



Expression Editor

Software version: 1.66
Document version: 1.01
Release date: 2018-10-01

This Manual contains information of the Techman Robot product series (hereinafter referred to as the TM Robot). The information contained herein is the property of Techman Robot Inc. (hereinafter referred to as the Corporation). No part of this publication may be reproduced or copied in any way, shape or form without prior authorization from the Corporation. No information contained herein shall be considered an offer or commitment. It may be subject to change without notice. This Manual will be reviewed periodically. The Corporation will not be liable for any error or omission.

TM logos are registered trademarks of TECHMAN ROBOT INC. and the company reserves the ownership of this manual and its copy and its copyrights.

TM TECHMAN ROBOT INC.

REVISION HISTORY TABLE.....	6
1. EXPRESSION	7
1.1 Types.....	7
1.2 Variables and Constants	7
1.3 Array	11
1.4 Operator Symbols	11
1.5 Warning	14
2. FUNCTIONS	16
2.1 Byte_ToInt16().....	16
2.2 Byte_ToInt32().....	17
2.3 Byte_ToFloat().....	18
2.4 Byte_ToDouble().....	19
2.5 Byte_ToInt16Array()	20
2.6 Byte_ToInt32Array()	22
2.7 Byte_ToFloatArray().....	23
2.8 Byte_ToDoubleArray()	24
2.9 Byte_ToString().....	25
2.10 Byte_Concat()	26
2.11 String_ToInteger().....	30
2.12 String_ToFloat().....	32
2.13 String_ToDouble().....	34
2.14 String_ToByte().....	36
2.15 String_IndexOf()	38
2.16 String_LastIndexOf()	39
2.17 String_Substring()	40
2.18 String_Split().....	44
2.19 String_Replace()	46

2.20	<i>String_Trim()</i>	47
2.21	<i>String_ToLower()</i>	48
2.22	<i>String_ToUpper()</i>	48
2.23	<i>Array_Equals()</i>	49
2.24	<i>Array_IndexOf()</i>	51
2.25	<i>Array_LastIndexOf()</i>	52
2.26	<i>Array_Reverse()</i>	53
2.27	<i>Array_Sort()</i>	56
2.28	<i>Array_SubElements()</i>	57
2.29	<i>ValueReverse()</i>	59
2.30	<i>GetBytes()</i>	64
2.31	<i>GetString()</i>	69
2.32	<i>GetToken()</i>	75
2.33	<i>GetAllTokens()</i>	84
2.34	<i>GetNow()</i>	86
2.35	<i>GetNowStamp()</i>	88
2.36	<i>Length()</i>	89
2.37	<i>Ctrl()</i>	91
2.38	<i>XOR8()</i>	93
2.39	<i>SUM8()</i>	95
2.40	<i>SUM16()</i>	97
2.41	<i>SUM32()</i>	99
2.42	<i>CRC16()</i>	101
2.43	<i>CRC32()</i>	104
3.	<i>MODBUS FUNCTIONS</i>	106
3.1	<i>modbus_read()</i>	106
3.2	<i>modbus_read_int16()</i>	109
3.3	<i>modbus_read_int32()</i>	111

3.4	<i>modbus_read_float()</i>	113
3.5	<i>modbus_read_double()</i>	115
3.6	<i>modbus_read_string()</i>	117
3.7	<i>modbus_write()</i>	119

Revision History Table

Revision	Date	Revised Content
1.00	2018-09-05	Original release
1.01	2018-10-01	Modify document name

1. Expression

1.1 Types

Different data types of variable can be declared in Variables Manager

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

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

int16	16bit interger	signed	-32768 to 32767	significant digit 5
int32	32bit interger	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 charactors are supported.

Numbers 0123456789

Charactors a-z, A-Z, _

Example

int i = 0

string s = "ABC"

string s1 = "DEF"

string s2 = "123"

When utilizing variables, double quotation marks shall not be applied as shown below:

```
s = s1 + " and " + s2    // s = "DEF and 123"
                        // s, s1, s2 are variable, and " and " is a string.
```

When using constant, including number, string, and booling value, only sting shall double quotation marks be applied to.

2. Numbers

- Decimal integer, decimal float number, 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 float number.
- 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 float number 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 b = -100 // b = 156 // -100 is present as 0xFFFFF9C 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 ("") in stead of one (").

- Control charactor in double quation mark is 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 charactors 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 combiled as variables

For example:

```
s = "TM5"           // s = "TM5"
s1 = "Hi, s Robot"  // 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:

```
s1 = "Hi, " + s + " Robot"  // 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"           // s1 = "Hi, TM5 Robot"
                        "Hi, 'x' Robot"           // s1 = "Hi, 'x' Robot" // Because variable x does not exist, 'x' is viewed as
string
```

4. Single quotation marks can not be presented by "'". If the users would like to input '(variable name)', the standard format with double quation marks should be used.

For example

```
"Hi, 's' Robot"           // s1 = "Hi, TM5 Robot"
```

// If s1 = "Hi, 's' Robot" is what you want, please use the following syntax.

"Hi, " + "s" + " Robot" // s1 = "Hi, 's' Robot"

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

For example

s1 = "Hi, " + Ctrl("\r\n") + s + " Robot" or "Hi, " + NewLine + 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 或 NewLine new line, equals to Ctrl("\r\n") or Ctrl(0x0D0A)

4. Boolean

True or false value of logic.

Represent true value true

True

Represent false value false

False

The Boolean value is case sensitive. If the eligal notation is applied, it will beviewed as variable or 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[] i = {0,1,2,3}           // elements in number data type
string[] s = {"ABC", "DEF", "GHI"} // elements in string data type
bool[] 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								
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]

eight elements in total

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 is dynamic, which may alter according to the return value of functions or assigned values. The maximum element number is 2048. This feature make array meets the needs of different functions and applications in Network Node.

For Example:

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

1.4 Operator Symbols

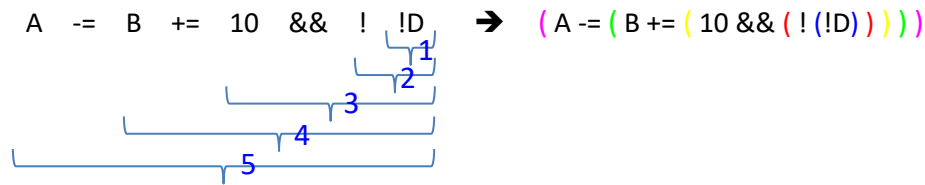
- The operator table is listed below.
- The calculation follows the precedence of operator first then the associativity.

For example

left-to-right associativity

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

right-to-left associativity



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)		
3	=	Basic assignment	int a = b	Left side: Numeric variable Right side: Numeric variable, Constant	right-to-left
	+=	Addition assignment	a += 3		
	-=	Subtraction assignment	b -= 4		
	*=	Multiplication assignment	a *= 5		
	/=	Division assignment	a /= 2		
	%=	Modulo assignment	a %= 3	Left side: Integer variable Right side: Integer variable, Constant	
	<<=	Bitwise left shift assignment	flags <<= 2		
	>>=	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 Warning

A warning message will be send out, under the condition listed below.

- Double quotation marks does not placed around the string constant.
- There is single quotation marks inside the string constant.
- When assigning float value to integer constant, some digits may get lost

For example

```
int i = 1.234    // warning i = 1
float f = 1.234
i = f           // warning i = 1
```

- When assigning value to variables with fewer digits, some digits may get lost

For example

```
byte b = 100
int i = 1000
float f = 1.234
double d = 2.345
b = i           // warning b = 232    // byte can contain values from 0 to 255
f = d           // warning 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 be send out, or the project will be stopped by error.

For example

```
int i = "1234"    // warning i = 1234
int j = "0x89AB"  // warning j = 35243
int k = "0b1010"  // warning k = 10
string s1 = 123    // warning s1 = 123 // Number to string
string s2 = "123"
int x = s2         // warning // string to number
// The following code can be compiled with warning, but will be stopped by error when executing.
S2 = "XYZ"
x = s2             // warning // Stop executing by error// s = "XYZ" cannot be converted to number
s2 = ""
x = s2            // warning // Stop executing by error// s = "" cannot be converted to number
```

- String parameters is used as numeric parameters in functions.

