18. 宏指令说明

本章介绍宏指令的语法、编辑、及使用方法。

18.1. 概要

宏指令提供了应用程序之外所需的附加功能。在触摸屏触摸屏界面运行时，宏指令可以自动的执行这些命令。它可以担负执行譬如复杂的运算、字符串处理，和用户与工程之间的交流等功能。本章主要介绍宏指令的语法、如何使用和编辑方法等功能。希望通过本章的说明，能够使各位能够快速的掌握 EasyBuilder Pro 软件提供的强大的宏指令功能。

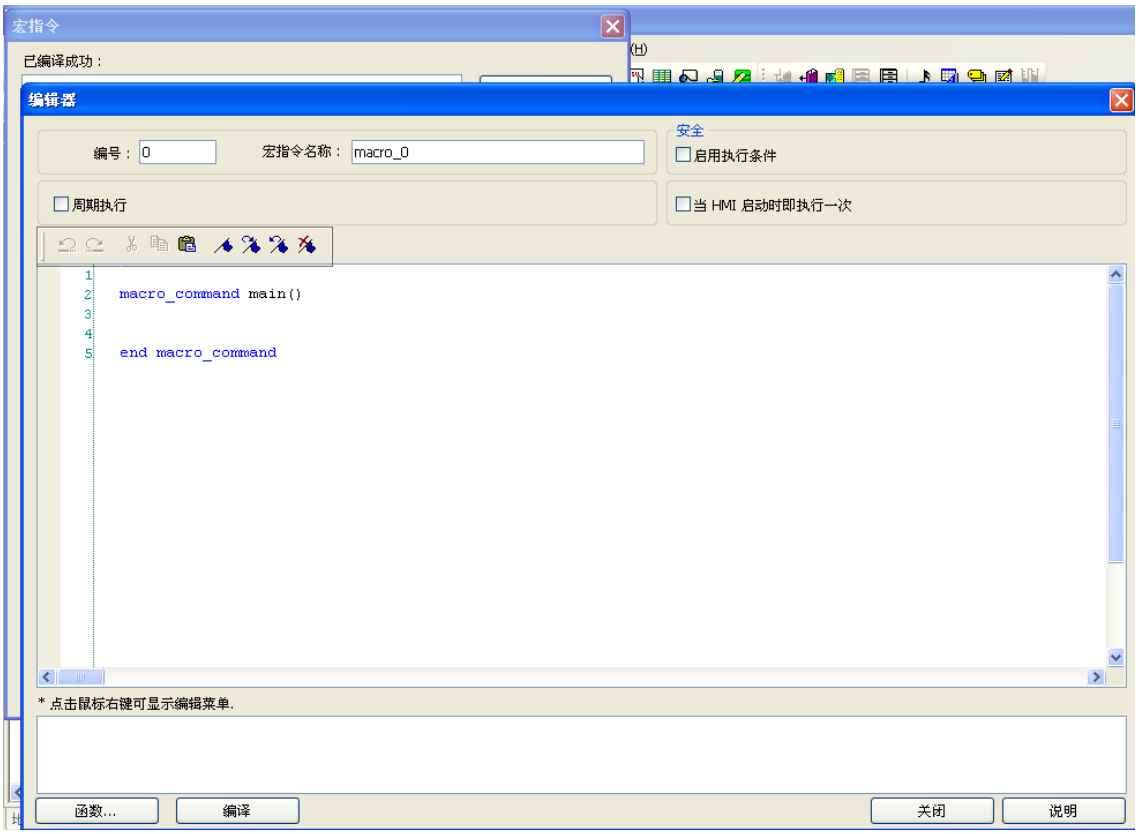
18.2. 宏指令编辑器功能使用说明

宏指令编辑器提供下列新功能：

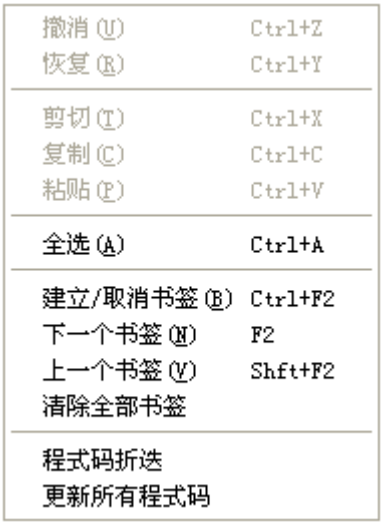
- 显示行号
- 复原 (Undo) / 重复 (Redo)
- 剪下 (Cut) / 复制 (Copy) / 贴上 (Paste)
- 全选 (Select All)
- 建立 / 取消书签 (Toggle Bookmark) / 上一个书签 (Previous Bookmark) / 下一个书签 (Next Bookmark) / 清除全部书签 (Clear All Bookmarks)
- 程序代码折迭 (Toggle All Outlining)
- 安全 -> 启用执行条件
- 周期执行
- 当触摸屏启动时即执行一次

以下将详细描述如何使用各项功能。

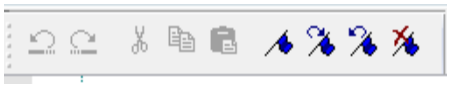
1. 打开宏指令编辑器，可以看到编辑区左边将自动显示行号。



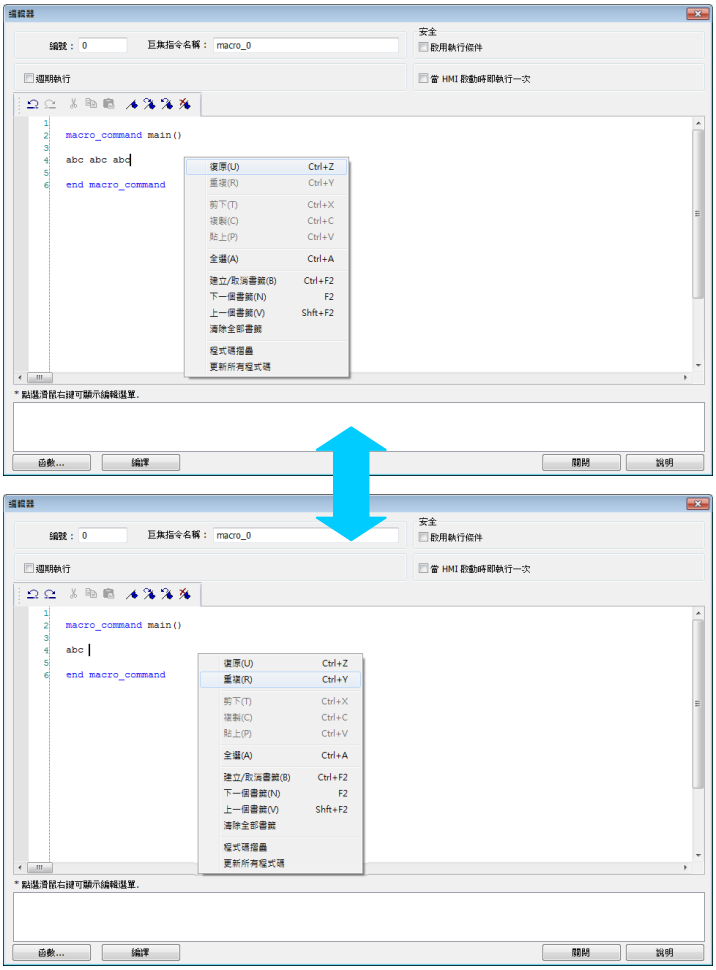
2. 编辑区中按鼠标右键，呼叫出右键选单如下图。目前的状态无法使用的功能将显示灰色。例如必须在编辑区中选取一段文字才会开启复制功能，因此未选取任何文字的状态下，复制功能暂不开启。提供快速键，如选单内所提示。



3. 编辑区上方有工具列，提供“复原”、”重复”、”剪下”、”复制”、”贴上”、”建立/取消书签”、”下一个书签”、”上一个书签”、”清除全部书签”等按钮方便快速选取。

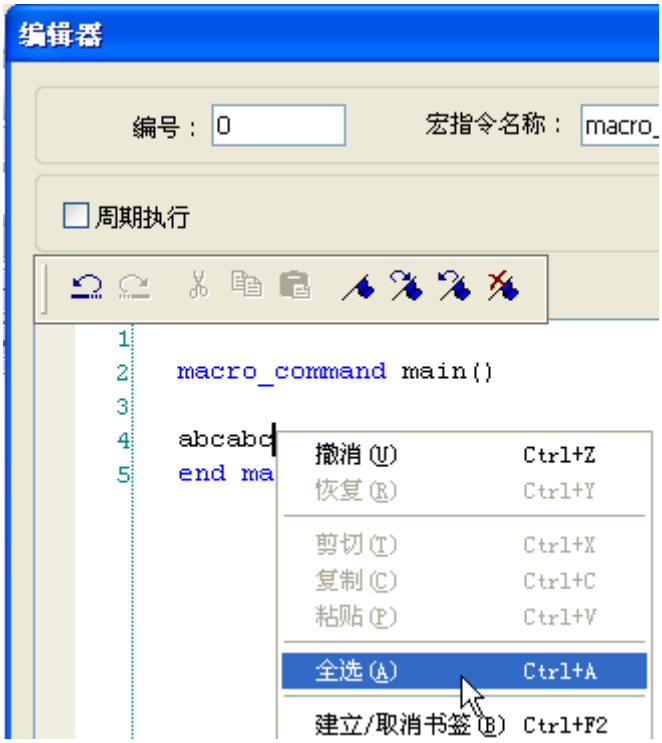


4. 改变编辑区内容将开启“复原”功能，用户执行复原后可用“重复”复原。用户可从右键选单或是利用热键 (Undo：Ctrl+Z，Redo：Ctrl+Y) 执行此功能。



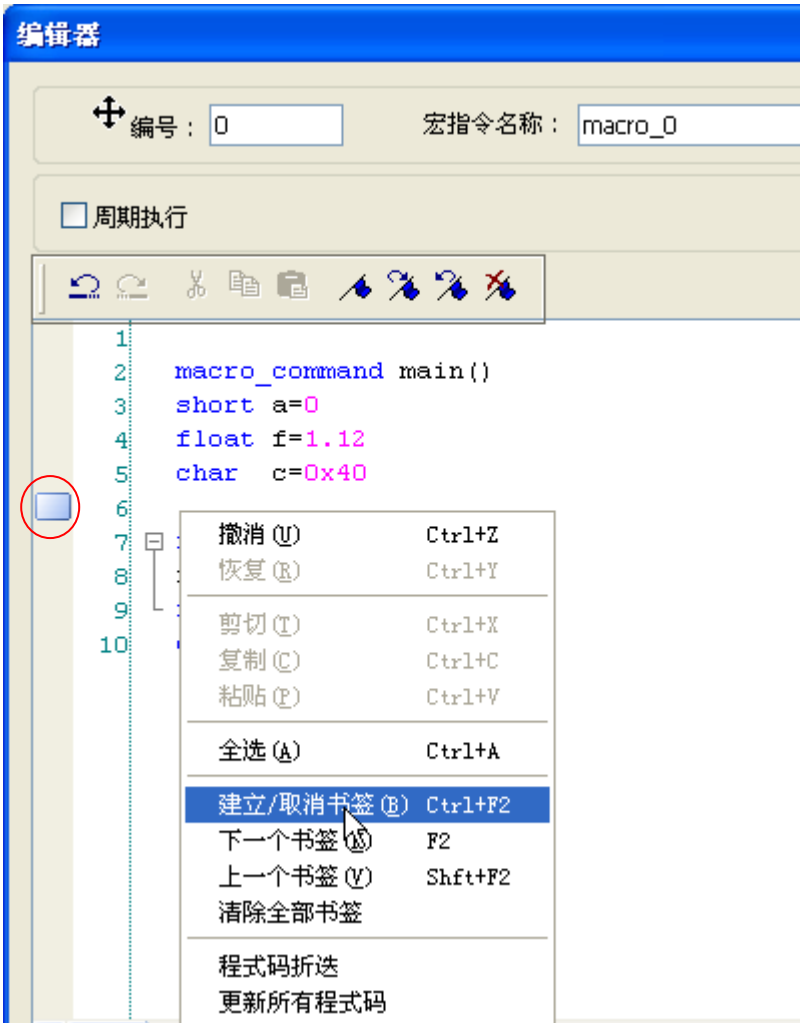
5. 在编辑区选取一段文字后可进行“剪下”、”复制”，之后可用“贴上”将选取的文字贴上。

6. 选择“全选”可选取编辑区全部内容。

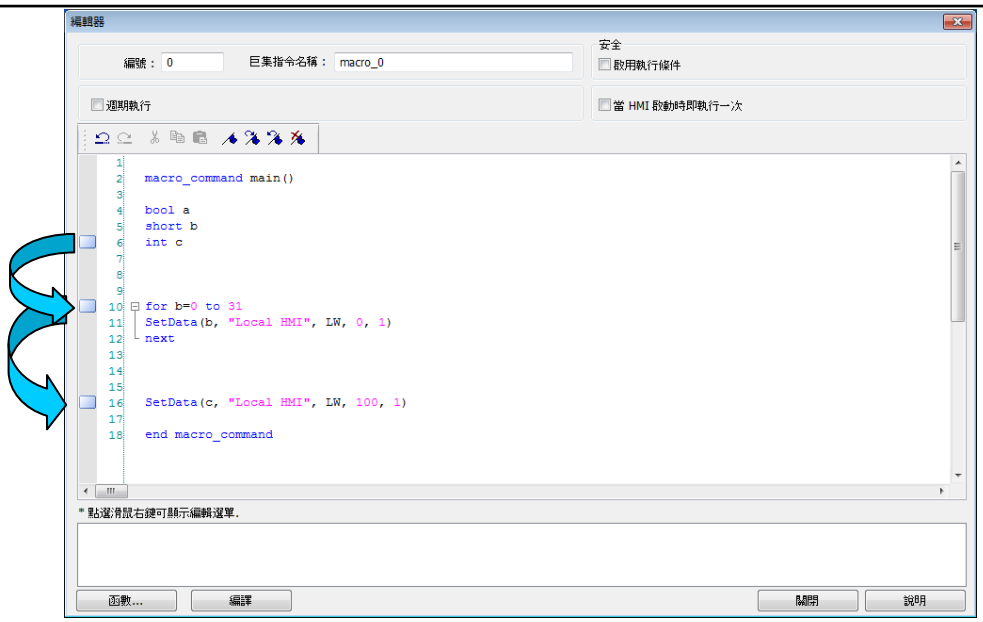


7. 当程序代码很长的时候，为方便用户阅读，提供了书签功能。下面说明如何使用此功能。

- 将光标移至编辑区中想要插入书签的位置，按右键，选择“建立 / 取消书签”。编辑区左边将会看到一个代表书签的蓝色小方块。

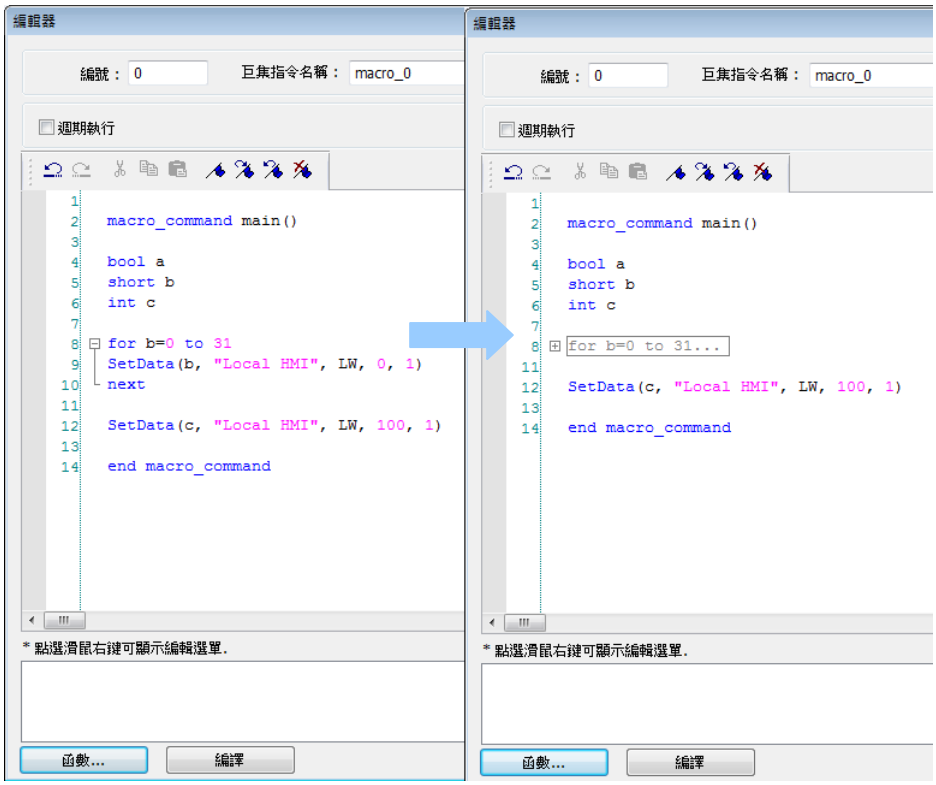


- 若光标所在位置已存在书签，选择“建立 / 取消书签”可将其关闭，反之，选择“建立 / 取消书签”可将其开启。
- 右键选择“下一个书签”光标将会移至下一个书签所在位置。选择“上一个书签”光标将会移至上一个书签所在位置。

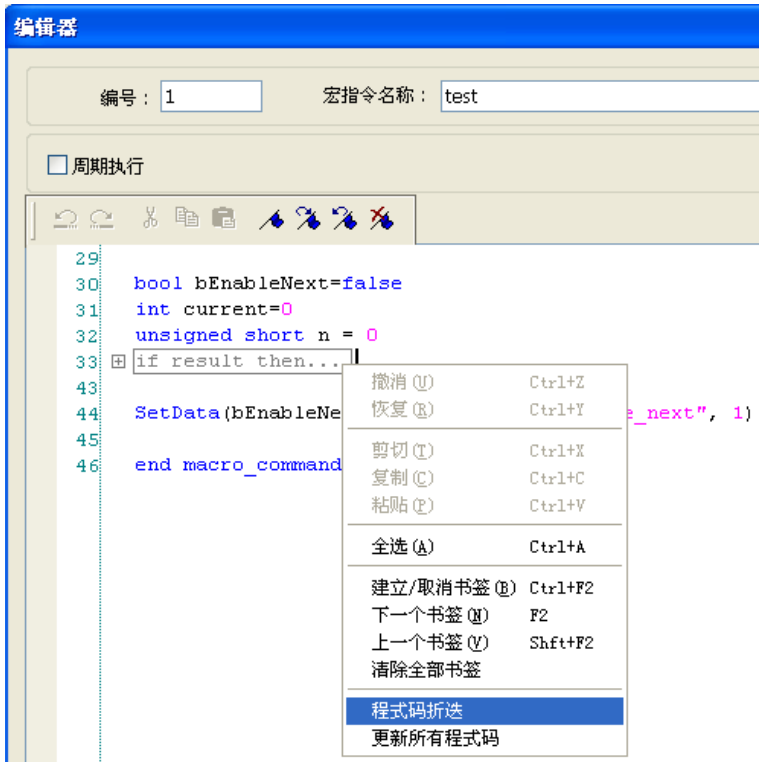


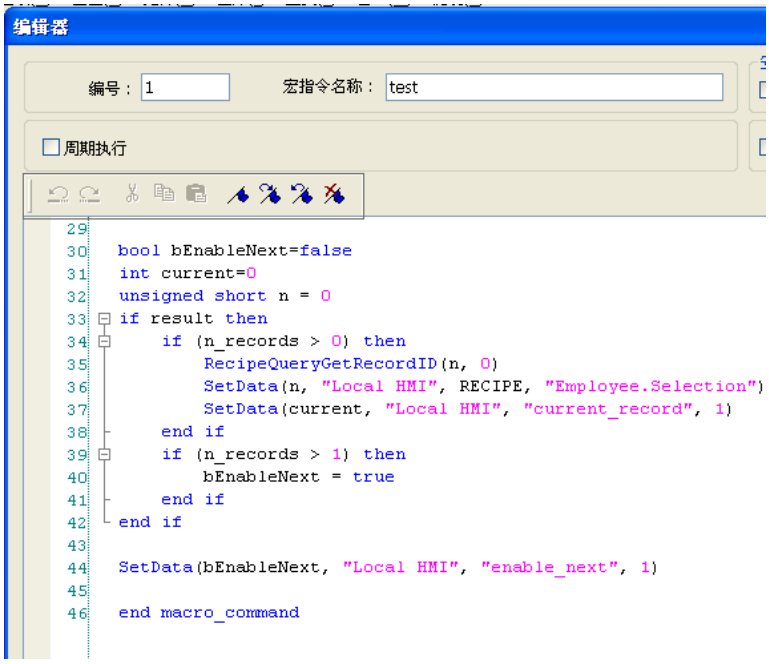
- 选择“清除全部书签”将关闭所有书签。

8. 宏指令编辑器提供程序代码折迭功能，方便用户浏览程序代码。所谓程序代码折迭，是指编辑器可将属于同一区块的程序代码隐藏起来，被隐藏起来的程序代码在编辑区里会显示成。编辑 ⊞ ... 区左侧会显示树形图，用户可按下⊟隐藏程序区块，按下⊞展开程序区块。如下图所示：



9. 右键选择“程序代码折迭”可展开所有程序代码区块。



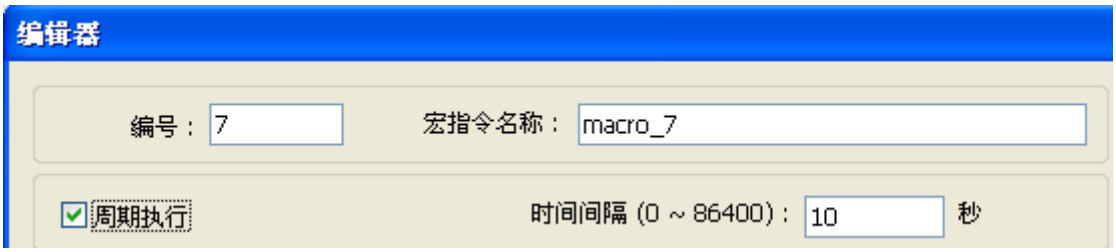


10. 有时候程序代码区块可能会误判。这种误判起因于编辑器没有办法区分当前输入的关键词是否存在于注释中。例如下图。用户可以从右键选单选择“更新所有程序代码”来更正这个错误。

11. 包围在特定关键词内的程序代码称为一程序代码区块。内定的程序代码区块如下列：

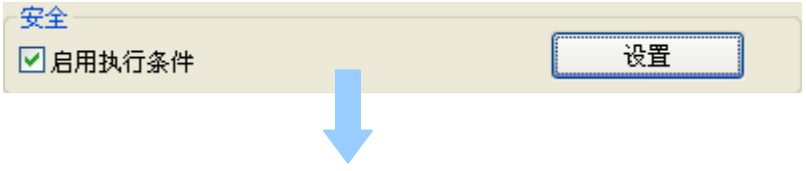
- 子函数： sub – end sub
- 循环语句：
  - i. for – next
  - ii. while – wend
- 逻辑运算语句：
  - i. if – end if
- 多重判断语句： select case – end select

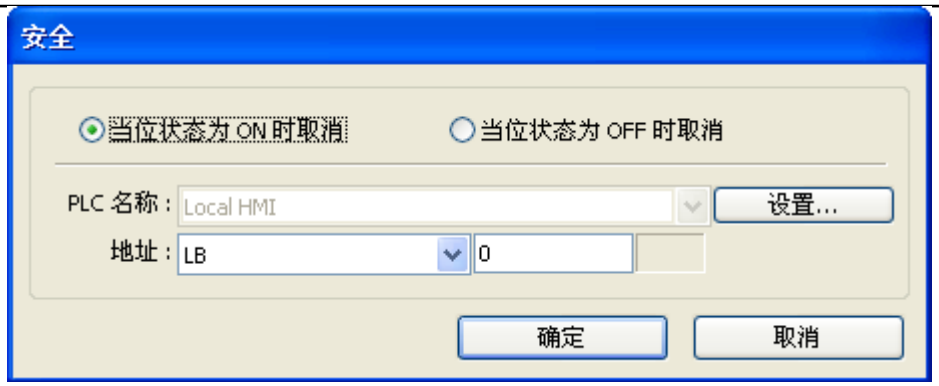
12. 用户勾选“周期执行”时，会周期性的触发此宏。



13. 用户勾选“安全”–> “启用执行条件”–> “设置”后,可以进行安全设定：

- 当位状态 ON 时取消:当位状态 ON 时禁止执行此宏。
- 当位状态 OFF 时取消: 当位状态 OFF 时禁止执行此宏。





14. 用户勾选“当 HMI 启动时即执行一次”时，在触摸屏启动时会自动执行宏一次。

18.3. 宏指令的结构

宏指令是由各种语句组成的。这些语句包含常数、变量和各种运算符号。这些语句放置在特定的顺序位置以便执行后达到一个希望的执行结果。

宏指令的结构一般为以下格式：

```
全局变量声明                          ----------可选

Sub Function Block Declarations(子函数声明) ----------可选
    局部变数声明
End Sub(结束子函数)

macro_command main()    [主函数]          ----------必须
    局部变数声明
    [各式语句]
end macro_command       [结束主函数]      ----------必须
```

一个宏指令必须有一个且只有一个主函数，用来开始宏指令的执行。格式为：

macro_command 函数名称()

end macro_command

变量声明必须放在宏指令语句的前面，否则如果语句放置在变量声明的前面，将会造成宏指令无法编译通过。
局部变量一般用在宏指令主函数或者自定义的子函数中。它的合法性只在指定的函数中有效。
全局变量一般是定义在所有宏指令函数的前面，且它在整个宏指令中均具有有效性。当局部变量和全局变量被定义为相同的名称时，只有局部变量有效。

下面就是一个简单的宏指令，其中就包含了变量声明和函数呼叫。双斜线 "//" 代表程序批注，在它后面的文字不会被执行。

```
macro_command main()
    short pressure = 10                                    //   局部变数声明
    SetData(pressure, "Allen-Bradley DF1", N7, 0, 1)       //   函数呼叫
end macro_command
```

18.4. 宏指令的语法

18.4.1. 常数和变数

常数

常数是一个可以被各式语句直接使用的固定的资料。有如下格式：

常数类型	使用说明	举例
十进制整数		345, -234, 0, 23456
十六进制数	必须以 0x 开头	0x3b, 0xffff, 0x237
字符型	字符必须使用单引号，字符串使用双引号	‘a’, "data", "函数名称"
布尔型		true, false

下面即为一个简单的常数使用的范例。

```
macro_command main()
    short A, B   //   声明 A 和 B 为短整型变数
    A = 1234
    B = 0x12     //   1234 和 0x12 即为常数
end macro_command
```

变数

变量是一个代表着各种资料的名称。在宏指令中，这些资料可以随着宏指令语句执行的结果改变而改变。

变量的命名规则

- 必须以英文字母开头
- 变量名称长度不超过 32 个字符
- 系统保留寄存器名称不能作为变量名称。

下面为 8 种不同的变量类型，前 5 种为有号数值类型，后 3 种为无号数值类型：

变量类型	描述	范围
bool 布尔型	1 bit (一个位)	0, 1
char 字符型	8 bits (一个字节)	+127 ~ -128
short 短整型	16 bits (一个字符)	+32767 ~ -32768
int 双整型	32 bits (双字符)	+2147483647 ~ -2147483648
float 浮点型	32 bits (双字符)	
unsigned char 字符型	8 bits (一个字节)	0 到 255
unsigned short 短整型	16 bits (一个字符)	0 到 65535
unsigned int 双整型	32 bits (双字符)	0 到 4,294,967,295

变数声明

变量必须在使用前声明。所以，在宏指令，所有的变量都必须在语句使用前都被声明完成。声明变量时，先定义变量的类型，后面再跟着变量名称。

如下范例:

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

数组声明

宏指令支持一维数组 (下标从 0 开始)。声明数组变量时，先定义数组变量的类型，变量名称，接着就是该数组变量的个数，变量个数必须放置在“””符号中。数组变量的长度为 1 ~ 4096。一个宏指令中最多只支持 4096 个变量。

如下范例:

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

数组的下标最小为 0，最大下标为(数组的长度-1)

如下范例:

```
    char data[100]          //   数组变量的长度是 100
```

所以: 最小的数组为 “data”0””，最大的数组为 “data”99””，即 100 - 1 = 99。

变量和数组初始化

有两种方法可以让变量初始化：

- 使用语句中的赋值语句 (=)

如下范例:

```
int a
float b[3]
a = 10
b[0] = 1
```

- 声明变量时直接赋值

```
char a = '5', b = 9
```

数组变量的声明是一个特殊的情况。一个完整的数组被初始化时，可以在数组变量声明时，将数据放置在波形括号**"{}"**里面，各数据使用逗号分开。

如下所示:

```
float data[4] = {11, 22, 33, 44}   // 这样 data[0] = 11, data[1] = 22....
```

18.4.2. 运算符号

运算符通常被用来指定数据是如何被操作和运算，如下：(在任何一个语句中，运算符左边的变量结果均依据运算符右边的条件而获得。)

运算符号	描述	举例
=	赋值运算符号	pressure = 10

数学运算符号	描述	举例
+	加	A = B + C
-	减	A = B - C
*	乘	A = B * C
/	除	A = B / C
%	求余 (返回剩余数)	A = B % 5

比较运算符号	描述	举例
<	小于	if A < 10 then B = 5
<=	小于或者等于	if A <= 10 then B = 5
>	大于	if A > 10 then B = 5
>=	大于或者等于	if A >= 10 then B = 5
==	等于	if A == 10 then B = 5
<>	不等于	if A <> 10 then B = 5

逻辑运算符号	描述	举例
And	与	if A < 10 and B > 5 then C = 10
Or	或	if A >= 10 or B > 5 then C = 10
Xor	异或	if A xor 256 then B = 5
Not	非	if not A then B = 5

移位元和位运算符号通常被用来操作字符型变量、短整型变量和双整型变量的位。在一个语句中，这些运算符号的优先权是在从该语句的左边到右边依此执行的。即在语句中左边位置的优先执行，依次从左到右执行。

移位运算符号	描述	举例
<<	往左移动指定的位数	A = B << 8
>>	往右移动指定的位数	A = B >> 8

位运算符号	描述	举例
&	位与运算	A = B & 0xf
|	位或运算	A = B | C
^	位异或运算	A = B ^ C
~	位取反运算	A = ~B

所有运算符号的优先权

上述所有运算符号的优先权从高到低详细如下所述：

1. 位于圆括号里面的运算符号最优先
2. 数学运算符号
3. 移位和位运算符号
4. 比较运算符号
5. 逻辑运算符号
6. 赋值运算符号

关键词

下面的关键词为宏指令保留使用。这些均不能用来作为变量名称、数组名或者函数名称等。

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

exit, macro_command, for, to, down, step, next, return, bool, short, int, char, float, void, if, then, else, break, continue, set, sub, end, while, wend, true, false

SQRT, CUBERT, LOG, LOG10, SIN, COS, TAN, COT, SEC, CSC, ASIN, ACOS, ATAN, BIN2BCD, BCD2BIN, DEC2ASCII, FLOAT2ASCII, HEX2ASCII, ASCII2DEC, ASCII2FLOAT, ASCII2HEX, FILL, RAND, DELAY, SWAPB, SWAPW, LOBYTE, HIBYTE, LOWORD, HIWORD, GETBIT, SETBITON, SETBITOFF, INVBIT, ADDSUM, XORSUM, CRC, INPORT, OUTPORT, POW, GetError, GetData, GetDataEx,

SetData, SetDataEx, SetRTS, GetCTS, Beep, SYNC_TRIG_MACRO, ASYNC_TRIG_MACRO, TRACE, FindDataSamplingDate, FindDataSamplingIndex, FindEventLogDate, FindEventLogIndex
StringGet, StringGetEx, StringSet, StringSetEx, StringCopy, StringMid, StringDecAsc2Bin, StringBin2DecAsc, StringDecAsc2Float, StringFloat2DecAsc, StringHexAsc2Bin, StringBin2HexAsc, StringLength, StringCat, StringCompare, StringCompareNoCase, StringFind, StringReverseFind, StringFindOneOf, StringIncluding, StringExcluding, StringToUpper, StringToLower, StringToReverse, StringTrimLeft, StringTrimRight, StringInsert。

18.5. 语句

18.5.1. 定义语句

这个定义语句包含了变量和数组的声明。正式的格式如下：
定义一个变量的名称为"名称"且类型为"类型"。
举例:

```
int A       //定义了变量 A 为双整型格式
```

定义一个数组变量为"名称"，大小为"数组长度"且类型为"类型"时。
举例:

```
int B[10]  //定义了一维数组变量 B 的长度为 10，类型为双整型
```

18.5.2. 赋值语句

赋值语句使用赋值运算符号将赋值运算符号右边表达式运算的结果放置到运算符号左边的变量中。一个表达式是由变量、常数和各种运算符号组成，执行后产生一个新的数据。
举例:

```
A = 2           //这样变量 A 就被赋值为 2
```

