,0x00,0x80,0x3F} (Little Endian) to float

// float = 0x3F800000 (Little Endian)

// float = 1.0

float var_value = **profinet_read_input_float**(284)

// byte[] = {0x3F,0x80,0x00,0x00} (Big Endian) to float

// float = {0x3F800000} (Big Endian)

// float = {1.0}

11.4 profinet_read_input_string()

Read the input table content and convert the data to the string encoded in UTF8.

Syntax 1

```
string profinet_read_input_string(
    int,
    int
)
```

Paramaters

`int` Starting address

`int` The address length to read

Return

`string` Return data in a UTF8 string (ending with 0x00 encountered).

Note

```
string var_value = profinet_read_input_string(148,16)
    // byte[] = {0x54,0x4D,0x35,0x2D,0x37,0x30,0x30,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
    // string = “TM5-700”
string var_value = profinet_read_input_string(148,32)
    // byte[] = {0x30,0x31,0x30,0x36,0x30,0x31,0x31,0x32,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                 0x54,0x4D,0x35,0x2D,0x37,0x30,0x30,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
    // string = “01060112”
string var_value = profinet_read_input_string(148,32)
    // byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6,0x98,0x8E,0xE6,0xA9,0x9F,0xE5,0x99,0xA8,
                 0xE4,0xBA,0xBA,0x31,0x32,0x33,0x34,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}
    // string = “abcd 達明機器人 1234”
string var_value = profinet_read_input_string(148,10)
    // byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6,0x98,0x8E}
    // string = “abcd 達明”
string var_value = profinet_read_input_string(148,8)
    // byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6}
    // string = “abcd 達”
```

11.5 profinet_read_input_bit()

Read the input table content and retrieve the n^{th} bit value of the data byte.

Syntax 1

```
byte profinet_read_input_bit(
    int,
    int
)
```

Paramaters

int	Starting address
int	The n^{th} bit value in the data byte

Return

byte	Return data in byte. Return 1 for bit value == 1. Return 0 for bit value == 0.

Note

```
byte var_value = profinet_read_input_bit(148,0)
    // 0x30      get bit: ”0”
    // 0
byte var_value = profinet_read_input_bit(148,5)
    // 0x30      get bit: “5”
    // 1
```

11.6 profinet_read_output()

Read the output table content.

Syntax 1

```
byte[] profinet_read_output(
    int,
    int
)
```

Paramaters

int	Starting address
int	The address length to read

Return

`byte[]` Return data in a byte array.

Note

byte[] var_value = **profinet_read_output**(540, 16)

// {0x30,0x31,0x30,0x36,0x30,0x31,0x31,0x32,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

Syntax 2

```
byte profinet_read_output(
    int
)
```

Paramaters

`int` Starting address

Return

`byte` Return data in byte

Note

byte var_value = **profinet_read_output**(540)

// 0x30

11.7 profinet_read_output_int()

Read the output table content and convert the data to the 32-bit integer.

Syntax 1

```
int[] profinet_read_output_int(
    int,
    int,
    int
)
```

Paramaters

`int` Starting address

`int` The address length to read

`int` The conversion of the read data to an int array based on Little Endian (DCBA) or Big Endian (ABCD).

- `0` Little-Endian
- `1` Big-Endian
- `2` Based on the configuration file.

Return

`int[]` Return data in an integer array.

Note

int[] var_value = **profinet_read_output_int**(556, 12, 0)

// byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian) to int[]

// int[] = {0x00007FFF,0x0001869F,0xFFFF8000} (Little Endian)

// int[] = {32767,99999,-32768}

int[] var_value = **profinet_read_output_int**(556, 11, 0)

// byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF} (Little Endian) to int[]

// int[] = {0x00007FFF,0x0001869F,0x00FF8000} (Little Endian)

// int[] = {32767,99999,16744448}

int[] var_value = **profinet_read_output_int**(556, 10, 0)

// byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80} (Little Endian) to int[]

// int[] = {0x00007FFF,0x0001869F,0x00008000} (Little Endian)

// int[] = {32767,99999,32768}

int[] var_value = **profinet_read_output_int**(556, 12, 1)

// byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian) to int[]

// int[] = {0xFF7F0000,0x9F860100,0x0080FFFF} (Big Endian)

// int[] = {-8454144,-1618607872,8454143}

int[] var_value = **profinet_read_output_int**(556, 12, 2)

// byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian) to int[]

// int[] = {0x00007FFF,0x0001869F,0xFFFF8000} (Little Endian)

// int[] = {32767,99999,-32768}

Syntax 2

```
int[] profinet_read_output_int(
    int,
    int
)
```

Note

Same as Syntax 1 with the parameter of the conversion of the read data defaults to 2.