For example

```
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 warning message, correcting all the errors in warning message is highly recommended, which can eliminate unpredictable problems and prevent the project being stopped by errors.

2. Functions

2.1 Byte_ToInt16()

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

Syntax 1

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

Parameters

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

Return

`int` A 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[] bb1 = {0x90, 0x01, 0x05}
byte[] bb2 = {0x01} // Cause bb2[] does not fill 2 bytes. It would be filled to 2 bytes before transforming.
value = Byte_ToInt16(bb1, 0) // 0x0201 value = 513
value = Byte_ToInt16(bb1, 1) // 0x0102 value = 258
value = Byte_ToInt16(bb2, 0) // 0x0001 value = 1
value = Byte_ToInt16(bb2, 1) // 0x0100 value = 256
```

Syntax 2

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

Note

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

Byte_ToInt16(bb1) => Byte_ToInt16(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` A 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[] bb1 = {0x01, 0x02, 0x03, 0x4F, 1}`

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

`value = Byte_ToInt32(bb1, 0) // 0x4F030201 value = 1325597185`

`value = Byte_ToInt32(bb1, 1) // 0x0102034F value = 16909135`

`value = Byte_ToInt32(bb2, 0) // 0x00030201 value = 197121`

`value = Byte_ToInt32(bb2, 1) // 0x01020300 value = 16909056`

Syntax 2

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

Note

Similar to Syntax 1 with little endian input

`Byte_ToInt32(bb1) => Byte_ToInt32(bb1, 0)`

2.3 Byte_ToFloat()

Transform the first four bytes of byte array to float number, and return in float 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[] bb1 = {0x01, 0x02, 0x03, 0x4F, 1}`

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

`value = Byte_ToFloat(bb1, 0) // 0x4F030201 value = 2.197947E+09`

`value = Byte_ToFloat(bb1, 1) // 0x0102034F value = 2.38796E-38`

`value = Byte_ToFloat(bb2, 0) // 0x00030201 value = 2.762254E-40`

`value = Byte_ToFloat(bb2, 1) // 0x01020300 value = 2.387938E-38`

Syntax 2

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

Note

Similar to Syntax 1 with little endian input

`Byte_ToFloat(bb1) => Byte_ToFloat(bb1, 0)`

2.4 Byte_ToDouble()

Transform the first eight bytes of byte array to float 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[] bb1 = {0x01, 0x02, 0x03, 0x4F, 1} // Cause bb1[] does not fill 8 bytes. It would be filled to 8 bytes before transforming.`

`byte[] bb2 = {0x01, 0x02, 0x03} // Cause bb1[] does not fill 8 bytes. It would be filled to 8 bytes before transforming.`

```
value = Byte_ToDouble(bb1, 0) // 0x000000014F030201 value = 2.77692782029764E-314
value = Byte_ToDouble(bb1, 1) // 0x0102034F01000000 value = 8.20840179153173E-304
value = Byte_ToDouble(bb2, 0) // 0x0000000000030201 value = 9.73907141738724E-319
value = Byte_ToDouble(bb2, 1) // 0x0102030000000000 value = 8.20785244926136E-304
```

Syntax 2

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

Note

Similar to Syntax 1 with little endian input

Byte_ToDouble(bb1) => **Byte_ToDouble**(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  
)
```

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 2 bytes of byte array beginning at index [0]

Note

byte[] 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[] bb2 = {1, 2, 3, 4}

value = Byte_ToInt16Array(bb1, 0) // {0x0201, 0x0403, 0x0005} value = {513, 1027, 5}

value = Byte_ToInt16Array(bb1, 1) // {0x0102, 0x0304, 0x0500} value = {258, 772, 1280}

value = Byte_ToInt16Array(bb2, 0) // {0x0201, 0x0403} value = {513, 1027}

value = Byte_ToInt16Array(bb2, 1) // {0x0102, 0x0304} value = {258, 772}

Note

Similar to Syntax 1 with return value as signed int16

Byte_ToInt16Array(bb1, 0) => **Byte_ToInt16Array**(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(bb1) => **Byte_ToInt16Array**(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[] 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[] bb2 = {1, 2, 3, 4}`

`value = Byte_ToInt32Array(bb1, 0) // {0x04030201, 0x00000005} value = {67305985, 5}`

`value = Byte_ToInt32Array(bb1, 1) // {0x01020304, 0x05000000} value = {16909060, 83886080}`

`value = Byte_ToInt32Array(bb2, 0) // {0x04030201} value = {67305985}`

`value = Byte_ToInt32Array(bb2, 1) // {0x01020304} value = {16909060}`

Syntax 2

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

Note

Similar to Syntax 1 with little endian input.

`Byte_ToInt32Array(bb1) => Byte_ToInt32Array(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[] 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[] bb2 = {1, 2, 3, 4}`

`value = Byte_ToFloatArray(bb1, 0)` // {0x04030201, 0x00000005} value = {1.53999E-36, 7.006492E-45}

`value = Byte_ToFloatArray(bb1, 1)` // {0x01020304, 0x05000000} value = {2.387939E-38, 6.018531E-36}

`value = Byte_ToFloatArray(bb2, 0)` // {0x04030201} value = {1.53999E-36}

`value = Byte_ToFloatArray(bb2, 1)` // {0x01020304} value = {2.387939E-38}

Syntax 2

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

Note

Similar to Syntax 1 with little endian input

`Byte_ToFloatArray(bb1) => Byte_ToFloatArray(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[] 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[] bb2 = {1, 2, 3, 4} // When the remaining part does not fill 8 byte, it would be filled to 8 bytes before transforming.

```
value = Byte_ToDoubleArray(bb1, 0) // {0x0000000504030201} value = {1.06432325297744E-313}
value = Byte_ToDoubleArray(bb1, 1) // {0x0102030405000000} value = {8.20788039849233E-304}
value = Byte_ToDoubleArray(bb2, 0) // {0x0000000004030201} value = {3.32535749480063E-316}
value = Byte_ToDoubleArray(bb2, 1) // {0x0102030400000000} value = {8.2078802626846E-304}
```

Syntax 2

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

Note

Similar to Syntax 1 with little endian input

Byte_ToDoubleArray(bb1) => Byte_ToDoubleArray(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[] bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}

byte[] bb2 = {0x01, 0x54, 0x4D, 0x35, 0xE6, 0xA9, 0x9F, 0xE5, 0x99, 0xA8, 0xE4, 0xBA, 0xBA}

value = **Byte_ToString**(bb1, 0) // value = "123" (UTF8 stop at 0x00)

value = **Byte_ToString**(bb1, 1) // value = "313233004F01"

value = **Byte_ToString**(bb1, 2) // value = "123" (ASCII stop at 0x00)

value = **Byte_ToString**(bb2, 0) // value = "\u01TM5\u4eba" (UTF8)

value = **Byte_ToString**(bb2, 1) // value = "01544D35E6A99FE599A8E4BABA"

value = **Byte_ToString**(bb2, 2) // value = "\u01TM5?????????" (ASCII)

* \u01 represent 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(bb1) => **Byte_ToString**(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[] bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}`

`value = Byte_Concat(bb1, 12) // 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[] bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}`

`byte[] bb2 = {0x01, 0x02, 0x03}`

`value = Byte_Concat(bb1, bb2) // 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[] bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}

byte[] bb2 = {0x01, 0x02, 0x03}

value = **Byte_Concat**(bb1, bb2, 2) // value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02} // Concatenate only 2 elements from array2

value = **Byte_Concat**(bb1, bb2, -1) // value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02, 0x03} // -1 is invalid value

value = **Byte_Concat**(bb1, bb2, 10) // value = {0x31, 0x32, 0x33, 0x00, 0x4F, 0x01, 0x01, 0x02, 0x03} // 10 exceeds the array size

// Length() can be utilized to acquire the array size

value = **Byte_Concat**(bb1, bb2, Length(bb2)) // 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[] bb1 = {0x31, 0x32, 0x33, 0x00, 0x4F, 1}

byte[] bb2 = {0x01, 0x02, 0x03}

value = **Byte_Concat**(bb1, 1, 3, bb2, 1, 2) // value = {0x32, 0x33, 0x00, 0x02, 0x03}

value = **Byte_Concat**(bb1, -1, 3, bb2, 3, -1) // 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

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

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

<code>string[]</code>	Input string array
<code>int</code>	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 notation of all the elements in a single array has to be identical	

Return

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

Note

```
ss = {"12", "ab", "cc", "dd", "10"}
value = String_ToInteger(ss)           // value = {12, 0, 0, 0, 10} // "ab","cc","dd" are invalid decimal numbers
value = String_ToInteger(ss, 2)        // value = {0, 0, 0, 0, 2} // "12","ab","cc","dd" are invalid binary numbers
value = String_ToInteger(ss, 16)       // value = {18, 171, 204, 221, 16}
value = String_ToInteger(ss, 10)       // value = {12, 0, 0, 0, 10} // "ab","cc","dd" are invalid decimal numbers
```

2.12 String_ToFloat()

Transform string to float number (float type)

Syntax 1

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

Parameters

<code>string</code>	Input string
<code>int</code>	Input string's notation is decimal, hexadecimal or binary format
<code>10</code>	decimal or auto format detecting (Default)
<code>16</code>	hexadecimal
<code>2</code>	binary
	String's notation
<code>123</code>	decimal
<code>0x7F</code>	hexadecimal
<code>0b101</code>	binary

Return

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

Note

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

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

<code>string[]</code>	Input string array
<code>int</code>	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

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

Note

```
ss = {"12.345", "ab", "cc", "dd", "10.111"}  
value = String_ToFloat(ss)           // value = {12.345, 0, 0, 0, 10.111}  
value = String_ToFloat(ss, 2)        // value = {0, 0, 0, 0, 0}  
value = String_ToFloat(ss, 16)       // value = {0, 2.39622E-43, 2.858649E-43, 3.09687E-43, 0}  
value = String_ToFloat(ss, 10)       // value = {12.345, 0, 0, 0, 10.111}
```

2.13 String_ToDouble()

Transform string to float 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

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

```
value = String_ToDouble("-0b1100", 2)    // value = 0    // Invalid 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

<code>string[]</code>	Input string array
<code>int</code>	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

```
ss = {"12.345", "ab", "cc", "dd", "10.111"}
```

```
value = String_ToDouble(ss)    // value = {12.345, 0, 0, 0, 10.111}
```

```
value = String_ToDouble(ss, 2) // value = {0, 0, 0, 0, 0}
```

```
value = String_ToDouble(ss, 16) // value = {0, 8.44852254388532E-322, 1.00789391751614E-321, 1.09188507730915E-321, 0}
```

```
value = String_ToDouble(ss, 10) // 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

