深圳市雷赛控制技术有限公司
SHENZHEN LEADSHINE CONTROL TECHNOLOGY CO.,LTD

PMC600中型PLC编程手册（三）

编程语言篇

2021年06月

版权说明

目 录

第1章　PLC编程语言基础

1.1.　公共元素

PLC程序是由一定数量的基本语言元素（最小单元）组合而成的，使用者按照一定的规则把它们组合在一起以形成说明或语句。PLC编程语言包括分界符、关键字、标识符和注释等基本语言元素。

1.1.1.　　字符集

国家标准GB/T 15969.3-2005规定了可编程控制器使用的文本和图形类编程语言中的文本元素应根据国家标准GB/T 1988-1988字符集的“基本代码表”的3~7列字符组成，并根据GB2312-1980《信息交换用汉字编辑字符集-基本集》来表示汉字。

1.1.2.　　分界符

分界符用于分隔程序语言元素的字符或字符组合。它是专用字符，不同的分界符含义不同，表1.1中列出了各种分界符的应用示例。

表1.1　分界符

分界符	应用场合	备注或示例
空格	允许在PLC程序中插入一个或多个空格	
Tab	允许在PLC程序中插入Tab	
(*	注释开始	
*)	注释结束	
+	十进制的前缀符号（正数）	
	加操作符	
-	十进制的前缀符号（负数）	
	年、月、日的分隔符	
	减操作符	
#	基底数的分隔符	
	数据类型分隔符	
	时间文字的分隔符	
.	整数和小数的分隔符	
	分级寻址分隔符	
	结构元素分隔符	
	功能块结构分隔符	TON1.Q; SR_3.S1

1.1.3. 关键字

关键字是语言元素特征化的词法单元。在IEC61131-3标准中，关键字作为编程语言的字用于定义不同结构或特定的软件元素。

部分关键字需配对使用，如“FUNCTION”与“END_FUNCTION”等，部分关键字可单独使用，如“ABS”等。关键字不能用于任何其他目的，如不能作为变量名或扩展名，也就是说，既不能用TON作为变量名，也不能用VAR作为扩展名。

由于关键字是标准标识符，因此不能包含空格。PLC编程语言中常用的关键字和说明，如表1.2所示。

表1.2 常用的关键字和说明

关键字	说明	关键字	说明
PROGRAM END_PROGRAM	程序段开始 程序段结束	EN,ENO	使能输入/输出
FUNCTION END_FUNCTION	函数段开始 函数段结束	TRUE FALSE	逻辑真 逻辑假
FUNCTION_BLOCK END_FUNCTION_BLOCK	功能块段开始 功能块段结束	TYPE END_TYPE	数据类型段开始 数据类型段结束
VAR END_VAR	内部变量段开始 内部变量段结束	STRUCT END_STRUCT	结构体开始 结构体结束
VAR_INPUT END_VAR	输入变量段开始 输入变量段结束	IF THEN ELSIF ELSE END_IF	IF语句
VAR_OUTPUT END_VAR	输出变量段开始 输出变量段结束	CASE OF ELSE END_CASE	CASE语句
VAR_IN_OUT END_VAR	输入输出变量段开始 输入输出变量段结束	FOR TO BY DO END FOR	FOR循环语句
VAR_GLOBAL END_VAR	全局变量段开始 全局变量段结束	REPEAT UNTIL END_REPEAT	REPEAT循环语句
CONSTANT	常数变量	WHILE DO END_WHILE	WHILE循环语句
ARRAY OF	数组	RETURN	跳转返回值
AT	直接地址	NOT、AND、 OR、XOR	逻辑操作符

此外，下列功能模块和函数的标识符也被保留作为关键字。

（1）标准数据类型：BOOL、REAL或INT等。

（2）标准函数名和功能模块名：SIN、COS、RS或TON等。

（3）指令表语言中的文本操作符：LD、ST、ADD或GT等。

（4）结构化文本语言中的文本操作符：NOT、MOD或AND等。

1.1.4. 常数

数值文字用于定义一个数值，它可以是十进制或其他进制的数。数值文字分为两类：整数文字和实数文字，其书写格式如下。

<类型>#<数值>

<类型>：指定所需的数据类型。支持的类型有BOOL、SINT、USINT、BYTE、INT、UINT、WORD、DINT、UDINT、DWORD、REAL和LREAL。

<数值>：指定常数。输入的数据必须符合指定的数据类型<类型>。

（1）数字常数

数值常数可用二进制、十进制和十六进制数表示。加入一个整数不是十进制数，用户必须在该整数之前写出它的基数并加上数字符号（#），如二进制数101的表示方法为2#101。

BIN（二进制）

二进制的1位（bit）只能取0或1，用来表示开关量（或数字量）的两种不同的状态。例如，若QX0.0:=1,则表示线圈“通电”状态，称该编程单元为ON或1状态；若QX0.0:=0,则表示线圈“断电”状态，称该编程单元为OFF或0状态。

DEC（十进制）

十进制是以10为基础的数字系统，满10进1。十进制前缀10#可不必输入。例如，255，直接表示十进制数的255.系统默认采用十进制表示方式。

HEX（十六进制）

十六进制的16个数字由0~9和A~F（对应于10进制数10~15）组成，其计数规则为满16进1，每个数字占二进制数的4位。在iStudio中，16#作为十六进制数的前缀，如16#E5。

数值文字的表示和示例见下表，这些数值可以是变量类型BYTE、WORD、DWORD、SINT、USINT、INT、UINT、DINT、REAL或LREAL。

表1.3 数值文字的表示和示例

数值文字类型	表示方法	示例
整数	[整数类型名#]符号整数或二（八、十、十六）进制整数	INT#-34,UINT#32
	二进制整数	2#0010_0010(34)
	八进制整数	8#77(63)
	十六进制整数	16#E5(229)
实数	[实数类型名#]符号整数.整数[指数]	REAL#4.94,6.3e-7
	符号整数.整数	3.14159
布尔数	[布尔类型名#]0或1,False或True	0,1,True,BOOL#1

（2）BOOL常数

BOOL常数为逻辑值TRUE（真）与逻辑值FALSE（假），也可以用1（真）和0（假）表示，其含义相同。

（3）TIME常数

TIME常数由一个起始符T或t（或者用TIME或time）和数字标识符#，再加上实际时间表示，包括天（d）、时（h）、分（m）、秒（s）和毫秒（ms）。注意，时间各项必须根

据时间长度单元顺序进行设置（d在h之前，h在m之前，m在s之前，s在ms之前），但无需包含所有的时间长度单位。

在结构化文本编程语言的赋值语句中，正确使用时间常数的示例如下：

TIME1 := t#14ms;

TIME1 := T#100S12ms(*最高单位的值可以超出其限制*)

TIME1 := T#12h34m15s

（4）DATE常数

这些常数可用来表示日期。声明一个DATE常数时，起始符为d、D、DATE或date后跟随一个#号，然后就可按照“年-月-日”的格式表示任何日期。例如：

DATE#1996-05-06

d#1972-03-29

（5）TIME_OF_DAY常数

使用这种类型常数可以保存一天中的不同时间。TIME_OF_DAY常数的圣铭使用起始符tod#、TOD#、TIME_OF_DAY#或time_of_day#，后面跟随一个时间格式为“时：分：秒”的时间。秒值可以是整数也可以小数，如：

TIME_OF_DAY#15:36:30.123

tod#00:00:00

（6）DATE_AND_TIME常数

日期常数和一天中的时间常数可合并起来构成一个DATE_AND_TIME常数。DATE_AND_TIME常数的起始符位dt#、DT#、DATE_AND_TIME#或date_and_time#，然后在日期之后用“-”字符连接时间，如：

DATE_AND_TIME#1996-05-06-15:36:30

Dt#1972-03-29-00:00:00

（7）REAL/LREAL常数

REAL和LREAL常数可以使用十进制小数和指数形式表示，使用带小数点的格式表示实数（REAL/LREAL），如：

7.4取代7,4

1.64e+009取代1,64e+009

（8）STRING常数

字符串是一个字符队列，STRING（字符串）常数使用一个单引号作为其前缀和后缀，也可以输入空格和专用字符，这些字符将同所有其他字符一样进行处理，字符的表达式示例如下：

‘CODESYS!!’

‘Leadshine’

‘:-)’

1.1.5. 句法颜色

所有编辑器中不同的文本带有不同的颜色，不但可以辨识错误而且有助于快速地发现错误。例如，注释没有被括上，从字体颜色上就会立即得到提示。句法颜色示例如图1.1所示。

- 蓝色：关键字
- 绿色：编辑器中的注释
- 粉红：特殊的常数（如 TRUE/FALSE、T#3s、%IX0.0）。
- 红色：输入错误（如无效时间常数、小写的关键字）
- 黑色：变量、常数、标点符号

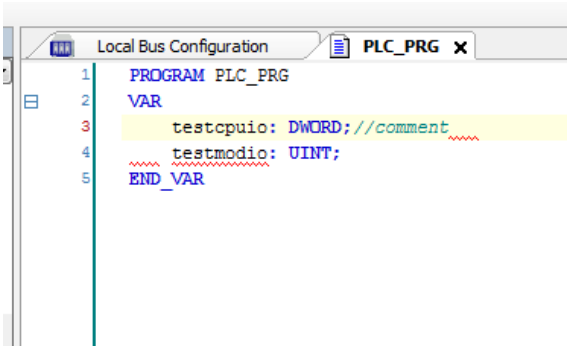


图1.1 句法颜色实例

1.2. 变量的表示和声明

变量可以用来表示一个数值、一个字符串值或一个数组等。CODESYS将变量的数据类型分为标准数据类型、扩展数据类型以及自定义数据类型。

1.2.1. 变量

变量是保存在存储器中待处理的抽象数据，是为了识别PLC的输入/输出、PLC内部的存储区域而使用的名称，可以代替物理地址在程序中使用。可以根据需要随时改变变量中所存储的数据值。在程序执行过程中，变量的值可以发生变化。使用变量之前必须先声明变量，以及指定变量的类型和名称。变量具有名称、类型和值。变量的数据类型确定它所代表的内存大小和类型。变量名即指在程序源代码中的标识符。

1.2.2. 标识符

标识符就是变量的名称。在定义标识符时，根据IEC61131-3标准，必须由字母、数字和下划线字符组成。此外，还应遵循如下规则：

- 标识符的首字母必须是字母或下划线，最后一个字符必须是字母或数字，中间允许是字母、数字、下划线

- 标识符中不区分字母的大小写
- 下划线是标识符的一部分，单标识符中不允许有两个或两个以上连续的下划线。
- 不得含有空格。

例如，ab_c，AB_de和AbC是允许的标识符，而1abc、__abc和a__bc均不允许。

1.2.3. 变量的声明

变量声明就是制定变量的名称、类型、以及赋初始值。变量的声明非常重要，未经声明的变量是不能通过编译的，也无法在程序中使用。用户可以在程序组织单元（POU）、全局变量列表（GVL）和自动声明对话框中进行变量的声明。在CODESYS中，变量声明分为两类：普通变量声明和直接变量声明。

普通变量声明：普通变量声明是最常用的变量声明，不需要和硬件外设或通信进行关联的变量，仅供项目内部逻辑使用。普通变量声明需要符合以下规则：

<标识符>：<数据类型>｛:=<初始值>｝；

｛｝中的部分是可选部分。例如nTest:BOOL;nTest:BOOL:=TRUE;

直接变量声明：当需要和PLC的IO模块进行变量映射或和外部设备进行通信时，需要采用此声明方法。使用关键字AT把变量直接连接到确定的地址。直接变量声明需要符合以下规则：

AT<地址>

<标识符>AT<地址>:<数据类型>｛:=<初始化值>｝；

{}中的部分是可选部分。

使用“%”开始，随后是位置前缀符号和类型前缀符号，如果有分级，则用整数表示分级，并用小数点符号“.”表示，如%IX0.0、%QW0。直接变量声明具体的格式如图1.2所示。

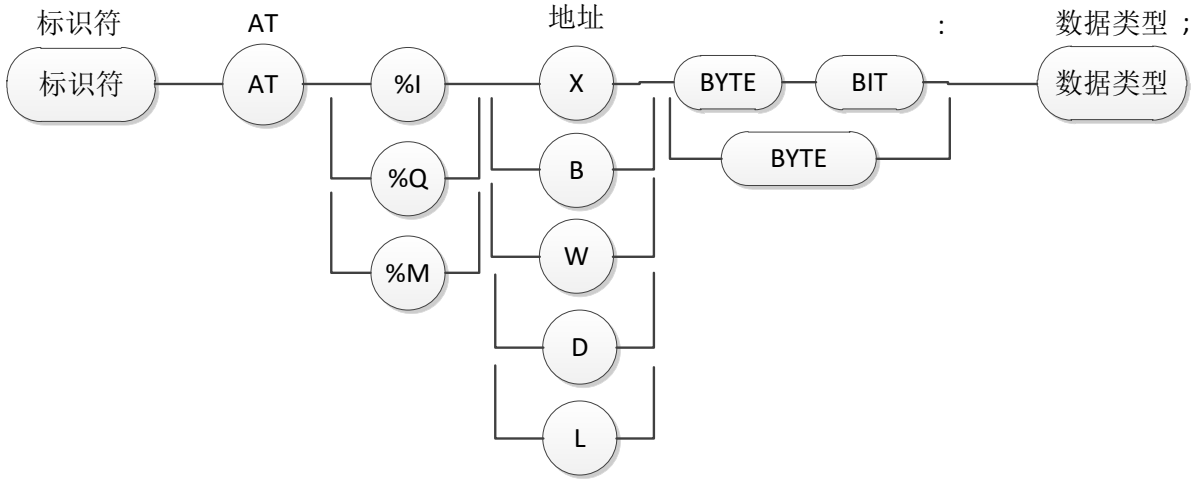


图1.2 直接变量声明的具体格式

图1.2中的位置前缀的定义如下：

（1）I：表示输入单元。

（2）Q：表示输出单元。

（3）M：表示存储区单元。

类型前缀的定义如表1.4所示。

表1.4　类型前缀的定义

前缀符号	定义	约定数据类型
X	位（BIT）	BOOL
B	字节（BYTE）	BYTE
W	字（WORD）	WORD
D	双字（DWORD）	DWORD
L	长字（LWORD）	LWORD
*	未指定位置的内部变量，系统自动分配	