18.5.3. 逻辑运算语句

逻辑运算语句是根据逻辑（布尔）表达式的结果来执行相应的动作。它的语句如下所示：
单行格式

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

举例:

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

区块格式

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

举例:

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

语法描述

if	必须用在该语句的开始部分。
<Condition>	必要条件。这是一个控制语句。当 <Condition> 为 0 时，即为 “FALES”，(条件为假)；当 <Condition> 为非 0 时，即为“True”(条件为真)。
then	当 <Condition> 执行为 “TRUE” (真) 时，必须放置在需要执行的语句之前。
[Statements]	在区块形式中是可选择的参数，在单行形式中，且没有 else 子句时，为必要参数，该语句在 <Condition> 为真时执行。
else if	可选，一条或多条语句，在相对应的 <Condition - n> 为 true 时执行。
<Condition-n>	可选，解释同 Condition
else	可选，在上述 Condition 和 Condition - n 都不为 true 时执行。
end if	必须。在一个 if-then 语句中使用这个来结束 if-then 语句。

18.5.4. 多重判断语句

Select-case 可用来处理多重判断的叙述，其功能类似 if-else 语句。根据所指定变量的值，分别对应到符合该值的 case，并执行 case 下面的叙述，直到遇到 break 叙述时，才跳到结束符号 end select 处。语法结构如下：

没有预设 case 的形式：

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

举例:

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

有预设 case 的形式：

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

举例:

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

```
end Select
```

多个不同 case 对应到相同区块：

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

end Select
```

举例:

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

语法描述

Select Case	必须用在该语句的开始部分。
“variable”	必要条件。此变量将会与每一个 case 做比较。
Case else	可选。代表预设 case。当 “variable” 的值不符合任何一个 case 时，将会执行此叙述下面的区块。在没有预设 case 的情况，当 “variable” 的值不符合任何一个 case 时，将不会做任何动作而直接跳出 select 控制结构。
break	可选。跳到某一个 case 下面执行时，将一句一句执行 case 语句下面的叙述直到遇到 break 命令才结束，并跳到 end select 叙述。当 case 叙述下面没有任何 break 命令时，流程将不断往下执行，直到遇到 end select 叙述，才结束并跳出 select 控制结构。
end Select	select-case 语句的结束标志。

18.5.5. 循环语句

循环语句依据循环条件来反复的执行一个任务。循环语句有两种表达方式。

for next 语句

For-next 语句通常用来执行次数固定的循环任务。一个变量用作为任务执行次数的计数器和结束循环任务

执行的条件。这个变量为固定执行的次数。语法结构如下：

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

或者

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

举例:

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

语法描述

for	必须用在该语句的开始部分。
[Counter]	必要，循环计数器的数值变量，该变量的结果用来计数循环的次数。
<StartValue>	必要，Counter 的初值。
to/down	必要。用来决定步长是递增还是递减。 “to” 以 <StepValue> 为步长递增 <Counter> “down” 以 <StepValue> 为步长递减 <Counter>
<EndValue>	必要，Counter 的终值、测试点。当 <Connter> 大于该值时，宏指令将结束这个循环任务。
step	可选，指定 <Step Value> 的步长，指定为 1 以外的数值。
[StepValue]	可选，Counter 的步长，只能是数值，如果没有指定，则预设为 1。
[Statements]	可选，for 和 next 之间的语句区块，该语句区块将执行所指定的次数。
next	必须的。
[Counter]	可选。

while-wend 语句

While-wend 语句是用来执行不确定次数的循环任务。设置一个变量用来判断结束循环的条件。当条件为“True”时，该语句将一直循环执行直到条件变为 “False”。语法结构如下：

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

举例:

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

语法描述

while	必须用在该语句的开始部分。
continue	必要条件。 这是一个控制语句。当为“True”时，开始执行循环命令，当为“False”时，结束执行循环命令。
return [value]	当判断为“TRUE”时，继续执行循环命令
wend	While-wend 语句的结束标志。

其它控制命令

break	用在 for-next 和 while-wend 语句中。当遇到此语句时，立即跳到语句的结束部分。
continue	用在 for-next 和 while-wend 语句中。当遇到此语句时，立即结束当前循环命令而开始执行下一个循环命令。
return	可用在自订 function 的回传值叙述。写在主函数里面时，用来强制跳出主函数。

18.6. 子函数

使用子函数可以有效的减少循环命令的代码，子函数必须在使用前被定义，且可以使用任何变量和语句类型。在主函数中，将子函数的参数放置在子函数名称后面的圆括号中，即可调用子函数。子函数被执行后，将执行后的结果返回到主函数需要的赋值语句或者条件中。定义子函数时，不一定要有返回值，且参数部分可以为空。在主函数中调用子函数时，调用方式应符合其定义。语法结构如下：

有返回值的子函数语法：

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

举例:

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

```
        return result
    end sub

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

或:

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

    macro_command main()
        int   sum
        sum = Add()
    end macro_command
```

没有返回值的子函数语法:

```
sub <函数名称>   [(parameters)]
    Local variable declarations
    [Statements]
end sub
```

举例:

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

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

或:

```
sub Add()
```

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

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

语法描述

sub	必须用在该子函数的开始部分。
type	可选。用来定义子函数执行后返回的数据类型。子函数也可以不回传任何值。
(parameters)	可选。这些参数保留了从主函数传入的数值。这些被传入的参数必须使用与在参数变量声明的类型一致。 举例： sub int MyFunction(int x, int y). x 和 y 必须为从主函数中传过来的双整型数据格式的数据。调用此子函数的语句格式大致为这样： ret = MyFunction(456, pressure)，其中 pressure 需为双整型数据格式方符合子函数参数变量的声明。 请注意调用语句的参数部分可以是常数也可以是变量。当执行这个子函数后，一个双整型数据将会返回给变量“ret”。
Local variable declaration	除了被传递的参数之外，子函数中使用的变量必须事先声明。在上面的“举例”中，X 和 Y 就是子函数可以使用的变量。全局变量也可以用在子函数中。
[Statements]	需要执行的语句。
[return [value]]	可选。用来将执行的结果返回给调用语句。这个结果可以是一个常数或者变量。返回后同时也结束了子函数的执行。子函数也可以不回传任何值，但是当 type 部分有定义时，则必须加上此 return 叙述。
end sub	必须的。用来结束子函数。

18.7. 内置函数功能

EasyBuilder Pro 软件宏指令中本身提供了一些内建的函数用来从 PLC 获取数据和传输数据到 PLC、数据处理和数学运算等。

18.7.1. 数学运算函数

函数名称	SQRT
语法	SQRT(source, result)
描述	开平方根。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。数据来源必须为一个正数。

举例	macro_command main() float source, result SQRT(15, result) source = 9.0 SQRT(source, result)// 执行后 result = 3.0 end macro_command

函数名称	CUBERT
语法	CUBERT (source, result)
描述	开三次方根。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。数据来源必须为一个正数。

举例	macro_command main()
	float source, result
	
	CUBERT (27, result) // 执行后 result = 3.0
	
	source = 27.0
	CUBERT (source, result) // 执行后 result = 3.0
	
	end macro_command

函数名称	POW
语法	POW (source1, source2, result)
描述	计算 source1 的某次方 (source2) 。数据来源 source1 和 source2 可以是常数或者变量，但是存放结果的 result 必须为变量。数据来源必须为一个正数。
举例	macro_command main() float y, result y = 0.5 POW (25, y, result) // 执行后 result = 5 end macro_command

函数名称	SIN
语法	SIN(source, result)
描述	三角函数的正弦计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。
举例	macro_command main() float source, result SIN(90, result) // result is 1 source = 30

	SIN(source, result) // result is 0.5 end macro_command

函数名称	COS
语法	COS(source, result)
描述	三角函数的余弦计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

举例	macro_command main()
	float source, result
	
	COS(90, result) // result is 0
	
	source = 60
	GetData(source, "Local HMI", LW, 0, 1)
	COS(source, result) // result is 0.5
	
	end macro_command

函数名称	TAN
语法	TAN(source, result)
描述	三角函数的正切计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	COT
语法	COT(source, result)
描述	三角函数的余切计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。
举例	macro_command main() float source, result COT(45, result) // result is 1 source = 60

	COT(source, result) // result is 0.5774 end macro_command

函数名称	SEC
语法	SEC(source, result)
描述	反三角函数中反正割计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	CSC
语法	CSC(source, result)
描述	反三角函数中反余割计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	ASIN
语法	ASIN(source, result)
描述	反三角函数中反正弦计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	ACOS
语法	ACOS(source, result)
描述	反三角函数中反余弦计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	ATAN
语法	ATAN(source, result)
描述	反三角函数中反正切计算。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	LOG
语法	LOG (source, result)
描述	从取得 source 的自然对数，存入 result 变量。 source 可为变数或常数。 存放结果的 result 必须为变量。
举例	macro_command main() float source=100, result LOG (source, result) // result 约等于 4.6052 end macro_command

函数名称	LOG10
语法	LOG10 (source, result)
描述	从取得 source 的以 10 为基底的对数，存入 result 变数。 Source 可为变数或常数。 存放结果的 result 必须为变量。
举例	macro_command main() float source=100, result LOG10 (source, result) // result 等于 2 end macro_command

函数名称	RAND
语法	RAND(result)
描述	产生一个随机数 存放结果的 result 必须为变量。
举例	macro_command main() short result RAND (result) // result is not a fixed value when executes macro every time end macro_command

18.7.2. 数据转换函数

函数名称	BIN2BCD
语法	BIN2BCD(source, result)
描述	将 BIN 格式的数据 (source) 转换为 BCD 格式的数据 (result)。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	BCD2BIN
语法	BCD2BIN(source, result)
描述	将 BCD 格式的数据 (source) 转换为 BIN 格式的数据 (result)。数据来源 source 可以是常数或者变量，但是存放结果的 result 必须为变量。

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

函数名称	DEC2ASCII
语法	DEC2ASCII(source, result[start], len)
描述	将十进制的数据 (source) 转换为 ASCII 格式的数据，并存放在一个一维数组 (result) 中。len 表示这个转换后的字符串的长度，同时这个长度也取决于存放结果的一维数组的数据格式。例如：如果 resul t 一维数组的格式为“char” (字符型，长度为一个字节)，则长度为 “字节数*len” 。如果 result 一维数组的格式为 “short” (短整型数据，2 个字节)，则长度为 “word*len”。依此类推。 转换后的第一个字符放在 result”start” 中，第二个字符放在 result”start+1” 中，最后一个字符放在 result”start+(len-1)”中。 source 和 len 可以是常数或者变量，单数 result 必须为变量。start 必须为常数。

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

函数名称	HEX2ASCII
语法	HEX2ASCII(source, result”start”, len)
描述	十六进制格式数据 (source) 转换为 ASCII 格式的数据，并将结果存放在一个一维数组 (result) 中。len 表示这个转换后的字符串的长度，同时这个长度也取决于存放结果的一维数组的数据格式。例如：如果 result 一维数组的格式为“char” (字符型，长度为一个字节)，则长度为 “字节数*len” 。如果 result 一维数组的格式为 “short” (短整型数据，2 个字节)，则长度为 “word*len” 。依此类推。 source 和 len 可以是常数或者变量，单数 result 必须为变量。start 必须为常数。
举例	macro_command main() short source char result[4] source = 0x5678 HEX2ASCII(source, result[0], 4) // result[0] is '5', result[1] is '6', result[2] is '7', result[3] is '8' end macro_command

函数名称	FLOAT2ASCII
语法	FLOAT2ASCII (source, result[start], len)
描述	浮点数格式数据 (source) 转换为 ASCII 格式的数据，并将结果存放在一个一维数组 (result) 中。len 表示这个转换后的字符串的长度，同时这个长度也取决于存放结果的一维数组的数据格式。例如：如果 result 一维数组的格式为 “char” (字符型，长度为一个字节)，则长度为 “字节数*len” 。如果 result 一维数组的格式为 “short” (短整型数据，2 个字节)，则长度为 “word*len” 。依此类推。 source 和 len 可以是常数或者变量，单数 result 必须为变量。start 必须为常数。
举例	macro_command main() float source char result[4] source = 56.8 FLOAT2ASCII (source, result[0], 4) // result[0] is '5', result[1] is '6', result[2] is '.', result[3] is '8' end macro_command

函数名称	ASCII2DEC
语法	ASCII2DEC(source[start], result, len)
描述	将字符型 ASCII 数据 (source) 转换为十进制格式的数据，并存放在 result 变数中。ASCII 的长度即为 len，第一个字符的位置即为 source[start] 的数据。 source 和 len 可以是常数或者变量，单数 result 必须为变量。start 必须为常数。
举例	macro_command main() char source[4] short result source[0] = '5' source[1] = '6' source[2] = '7' source[3] = '8' ASCII2DEC(source[0], result, 4) // result is 5678 end macro_command

函数名称	ASCII2HEX
语法	ASCII2HEX (source[start], result, len)
描述	将 ASCII 字符型数据 (source) 转换为十六进制的数据，并存放在 result 变数中。字符的长度即为 len 的数据。第一个字符存放在 source[start] 中。 source 和 len 可以是常数或者变量，单数 result 必须为变量。start 必须为常数。
举例	macro_command main() char source[4] short result source[0] = '5' source[1] = '6' source[2] = '7' source[3] = '8' ASCII2HEX(source[0], result, 4) // result is 0x5678 end macro_command

函数名称	ASCII2FLOAT
语法	ASCII2FLOAT (source[start], result, len)
描述	将字符型 ASCII 数据 (source) 转换为浮点数格式的数据，并存放在 result 变数中。ASCII 的长度即为 len，第一个字符的位置为 source[start] 的数据。 source 和 len 可以是常数或者变量，单数 result 必须为变量。Start 必须为常数。
举例	macro_command main() char source[4] float result source[0] = '5' source[1] = '6' source[2] = '.' source[3] = '8' ASCII2FLOAT(source[0], result, 4) // result is 56.8 end macro_command

18.7.3. 数据操作函数

函数名称	FILL
语法	FILL(source[start], preset, count)
描述	依序将默认值 (preset) 放置到一维数组 source[start] 开始的数组中，放置的数据个数由 count 决定。 source 和 start 必须为变量，preset 可以为一个常数或者变量。

举例	macro_command main()
	char result[4]
	char preset
	
	FILL(result[0], 0x30, 4)
	// result[0] is 0x30, result[1] is 0x30, , result[2] is 0x30, , result[3] is 0x30
	
	preset = 0x31
	FILL(result[0], preset, 2) // result[0] is 0x31, result[1] is 0x31
	
	end macro_command

函数名称	SWAPB
语法	SWAPB(source, result)
描述	将一个 16 位字的高低字节颠倒，并将结果存放在 result 变量中。 source 可以是常数或者是变量，单数 result 必须为变量。

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

函数名称	SWAPW
语法	SWAPW(source, result)
描述	将一个 32 位双整型数据的高位字符和低位字符颠倒，并将结果存放在 result 变量中。source 可以是常数或者变量，但是 result 必须为变量。

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

	end macro_command

函数名称	LOBYTE
语法	LOBYTE(source, result)
描述	获取一个 16 位数据的低字节，并且放置在 result 变量中。 source 可以是常数或者变量，但是 result 必须为变量。

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

函数名称	HIBYTE
语法	HIBYTE(source, result)
描述	获取一个 16 位数据的高字节，并且放置在 result 变量中。 source 可以是常数或者变量，但是 result 必须为变量。

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

函数名称	LOWORD
语法	LOWORD(source, result)
描述	获取一个 32 位数据的低位字符，并将结果放置在 result 变量中。 source 可以是常数或者变量，但是 result 必须为变量。

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

函数名称	HIWORD
语法	HIWORD(source, result)
描述	获取一个 32 位数据的高位字符，并将结果放置在 result 变量中。 source 可以是常数或者变量，但是 result 必须为变量。

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

18.7.4. 位状态转换

函数名称	GETBIT
语法	GETBIT(source, result, bit_pos)
描述	获取数据或者变量 (source) 指定的位的状态，并将结果放置在 result 变量中。 result 的数据将为 1 或者 0。 source 和 bit_pos 可以是常数或者变量，但是 result 必须为变量。

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

函数名称	SETBITON
语法	SETBITON(source, result, bit_pos)
描述	将数据或者变量 (source) 指定的位地址设置为 1，并将改变后的数据存放在 result 变量中。 source 和 bit-pos 可以是常数或者变量，但是 result 必须为变量。

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

函数名称	SETBITOFF
语法	SETBITOFF(source, result, bit_pos)
描述	将数据或者变量 (source) 指定的位地址设置为 0，并将改变后的数据存放在 result 变量中。 source 和 bit-pos 可以是常数或者变量，但是 result 必须为变量。

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

函数名称	INVBIT
语法	INVBIT(source, result, bit_pos)
描述	将数据或者变量 (source) 指定的位地址状态相反，并将改变后的数据存放在 result 变量中。 source 和 bit-pos 可以是常数或者变量，但是 result 必须为变量。

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

18.7.5. 通讯有关的函数

函数名称	DELAY
语法	DELAY(time)
描述	让宏指令暂停执行，持续的时间至少是指定的这个时间。时间的单位为毫秒。 time 可以是常数或者变量。

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

函数名称	ADDSUM
语法	ADDSUM(source[start], result, data_count)
描述	将 source[start] 到 source[start+data-count-1] 的所有一维数组的数据累加起来，以获得 checksum (校验和)，并将结果存放在 result 变量中。 result 必须为变量，data_count 是进行累加的资料的个数，可以是常数或者是变量。
举例	macro_command main() char data[5] short checksum data[0] = 0x1 data[1] = 0x2 data[2] = 0x3 data[3] = 0x4 data[4] = 0x5 ADDSUM(data[0], checksum, 5) // checksum is 0xf end macro_command

函数名称	XORSUM
语法	XORSUM(source[start], result, data_count)
描述	将 source[start] 到 source[start+data-count-1] 的所有一维数组的数据进行异或运算，以获得 checksum (校验和)，并将结果存放在 result 变量中。 result 必须为变量，data_count 是进行异或计算的数据的个数，可以是常数或者是变量。
举例	macro_command main() char data[5] = {0x1, 0x2, 0x3, 0x4, 0x5} short checksum XORSUM(data[0], checksum, 5) // checksum is 0x1 end macro_command

函数名称	CRC
语法	CRC(source[start], result, data_count)
描述	将 source[start] 到 source[start+data-count-1] 的所有一维数组的数据进 16-bit CRC 计算，以获得 checksum (校验和)，并将结果存放在 result 变量中。 result 必须为变量，data_count 是进行计算的资料的个数，可以是常数或者是变量。
举例	macro_command main() char data[] = {0x1, 0x2, 0x3, 0x4, 0x5} short 16bit_CRC CRC(data[0], 16bit_CRC, 5) // 16bit_CRC is 0xbb2a end macro_command

函数名称	OUTPORT
语法	OUTPORT(source[start], device_函数名称, data_count)
描述	将放置在从 source[start] 到 source[start+count-1] 的所有数据通过串行端口或者以太网口传送给 PLC 或者控制器中。 device_函数名称是在 “设备列表” 中定义的 “PLC 名称”，而这个 device 必须选择为 “Free Protocol” 这个 PLC 类型。 Data_count 是发送数据的个数，可以是常数或变量。

举例	要使用 OUTPORT 函数，必须要在 PLC 类型中选择 “Free Protocol”，如下图所示。
	 
	
	这里的 device_函数名称即为 “MODBUD RTU Device”。端口的属性也是依据在这个 “系统参数” 中的设定，譬如，在此设定为 (9600，E，8，1…)
	
	下面是一个范例程序，使用自由协议，以 MODBUS RTU 的协议格式，将单个寄存器设置为 ON。
	
	macro_command main()
	
	char command[32]
	short address, checksum
	
	FILL(command[0], 0, 32) // 初始化命令
	
	Command[0] = 0x1 // 站号
	Command[1] = 0x5 // 功能码 : 写单个位
	
	address = 0
	HIBYTE(address, command[2])
	LOBYTE(address, command[3])
	
	Command[4] = 0xff // 使该bit设置为ON
	Command[5] = 0
	
	CRC(command[0], checksum, 6)

	LOBYTE(checksum, command[6])
	HIBYTE(checksum, command[7])
	
	// 将命令通过串行埠送出去
	OUTPORT(command[0], "MODBUS RTU Device", 8)
	
	end macro_command

函数名称	INPORT
语法	INPORT(read_data[start], device_函数名称, read_count, return_value)
描述	从串行端口或者以太网口读数据到触摸屏界面上。这些资料保存在 read-data[start]~read-data[start+read-count-1] 这个一维数组中。同样的，device_函数名称见上面的说明，在此不再详述。 read-count 是设定的需要读取的命令的位组长度，它可以是一个常数或变量。如果这个函数能够成功的从 PLC 或者控制器中读取到数据，则return_value 的值为 1，否则就为 0。

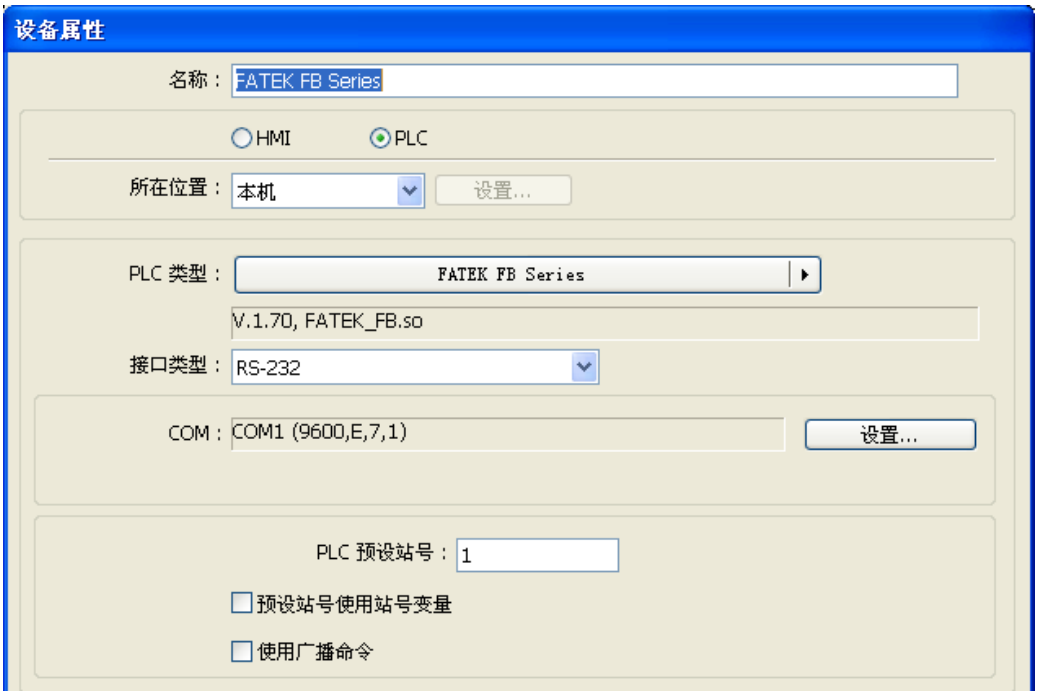
举例	下面就是一个使用 INPORT 函数读取一个 MODBUS 设备保持寄存器数据的范例。
	
	// 读取保持寄存器数据
	macro_command main()
	char command[32], response[32]
	short address, checksum
	short read_no, return_value, read_data[2]
	
	FILL(command[0], 0, 32) // 命令初始化
	FILL(response[0], 0, 32)
	
	Command[0] = 0x1 // 站号
	Command[1] = 0x3 // 功能码：读取保持寄存器
	
	address = 0
	HIBYTE(address, command[2])
	LOBYTE(address, command[3])
	
	read_no = 2 // read 2 words (4x_1 and 4x_2)
	HIBYTE(read_no, command[4])
	LOBYTE(read_no, command[5])
	
	CRC(command[0], checksum, 6)
	
	LOBYTE(checksum, command[6])
	HIBYTE(checksum, command[7])
	
	// 使用 OUTPORT 函数将命令送出去
	OUTPORT(command[0], "MODBUS RTU Device", 8)
	
	// 使用 INPORT 函数读取返回的命令

```
INPORT(response[0], "MODBUS RTU Device", 9, return_value)

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

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

end macro_command
```

函数名称	INPORT2
语法	INPORT2(response[start], device_name, receive_len, wait_time)
描述	从串行端口或者以太网口读数据到触摸屏界面上。这些数据保存在 response 这个一维数组中。device_name 的说明与 OUTPORT 相同，在此不再详述。 receive_len 存放所接收到的数据位组长度，必须为变量。receive_len 的最大长度将不会超过 response 数组宣告的大小。wait_time 表示等待时间（单位是 millisecond），可以是一个常数或变量。当数据读取完毕后，若在指定的等待时间内未再收到任何数据，此函数便结束执行并回传结果。

举例	macro_command main()
	
	short wResponse[6], receive_len, wait_time=20
	
	INPORT2(wResponse[0], "Free Protocol", receive_len, wait_time)
	// wait_time 单位 ：millisecond
	
	if receive_len > 0 then
	  SetData(wResponse[0], "Local HMI", LW, 0, 6)
	  // 把响应的数据写入LW0
	end if
	
	end macro_command

函数名称	GetData
语法	GetData(read_data[start], device_函数名称, device_type, address_offset, data_count) or GetData(read_data, device_函数名称, device_type, address_offset, 1)
描述	获取 PLC 的资料。数据是存储在 read_data[start]~read_data[start+data_count-1] 这些一维数组变量中。 data_count 是设定的读取数据的个数。一般来说，read_data 是一个一维数组，但是如果 data_count 是 1，read_data 可以是一个一维数组，也可以是一个普通的变量。下面是两种从 PLC 中读取一个字的方法。 macro_command main() short read_data_1[2], read_data_2 GetData(read_data_1[0], “FATEK FB Series”, RT, 5, 1) GetData(read_data_2, “FATEK FB Series”, RT, 5, 1) end macro_command 此处的 device_函数名称，即为在 “系统参数” 中建立 PLC 类型时，设定的“PLC 名称”。在此，PLC 名称被设定为 “FATEK FB Series”，如下图所示。  device_type 是设备类型和 PLC 中数据的编码方式。例如：如果 device_type 是 LW_BIN，那么读取的设备类型为 LW，数据编码方式为 BIN。如果使用 BIN 编码方式，“_BIN” 可以忽略。 如果 device_type 是 LW_BCD，表示设备类型 LW，数据的编码方式为 BCD 格式。 address_offset 是 PLC 中的地址偏移量。 例如，GetData(read_data_1[0], “FATEK FB Series”, RT, 5, 1) 代表读取的设备地址偏移量为 5。 如果 address_offset 使用格式为 “N#AAAAA”，N 表示 PLC 的站号，AAAAA 表示地址偏移量。此情况一般使用在同一个串行埠上连接有多台 PLC 或者控制器的情况下。例如：GetData(read_data_1[0], “FATEK FB Series”, RT, 2#5, 1) 表示读取站号为 2 的 PLC 的数据。如果 GetData() 使用 “系统参数 / 设备列表” 中设定的默认的站号，在此可以不填这个站号。



从 PLC 中读取的资料个数，根据 read_data 变量的类型和 data_count 的值来决定的。如下表所示：

read_data的类型	data_count的值	读取16位数据的个数
char (8-bit)	1	1
char (8-bit)	2	1
boo□(8-b□t)	1	1
b□ol (8-bit)	2	1
short (16-bit)	1	1
short (16-bit)	2	2
int (32-bit)	1	2
int (32-bit)	2	4
float (32-bit)	1	2
float (32-bit)	2	4

当 Getdata() 函数读取 32 位的数据类型 (int 或者float型) 时，此函数会自动的转换这个数据。例如：

```
macro_command main()
    float f
    GetData(f, "MODBUS", 6x, 2, 1)  // f 中将会是浮点型的数据

end macro_command
```

举例	macro_command main()
	bool a
	bool b[30]
	short c
	short d[50]
	int e
	int f[10]
	double g[10]
	
	// 读取 LB2 的状态到变量 a 中
	GetData(a, “Local HMI”, LB, 2, 1)
	
	// 读取 LB0~LB29共 30 个状态，到变量 b[0] ~ b[29] 中
	GetData(b[0], “Local HMI”, LB, 0, 30)
	
	// 读取 LW-2 的数据到变量 c 中
	GetData(c, “Local HMI”, LW, 2, 1)
	
	// 读取 LW-0 ~ LW-49 共 50 个字到变数 d[0] 到 d[49] 中
	GetData(d[0], “Local HMI”, LW, 0, 50)
	
	// 读取两个字 LW-6 ~ LW-7 到变数 e 中
	// 注意此时变量 e 的类型为 int
	GetData(e, “Local HMI”, LW, 6, 1)
	
	// 读取 LW-0 ~ LW-19 共 20 个字到变数 f[0] ~ f[9] 中 (共 10 个 int 型变量)，数组 f[10] 的变量类型定义为 int。
	// 注意一个 int 资料将占据 2 个字符
	GetData(f[0], “Local HMI”, LW, 0, 10)
	
	// 读取 LW-2 ~ LW-3 共 2 个字到变数 f 中
	GetData(f, “Local HMI”, LW, 2, 1)
	
	end macro_command

函数名称	GetDataEx
语法	GetDataEx (read_data[start], device_函数名称, device_type, address_offset, data_count) or GetDataEx (read_data, device_函数名称, device_type, address_offset, 1)
描述	获取 PLC 的数据，不等待 PLC 响应，径自往下执行。 read_data、device_函数名称、device_type、address_offset 和 data_count的说明和 GetData 相同。
举例	macro_command main() bool a bool b[30] short c

```
short d[50]
int e
int f[10]
double g[10]

// 读取 LB-2 的状态到变量 a 中
GetDataEx (a, “Local HMI”, LB, 2, 1)

// 读取 LB-0 ~ LB-29 共 30 个状态，到变量 b[0] ~ b[29] 中
GetDataEx (b[0], “Local HMI”, LB, 0, 30)

// 读取 LW-2 的数据到变量 c 中
GetDataEx (c, “Local HMI”, LW, 2, 1)

// 读取 LW-0 ~ LW-49 共 50 个字到变数 d[0] 到 d[49 ]中
GetDataEx (d[0], “Local HMI”, LW, 0, 50)

// 读取两个字 LW-6 ~ LW-7 到变数 e 中
// 注意此时变量 e 的类型为 int
GetDataEx (e, “Local HMI”, LW, 6, 1)

// 读取 LW-0 ~ LW-19 共 20 个字到变数 f[0] ~ f[9] 中，数组 f[10] 的变量类型定义为 int。
// 注意一个 int 资料将占据 2 个字符
GetDataEx (f[0], “Local HMI”, LW, 0, 10)

// 读取 LW-2 ~ LW-3 共 2 个字到变数 f 中
GetDataEx (f, “Local HMI”, LW, 2, 1)

end macro_command
```

函数名称	SetData
语法	SetData(send_data[start], device_函数名称, device_type, address_offset, data_count) or SetData(send_data, device_函数名称, device_type, address_offset, 1)
描述	将数据写到 PLC 中。资料保存在 send_data[start]~send_data[start+data_count-1] 中。 data_count 是写入到 PLC 中资料的个数。一般来说，send_data 是一个数组。但是如果 data_count 是 1，send_data 可以是一个数组也可以是一个普通的变量。下面是写一个数据到 PLC 中的方法。 macro_command main() short send_data_1[2] = { 5, 6}, send_data_2 = 5 SetData(send_data_1[0], “FATEK FB Series”, RT, 5, 1) SetData(send_data_2, “FATEK FB Series”, RT, 5, 1)

```
end macro_command
```

device_函数名称详见上面的说明，在此不在说明。
device_type 是设备类型和 PLC 中数据的编码方式。例如：如果 device_type 是 LW_BIN，那么读取的设备类型为 LW，数据编码方式为 BIN。如果使用 BIN 编码方式，“_BIN”可以忽略。
如果 device_type 是 LW_BCD，表示设备类型 LW，数据的编码方式为 BCD 格式。
address_offset 是 PLC 中的地址偏移量。
例如，SetData (read_data_1[0], “FATEK FB Series”, RT, 5, 1) 代表读取的设备地址偏移量为 5。
如果 address_offset 使用格式为 “N#AAAAA”，N 表示 PLC 的站号，AAAAA 表示地址偏移量。此情况一般使用在同一个串行埠上连接有多台 PLC 或者控制器的情况下。例如：SetData(send_data_1[0], “FATEK FB Series”, RT, 2#5, 1) 表示设定站号为 2 的 PLC 的数据。如果 SetData()使用 “系统参数 / 设备列表” 中设定的默认的站号，在此可以不填这个站号。

设定到 PLC 的数据个数，根据 sead_data 变量的类型和 data_count 的值来决定的。如下表所示：

sead_data 的类型	data_count 的值	设定 16 位数据的个数
char (8-bit)	1	1
char (8-bit)	2	1
bool (8-bit)	1	1
bool (8-bit)	2	1
short (16-bit)	1	1
short (16-bit)	2	2
int (32-bit)	1	2
int (32-bit)	2	4
float (32-bit)	1	2
float (32-bit)	2	4

当 Setdata() 函数写入 32 位的数据类型 (int 或者 float 型) 到 PLC 时，此函数会自动的转换这个数据。例如：