```
value = String_ToByte("12345", 0) // value = {0x31, 0x32, 0x33, 0x34, 0x35}  
value = String_ToByte("12345", 1) // value = {0x12, 0x34, 0x50} // the insufficient part will be filled  
                                // with 0  
value = String_ToByte("12345", 2) // value = {0x31, 0x32, 0x33, 0x34, 0x35}  
value = String_ToByte("0x12345", 0) // value = {0x30, 0x78, 0x31, 0x32, 0x33, 0x34, 0x35}  
value = String_ToByte("0x12345", 1) // value = {0x00} // Only 0 be transformed, cause x is an  
                                // invalid Hex value  
value = String_ToByte("0x12345", 2) // value = {0x30, 0x78, 0x31, 0x32, 0x33, 0x34, 0x35}  
value = String_ToByte("TM5機器人", 0) // value = {0x54, 0x4D, 0x35, 0xE6, 0xA9, 0x9F, 0xE5, 0x99, 0xA8, 0xE4, 0xBA, 0xBA}  
value = String_ToByte("TM5機器人", 1) // value = {0x00} // T is an invalid Hex value  
value = String_ToByte("TM5機器人", 2) // value = {0x54, 0x4D, 0x35, 0x3F, 0x3F, 0x3F}  
  
value = String_ToByte("0123456", 1) // value = {0x01, 0x23, 0x45, 0x60}  
value = String_ToByte("01234G5", 1) // 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()

Reports 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

value = String_IndexOf("012314", "1")	// value = 1
value = String_IndexOf("012314", "")	// value = 0
value = String_IndexOf("012314", empty)	// value = 0
value = String_IndexOf("012314", "d")	// value = -1
value = String_IndexOf("", "d")	// value = -1

2.16 String_LastIndexOf()

Reports 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

<code>int</code>	<code>0..(Length of string)-1</code>	If the specified string is found, returns the index number
	<code>-1</code>	Not found
	<code>0</code>	The specified string is "" or empty

Note

<code>value = String_LastIndexOf("012314", "1")</code>	<code>// value = 4</code>
<code>value = String_LastIndexOf("012314", "")</code>	<code>// value = 5</code>
<code>value = String_LastIndexOf("012314", empty)</code>	<code>// value = 5</code>
<code>value = String_LastIndexOf("012314", "d")</code>	<code>// value = -1</code>
<code>value = String_LastIndexOf("", "d")</code>	<code>// value = -1</code>

2.17 String_Substring()

Retrieves 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

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

```
value = String_Substring("0x12345", "1", 4)    // value = "1234"  
value = String_Substring("0x12345", "", 4)     // value = "0x12"  
value = String_Substring("0x12345", "7", 4)    // value = ""  
value = String_Substring("0x12345", "1", -1)   // value = "12345"  
value = String_Substring("0x12345", "1", 100)  // 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

```
value = String_Substring("0x12345", "", "", 0)           // value = "0x12345"  
value = String_Substring("0x12345", "1", "4", 1)         // value = "1234"  
value = String_Substring("0x12345", "1", "4", 2)         // value = ""  
value = String_Substring("0x12345", "1", "4", 0)         // value = ""  
value = String_Substring("0x123450x12-345", "1", "4", 1)  // value = "1234"  
value = String_Substring("0x123450x12-345", "1", "4", 2)  // value = "12-34"  
value = String_Substring("0x123450x12-345", "1", "4", 3)  // value = ""  
value = String_Substring("0x12345122", "1", "", 1)        // value = "12345122" // All the character after
```

prefix

```
value = String_Substring("0x12345122", "1", "", 2)       // value = "122"  
value = String_Substring("0x12345122", "1", "", 4)       // value = ""  
value = String_Substring("0x12345433", "", "4", 1)       // value = "0x123454" // All the character before
```

suffix

```
value = String_Substring("0x12345433", "", "4", 2)       // 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()

Splits 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[]` Splitted 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

```
value = String_Split("0x112345", "1", 0)    // value = {"0x", "", "2345"}  
value = String_Split("0x112345", "", 0)     // value = {"0x112345"}  
value = String_Split("0x112345", "1", 1)    // value = {"0x", "2345"}  
s1 = "123, ""456,67"",89"  
value = String_Split(s1, ",", 0)            // value = {"123", ""456", "67"", "89"} // length = 4  
value = String_Split(s1, ",", 2)            // 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()

Returns 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

```
value = String_Replace("0x112345", "1", "2")    // value = "0x222345"  
value = String_Replace("0x112345", "", "2")     // value = "0x112345"  
value = String_Replace("0x112345", "1", "")     // value = "0x2345"
```

2.20 String_Trim()

Returns 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

```
value = String_Trim("0x112345 ") // value = "0x112345"  
value = String_Trim(" 0x112345") // value = "0x112345"  
value = String_Trim(" 0x112345 ") // 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					

2.21 String_ToLower()

Change all the character in 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

```
value = String_ToLower("0x11Acz34")    // value = "0x11acz34"
```

2.22 String_ToUpper()

Change all the character in 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 be remained the same.

Note

```
value = String_ToUpper("0x11Acz34")    // value = "0X11ACZ34"
```

2.23 Array_Equals()

Determines 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 array1)-1)
?
[] Input array2 (Data type can be byte, int, float, double, bool, string)
int The starting index of array2 (0 .. (length of array2)-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[] n1 = {100, 200, 30}`

`byte[] n2 = {100, 200, 30}`

`Array_Equals(n1, n2)` // true

`Array_Equals(n1, 0, n2, 0, 3)` // true

`Array_Equals(n1, 0, n2, 0, Length(n2))` // true

? `int[] n1 = {1000, 2000, 3000}`

`int[] n2 = {1000, 2000, 3000, 4000}`

`Array_Equals(n1, n2)` // false

`Array_Equals(n1, 0, n2, 0, Length(n2))` // false // compare 4 elements

`Array_Equals(n1, 0, n2, 0, 3)` // true

? `float[] n1 = {1.1, 2.2, 3.3}`

`float[] n2 = {1.1, 2.2}`

`Array_Equals(n1, n2)` // false

`Array_Equals(n1, 0, n2, 0, Length(n2))` // true // compare 2 elements

`Array_Equals(n1, 0, n2, 0, Length(n1))` // false

? `double[] n1 = {100, 200, 300, 3.3, 2.2, 1.1}`

`double[] n2 = {100, 200, 400, 3.3, 2.2, 4.4}`

`Array_Equals(n1, n2)` // false

`Array_Equals(n1, 0, n2, 0, Length(n2))` // false

`Array_Equals(n1, 0, n2, 0, 2)` // true

`Array_Equals(n1, 3, n2, 3, 2)` // true

? `bool[] n1 = {true, false, true, true, true}`

`bool[] n2 = {true, false, true, false, true}`

`Array_Equals(n1, n2)` // false

`Array_Equals(n1, 0, n2, 0, -1)` // false

`Array_Equals(n1, 0, n2, 0, 0)` // true // compare 0 element

? `string[] n1 = {"123", "ABC", "456", "DEF"}`

`string[] n2 = {"123", "ABC", "456", "DEF"}`

`Array_Equals(n1, n2)` // true

`Array_Equals(n1, -1, n2, 0, 4)` // false // Invalid starting index

2.24 Array_IndexOf()

Searches 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[] n = {100, 200, 30}
value = Array_IndexOf(n, 200) // 1
value = Array_IndexOf(n, 2000) // error // 2000 is not byte data
- ? int[] n = {1000, 2000, 3000}
value = Array_IndexOf(n, 200) // -1
- ? float[] n = {1.1, 2.2, 3.3}
value = Array_IndexOf(n, 1.1) // 0
- ? double[] n = {100, 200, 300, 3.3, 2.2, 1.1}
value = Array_IndexOf(n, 1.1) // 5
- ? bool[] n = {true, false, true, true, true}
value = Array_IndexOf(n, true) // 0
- ? string[] n = {"123", "ABC", "456", "DEF"}
value = Array_IndexOf(n, "456") // 2

2.25 Array_LastIndexOf()

Searches 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[] n = {100, 200, 30}
value = Array_LastIndexOf(n, 200) // 1
value = Array_LastIndexOf(n, 2000) // error // 2000 is not byte data
- ? int[] n = {1000, 2000, 3000}
value = Array_LastIndexOf(n, 200) // -1
- ? float[] n = {1.1, 2.2, 3.3}
value = Array_LastIndexOf(n, 1.1) // 0
- ? double[] n = {100, 200, 300, 3.3, 2.2, 1.1}
value = Array_LastIndexOf(n, 1.1) // 5
- ? bool[] n = {true, false, true, true, true}
value = Array_LastIndexOf(n, true) // 4
- ? string[] n = {"123", "ABC", "456", "DEF"}
value = Array_LastIndexOf(n, "456") // 2

2.26 Array_Reverse()

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

```
n = Array_Reverse(n, 2)    // n = {200, 100, 30}    // 2 elements as a section, that is {100,200}{30}  
n = Array_Reverse(n, 4)    // n = {30, 200, 100}    // 4 elements as a section, that is {100,200,30}  
n = Array_Reverse(n, 8)    // n = {30, 200, 100}
```