下面举例做进一步的说明：

```
VAR
  VAR1  AT%ID48:DWORD;
END_VAR
```

%I 说明了该变量输入单元，具体的地址为%ID48。以下为内存映射的距离，系统会根据数据类型的大小（X:bit、B:byte、W:word、D:dword）来进行自动分配。

在该地址内存映射表中，以%IW和%ID举例,由于WORD是DWORD类型的数据长度的一半，因此第48个DWORD单元地址等于第96个和第97个字单元综合。同理，字节地址%IB192 ~ %IB195这4个字节变量组合后对应%ID48，一个双字等于4字节，故48X4后对应的字节首地址正好为192。

表1.5　内存映射表

%IX	192.0 – 192.7	193.0 – 193.7	194.0 – 194.7	195.0 – 195.7
%IB	192	193	194	195
%IW	96		97	
%D	48			

同样的方法，就很容易理解如下的地址映射关系。

(1) %MX12.0：%MB12的第0位。

(2) %IW4：表示输入字单元4（字节单元8和9）。

(3) 3%IX1.3：表示输入第1字节单元的第3位。

1.3.　数据类型

无论声明的是变量还是常量，都必须使用数据类型。数据类型的标准化是编程语言开放性的重要标志，CODESYS的数据类型完全符合IEC61131-3所定义的标准，它将数据类型分为标准数据类型、IEC61131-3标准的扩展数据类型和自定义数据类型。数据类型决定了它将占用多大的存储空间及将存储何种类型的值。

1.3.1. 标准数据类型

标准数据类型共分为5大类，分别为布尔类型、整型类型、实数类型、字符串类型和时间数据类型。表1.6列举了支持的标准数据类型。

表1.6 标准数据类型表

数据大类	类型名称	关键字	数据下限	数据上限	数据宽度
布尔	布尔	BOOL	0	1	1bit
整型	字节	BYTE	0	255	8bit
	字	WORD	0	65535	16bit
	双字	DWORD	0	4294967295	32bit
	长字	LWORD	0	2^{64} - 1	64bit
	短整型	SINT	-128	127	8bit
	无符号短整型	USINT	0	255	8bit
	整型	INT	-32768	32767	16bit
	无符号整型	UINT	0	65535	16bit
	双整型	DINT	-2147483648	2147483647	32bit
	无符号长整型	UDINT	0	4294967295	32bit
	长整型	LDINT	-2^{63}	2^{63}-1	64bit
实数	实数型	REAL	1.175494351e-38	3.402823466e+38	32bit
	实数型	LREAL	2.225073855072014e-308	1.7976931348623158e+308	64bit
时间数据	时间型	TIME	T#0ms	T#71582m47s295ms	32bit
	时刻型	TIME_OF_DAY	TOD#0:0:0	TOD#1193:02:47.295	32bit
	日期型	DATE	D#1970-1-1	D#2106-02-06	32bit
	日期时刻型	DATE_AND_TIME	DT#1970-1-1-0:0:0	DT#2106-02-06-06:28:15	32bit
字符串	字符串型	STRING			8 X N

1.3.2. 标准的扩展数据类型

作为对IEC61131-3标准中数据类型的补充，CODESYS隐含一些标准的扩展数据类型，有联合体、长时间、款字节字符串、引用和指针，如表1.7所示。

表1.7　标准扩展数据类型

数据大类	数据类型	关键字	位数	取值范围
联合	联合体	UNION		自定义
时间数据	长时间	LTIME	64	纳秒~天
字符串	宽字节字符串	WSTRING	16 X (N+1)	
引用	引用	REFERENCE TO		自定义
指针	指针	POINTER TO		自定义

（1）联合体

有时需要将几种不同类型的变量存放戴套同一段内存单元中，如可以把一个INT型变量、一个BYTE型变量和一个DWORD型变量放在同一地址开始的内存单元中，表1.8所示，从同一地址16#100开始存放。

表1.8　联合体

	16#100	16#101	16#102	16#103
INT				
BYTE				
DWORD				

表1.8中，16#100~16#103为覆盖区域，这种使用几个不同的变量共占同一段内存的结构称为联合体。

联合体声明的语法如下：

```
TYPE<联合体名>:
    UNION
    <变量的声明1>
    …
    …
    <变量的声明n>
    END_UNION
END_TYPE
```

以下以实际案操作来说明在iStudio中联合体操作的过程。

首先，在数据单元类型中新建一个名为NAME的联合体，右键单击“Application”，在弹出的快捷菜单中选择“添加对象”→“DUT”，弹出如图所示的“添加DUT”对话框，输入名称并单击选中“联合”按钮，单击“打开”按钮，然后在联合编辑器中输入以下内容：

```
TYPE NAME :
    UNION
        VAR1:STRING(20);
        VAR2:STRING(20);
        VAR3:STRING(20);
      END_VAR
    END_UNION
END_TYPE
```

在程序中编写的代码如下：

```
VAR
    nName:NAME;
END_VAR

nName.VAR1 := 'Leadshine'
```

最终输出的结果如上所示，虽然只给了其中的一个成员进行了var1赋值，但是从结果中可以看到，nName中所有成员的数值都被统一地修改为’Leadshine’。此外，联合体内的成员数据类型可以不一样。

（2）引用

引用是一个对象的别名，这个别名可以通过标识符读写。与指针不同的是，引用所指向的数据将被直接改变，因此引用的赋值和所指向的数据是相同的。设置引用的地址用一个特定的赋值操作完成（REF=），一个引用是否指向一个有效的数据（不等于0），可以使用一个专门的操作符(__ISVALIDREF)来检查，如下所示。

用以下语法声明引用：

语法

<标识符> : REFERENCE TO <数据类型>

声明示例：

```
ref_int : REFERENCE TO INT;
a : INT;
b : INT;
```

此时ref_int可以作为整型变量的别名使用

使用示例：

```
ref_int REF = a;    (* 此时ref_int指向a *)
ref_int := 12;      (* 此时a的值为12 *)
b := ref_int * 2;   (* 此时b的值为24 *)
ref_int REF= b;    (* 此时ref_int指向b *)
ref_int := a / 2;     (* 此时b的值为6 *)
ref_int REF= 0;    ( *引用的显式初始化 *)
```

注　意：不可声明诸如REFERENCE、ARRAY和POINTER的引用。

有效引用的检查：

操作符“__ISVALIDREF”可用来检查引用是否指向一个不等于0的有效值。

语法：

<boolean variable> := __ISVALIDREF(<datatype>声明为REFERENCE类型);

如果引用指向一个有效值，则<boolean variable>为真（TRUE），否则为假（FALSE）。

示例：

声明:

```
ivar : INT;
ref_int : REFERENCE TO INT;
ref_int0: REFERENCE TO INT;
```

```
testref: BOOL := FALSE;
testref0: BOOL := FALSE;

实现:
ivar := ivar +1;
ref_int REF= ivar;
ref_int0 REF= 0;
testref := __ISVALIDREF(ref_int);     (* 为TRUE，因为ref_int指向ivar，它不等于0 *)
testref0 := __ISVALIDREF(ref_int0);  (* 为FALSE，因为ref_int0被设为0 *)
```

（3）指针

指针用来在应用程序运行时存储变量、程序、功能块、方法和函数的地址。它可以指向上述的任何一个对象以及任意数据类型，包括用户定义数据类型。请注意，可以使用隐含的指针校验功能。

语法:

<标识符>: POINTER TO <数据类型|功能块|程序|方法|函数>;

取指针地址内容即意味着读取指针当前所指地址中存储的数据。通过在指针标识符后添加内容操作符“^”，可以取得指针所指地址的内容： 请看下面“pt^”的使用示例。

通过地址操作符ADR可以将变量的地址赋给指针。

示例：

```
VAR
pt:POINTER TO INT;  (* 声明一个指针pt *)
var_int1:INT := 5;     (* 声明变量var_int1和var_int2 *)
var_int2:INT;
END_VAR
pt := ADR(var_int1);   (* 将varint1的地址赋给指针pt *)
var_int2:= pt^;       (* 通过取指针pt的地址内容，将var_int1的值5赋给var_int2 *)
```

函数指针

这类指针可以指向外部库，但不可以在编程系统的应用程序内部调用函数指针！注册回调函数（系统库函数）的运行时函数需要函数指针，之后根据注册所需的回调函数的不同，各自的函数将由运行时系统隐式调用（例如STOP）。若需要使能这样的系统调用（运行时系统），必须为函数对象设置各自的属性(类别“编译”)。

ADR运算符可以用于函数名、程序名、功能块名和方法名。由于函数在在线修改之后有可能被移动，因此运算结果不是函数的地址，而是指向函数的指针的地址。只要函数存在，这一地址就将一直有效。

指针的索引访问：作为IEC 61131-3标准的扩展，允许对POINTER、STRING和WSTRING等类型的变量进行索引访问“[]”。

pint[i]返回被指向的数据类型。

指针的索引访问的算法：如果对指针变量进行索引访问的操作，则偏移量将按照下式

计算：pint[i] = (pint + i *被指向的数据类型的长度)^。索引访问也隐含了获取指针地址内容的操作。所得结果的数据类型是指针所指的数据类型。请注意：pint[7] != (pint + 7)^!

如果对STRING类型的变量进行索引访问的操作，则会得到索引表达式偏移量处的字符。所得结果是BYTE类型。str[i]将会以SINT的形式（ASCII编码）返回字符串中的第i个字符。

如果对WSTRING类型变量进行索引访问的操作，则会得到索引表达式偏移量处的字符。所得结果是WORD类型。wstr[i]将会以INT形式（Unicode编码）返回字符串中的第i个字符。

注　意：引用与指针不同，它们将直接修改所指的数据。

1.3.3. 自定义数据类型

（1）数组

一维、二维和三维数组属于基本的数据类型。可以在POU的声明部分或者全局变量表中定义数组。请注意可以使用隐含的边界检查。

语法：

<数组名>:ARRAY [<l11>..<u11>,<l12>..<u12>,<l13>..<u13>]OF <基本数据类型>

l11,l12,l13表示字段范围的最小值，u11,u12,u13表示字段范围的最大值。字段范围必须是整数。

例如：

```
Card_game: ARRAY [1..13, 1..4] OF INT;
```

数组的初始化：

数组的完全初始化举例：

```
arr1 : ARRAY [1..5] OF INT := [1,2,3,4,5];
arr2 : ARRAY [1..2,3..4] OF INT := [1,3(7)]; (* 即1,7,7,7的缩写形式 *)
arr3 : ARRAY [1..2,2..3,3..4] OF INT := [2(0),4(4),2,3]; (* 即0,0,4,4,4,4,2,3的缩写*)
```

结构数组的初始化举例：

结构定义：

```
TYPE STRUCT1
STRUCT
p1:int;
p2:int;
p3:dword;
END_STRUCT
END_TYPE
```

数组初始化：

```
ARRAY[1..3] OF STRUCT1:= [(p1:=1,p2:=10,p3:=4723),(p1:=2, p2:=0,p3:=299),
(p1:=14,p2:=5,p3:=112)];
```

数组的部分初始化举例：

```
arr1 : ARRAY [1..10] OF INT := [1,2];
```

对于那些没有预先赋值的元素，按照基本数据类型的缺省初始值进行初始化。在上例

中，元素[3]到[10]被初始化为0。

数组元素的访问：

在二维数组中访问数组元素，使用下面的语法：

<数组名>[Index1,Index2]

例如:Card_game [9,2]

（2）结构体

结构体是将不同类型的数据组合成一个整体以便于引用。结构体（struct）就是由一系列具有相同类型或不同类型的数据构成的数据集合。如表1.9所示，一台电机通常都有其对应的信息，如产品型号（Product_ID）、生产厂家（Vendor）、额定电压（Nominal Voltage）、额定电流（Nominal Current）、极性（Poles）、刹车（Brake）等信息。这些信息都与这个电机相关联，从下表可以看出来，如果将这些信息分别以独立的变量进行声明，很难反应出它们和这个电机相关联，如果有一种数据结构类型可以将它们组合起来的话，就可以解决这个问题，在CODESYS中，结构体就能起到这样的作用。

表1.9　电机产品信息参数

Product_ID	Vendor	Nominal Voltage	Nominal Current	Poles	Brake
11000	Leadshine	220	3	4	YES