```
macro_command main()
float f = 2.6
SetData(f, "MODBUS", 6x, 2, 1) // 在此将会设定一个浮点数到 PLC 中
end macro_command
```

举例

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

```
int f[10]

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

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

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

// 变量 a 的数值设定到 LB2 中
SetData(a, “Local HMI”, LB, 2, 1)


// 设定 LB0 ~ LB29 共 30 个位的状态
SetData(b[0], “Local HMI”, LB, 0, 30)

// 将变量 c 的值设定到 LW-2 中
SetData(c, “Local HMI”, LW, 2, 1)

// 设定 LW-0 ~ LW-49 共 50 个数据
SetData(d[0], “Local HMI”, LW, 0, 50)

// 将变量 e 的值写入到 LW-6 ~ LW-7 两个寄存器中，注意变量 e 的类型为int。
SetData(e, “Local HMI”, LW, 6, 1)

// 设定 LW-0 ~ LW-19 共 20 个字的数据
// 10 个双整型数据等于 20 个 16 位整型数 (1个字符)，因一个 int 数据将占据 2 个字符
SetData(f[0], “Local HMI”, LW, 0, 10)

end macro_command
```

函数名称	SetDataEx
语法	SetDataEx (send_data[start], device_函数名称, device_type, address_offset, data_count) or SetDataEx (send_data, device_函数名称, device_type, address_offset, 1)
描述	将数据写到 PLC 中，不等待 PLC 回应，径自往下执行。 send_data、device_函数名称、device_type、address_offset和data_count的说明和 SetData 相同。
举例	macro_command main() int i

```
bool a = true
bool b[30]
short c = false
short d[50]
int e = 5
int f[10]

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

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

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

// 将变量 a 的数值设定到 LB2 中
SetDataEx(a, “Local HMI”, LB, 2, 1)


// 设定 LB0 ~ LB29 共 30 个位的状态
SetDataEx (b[0], “Local HMI”, LB, 0, 30)

// 将变量 c 的值设定到 LW-2 中
SetDataEx (c, “Local HMI”, LW, 2, 1)

// 设定 LW-0 ~ LW-49 共 50 个数据
SetDataEx (d[0], “Local HMI”, LW, 0, 50)

// 将变量 e 的值写入到 LW-6 ~ LW-7 两个寄存器中，注意变量 e 的类型为int。
SetDataEx (e, “Local HMI”, LW, 6, 1)

// 设定 LW-0 ~ LW-19 共 20 个字的数据
// 10 个双整型数据等于 20 个 16 位整型数 (1个字符)，因一个 int 数据将占据 2 个字符
SetDataEx (f[0], “Local HMI”, LW, 0, 10)

end macro_command
```

函数名称	GetError
语法	GetError(err)
描述	取得错误码。

举例	macro_command main()
	short err
	char byData[10]
	
	GetDataEx(byData[0], “MODBUS RTU”, 4x, 1, 10) // 读取 10 byte数据
	
	// 当错误码 (err) 为 0，表示 GetDataEx 成功执行
	GetErr(err) // 读取错误码，存入变量err
	
	end macro_command

函数名称	PURGE
语法	PURGE (com_port)
描述	com_port 表示串行端口编号，支持 COM1 ~ COM3。此参数可以为变量或常数。 可利用这个指令来清空 COM port 的输出入缓冲区。
举例	macro_command main() int com_port=3 PURGE (com_port) PURGE (1) end macro_command

函数名称	SetRTS
语法	SetRTS(com_port, source)
描述	拉高或拉低 RS-232 之 RTS 讯号。 com_port 表示串行端口编号，支持 COM1。此参数可以为变量或常数。source参数也可以为变量或常数。 当传入的 source 参数值大于 0 时，将拉高 RTS 电位，当 source 参数值等于 0 时，将拉低 RTS 电位。

举例	macro_command main()
	char com_port=1
	char value=1
	
	SetRTS(com_port, value) // value > 0，拉高 COM1 之 RTS 电位
	
	SetRTS(1, 0) // 拉低 COM1 之 RTS 电位
	
	end macro_command

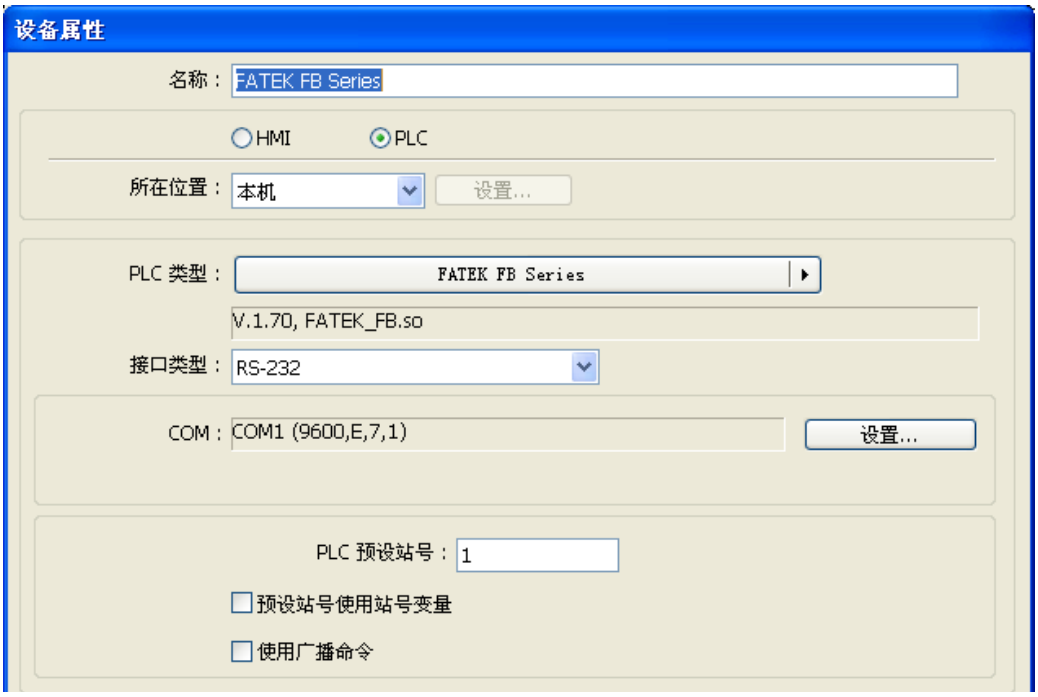
函数名称	GetCTS
语法	GetCTS(com_port, result)
描述	侦测 RS-232 之 CTS 讯号。 com_port 表示串行端口编号，支持 COM1。此参数可以为变量或常数。result参数必须为变量。 此函数可侦测 CTS 电位值，当 CTS 在高电位时，result 将写入 1，否则写入 0。

举例	macro_command main()
	char com_port=1
	char result
	
	GetCTS(com_port, result) // 侦测 COM1 之 CTS 电位值
	
	GetCTS (1, result) // 侦测 COM1 之 CTS 电位值
	
	end macro_command

18.7.6. 字符串处理函数

函数名称	StringGet
语法	StringGet(read_data[start], device_函数名称, device_type, address_offset, data_count)
描述	获取 PLC 的资料。字符串的数据类型为字符数组，是存储在 read_data[start]~read_data[start+data_count-1] 这些一维数组变量中。 read_data 必须为一维字符数组。 data_count 是设定的读取字符的个数，可以是常数也可以是变量。 此处的 device_函数名称，即为在“系统参数”中建立 PLC 类型时，设定的“PLC 名称”。在此，PLC 名称被设定为 “FATEK FB Series”，如下图所示。  device_type 是设备类型和 PLC 中数据的编码方式。例如：如果 device_type 是 LW_BIN，那么读取的设备类型为 LW，数据编码方式为 BIN。如果使用 BIN 编码方式，“_BIN”可以忽略。 如果 device_type 是 LW_BCD，表示设备类型 LW，数据的编码方式为 BCD 格式。 address_offset 是 PLC 中的地址偏移量。 例如，StringGet(read_data_1[0], “FATEK FB Series”, RT, 5, 1）代表读取的设备地

址偏移量为 5。
如果 address_offset 使用格式为 “N#AAAAA”，N 表示 PLC 的站号，AAAAA 表示地址偏移量。此情况一般使用在同一个串行埠上连接有多台 PLC 或者控制器的情况下。例如：StringGet(read_data_1”0”, “FATEK FB Series”, RT, 2#5, 1）表示读取站号为 2 的 PLC 的数据。如果 StringGet()使用 “系统参数 / 设备列表” 中设定的默认的站号，在此可以不填这个站号。



从 PLC 中读取的数据个数，由 data_count 的值来决定，因 read_data 变量仅接受 char 数组类型。如下表所示：

read_data 的类型	data_count 的值	读取 16 位数据的个数
char (8-bit)	1	1
char (8-bit)	2	1

因为一个 WORD (16 位) 等于 2 个 ASCII 字符的长度，当设备类型长度为 WORD 时，根据上表，读取 2 个 ASCII 字符实际上是读 1 个 WORD 的数据。

举例

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

// 读取 LW-0 ~ LW-9 共 10 个 WORD 到变数 str1[0] 到 str1[19] 中
// 因 1 WORD 可存放 2 个 ASCII 字符，欲读取 20 个 ASCII 字符
// 实际上共读取了 10 个WORD
StringGet(str1[0], “Local HMI”, LW, 0, 20)

end macro_command
```

函数名称	StringGetEx
语法	StringGetEx (read_data[start], device_函数名称, device_type, address_offset, data_count)
描述	获取 PLC 的数据，不等待 PLC 响应，径自往下执行。 read_data、device_函数名称、device_type、address_offset 和 data_count 的说明和 GetData 相同。

举例	macro_command main()
	char str1[20]
	short test=0
	
	// 当 MODBUS 设备未回应，test = 1将照常执行
	StringGetEx(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 1
	
	// 当 MODBUS 设备未响应，test = 2 将不会被执行，直到得到响应
	StringGet(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 2
	
	end macro_command

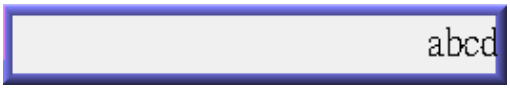
函数名称	StringSet
语法	StringSet (send_data[start], device_函数名称, device_type, address_offset, data_count)
描述	将数据写到 PLC 中。字符串数据保存在 send_data[start]~send_data[start+data_count-1] 中，send_data 必须为一维字符数组类型。 data_count 是写入到 PLC 中字符数据的个数，可以是常数也可以是变量。 device_函数名称详见上面的说明，在此不再说明。 device_type 是设备类型和 PLC 中数据的编码方式。例如：如果 device_type 是 LW_BIN，那么读取的设备类型为 LW，数据编码方式为 BIN。如果使用 BIN 编码方式，“_BIN”可以忽略。 如果 device_type 是 LW_BCD，表示设备类型 LW，数据的编码方式为 BCD 格式。 address_offset 是 PLC 中的地址偏移量。 例如，StringSet(read_data_1[0], “FATEK FB Series”, RT, 5, 1) 代表读取的设备地址偏移量为 5。 如果 address_offset 使用格式为 “N#AAAAA”，N 表示 PLC 的站号，AAAAA 表示地址偏移量。此情况一般使用在同一个串行埠上连接有多台 PLC 或者控制器的情况下。例如：StringSet(send_data_1[0], “FATEK FB Series”, RT, 2#5, 1) 表示设定站号为 2 的 PLC 的数据。如果 StringSet() 使用 “系统参数 / 设备列表” 中设定的默认的站号，在此可以不填这个站号。 设定到 PLC 的资料个数，根据 data_count 的值来决定，因 send_data 变量仅接受 char 数组类型。如下表所示：

sead_data 的类型	data_count 的值	设定 16 位数据的个数
char (8-bit)	1	1
char (8-bit)	2	1

因为一个 WORD (16 位) 等于 2 个 ASCII 字符的长度，当设备类型长度为 WORD 时，根据上表，写入 2 个 ASCII 字符实际上是写 1 个 WORD 的数据。宏指令会以先写 low byte 再写 hight byte 的顺序，依序将 ASCII 字符写入。
使用“字符显示”元件显示数据时，data_count 必须填入 2 的倍数才能正确显示。例如：

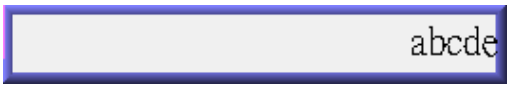
```
macro_command main()
char src1[10]="abcde"
StringSet(src1[0], "Local HMI", LW, 0, 5)
end macro_command
```

“字符显示”元件显示如下：



当 data_count 为一个大于或等于字符串长度的偶数时，可以完整显示整个字符串：

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



举例

```
macro_command main()
char str1[10]= “abcde”

// 字符串 str1 写入 LW-0 ~ LW-2 共三个 WORD
// 即使 data_count 为 10，写到第 3 个 WORD 时发现字符串已结束，
// 便不再写入后面的数据
StringSet(str1[0], "Local HMI", LW, 0, 10)

end macro_command
```

函数名称	StringSetEx
语法	StringSetEx(send_data[start], device_ 函数名称, device_type, address_offset,

	data_count)
描述	将数据写到 PLC 中，不等待 PLC 回应，径自往下执行。 send_data、device_函数名称、device_type、address_offset 和 data_count的说明和 StringSet 相同。

举例	macro_command main()
	char str1[20]= “abcde”
	short test=0
	
	// 当 MODBUS 设备未回应，test = 1 将照常执行
	StringSetEx(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 1
	
	// 当 MODBUS 设备未响应，test = 2 将不会被执行，直到得到响应
	StringSet(str1[0], "MODBUS RTU", 4x, 0, 20)
	test = 2
	
	end macro_command

函数名称	StringCopy
语法	success = StringCopy ([source], destination[start]) 或 success = StringCopy (source[start], destination[start])
描述	利用此函数进行字符串的复制。此函数可以将传入的静态字符串 (以双引号”” 括起来的字符串)，或是存放在字符数组中的字符串复制到目的 buffer。 来源字符串 sourc e可以为静态字符串 (如 “source”)或是一维字符数组变量 (如source[start])。 Destination[start] 必须为一维字符数组变量。 复制完毕会回传一 bool 类型的值给 success 字段。当复制成功，success等于 true，否则等于 false。当来源字符串的长度大于目的 buffer 的大小时，将不做任何处理，并回传 false 到 success 字段。 success 字段可写可不写。

举例	macro_command main()
	char src1[5] = "abcde"
	char dest1[5]
	bool success1
	success1 = StringCopy(src1[0], dest1[0])
	// success1=true，dest1为 “abcde”
	char dest2[5]
	bool success2
	success2 = StringCopy("12345", dest2[0])
	// success2 = true，dest2为 “12345”
	char src3[10] = "abcdefghij"
	char dest3[5]
	bool success3
	success3 = StringCopy(src3[0], dest3[0])
	// success3 = false，dest3内容不变

	char src4[10] = "abcdefghij" char dest4[5] bool success4 success4 = StringCopy(src4[5], dest4[0]) // success4=true，dest4为 “fghij” end macro_command

函数名称	StringDecAsc2Bin
语法	success = StringDecAsc2Bin(source[start], destination) 或 success = StringDecAsc2Bin(“source” , destination)
描述	此函数将十进制字符串转换成整数。 来源字符串 source 可以为静态字符串 (如 “source”)或是一维字符数组变量(如source[start])。 destination 必须为一变量，用以存放转换后的整数值。 执行完毕会回传一 bool 类型的值给 success 字段。当转换成功，success等于true，否则等于 false。 来源字符串必需为十进制字符串，若其包含 ‘0’ ~ ‘9’ 以外的字符，函数将会回传false。 success 字段可写可不写。

举例	macro_command main()
	char src1[5] = "12345"
	int result1
	bool success1
	success1 = StringDecAsc2Bin(src1[0], result1)
	// success1 = true，result1 为 “12345”
	
	char result2
	bool success2
	success2 = StringDecAsc2Bin("32768", result2)
	// success2 = true，但结果超出 result2 所能表达的范围
	
	char src3[2] = "4b"
	char result3
	bool success3
	success3 = StringDecAsc2Bin (src3[0], result3)
	// success3 = false，因 src3 包含 ‘0’ ~ ‘9’ 以外的字符
	
	end macro_command

函数名称	StringBin2DecAsc
语法	success = StringBin2DecAsc (source, destination[start])
描述	此函数将整数转换成十进制字符串。 来源 source 可以为常数或变量。 destination” start” 必须为一维字符数组变量，用以存放转换后的十进制字符串。 执行完毕会回传一 bool 类型的值给 success 字段。当转换成功，success等于 true，否则等于 false。 若转换后的十进制字符串长度大于目标数组的大小，函数将会回传 false。 success 字段可写可不写。

举例	macro_command main()
	int src1 = 2147483647
	char dest1[20]
	bool success1
	success1 = StringBin2DecAsc(src1, dest1[0])
	// success1 = true，dest1为 “2147483647”
	
	short src2 = 0x3c
	char dest2[20]
	bool success2
	success2 = StringBin2DecAsc(src2, dest2[0])
	// success2 = true，dest2为 “60”
	
	int src3 = 2147483647
	char dest3[5]
	bool success3
	success3 = StringBin2DecAsc(src3, dest3[0])
	// success3 = false，dest3 内容不变
	
	end macro_command

函数名称	StringDecAsc2Float
语法	success = StringDecAsc2Float (source[start], destination) 或 success = StringDecAsc2Float (“source” , destination)
描述	此函数将十进制字符串转换成浮点数。 来源字符串 source 可以为静态字符串 (如 “source”)或是一维字符数组变量 (如 source[start])。 destination 必须为一变量，用以存放转换后的浮点数值。 执行完毕会回传一 bool 类型的值给 success 字段。当转换成功，success等于 true，否则等于 false。 来源字符串必需为十进制字符串，若其包含 ‘0’ ~ ‘9’ 或 ‘.’ 以外的字符，函数将会回传 false。 success 字段可写可不写。

举例	macro_command main()
	char src1[10] = "12.345"
	float result1
	bool success1
	success1 = StringDecAsc2Float(src1[0], result1)
	// success1 = true，result1 为 “12.345”
	
	float result2
	bool success2
	success2 = StringDecAsc2Float("1.234567890", result2)
	// success2 = true，但结果超出 result2 所能表达的范围，可能丧失精确度
	
	char src3[2] = "4b"
	float result3
	bool success3
	success3 = StringDecAsc2Float(src3[0], result3)
	// success3 = false，因 src3 包含 ‘0’ ~ ‘9’ 或 ‘.’ 以外的字符
	
	end macro_command

函数名称	StringFloat2DecAsc
语法	success = StringFloat2DecAsc(source, destination[start])
描述	此函数将浮点数转换成十进制字符串。 来源 source 可以为常数或变量。 Destination[start] 必须为一维字符数组变量，用以存放转换后的十进制字符串。 执行完毕会回传一 bool 类型的值给 success 字段。当转换成功，success等于 true，否则等于 false。 若转换后的十进制字符串长度大于目标数组的大小，函数将会回传 false。 success 字段可写可不写。

举例	macro_command main()
	float src1 = 1.2345
	char dest1[20]
	bool success1
	success1 = StringFloat2DecAsc(src1, dest1[0])
	// success1 = true，dest1为 “1.2345”
	
	float src2 = 1.23456789
	char dest2 [20]
	bool success2
	success2 = StringFloat2DecAsc(src2, dest2 [0])
	// success2 = true，但可能丧失精确度
	
	float src3 = 1.2345
	char dest3[5]
	bool success3
	success3 = StringFloat2DecAsc(src3, dest3 [0])
	// success3 = false，dest3 内容不变
	
	end macro_command

函数名称	StringHexAsc2Bin
语法	success = StringHexAsc2Bin (source[start], destination) 或 success = StringHexAsc2Bin (“source” , destination)
描述	此函数将十六进制字符串转换成整数。 来源字符串 source 可以为静态字符串 (如 “source”)或是一维字符数组变量 (如 source” start”)。 destination 必须为一变量，用以存放转换后的整数值。 执行完毕会回传一 bool 类型的值给 success 字段。当转换成功，success等于 true，否则等于 false。 来源字符串必需为十六进制字符串，若其包含 ‘0’ ~ ‘9’ 或 ‘a’ ~ ‘f’ 或 ‘A’ ~ ‘F’ 以外的字符，函数将会回传 false。 success 字段可写可不写。

举例	macro_command main()
	char src1[5]="0x3c"
	int result1
	bool success1
	success1 = StringHexAsc2Bin(src1[0], result1)
	// success1 = true，result1 为 3c
	
	short result2
	bool success2
	success2 = StringDecAsc2Bin("1a2b3c4d", result2)
	// success2 = true，但结果超出 result2 所能表达的范围，result2 = 3c4d
	
	char src3[2] = "4g"
	char result3
	bool success3
	success3 = StringDecAsc2Bin (src3[0], result3)
	// success3=false，因 src3包含 ‘0’ ~ ‘9’ 或 ‘a’ ~ ‘f’ 或 ‘A’ ~ ‘F’ 以外的字符
	
	end macro_command

函数名称	StringBin2HexAsc
语法	success = StringBin2HexAsc (source, destination[start])
描述	此函数将整数转换成十六进制字符串。 来源source可以为常数或变量。 Destination[start] 必须为一维字符数组变量，用以存放转换后的十六进制字符串。 执行完毕会回传一 bool 类型的值给 success 字段。当转换成功，success等于true，否则等于 false。 若转换后的十六进制字符串长度大于目标数组的大小，函数将会回传 false。 success 字段可写可不写。

举例	macro_command main()
	int src1 = 20
	char dest1[20]
	bool success1
	success1 = StringBin2HexAsc(src1, dest1[0])
	// success1 = true，dest1 为 “14”
	
	short src2 = 0x3c
	char dest2[20]
	bool success2
	success2 = StringBin2HexAsc(src2, dest2[0])
	// success2 = true，dest2 为 “3c”
	
	int src3 = 0x1a2b3c4d
	char dest3[6]
	bool success3
	success3 = StringBin2HexAsc(src3, dest3[0])
	// success3 = false，dest3 内容不变
	
	end macro_command

函数名称	StringMid
语法	success = StringMid (source[start], count, destination[start]) 或 success = StringMid (“string” , start, count, destination[start])
描述	利用此函数可将一个字符串中的某一段子字符串提取出来。 来源字符串 source 可以为静态字符串 (如 “source”)或是一维字符数组变量(如 source[start])。当来源字符串为字符数组变量时，由数组下标决定子字符串起始位置。当来源字符串为静态字符串时，由第二个参数 start 决定子字符串起始位置。 count 决定要提取的子字符串长度。 Destination[start] 必须为一维字符数组变量，用以存放提取出来的子字符串。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于true，否则等于 false。 若提取出来的子字符串长度大于目标数组的大小，函数将会回传false。 success 字段可写可不写。

举例	macro_command main()
	char src1[20] = "abcdefghijklmnopqrst"
	char dest1[20]
	bool success1
	success1 = StringMid(src1[5], 6, dest1[0])
	// success1 = true，dest1 为 “fghijk”
	
	char src2[20] = "abcdefghijklmnopqrst"
	char dest2[5]
	bool success2
	success2 = StringMid(src2[5], 6, dest2[0])
	// success2 = false，dest2 内容不变
	
	char dest3[20] = "12345678901234567890"
	bool success3
	success3 = StringMid("abcdefghijklmnopqrst", 5, 5, dest3[15])
	// success3 = true，dest3 = “123456789012345fghij”
	
	end macro_command

函数名称	StringLength
语法	length = StringLength (source[start]) 或 length = StringLength ([source])
描述	取得字符串的长度。 来源字符串 source 可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 函数回传值代表来源字符串的长度。

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

函数名称	StringCat
语法	success = StringCat (source[start], destination[start]) 或 success = StringCat (“source” , destination[start])
描述	利用此函数将来源字符串衔接于目标字符串之后。此函数执行成功后，目标字符串将等于两字符串衔接后的结果。 来源字符串source可以为静态字符串 (如 “source”) 或是一维字符数组变量(如source[start])。 Destination[start]必须为一维字符数组变量。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于true，否则等于 false。当两字符串衔接后的长度超过目标数组的长度时，目标字符串将保留衔接以前的内容，不做任何改变，并回传 false。

举例	macro_command main()
	char src1[20] = "abcdefghij"
	char dest1[20] = "1234567890"
	bool success1
	success1= StringCat(src1[0], dest1[0])
	// success1 = true，dest1 = "123456790abcdefghij"
	
	char dest2 [10] = "1234567890"
	bool success2
	success2= StringCat("abcde", dest2 “0”)
	// success2 = false，dest2 内容不变
	
	char src3[20] = "abcdefghij"
	char dest3[20]
	bool success3
	success3 = StringCat(src3[0], dest3[15])
	// success3 = false，dest3内容不变
	
	end macro_command

函数名称	StringCompare
语法	ret = StringCompare (str1[start], str2[start]) ret = StringCompare (“string1”, str2[start]) ret = StringCompare (str1[start], “string2”) ret = StringCompare (“string1”, “string2”)
描述	比较两字符串的内容是否相等。此函数将大小写视为不同。 两个传入的字符串皆可以为静态字符串 (如 “source”)或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool类型的值给 ret 字段。若两字符串相等，ret 为 true，否则为 false。

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

函数名称	StringCompareNoCase
语法	ret = StringCompareNoCase(str1[start], str2[start]) ret = StringCompareNoCase(“string1”, str2[start]) ret = StringCompareNoCase(str1[start], “string2”) ret = StringCompareNoCase(“string1”, “string2”)
描述	比较两字符串的内容是否相等。此函数将大小写视为相同。 两个传入的字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 ret 字段。若两字符串相等，ret 为 true，否则为 false。

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

函数名称	StringFind
语法	position = StringFind (source[start], target[start]) position = StringFind (“source”, target[start]) position = StringFind (source[start], “target”) position = StringFind (“source”, “target”)
描述	寻找某一字符串 (target) 在另一个字符串 (source) 里第一次出现的位置。 两个传入的字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传target字符串在 source 字符串里出现的位置。Source 字符串由 0 开始递增为字符编索引值。若 source 字符串存在一个子字符串，其所包含的字符与排列顺序跟 target 字符串完全相等，则函数会回传此子字符串开头的索引值，若没有找到，则回传 -1。

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

函数名称	StringReverseFind
语法	position = StringReverseFind (source[start], target[start]) position = StringReverseFind (“source” , target[start]) position = StringReverseFind (source[start], “target”) position = StringReverseFind (“source” , “target “)
描述	寻找某一字符串 (target) 在另一个字符串 (source) 里最后一次出现的位置。 两个传入的字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传 target 字符串在 source 字符串里最后一次出现的位置。source 字符串由 0 开始递增为字符编索引值。若 source 字符串存在一个子字符串，其所包含的字符与排列顺序跟 target 字符串完全相等，则函数会回传此子字符串开头的索引值，若没有找到，则回传 -1。若 source 字符串中存在多个与 target字符串相等的子字符串，则回传最后一个出现的子字符串的位置。

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

函数名称	StringFindOneOf
语法	position = StringFindOneOf (source[start], target[start]) position = StringFindOneOf (“source”, target[start]) position = StringFindOneOf (source[start], “target”) position = StringFindOneOf (“source”, “target”)
描述	寻找 target 字符串中任一个字符在 source 字符串中第一次出现的位置。 两个传入的字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传 target 字符串中任一个字符在 source 字符串里第一次出现的位置，即 source 字符串中为该字符编的索引值 (由 0 开始)。若没有找到，则回传 -1。

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

函数名称	StringIncluding
语法	success = StringIncluding (source[start], set[start], destination[start]) success = StringIncluding (“source”, set[start], destination[start]) success = StringIncluding (source[start], “set”, destination[start]) success = StringIncluding (“source”, “set”, destination[start])
描述	提取 source 字符串中某个以索引值 0 的字符 (第一个字符) 开头的子字符串，而且此子字符串的每个字符都能在 set 字符串中找到相同的字符。此函数将会从 source 字符串的第一个字符开始寻找，直到找到不存在于 set 字符串中的字符为止。 source 字符串与 set 字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于 true，否则等于 false。当提取出来的字符串长度大于目标数组的大小，将会回传 false。

举例	macro_command main()
	char src1[20] = "cabbageabc"
	char set1[20] = "abc"
	char dest1[20]
	bool success1
	success1 = SDtringIncluding(src1[0], set1[0], dest1[0])
	// success1 = true，dest1 = “cabba”
	
	char src2[20] = "gecabba"
	char dest2[20]
	bool success2
	success2 = StringIncluding(src2[0], "abc", dest2[0])
	// success2 = true，dest2 = “ ”
	
	char set3[20] = "abc"
	char dest3[4]
	bool success3
	success3 = StringIncluding("cabbage", set3[0], dest3[0])
	// success3 = false，dest3内容不变
	
	end macro_command

函数名称	StringExcluding
语法	success = StringExcluding (source[start], set[start], destination[start]) success = StringExcluding (“source”, set[start], destination[start]) success = StringExcluding (source[start], “set”, destination[start]) success = StringExcluding (“source”, “set”, destination[start])
描述	提取 source 字符串中某个以索引值 0 的字符 (第一个字符) 开头的子字符串，而且此子字符串的每个字符必不存在于set字符串中。此函数将会从 source字符串的第一个字符开始寻找，直到找到存在于 set 字符串中的字符为止。 source 字符串与 set 字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于true，否则等于 false。当提取出来的字符串长度大于目标数组的大小，将会回传false。

举例	macro_command main()
	char src1[20] = "cabbageabc"
	char set1[20] = "ge"
	char dest1[20]
	bool success1
	success1 = StringExcluding(src1[0], set1[0], dest1[0])
	// success1 = true， dest1 = "cabba"
	
	char src2[20] = "cabbage"
	char dest2[20]
	bool success2
	success2 = StringExcluding(src2[0], "abc", dest2[0])
	// success2 = true， dest2 =" "
	
	char set3[20] = "ge"
	char dest3[4]
	bool success3
	success3 = StringExcluding("cabbage", set3[0], dest3[0])
	// success3 = false， dest3 内容不变
	
	end macro_command

函数名称	StringToUpper
语法	success = StringToUpper (source[start], destination[start]) success = StringToUpper (“source” , destination[start])
描述	将 source 字符串的字符全部转换成大写，并写入 destination 字符串。 source 字符串可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于 true，否则等于 false。source 字符串长度大于目标数组的大小，将会回传 false。

举例	macro_command main()
	char src1[20] = "aBcDe"
	char dest1[20]
	bool success1
	success1 = StringToUpper(src1[0], dest1[0])
	// success1 = true，dest1 = “ABCDE”
	
	char dest2[4]
	bool success2
	success2 = StringToUpper("aBcDe", dest2[0])
	// success2 = false，dest2 内容不变
	
	end macro_command

函数名称	StringToLower
语法	success = StringToLower (source[start], destination[start]) success = StringToLower (“source” , destination[start])
描述	将 source 字符串的字符全部转换成小写，并写入 destination 字符串。 source 字符串可以为静态字符串 (如 “source”)或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于 true，否则等于 false。source 字符串长度大于目标数组的大小，将会回传 false。

举例	macro_command main()
	char src1[20] = "aBcDe"
	char dest1[20]
	bool success1
	success1 = StringToUpper(src1[0], dest1[0])
	// success1 = true，dest1 = “abcde”
	
	char dest2[4]
	bool success2
	success2 = StringToUpper("aBcDe", dest2[0])
	// success2 = false，dest2 内容不变
	
	end macro_command

函数名称	StringToReverse
语法	success = StringToReverse (source[start], destination[start]) success = StringToReverse (“source” , destination[start])
描述	将 source 字符串反转，并写入 destination 字符串。 source 字符串可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于 true，否则等于 false。source 字符串长度大于目标数组的大小，将会回传 false。

举例	macro_command main()
	char src1[20] = "abcde"
	char dest1[20]
	bool success1
	success1 = StringToUpper(src1[0], dest1[0])
	// success1 = true，dest1 = “edcba”
	
	char dest2[4]
	bool success2
	success2 = StringToUpper("abcde", dest2[0])
	// success2 = false，dest2 内容不变
	
	end macro_command

函数名称	StringTrimLeft
语法	success = StringTrimLeft (source[start], set[start], destination[start]) success = StringTrimLeft (“source”, set[start], destination[start]) success = StringTrimLeft (source[start], “set”, destination[start]) success = StringTrimLeft (“source”, “set”, destination[start])
描述	从 source 字符串的第一个字符开始往后寻找，若找到与 set 字符串相同的字符便裁剪掉该字符，直到遇到不存在于 set 字符串中的字符为止。 source 字符串与 set 字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于 true，否则等于 false。当裁剪完后的字符串长度大于目标数组的大小，将会回传 false。