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

```
n = Array_Reverse(n, 2)    // n = {200, 100, 400, 300}    // 2 elements as a section, that is {100,200}{300,400}  
n = Array_Reverse(n, 4)    // n = {400, 300, 200, 100}    // 4 elements as a section, that is {100,200,300,400}  
n = Array_Reverse(n, 8)    // n = {400, 300, 200, 100}
```

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

```
n = Array_Reverse(n, 2)    // 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}  
n = Array_Reverse(n, 4)    // 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}  
n = Array_Reverse(n, 8)    // n = {5.5, 4.4, 3.3, 2.2, 1.1}
```

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

```
n = Array_Reverse(n, 2)    // n = {200, 100, 400, 300, 3.3, 4.4, 1.1, 2.2, 60, 50, 80, 70}  
n = Array_Reverse(n, 4)    // n = {400, 300, 200, 100, 1.1, 2.2, 3.3, 4.4, 80, 70, 60, 50}  
n = Array_Reverse(n, 8)    // n = {1.1, 2.2, 3.3, 4.4, 400, 300, 200, 100, 80, 70, 60, 50}
```

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

`n = Array_Reverse(n, 2) // n = {false, true, true, true, false, true, false, true }`

`n = Array_Reverse(n, 4) // n = {true, true, false, true, false, true, false, true}`

`n = Array_Reverse(n, 8) // n = {false, true, false, true, true, true, false, true}`

? `string[] n = {"123", "ABC", "456", "DEF", "000", "111"}`

`n = Array_Reverse(n, 2) // n = {"ABC", "123", "DEF", "456", "111", "000"}`

`n = Array_Reverse(n, 4) // n = {"DEF", "456", "ABC", "123", "111", "000"}`

`n = Array_Reverse(n, 8) // n = {"111", "000", "DEF", "456", "ABC", "123"}`

2.27 Array_Sort()

Sorts 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[] n = {1000, 2000, 3000}  
n = Array_Sort(n)           // n = {1000, 2000, 3000}  
  
? double[] n = {100, 200, 300, 3.3, 2.2, 1.1}  
n = Array_Sort(n, 1)        // n = {300, 200, 100, 3.3, 2.2, 1.1}  
  
? bool[] n = {true, false, true, true, true}  
n = Array_Sort(n, 1)        // n = {true, true, true, true, false}  
  
? string[] n = {"123", "ABC", "456", "DEF"}  
n = Array_Sort(n)           // n = {"123", "456", "ABC", "DEF"}
```

2.28 Array_SubElements()

Retrieves 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 Syntax1, 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[] n = {100, 200, 30}  
  
n1 = Array_SubElements(n, 0)        // n1 = {100, 200, 30}  
n1 = Array_SubElements(n, -1)       // n1 = {}  
n1 = Array_SubElements(n, 0, 3)     // n1 = {100, 200, 30}  
n1 = Array_SubElements(n, 1, 3)     // n1 = {200, 30}
```

```
n1 = Array_SubElements(n, 2)    // n1 = {30}
n1 = Array_SubElements(n, 3, 3) // n1 = {}
```

? **int[]** n = {1000, 2000, 3000}

```
n1 = Array_SubElements(n, 0)    // n1 = {1000, 2000, 3000}
n1 = Array_SubElements(n, -1)   // n1 = {}
n1 = Array_SubElements(n, 1, 3) // n1 = {2000, 3000}
n1 = Array_SubElements(n, 2)    // n1 = {3000}
```

? **float[]** n = {1.1, 2.2, 3.3}

```
n1 = Array_SubElements(n, 0)    // n1 = {1.1, 2.2, 3.3}
n1 = Array_SubElements(n, -1)   // n1 = {}
n1 = Array_SubElements(n, 1, 3) // n1 = {2.2, 3.3}
n1 = Array_SubElements(n, 2)    // n1 = {3.3}
```

? **double[]** n = {100, 200, 3.3, 2.2, 1.1}

```
n1 = Array_SubElements(n, 0)    // n1 = {100, 200, 3.3, 2.2, 1.1}
n1 = Array_SubElements(n, -1)   // n1 = {}
n1 = Array_SubElements(n, 1, 3) // n1 = {200, 3.3, 2.2}
n1 = Array_SubElements(n, 2)    // n1 = {3.3, 2.2, 1.1}
```

? **bool[]** n = {true, false, true, true, true}

```
n1 = Array_SubElements(n, 0)    // n1 = {true, false, true, true, true}
n1 = Array_SubElements(n, -1)   // n1 = {}
n1 = Array_SubElements(n, 1, 3) // n1 = {false, true, true}
n1 = Array_SubElements(n, 2)    // n1 = {true, true, true}
```

? **string[]** n = {"123", "ABC", "456", "DEF"}

```
n1 = Array_SubElements(n, 0)    // n1 = {"123", "ABC", "456", "DEF"}
n1 = Array_SubElements(n, -1)   // n1 = {}
n1 = Array_SubElements(n, 1, 3) // n1 = {"ABC", "456", "DEF"}
n1 = Array_SubElements(n, 2)    // n1 = {"456", "DEF"}
```

2.29 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 i = 10000  
value = ValueReverse(i, 0) // 10000=0x00002710 → 0x10270000 // value = 270991360  
value = ValueReverse(i, 1) // 10000=0x2710 → 0x1027 // value = 4135  
i = 100000 // int32 數值  
value = ValueReverse(i, 0) // 100000=0x000186A0 → 0xA0860100 // value = -1601830656  
value = ValueReverse(i, 1) // 100000=0x000186A0 → 0xA0860100 // value = -1601830656  
value = ValueReverse(i, 2) // 100000=0x000086A0 → 0xA0860000 // 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 i = 40000

value = **ValueReverse**(i) // 40000=0x471C4000 → 0x00401C47 // 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 i = 80000

value = **ValueReverse**(i) // 80000=0x40F3880000000000 → 0x00000000008F340 // 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 i = "ABCDEF"
```

```
value = ValueReverse(i) // 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[] i = {10000, 20000, 60000, 80000}
```

```
value = ValueReverse(i, 0) // value = {270991360, 541982720, 1625948160, -2143813376}
```

```
value = ValueReverse(i, 1) // value = {4135, 8270, 1625948160, -2143813376}
```

```
value = ValueReverse(i, 2) // 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[] i = {10000, 20000}
```

```
value = ValueReverse(i)           // 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[] i = {10000, 20000}
```

```
value = ValueReverse(i)           // 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[] i = {"ABCDEFGH", "12345678"}
```

```
value = ValueReverse(i)           // value = {"GFEDCBA", "87654321"}
```

2.30 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 follows Little Endian or Big Endian
0	Little Endian (Default)
1	Big Endian

Return

byte[] The byte array formed by input data

Syntax 2

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

Note

Similar to Syntax1 with Little Endian

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

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

? **int**

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

? **int[]** n = {10000, 20000, 80000} // Convert every single element of the array to byte. For int32 data, works on 4 bytes sequentially.

```
value = GetBytes(n)          // value = {0x10, 0x27, 0x00, 0x00, 0x20, 0x4E, 0x00, 0x00, 0x80, 0x38, 0x01, 0x00}
value = GetBytes(n, 0)        // value = {0x10, 0x27, 0x00, 0x00, 0x20, 0x4E, 0x00, 0x00, 0x80, 0x38, 0x01, 0x00}
value = GetBytes(n, 1)        // value = {0x00, 0x00, 0x27, 0x10, 0x00, 0x00, 0x4E, 0x20, 0x00, 0x01, 0x38, 0x80}
```

? **float**

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

? **float[]** n = {1.23, 4.56, -7.89} // Convert every single element of the array to byte. For float data, works on 4 bytes sequentially.

```
value = GetBytes(n)          // value = {0xA4, 0x70, 0x9D, 0x3F, 0x85, 0xEB, 0x91, 0x40, 0xE1, 0x7A, 0xFC, 0xC0}
value = GetBytes(n, 0)        // value = {0xA4, 0x70, 0x9D, 0x3F, 0x85, 0xEB, 0x91, 0x40, 0xE1, 0x7A, 0xFC, 0xC0}
value = GetBytes(n, 1)        // value = {0x3F, 0x9D, 0x70, 0xA4, 0x40, 0x91, 0xEB, 0x85, 0xC0, 0xFC, 0x7A, 0xE1}
```

? **double** n = -1.2345

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

? **double[]** n = {1.23, -7.89} // Convert every single element of the array to byte. For double data, works on 8 bytes sequentially.

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

```

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

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

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

value = GetBytes(flag)       // value = {1, 0, 1, 0, 0, 1, 1}
value = GetBytes(flag, 0)     // value = {1, 0, 1, 0, 0, 1, 1}
value = GetBytes(flag, 1)     // value = {1, 0, 1, 0, 0, 1, 1}