* Convert the read data to an int array based on Little Endian (DCBA) or Big Endian (ABCD).

```
int[] var_value = profinet_read_output_int(556, 12, 2)
    // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian)      to int[]
    // int[] = {0x00007FFF,0x0001869F,0xFFFF8000} (Little Endian)
    // int[] = {32767,99999,-32768}
```

Syntax 3

```
int profinet_read_output_int(
    int
)
```

Paramaters

`int` Starting address

* Convert the read data to an int array based on Little Endian (DCBA) or Big Endian (ABCD).

Return

`int` Return data in integer

Note

```
int var_value = profinet_read_output_int(556)
    // byte[] = {0xE4,0x07,0x00,0x00} (Little Endian)      to int
    // int = 0x000007E4 (Little Endian)
    // int = 2020
int var_value = profinet_read_output_int(556)
    // byte[] = {0x00,0x00,0x07,0xE4} (Big Endian)      to int
    // int = 0x000007E4 (Big Endian)
    // int = 2020
```

11.8 profinet_read_output_float()

Read the output table content and convert the data to the 32-bit floating-point number.

Syntax 1

```
float[] profinet_read_output_float(
    int,
    int,
    int
)
```

Paramaters

`int` Starting address

`int` The address length to read

`int` The conversion of the read data to a float array based on Little Endian (DCBA) or Big Endian (ABCD).

`0` Little-Endian

`1` Big-Endian

`2` Based on the configuration file.

Return

`float[]` Return data in a floating-point number array.

Note

float[] var_value = **profinet_read_output_float**(676, 12, 0)

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian) to float[]

// float[] = {0x3F800000,0x40000000,0x40400000} (Little Endian)

// float[] = {1.0,2.0,3.0}

float[] var_value = **profinet_read_output_float**(676, 11, 0)

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40} (Little Endian) to float[]

// float[] = {0x3F800000,0x40000000,0x00400000} (Little Endian)

// float[] = {1.0,2.0,5.877472E-39}

float[] var_value = **profinet_read_output_float**(676, 10, 0)

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00} (Little Endian) to float[]

// float[] = {0x3F800000,0x40000000,0x00000000} (Little Endian)

// float[] = {1.0,2.0,0.0}

float[] var_value = **profinet_read_output_float**(676, 12, 1)

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian) to float[]

// float[] = {0x0000803F,0x00000040,0x00004040} (Big Endian)

// float[] = {4.600603E-41,8.96831E-44,2.304856E-41}

float[] var_value = **profinet_read_output_float**(676, 12, 2)

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian) to float[]

// float[] = {0x3F800000,0x40000000,0x40400000} (Little Endian)

// float[] = {1.0,2.0,3.0}

Syntax 2

```
float[] profinet_read_output_float(
    int,
    int
)
```

Note

Same as Syntax 1 with the parameter of the conversion of the read data defaults to 2.

* Convert the read data to a float array based on Little Endian (DCBA) or Big Endian (ABCD).

float[] var_value = **profinet_read_output_float**(676, 12, 2)

// byte[] = {0x00,0x00,0x80,0x3F,0x00,0x00,0x00,0x40,0x00,0x00,0x40,0x40} (Little Endian) to float[]

// float[] = {0x3F800000,0x40000000,0x40400000} (Little Endian)

// float[] = {1.0,2.0,3.0}

Syntax 3

```
float profinet_read_output_float(
    int
)
```

Paramaters

`int` Starting address

* Convert the read data to a float array based on Little Endian (DCBA) or Big Endian (ABCD).

Return

`float` Return data in floating-point

Note

float var_value = **profinet_read_output_float**(676)

// byte[] = {0x00,0x00,0x80,0x3F} (Little Endian) to float

// float = 0x3F800000 (Little Endian)

// float = 1.0

float var_value = **profinet_read_output_float**(676)

// byte[] = {0x3F,0x80,0x00,0x00} (Big Endian) to float

// float = 0x3F800000 (Big Endian)

// float = 1.0

11.9 profinet_read_output_string()

Read the output table content and convert the data to the string encoded in UTF8.