结构体可以包含其他基础数据结构类型，如INT、STRING等。结构体还可以设计成用户想要的数据类型以便后续的调用。在实际项目中，结构体是大量存在的，工程人员常使用结构体来封装一些属性以组成新的类型。因此，在项目中通过对结构体内部变量的操作将大量的数据存储在内存中，以完成对数据的存储和操作。结构体最主要的作用就是封装，封装的好处就是可以再次利用。

结构体以关键字TYPE和STRUCT开始，并以END_STRUCT和END_TYPE结束。

结构声明的语法：

```
TYPE <结构名>:
    STRUCT
      <变量的声明1>
      ……
      <变量声明n>
    END_STRUCT
END_TYPE
```

<结构名>是一种可以在整个工程中被识别的数据类型，可以像标准数据类型一样使用。结构体可以实现嵌套，如图1.3所示，其中子结构也可以是一个结构体。

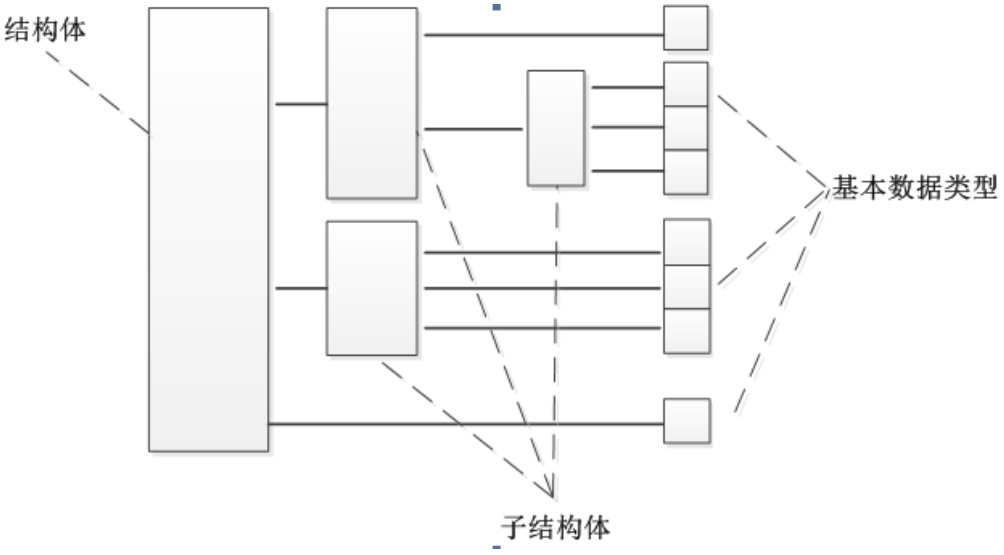


图1.3　数据结构体

添加结构体：

右键单击“Application”，在弹出的快捷菜单中以此选择“添加对象”→ “DUT…”，如图1.4所示。

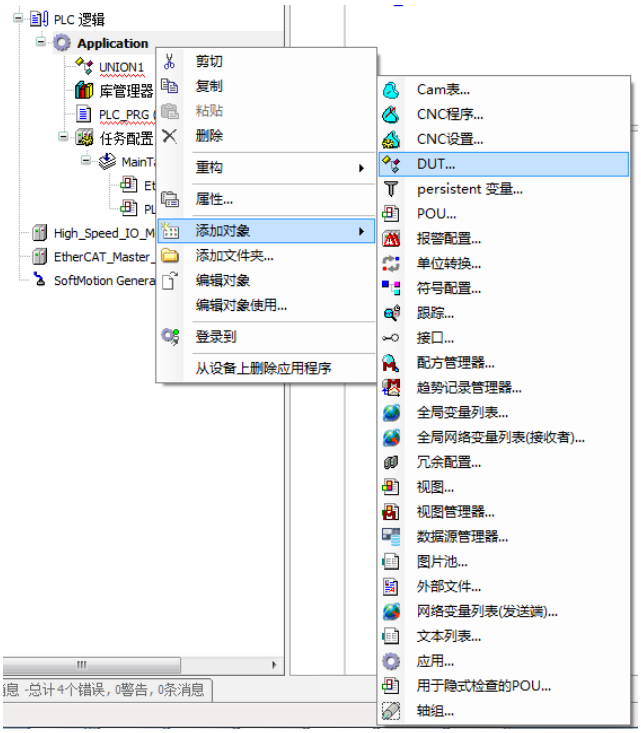


图1.4　添加结构体

在“添加DUT”对话框中输入结构体名称，并单击选中“结构（S）”单选按钮，然后单击“打开”按钮，如图所示。然后，系统会自动进入DUT编辑器，如图1.5所示。

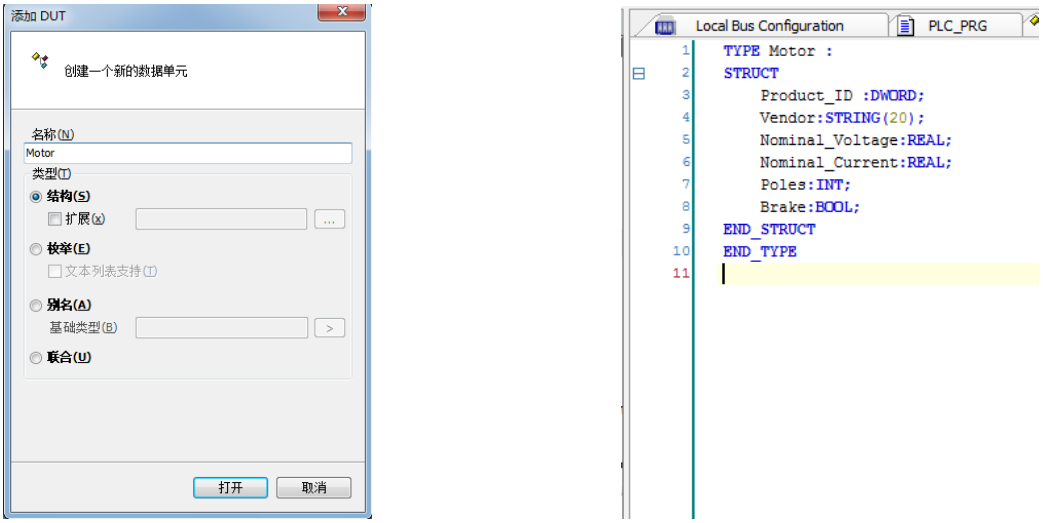


图1.5　编辑结构体

建立完结构体后，只需要在程序中新建一个变量，类型为刚刚建立的“结构体名”，即Motor。在程序中输入“变量名.”后，系统会自动弹出结构体内具体对应的信息，如图1.6所示，通过单击鼠标选择，再配合赋值语句即可实现对结构体的读写操作。

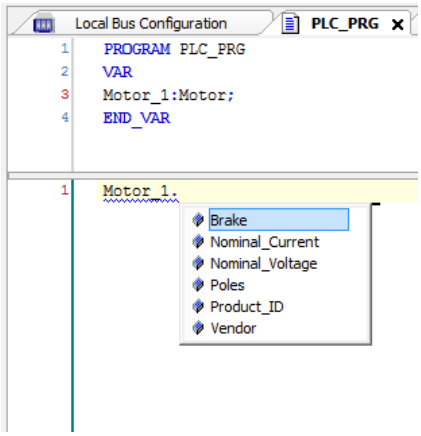


图1.6　修改结构体参数

本例在数据单元类型中新建一个名为Motor的结构体，具体代码如下：

```
TYPE Motor :
STRUCT
  Product_ID :DWORD;
  Vendor:STRING(20);
  Nominal_Voltage:REAL;
  Nominal_Current:REAL;
  Poles:INT;
  Brake:BOOL;
END_STRUCT
END_TYPE
```

根据表1.9所示的内容给结构图赋值：

```
VAR
Motor_1:Motor;
END_VAR

Motor_1.Product_ID:=11000;
Motor_1.Vendor:='Leadshine';
Motor_1.Nominal_Voltage:=220;
Motor_1.Nominal_Current:=3;
Motor_1.Poles:=4;
Motor_1.Brake:=TRUE;
```

结构体变量的存储结构：结构体变量从双字节边界对齐开始，也就是说，起始地址为偶数字节地址。随后，每个结构元素以其声明时的顺序依次存储到内存中。

数据类型为BOOL、BYTE的结构体元素从偶数字节开始存储，其他数据类型的数组元素从字地址开始存储。

（3）结构体数组

一个结构体变量可以存放一组数据（如一个设备的安装位置、温度和湿度等数据）。如果有10个设备的数据需要参加运算，显然应该用数组，这就是结构体数组。结构体数组与数值型数组不同，每个数组元素都是一个结构体类型的数据，他们分别包括各个成员（分量）项。

定义结构体数组和定义结构体变量的方法相似，只需要说明其为数组即可，例

```
STRUCT
  sDeviceName:STRING;
  nInstallLocation:INT;
  bWorkStatus:BOOL;
  rHumidity:REAL;
rTemperature:REAL;
END_STRUCT
END_TYPE

VAR
arrMachineStatus:ARRAY[0..2]OF Machine;
END_VAR
```

以上定义了一个数组arrMachineStatus，数组中有3个元素，均为Machine类型数据，如表1.10所示。

表1.10　结构体数组arrMachineStatus

	sDeviceName	nInstallLocation	bWorkStatus	rHumidity	rTemperature
arrMachineStatus[0]	Cutting	101	TRUE	70	23.3
arrMachineStatus[1]	Sorting	102	FALSE	72	26.1
arrMachineStatus[2]	Packing	103	TRUE	71.5	24.1

（4）枚举

如果一种变量有几种可能的值，就可以定义为枚举类型。所谓“枚举”试讲变量的值一一列举出来，变量的值只能在列举出来的值的范围。例如定义一个变量，该变量的值表示一星期的一天。该变量只能存储7个有意义的值。若要定义这些值的，可以使用枚举类型。例如，一星期内可能取值的集合为：

{Sun,Mon,Tue,Wed,Thu,Fri,Sat}

该集合可定位描述星期的枚举类型，该枚举类型共有7个元素，因而用枚举类型的枚举便令只能取集合中的某一元素值，枚举类型声明的语法如下：

TYPE <标识符>：
(<Enum_0>,
<Enum_1>,
…,
<Enum_n>) |<基本数据类型>;
END_TYPE
<基本数据类型>为可选项，如果不填写数据类型，系统默认为INT类型。

（1）枚举类型的基本数据类型默认是INT类型，也可由用户明确指定，例如：

```
TYPE BigEnum : (yellow,blue,green:=16#8000) DINT;
END_TYPE
```

（2）枚举值可以直接用来做判断条件，例如

```
IF nWeeKday>5 THEN
nWeekday := 0;
```

（3）整数可以直接付给一个枚举变量，例如：

```
nWeekday := 10;
```

（4）同样的POU内，同样的枚举值不得用两次，例如：

```
TRAFFICAL_SIGNAL : (red,yellow,green);
COLOR: (blue,white,red);
Error:red may not be used for both TRAFFIC_SIGNAL and COLOR..
```

1.4. 变量的类型和初始化

CODESYS根据IEC61131-3标准对变量定义了属性，通过设置变量属性将他们的有关性能赋予变量。变量可根据其应用范围进行分类，除变量类型之外，CODESYS还提供了变量的附加属性。

1.4.1. 变量的类型

表1.11 CODESYS支持的变量属性

变量类型关键字	变量属性	外部读写	内部读写
VAR	局部变量		R/W
VAR_INPUT	输入变量，由外部提供	R/W	R
VAR_OUTPUT	输出变量，由内部变量提供给外部	W	R/W
VAR_IN_OUT	输入-输出变量	R/W	R/W
VAR_GLOBAL	全局变量，能在所有配置、资源内使用	R/W	R/W
VAR_TEMP	临时变量，程序和功能块内部存储使用的变量		R
VAR_STAT	静态变量		
VAR_EXTERNAL	外部变量，能在程序内部修改，但需由全局变量提供	R/W	R/W

VAR、VAR_INPUT、VAR_OUTPUT和VAR_IN_OUT是在程序组织单元（POU）中应用较多的几种变量类型。VAR_GLOBAL全局变量在实际的工程项目中也需要大量使用。

表1.12 变量附加属性及其关键字

变量附加属性关键字	变量附加属性
RETAIN	保持性变量，用于掉电保持
PERSISTENT	保持型变量，用于掉电保持
VAR RETAIN PERSISTENT VAR PERSISTENT RETAIN	两者功能一样，皆为保持型变量，用于掉电保持
CONSTANT	常量

（1）RETAIN

以关键字RETAIN声明类型变量。RETAIN型变量在控制器正常关闭、打开（或收到在线命令“热复位”），甚至意外关闭之后仍能够保持原来的值。随着程序重新开始运行，存储的值能够继续发挥作用。RETAIN 类型变量声明格式如下：

```
VAR RETAIN
<标识符>:<数据类型>;
END_VAR
```