举例	macro_command main()
	char src1[20] = "# *a*#bc"
	char set1[20] = "# *"
	char dest1[20]
	bool success1
	success1 = StringTrimLeft (src1[0], set1[0], dest1[0])
	// success1 = true， dest1 = "a*#bc"
	
	char set2[20] = {'#', ' ', '*'}
	char dest2[4]
	success2 = StringTrimLeft ("# *a*#bc", set2[0], dest2[0])
	// success2 = false， dest2 内容不变
	
	char src3[20] = "abc *#"
	char dest3[20]
	bool success3
	success3 = StringTrimLeft (src3[0], "# *", dest3[0])
	// success3 = true， dest3 = "abc *#"
	
	end macro_command

函数名称	StringTrimRight
语法	success = StringTrimRight (source[start], set[start], destination[start]) success = StringTrimRight (“source”, set[start], destination[start]) success = StringTrimRight (source[start], “set”, destination[start]) success = StringTrimRight (“source”, “set”, destination[start])
描述	从 source 字符串的最后一个字符开始往前寻找，若找到与 set 字符串相同的字符便裁剪掉该字符，直到遇到不存在于 set 字符串中的字符为止。 source 字符串与 set 字符串皆可以为静态字符串 (如 “source”) 或是一维字符数组变量 (如 source[start])。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于true，否则等于 false。当裁剪完后的字符串长度大于目标数组的大小，将会回传false。

举例	macro_command main()
	char src1[20] = "# *a*#bc# * "
	char set1[20] = "# *"
	char dest1[20]
	bool success1
	success1 = StringTrimRight(src1[0], set1[0], dest1[0])
	// success1 = true，dest1= “# *a*#bc”
	
	char set2[20] = {'#', ' ', '*'}
	char dest2[20]
	success2 = StringTrimRight("# *a*#bc", set2[0], dest2[0])
	// success2 = true，dest2 = “# *a*#bc”
	
	char src3[20] = "ab**c *#"
	char dest3[4]
	bool success3
	success3 = StringTrimRight(src3[0], "# *", dest3[0])
	// success3 = false，dest3 内容不变
	
	end macro_command

函数名称	StringInsert
语法	success = StringInsert (pos, insert[start], destination[start]) success = StringInsert (pos, “insert”, destination[start]) success = StringInsert (pos, insert[start], length, destination[start]) success = StringInsert (pos, “insert”, length, destination[start])
描述	将 insert 字符串插入到目标字符串中的特定位置，插入位置由 pos 所指定。 Insert 字符串可以为静态字符串 (如 “insert”) 或是一维字符数组变量 (如 insert[start])。 用户亦可以在 length 字段指定 insert 字符串的长度。 执行完毕会回传一 bool 类型的值给 success 字段。当执行成功，success等于 true，否则等于 false。当插入完成后的字符串长度大于目标数组的大小，将会回传 false。

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

18.7.7. 配方数据函数

函数名称	RecipeGetData
语法	RecipeGetData (destination, recipe_address, record_ID)
描述	取得配方中的资料。所取得的数据存放在destination且必须为变量。recipe_address由配方名称与项目名称组成："recipe_name.item_name"。record_ID指定欲取得配方中的第几笔数据，也就是数据列编号。

举例	macro_command main()
	int data = 0
	char str[20]
	int recordID
	bool result
	
	recordID = 0
	result = RecipeGetData(data, "TypeA.item_weight", recordID)
	// 从配方 "TypeA" 中取得第0笔，且item name为 "item_weight" 的资料
	
	recordID = 1
	result = RecipeGetData(str[0], "TypeB.item_name", recordID)
	// 从配方 "TypeB" 中取得第1笔，且 item name为 "item_name" 的资料
	
	end macro_command

函数名称	RecipeQuery
语法	RecipeQuery (SQL command, destination)
描述	通过SQL语句，对配方中的数据进行查询。查询结果的数据笔数存放在destination，必须为变量。SQL command 的部份可为静态文字或传入字符数组，例如： RecipeQuery(“SELECT * FROM TypeA” , destination) 或 RecipeQuery(sql[0], destination) SQL 语句必须以 “SELECT * FROM” 开头，后面接配方名称及查询条件。

举例	macro_command main()
	
	int total_row = 0
	char sql[100] = "SELECT * FROM TypeB"
	bool result
	
	result = RecipeQuery("SELECT * FROM TypeA", total_row)
	// 查询配方 "TypeA"。查询结果的数据笔数存放在 total_row.
	
	result = RecipeQuery(sql[0], total_row)
	// 查询配方 "TypeB"。查询结果的数据笔数存放在 total_row.
	
	end macro_command

函数名称	RecipeQueryGetData
语法	RecipeQueryGetData (destination, recipe_address, result_row_no)
描述	取得 RecipeQuery 的查询结果。呼叫此函数前必须先呼叫 RecipeQuery，且 recipe_address 中需指定与 RecipeQuery 相同的配方名称。 result_row_no 指定欲取得查询结果中的第几笔数据。

举例	macro_command main()
	
	int data = 0
	int total_row = 0
	int row_number = 0
	bool result_query
	bool result_data
	
	result_query = RecipeQuery("SELECT * FROM TypeA", total_row)
	// 查询配方 "TypeA"。查询结果的数据笔数存放在 total_row.
	
	if (result_query) then
	for row_number = 0 to total_row - 1
	result_data = RecipeQueryGetData(data, "TypeA.item_weight", row_number)
	next row_number
	end if
	
	end macro_command

函数名称	RecipeQueryGetRecordID
语法	RecipeQueryGetRecordID (destination, result_row_no)
描述	取得 RecipeQuery 查询结果的数据列编号。呼叫此函数前必须先呼叫 RecipeQuery。 result_row_no 指定欲取得查询结果中的第几笔数据的数据列编号，并将资料列编号写入 destination。

举例	macro_command main()
	
	int recorded = 0
	int total_row = 0
	int row_number = 0
	bool result_query
	bool result_id
	
	result_query = RecipeQuery("SELECT * FROM TypeA", total_row)
	// 查询配方 "TypeA"。查询结果的数据笔数存放在 total_row.
	
	if (result_query) then
	for row_number = 0 to total_row - 1
	result_id = RecipeQueryGetRecordID(recordID, row_number)
	next row_number

	end if end macro_command

函数名称	RecipeSetData
语法	RecipeSetData(*source*, *recipe_address*, *record_ID*)
描述	将数据写入配方数据库中。如果成功将传回 True，否则将传回 False。 recipe_address 由配方名称与项目名称组成："recipe_name.item_name"。 record_ID 指定欲修改配方中的第几笔数据，也就是数据列编号。

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

18.7.8. 其它函数

函数名称	Beep
语法	Beep ()
描述	发出系统警示音。 此函数可发出 800 赫兹，30 毫秒的系统警示音。
举例	macro_command main() Beep() end macro_command

函数名称	SYNC_TRIG_MACRO
语法	SYNC_TRIG_MACRO(macro_id)
描述	一个执行中的宏指令可以使用此函数利用同步的方式触发执行其它的宏指令，使用 macro_id 指定被触发的宏指令 “编号” 。

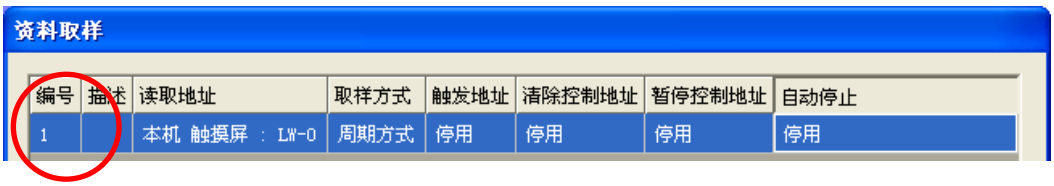
	使用此函数的宏指令将暂停执行，直到被触发的宏指令执行完成才会继续执行后续的命令。 macro_id 可以是常数或使用变量来表示。
举例	macro_command main() char ON = 1, OFF = 0 SetData(ON, “Local HMI”, LB, 0, 1) SYNC_TRIG_MACRO(5) // call a macro (its ID is 5) SetData(OFF, “Local HMI”, LB, 0, 1) end macro_command

函数名称	ASYNC_TRIG_MACRO
语法	ASYNC_TRIG_MACRO(macro_id)
描述	一个执行中的宏指令可以使用此函数，利用异步的方式触发执行其它宏指令，使用 macro_id 指定被触发的宏指令 “编号”。 宏指偷在使用此函数后将立即执行后续的命令，不需等待被触发的宏指令执行完成。 macro_id 可以是常数或使用变量来表示。
举例	macro_command main() char ON = 1, OFF = 0 SetData(ON, “Local HMI”, LB, 0, 1) ASYNC_TRIG_MACRO(5) // call a macro (its ID is 5) SetData(OFF, “Local HMI”, LB, 0, 1) end macro_command

函数名称	TRACE
语法	TRACE(format, argument)
描述	一个执行中的 宏指令可以使用此函数，监视变量内容的变化，并打印字符串，以协助除错。用户应开启 EasyDiagnoser 观看此函数的输出结果。 当 TRACE 函数抓取一个%开头的特殊字符，将同时从 argument 抓取一个参数做格式化后输出。 Format 代表打印格式，支持 % 开头的特殊字符。特殊字符格式如下，其中方括号内的字段为可选，粗体字字段为必需： %” flags” “width” “.precision” type 每个字段的意义如下所述： flags (可选)： -

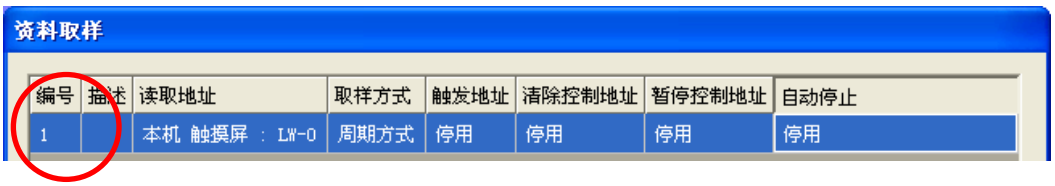
	+ width (可选)： 十进制正整数，指定应预留的字符宽度，不足部份补空格符。 precision (可选)： 十进制正整数，指定精确度，以及输出字符数。 type： C 或 c ： 以字符方式输出 d ： 以 signed 十进制整数输出 i ： 以 signed 十进制整数输出 o ： 以 unsigned 八进位整数输出 u ： 以 unsigned 十进制整数输出 X 或 x ： 以 unsigned 十六进制整数输出 E 或 e ： 以科学表示法输出。格式为” - “d.dddd e “sign” ddd。其中字段d是十进制数，字段dddd是一至多个十进制数，字段ddd必须是三个十进制数。sign是+或-。 f ： 以单倍精确度浮点数输出。格式为” - “dddd.dddd。 *Format* 字符串最长支持 256 个字符，多出的字符将被忽略。 *Argument* 部份可写可不写。但一个特殊字符应搭配一个变量。

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

函数名称	FindDataSamplingDate
语法	return_value = FindDataSamplingDate (data_log_number, index, year, month, day) or FindDataSamplingDate (data_log_number, index, year, month, day)
描述	利用此函数查询资料取样文件的日期。根据所输入的资料取样编号 (data_log_number) 与资料取样文件索引 (index) 可查询该资料取样的日期，依照年、月、日的顺序写入 year、month、day 的字段中。  资料取样文件的储存方式为：” 保存位置” \ “取样文件夹名称” \ yyyymmdd.dtl。而资料取样文件索引 (index) 指的是取样文件夹下面所有资料取样文件依文件名

	排序后的顺位值 (从0开始)。Index 值越小者，日期越新。例如假设某取样文件夹下面有四个资料取样文件： 20101210.dtl 20101230.dtl 20110110.dtl 20110111.dtl 则index依序为： 20101210.dtl -> index 为 3 20101230.dtl -> index 为 2 20110110.dtl -> index 为 1 20110111.dtl -> index 为 0 当成功搜寻到所欲查询的资料取样文件时，将回传 1 至 return_value，否则将回传 0。 data_log_number 与 index 可以为常数或是变量，但 year、month、day与return_value 必须为变量。 return_value 为可选。

举例	macro_command main()
	short data_log_number = 1, index = 2, year, month, day
	short success
	
	// 若存在一资料取样文件 20101230.dtl，其资料取样编号为 1，文件索引为 2
	// 则success == 1，year == 2010，month == 12，day == 30
	success = FindDataSamplingDate(data_log_number, index, year, month, day)
	
	end macro_command

函数名称	FindDataSamplingIndex
语法	return_value = FindDataSamplingIndex (data_log_number, year, month, day, index) or FindDataSamplingIndex (data_log_number, year, month, day, index)
描述	利用此函数查询资料取样文件的文件索引值。根据所输入的资料取样编号 (data_log_number) 与日期可查询该资料取样文件的文件索引，并将其写入 index。year、month、day 三个字段依序代表年、月、日，其输入格式为 YYYY (年)、MM (月)、DD (日)。  资料取样文件的储存方式为：”保存位置” \ “取样文件夹名称” \ yyyymmdd.dtl。而资料取样文件索引 (index) 指的是取样文件夹下面所有资料取样文件依文件名排序后的顺位值 (从 0 开始)。Index 值越小者，日期越新。例如假设某取样文件夹下面有四个资料取样文件： 20101210.dtl 20101230.dtl 20110110.dtl 20110111.dtl 则 index 依序为： 20101210.dtl -> index 为 3 20101230.dtl -> index 为 2 20110110.dtl -> index 为 1 20110111.dtl -> index 为 0 当成功搜寻到所欲查询的资料取样文件时，将回传 1 至 return_value，否则将回传 0。 data_log_number 与 year、month、day 可以为常数或是变量，但 index 与 return_value 必须为变量。 return_value 为可选。

举例	macro_command main()
	short data_log_number = 1, year = 2010, month = 12, day = 10, index
	short success
	
	// 若存在一资料取样文件 20101210.dtl，其资料取样编号为 1，文件索引为 2
	// 则success == 1，index == 2
	success = FindDataSamplingIndex (data_log_number, year, month, day, index)
	
	end macro_command

函数名称	FindEventLogDate
语法	return_value = FindEventLogDate (index, year, month, day) or FindEventLogDate (index, year, month, day)
描述	利用此函数查询事件登录文件的日期。根据所输入的事件登录文件索引 (index)可查询该事件登录的日期，依照年、月、日的顺序写入 year、month、day的字段中。事件登录文件索引 (index) 指的是指定的保存位置 (HMI、SD 卡或 USB)下事件登录文件依档名排序后的顺位值 (从 0 开始)。Index 值越小者，日期越新。例如假设有四个事件登录档： EL_20101210.evt EL_20101230.evt EL_20110110.evt EL_20110111.evt 则index依序为： EL_20101210.evt -> index为3 EL_20101230.evt -> index为2 EL_20110110.evt -> index为1 EL_20110111.dtl -> index为0 当成功搜寻到所欲查询的事件登录档时，将回传 1 至return_value，否则将回传 0。 Index 可以为常数或是变量，但 year、month、day与return_value 必须为变量。return_value 为可选。

举例	macro_command main()
	short index = 1, year, month, day
	short success
	
	// 若存在一事件登录档 EL_20101230.evt，文件索引为 1
	// 则 success == 1，year == 2010，month == 12，day == 30
	success = FindEventLogDate (index, year, month, day)
	
	end macro_command

函数名称	FindEventLogIndex
语法	return_value = FindEventLogIndex (year, month, day, index) or FindEventLogIndex (year, month, day, index)
描述	利用此函数查询事件登录文件的文件索引值。根据所输入的日期可查询该事件登录文件的文件索引，并将其写入 index。year、month、day 三个字段依序代表年、月、日，其输入格式为 YYYY (年)、MM (月)、DD (日)。 事件登录文件索引 (index) 指的是指定的保存位置 (HMI、SD 卡或 USB)下事件登录文件依档名排序后的顺位值 (从 0 开始)。Index 值越小者，日期越新。例如假设有四个事件登录档： EL_20101210.evt EL_20101230.evt EL_20110110.evt EL_20110111.evt 则 index 依序为： EL_20101210.evt -> index 为 3 EL_20101230.evt -> index 为 2 EL_20110110.evt -> index 为 1 EL_20110111.dtl -> index 为 0 当成功搜寻到所欲查询的事件登录档时，将回传 1 至 return_value，否则将回传 0。 year、month、day 可以为常数或是变量，但 index 与 return_value 必须为变量。return_value 为可选。

举例	macro_command main()
	short year = 2010, month = 12, day = 10, index
	short success
	
	// 若存在一事件登录档 EL_20101210.evt，文件索引为 2
	// 则success == 1，index == 2
	success = FindEventLogIndex (year, month, day, index)
	
	end macro_command

18.8. 怎样建立和执行宏指令

18.8.1. 怎样建立一个宏指令

按照以下步骤以建立一个宏指令。

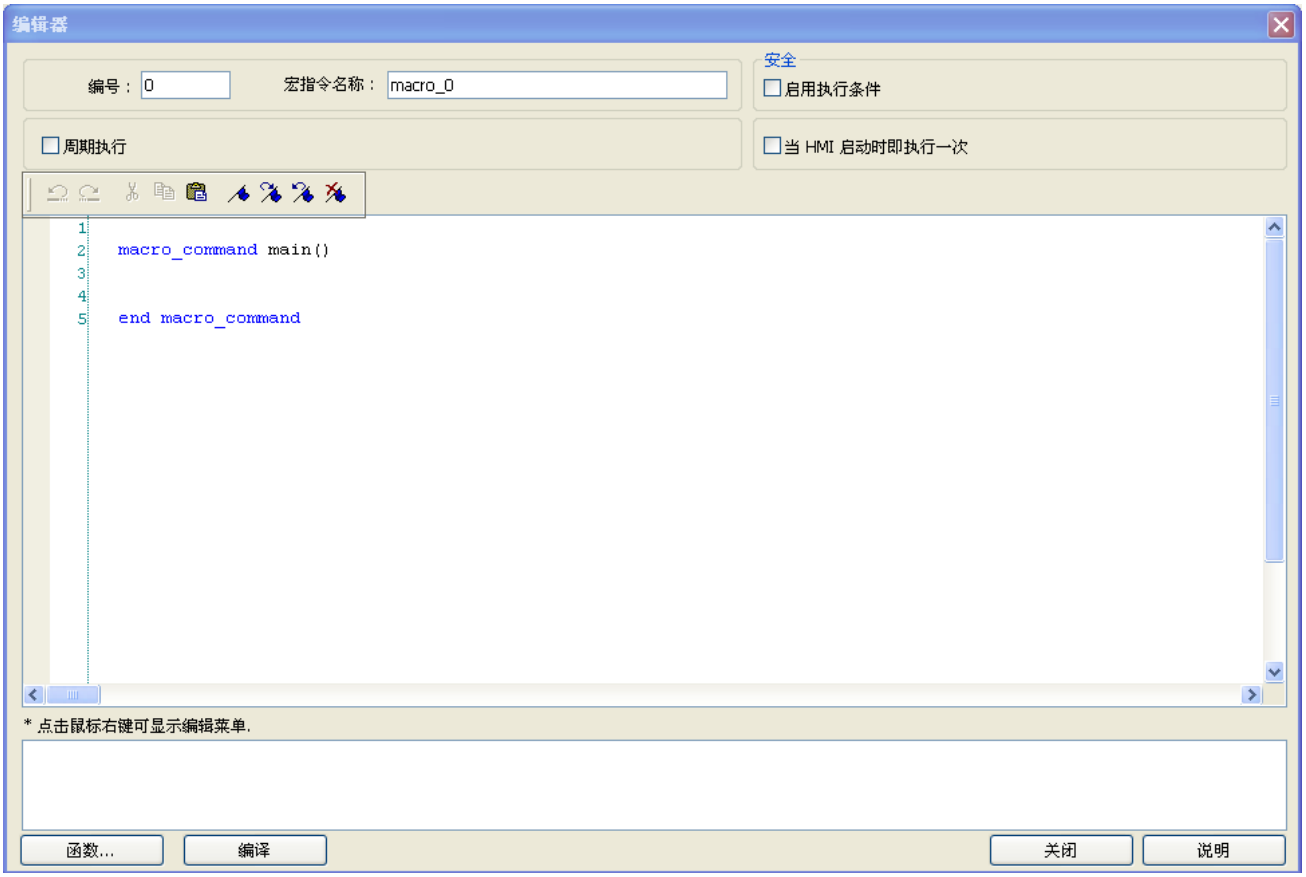
1. 单击 EasyBuilder Pro 软件工具栏上的宏指令图示 打开宏指令管理对话框。

在宏指令管理对话框中，已经编译成功的宏指令会出现在“已编译成功”列表上，未完成编译的会出现在“未完成编译”列表中。下面是宏指令管理对话框中各按键的功能描述。

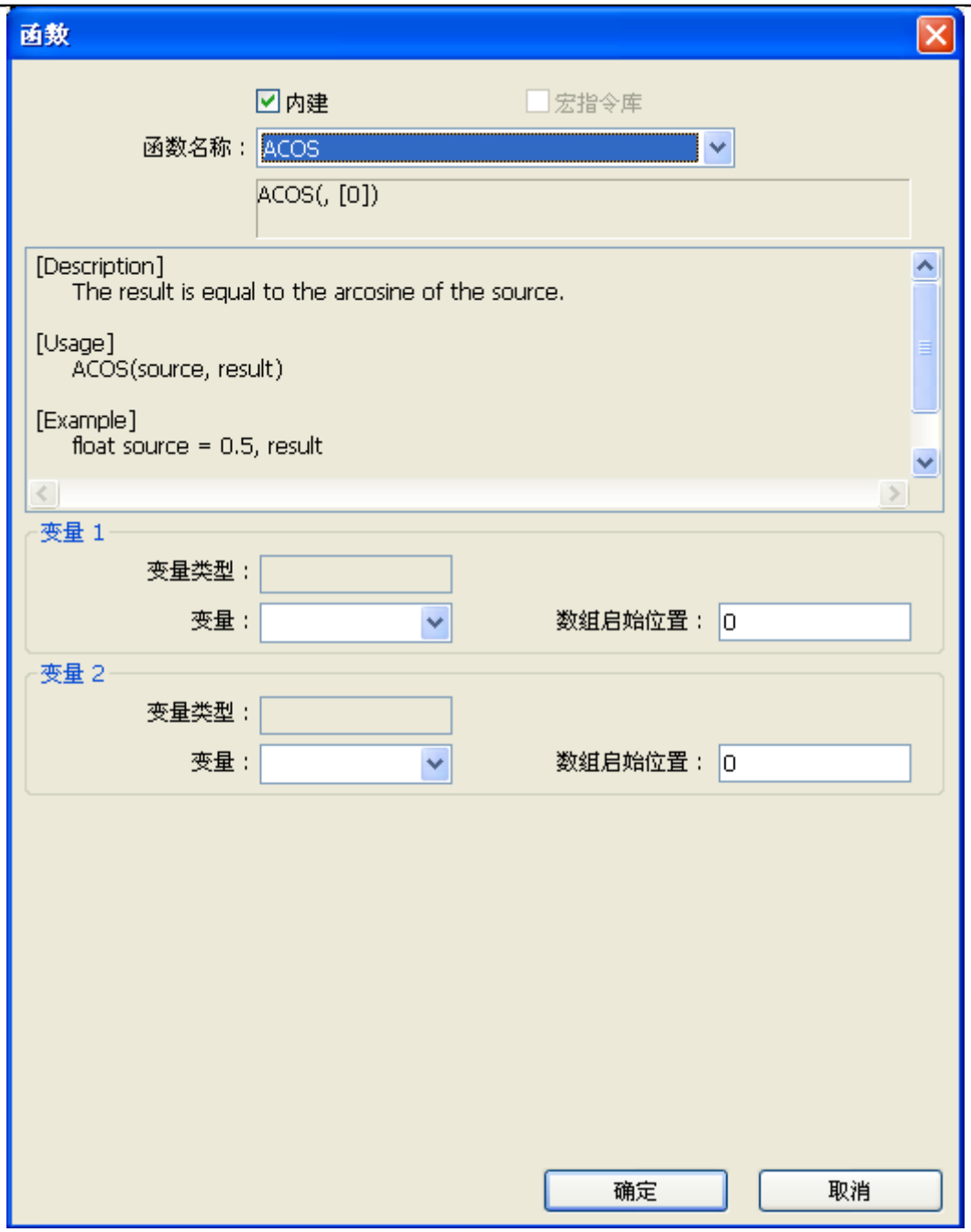
设定	描述
新增	新增一个宏指令，并开启新建的宏指令编辑区。
删除	删除选择的宏指令。
编辑	打开宏指令编辑区，并开启选择的宏指令。
复制	复制选择的宏指令。
贴上	将刚刚选择需要复制的宏指令，贴至“已编译完成”列表区，并产生一个

	新的宏指令名称。
确认	确认此次所编辑的所有宏指令后离开此宏指令管理对话框，必须按此键才会储存　此次编辑内容。
取消	取消此次所编辑的所有宏指令，离开宏指令管理对话框。
宏指令库	进入宏函数库管理对话框。

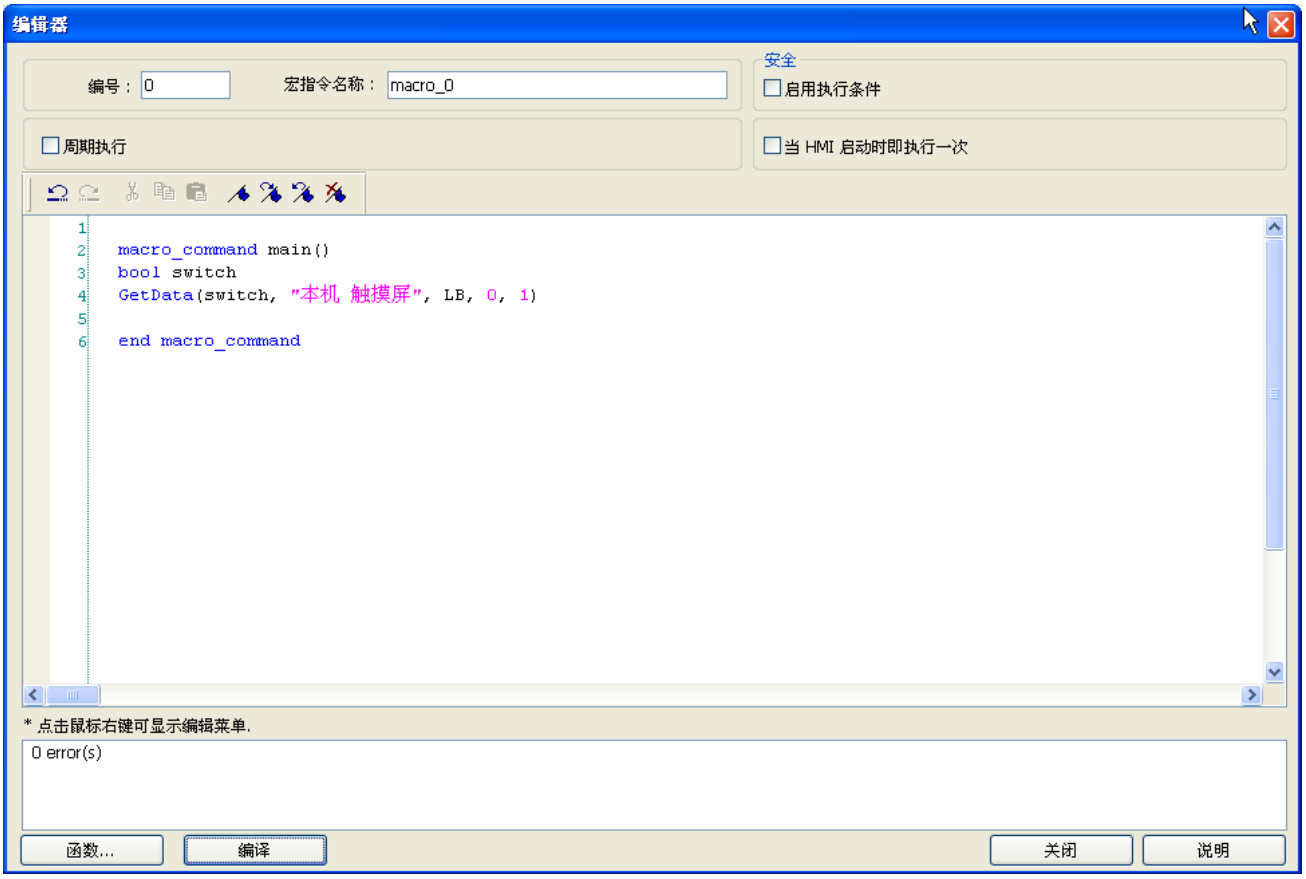
2. 按下“新增”按钮，打开一个新增的宏指令编辑区。每一个宏指令都有一个唯一的编号，定义在“编号”这个位置。在“宏指令名称”这个栏目中也必须输入宏指令的名称，否则编译将无法通过。



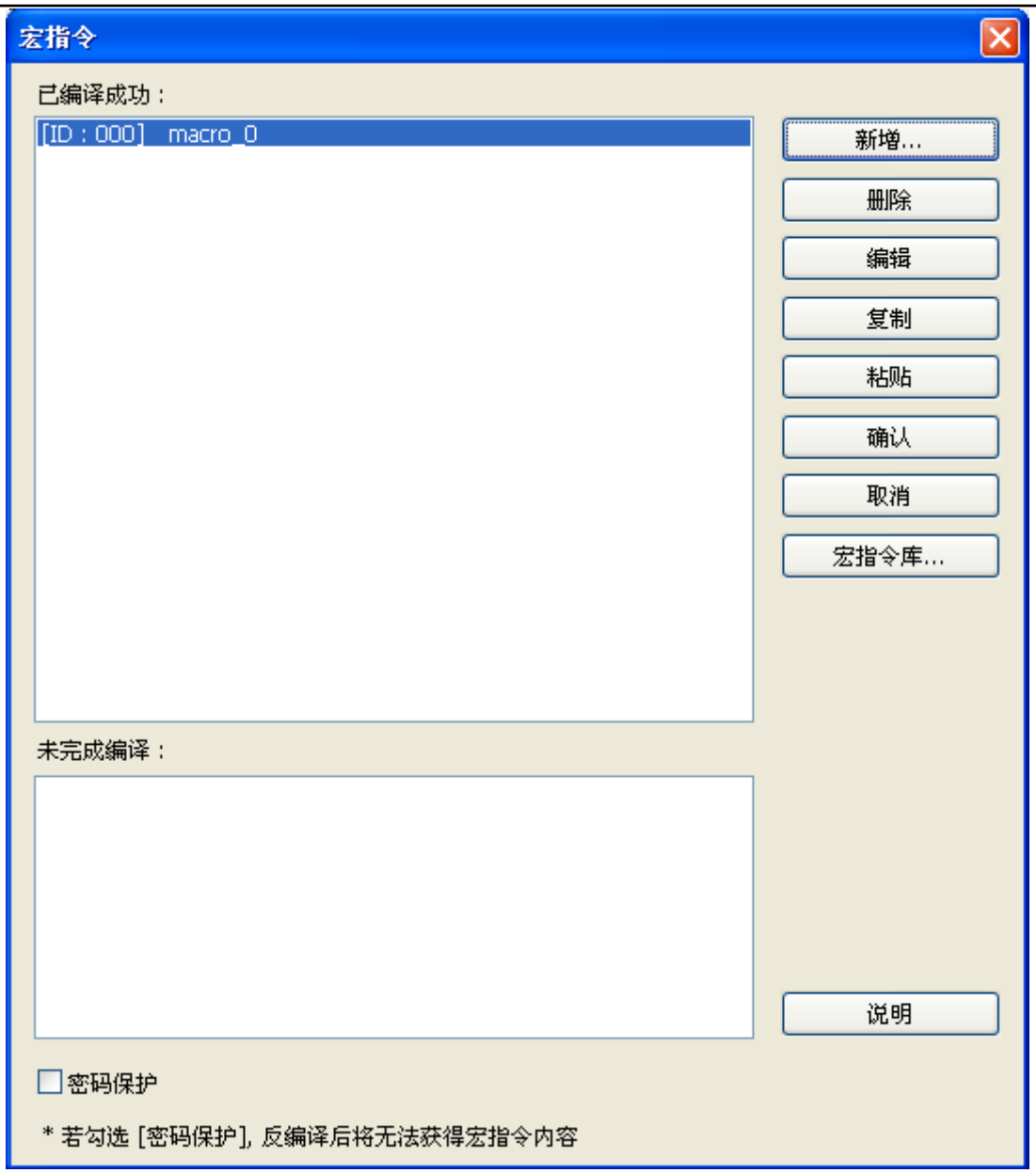
3. 设计属于您的宏指令程序。如果要使用内建的函数，譬如 Setdata() 或者 Getdata() 等函数，单击“函数”按键开启函数列表对话框，选择需要的函数，并设定必要的参数。