?  string n = "ABCDEFGH"      // string 使用 UTF8 轉換

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

?  string[] n = {"ABC", "DEF", "達明機器人"}

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

```

Syntax 3

Convert integer (int type) to byte array.

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

Parameters

- | | |
|------------------|---|
| <code>int</code> | The input integer (int type) |
| <code>int</code> | The input value follows Little Endian or Big Endian |
| 0 | Little Endian (Default) |
| 1 | Big Endian |
| <code>int</code> | 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

- | | |
|---------------------|---|
| <code>byte[]</code> | The byte array formed by input integer. For int32 data, convert with 4 bytes. For int16 data, convert with 2 bytes. |
|---------------------|---|

Note

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

Syntax 4

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

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

Parameters

<code>int[]</code>	The input integer array (int[] type)
<code>int</code>	The input integer array follows Little Endian or Big Endian
0	Little Endian (Default)
1	Big Endian
<code>int</code>	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

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

2.31 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) <table><tr><td>10</td><td>decimal</td></tr><tr><td>16</td><td>hexadecimal</td></tr><tr><td>2</td><td>binary</td></tr></table> <table><tr><td colspan="2">String's notation</td></tr><tr><td>123</td><td>decimal</td></tr><tr><td>0x7F</td><td>hexadecimal</td></tr><tr><td>0b101</td><td>binary</td></tr></table>	10	decimal	16	hexadecimal	2	binary	String's notation		123	decimal	0x7F	hexadecimal	0b101	binary
10	decimal														
16	hexadecimal														
2	binary														
String's notation															
123	decimal														
0x7F	hexadecimal														
0b101	binary														
int	The output string format (Can be only applied to hexadecimal or binary number) <table><tr><td>0</td><td>Fill up digits. Add prefix 0x or 0b, e.g. 0x0C or 0b00001100</td></tr><tr><td>1</td><td>Fill up digits. No prefix 0x or 0b, e.g. 0C or 00001100</td></tr><tr><td>2</td><td>Don't fill up digits. Add prefix 0x or 0b, e.g. 0xC or 0b1100</td></tr><tr><td>3</td><td>Don't fill up digits. No prefix 0x or 0b, e.g. C or 1100</td></tr></table>	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						
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

Similar to Syntax1 with decimal output, filling up digits and adding prefix.

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

? **byte** n = 123

```
value = GetString(n)           // value = "123"  
value = GetString(n, 10)       // value = "123"  
value = GetString(n, 16)       // value = "0x7B"  
value = GetString(n, 2)        // value = "0b01111011"  
value = GetString(n, 16, 3)    // value = "7B"  
value = GetString(n, 2, 2)     // value = "0b1111011"
```

? **byte[]** n = {12, 34, 56}

```
value = GetString(n)           // value = "{12,34,56}"  
value = GetString(n, 10)       // value = "{12,34,56}"  
value = GetString(n, 16)       // value = "{0x0C,0x22,0x38}"  
value = GetString(n, 2)        // value = "{0b00001100,0b00100010,0b00111000}"  
value = GetString(n, 16, 3)    // value = "{C,22,38}"  
value = GetString(n, 2, 2)     // value = "{0b1100,0b100010,0b111000}"
```

? **int** n = 1234

```
value = GetString(n)           // value = "1234"  
value = GetString(n, 10)       // value = "1234"  
value = GetString(n, 16)       // value = "0x000004D2"  
value = GetString(n, 2)        // value = "0b0000000000000000000010011010010"  
value = GetString(n, 16, 3)    // value = "4D2"  
value = GetString(n, 2, 2)     // value = "0b10011010010"
```

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

```
value = GetString(n)           // value = "{123,345,-123,-456}"
value = GetString(n, 10)       // value = "{123,345,-123,-456}"
value = GetString(n, 16)       // value = "{0x0000007B,0x00000159,0xFFFFF85,0xFFFFFE38}"
value = GetString(n, 2)        // value = "{0b00000000000000000000000001111011,
                                     0b000000000000000000000000101011001,
                                     0b11111111111111111111111110000101,
                                     0b1111111111111111111111111000111000}"

value = GetString(n, 16, 3)    // value = "{7B,159,FFFFFF85,FFFFFE38}"
value = GetString(n, 2, 2)     // value = "{0b1111011,
                                     0b101011001,
                                     0b11111111111111111111111110000101,
                                     0b1111111111111111111111111000111000}"
```

? **float** n = 12.34

```
value = GetString(n)           // value = "12.34"
value = GetString(n, 10)       // value = "12.34"
value = GetString(n, 16)       // value = "0x414570A4"
value = GetString(n, 2)        // value = "0b01000001010001010111000010100100"
value = GetString(n, 16, 3)    // value = "414570A4"
value = GetString(n, 2, 2)     // value = "0b1000001010001010111000010100100"
```

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

```
value = GetString(n)           // value = "{123.4,345.6,-123.4,-456.7}"
value = GetString(n, 10)       // value = "{123.4,345.6,-123.4,-456.7}"
value = GetString(n, 16)       // value = "{0x42F6CCCD,0x43ACCCCD,0xC2F6CCCD,0xC3E4599A}"
value = GetString(n, 16, 3)    // value = "{42F6CCCD,43ACCCCD,C2F6CCCD,C3E4599A}"
```

? **double** n = 12.34

```
value = GetString(n)           // value = "12.34"
value = GetString(n, 10)       // value = "12.34"
value = GetString(n, 16)       // value = "0x4028AE147AE147AE"
value = GetString(n, 16, 3)    // value = "4028AE147AE147AE"
```

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

```
value = GetString(n)           // value = "{123.45,345.67,-123.48,-456.79}"
value = GetString(n, 10)       // value = "{123.45,345.67,-123.48,-456.79}"
value = GetString(n, 16)       // value = "{0x405EDCCCCCCCCCD,0x40759AB851EB851F,
                                0xC05EDEB851EB851F,0xC07C8CA3D70A3D71}"
value = GetString(n, 16, 3)    // value = "{405EDCCCCCCCCCD,40759AB851EB851F,
                                C05EDEB851EB851F,C07C8CA3D70A3D71}"
```

? **bool** n = true

```
value = GetString(n)           // value = "true"
value = GetString(n, 16)       // value = "true"
value = GetString(n, 2)        // value = "true"
value = GetString(n, 16, 3)    // value = "true"
```

? **bool[]** n = {true, false, true, false, false, true}

```
value = GetString(n)           // value = "{true,false,true,false,false,true}"
value = GetString(n, 16)       // value = "{true,false,true,false,false,true}"
value = GetString(n, 2)        // value = "{true,false,true,false,false,true}"
value = GetString(n, 16, 3)    // value = "{true,false,true,false,false,true}"
```

? **string** n = "1234567890"

```
value = GetString(n)           // value = "1234567890"
value = GetString(n, 16)       // value = "1234567890"
value = GetString(n, 2)        // value = "1234567890"
value = GetString(n, 16, 3)    // value = "1234567890"
```

? **string[]** n = {"123.45", "345.67", "-12""3.48", "-45A6.79"}

```
value = GetString(n)           // value = "{123.45,345.67,-12""3.48,-45A6.79}" // -12""3.48 displayed as -12"3.48
value = GetString(n, 16)       // value = "{123.45,345.67,-12""3.48,-45A6.79}"
value = GetString(n, 2)        // value = "{123.45,345.67,-12""3.48,-45A6.79}"
value = GetString(n, 16, 3)    // value = "{123.45,345.67,-12""3.48,-45A6.79}"
```

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 |
- 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

Similar to Syntax4 with filling up digits and adding prefix 0x or 0b

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

Syntax 6

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

Note

Similar to Syntax4 with decimal output, filling up digits and adding prefix

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

? **byte** n = 123

```
value = GetString(n)           // value = "123"  
value = GetString(n, ",", 10)  // value = "123"  
value = GetString(n, "-", 16)   // value = "0x7B"  
value = GetString(n, "#", 2)    // value = "0b01111011"  
value = GetString(n, ",", 16, 3) // value = "7B"  
value = GetString(n, ",", 2, 2) // value = "0b1111011"
```

* Separator is only effective to array input

? **byte[]** n = {12, 34, 56}

```
value = GetString(n, "-")       // value = "12-34-56"  
value = GetString(n, Ctrl("\r\n"), 10) // value = "12\u00D0A34\u00D0A56"  
value = GetString(n, newline, 16) // value = "0x0C\u00D0A0x22\u00D0A0x38"  
value = GetString(n, NewLine, 2)  // value = "0b00001100\u00D0A0b00100010\u00D0A0b00111000"  
value = GetString(n, "-", 16, 3)  // value = "C-22-38"  
value = GetString(n, "-", 2, 2)   // value = "0b1100-0b100010-0b111000"
```

* **\u00D0A** is Newline control character, not string value.

2.32 GetToken()

Retrieves 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 occurrence
`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`, returns the input string
If `the number of occurrence<=0`, 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 n = "$abcd$1234$ABCD$"
```

```
value = GetToken(n, "", "", 0)           // value = "$abcd$1234$ABCD$"  
value = GetToken(n, "$", "$")           // value = "$abcd$"  
value = GetToken(n, "$", "$", 0)         // value = ""  
value = GetToken(n, "$", "$", 1)         // value = "$abcd$"  
value = GetToken(n, "$", "$", 2)         // value = "$ABCD$"  
value = GetToken(n, "$", "$", 3)         // value = ""  
value = GetToken(n, "$", "$", 1, 1)      // value = "abcd"  
value = GetToken(n, "$", "$", 2, 1)      // value = "ABCD"  
value = GetToken(n, "$", "", 1)          // value = "$abcd"  
value = GetToken(n, "$", "", 2)          // value = "$1234"  
value = GetToken(n, "$", "", 3)          // value = "$ABCD"  
value = GetToken(n, "$", "", 4)          // value = "$"  
value = GetToken(n, "", "$", 1)          // value = "$"  
value = GetToken(n, "", "$", 2)          // value = "abcd$"  
value = GetToken(n, "", "$", 3)          // value = "1234$"  
value = GetToken(n, "", "$", 4)          // value = "ABCD$"
```

```
string n = "$abcd$1234$ABCD$" + Ctrl("\r\n") + "56\r\n78$"
```

```
value = GetToken(n, "$", Ctrl("\r\n"), 1) // value = "$abcd$1234$ABCD$\u000A"  
value = GetToken(n, "$", newline, 2)      // value = ""  
value = GetToken(n, "$", NewLine, 1, 1)    // value = "abcd$1234$ABCD$" // Remove prefix and suffix  
value = GetToken(n, Ctrl("\r\n"), "$", 1)  // value = "\u000A56\r\n78$"  
value = GetToken(n, newline, "$", 2)       // value = ""  
value = GetToken(n, NewLine, "$", 1, 1)    // value = "56\r\n78"
```