但RETAIN变量在“初始值复位”“冷复位”和程序下载之后将会重新初始化。内存存储位置：RETAIN型变量仅仅存储在一个单独的内存区中。

（2）PERSISTENT RETAIN / RETAIN PERSISITENT

PERSISTENT类型变量声明格式如下：

```
VAR GLOBAL PERSISTENT RETAIN
<标识符>:<数据类型>
END_VAR
```

内存存储位置：与RETAIN变量一样，RETAIN PERSISTENT和PERSISTENT RETAIN变量也存储在一个独立的内存区中。

（3）CONSTANT

在程序运行过程中智能对齐读取数据而不能进行修改的量称为常量，关键字为CONSTANTA。可以将常量声明为局部变量，也可以声明为全局常量。

CONSTANT(常量)声明格式如下。

```
VAR CONSTANT
<标识符>:<数据类型>:=<初始化值>;
END_VAR
```

在实际应用中，通常可以将一些重要参数或系数设置成常量，这样可以有效地避免由于其他变量对其修改最终影响系统整体稳定性及安全性，举例如下：

```
VAR CONSTANT
   pi:REAL:=3.1415926;
END_VAR
```

程序一旦开始运行，通过CONSTANT声明的变量在程序运行过程中是不允许被修改的。

1.4.2. 变量的初始化

在变量中，有时需要对一些变量预先赋值初值，CODESYS允许在定义变量时对变量进行赋初值处理，在赋值运算符“:=”的左边是变量及变量类型，该变量接收右边地址或表达式的值，例如：

```
VAR
   bBoolValue:BOOL:=TRUE;
   rRealValue:REAL:=3.1415926;
   nIntValue:INT:=6;
   strValue:STING:='Hello Leadshine'
END_VAR
```

此外，也可以对统一类型的多个变量进行同时赋初值，例如：

```
bBoolValue1, bBoolValue2, bBoolValue3, bBoolValue4:BOOL:=TRUE;
```

上述程序运行后，以上4个变量的值都为TURE。

数组的赋初值示例如下：

```
VAR
   arr1: ARRAY[1..5]OF INT:=[1,2,3,4,5];
   arr2:ARRAY[1..2,3..4] OF INT := [1,3(7)];
   arr3:ARRAY[1..2,2..3,3..4]OF INT:=[0,0,4,4,4,4,2,3];
END_VAR
```

结构体赋初值示例如下：

```
VAR
   Motor_1:Motor := (Vendor:='Leadshine',Brake:=TRUE)
END_VAR
```

用户自定义的初值只在控制器刚启动后的第一个任务周期对该变量写入一次。在程序中，如果直接在变量声明去修改了初始值，“登入到”PLC后选择“在线修改”并不会对该变量的数据有任何改变，只有在PLC复位后重新运行程序才有用。

1.5. 变量的声明及字段指令

1.5.1. 变量匈牙利命名法

匈牙利命名法是一种编程时的命名规范，它的基本原则是：变量名=属性+类型+对象描述，其中每一个对象的名称都要求有明确含义，可以取对象名字全称或者名字的一部分。命名要基于“容易记忆、容易理解”的原则。保证名字的连贯是非常重要的。

CODESYS的标准库也采用匈牙利命名法则。在声明变量、用户自定义数据类型和创建POU（函数、功能块、程序）时定义标识符。为了使标识符的名称尽量不予其他名称重复，除了必须遵守的事项之外，用户可能还需要参考以下一些建议。

给应用程序和库中的变量命名时应尽可能地遵循匈牙利命名法。每一个变量的基本名字中应该包含一个有意义的简短描述。基本名字中每一个单词的首字母应当大写，其他字母则为小写，如FileSize。再根据变量的数据类型，在基本名字前加上小写字母前缀。可参考下表中列举出来的一些特定数据类型的推荐前缀和其他相关信息。

第2章　ST指令总览

ST是结构化文本Structured Text的英文简称，是用结构化的描述文本来编写程序的一种编程语言。类似于C语言，程序代码由指令组成，指令由关键字和表达式组成。ST语言适合于算法和结构较为复杂以及其它编程语言（如梯形图等）实现比较困难的情况，具有高效、快捷、简洁的优点。

2.1 ST基础指令

ST基础指令主要包括：ST语法结构指令、数据操作指令、数据转换指令、定时与计数指令以及辅助指令五大类。

2.1.1　　ST语法结构指令

表2.1　ST语法指令表

指令	指令名称	功能说明
:=	赋值	将变量、表达式或FUN/FB的值赋给另一个变量
IF	判断	根据指定条件的判断结果，二选一选择要执行的语句
CASE	选择	根据指定变量的值，从多个语句中选择要执行的语句
FOR	循环	指定循环条件，重复执行从FOR语句到END_FOR语句之间的内容
WHILE	循环	指定循环条件的判断结果为TRUE时，重复执行某语句
REPEAT	循环	执行一次某语句，在指定条件式变为TRUE前，重复执行该语句
CONTINUE	继续	中断本次循环，直接执行下次循环
RETURN	跳出	结束POU、函数或功能块，将处理恢复到调用源
EXIT	跳出	EXIT用于退出FOR循环、WHILE循环、REPEAT循环
JMP	跳转	跳转到label所在的位置执行程序

2.1.2　　数据操作指令

表2.2　ST数据操作指令表

种类	指令	指令名称	功能说明
数学函数	ADD，+	加法	变量或常量的加法运算
	MUL，*	乘法	变量或常量的乘法运算
	SUB，-	减法	变量或常量的减法运算
	DIV，/	除法	变量或常量的除法运算
	MOD	取余	计算除法运算时的余数
	ABS	绝对值	计算一个数的绝对值

种类	指令	指令名称	功能说明
	SIN	正弦	计算一个数的正弦值
	COS	余弦	计算一个数的余弦值
	TAN	正切	计算一个数的正切值
	ASIN	反正弦	计算一个数的反正弦值
	ACOS	反余弦	计算一个数的反余弦值
	ATAN	反正切	计算一个数的反正切值
	SQRT	平方根	计算一个数的平方根
	LOG	10为底对数	计算以10为底的对数
	LN	自然对数	计算一个数的自然对数
	EXP	自然常数N次幂	计算以自然常数（e）为底的N次幂
	EXPT	幂	计算两个数的乘幂
	SIZEOF	获取字节数	计算输入变量的类型所占的字节数
选择操作符	MIN	最小值	计算两个或多个输入变量的最小值
	MAX	最大值	计算两个或多个输入变量的最大值
	LIMIT	限制值	根据设定的上下限值，限制输入变量的值
	SEL	二选一	从两个操作数中选择一个
	MUX	多选一	从多个输入变量中选择指定的输入变量作为输出
数据转移	MOVE	赋值	将一个变量或常量的值赋给另一个相应类型的变量
字逻辑运算	AND	与	按位进行与运算
	OR	或	按位进行或运算
	XOR	异或	按位进行异或运算
	NOT	非	按位进行非运算
移位操作符	SHL	左移	将操作数按位左移
	SHR	右移	将操作数按位右移
	ROL	循环左移	将操作数按位循环左移
	ROR	循环右移	将操作数按位循环右移
比较操作符	GT，>	大于	进行两个操作数的大小比较（>）
	LT，<	小于	进行两个操作数的大小比较（<）
	LE，<=	小于或等于	进行两个操作数的大小比较（<=）
	GE，>=	大于或等于	进行两个操作数的大小比较（>=）
	EQ，=	等于	进行两个操作数的大小比较（=）
	NE，<>	不等于	进行两个操作数的大小比较（<>）
字符串操作符	LEN	字符串长度	计数输入字符串的长度
	LEFT	左边取字符串	从左往右提取指定数量的字符
	RIGHT	右边取字符串	从右往左提取指定数量的字符
	MID	中间取字符串	从任意位置提取指定数量的字符
	CONCAT	合并字符串	连接2个字符串
	INSERT	插入字符串	将一个字符串插入另一个字符串的指定位置
	DELETE	删除字符串	在字符串中删除指定的字符串
	REPLACE	替换字符串	用其它字符串替换部分字符串
	FIND	查找字符串	在一个字符串中找到指定字符串的初始位置

2.1.3 数据转换指令

表2.3 ST数据转换指令表

种类	指令	指令名称	功能说明
类型转换操作符	BOOL_TO_	BOOL→其它类型	将布尔类型转换为其它类型，包括BYTE、WORD、DWORD、INT、REAL、STRING、TIME、TOD、DATE、DT
	_TO_BOOL	其它类型→BOOL	将其它类型转换为BOOL类型，包括BYTE、WORD、DWORD、INT、REAL、STRING、TIME、TOD、DATE、DT
	<INT>_TO_<INT>	整数类型之间的转换	从整数转换为不同类型的整数，包括SINT、USINT、DINT、UDINT、LINT、ULINT
	REAL/LREAL_TO_	REAL/LREAL→其它类型	从实数/长实数变量类型转换为其它数据类型
	<时间>_TO_	TIME/TOD→其它类型	把时间（TIME）或时刻（TIME_OF_DAY）类型变量转换为其他类型变量
	DATE/DT_TO_	DATE/DT→其它类型	从日期和日期时间类型变量转换为其他类型变量
	STRING_TO_	STRING →其它类型	把字符串类型变量转换为其它类型，包括BYTE、WORD、INT、REAL、DATE、TIME、TOD、DT等
	TRUNC_INT	REAL→INT	将REAL类型转化为INT类型
	TRUNC	REAL→DINT	将REAL类型转化为DINT类型
	TO_	任意类型→另一种类型	把任意类型、或任意数字类型的数据转化为另一种类型
位/字节转换功能	EXTRACT	位抽取	提取双字中某一位的状态输出
	PACK	位整合	将8个BOOL输入整合为1个字节
	PUTBIT	设置位值	将DWORD数据中的某一位置为TRUE或FALSE后输出新的DWORD数据
	UNPACK	位拆分	将BYTE类型拆分为8个BOOL类型输出
	SWITCHBIT	位取反	将DWORD数据中的某一位取反后，输出新的DWORD数据
	WORD_AS_BIT	字拆分	将WORD类型拆分为16个BOOL类型输出
	BIT_AS_WORD	位整合字	将16个BOOL类型输入整合为1个字输出
BCD转换	BCD_TO_INT	BCD→INT	将BCD码转换成INT值
	BCD_TO_BYTE	BCD→BYTE	将BCD码转换成BYTE值
	BCD_TO_WORD	BCD→WORD	将BCD码转换成WORD值
	BCD_TO_DWORD	BCD→DWORD	将BCD码转换成DWORD值
	INT_TO_BCD	INT→BCD	将整数值转换成BCD码
	BYTE_TO_BCD	BYTE→BCD	将字节转换成BCD码
	WORD_TO_BCD	WORD→BCD	将字转换成BCD码
	DWORD_TO_BCD	DWORD→BCD	将双字转换成BCD码

2.1.4 定时与计数指令

表2.4 ST定时与计数指令表

种类	指令	指令名称	功能说明
定时器	TP	定时器	仅在设定时间内输出TRUE的定时器
	TON	通电延时定时器	通电后经过设定时间输出TRUE的定时器
	TOF	断电延时计时器	断电后经过设定时间输出FALSE的定时器
	RTC	实时时钟	返回从给定时间开始计时的当前日期和时间
计数器	CTU	递增计数器	每输入一个输入信号计数增加的计数器
	CTD	递减计数器	每输入一个输入信号计数减少的计数器
	CTUD	递增递减计数器	双向计数器

2.1.5 辅助指令

表2.5 ST辅助指令表

种类	指令	指令名称	功能说明
地址操作符	ADR	取地址	返回变量自身的地址
	BITADR	取位地址	返回位偏移量的地址
触发器	R_TRIG	上升沿检测触发器	检测上升沿信号
	F_TRIG	下降沿检测触发器	检测下降沿信号
双稳态	SR	置位优先触发器	置位复位输入同时为TRUE时，优先置位输入
	RS	复位优先触发器	置位复位输入同时为TRUE时，优先复位输入
数学辅助功能	DERIVATIVE	求导	对连续输入的变量IN按照时间TM（ms）求导
	INTEGRAL	积分	对连续输入的变量IN按照时间TM（ms）积分
	LIN_TRAFO	线性转换	将在原始下限和上限值确定的范围内的实数，转换为由新的工程下限和上限值确定的范围内的实数
	STATISTICS_INT	统计整数的最大、最小和平均值	用统计方法计算输入整数的最大、最小和平均值
	STATISTICS_REAL	统计实数的最大、最小和平均值	用统计方法计算输入实数的最大、最小和平均值
	VARIANCE	方差	计算输入值的方差
调节器	PD	比例微分控制器	比例微分控制器
	PID	比例积分微分控制器	比例积分微分控制器，比PD控制器多一个积分项
	PID_FIXCYCLE	周期恒定的比例积分微分控制器	周期固定的比例积分微分控制器
信号发生器	BLINK	方波	方波脉冲信号发生器
	FREQ_MEASURE	频率测量	测量一个布尔类型输入信号的（平均）频率（单位Hz）
	GEN	函数发生器	可产生7种典型的波形
函数操作	CHARCURVE	数据点线性化	将给定的若干个数据点，分段进行线性化
	RAMP_INT	INT输入的斜波函数	根据输入INT值的变化，输出OUT斜坡上升或下降函数

种类	指令	指令名称	功能说明
	RAMP_REAL	REAL输入的斜波函数	根据输入REAL值的变化，输出OUT斜坡上升或下降函数
模拟量监视	HYSTERESIS	阈值限制状态	根据设定的上下限阈值，控制BOOL型输出的状态
	LIMITALARM	阈值限制标志	根据设定的上下限阈值，输出BOOL型状态标志位
IEC扩展操作符	__NEW	分配内存	创建一个指定类型的新对象并返回指向该对象的指针
	__DELETE	释放内存	释放由__NEW分配的内存
	__ISVALIDREF	检查引用是否有效	检查一个引用是否指向一个不等于0的有效值
	__QUERYINTERFACE	允许接口引用的类型转换	允许接口引用的类型转换，该运算符返回一个BOOL类型的结果
	__QUERYPOINTER	分配无类型指针的接口引用	分配一个无类型指针的接口引用，该运算符返回一个BOOL类型的结果