4. 编辑完成一个新建的宏指令程序后，单击“编译”按键，对该宏指令进行编译工作。



5. 如果没有错误，单击“关闭”按键，这样在“已编译成功”区会发现新增了一个“test”这个名称的宏指令。


宏指令
已编译成功：
[ID : 000] macro_0
新增...
删除
编辑
复制
粘贴
确认
取消
宏指令库...
未完成编译：
说明
密码保护
* 若勾选 [密码保护], 反编译后将无法获得宏指令内容

18.8.2. 执行宏指令

执行宏指令有多种不同的方法，下面分别说明。

- 使用“PLC 控制”元件

1. 打开“PLC 控制”元件，并设定属性为“执行宏指令”。
2. 选择需要执行的宏指令名称。选择一个位作为触发宏指令并设定触发宏指令的条件。在条件满足时，该宏指令将会被重复执行。为了每次只让宏指令执行一次，设计时需在宏指令将该触发位复位。
3. 使用一个“位状态设置” 元件或者“位状态切换开关”元件作为这个位的控制开关。

- 使用“位状态设置” 元件或者“位状态切换开关”元件

1. 在“位状态设置” 元件或者“位状态切换开关”元件的一般属性页中，勾选“使用宏指令”。
2. 选择要执行的宏指令。当这个元件被执行时，选择的宏指令就会被执行一次。

- 使用“功能键”

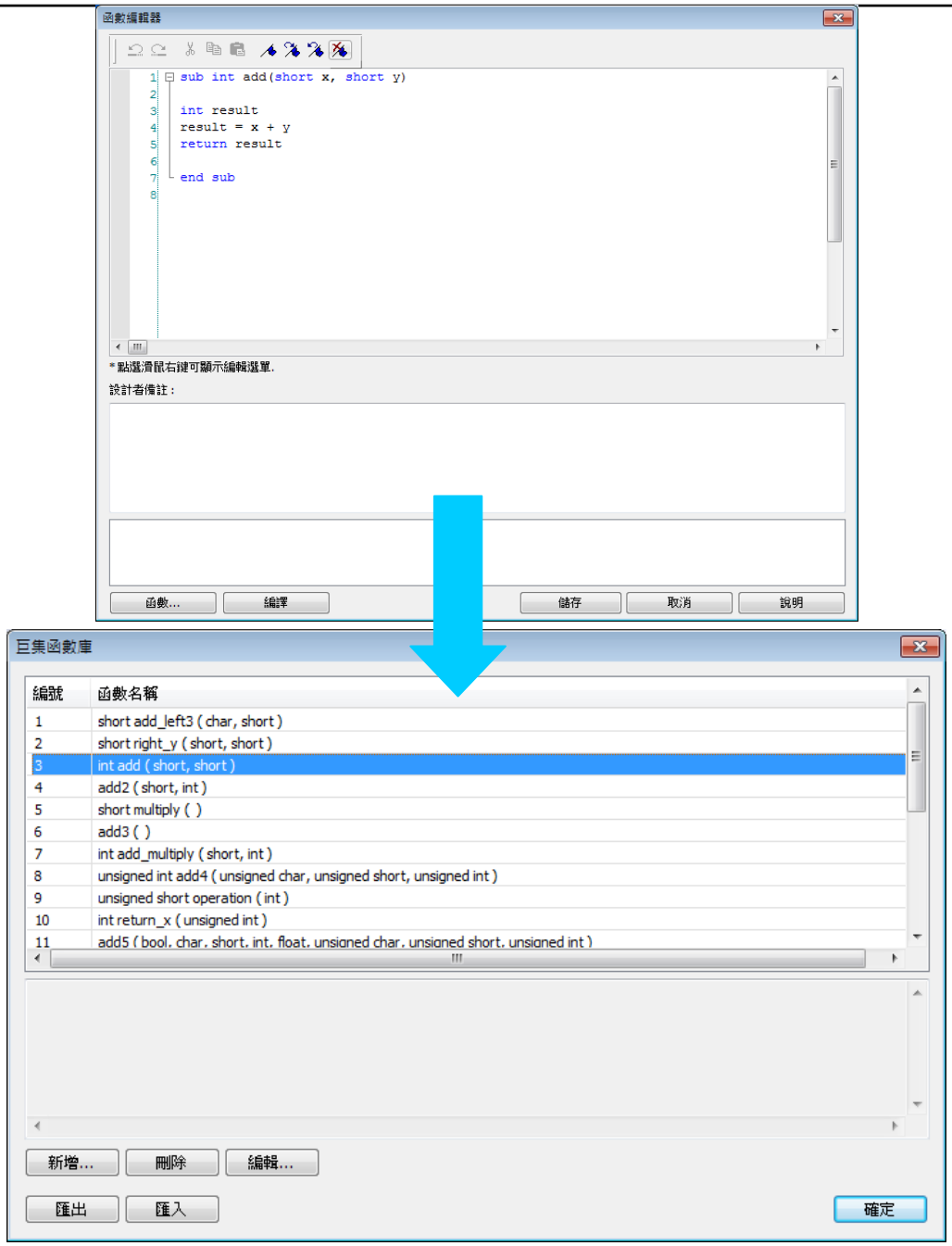
1. 在“功能键”的一般属性页对话框中，勾选“触发宏指令”。
2. 选择需要执行的宏指令。每按一下这个功能键时，选择的宏指令就会被执行一次。

- 使用“宏指令编辑器”设定条件

1. “周期执行”：可以设定每间隔几秒自动执行宏指令一次。
2. “当 HMI 启动时即执行一次”：当触摸屏重新上电或重新启动时会执行此宏指令一次。

18.9. 用户自定义函数功能

在使用宏编辑器时，为了减少定义函数的时间，用户可从内建的函数库搜寻所需的函数。然而，当用户编辑宏时，若有某些特定的函数常常使用却无法从内建函数库搜寻到时，就可以自行定义所需的函数并储存起来。当下次需要再定义相同的函数时，可由“宏指令库”呼叫出已储存的函数，方便函数编辑。另外，”宏指令库”也大幅提升了用户自订函数之可移植性。建立函数前可先查看现有内建函数或在线函数库是否有现成函数可使用。



18.9.1. 汇入函数库文件

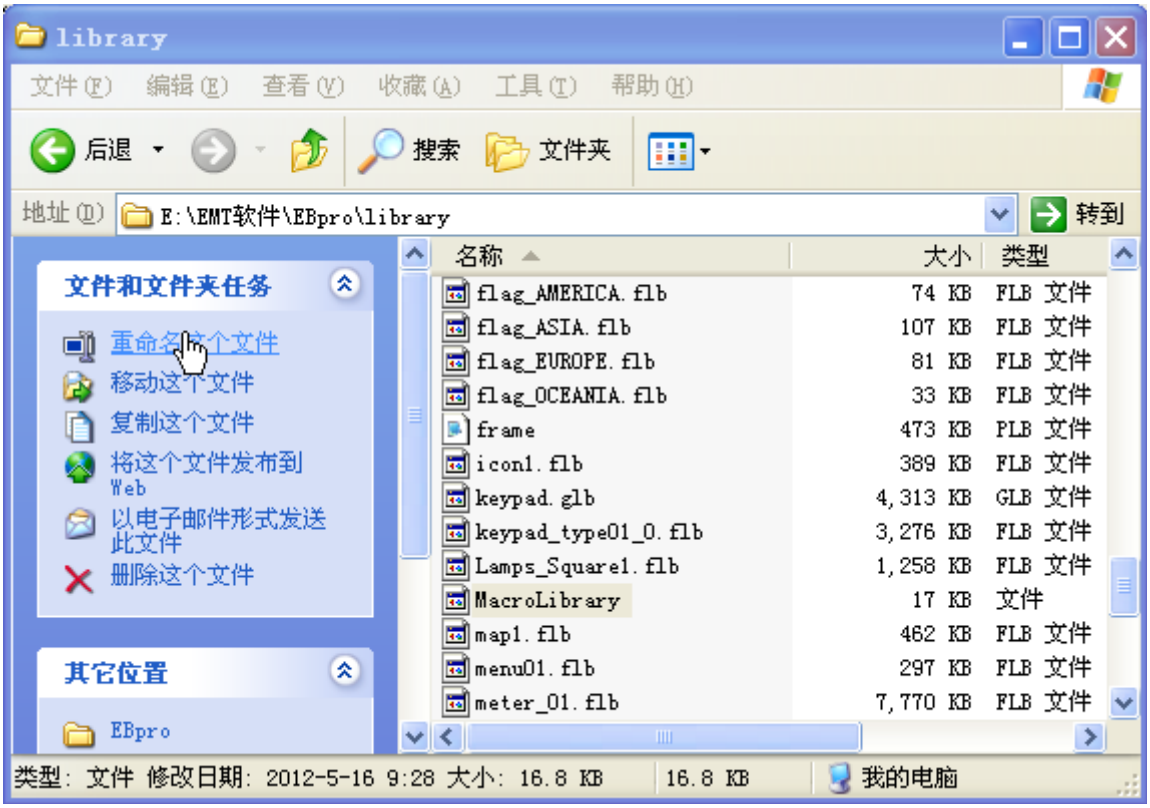
HMI 编辑软件开启一个工程文件时，会自动去读入一个预设函数库文件，并加载函数信息。当开启一项目有呼叫到用户定义函数时必须先加载相关的 .mlb 文件。

1. 预设函数库文件名：MacroLibrary (没有扩展名)
2. 函数库路径：HMI 编辑软件的安装目录的 \library 文件夹下面。
3. \library 文件夹下面可以看到两种函数库文件：

   没有扩展名： MacroLibrary，为预设函数库，触摸屏编辑软件指定读入此文件。

有扩展名（.mlb）：例如 math.mlb，用户汇入/导出时会去读 / 写的文件，具可移植性，欲使用时再从文件夹呼叫出文件即可。

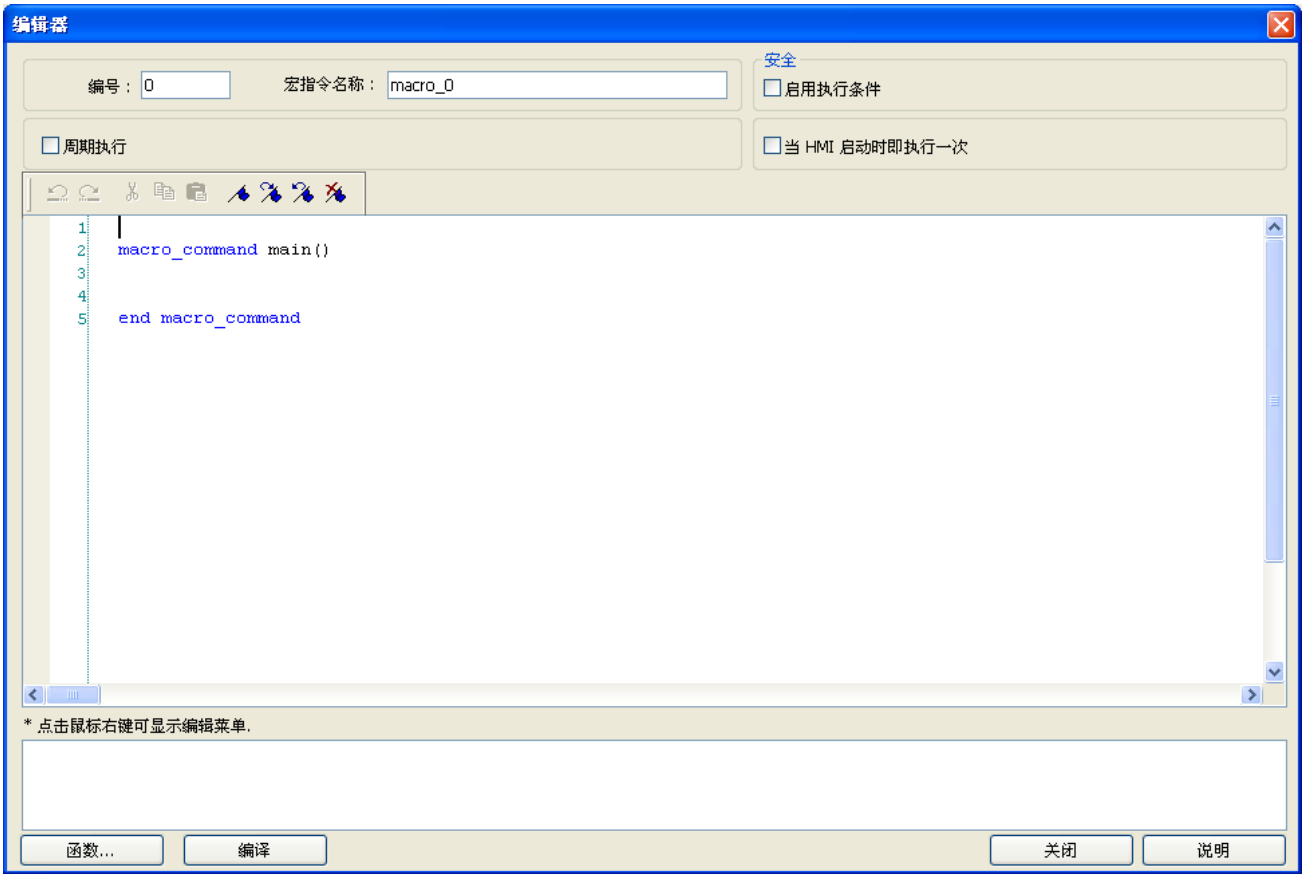
4. EasyBuilder Pro 开启时，只会去加载预设函数库中的函数，用户若需要用到 .mlb 文件内的函数时，必须自行将其汇入。



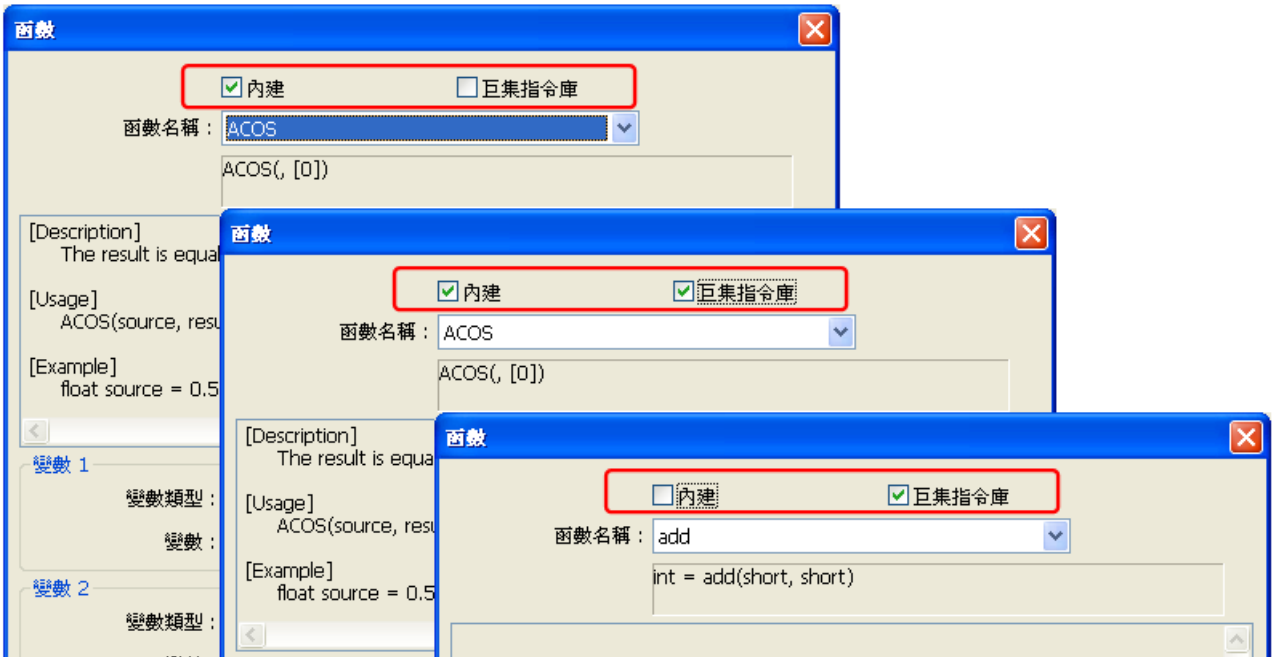
18.9.2. 如何使用宏函数库

1. 在宏编辑器中直接呼叫函数。

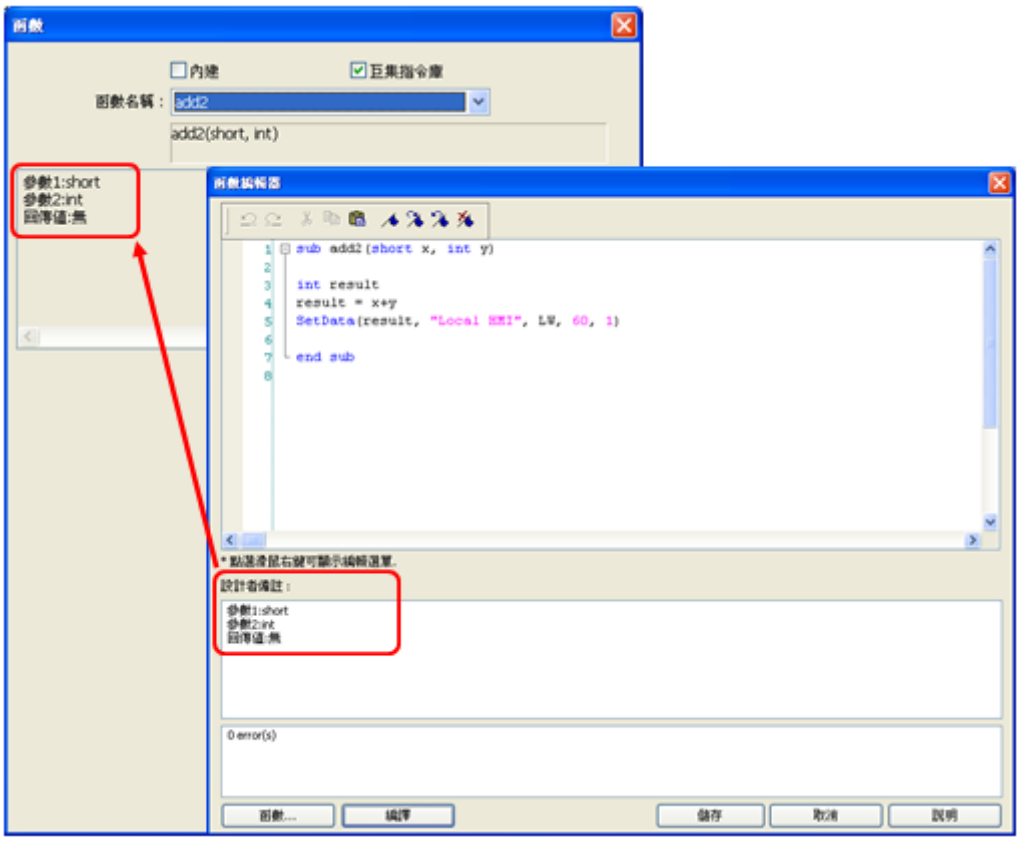
2. 按下宏编辑器左下角的“函数”按钮开启函数对话框。



3. 须至少勾选一项“宏指令库”或“内建”，并选择欲使用的函数。



4. 显示函数说明文字，即是用户在函数编辑器中编辑的说明文字。



5. 选取欲使用的函数，将函数库中的“变量类型”修改为指定的变量名称。

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

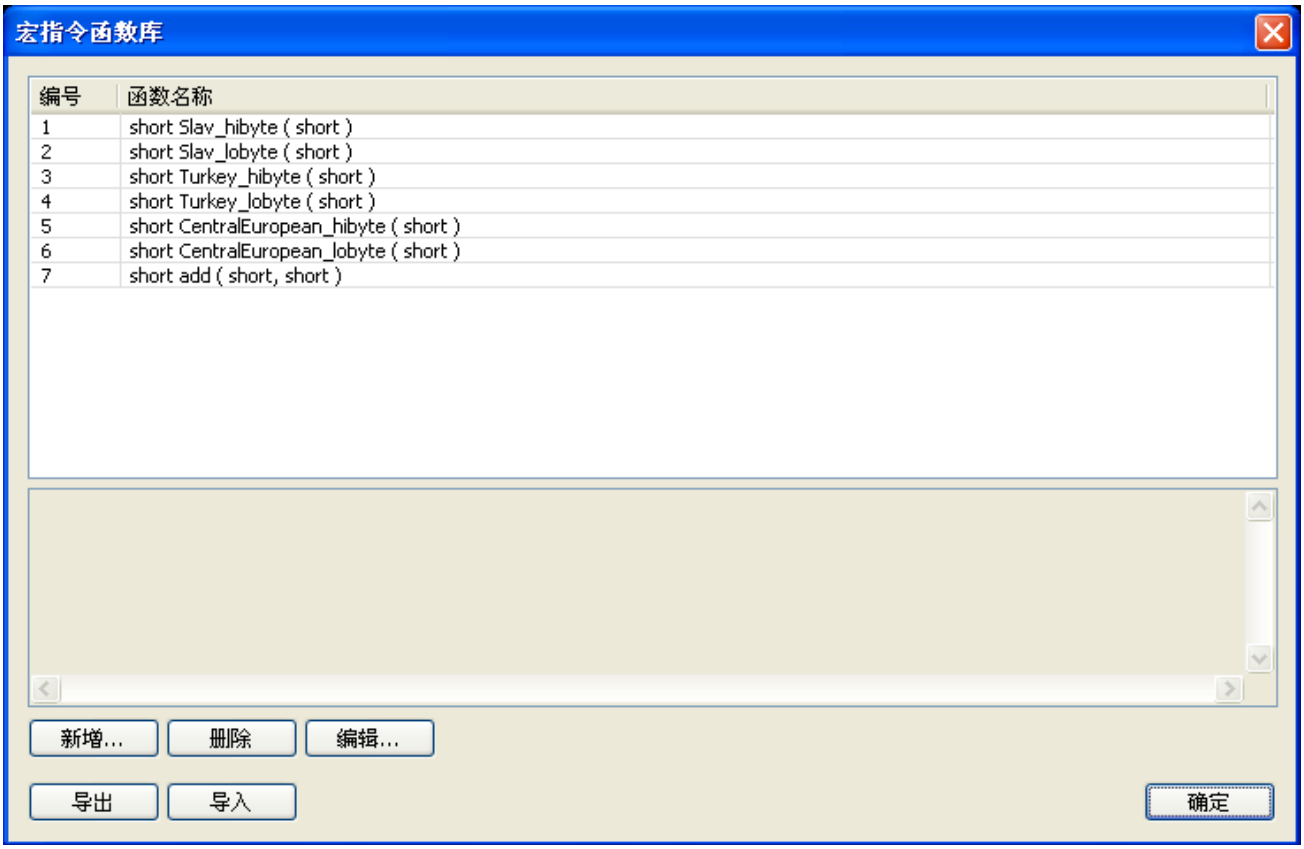
```
1
2   macro_command main()
3
4   short a
5   int b,result
6
7   result = add2(a, b)
8
9   end macro_command
```

6. 用户根据以上的步骤，即可完成使用自订函数的功能，有效节省了重复定义相同的函数。

18.9.3. 函数库管理接口

1. 开启宏管理对话框，按下右下角“宏指令库”按钮，进入函数库管理对话框接口。

2. 函数库管理对话框中有函数列表。此列表列出工程文件开启时，触摸屏编辑软件会从预设函数库加载所有函数。



3. 函数列表中每一行的格式如下：

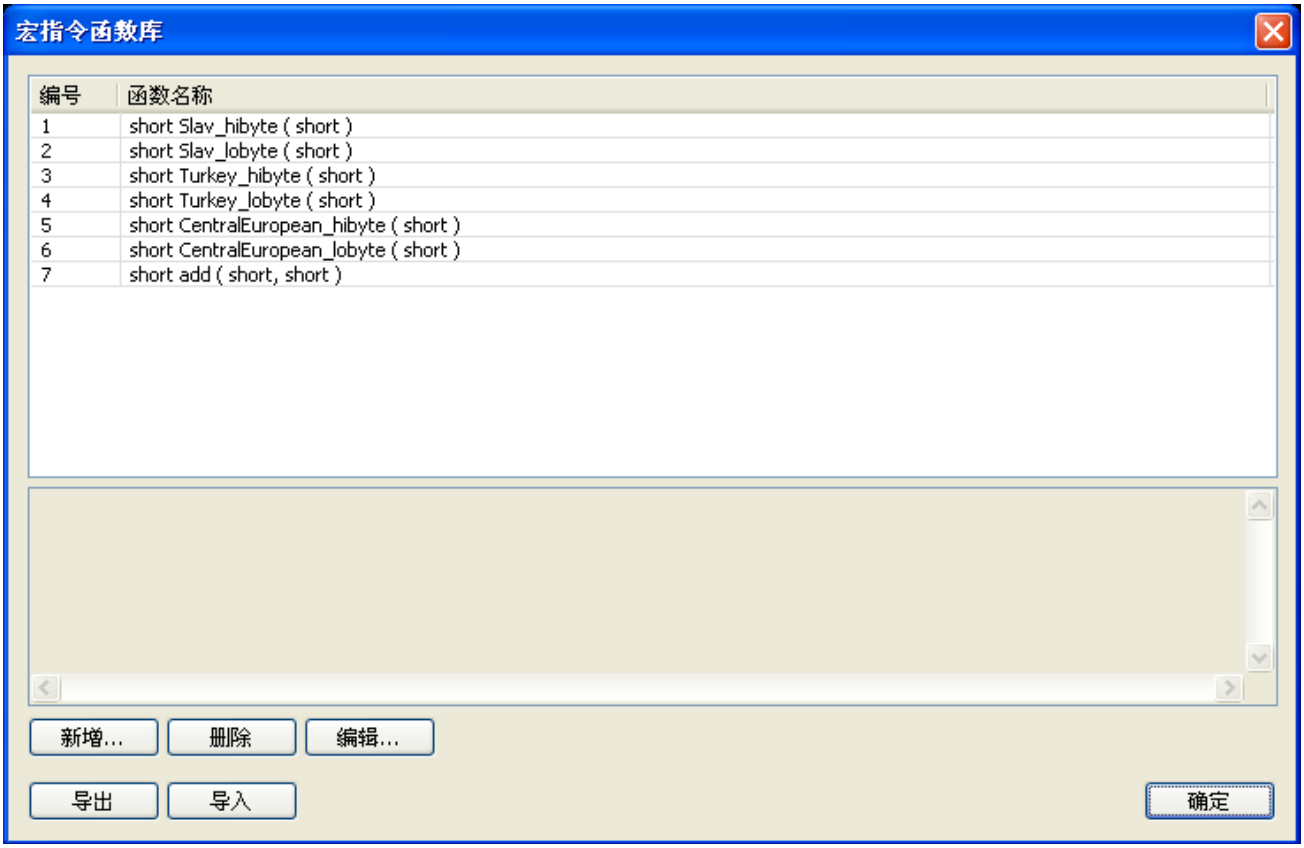
return_type function_name (*parameter_type1*, ..., *parameter_typeN*)

return_type 表示回传值类型，若无回传值则此字段省略；function_name 表示函数名称。*parameter_typeN* 表示第 N 个参数类型，若此函数不接受任何参数，此字段省略。

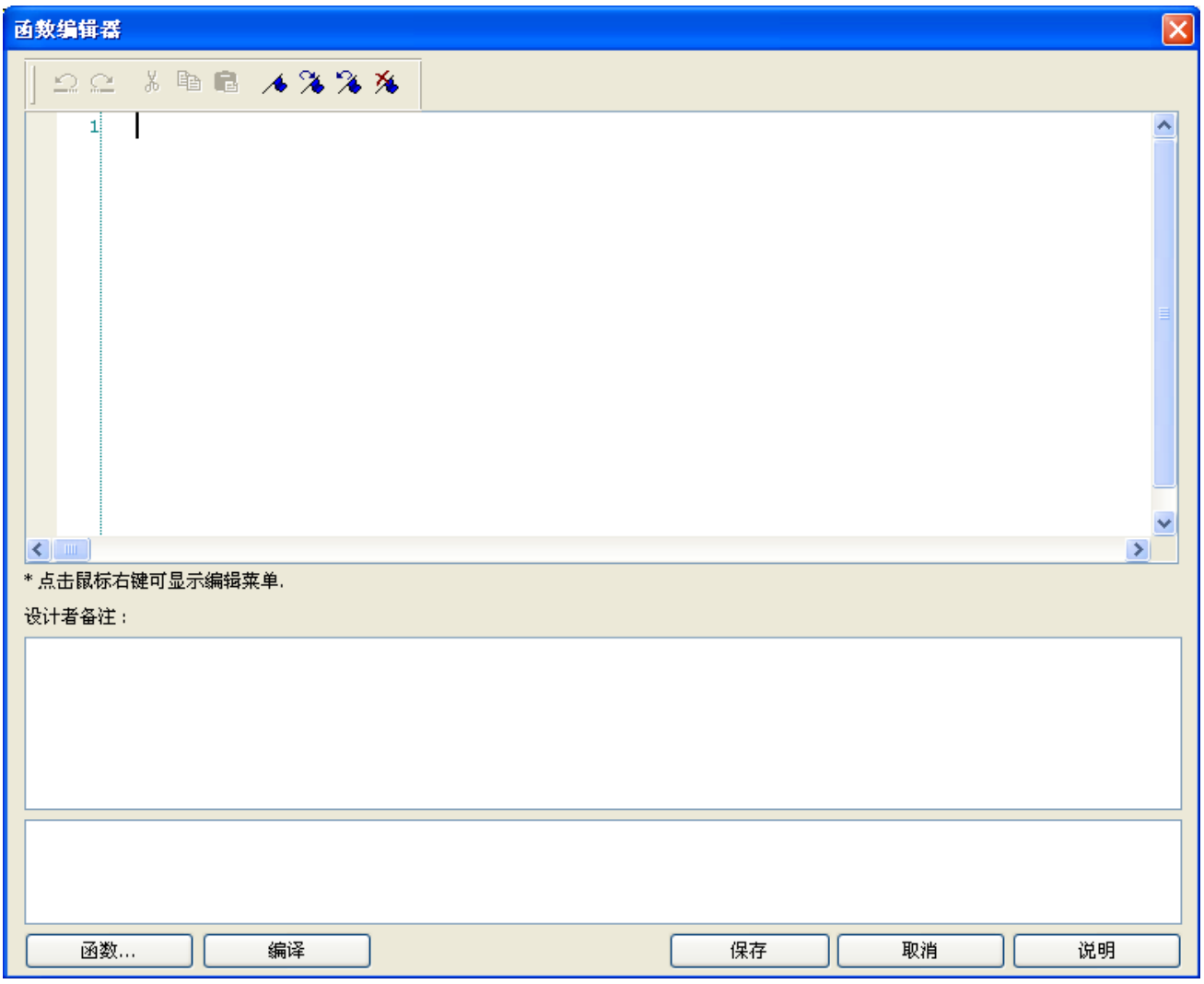
```
1  sub int ADD(int a, int b)
2    int ret
3    ret = a+b
4    return ret
5  end sub
6
```