* **\u000A** 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 occurrence
`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`, returns the input string
If `the number of occurrence<=0`, 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 n = "$abcd$1234$ABCD$"
```

```
byte[] bb0 = {}, bb1 = {0x24}    // 0x24 is $
```

```
value = GetToken(n, bb0, bb0, 0)    // value = "$abcd$1234$ABCD$"  
value = GetToken(n, bb1, bb1)      // value = "$abcd$"  
value = GetToken(n, bb1, bb1, 0)    // value = ""  
value = GetToken(n, bb1, bb1, 1)    // value = "$abcd$"  
value = GetToken(n, bb1, bb1, 2)    // value = "$ABCD$"  
value = GetToken(n, bb1, bb1, 3)    // value = ""  
value = GetToken(n, bb1, bb1, 1, 1) // value = "abcd"  
value = GetToken(n, bb1, bb1, 2, 1) // value = "ABCD"  
value = GetToken(n, bb1, bb0, 1)    // value = "$abcd"  
value = GetToken(n, bb1, bb0, 2)    // value = "$1234"  
value = GetToken(n, bb1, bb0, 3)    // value = "$ABCD"  
value = GetToken(n, bb1, bb0, 4)    // value = "$"  
value = GetToken(n, bb0, bb1, 1)    // value = "$"  
value = GetToken(n, bb0, bb1, 2)    // value = "abcd$"  
value = GetToken(n, bb0, bb1, 3)    // value = "1234$"  
value = GetToken(n, bb0, bb1, 4)    // value = "ABCD$"
```

```
string n = "$abcd$1234$ABCD$" + Ctrl("\r\n") + "56\r\n78$"
```

```
byte[] bb0 = {0x0D,0x0A}, bb1 = {0x24}    // 0x24 is $ // 0x0D,0x0A is \u000A
```

```
value = GetToken(n, bb1, bb0, 1)    // value = "$abcd$1234$ABCD$\u000A"  
value = GetToken(n, bb1, bb0, 2)    // value = ""  
value = GetToken(n, bb1, bb0, 1, 1) // value = "abcd$1234$ABCD$"    // 去除前置與後置  
value = GetToken(n, bb0, bb1, 1)    // value = "\u000A56\r\n78$"  
value = GetToken(n, bb0, bb1, 2)    // value = ""  
value = GetToken(n, bb0, bb1, 1, 1) // value = "56\r\n78"
```

* **\u000A** is Newline control character, not string value.

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 occurrence
`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 occurrence<=0](#), 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 s = "$abcd$1234$ABCD$"
```

```
byte[] n = GetBytes(s)
```

```
value = GetToken(n, "", "", 0)    // value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}  
value = GetToken(n, "$", "$")      // value = {0x24,0x61,0x62,0x63,0x64,0x24}  
value = GetToken(n, "$", "$", 0)    // value = {}  
value = GetToken(n, "$", "$", 1)    // value = {0x24,0x61,0x62,0x63,0x64,0x24}  
value = GetToken(n, "$", "$", 2)    // value = {0x24,0x41,0x42,0x43,0x44,0x24}  
value = GetToken(n, "$", "$", 1, 1) // value = {0x61,0x62,0x63,0x64}  
value = GetToken(n, "$", "$", 2, 1) // value = {0x41,0x42,0x43,0x44}  
value = GetToken(n, "$", "", 1)     // value = {0x24,0x61,0x62,0x63,0x64}  
value = GetToken(n, "$", "", 2)     // value = {0x24,0x31,0x32,0x33,0x34}  
value = GetToken(n, "$", "", 3)     // value = {0x24,0x41,0x42,0x43,0x44}  
value = GetToken(n, "$", "", 4)     // value = {0x24}  
value = GetToken(n, "", "$", 1)     // value = {0x24}  
value = GetToken(n, "", "$", 2)     // value = {0x61,0x62,0x63,0x64,0x24}  
value = GetToken(n, "", "$", 3)     // value = {0x31,0x32,0x33,0x34,0x24}  
value = GetToken(n, "", "$", 4)     // value = {0x41,0x42,0x43,0x44,0x24}
```

```
string s = "$abcd$1234$ABCD$" + Ctrl("\r\n") + "56\r\n78$"
```

```
byte[] n = GetBytes(s)
```

```
value = GetToken(n, "$", Ctrl("\r\n"), 1)  
    // value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24,0x0D,0x0A}  
value = GetToken(n, "$", Ctrl("\r\n"), 1, 1)  
    // value = {0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}    // Removing prefix and  
    suffix  
value = GetToken(n, Ctrl("\r\n"), "$", 1)  
    // value = {0x0D,0x0A,0x35,0x36,0x5C,0x72,0x5C,0x6E,0x37,0x38,0x24}  
value = GetToken(n, Ctrl("\r\n"), "$", 1, 1)
```

```
// 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 occurrence

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 occurrence<=0**, 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 s = "$abcd$1234$ABCD$"
```

```
byte[] n = GetBytes(s)
```

```
byte[] bb0 = {}, bb1 = {0x24}    // 0x24 is $
```

```
value = GetToken(n, bb0, bb0, 0) // value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}
```

```
value = GetToken(n, bb1, bb1)    // value = {0x24,0x61,0x62,0x63,0x64,0x24}
```

```
value = GetToken(n, bb1, bb1, 0) // value = {}
```

```
value = GetToken(n, bb1, bb1, 1) // value = {0x24,0x61,0x62,0x63,0x64,0x24}
```

```
value = GetToken(n, bb1, bb1, 2) // value = {0x24,0x41,0x42,0x43,0x44,0x24}
```

```
value = GetToken(n, bb1, bb1, 1, 1) // value = {0x61,0x62,0x63,0x64}
```

```
value = GetToken(n, bb1, bb1, 2, 1) // value = {0x41,0x42,0x43,0x44}
```

```
value = GetToken(n, bb1, bb0, 1)    // value = {0x24,0x61,0x62,0x63,0x64}
```

```
value = GetToken(n, bb1, bb0, 2)    // value = {0x24,0x31,0x32,0x33,0x34}
```

```
value = GetToken(n, bb1, bb0, 3)    // value = {0x24,0x41,0x42,0x43,0x44}
```

```
value = GetToken(n, bb0, bb1, 1)    // value = {0x24}
```

```
value = GetToken(n, bb0, bb1, 2)    // value = {0x61,0x62,0x63,0x64,0x24}
```

```
value = GetToken(n, bb0, bb1, 3)    // value = {0x31,0x32,0x33,0x34,0x24}
```

```
string s = "$abcd$1234$ABCD$" + Ctrl("\r\n") + "56\r\n78$"
```

```
byte[] n = GetBytes(s)
```

```
byte[] bb0 = {0x0D,0x0A}, bb1 = {0x24}
```

```
value = GetToken(n, bb1, bb0, 1)
```

```
    // value = {0x24,0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24,0x0D,0x0A}
```


```
value = GetToken(n, bb1, bb0, 1, 1)
```

```
    // value = {0x61,0x62,0x63,0x64,0x24,0x31,0x32,0x33,0x34,0x24,0x41,0x42,0x43,0x44,0x24}    // Remove prefix and
```

```
suffix
```

```
value = GetToken(n, bb0, bb1, 1)
```

```
    // value = {0x0D,0x0A,0x35,0x36,0x5C,0x72,0x5C,0x6E,0x37,0x38,0x24}
```



```
value = GetToken(n, bb0, bb1, 1, 1)
```

```
// value = {0x35,0x36,0x5C,0x72,0x5C,0x6E,0x37,0x38}
```

2.33 GetAllTokens()

Retrieves 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
----------	---

Year	Number of people (millions)
1990	60
1995	70
2000	68
2005	75
2010	85

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

Note

Similar to Syntax1 with reserving prefix and suffix

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

```
string n = "$abcd$1234$ABCD$"

value = GetAllTokens(n, "", "") // value = {"$abcd$1234$ABCD$"}
value = GetAllTokens(n, "$", "$") // value = {"$abcd$", "$ABCD$"}
value = GetAllTokens(n, "$", "$", 1) // value = {"abcd", "ABCD"}
value = GetAllTokens(n, "$", "") // value = {"$abcd", "$1234", "$ABCD", "$"}
value = GetAllTokens(n, "", "$", 1) // value = {"", "abcd", "1234", "ABCD"}
```

2.34 GetNow()

Capture 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 remain the same.

d	One-digit day of the month for days below 10, from 1 to 31
dd	Two-digit day of the month, from 01 to 31
ddd	Three-letter abbreviation for day of the week, e.g. Mon
dddd	Day of the week spelled out in full, e.g. Monday
f	Second to 0.1 second
ff	Second to 0.01 Second
fff	Second to 0.001 Second
ffff	Second to 0.0001 Second
h	One-digit hour for hours below 10 in 12-hour format, from 1 to 12
hh	Two-digit hour in 12-hour format, from 01 to 12
H	One-digit hour for hours below 10 in 12-hour format, from 0 to 23
HH	Two-digit hour in 24-hour format, from 00 to 23
m	One-digit minute for minutes below 10, from 0 to 59
mm	Two-digit minute, from 00 to 59
M	One-digit month for months below 10, from 1 to 12
MM	Two-digit month, from 01 to 12
MMM	Three-letter abbreviation for month, Jun
MMMM	Month spelled out in full, June
s	One-digit second for seconds below 10, from 0 to 59
ss	Two-digit second, from 00 to 59
t	The first digit of AM/PM
tt	AM/PM
y	One-digit year for years below 10, from 0 to 99
yy	Two-digit year, from 00 to 99
yyyy	four-digit year
/	Separator for date. / or – or . (Based on different languages)

Return

string Current date and time. If there is an error in format setting, the default format will be applied.