2.2雷赛专用指令库

雷赛专用指令是雷赛产品研发人员为了方便用户使用PMC600系列产品，专门针对该系列控制器封装的库。当用户需要使用控制器的某一功能时，只需直接在程序内调用库中对应功能的指令即可。对于PMC600系列产品来说，主要包括PMC_Controller库、文件操作库FileManage和Communication通讯库。

2.2.1 PMC_Controller库

表2.6 PMC_Controller库

种类	指令	指令名称	功能说明
PWM	LS_PWM_SetVal	PWM输出	按设置的PWM端口号、频率及占空比输出PWM波形
	LS_PWM_GetVal	回读PWM参数	回读功能块LS_PWM_SetVal设置的PWM参数信息
系统时间	LS_DateAndTime_ReadDint	获取控制器系统时间	将年月日时分秒拆分读取控制器系统时间
	LS_DateAndTime_ReadString	获取控制器系统时间	按日期时间读取控制器系统时间
	LS_DateAndTime_SetDint	设置控制器系统时间	按年月日时分秒字段设置系统时间
	LS_DateAndTime_SetString	设置控制器系统时间	按日期时间设置系统时间。
固件版本	LS_PLC_ControllerVer	读取硬件版本	获取控制器的硬件版本信息
	LS_PLC_CPUID	读取CPU ID号	获取控制器的CPU ID号
	LS_PLC_HardwareVerInformation	读取固件FPGA版本	获取控制器的固件FPGA版本信息
	LS_PLC_SoftwareVerInformation	读取固件ARM版本	获取控制器的固件ARM版本信息
	LS_PLC_VerInformation	读取控制器版本	获取控制器的所有版本信息

2.2.2 FileManage文件操作库

表2.7 FileManage文件操作库

种类	指令	指令名称	功能说明
UsrData内文件操作	FileWrite	创建并写入文件	在控制器“UsrData”文件夹内创建新文件并写文件内容
	FileWriteAppend	追加写入文件	在控制器“UsrData”文件夹内采用追加的方式写文件内容
	FileRead	读取文件	在控制器“UsrData”文件夹内读取文件
	FileCopy	拷贝文件	在控制器“UsrData”文件夹内拷贝文件
	FileRename	重命名文件	在控制器“UsrData”文件夹内对文件进行重命名操作
	FileDelete	删除文件	在控制器“UsrData”文件夹内删除文件操作
	FileDeleteAll	批量删除文件	在控制器“UsrData”文件夹内批量删除文件操作
	GetDirectoryFile	获取指定路径下的文件信息	从控制器“UsrData”文件夹内获取该目录下的所有文件名及相关文件信息
	GetFileInformation	获取文件信息	在控制器“UsrData”文件夹内获取文件信息
	UDisk_CopyFromUDisk	从U盘拷贝文件	从U盘拷贝文件至控制器“UsrData”文件夹内
	UDisk_CopyToUDisk	拷贝文件至U盘	从控制器“UsrData”文件夹内将文件拷贝至U盘
	UDisk_GetDirectoyFile	获取U盘目录下文件信息	获取U盘目录下的文件名及相关文件信息
UsrConfig内文件操作	ConfigFile_Write	创建并写入文件	在控制器“UsrConfig”文件夹内创建新文件，写文件内容
	ConfigFile_WriteAppend	追加写入文件	在控制器“UsrConfig”文件夹内采用追加的方式写入文件内容
	ConfigFile_Read	读取文件	在控制器“UsrConfig”文件夹内读取文件内容
	ConfigFile_Del	删除文件	在控制器“UsrConfig”文件夹内删除文件
	ConfigFile_GetDirectoryFile	获取指定路径下的文件信息	从控制器“UsrConfig”文件夹内获取该目录下的所有文件名及相关文件信息
	ConfigFile_CopyFromUDisk	从U盘拷贝文件	从U盘拷贝文件至控制器“UsrConfig”文件夹内
	ConfigFile_CopyToUDisk	拷贝文件至U盘	从控制器“UsrConfig”文件夹内将文件拷贝至U盘

2.2.3 Communication通讯库

表2.8 Communication通讯库

种类	指令	指令名称	功能说明
HMI库变量批量操作	LS_Modbus_SetM	BOOL变量批量置位	批量置位BOOL型变量
	LS_Modbus_ResetM	BOOL变量批量复位	批量复位BOOL型变量
	LS_Modbus_SetByteToM	BYTE变量批量置位	按位设置字节型变量
	LS_Modbus_SetDWS	INT变量批量置值	设置INT型变量的值
	LS_Modbus_SetDWU	UINT变量批量置值	设置UINT型变量的值
	LS_Modbus_SetDDS	DINT变量批量置值	设置DINT型变量的值
	LS_Modbus_SetDDU	UDINT变量批量置值	设置UDINT型变量的值
	LS_Modbus_SetDR	REAL变量批量置值	设置REAL型变量的值
	LS_Modbus_SetDS	STRING变量批量置值	设置STRING型变量的值
RS232无协议通信	LS_RS232_Open	打开串口	打开指定的串行端口
	LS_RS232_Close	关闭串口	关闭指定的串行端口
	LS_RS232_Read	接收串口数据	从串行端口读取无协议的接收数据
	LS_RS232_Write	发送串口数据	从串行端口进行无协议数据发送
RS485无协议通信	LS_RS485_Open	打开RS485口	打开指定的RS485端口
	LS_RS485_Close	关闭RS485口	关闭指定的RS485端口
	LS_RS485_Read	接收串口数据	从RS485端口读取无协议的接收数据
	LS_RS485_Write	发送串口数据	从RS485端口发送无协议数据
以太网通信	LS_TCP_SetIpAddress	设置网口IP	设置本机内置指定网口的IP地址
	LS_TCP_GetIpAddress	读取网口IP	获取内置指定网口的IP地址
	LS_TCP_SetServer	设为TCP服务器	将指定IP地址的网口设置为TCP服务器
	LS_TCP_SetClient	设为TCP客户端	将指定IP地址的网口设置为TCP客户端
	LS_TCP_WriteData	TCP发送数据	TCP通信发送数据
	LS_TCP_ReadData	TCP接收数据	TCP通信接收数据
	LS_UDP_SetConnect	启动UDP/IP服务	将指定IP地址的网口启用UDP/IP服务
	LS_UDP_WriteData	UDP发送数据	UDP通信发送数据
	LS_UDP_ReadData	UDP接收数据	UDP通信接收数据

第3章 ST基础指令

本章主要是对ST语法结构指令、数据操作指令、数据转换指令、定时与计数指令以及辅助指令进行更加详细介绍，同时结合相应的例句让用户更清楚、更快速地了解并使用ST指令。

3.1 ST语法结构指令

3.1.1 表达式

表达式中包括操作符和操作数，操作数按照操作符指定的规则进行运算，得到结果并返回。操作数可以为变量、常量、寄存器地址、函数等。例如：

a+b+c+d

2*3.14*R

如果在表达式中有若干个操作符，则操作符会按照规定的优先级顺序执行；先执行优先级高的操作符运算，再顺序执行优先级低的操作符运算。如果在表达式中具有优先级相同的操作符，则这些操作符按照书写顺序从左至右执行。

操作符的优先级如下表所示：

表3.1 操作符优先级表

操作符	符号	优先权
小括号	( )	最高
函数调用	函数名(参数列表，由逗号分隔)	
求幂	EXPT	
求负值 取反	- NOT	
乘法 除法 取模	* / MOD	
加法 减法	+ -	
比较	<,>,<=,>=	
等于 不等于	= <>	
逻辑与	AND	
逻辑异或	XOR	
逻辑或	OR	最低

3.1.2 赋值指令:=

赋值指令用作将变量、表达式或FUN/FB的值赋给另一个变量。

语法：

变量A := 变量、表达式或FUN/FB的值

将操作符 ”:=” 右边变量、表达式或FUN/FB的值赋给左边的变量

例程：

V1的值加1，再将得到的结果赋给V2

```
V2:=V1+1;
```

3.1.3 判断指令IF

使用IF指令可以检查条件，并根据此条件执行相应的指令。

语法：

常用的IF指令结构有以下3种：

```
（1）IF  条件A   THEN   //当条件A满足时，执行语句A
        语句A ;
     END_IF
（2）IF  条件A THEN //当条件A满足时，执行语句A;否则，执行语句B
        语句A ;
     ELSE
        语句B;
     END_IF
（3）IF  条件A THEN       //当条件A满足时，执行语句A;
        语句A ;
   ELSIF  条件B THEN     //当条件B满足时，执行语句B;
        语句B ;
        …
   ELSIF  条件N-1 THEN   //当条件N-1满足时，执行语句N-1;
        语句N-1;
   ELSE                  //如果以上条件都不满足，则执行语句N;
        语句N ;
   END_IF                //指令结束。
```

表3.2　条件表达式

可作为条件的内容	示例	评估结果
逻辑表达式	a=b	变量a、b的值相等，则为TRUE，其他为FASE
BOOL型变量	c	若c的值为TRUE，则为TRUE，若为FALSE，则为FALSE
BOOL型常数	TRUE	TRUE
包括BOOL型返回值的函数	FUN名称	若函数的返回值为TRUE，则为TRUE，若为FALSE，则为FALSE

逻辑表达式中可使用表3.3中的运算符。

表3.3　逻辑运算符

运算符	含义	记述示例	评估结果
=	等于	a=b	变量a、b的值相等，则为TRUE，其他为FALSE
<>	不等于	a<>b	变量a、b的值不相等，则为TRUE，其他为FALSE
<	比较	a<b	变量a的值小于b，则为TRUE，其他为FALSE
<=		a<=b	变量a的值小于等于b，则为TRUE，其他为FALSE
>		a>b	变量a的值大于b，则为TRUE，其他为FALSE
>=		a>=b	变量a的值大于等于b，则为TRUE，其他为FALSE
AND	逻辑与	a AND b	BOOL型变量a、b的逻辑与
OR	逻辑或	a OR b	BOOL型变量a、b的逻辑或
XOR	异或	a XOR b	BOOL型变量a、b的逻辑异或
NOT	逻辑非	NOT a	BOOL型变量a的逻辑非

注意：

- ✧ IF 和 END_IF 不可省略。此外，以上指令请务必成对使用。
- ✧ 包括 IF 指令、CASE 指令、FOR 指令、WHILE 指令、REPEAT 指令在内，层级深度最多为 15。

例程：

若变量a的值小于0，则变量b的值为0。若变量a的值为0，则变量b的值为1。若变量a的值不在上述范围内，则变量b的值为2。

```
IF a<0 THEN
    b:=0;
  ELSIF a=0 THEN
    b:=1;
  ELSE
    b:=2;
END_IF
```

3.1.4 选择指令CASE

在根据控制变量的值，从多个语句中选择要执行的语句。当使用IF指令有过多分层，或者需要使用多个ELSEIF，才能完成程序功能时，使用CASE指令替代IF指令，可以简化程序，并且能提高程序的可读性。

语法：

```
CASE 控制变量 OF
    选择值:
        语句;
    选择值:
        语句;
    …
    ELSE
        语句;
END_CASE
```

CASE指令用于将控制变量和若干个操作数进行比较，如果控制变量与其中一个值相同，则执行该值对应的语句。如果与任何一个值都不相同，则执行ELSE指令的语句。控制变量和选择值类型如表3.4所示。

表3.4 控制变量和选择值

	可记述内容
控制变量	包含整数型变量、整数型表达式、整数型返回值的函数。 包含枚举型变量、枚举型表达式、枚举型返回值的函数
选择值	整数型常数

本指令可以是层级结构。如下例所示：语句11 在控制变量1 的值为 1 且控制变量10 的值为 2 时执行

```
CASE    控制变量1   OF
1 :
    CASE    控制变量10   OF
        1:
            语句10;
        2:
            语句11;
        ELSE
            语句m;
    END_CASE;
2 :
    语句2;
3 :
    语句3;
ELSE
    语句n;
```

```
END_CASE
```

选择值中可记述多个值。多个值用 ’,’(逗号) 分隔。如下例所示：控制变量1的值为 1 或 2 时执行语句1。

```
CASE    控制变量1   OF
1,2 :
    语句1;
3 :
    语句2;
4 :
    语句3;
ELSE
    语句m;
END_CASE
```

选择值中可记述连续值。连续值用’..’(两个句点)分隔。如下例所示：整数式的值为10以上15以下时执行语句1。

```
CASE  控制变量1   OF
  10..15 :
        语句1;
    16 :
        语句2;
    17 :
        语句3;
ELSE
        语句m;
END_CASE
```

ELSE 可省略。此时，若整数式的值不等于任何选择值，则不执行任何语句。与C语言的switch语句格式有如下不同。C语言的switch语句中只要不使用break语句，与整数式相等的选择值之后的语句将全部执行。而 CASE 语句中只会执行与整数式相等的选择值对应的语句。例如表3.5的示例，C语言的switch语句将执行从语句1到语句3的所有语句。而CASE指令中只会执行语句1。