新建函数

1. 按下“新增”进入函数编辑器。

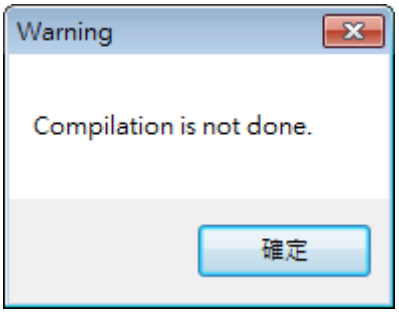


2. 在函数编辑区编辑函数，接着按“编译”再按“储存”。

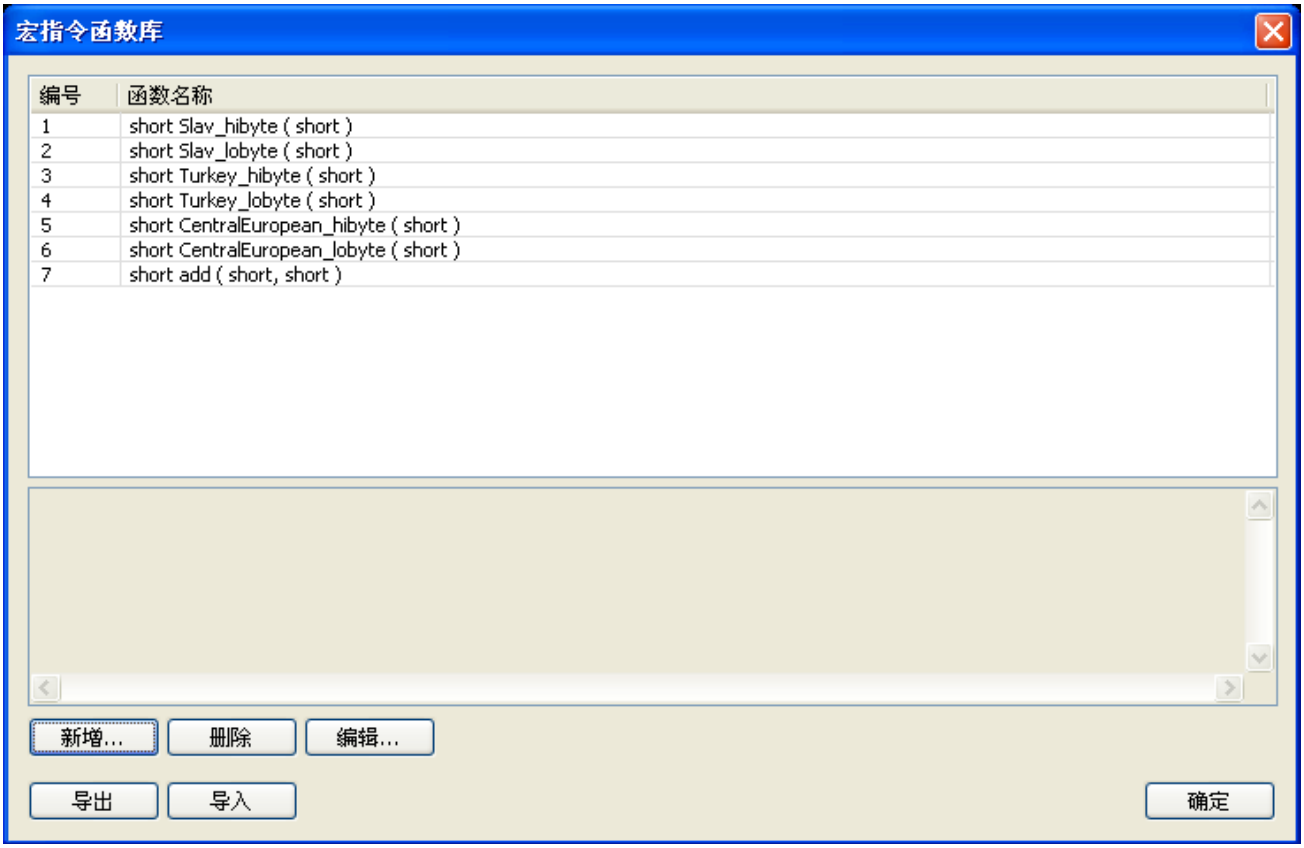


3. 用户可以在函数说明编辑区中编辑说明文字，说明此函数的规格、使用方法、作者声明等等。

4. 函数编辑完成后，必须通过编译，才可以按下“储存”写入函数库。否则会出现下图弹出窗口，警告用户此函数未完成编译。



5. 成功加入函数库。

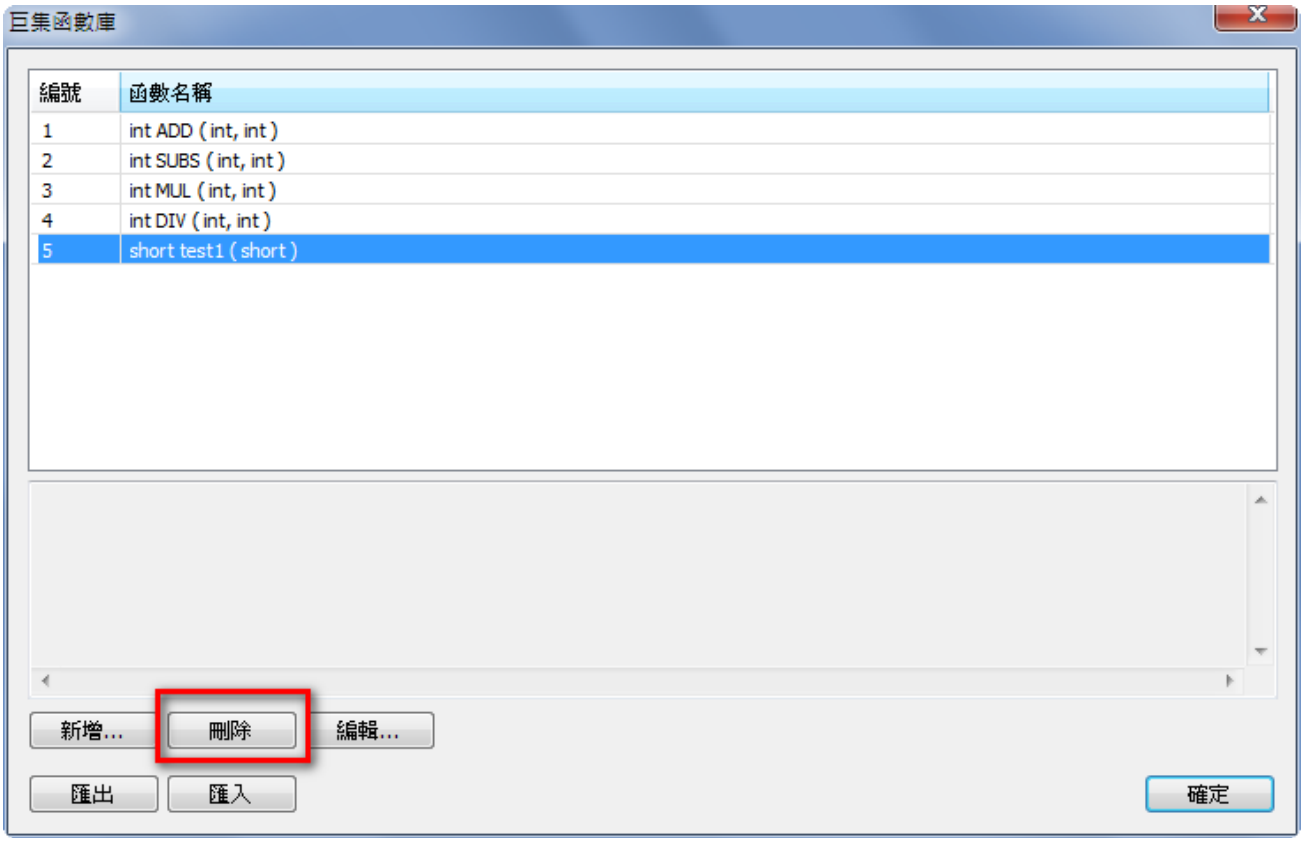


Note

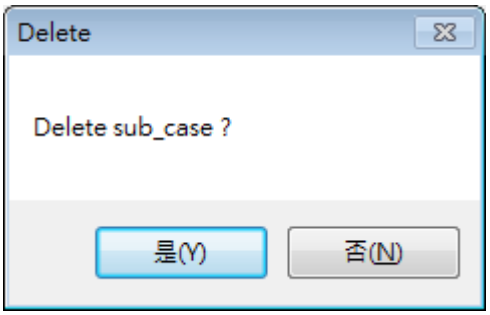
- 函数中可宣告的数据类型总数为 4096 个字节。
- 函数名称必须为英数字符，且不可为数字开头。

删除函数

1. 在函数列表中选定要删除的函数，按下“删除”按钮，即可删除该函数。

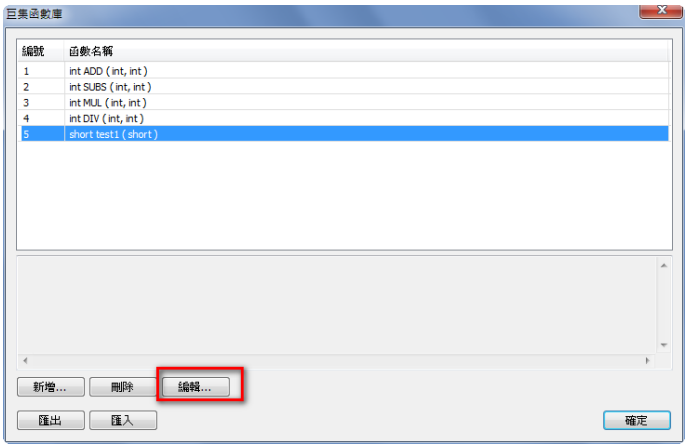


2. 按下“是”确认删除，按下“否”取消删除。按下“是”删除 MAX_SHORT 此函数。

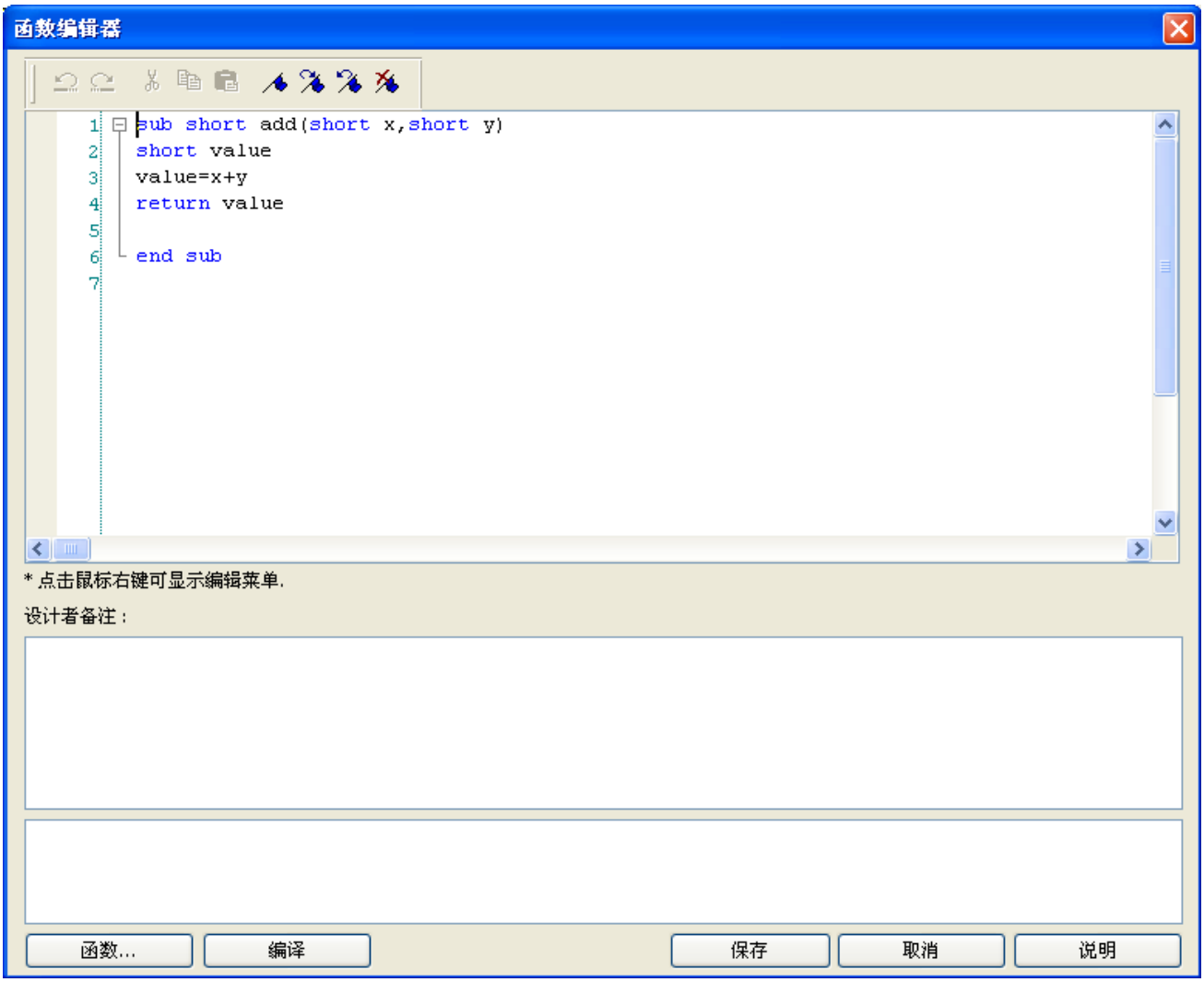


修改函数

1. 用户可以修改函数库中的函数。
2. 选定要修改的函数，按下“编辑”按钮，进入函数编辑器。



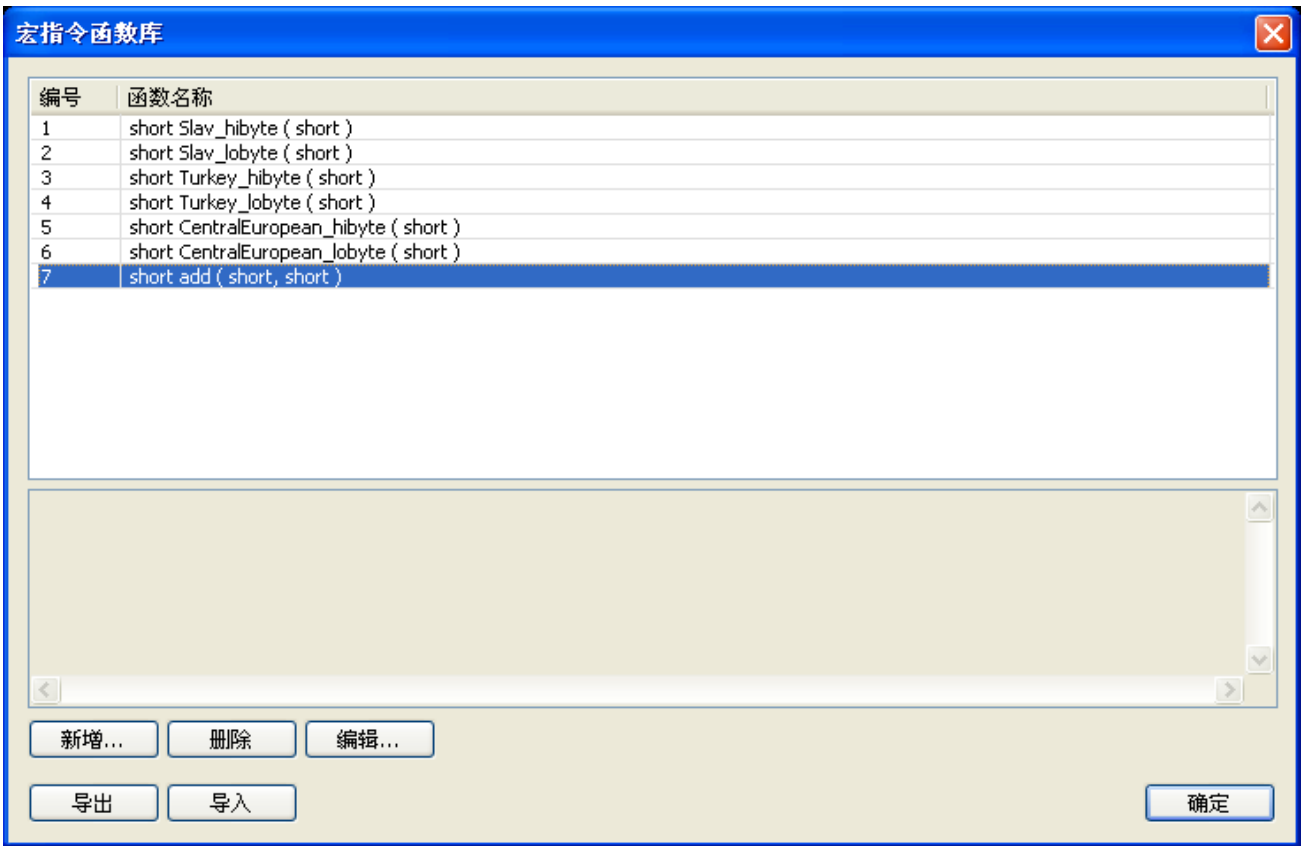
3. 也可直接在该函数上双点鼠标左键，进入函数编辑器。



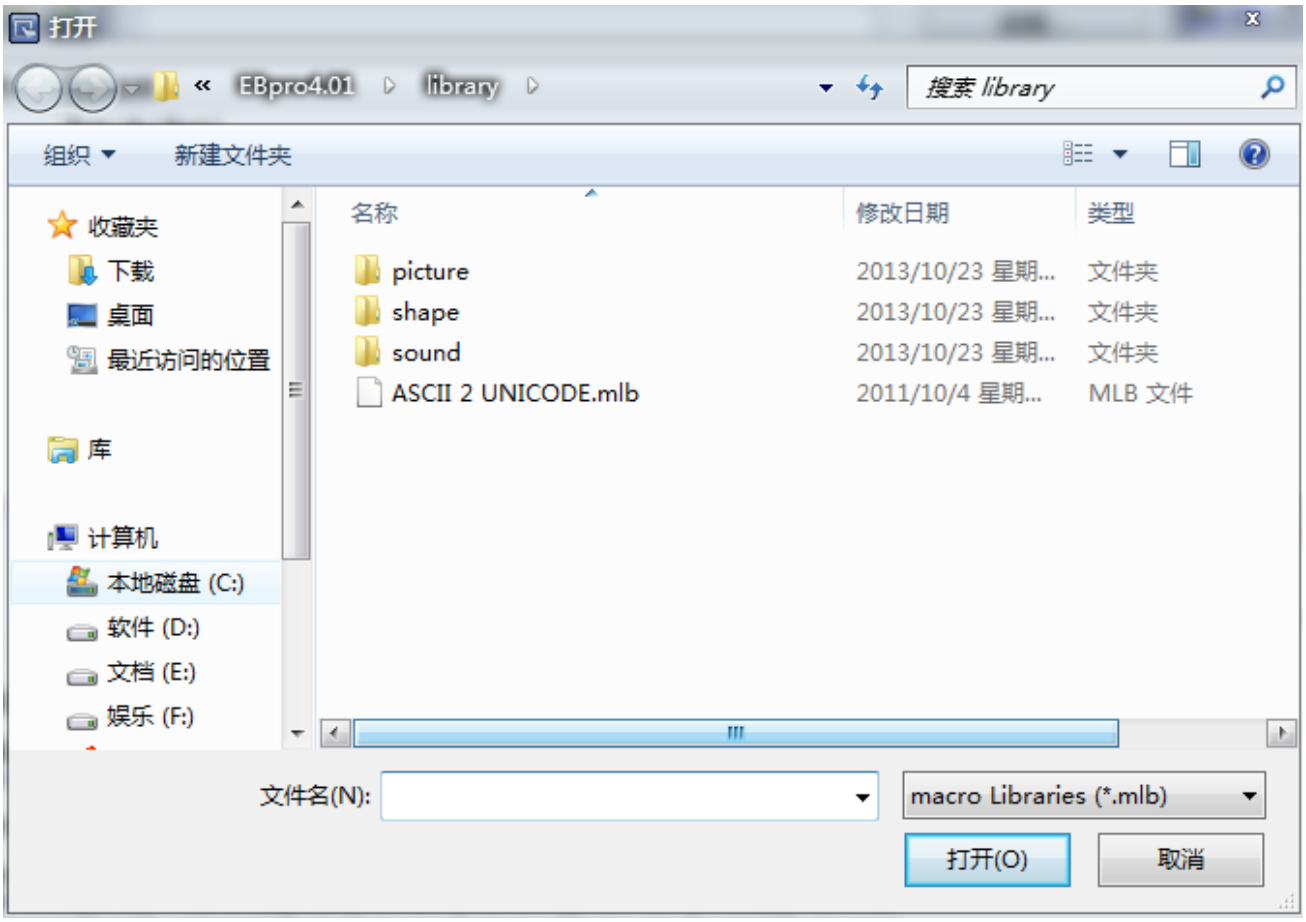
4. 修改完成后，一样要先通过编译，才可以储存离开。

导入函数

1. 用户可以从外部的 .mlb 文件将函数导入。


宏指令函数库
编号
函数名称
1 short Slav_hibyte (short)
2 short Slav_lobyte (short)
3 short Turkey_hibyte (short)
4 short Turkey_lobyte (short)
5 short CentralEuropean_hibyte (short)
6 short CentralEuropean_lobyte (short)
7 short add (short, short)
新增...
删除
编辑...
导出
导入
确定

2. 假设欲加入函数库 math.mlb，此函数库内含有一函数 test1。按下”开启旧文件”按钮。



3. 导入函数时，若文件库中已存在相同名称的函数，会出现下面的弹出窗口。

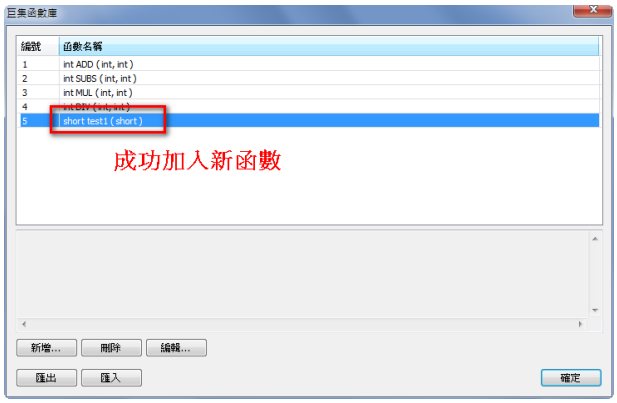
“确定”：用外部导入的函数覆盖函数库中现有的同名函数。

“否”：放弃外部导入此同名函数。

“全部确定”：全部用外部导入的同名函数覆盖所有同名函数。

“全否”：全部放弃覆盖导入同名的函数。

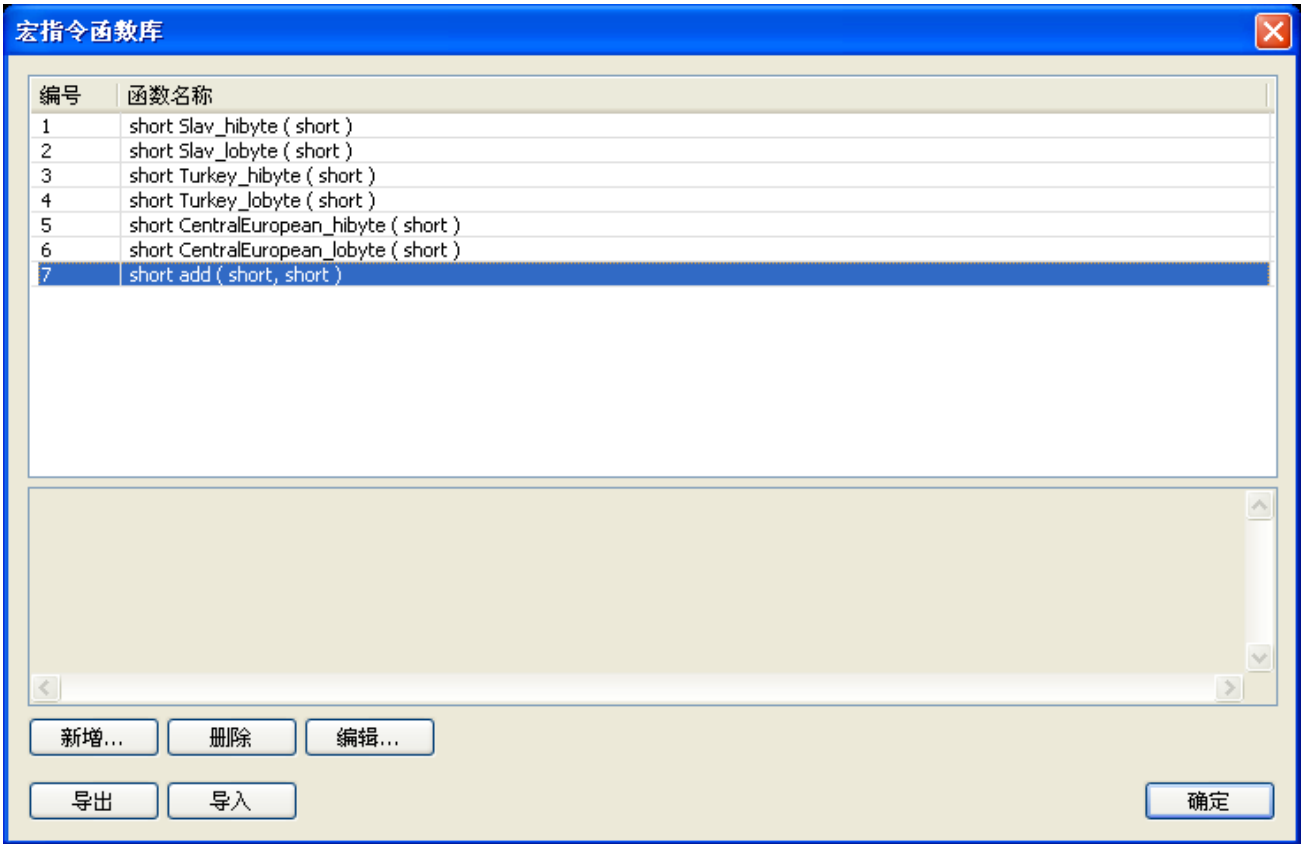
4. 完成导入后，导入的函数都已写入到预设函数库中，因此用户可以将 math.mlb 文件删除，而 test1 仍会存在函数库中，即使重新开启触摸屏编辑软件亦然。



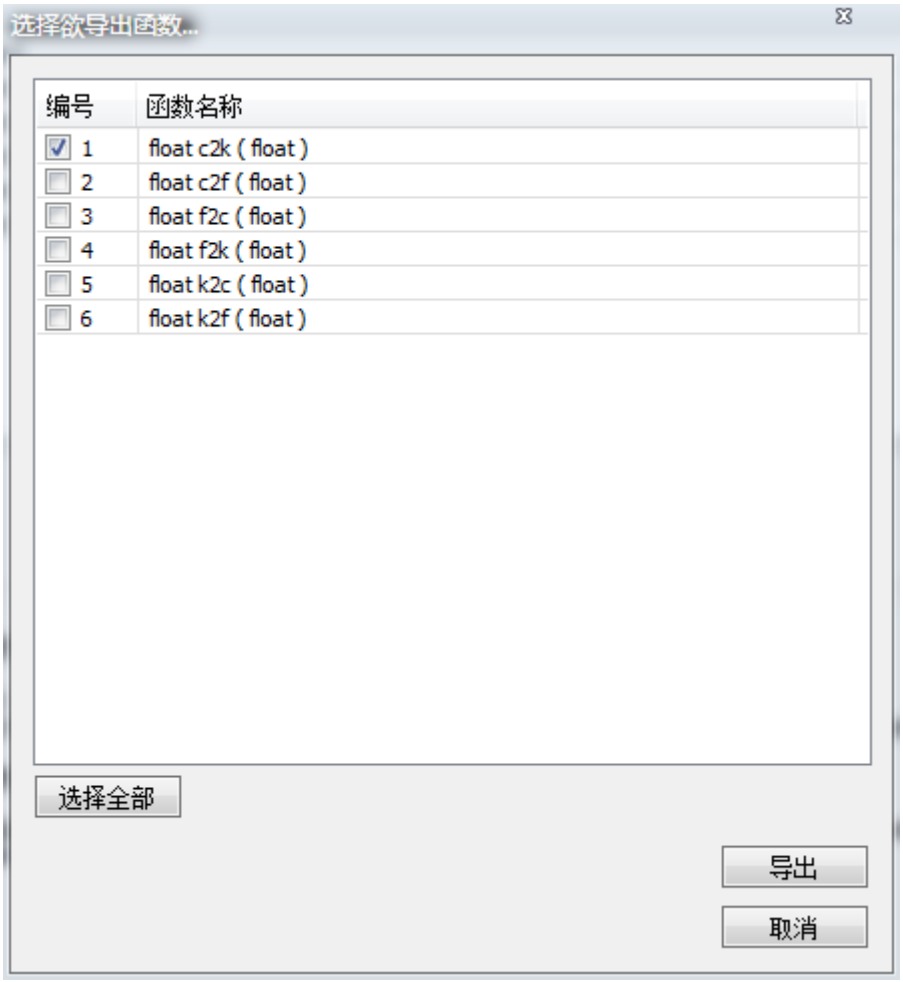
成功加入新函數

导出函数

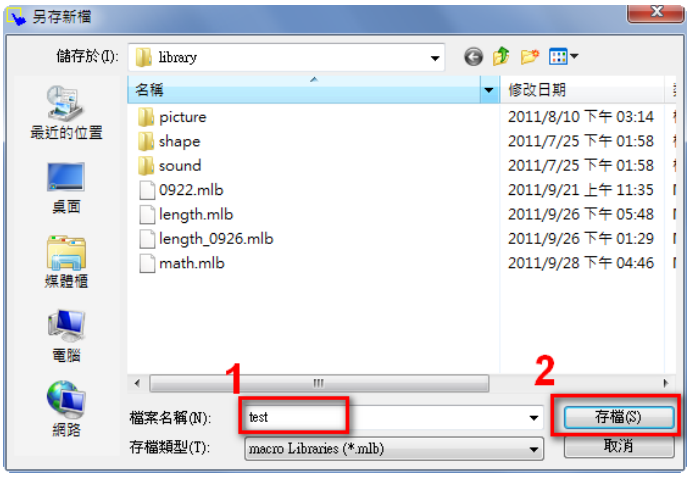
1. 用户可以将函数库中的函数导出到 .mlb 檔。按下“导出”按钮。



2. 选择要导出的函数，然后按“导出”。



3. 导出的目录下出现一个 math.mlb 的函数库文件，其中包含 ADD、SUBS、MUL、DIV 这四支函数。
4. 导出的 .mlb 文件可以携带到别的计算机上，只要用户开启触摸屏编辑软件并完成导入动作后，即可使用此文件内所提供的所有函数。



18.10. 使用宏指令时的注意事项

1. 储存局部变量的空间是 4KB，所以各种不同变量类型的最大数组大小为如下：

   char a[4096]
   bool b[4096]
   short c[2048]
   int d[1024]
   float e[1024]

2. 一个 EasyBuider Pro 工程中最多包含 255 个宏指令。
3. 宏指令有可能造成触摸屏当机，可能的原因为：

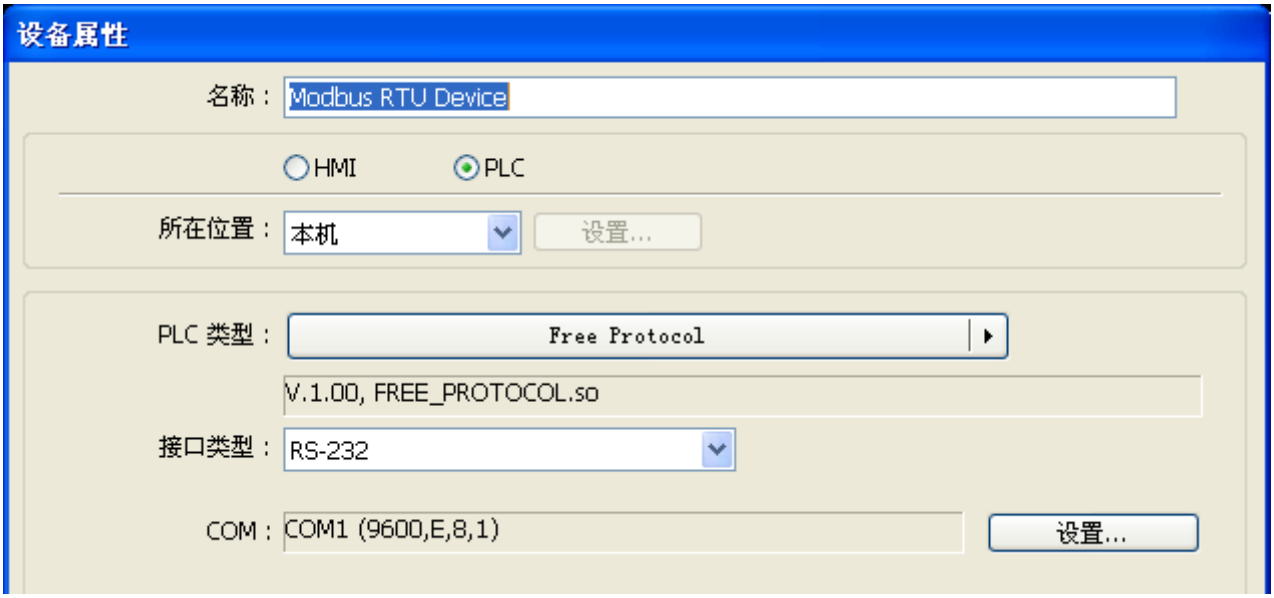
- 宏指令中执行了一个死循环命令
- 数组的大小超过了宏指令的变量容量

4. PLC 的通讯速度可能影响宏指令的执行速度。相对的，使用过多的宏指令，可能会造成与 PLC 的通讯速度变慢。

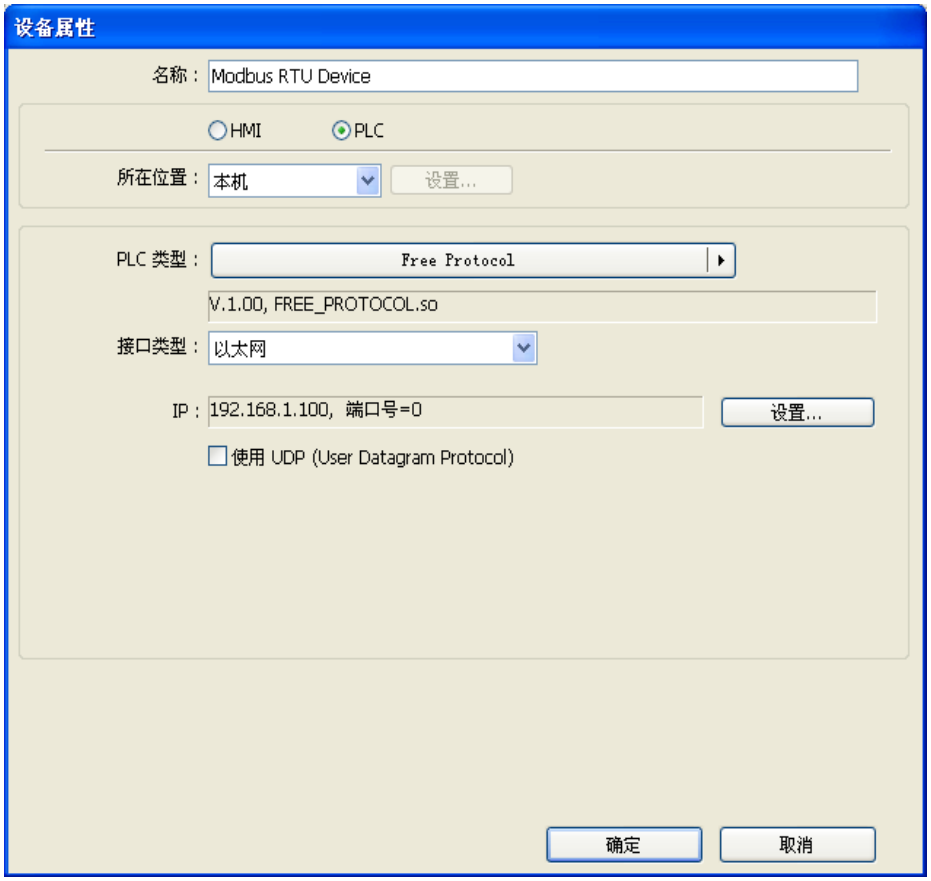
18.11. 使用自由协议去控制一个设备

当 EasyBuilder Pro 没有内建与某一个设备通讯的驱动程序时，用户也可以使用宏指令中的 OUTPORT 和 INPORT 函数来实现与该设备的通讯。使用 OUTPORT 和 INPORT 函数发送和接收的数据，必须遵行该设备的通讯协议。下面的范例程序说明了如何使用这两个函数来控制一个 MODBUS RTU 设备。