Note

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

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

2.35 GetNowStamp()

Return the total time or time difference of system time

Syntax 1

```
int GetNowStamp(  
)
```

Parameters

`void` No parameter

Return

`int` The total time of system in ms.
The total time = (hours)*60*60*1000 + (minutes)*60*1000 + (seconds)*1000
*only returns the time today

Note

```
value = GetNowStamp()      // value = 41730494
```

Syntax 2

```
int GetNowStamp(  
    int  
)
```

Parameters

`int` Previous recorded total time in ms

Return

`int` The difference between the current total time and the input total time in ms.
Time difference = current total time – input total time
*Only returns in int

Note

```
value = GetNowStamp()      // value = 41730494  
... others ...  
diff = GetNowStamp(value)      // Assume GetNowStamp()=41730662 in the same day      // diff = 168  
  
value = GetNowStamp()      // value = 86399990  
... others ...  
diff = GetNowStamp(value)      // Assume GetNowStamp()=10 a day after      // diff = 20
```

2.36 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 data type could be integer, floating number, 輸入的原始值, booling, string or array

Return

int Length of data
For input as integer, floating number and booling, 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 n = 100  
value = Length(n)      // value = 1  
value = Length(100)    // value = 1  
  
? int n = 400  
value = Length(n)      // value = 4  
value = Length(400)    // value = 4  
  
? float n = 1.234  
value = Length(n)      // value = 4  
value = Length(1.234)    // value = 4  
  
? double n = 1.234  
value = Length(n)      // value = 8  
value = Length(1.234)    // value = 4      // float // Numbers would be stored as the smaller data type first.  
  
? bool n = true  
value = Length(n)      // value = 1  
value = Length(false)    // value = 1
```

? **string** n = "A"BC"

value = **Length**(n) // value = 4 // The string is A"BC. Two double quotation marks represent " in string

value = **Length**("") // value = 0

value = **Length**("123") // value = 3

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

? **byte[]** n = {100, 200, 30}

value = **Length**(n) // value = 3

? **int[]** n = {}

value = **Length**(n) // value = 0

n = {400, 500, 600}

value = **Length**(n) // value = 3

? **float[]** n = {1.234}

value = **Length**(n) // value = 1

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

value = **Length**(n) // value = 4

? **bool[]** n = {true, false, true, true, true, true, false}

value = **Length**(n) // value = 7

? **string[]** n = {"A"BC", "123", "456", "ABC"}

value = **Length**(n) // value = 4

2.37 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

```
b = 0x0D0A  
value = Ctrl(b)           // value = \r\n  
value = Ctrl(0x0D0A)      // value = \r\n  
value = Ctrl(0x0D000A09)  // value = \r\n\t      // 0x00 will not be transformed  
value = Ctrl(0x0D300A09)  // value = \r0\n\t      // 0x30 is transformed to 0  
value = Ctrl(0x00)        // 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.

<code>\0</code>	0x00 null
<code>\a</code>	0x07 bell
<code>\b</code>	0x08 backspace
<code>\t</code>	0x09 horizontal tab
<code>\r</code>	0x0D carriage return
<code>\v</code>	0x0B vertical tab
<code>\f</code>	0x0C form feed
<code>\n</code>	0x0A line feed

Return

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

Note

```
b = "\r\n"  
value = Ctrl(b)           // value = \r\n  
value = Ctrl("\r\n")      // value = \r\n  
value = Ctrl("\r\n\t")    // value = \r\n\t  
value = Ctrl("\r0\n\t")   // value = \r0\n\t  
value = Ctrl("\0")        // value = \0    // NULL
```

2.38 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = XOR8(bb1,0,Length(bb1))    // value = 0x6F
```

```
value = XOR8(bb1,0,-1)              // value = 0x6F
```

```
value = XOR8(bb1,1,-1)              // value = 0x7F
```

```
value = XOR8(bb1,-1,-1)             // 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = XOR8(bb1,0,Length(bb1))           // value = 0x6F
```

```
value = XOR8(bb1,0)                       // value = 0x6F
```

```
value = XOR8(bb1)                         // value = 0x6F
```

```
bb1 = Byte_Concat(bb1, XOR(bb1))          // bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x6F}
```

2.39 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 strating index and number of element exceeds the array size, the calculation ends at the last element of the array.

Return

`byte` Checksum.

Note

```
byte[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = SUM8(bb1,0,Length(bb1))    // value = 0x6D
```

```
value = SUM8(bb1,0,-1)              // value = 0x6D
```

```
value = SUM8(bb1,1,-1)              // value = 0x5D
```

```
value = SUM8(bb1,-1,-1)             // 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = SUM8(bb1,0,Length(bb1))           // value = 0x6D
```

```
value = SUM8(bb1,0)                       // value = 0x6D
```

```
value = SUM8(bb1)                         // value = 0x6D
```

```
bb1 = Byte_Concat(bb1, SUM8(bb1))        // bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x6D}
```

2.40 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 strating 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = SUM16(bb1,0,Length(bb1)) // value = {0x04, 0x6D}
```

```
value = SUM16(bb1,0,-1)           // value = {0x04, 0x6D}
```

```
value = SUM16(bb1,1,-1)           // value = {0x04, 0x5D}
```

```
value = SUM16(bb1,-1,-1)          // 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = SUM16(bb1,0,Length(bb1))           // value = {0x04, 0x6D}
```

```
value = SUM16(bb1,0)                         // value = {0x04, 0x6D}
```

```
value = SUM16(bb1)                           // value = {0x04, 0x6D}
```

```
bb1 = Byte_Concat(bb1, SUM16(bb1))          // bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x04, 0x6D}
```

2.41 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 strating 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}`

`value = SUM32(bb1,0,Length(bb1))` // value = {0x00, 0x00, 0x04, 0x6D}

`value = SUM32(bb1,0,-1)` // value = {0x00, 0x00, 0x04, 0x6D}

`value = SUM32(bb1,1,-1)` // value = {0x00, 0x00, 0x04, 0x5D}

`value = SUM32(bb1,-1,-1)` // 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = SUM32(bb1,0,Length(bb1))           // value = {0x00, 0x00, 0x04, 0x6D}
```

```
value = SUM32(bb1,0)                       // value = {0x00, 0x00, 0x04, 0x6D}
```

```
value = SUM32(bb1)                         // value = {0x00, 0x00, 0x04, 0x6D}
```

```
bb1 = Byte_Concat(bb1, SUM32(bb1))        // bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x04, 0x6D}
```

2.42 CRC16()

Utilize CRC 16 bits algorithm to computes the checksum

Syntax 1

```
byte[] CRC16 (  
    int,  
    byte[],  
    int,  
    int  
)
```

Parameters

<code>int</code>	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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

value = CRC16(0, bb1, 0, Length(bb1))    // value = {0x2D, 0xD4}
value = CRC16(0, bb1, 0, -1)              // value = {0x2D, 0xD4}
value = CRC16(0, bb1, 1, -1)              // value = {0xEC, 0xC5}
value = CRC16(0, bb1, -1, -1)             // value = {0x00, 0x00}
value = CRC16(3, bb1, 0, Length(bb1))    // value = {0x42, 0x12}
value = CRC16(4, bb1, 0, Length(bb1))    // 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}

value = CRC16(0, bb1, 0, Length(bb1))    // value = {0x2D, 0xD4}
value = CRC16(0, bb1, 0)                  // value = {0x2D, 0xD4}
value = CRC16(0, bb1)                     // value = {0x2D, 0xD4}
bb1 = Byte_Concat(bb1, CRC16(0, bb1))    // 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[]))

2.43 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = CRC32(bb1,0,Length(bb1))    // value = {0x43, 0xD5, 0xB9, 0xF8}
```

```
value = CRC32(bb1,0,-1)              // value = {0x43, 0xD5, 0xB9, 0xF8}
```

```
value = CRC32(bb1,1,-1)              // value = {0x08, 0xA5, 0x5B, 0xEB}
```

```
value = CRC32(bb1,-1,-1)             // 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[] bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF}
```

```
value = CRC32(bb1,0,Length(bb1)) // value = {0x43, 0xD5, 0xB9, 0xF8}
```

```
value = CRC32(bb1,0) // value = {0x43, 0xD5, 0xB9, 0xF8}
```

```
value = CRC32(bb1) // value = {0x43, 0xD5, 0xB9, 0xF8}
```

```
bb1 = Byte_Concat(bb1, CRC32(bb1)) // bb1 = {0x10, 0x20, 0x50, 0xF0, 0xFF, 0xFF, 0xFF, 0x43, 0xD5, 0xB9, 0xF8}
```