表3.5　ST和C语言选择语句的区别

C语言	ST

val=1;
switch val
{
　case 1:
　　语句1;
　case 2:
　　语句2;
　case 3:
　　语句3;
}
	val:=1;
	CASE val OF
	　　1:
	　　语句1;
	　　2:
	　　语句2;
	　　3:
	　　语句3;
	END_CASE

注意：

- ✧ CASE 和 END_CASE 不可省略。此外，以上指令请务必成对使用。
- ✧ 控制变量和选择值的数据类型不同亦可。
- ✧ 选择值中不可重复记述相同的值。
- ✧ 包括 IF 指令、 CASE 指令、 FOR 指令、 WHILE 指令、 REPEAT 指令在内，层级深度最多为 15。

例程：

若变量a = 1或2时，则变量b的值为10。若变量a的值为3，则变量b的值为20。若变量a的值不在上述范围内，则变量b的值为30。

```
CASE   a   OF
1,2:
     b:=10;
3:
        b:=20;
ELSE
        b:=30;
END_CASE
```

3.1.5 循环指令FOR

FOR循环指令用于一些需要重复执行的语句，它可以使程序简短并且一目了然。但需要注意避免陷入死循环。FOR循环指令是有限制的循环指令，当限制条件满足（变量值等于“循环结束时变量值”）时，程序就将退出FOR循环，执行下一条指令。FOR循环用于在发生次数可预先确认的情况下。否则可使用WHILE或REPEAT。

语法：

```
FOR 控制变量 := 循环起始值 TO 循环结束值  {BY 递增步长}  DO
    语句;
END_FOR
```

其中，{}内语句可根据需要省略，省略时步长默认为1。若要中断循环指令，请执行EXIT指令。控制变量、起始值、结束值、递增步长的类型如表3.6所示。

表3.6 FOR变量类型

变量	类型
控制变量	包含整数型变量、整数型表达式、整数型返回值的函数。 包含枚举型变量、枚举型表达式、枚举型返回值的函数
起始值、结束值、递增步长	整数型常数

注意：

- ✧ 控制变量、起始值和结束值必须具有相同的数据类型。
- ✧ 循环结束值不能等于其数据类型的最大值，否则会进入死循环。
- ✧ 当循环起始值小于循环结束值时，递增步长不能等于 0，否则会进入死循环。
- ✧ 包括 IF 指令、 CASE 指令、 FOR 指令、 WHILE 指令、 REPEAT 指令在内，层级深度最多为 15。

例程：

循环控制变量为Counter，循环开始时控制变量初值为1，每一次循环Counter+1；当Counter等于5时，执行完FOR循环内容后，退出循环，执行下一条语句。
语句Var1:=Var1*2一共执行5次；若Var1的初始值是1，那么循环结束后，Var1的值为32。

```
FOR Counter:=1 TO 5 BY 1 DO
    Var1:=Var1*2;
END_FOR
```

若上例中所使用的计数变量Counter的类型是SINT（范围从-128到127），如果语句为“FOR Counter:=1 TO 127 BY 1 DO”则会进入死循环。编程时应避免此类情况的发生。

3.1.6 循环指令WHILE

WHILE循环与FOR循环使用方法类似。二者的不同之处是，WHILE循环的结束条件不是指定的循环次数，而是任意的逻辑表达式。当该表达式叙述的条件满足时，执行循环。

语法：

```
WHILE 循环条件 DO
        语句;
END_WHILE
```

WHILE循环执行前先检查<循环条件>是否为TRUE，如果为TRUE，则执行<语句>；当执行完一次后，再次检查<循环条件>，如果仍为TRUE，则再次执行，直到<循环条件>为FALSE。如果一开始<循环条件>就为FALSE，则不会执行WHILE循环里的指令。

若要中断循环处理，请执行EXIT 指令。此时，将不执行从EXIT 指令到END_WHILE指令之间的处理。

循环条件如表3.7所示。

表3.7　循环条件

循环条件	示例	评估结果
逻辑表达式	a=b	变量a、b的值相等，则为TRUE，其它为FALSE
BOOL型变量	a	若a的值为TRUE，则为TRUE，若为FALSE，则为FALSE
BOOL型常量	TRUE	TRUE
包含BOOL型返回值的函数	FUN名称	若函数的返回值为TRUE，则为TRUE，若为FALSE，则为FALSE

逻辑表达式如表3.8所示。

表3.8　逻辑运算符

运算符	含义	记述示例	评估结果
=	等于	a=b	变量a、b的值相等，则为TRUE，其他为FALSE
<>	不等于	a<>b	变量a、b的值不相等，则为TRUE，其他为FALSE
<	比较	a<b	变量a的值小于b，则为TRUE，其他为FALSE
<=		a<=b	变量a的值小于等于b，则为TRUE，其他为FALSE
>		a>b	变量a的值大于b，则为TRUE，其他为FALSE
>=		a>=b	变量a的值大于等于b，则为TRUE，其他为FALSE
AND	逻辑与	a AND b	BOOL型变量a、b的逻辑与
OR	逻辑或	a OR b	BOOL型变量a、b的逻辑或
XOR	异或	a XOR b	BOOL型变量a、b的逻辑异或
NOT	逻辑非	NOT a	BOOL型变量a的逻辑非

注意：

- ✧ WHILE 和 END_WHILE 不可省略。此外，以上指令请务必成对使用。
- ✧ 包括 IF 指令、 CASE 指令、 FOR 指令、 WHILE 指令、 REPEAT 指令在内，层级深度最多为 15。
- ✧ 应当避免死循环的产生。可以通过改变循环指令部分的条件来避免死循环的产生。例如：利用可增减的计数器避免死循环的产生。

例程：

变量Counter大于0，则一直会执行WHILE循环中的指令，直到Counter小于等于0为止。每执行一次循环，通过指令“Counter := Counter-1”使Counter的值减1，当Counter小于等于0时，循环结束。

```
WHILE Counter>0 DO
        Counter := Counter-1;
END_WHILE
```

3.1.7　循环指令REPEAT

REPEAT循环与WHILE循环一样，也是没有明确循环次数的循环。与WHILE循环的区别在于，REPEAT循环在指令执行以后，才检查结束条件。因此无论结束条件怎样，循环至少执行一次。

若要中断重复处理，请执行 EXIT 指令。此时，将不执行从 EXIT 指令到 END_REPEAT 指令之间的处理。

语法：

```
REPEAT
    语句;
UNTIL 循环结束条件
END_REPEAT
```

语句一直执行，直到<循环结束条件>为TRUE时，REPEAT循环结束。如果<循环结束条件>一开始就为TRUE，则循环只执行一次。

循环条件与WHILE循环指令相同，参见表3.7。

逻辑表达式与WHILE循环指令相同，参见表3.8。

注意：

- ✧ REPEAT、 UNTIL、 END_REPEAT 均不可省略。此外，以上指令请务必成对使用。
- ✧ 包括 IF 指令、 CASE 指令、 FOR 指令、 WHILE 指令、 REPEAT 指令在内，层级深度最多为 15。
- ✧ 应当避免死循环的产生。可以通过改变循环指令部分的条件来避免死循环的产生。例如：利用可增减的计数器避免死循环的产生。

例程：

上述WHILE示例程序也可写为：

```
REPEAT
    Counter := Counter-1;
UNTIL Counter < 0
END_REPEAT。
```

在一定意义上来说，WHILE循环和REPEAT循环比FOR循环功能更强大，因为不需要在执行循环之前计算循环次数。因此，在有些情况下，用WHILE循环和REPEAT循环两种循环就可以了。然而，如果清楚知道循环次数，那么FOR循环更简单。

3.1.8 继续指令CONTINUE

CONTINUE指令可以在FOR、WHILE和REPEAT三种循环中使用，其作用为中断本次循环，直接执行下次循环。

语法：

CONTINUE;

例程：

当I1等于0时，直接结束本次循环，开始下一次循环，以避免指令“V1:=V1/I1”中对I1的除零操作。

```
FOR Counter:=1 TO 5 BY DO
    I1:=I1/2;
    IF I1=0 THEN
        CONTINUE;
    END_IF
    V1:=V1/I1
END_FOR
```

3.1.9 跳出指令RETURN

结束POU、函数或功能块，将处理恢复到调用源。

语法：

RETURN;

注意：

- ✧ 执行本指令前，请设置该POU的返回值、输出变量的值
- ✧ 若经常使用本指令，处理流程将变的复杂。敬请注意。

例程：

如果b为TRUE，将不会执行a := a +1，而是直接退出POU。

```
IF b=TRUE THEN
    RETURN;
END_IF
a := a + 1;
```

3.1.10 跳出指令EXIT

EXIT用于退出FOR循环、WHILE循环、REPEAT循环。多级循环嵌套时，退出最内侧的循环。

语法：

EXIT;

注意：

- ✧ 本指令请务必在FOR和END_FOR之间、WHILE和END_WHILE之间或REPEAT和END_REPEAT之间使用。
- ✧ 分层使用循环（嵌套）时，若要中断所有循环，本指令的数量需要与层数对应。

例程：

当INT1等于0时，FOR循环结束。

```
FOR Counter:=1 TO 5 BY 1 DO
    I1:= I1/2;
    IF I1=0 THEN
        EXIT;
    END_IF
    V1:=Var1/I1;
END_FOR
```

3.1.11 跳转指令JMP

跳转指令，跳转到label所在的位置执行程序。JMP指令容易造成程序结构混乱，降低代码可读性，不建议使用。

语法：

```
label：
    ……
JMP label;
```

<label>可以是任意确定的标识符，它被放置在程序行的开始。JMP指令后面一定紧跟着跳转目的地，即一个预先定义的标识符。 当执行到JMP指令时，将跳转到标识符所指的程序行。

注意：必须避免制造死循环，例如可以使用IF条件控制跳转指令。

例程：

当i<10时，跳转回label所在行，执行i:=i+1。

```
i := 0;
label: i := i+1;
.....
    IF i<10 THEN
        JMP label;
    END_IF
```

一旦变量i被初始化为0，这是一个小于10的值，上述例子中的条件跳转指令将导致重复执行跳转到标识符label对应的程序行，因此将重复处理包含在标识符label和JMP指令之间的指令。由于这些指令中包含了变量i的增量操作，这样可以确保跳转条件最终不再被满足（在第9次检查时），从而可以执行后续指令。

这个例子中的功能同样可以通过使用WHILE或者REPEAT循环来实现。通常情况下，能够并且也应该避免使用跳转指令，因为这降低了代码的可读性。

3.1.12 调用功能块

如果需要在ST中调用功能块，可直接输入功能块的实例名称，并在随后的括号中给功能块的各参数分配数值或变量，参数之间以逗号隔开；功能块调用以分号结束。

按照如下的语法规则，在结构化文本中调用功能块（以下简称“FB”）。

语法：

<name of FB instance>(
FB input variable := <value or address>,
further FB input variable := <value or address>,
…,
FB output variables => <value or address >,
further FB output variables => <value or address >,
…);

举例：

在ST中调用TON定时器，假设其实例名为TON_1：

```
TON_1(IN := btrue , PT := T#500ms ,Q => T1_OUT , ET => T1_ET );
```

3.1.13 ST中的注释

注释是程序中非常重要的一部分，它使程序更加具有可读性，同时不会影响程序的执行。在ST编辑器的声明部分或执行部分的任何地方，都可以添加注释。ST语言中，有两种注释方法：

（1）注释以 (* 开始，以 *) 结束。这种注释方法允许多行注释。

例如：

```
(*
a:=inst.out; (* to be checked *)
b:=b+1;
*)
```

（2）注释以“//”开始，一直到本行结束。这是单行注释的方法。

例如：// This is a comment.