1. 首先，在系统参数/设备列表中建立一个新的设备。这个新建的 “PLC 类型” 设置为 “Free Protocol”，”PLC 名称” 设置为 “MODBUS RTU Device”，如下图所示。



2. 这里的设备连接使用 RS232，如果连接一个 MODBUS TCP / IP 设备，这个接口类型必须设定为“以太网”，同时必须设定正确的 IP 地址和连接端口号，如下图所示。



假设触摸屏触摸屏界面将要读取设备中的 4x_1 和 4x_2 两个数据寄存器。首先，使用 OUTPORT 函数发送读命令给这个设备，OUTPORT 函数的写法为：

OUTPORT(command[start], device_函数名称, cmd_count)

因为“MODBUS RTU Device”是一个 MODBUS RTU 设备，读数据的命令必须遵行 MODBUS RTU 协议的命令规则。所以必须使用 0x03 这个命令去读取 4x_1 和 4x_2 这两个数据寄存器的数据。下图说明了读命令的格式。(省略了设备的站号和最后的两个 CRC 字节)

Request

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

Response

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

*N = Quantity of Registers

Error

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

根据这个 MODBUS RTU 协议，发送命令的内容如下所示：（总共 8 个字节）

Command[0]: station number (BYTE 0) 站号

Command[1]: function code (BYTE 1) 功能码

Command[2]: high byte of starting address (BYTE 2) 起始地址高位

Command[3]: low byte of starting address (BYTE 3) 起始地址低位

Command[4]: high byte of quantity of registers (BYTE 4) 数据寄存器的高位

Command[5]: low byte of quantity of registers (BYTE 5) 数据寄存器的低位

Command[6]: low byte of 16-bit CRC (BYTE 6) CRC 的低字节

Command[7]: high byte of 16-bit CRC (BYTE 7) CRC 的高字节

所以读数据的命令宏指令程序设计如下：

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

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

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

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

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

最后，使用 OUTPORT 函数将这个读命令发送给这个 MODBUS RTU 设备。

```
OUTPORT(command[0], “MODBUS RTU Device”, 8)   //   发送命令
```

发送完这个命令后，使用 INPORT 函数读取该 MODBUS RTU 设备返回的命令。根据 MODBUS RTU 协议，这个回复的命令内容为如下 (总共 9 个字节)：

Command[0]: station number (BYTE 0) 站号
Command[1]: function code (BYTE 1) 功能码
Command[2]: byte count (BYTE 2) 位组长度
Command[3]: high byte of 4x_1 (BYTE 3) 第一个数据的高字节
Command[4]: low byte of 4x_1 (BYTE 4) 第一个数据的低字节
Command[5]: high byte of 4x_2 (BYTE 5) 第二个数据的高字节
Command[6]: high byte of 4x_2 (BYTE 6) 第二个数据的低字节
Command[7]: low byte of 16-bit CRC (BYTE 7) CRC 的低字节
Command[8]: high byte of 16-bit CRC (BYTE 8) CRC 的高字节

此时，INPORT 函数的语句如下：

```
INPORT(response[0], “MODBUS RTU Device”, 9, return_value)  //  读取回复命令
```

函数中，实际读取到的位组长度存放在变量 return_value (变量类型为字节) 中。如果 return_value 的数据为 0，则表示使用 INPORT 读取命令失败。

根据 MODBUS RTU 协议，如果命令回复的正确，则 response[1] 必须为 0x03。当读取到正确的命令后，计算出 4x_1 和 4x_2 这两个寄存器的值，并将这两个数据送到触摸屏界面的 LW-100 和 LW-101 寄存器中。

```
If (return_value) >0 and response[1] == 0x3) then
read_data[0] = response[4] + (response[3] << 8) //  计算 4x_1 的资料
read_data[1] = response[6] + (response[5] << 8) //  计算 4x_2 的资料
SetData(read_data[0], “Local HMI”, LW, 100, 2)  //  将资料存至 HMI 上
endif
```

完整的宏指令程序如下：

```
// Read Holding Registers
macro_command main()
char command[32], response[32]
short address, checksum
short read_no, return_value, read_data”2], i
FILL(command[0], 0, 32)//   initialize command[0]~command[31] to 0
FILL(response[0], 0, 32)
Command[0] = 0x1//   station number
Command[1] = 0x3//   read holding registers (function code is 0x3)
address = 0
address =   0//   starting address (4x_1) is 0
HIBYTE(address, command[2])
LOBYTE(address, command[3])
read_no = 2/   the total words of reading is 2 words
HIBYTE(read_no, command[4])
LOBYTE(read_no, command[5])
CRC(command[0], checksum, 6)//   calculate 16-bit CRC
LOBYTE(checksum, command[6])
HIBYTE(checksum, command[7])
OUTPORT(command[0], "MODBUS RTU Device", 8 )//   send request
INPORT(response[0], "MODBUS RTU Device", 9, return_value)//   read response
if (return_value > 0 and response[1] == 0x3) then
   read_data[0] = response[4] + (response[3] << 8)//   4x_1
   read_data[1] = response[6] + (response[5] << 8)//   4x_2

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

end macro_command
```

下面的举例说明如何使用自由协议设定 MODBUS RTU 设备中 0x_1 的状态。这个是使用 MODBUS RTU 协议中的“写单个寄存器”的功能码“0x05”来实现的。

Request

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

Response

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

Error

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

完整的宏指令程序如下：

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

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

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

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

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

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

CRC(command[0], checksum, 6)

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

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

end macro_command
```

18.12. 编译错误提示信息

- 错误信息格式：

error C# : error 描述

(# 是错误信息编号)

举例：error C37：undeclared identifier：i

当编译后提示有错误信息时，这个错误的描述内容可以参考错误信息编号。

- Error 描述

(C1) 语法 error：“标识符”

出现这个信息时，有许多种可能：

举例:

```
macro_command main()
char i, 123xyz    // 不支持这个变量类型
end macro_command
```

(C2) 'identifier' used without having been initialized (使用的标识符没有初始化)

宏指令必须定义声明的数组变量的大小

举例:

```
macro_command main()
char i
int g[i]    // i 必须为一个数值常数
end macro_command
```

(C3) redefinition error : 'identifier' (标识符被重复定义)

函数名称和变量名称在有效范围内，必须是唯一的。

举例:

```
macro_command main()
int g[10]，g    // 重复定义错误
end macro_command
```

(C4) function 函数名称 error : 'identifier' (函数名称定义错误)

保留的关键词和常数，不能被定义为函数名称

举例：

```
sub int if()  // 函数名称定义错误
```

(C5) parentheses have not come in pairs (圆括号没有成对的出现)

语句中缺少了 "(" or ")"

举例：

```
macro_command main  )  // 缺少了 "("
```

(C6) illegal expression without matching 'if' (if 语句中没有合法的表达式)

也就是在 if 语句中缺少表达式

(C7) illegal expression (no 'then') without matching 'if' (if 语句中缺少了 then)

也就是 if 和 then 没有成对

(C8) illegal expression (no 'end if') (if 语句中缺少了 enf if)

缺少了 "end if"

(C9) illegal 'end if' without matching 'if' (end if 语句前缺少了 if)

End if 语句前缺少了 if 语句

(C10) illegal 'else' (不合法的 else 语句)

这个 "if" 语句的格式为：

if "逻辑表达式" then

" else "if "逻辑表达式" then " "

end if

任何与以上格式不符合的语句，在编译时就会错误。

(C17) illegal expression (no 'for') without matching 'next' (没有与 next 相配的 for 语句)

"for" 语句错误：在 "next" 前，缺少了 "for" 语句

(C18) illegal variable type (not integer or char) (不合法的变量类型)

变量类型定义错误，此处应为整数类型或字符类型变量

(C19) variable type error

缺少赋值语句

(C20) must be keyword 'to' or 'down' (缺少了关键词 "to" 或者 "down")

缺少了关键词 "to" 或者 "down"

(C21) illegal expression (no 'next') (非法的表达式，缺少了 "next")

"for" 语句的格式为：

for "变量" = "初始值" to "结束值" "step"

next "变数"

任何与上述格式不符合的语句，编译时会错误。

(C22) 'wend' statement contains no 'while'

循环缺少“while”关键词，”wend”前面应有“while”关键词

(C23) illegal expression without matching 'wend'

缺少“wend”关键词

“while”语句的格式为：

while “逻辑表达式”

wend

任何不符合上述语法的，在编译时会错误。

(C24) 语法 error : 'break'

不合法的 ”break”语句。break 语句只能在 for 循环、while 循环选择结构中使用。

(C25) 语法 error : 'continue'

不合法的“continue”语句。continue 语句只会在“for”或者“while”语句中出现。

(C26) 语法 error

表达式不正确

(C27) 语法 error

表达式中缺少了一个运算符号可能会造成这个编译错误信息。

举例 ：

```
macro_command main( )
int a, b
for a = 0 to 2
b = 4 + xyz     // 不合法之处：xyz 变量没有被定义
next a
end macro_command
```

(C28) must be 'macro_command'

此处应该为“macro_command”

(C29) must be key word 'sub'

子函数的定义格式为：

sub "data type" function_函数名称 (...)

……..

end sub

举例:

sub int pow (int exp)

…….

end sub

任何不符合上述语法结构的，在编译时会错误。

(C30) number of parameters is incorrect

参数个数不对。

(C31) parameter type is incorrect

参数数据类型不相配。调用函数时，参数必须在数据类型、个数上一一对应才能通过编译，否则编译时将出现此项错误讯息。

(C32) variable is incorrect

变量类型不正确。当变量被当成参数传递给一个函数时，变量的数据类型应与函数所宣告的类型相同，否则将出现此项错误讯息。

(C33) function 函数名称 : undeclared function

没有定义的函数名称

(C34) expected constant expression

不合法的数组下标表达形式

(C35) invalid array declaration

不合法的数组定义

(C36) array index error

不合法的数组下标

(C37) undeclared identifier : i 'identifier'

使用没有定义的变量。只能使用已经定义的变量和函数，否则编译时将出现此项错误讯息。

(C38) un-supported PLC data address

通讯函数 GetData(…)、SetData(…)的参数中有包含 PLC 地址类型信息，当 PLC 地址类型不是此种 PLC 支持的地址类型时，编译时将出现此项错误信息。

(C39) 'idenifier' must be integer, char or constant

数组的格式为：

声明： array_函数名称"constant" (constant is the size of the array)

使用： array_函数名称"integer, character or constant"

任何不符合上述规则的数组表达式，编译时将会错误

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

变量定义语句的前面不能有执行语句

举例 ：

```
macro_command main(  )
int a, b
for a = 0 To 2
b = 4 + a
int h , k   // 定义变量语句在此处是错误的，在一个函数内定义变量语句的前面
不能有执行语句，例如 b = 4 + a
next a
end macro_command
```

(C41) float variables cannot be contained in shift calculation

移位运算中，操作数不能为浮点数。

(C42) function must return a value

函数应有返回值

(C43) function should not return a value

函数不应有返回值

(C44) float variables cannot be contained in calculation

运算中不能有 float 型数据

(C45) PLC address error

PLC 地址错误

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

一维数组的大小超过 4k

(C47) macro command entry function is not only one

宏指令程序入口只能有一个

(C48) macro command entry function must be only one

宏指令入口函数不是唯一。宏指令的入口函数只能有一个，形式为：

macro_command function_函数名称( )

end macro_command

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

在宏指令中，扩展地址内的站号大小只能从 0 到 255

举例 ：

SetData(bits[0] , "PLC 1", LB , 300#123, 100)

// illegal : 300#123 意思是站号为 300，但是最大值是 255

(C50) an invalid PLC 函数名称

在宏指令中，PLC 的名称并未定义在系统参数的设备列表中

(C51) macro command do not control a remote device

宏指令只能控制本机连接的设备

举例 ：

SetData(bits[0] , "PLC 1" , LB , 300#123, 100)

"PLC1"连接在远程的触摸屏上，所以它不能被执行。

18.13. 宏指令范例程序

- for 循环，各种表达式 (算术，移位元，逻辑，关系表达式)

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

  b[0] = (400 + 400 << 2) / 401
  b[1] = 22 *2 - 30 % 7
  b[2] = 111 >> 2
  b[3] = 403 > 9 + 3 >= 9 + 3 < 4 + 3 <= 8 + 8 == 8
  b[4] = not 8 + 1 and 2 + 1 or 0 + 1 xor 2
  b[5] = 405 and 3 and not 0
  b[6] = 8 & 4 + 4 & 4 + 8 | 4 + 8 ^ 4
  b[7] = 6 - (~4)
  b[8] = 0x11
  b[9] = 409

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

- while, if, break 语句

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

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

- 全局变量和子函数调用

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

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

    return y
end Sub

macro_command main()
    int a, b, i

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

- if 结构语句

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

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

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

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

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

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

- while 和 wend 结构语句

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

      b[0] = 13

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

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

      i = i + 1
    wend

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

- break 和 continue 语句结构

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

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

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

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

- 数组结构

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

    b[0] = 13

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

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

18.14. 宏指令 TRACE 函数

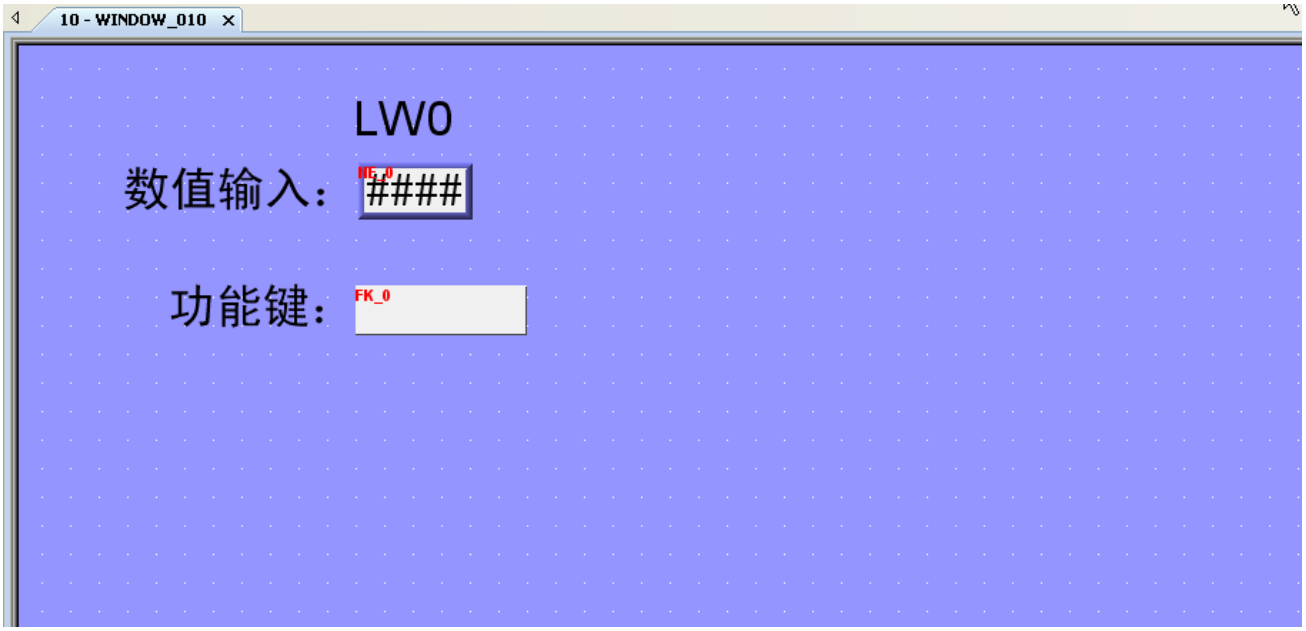
宏指令的 TRACE 函数，搭配使用 EasyDiagnoser，可用来检视所使用变量目前的内容。

下面示范如何在宏指令中利用 TRACE 命令。

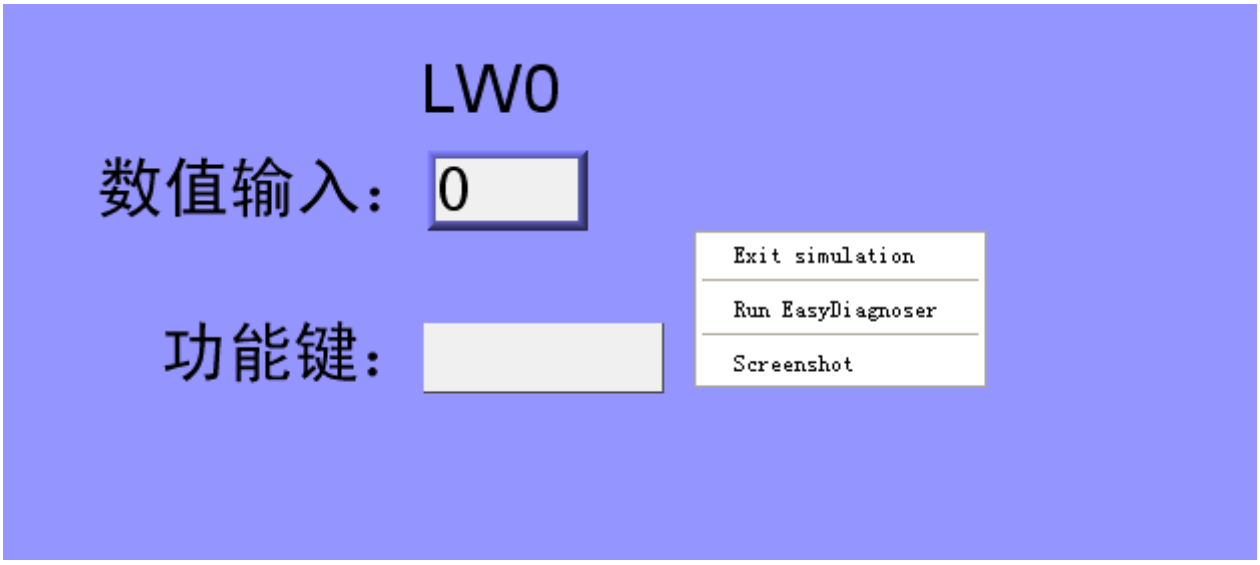
1. 首先，请在工程文件中新增 macro_0，并在 macro_0 的内容中加入 TRACE("LW = %d", a)，"%d" 表示使用 10 进制显示 LW 目前的数值。macro_0 的内容如下：

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

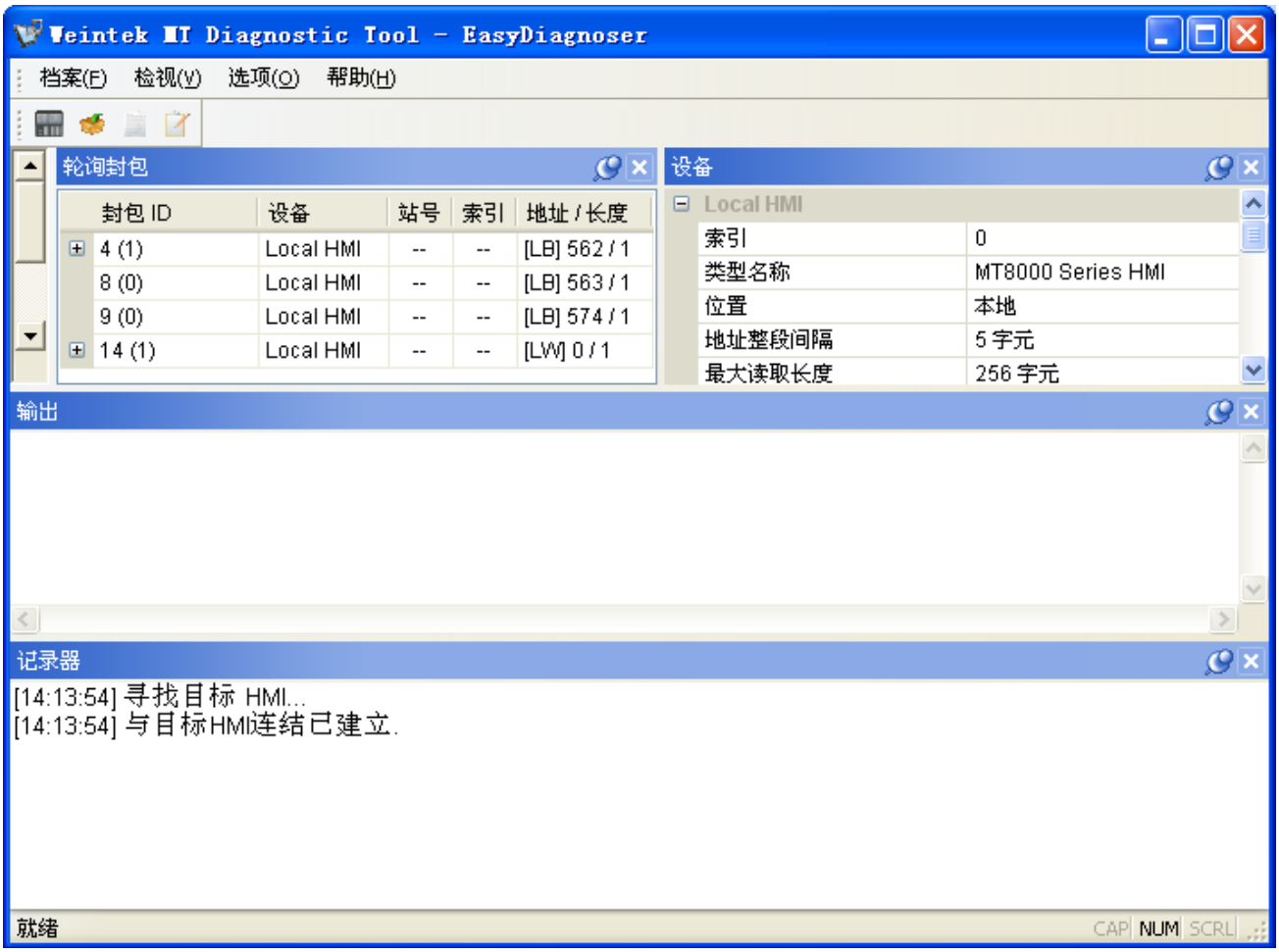
2. 接着在工程文件的第 10 页分别加上“数值显示”与“功能键”元件，”功能键”元件用来执行 macro_0。



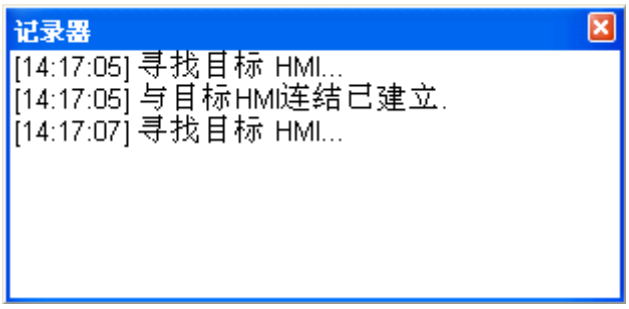
3. 最后编译已完成的工程文件并执行离线或在线模拟。
4. 在 PC 上进行仿真功能时，点击鼠标右键后选择选单上的 "Run EasyDiagnoser"。



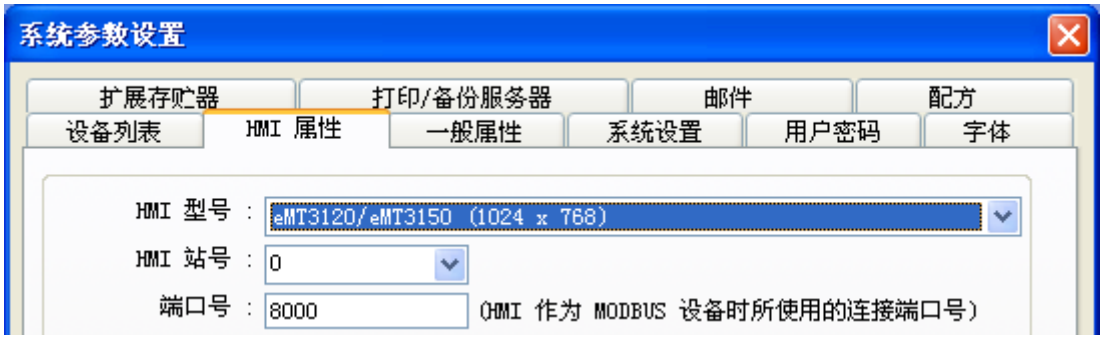
5. 此时即会出现 EasyDiagnoser 的画面，”记录器”窗口用来显示 EasyDiagnoser 是否可以连接上需要监视的 HMI，”输出”窗口用来显示 TRACE 的执行结果，下图表示 EasyDiagnoser 已成功连接上 HMI。



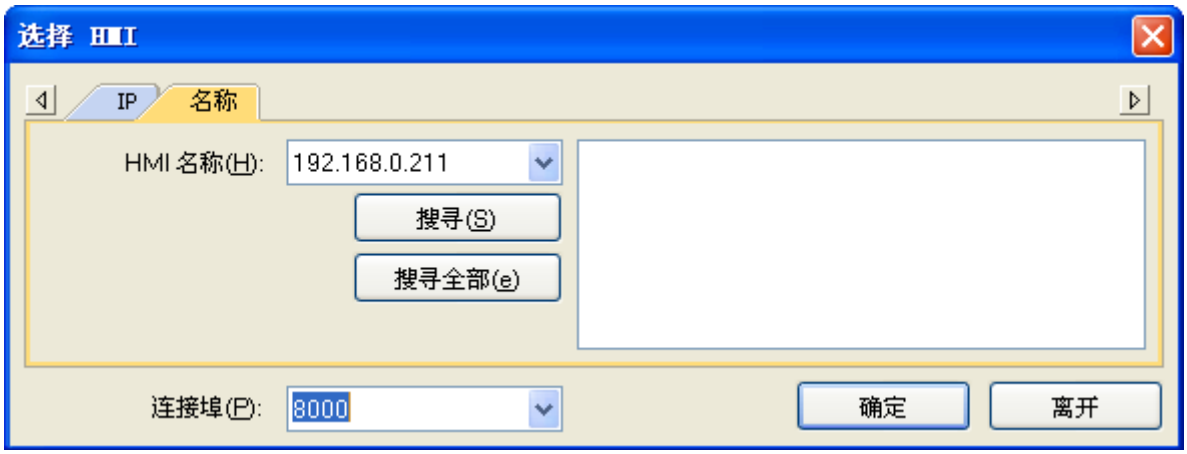
若未成功连接上 HMI，”记录器”的窗口会显示下面的内容：



6. 未联机成功的可能原因是电脑 未成功执行仿真功能；另一个原因是在电脑 执行仿真功能的工程文件所使用的“连接埠”不正确 (可能已被系统占用)，此时请更改工程文件的“连接埠”(请参考下图)，再次编译重新执行仿真功能即可。



7. 开启 EasyDiagnoser 时，应设定与工程文件相同的连接端口。



从所设定的“连接埠”算起连续 3 个 port 将保留给触摸屏做通讯用。例如，上图中设定“连接埠”为 8005，则 8005、8006、8007 三个 port 将被保留。因此在电脑 上模拟时，应确定这些被保留的 port 未被其它程序所占用。

TRACE 命令语法如下：

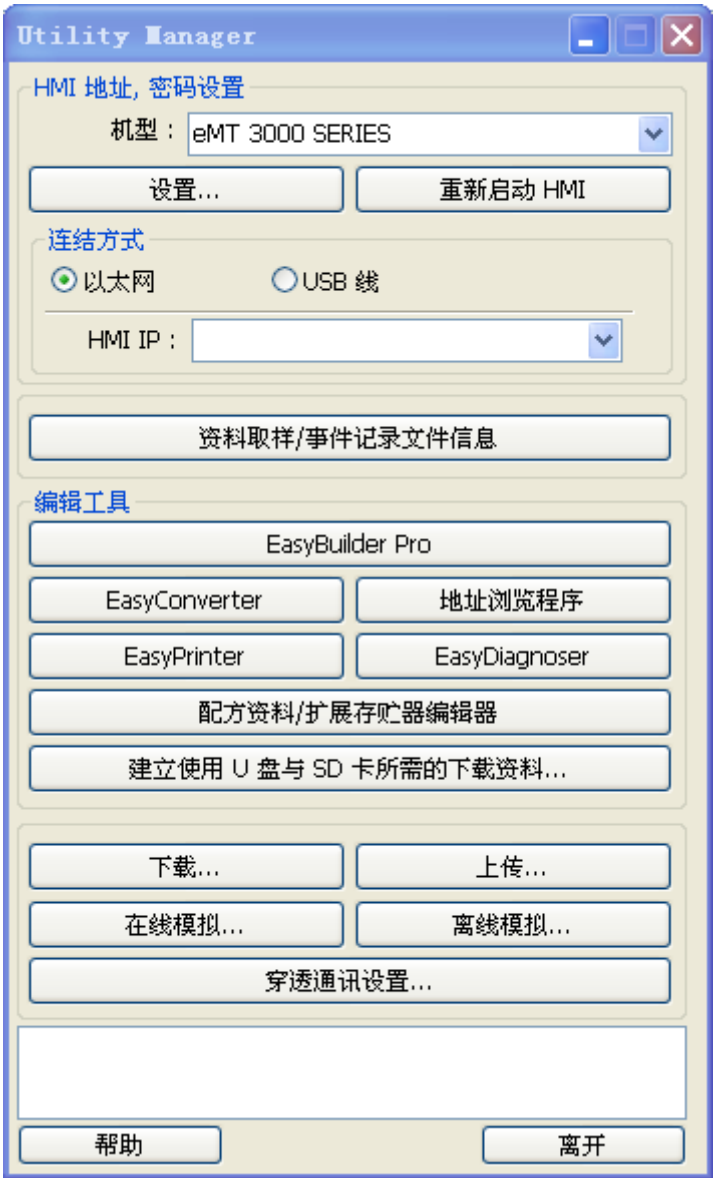
函数名称	TRACE
语法	TRACE(format, argument)
描述	一个执行中的宏指令可以使用此函数，监视变量内容的变化，并打印字符串，以协助除错。用户应开启 EasyDiagnoser 观看此函数的输出结果。

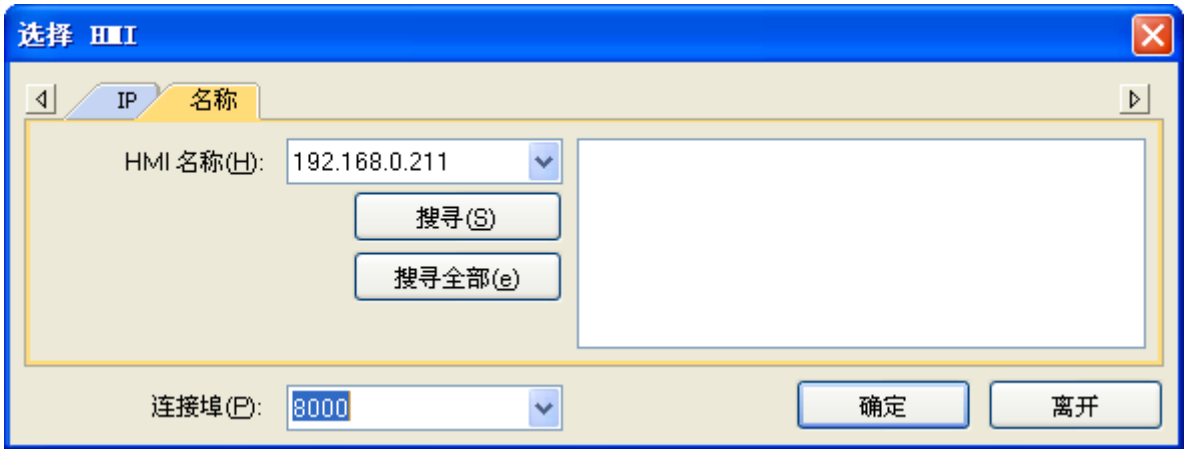
	当 TRACE 函数抓取一个 % 开头的特殊字符，将同时从 argument 抓取一个参数做格式化后输出。 format 代表打印格式，支持 % 开头的特殊字符。特殊字符格式如下，其中方括号内的字段为可选，粗体字字段为必需： %[flags] [width] [.precision] type 每个字段的意义如下所述： flags (可选)： - + width (可选)： 十进制正整数，指定应预留的字符宽度，不足部份补空格符。 precision (可选)： 十进制正整数，指定精确度，以及输出字符数。 type： C 或 c ： 以字符方式输出 d ： 以 signed 十进制整数输出 i ： 以 signed 十进制整数输出 o ： 以 unsigned 八进位整数输出 u ： 以 unsigned 十进制整数输出 X 或 x ： 以 unsigned 十六进制整数输出 E 或 e ： 以科学表示法输出 f ： 以单倍精确度浮点数输出 *format* 字符串最长支持 256 个字符，多出的字符将被忽略。 *argument* 部份可写可不写。但一个特殊字符应搭配一个变量。

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

8. 新增 LB-9059 -关闭宏指令 TRACE 功能 (当状态为 ON)。当设定为 ON 时，TRACE 将不会把数据输出到 EasyDiagnoser。

9. 也可以直接利用 Utility Manager 执行 EasyDiagnoser.exe，Utility Manager 将会显示网络上目前存在的 HMI，此时只要选择要监看通讯状态的 HMI 即可。请注意 Project Port 的部份应设定与工程文件所使用的“连接埠”相同。

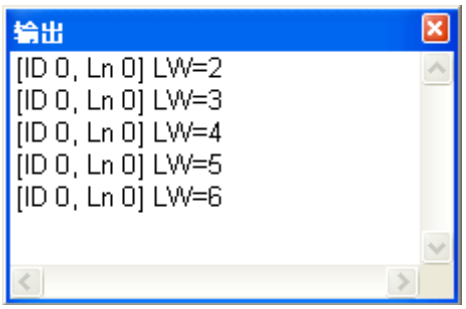




10. 将工程文件下载到触摸屏实际操作。当 EasyDiagnoser 无法连接上需要监视的触摸屏时，一般可能的原因是触摸屏未上电。或是“连接埠”不正确，可能发生 EasyDiagnoser 不断连上触摸屏又

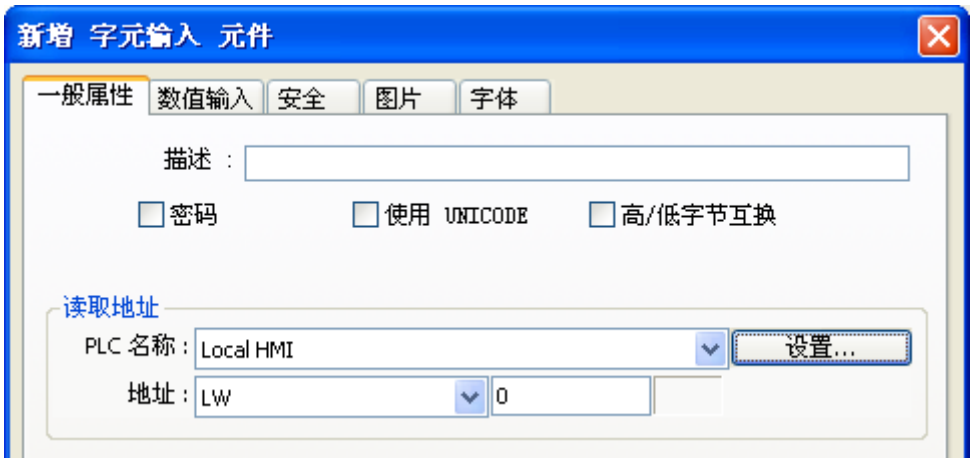
断线的情况。请确定 EasyDiagnoser 应设定与工程文件相同的 Port No.，更改方式如前面所述。

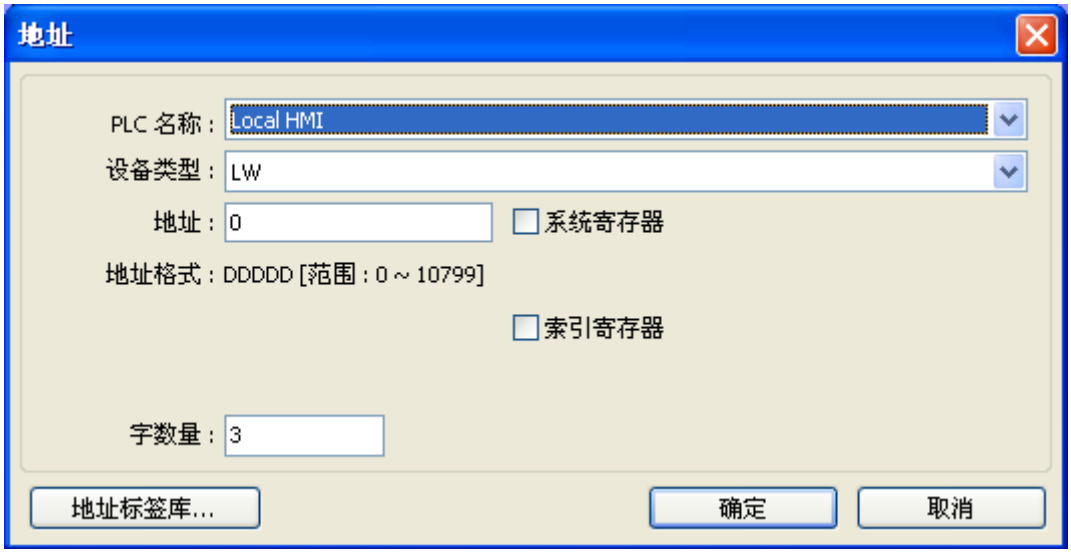
11. 当 EasyDiagnoser 成功与触摸屏连结后，只要执行 macro_0，即可发现“输出”窗口显示目前 TRACE 的执行结果。



18.15. 字符串处理函式使用方法

宏指令提供字符串处理函式，令用户可以很方便的对字符串进行各种操作。所谓字符串，是由一连串的 ASCII 字符组成，每一个 ASCII 字符占用 1 字节 (byte)，在 16 位寄存器中是以低字节优先方式储存。举例来说，我们利用一个“文字输入”元件在 LW-0 ~ LW-2 的位置，写入一条字符串 ”abcdef”，设定如下：





在此“文字输入”元件输入 ”abcdef”：

此字符串在 LW-0 ~ LW-2 中的存放方式如下图所示 (LB 代表低字节，HB 代表高字节)：

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

由于“文字输入”元件显示的数据长度单位为 word，而每个 ASCII 字符长度为一个 byte，所以每次最少会显示两个字符，此部份在“文字输入”元件的章节有详细的说明。因此上面的例子中，设定“文字输入”元件的字符数量为 3，表示最多可输入 / 显示 6 个 ASCII 字符。

目前提供的字符串处理函式，其功能整理如下表：

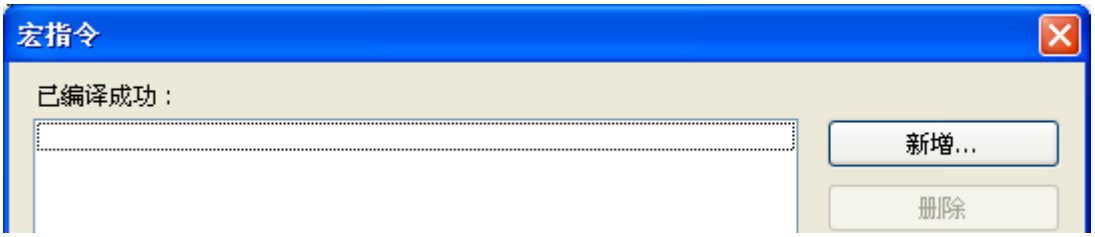
函数名称	描述
StringGet	获取 PLC 的字符串数据。
StringGetEx	获取 PLC 的字符串数据，不等待 PLC 响应，径自往下执行。
StringSet	将字符串数据写到 PLC 中。
StringSetEx	将字符串数据写到 PLC 中，不等待 PLC 响应，径自往下执行。
StringCopy	利用此函数进行字符串的复制。
StringMid	利用此函数可将一个字符串中的某一段子字符串提取出来。
StringDecAsc2Bin	此函数将十进制字符串转换成整数。
StringBin2DecAsc	此函数将整数转换成十进制字符串。
StringDecAsc2Float	此函数将十进制字符串转换成浮点数。
StringFloat2DecAsc	此函数将浮点数转换成十进制字符串。

StringHexAsc2Bin	此函数将十六进制字符串转换成整数。
StringBin2HexAsc	此函数将整数转换成十六进制字符串。
StringLength	取得字符串的长度。
StringCat	连接两字符串。
StringCompare	比较两字符串的内容是否相等。大小写视为不同。
StringCompareNoCase	比较两字符串的内容是否相等。大小写视为相同。
StringFind	在某个字符串中寻找另一个字符串。
StringReverseFind	在某个字符串中寻找另一个字符串，从后面开始找。
StringFindOneOf	寻找某字符串中任一个字符在另一个字符串中第一次出现的位置。
StringIncluding	从某字符串第一个字符开始提取包含特定字符的一段字符串。
StringExcluding	从某字符串第一个字符开始提取不包含特定字符的一段字符串。
StringToUpper	字符全部转换成大写。
StringToLower	字符全部转换成小写。
StringToReverse	字符串反转。
StringTrimLeft	裁剪字符串开头的特定字符。
StringTrimRight	裁剪字符串尾端的特定字符。
StringInsert	插入字符串到另一字符串中的特定位置。

上表中所有字符串处理函数的详细规格与使用范例请参考内置函数功能的章节。下面将举例说明如何从新增一个宏指令的方法开始，一直到执行模拟为止，带您一步步建立一个可执行的工程文件，以实际体会字符串处理函数强大的功能。

1. 以下说明如何从寄存器读入与写入一个字符串。

请新增 1 个宏指令：



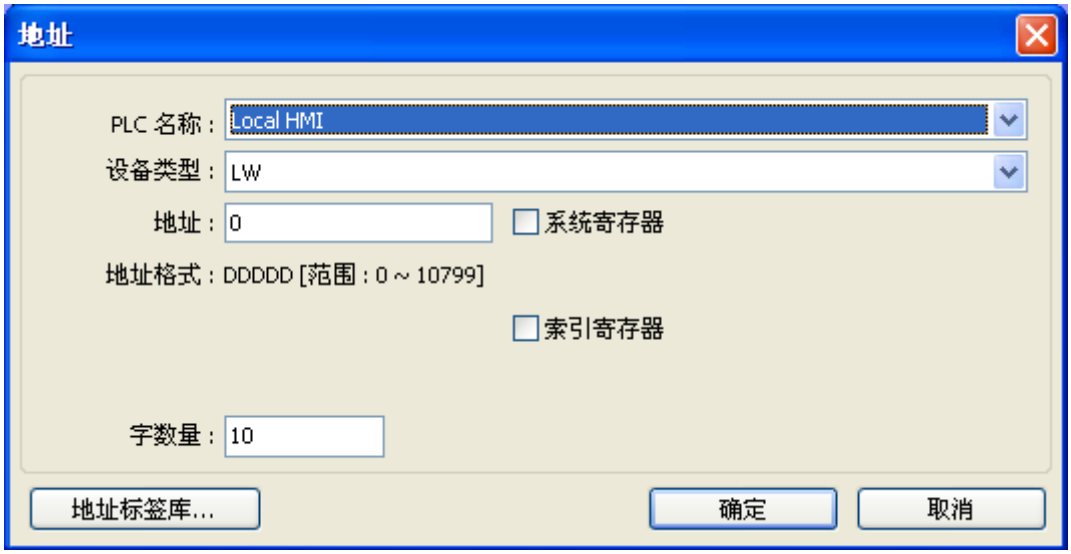
在宏指令编辑器编辑以下内容：

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

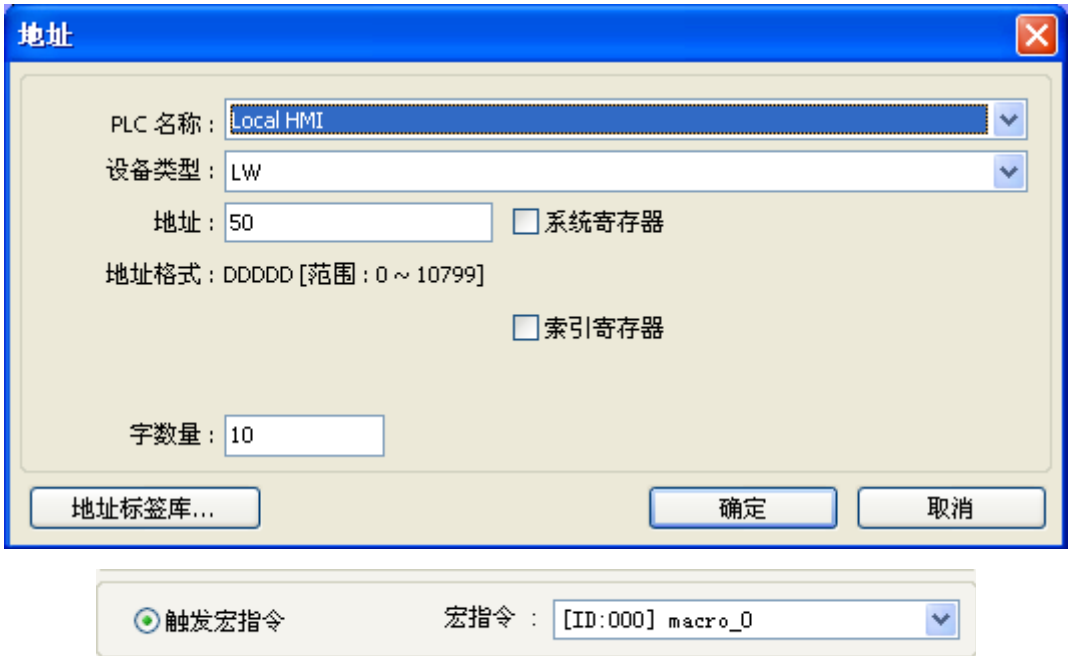
此宏指令的目的是利用 StringGet 函数从寄存器中读入一条字符串放入字符数组 str 中，并利用 StringSet 输出 str 的内容。

接着在工程文件的第 10 页分别加上“文字输入”与“功能键”元件，元件的设定内容请参考下图，”功能键”元件用来执行 macro_0。

“文字输入”元件



“功能键”元件



最后编译 已完成的工程文件并执行离线 (off-line) 或在线 (on-line) 模拟，并在模拟画面按照以下步骤操作：

Step 1. 输入字符串

Step 2. 按下“GO”按钮

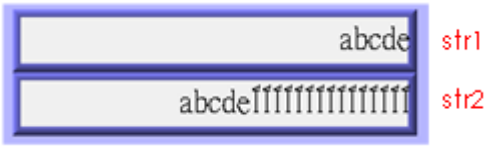
Step 3. 显示结果

2. 字符串之初始化。

在宏指令编辑器编辑以下内容：

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

用双引号（“ ”）刮起来的内容视为一个字符串。str1 是以字符串方式初始化，而 str2 是以字符数组方式初始化。



以字符串方式初始化时宏编译器会在字符串结尾后面加上结束符号 ’\0’ 代表字符串结束。利用 StringSet 将字符串写入寄存器时遇到结束符号便会停止继续写入数组后面的内容。即使 data count 被设为大于字符串长度的数值，显示字符串时只会显示结束符号之前的内容。

以字符数组方式初始化时数组内容并不会被视为一个字符串，因此也不会加上结束符号 ’\0’。写入寄存器的字符数会依据 data count 所设定的数值决定。

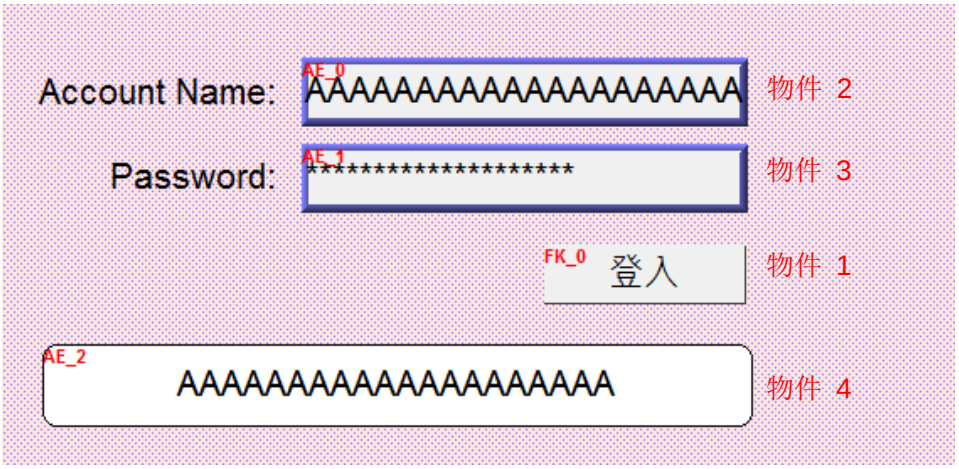
3. 一个简单的账号密码登入页面。

在宏指令编辑器编辑以下内容(宏指令 [ID:001] macro_1)：