Syntax 1

```
string profinet_read_output_string(
    int,
    int
```

)

Paramaters

`int` Starting address

`int` The address length to read

Return

`string` Return data in a UTF8 string (ending with 0x00 encountered).

Note

string var_value = **profinet_read_output_string**(556,16)

// byte[] = {0x54,0x4D,0x35,0x2D,0x37,0x30,0x30,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// string = “TM5-700”

string var_value = **profinet_read_output_string**(540,32)

// byte[] = {0x30,0x31,0x30,0x36,0x30,0x31,0x31,0x32,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x54,0x4D,0x35,0x2D,0x37,0x30,0x30,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// string = “01060112”

string var_value = **profinet_read_output_string**(540,32)

// byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6,0x98,0x8E,0xE6,0xA9,0x9F,0xE5,0x99,0xA8,
0xE4,0xBA,0xBA,0x31,0x32,0x33,0x34,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// string = “abcd 達明機器人 1234”

string var_value = **profinet_read_output_string**(540,10)

// byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6,0x98,0x8E}

// string = “abcd 達明”

string var_value = **profinet_read_output_string**(540,8)

// byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6}

// string = “abcd 達”

11.10 profinet_read_output_bit()

Read the output table content and retrieve the n^{th} bit value of the data byte.

Syntax 1

```
byte profinet_read_output_bit(
    int,
    int
)
```

Paramaters

`int`	Starting address
`int`	The nth bit value in the data byte

Return

`byte`	Return data in byte 型別。
	Return 1 for bit value == 1.
	Return 0 for bit value == 0.

Note

byte var_value = **profinet_read_output_bit**(540,0)

// 0x30 get bit: ”0”

// 0

byte var_value = **profinet_read_output_bit**(540,5)

// 0x30 get bit: “5”

// 1

11.11 profinet_write_output()

Write data to the output table.

Syntax 1

```
bool profinet_write_output (
    int,
    ?,
    int
)
```

Paramaters

`int`	Starting address	
`?`	The data to write	
	* Available data types include **`byte`**,**`byte[]`**,**`int`**,**`int[]`**,**`float`**,**`float[]`**,**`string`**	
`int`	The maximum data writing length	
	> 0	Legitimate data length. Write by the number of data length.
	<= 0	Illegitimate data length. Write by the complete length of the data to write.

Return

`bool`	`True`	Write successfully.
	`False`	Write unsuccessfully.