嵌套注释：可以在注释里再添加注释。

3.2 数据操作指令

3.2.1 数学函数指令

四则运算： +、-、*、/

例程： Var1 := a+b*5-c/8

功能说明：

对变量或常量进行四则运算。输入输出支持的数据类型有：BYTE、WORD、DWORD、LWORD、SINT、USINT、INT、 UINT、DINT、UDINT、LINT、ULINT、REAL、LREAL、TIME、TIME_OF_DAY (TOD)、 DATE、DATE_AND_TIME(DT)。

两个时间类型（如：TIME型、TOD型、DATE型、DT型）变量也可以运算，得到另一个时间类型变量。（例如：t#45s + t#50s = t#1m35s）。TIME 数据类型的可能组合：TIME+TIME = TIME，TOD+TIME = TOD，DT+TIME = DT。TIME-TIME = TIME, DATE-DATE = TIME, TOD-TIME = TOD, TOD-TOD = TIME, DT-TIME = DT, DT-DT = TIME。

注意：

- ✧ 各变量之间的类型可不同，但必须存在包含关系。此时应以包含所有数据的数据类型进行运算处理。被定义的输出数据类型应可以存储输出结果，否则可能引起数据错误。例如，一个变量类型为INT，一个变量类型为REAL，计算中会以REAL类型进行运算处理，所以计算结果为REAL型数据。
- ✧ 若运算结果超出变量类型的有效范围，将发生溢出。即使发生溢出，也不会发生异常。因发生溢出时，可能运算结果与预计不同。请确保输入变量、输出变量的数据类型大小，有充分的余量，以免发生溢出。
- ✧ 若相乘结果超出变量类型的有效范围，将发生溢出。即使因相乘而发生溢出，也不会发生异常。因相乘而发生溢出时，可能运算结果与预计不同。请确保输入变量、输出变量的数据类型大小，有充分的余量，以免发生溢出。

三角函数：正弦SIN、余弦COS、正切TAN

功能说明：

输入变量为角度，单位为弧度，可以是任何数值基本数据类型。输出变量允许的数据类型为REAL和LREAL。

例程：

```
x:=SIN(0.5);
y:=COS(0.5);
z:=TAN(0.5);
```

反三角函数：反正弦ASIN、反余弦ACOS、反正切ATAN

注意：

- ✧ 输出数据以弧度为单位。
- ✧ 被定义的输出数据类型应可以存储输出结果，否则可能引起数据错误。
- ✧ 反正弦ASIN、反余弦ACOS的输入值的范围[-1,1]，不在此范围时，输出值为非数值。

例程：

```
x:=ASIN(0.5);
y:=ACOS(0.5);
z:=ATAN(0.5);
```

其他函数：

- **取余MOD：**计算除法运算时的余数。例：var1 := 9 MOD 2; // 结果为1
- **绝对值ABS：**计算一个数的绝对值。例：i:=ABS(-2); // 结果为2

  注意：输入输出数据类型可为任何数字基本数据类型。类型不同亦可，但输出数据类型应能覆盖输入变量的绝对值。

- **平方根SQRT：**计算一个数的平方根。例：q:=SQRT(16);

  注意：输入值不能小于0。

- **10为底的对数LOG：**计算以10为底的对数。例：q:=LOG(314.5);

  注意：输入值需大于0, 。若输入值为负值时，输出值为非数值；若输入值为0，输出值为−∞。

- **自然对数LN：**计算一个数的自然对数（以常数e为底数）。例：q:=LN(45);

  注意：输入值需大于0,。若输入值为负值时，输出值为非数值；若输入值为0，输出值为-∞。

- **自然常数N次幂EXP：**计算以自然常数（e）为底的N次幂。例：q:=EXP(2);
- **幂EXPT：**计算两个数的乘幂。例：var1 := EXPT(7,2); // 结果为49
- **获取字节数SIZEOF：**用来确定输入变量所占用的字节数。SIZEOF操作符通常返回一个无符号数。返回值的大小为输入变量的类型所占的字节数。

  返回值的类型根据数值大小决定。如表3.9所示。

表3.9　SIZEOF(x)的返回值类型

SIZEOF(x)的返回值	返回值的类型
0 <= x的值 < 256	USINT
256 <= x的值 < 65536	UINT
65536 <= x的值 < 4294967296	UDINT
4294967296 <= x的值	ULINT

例程：

```
arr1:ARRAY[0..4] OF INT;
var1:INT;
var1 := SIZEOF(arr1);   // Var1的结果：10
```

3.2.2　选择操作指令

- **最小值MIN:**　OUT0 := MIN(In0,In1);

计算两个或多个输入变量的最小值。

例程:

```
Var1:=MIN(90,30);
Var1:=MIN(MIN(90,30),40);// 结果：30
```

- **最大值MAX:**　OUT0 := MAX(In0,In1);

计算两个或多个输入变量的最大值。

例程:

```
Var1:=MAX(90,30);
Var1:=MAX(MAX(90,30),40);// 结果：90
```

- **限制值LIMIT:**　OUT0 := LIMIT(MN,IN,MX);

根据设定的上下限值，限制输入变量的值。输入与输出的关系如表3.10所示。

表3.10　最大值MAX指令输入与输出的关系

输入值“IN”	输出值“OUT”
“IN”<“MN”	“MN”
“MN”<=“IN”<=“MX”	“IN”
“MX”<“IN”	“MX”

注意：

✧ 输入输出数据类型不同亦可，但必须是可以比较的数据类型，同时输出数据类型应能覆盖所有输入变量的类型。如一个输入变量为BYTE，另一个输入变量为INT，则输出变量类型至少为INT。如一个输入引脚为BOOL，另一个输入引脚为Real，

输出端变量为Real是不允许的，因为BOOL型与整数或浮点数无法比较。

✧ MX的值必须大于等于MN的值，若小于MN的值时，将发生异常。

例程： Var1:=LIMIT(30,90,80); // 结果：80

- **二选一SEL：** OUT0 := SEL(G, In0, In1);

从两个操作数中选择一个。由G 决定IN0还是IN1为输出。

G为BOOL型，当 G=FALSE，OUT:= IN0;

当 G=TRUE， OUT:= IN1

例程： Var1:=SEL(TRUE,3,4); // 结果: 4

- **多选一MUX：** OUT0 := MUX(k, In0, In1, ……, InN);

多项选择操作符，k的值决定选中第几个输入为输出。

例程： Var1:=MUX(0,30,40,50,60,70,80); // 结果：30

3.2.3 数据转移指令

赋值 MOVE： OUT0 := MOVE(In0); 或 OUT0:=In0;

将一个变量或常量的值赋给另一个相应类型的变量。

例程：

```
ivar2 := MOVE(ivar1);
ivar3 := ivar1;
```

3.2.4 字逻辑运算指令

- **与 AND：** OUT0 := In0 AND In1;

按位进行与运算。

例程： var1 := 2#1001_0011 AND 2#1000_1010; // Var1中的结果：2#1000_0010

- **或 OR：** OUT0 := In0 OR In1;

按位进行或运算。若至少有一个输入位为1，结果就为1，否则为0。

例程：

```
var1 := 2#1001_0011 OR 2#1000_1010;      // var1的结果：2#1001_1011
Out := In1 OR In2 OR In3 OR In4 OR In5 OR In6;
```

- **异或 XOR：** OUT0 := In0 XOR In1;

按位进行异或运算。

例程： Var1 := 2#1001_0011 XOR 2#1000_1010; // var1结果：2#0001_1001

- **非 NOT：** OUT0 := NOT(In0);

按位进行非运算。

例程：

```
Var1 := NOT 2#1001_0011;  // var1的结果：2#0110_1100
```

3.2.5 移位操作指令

- **左移 SHL：** OUT0 := SHL(In0, num);

将操作数按位左移，左边移出的位不作处理，右边自动补0。

例程：

```
VAR
    in_byte : BYTE:=16#45;
    in_word : WORD:=16#45;
    erg_byte : BYTE;
    erg_word : WORD;
    n: BYTE :=2;
END_VAR
erg_byte:=SHL(in_byte,n);          //结果为16#14
erg_word:=SHL(in_word;n);          //结果为16#0114
```

注意：尽管byte和word形式的输入变量值相等，但根据输入变量的数据类型的不同（BYTE或WORD），erg_byte和erg_word得到不同的结果。

- **右移 SHR：** OUT0 := SHR(In0, num);

将操作数按位右移，右边移出的位不作处理，左边自动补0。

例程：

```
PROGRAM shr_st
VAR
    in_byte : BYTE:=16#45;
    in_word : WORD:=16#45;
    erg_byte : BYTE;
    erg_word : WORD;
    n: BYTE :=2;
END_VAR
erg_byte:=SHR(in_byte,n);    // 结果为11
erg_word:=SHR(in_word;n);    // 结果为0011
```

注意：这里算术运算的结果取决于输出变量的类型BYTE或WORD。

- **循环左移ROL：** OUT0 := ROL(In0, num);

将操作数按位循环左移，左边移出的位直接补到右边最低位。

例程：

```
PROGRAM rol_st
VAR
    in_byte : BYTE:=16#45;
    in_word : WORD:=16#45;
    erg_byte : BYTE;
    erg_word : WORD;
    n: BYTE :=2;
END_VAR
erg_byte:=ROL(in_byte,n);   // 结果为16#15
erg_word:=ROL(in_word,n); // 结果为16#0114
```

注意：尽管byte和word形式的输入变量值相等，但根据输入变量数据类型的不同（BYTE或WORD），erg_byte和erg_word得到不同的结果。

- **循环右移ROR：** OUT0 := ROR(In0, num);

将操作数按位循环右移，右边移出的位直接补到左边最高位。

例程：

```
VAR
    in_byte : BYTE:=16#45;
    in_word : WORD:=16#45;
    erg_byte : BYTE;
    erg_word : WORD;
    n: BYTE :=2;
END_VAR
erg_byte:=ROR(in_byte,n);      // 结果为16#51
erg_word:=ROR(in_word;n);     // 结果为16#4011
```

3.2.6 比较操作指令

- **大于GT，>** OUT0 := In0 > In1;

当第一个操作数比第二个操作数大时，返回值为TRUE。

注意：

✧ 输入变量的数据类型不同时，将扩展为覆盖所有数据类型的有效范围的类型后，进行比较。

✧ TIME型、DATA型、TOD型、DT型、STRING型只能在相同的数据类型之间比较。

若指定不同的数据类型，链接时将发生异常。

例程: VAR1 := 20 > 30; // 结果为FALSE

- **小于LT，<** OUT0 := In0 < In1;

当第一个操作数比第二个操作数小时，返回值为TRUE。

例程: VAR1 := 20 < 30; // 结果为TRUE

- **小于或等于LE，<=** OUT0 := In0 <= In1;

当第一个操作数比第二个操作数小或者等于时，返回值为TRUE。

例程: VAR1 := 20 <= 30; // 结果为TRUE

- **大于等于GE，>=** OUT0 := In0 >= In1;

当第一个操作数比第二个操作数大或者等于时，返回值为TRUE。

例程: VAR1 := 60 >= 40; // 结果为TRUE

- **等于EQ，=** OUT0 := In0 = In1;

当第一个操作数与第二个操作数等于时，返回值为TRUE。

例程: VAR1 := 40 = 40; // 结果为TRUE

- **不相等NE，<>** OUT0 := In0 <> In1;

当第一个操作数与第二个操作数不相等时，返回值为TRUE。

例程: VAR1 := 40 <> 40; // 结果为FALSE

3.2.7 字符串操作指令

- **字符串长度LEN** OUT:= LEN(STR);

返回输入字符串的长度。

例程: VarINT1 := LEN ('SUSI'); // 结果：4

- **左边取字符串LEFT** StrOut := LEFT(STR, SIZE);

从字符串左侧开始向右提取SIZE个字符，组成一个新的字符串输出。

注意:

✧ SIZE的值大于STR的字符数或在有效范围内时，不会发生异常，返回值为STR的所

有字符。

✧ SIZE的值小于等于0时，不会发生异常，返回值为空字符。

例程： VarSTRING1 := LEFT ('SUSI',3); // 结果：“SUS”

- **右边取字符串RIGHT** StrOut := RIGHT(STR, SIZE);

从字符串右侧开始向左提取SIZE个字符，组成一个新的字符串输出。

例程： VarSTRING1 := RIGHT ('SUSI',3); // 结果：“USI”

- **中间取字符串MID** StrOut := MID(STR, LEN, POS);

从字符串第POS个字符开始提取LEN个字符，组成一个新的字符串输出。

注意：

✧ STR的字符数不足POS所示位置到LEN所示位置的字符数时，不会发生异常，返回值为POS开始到STR最后的所有字符。

✧ POS所示位置大于STR的字符长度时，返回值为空字符。

✧ LED的值小于等于0时，不会发生异常，返回值为空字符。

例程： VarSTRING1 := MID ('SUSI',2,2); // 结果：“US”

- **合并字符串CONCAT** StrOut := CONCAT(STR1, STR2);

合并两个字符串。

注意： 输出字符串个数限制为255个

例程： VarSTRING1 := CONCAT ('SUSI','WILLI'); // 结果： 'SUSIWILLI'

- **插入字符串INSERT** StrOut := INSERT(STR1, STR2, POS);

将字符串STR2插入至字符串STR1的第POS个字符后。

注意：

✧ 输出字符串个数限制为255个

✧ POS的值大于STR1的字符数时，插入后新的字符串为STR1

例程： VarSTRING1 := INSERT ('SUSI','XY',2); // 结果：“SUXYSI”

- **删除字符串DELETE**　　StrOut := DELETE(STR, LEN, POS);

在字符串STR中将第POS后的LEN个字符删除部分，剩余的字符组成字符串输出。

注意:

✧ 输入输出字符串个数限制为255个

✧ LEN的值小于等于0时，不会发生异常，返回值为STR的所有字符

例程:　　Var1 := DELETE ('SUXYSI',2,3);　// 结果：“SUSI”

- **替换字符串REPLACE**　　StrOut := REPLACE(STR1, STR2, L, P);

从字符串STR1的第P个位置（包含此位置）开始，用字符串STR2替换STR1中的L个字符。

注意:

✧ 输入输出字符串个数限制为255个

✧ L的值小于等于0时，不会发生异常，返回值为STR的所有字符

例程:　　VarSTRING1 := REPLACE ('SUXYSI','K',2,2);　// 结果：“SKYSI”

- **查找字符串FIND**　　out0 := FIND(STR1, STR2);

在检索对象STR1中检索字符串STR2第一次出现的位置。如果在STR1中没有找到STR2，则输出0。

注意:

✧ STR2的字符串应小于STR1，若非如此，返回值将为0

✧ STR1中含有多个STR2时，返回值为第一个检索到的位置

例程:　　arINT1 := FIND ('abcdef','de');　// 结果：“4”