```
1
2   macro_command main()
3
4   char name[20]="admin"
5   char password[20]="123456"
6   char name_input[20], password_input[20]
7   char message_success[40]="Success! Access Accepted."
8   char message_fail[40]="Fail! Access Denied."
9   char message_clear[40]
10  bool name_match=false, password_match=false
11
12  StringGet(name_input[0], "Local HMI", LW, 0, 20)
13  StringGet(password_input[0], "Local HMI", LW, 50, 20)
14
15  name_match = StringCompare(name_input[0], name[0])
16  password_match = StringCompare(password_input[0], password[0])
17
18  FILL(message_clear[0], 0x20, 40) //FILL with white space
19  StringSet(message_clear[0], "Local HMI", LW, 100, 40)
20
21  if (name_match==true and password_match==true) then
22      StringSet(message_success[0], "Local HMI", LW, 100, 40)
23  else
24      StringSet(message_fail[0], "Local HMI", LW, 100, 40)
25  end if
26
27  end macro_command
```

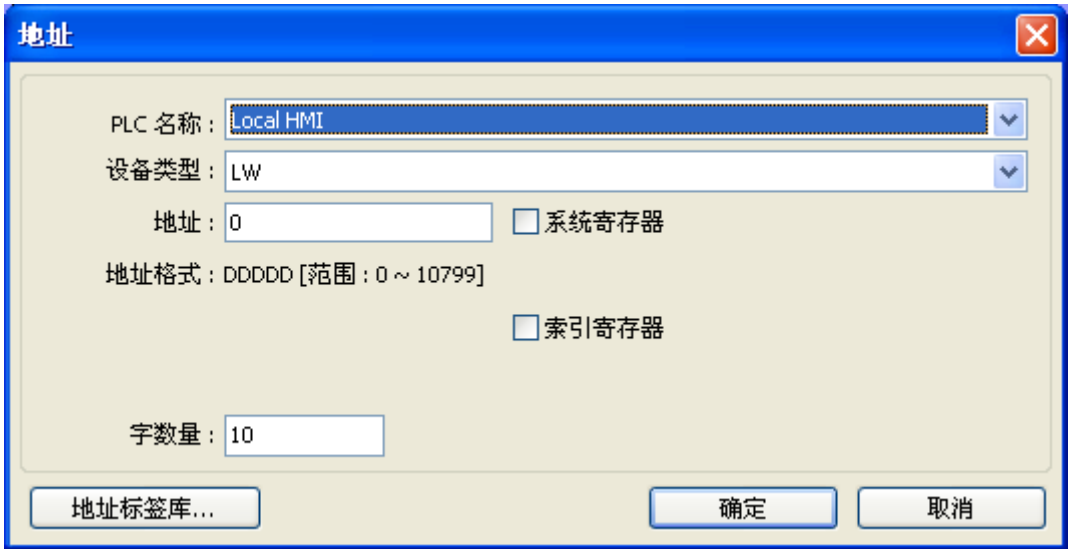
此宏将利用 “StringGet” 取得用户输入的账号密码的字符串，分别放入 name_input 与 password_input 两个数组中。接着利用 “StringCompare” 比对账号密码，若账号相符，则 name_match 设为 true；若密码相符，则 password_match 设为 true。最后检查 name_match 与 password_match 的值，若同时为 true，表示登入成功，并印出 ”Success! Access Accepted.” 字符串。若其中之一不相符，则印出 ”Fail! Access Denied.” 字符串。

接着在工程文件的第 10 页分别加上 “文字输入” 与 “功能键” 元件，元件的设定内容请参考下图，”功能键” 元件用来执行 macro_1。

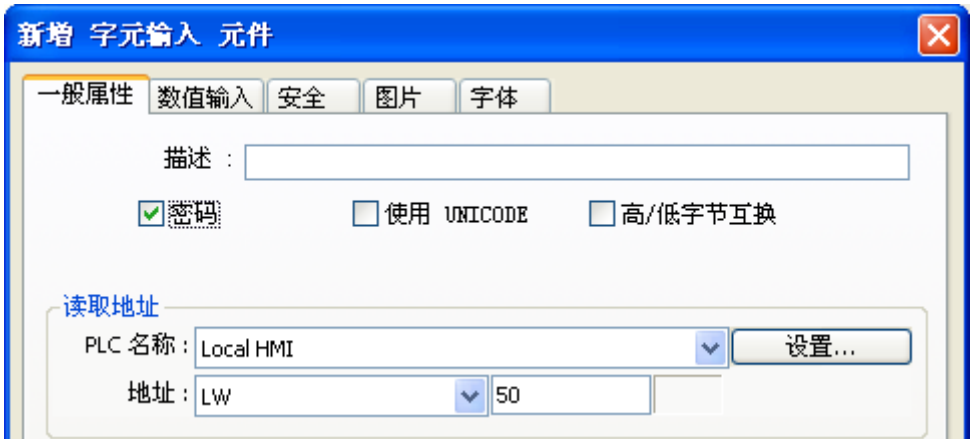


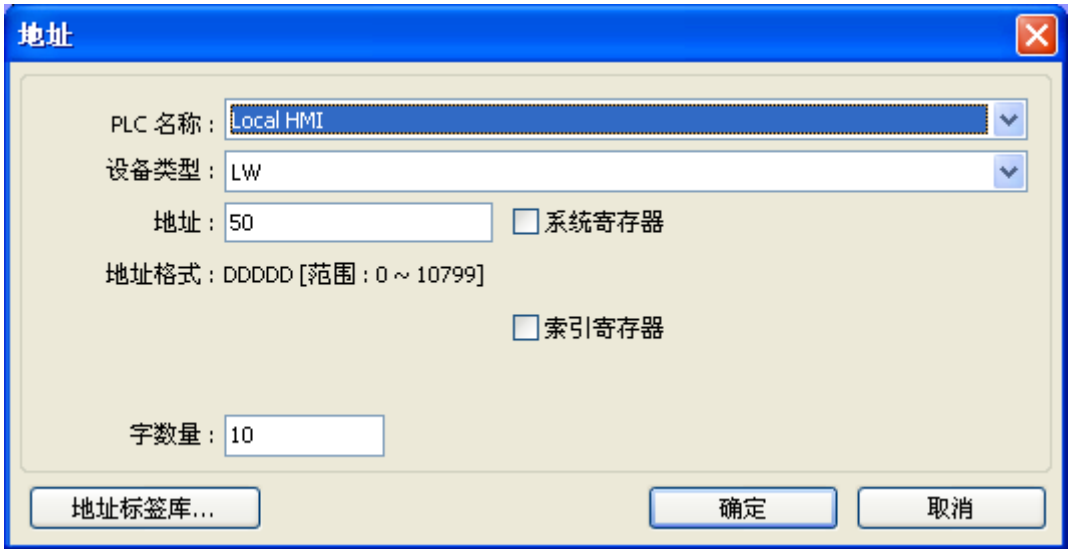
物件 1：”功能键”元件 ，点选“触发宏指令”并选择宏指令 [ID:001] macro_1。

物件 2：”文字输入”元件

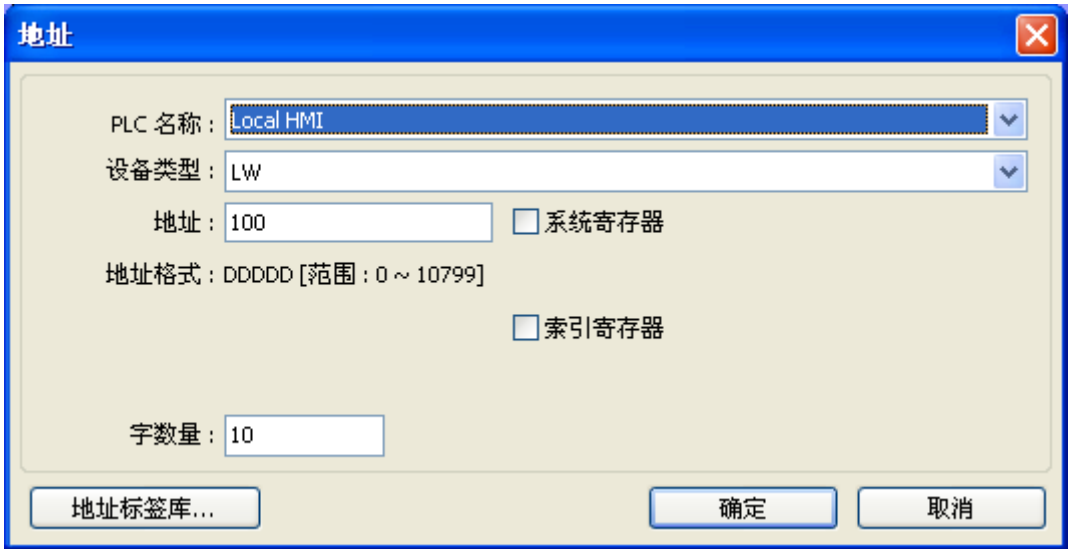


物件 3：”文字输入”元件



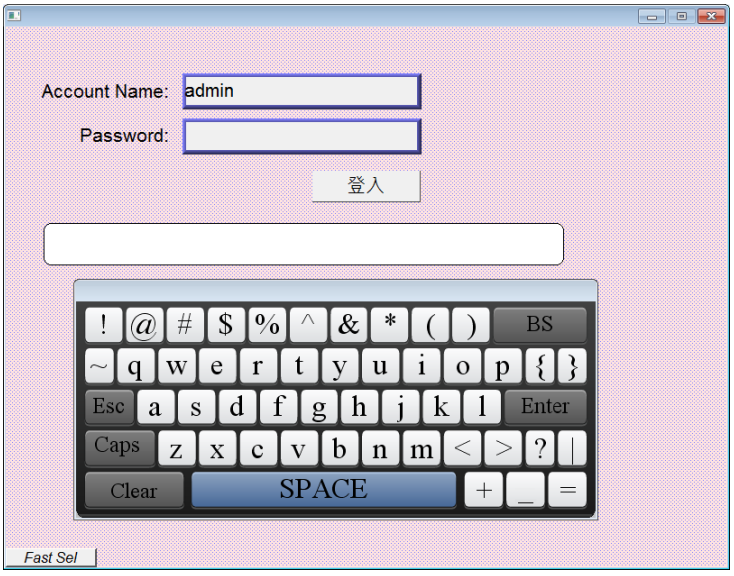


物件 4：”文字显示”元件

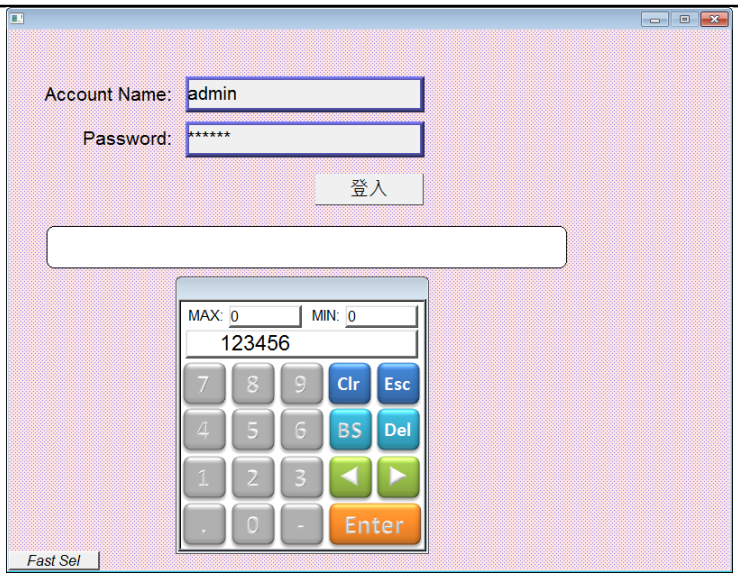


最后编译 已完成的工程文件并执行离线 (off-line) 或在线(on-line) 模拟，并在模拟画面按照以下步骤操作：

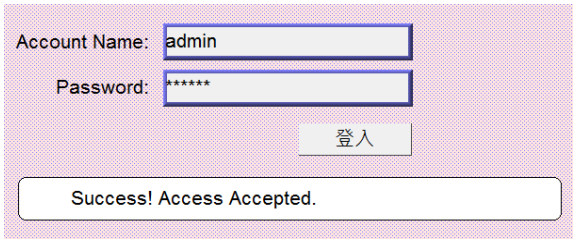
1. 输入账号。



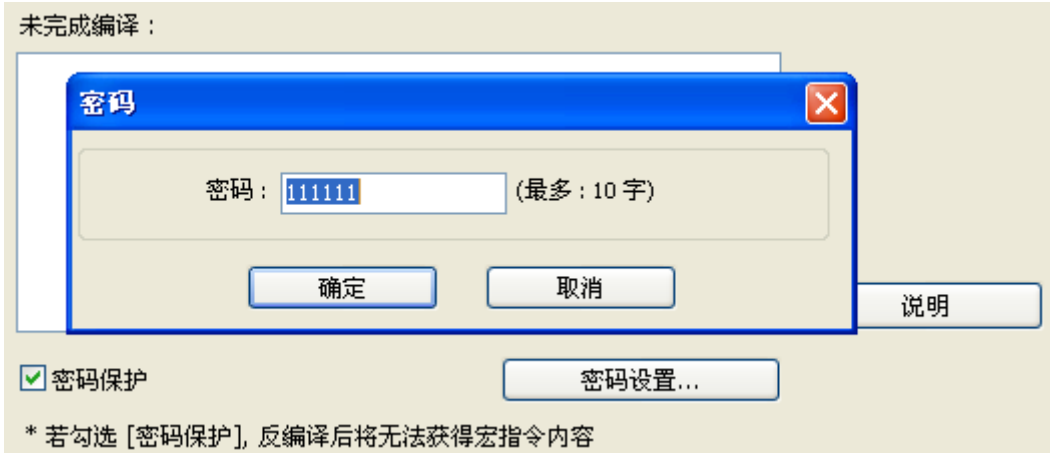
2. 输入密码并按下登入键。



3. 显示登入成功或登入失败。



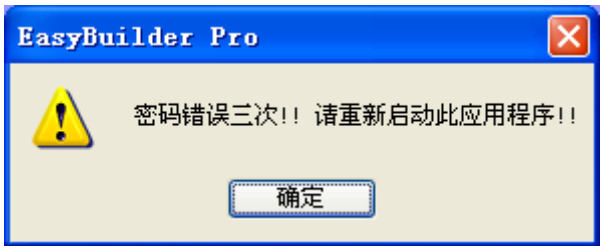
18.16. 宏指令密码保护



在宏指令编辑器窗口中提供“密码保护”选项，当勾选“密码保护”后按下“密码设置”按钮可以设定密码，密码最多不可超过 10 个字符（只支持 ASCII 文字，并区分大小写，例如可以输入 “a$#*hFds”）。

当设定宏指令“密码保护”功能后，用户欲开启宏指令编辑器窗口时，需先输入正确的密码。

当输入不正确的密码三次，需重新启动 EasyBuilder Pro，才能重新输入密码。



Note

- 当开启宏指令密码保护功能后，反编译 EXOB 文件将无法回复宏指令的内容。