1. If the data to write is an empty string or an emprty array
2. Unable to send correctly and receive via Profinet.

Note

* Write data based on Little Endian (DCBA) or Big Endian (ABCD) in the configuration file.

```
byte var_data = 255
profinet_write_output(540,var_data,1)
byte var_value = profinet_read_output(540)
    // 0x30
byte[] var_data = {1,127,255}
profinet_write_output(540,var_data,3)
byte[] var_value = profinet_read_output(540, 3)
    // {0x01,0x7F,0xFF}
profinet_write_output(540,var_data,2)
byte[] var_value = profinet_read_output(540, 3)
    // {0x01,0x7F,0x00}
profinet_write_output(540,var_data,-1)
byte[] var_value = profinet_read_output(540, 3)
    // {0x00,0x7F,0xFF}

int var_data = 32767
profinet_write_output(556,var_data,4)
int var_value = profinet_read_output_int(556)
    // byte[] = {0xFF,0x7F,0x00,0x00} (Little Endian)      to int
    // int = 0x00007FFF (Little Endian)
    // int = 32767
profinet_write_output(556, var_data,1)
int var_value = profinet_read_output_int(556)
    // byte[] = {0xFF,0x00,0x00,0x00} (Little Endian)      to int
    // int = 0x000000FF (Little Endian)
    // int = 255
int[] var_data = {32767,99999,-32768}
profinet_write_output(556, var_data,12)
int[] var_value = profinet_read_output_int(556,12)
```

```
        // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0xFF} (Little Endian)      to int[]
        // int[] = {0x00007FFF,0x0001869F,0xFFFF8000} (Little Endian)
        // int[] = {32767,99999,-32768}
profinet_write_output(556, var_data,3)
int[] var_value = profinet_read_output_int(556,12)
        // byte[] = {0xFF,0x7F,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00} (Little Endian)      to int[]
        // int[] = {0x00007FFF,0x00000000,0x00000000} (Little Endian)
        // int[] = {32767,0,0}
profinet_write_output(556, var_data,11)
int[] var_value = profinet_read_output_int(556,12)
        // byte[] = {0xFF,0x7F,0x00,0x00,0x9F,0x86,0x01,0x00,0x00,0x80,0xFF,0x00} (Little Endian)      to int[]
        // int[] = {0x00007FFF,0x0001869F,0x00FF8000} (Little Endian)
        // int[] = {32767,99999,16744448}


float var_data = -10.0
profinet_write_output(676, var_data,4)
float var_value = profinet_read_output_float(676)
        // byte[] = {0x00,0x00,0x20,0xC1} (Little Endian)        to float
        // float = 0xC1200000 (Little Endian)
        // float = -10.0
profinet_write_output(676, var_data,1)
float var_value = profinet_read_output_float(676)
        // byte[] = {0x00,0x00,0x00,0x00} (Little Endian)        to float
        // float = 0x00000000 (Little Endian)
        // float = 0
float[] var_data = {-10.0,3.3,123.45}
profinet_write_output(676, var_data,12)
float[] var_value = profinet_read_output_float(676,12)
        // byte[] = {0x00,0x00,0x20,0xC1,0x33,0x33,0x53,0x40,0x66,0xE6,0xF6,0x42} (Little Endian)      to float[]
        // float[] = {0xC1200000,0x40533333,0x42F6E666} (Little Endian)
        // float[] = {-10,3.3,123.45}
profinet_write_output(676, var_data,3)
```

float[] var_value = **profinet_read_output_float**(676,12)

// byte[] = {0x00,0x00,0x20,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00} (Little Endian) to float[]

// float[] = {0x00200000,0x00000000,0x00000000} (Little Endian)

// float[] = {2.938736E-39,0,0}

profinet_write_output(676, var_data,11)

float[] var_value = **profinet_read_output_float**(676,12)

// byte[] = {0x00,0x00,0x20,0xC1,0x33,0x33,0x53,0x40,0x66,0xE6,0xF6,0x00} (Little Endian) to float[]

// float[] = {0xC1200000,0x40533333,0x00F6E666} (Little Endian)

// float[] = {-10,3.3,2.267418E-38}

string var_data = "abcd 達明機器人 1234"

profinet_write_output(540, var_data,32)

string var_value = **profinet_read_output_string**(540,32)

// byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6,0x98,0x8E,0xE6,0xA9,0x9F,0xE5,0x99,0xA8,

0xE4,0xBA,0xBA,0x31,0x32,0x33,0x34,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// string = "abcd 達明機器人 1234"

profinet_write_output(540, var_data,10)

string var_value = **profinet_read_output_string**(540,32)

// byte[] = { 0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6,0x98,0x8E,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// string = "abcd 達明"

profinet_write_output(540, var_data,8)

string var_value = **profinet_read_output_string**(540,32)

// byte[] = {0x61,0x62,0x63,0x64,0xE9,0x81,0x94,0xE6,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,

0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// string = "abcd 達"

11.12 profinet_write_output_bit()

Write content to the nth bit value of the data byte in the output table.

Syntax 1

```
bool profinet_write_output_bit(
```

```
    int,
    int,
    int
)
```

Paramaters

`int`	Starting address
`int`	The n^{th} bit value in the data byte
`int`	The data to write

Return

`bool`	`True`	Write successfully.	
	`False`	Write unsuccessfully.	1. Unable to send correctly and receive via Profinet.

Note

```
byte var_data = 240
profinet_write_output(540,var_data)
byte var_value = profinet_read_output(540)
    // 0xF0
profinet_write_output_bit(540,1,1)
byte var_value = profinet_read_output_bit(540,1)
    // 0xF2      get bit: ”1”
    // 1
profinet_write_output_bit(540,7,0)
byte var_value = profinet_read_output_bit(540,7)
    // 0x72      get bit: “7”
    // 0
```

TECHMAN
ROBOT

www.tm-robot.com