3. Modbus Functions

3.1 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)

```
value = modbus_read("TCP_1", "preset_800") // value = 1           // DO 1 address = 1 bit
value = modbus_read("TCP_1", "preset_7202") // value = true       // DI 1 address = 1 bit
value = modbus_read("TCP_1", "preset_9000") // value = ab1234cd    // RO 4 address = 8 bytes size
value = modbus_read("TCP_1", "preset_7001") // 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

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

```
value = modbus_read("TCP_1", 0, "DO", 800, 4)
    // value = {0,0,0,0}      // DO 4 address = 4 bit to byte array
value = modbus_read("TCP_1", 0, "DI", 7202, 3)
    // value = {1,0,0}      // DI 3 address = 3 bit to byte array
value = modbus_read("TCP_1", 0, "RO", 9000, 6)
    // value = {0x54,0x65,0x63,0x68,0x6D,0x61,0x6E,0xE9,0x81,0x94,0xE6,0x98} // RO 6 address = 12 bytes size
value = modbus_read("TCP_1", 0, "RI", 7001, 12)
    // 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}      // RI 12 address = 24 bytes size
value = modbus_read("TCP_1", 0, "RI", 7301, 6)
    // value = {0x07,0xE2,0x00,0x05,0x00,0x12,0x00,0x0F,0x00,0x0A,0x00,0x39} // RI 6 address = 12 bytes size
```

3.2 *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

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

value = **modbus_read_int16**("TCP_1", 0, "DO", 800, 4)

// byte[] = {0,0,0,0} to int16[] value = {0,0} // byte[0][1] , byte[2][3]

value = **modbus_read_int16**("TCP_1", 0, "DI", 7202, 3)

// byte[] = {1,0,0} to int16[] value = {256,0} // byte[0][1] , byte[2][3] // Fill up to [3] automatically

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[] value = {21605,25448,28001,28393,-32364,-6504}

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[] value = {10544,-24756,-15492,-26214,17502,-4915,17076,0,-32768,0,0,0}

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[] value = {2018,5,18,15,49,35}

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[] value = {2018,5,18,15,49,35}

3.3 *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

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

value = **modbus_read_int32**("TCP_1", 0, "DO", 800, 4)

// byte[] = {0,0,0,0} to int32[] value = {0} // byte[0][1][2][3]

value = **modbus_read_int32**("TCP_1", 0, "DI", 7202, 3)

// byte[] = {1,0,0} to int32[] value = {16777216} // byte[0][1][2][3] // Fill up to [3] automatically.

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[] value = {1415930728,1835101929,-2120948072}

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,
0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// to int32[] value = {691052364,-1015244390,1147071693,1119092736,-2147483648,0}

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[] value = {132251653,1179663,3211299}

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}

3.4 *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

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

value = **modbus_read_float**("TCP_1", 0, "DO", 800, 4)

// byte[] = {0,0,0,0} to float[] value = {0} // byte[0][1][2][3]

value = **modbus_read_float**("TCP_1", 0, "DI", 7202, 3)

// byte[] = {1,0,0} to float[] value = {2.350989E-38} // byte[0][1][2][3] // Fill up to [3] automatically.

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[] value = {3.940861E+12,4.360513E+27,-5.46975E-38}

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[] value = {3.921802E-14,-252.6,891.7,90,0,0}

value = **modbus_read_float**("TCP_1", 0, "RI", 7001, 12, **0**) // byte[2][3][0][1]

// to float[] value = {0x9F4C2930,0x999AC37C,0xECCD445E,0x000042B4,0x00008000,0x00000000}

= {-4.323275E-20,-1.600218E-23,-1.985221E+27,2.392857E-41,4.591775E-41,0}

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[] value = {3.400471E-34,1.65306E-39,5.326512E-39}

3.5 *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

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

value = **modbus_read_double**("TCP_1", 0, "DO", 800, 4)

// byte[] = {0,0,0,0} to double[] value = {0} // byte[0][1][2][3][4][5][6][7]

value = **modbus_read_double**("TCP_1", 0, "DI", 7202, 3)

// byte[] = {1,0,0} to double[] value = {7.2911220195564E-304} // byte[0][1][2][3][4][5][6][7]

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[] value = {3.65481260356117E+98,-4.87647898854073E-301}

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,
0x80,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00}

// to double[] value = {2.76472410615396E-110,2.2818627604613E+21,0}

value = **modbus_read_double**("TCP_1", 0, "RI", 7001, 12, 0) // byte[6][7][4][5][2][3][0][1]

// to double[] value = {0x999AC37C9F4C2930,0x000042B4ECCD445E,0x0000000000008000}

= {-2.4604103205376E-185,3.62371629877526E-310,1.6189543082926E-319}

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[] value = {1.06475148078395E-270,1.52982527955113E-308}

3.6 *modbus_read_string()*

Modbus TCP/RTU read function, and transform modbus address data array to string text by 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

TCP device	0	RO	9000	12
------------	---	----	------	----

`modbus_write("TCP_1", 0, "RO", 9000) = "1234 達明機器手臂"`

// Undefined number of address to write, the default value 0 means write the whole data

// Write byte[] = {0x31,0x32,0x33,0x34,0xE9,0x81,0x94,0xE6,0x98,0x8E,
0xE6,0x9C,0xBA,0xE5,0x99,0xA8,0xE6,0x89,0x8B,0xE8,0x87,0x82}

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

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

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 達明機器手臂 AB // UTF8 format transformation

// The ending, 0x00, will not be included when writing data. When reading 12 addresses, the data out of the range set in first step will be captured

`modbus_write("TCP_1", 0, "RO", 9000) = "1234"+Ctrl("\0")`

// Write byte[] = {0x31,0x32,0x33,0x34,0x00} // Needs to write 3 Register address

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 original

data at those addresses // to string = 1234 // UTF8 format transformation stops at 0x00

3.7 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	
		int32	int	
Register Output	16	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

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 flag = true
```

```
modbus_write("TCP_1", "preset_800", flag)  // write 1 (true)
```

```
modbus_write("TCP_1", "preset_800", false) // write 0 (false)
```

```
string ss = "ABCDEFGHJKLMNOPQRST"
```

```
// With no address length number, default length = 4 will be applied, so 4 RO = 8 bytes size will be wrote.
```

```
modbus_write("TCP_1", "preset_9000", ss)    // write ABCDEFGH    // Over length part will not be  
wrote in.
```

```
// With no address length number, default length = 4 will be applied, so 4 RO = 8 bytes size data will be wrote.
```

```
modbus_write("TCP_1", "preset_9000", "1234567") // write 1234567\0 // Filled to 4 addresses with 0x00.
```

```
// With address length number = 0, write the whole data. "09AB123" needs 4 addresses.
```

```
modbus_write("TCP_1", "preset_9000", "09AB123", 0) // write 09AB123\0 // Filled to 4 address with 0x00.
```

```
// With address length number = 5, write the data with assigned length, so 5 RO = 10 bytes size data will be wrote.
```

modbus_write("TCP_1", "preset_9000", "09AB1234", 5) // write 09AB1234 // But only 4 addresses are needed.

Syntax 3 (TCP/RTU)

```
void 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

<code>bool</code>	<code>True</code>	Write success	
	<code>False</code>	Write failed	1. If the input data ? is empty string or array 2. If an error occurred in Modbus communication

Syntax 4 (TCP/RTU)

```
void modbus_write(  
    string,  
    byte,  
    string,  
    int,  
    ?  
)
```

Note

Similar to Syntax3 with address length as 0, it will write all the data.

`modbus_write("TCP_1", 0, "RO", 9000, bb) => modbus_write("TCP_1", 0, "RO", 9000, bb, 0)`

Modbus Address data size

Digital	1 address = 1 bit size
Register	1 address = 2 bytes size

假設 User Setting 使用自定義

TCP device	0	DO	800	4
TCP device	0	RO	9000	12

`byte[] bb = {10, 20, 30}`

`modbus_write("TCP_1", 0, "DO", 800, bb)` // write 1,1,1 // Zero value write as 0; Non-zero value write as 1

`modbus_write("TCP_1", 0, "DO", 800, bb, 2)` // write 1,1 // With address length = 2, write only 2 addresses

`modbus_write("TCP_1", 0, "DO", 800, true)` // write 1

`int i = 10000`

`modbus_write("TCP_1", 0, "RO", 9000, i)` // write 0x00,0x00,0x27,0x10 // default: int32 BigEndian (AB CD)

`bb = GetBytes(i, 0, 1)` // bb = {0x10,0x27} // write with int16 LittleEndian (CD AB)

modbus_write("TCP_1", 0, "RO", 9000, bb) // write 0x10,0x27

string[] n = {"ABC", "12", "34"}

modbus_write("TCP_1", 0, "RO", 9000, n, 2) // write ABC1 // Only 2 addressses data will be wrote.

modbus_write("TCP_1", 0, "RO", 9000, n, 5) // write ABC1234\0 // The data needs 4 addresses.

TECHMAN ROBOT



www.tm-robot.com