3.3数据类型转换指令

3.3.1　　类型转换指令

布尔类型转换BOOL_TO_

从布尔类型转换为其它任意类型。

指令外观：

指令	实例
BOOL_TO_BYTE	OUT0 := BOOL_TO_BYTE(In0);
BOOL_TO_WORD	OUT0 := BOOL_TO_WORD(In0);
BOOL_TO_DWORD	OUT0 := BOOL_TO_DWORD(In0);
BOOL_TO_INT	OUT0 := BOOL_TO_INT(In0);
BOOL_TO_SINT	OUT0 := BOOL_TO_SINT(In0);
BOOL_TO_UINT	OUT0 := BOOL_TO_UINT(In0);
BOOL_TO_USINT	OUT0 := BOOL_TO_USINT(In0);
BOOL_TO_DINT	OUT0 := BOOL_TO_DINT(In0);
BOOL_TO_UDINT	OUT0 := BOOL_TO_UDINT(In0);
BOOL_TO_LINT	OUT0 := BOOL_TO_LINT(In0);
BOOL_TO_ULINT	OUT0 := BOOL_TO_ULINT(In0);
BOOL_TO_REAL	OUT0 := BOOL_TO_REAL(In0);
BOOL_TO_LREAL	OUT0 := BOOL_TO_LREAL(In0);
BOOL_TO_STRING	OUT0 := BOOL_TO_STRING(In0);
BOOL_TO_TIME	OUT0 := BOOL_TO_TIME(In0);
BOOL_TO_TOD	OUT0 := BOOL_TO_TOD(In0);
BOOL_TO_DATE	OUT0 := BOOL_TO_DATE(In0);
BOOL_TO_DT	OUT0 := BOOL_TO_DT(In0);

变量：

输入：In0：布尔型（BOOL），输入为布尔值。

输出：OUT0：任意类型，输出对应类型的值。

功能说明：

转换为数字类型时，若操作数为TRUE，结果为1；若操作数为FALSE，结果为0。转换为字符串类型时，若操作数为TRUE，结果为“TRUE”，若操作数为FALSE，则结果为“FALSE”。

例程：

```
i:=BOOL_TO_INT(TRUE);                (* 结果为1 *)
str:=BOOL_TO_STRING(TRUE);           (* 结果为“TRUE” *)
t:=BOOL_TO_TIME(TRUE);               (* 结果为T#1ms *)
tof:=BOOL_TO_TOD(TRUE);              (* 结果为TOD#00:00:00.001 *)
dat:=BOOL_TO_DATE(FALSE);            (* 结果为D#1970 *)
dandt:=BOOL_TO_DT(TRUE);             (* 结果为DT#1970-01-01-00:00:01 *)
```

字节类型转换BYTE_TO_

从字节类型转换为其它任意类型。

指令外观：

指令	实例
BYTE_TO_BOOL	OUT0 := BYTE_TO_BOOL(In0);
BYTE_TO_WORD	OUT0 := BYTE_TO_WORD(In0);
BYTE_TO_DWORD	OUT0 := BYTE_TO_DWORD(In0);
BYTE_TO_INT	OUT0 := BYTE_TO_INT(In0);
BYTE_TO_SINT	OUT0 := BYTE_TO_SINT(In0);
BYTE_TO_UINT	OUT0 := BYTE_TO_UINT(In0);
BYTE_TO_USINT	OUT0 := BYTE_TO_USINT(In0);
BYTE_TO_DINT	OUT0 := BYTE_TO_DINT(In0);
BYTE_TO_UDINT	OUT0 := BYTE_TO_UDINT(In0);
BYTE_TO_LINT	OUT0 := BYTE_TO_LINT(In0);
BYTE_TO_ULINT	OUT0 := BYTE_TO_ULINT(In0);
BYTE_TO_REAL	OUT0 := BYTE_TO_REAL(In0);
BYTE_TO_LREAL	OUT0 := BYTE_TO_LREAL(In0);
BYTE_TO_STRING	OUT0 := BYTE_TO_STRING(In0);
BYTE_TO_TIME	OUT0 := BYTE_TO_TIME(In0);
BYTE_TO_TOD	OUT0 := BYTE_TO_TOD(In0);
BYTE_TO_DATE	OUT0 := BYTE_TO_DATE(In0);
BYTE_TO_DT	OUT0 := BYTE_TO_DT(In0);

变量：

输入：In0：字节型（BYTE），输入为字节值。

输出：OUT0：任意类型，输出对应类型的值。

功能说明：

当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：

i:=BYTE_TO_INT(255);	(* 结果为255 *)
str:= BYTE_TO_STRING(19);	(* 结果为“19” *)
t:= BYTE_TO_TIME(200);	(* 结果为T#200ms *)
tof:= BYTE_TO_TOD(1);	(* 结果为TOD#00:00:00.001 *)
dat:= BYTE_TO_DATE(70);	(* 结果为D#1970-01-01 *)
dandt:= BYTE_TO_DT(60);	(* 结果为DT#1970-01-01-00:01:00 *)

字类型转换WORD_TO_

从字类型转换为其它数据类型。

指令外观：

指令	实例
WORD_TO_BOOL	OUT0 := WORD_TO_BOOL(In0);
WORD_TO_BYTE	OUT0 := WORD_TO_BYTE(In0);
WORD_TO_DWORD	OUT0 := WORD_TO_DWORD(In0);
WORD_TO_SINT	OUT0 := WORD_TO_SINT(In0);
WORD_TO_USINT	OUT0:= WORD_TO_USINT(In0);
WORD_TO_INT	OUT0:= WORD_TO_INT(In0);
WORD_TO_UINT	OUT0:= WORD_TO_UINT(In0);
WORD_TO_DINT	OUT0:= WORD_TO_DINT(In0);
WORD_TO_UDINT	OUT0:= WORD_TO_UDINT(In0);
WORD_TO_LINT	OUT0:= WORD_TO_LINT(In0);
WORD_TO_ULINT	OUT0:= WORD_TO_ULINT(In0);
WORD_TO_REAL	OUT0:= WORD_TO_REAL(In0);
WORD_TO_LREAL	OUT0:= WORD_TO_LREAL(In0);
WORD_TO_STRING	OUT0:= WORD_TO_STRING(In0);
WORD_TO_TIME	OUT0:= WORD_TO_TIME(In0);
WORD_TO_DATE	OUT0:= WORD_TO_DATE(In0);
WORD_TO_DT	OUT0:= WORD_TO_DT(In0);
WORD_TO_TOD	OUT0:= WORD_TO_TOD(In0);

变量：

输入：In0：字类型（WORD），输入为字类型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明：

- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：将字型65535保存为INT变量。

```
si := WORD_TO_INT(65535); //  结果：-1
```

双字类型转换DWORD_TO_

从双字类型转换为其它数据类型。

指令外观：

指令	实例
DWORD_TO_BOOL	OUT0 := DWORD_TO_BOOL(In0);
DWORD_TO_BYTE	OUT0 := DWORD_TO_BYTE(In0);
DWORD_TO_WORD	OUT0 := DWORD_TO_WORD(In0);
DWORD_TO_SINT	OUT0 := DWORD_TO_SINT(In0);
DWORD_TO_USINT	OUT0:= DWORD_TO_USINT(In0);
DWORD_TO_INT	OUT0:= DWORD_TO_INT(In0);
DWORD_TO_UINT	OUT0:= DWORD_TO_UINT(In0);
DWORD_TO_DINT	OUT0:= DWORD_TO_DINT(In0);
DWORD_TO_UDINT	OUT0:= DWORD_TO_UDINT(In0);
DWORD_TO_LINT	OUT0:= DWORD_TO_LINT(In0);
DWORD_TO_ULINT	OUT0:= DWORD_TO_ULINT(In0);
DWORD_TO_REAL	OUT0:= DWORD_TO_REAL(In0);
DWORD_TO_LREAL	OUT0:= DWORD_TO_LREAL(In0);
DWORD_TO_STRING	OUT0:= DWORD_TO_STRING(In0);
DWORD_TO_TIME	OUT0:= DWORD_TO_TIME(In0);
DWORD_TO_DATE	OUT0:= DWORD_TO_DATE(In0);
DWORD_TO_DT	OUT0:= DWORD_TO_DT(In0);
DWORD_TO_TOD	OUT0:= DWORD_TO_TOD(In0);

变量：

输入：In0：双字类型（DWORD），输入为双字类型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明：

- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 转换为布尔类型，当操作数不等于0时，结果为TRUE。当操作数为0时，结果为FALSE。

例程：将双字型129保存为SINT变量。

```
si := DWORD_TO_SINT(129);   //   结果：-127
```

整数类型转换INT_TO_

从整数类型转换为其它数据类型。

指令外观：

指令	实例
INT_TO_BOOL	OUT0 := INT_TO_BOOL(In0);
INT_TO_BYTE	OUT0 := INT_TO_BYTE(In0);
INT_TO_WORD	OUT0 := INT_TO_WORD(In0);
INT_TO_DWORD	OUT0 := INT_TO_DWORD(In0);
INT_TO_SINT	OUT0 := INT_TO_SINT(In0);
INT_TO_USINT	OUT0:= INT_TO_USINT(In0);
INT_TO_DINT	OUT0:= INT_TO_DINT(In0);
INT_TO_UDINT	OUT0:= INT_TO_UDINT(In0);
INT_TO_LINT	OUT0:= INT_TO_LINT(In0);
INT_TO_ULINT	OUT0:= INT_TO_ULINT(In0);
INT_TO_REAL	OUT0:= INT_TO_REAL(In0);
INT_TO_LREAL	OUT0:= INT_TO_LREAL(In0);
INT_TO_STRING	OUT0:= INT_TO_STRING(In0);
INT_TO_TIME	OUT0:= INT_TO_TIME(In0);
INT_TO_DATE	OUT0:= INT_TO_DATE(In0);
INT_TO_DT	OUT0:= INT_TO_DT(In0);
INT_TO_TOD	OUT0:= INT_TO_TOD(In0);

变量：

输入：In0：整型（INT），输入为整型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明：

- ✧ 如果有符号数据类型向无符号数据类型转换时，先把有符号数据类型转换为补码的形式，再进行数据转换。
- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：将整数4223保存为SINT变量。

```
si := INT_TO_SINT(4223);  // 结果：127
```

无符号整数类型转换UINT_TO_

从整数类型转换为其它数据类型。

指令外观：

指令	实例
UINT_TO_BOOL	OUT0 := UINT_TO_BOOL(In0);
UINT_TO_BYTE	OUT0 := UINT_TO_BYTE(In0);
UINT_TO_WORD	OUT0 := UINT_TO_WORD(In0);
UINT_TO_DWORD	OUT0 := UINT_TO_DWORD(In0);
UINT_TO_SINT	OUT0 := UINT_TO_SINT(In0);
UINT_TO_USINT	OUT0:= UINT_TO_USINT(In0);
UINT_TO_INT	OUT0:= UINT_TO_INT(In0);
UINT_TO_DINT	OUT0:= UINT_TO_DINT(In0);
UINT_TO_UDINT	OUT0:= UINT_TO_UDINT(In0);
UINT_TO_LINT	OUT0:= UINT_TO_LINT(In0);
UINT_TO_ULINT	OUT0:= UINT_TO_ULINT(In0);
UINT_TO_REAL	OUT0:= UINT_TO_REAL(In0);
UINT_TO_LREAL	OUT0:= UINT_TO_LREAL(In0);
UINT_TO_STRING	OUT0:= UINT_TO_STRING(In0);
UINT_TO_TIME	OUT0:= UINT_TO_TIME(In0);
UINT_TO_DATE	OUT0:= UINT_TO_DATE(In0);
UINT_TO_DT	OUT0:= UINT_TO_DT(In0);
UINT_TO_TOD	OUT0:= UINT_TO_TOD(In0);

变量：

输入：In0：无符号整型（UINT），输入为整型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明：

- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：将整数4223保存为SINT变量。

```
si := UINT_TO_SINT(4223); // 结果：127（十六进制的16#7f）
```

短整型转换SINT_TO_

从短整型（8位）转换为其它数据类型。

指令外观：

指令	实例
SINT_TO_BOOL	OUT0 := SINT_TO_BOOL(In0);
SINT_TO_BYTE	OUT0 := SINT_TO_BYTE(In0);
SINT_TO_WORD	OUT0 := SINT_TO_WORD(In0);
SINT_TO_DWORD	OUT0 := SINT_TO_DWORD(In0);
SINT_TO_USINT	OUT0 := SINT_TO_USINT(In0);
SINT_TO_INT	OUT0:= SINT_TO_INT(In0);
SINT_TO_UINT	OUT0:= SINT_TO_UINT(In0);
SINT_TO_DINT	OUT0:= SINT_TO_DINT(In0);
SINT_TO_UDINT	OUT0:= SINT_TO_UDINT(In0);
SINT_TO_LINT	OUT0:= SINT_TO_LINT(In0);
SINT_TO_ULINT	OUT0:= SINT_TO_ULINT(In0);
SINT_TO_REAL	OUT0:= SINT_TO_REAL(In0);
SINT_TO_LREAL	OUT0:= SINT_TO_LREAL(In0);
SINT_TO_STRING	OUT0:= SINT_TO_STRING(In0);
SINT_TO_TIME	OUT0:= SINT_TO_TIME(In0);
SINT_TO_DATE	OUT0:= SINT_TO_DATE(In0);
SINT_TO_DT	OUT0:= SINT_TO_DT(In0);
SINT_TO_TOD	OUT0:= SINT_TO_TOD(In0);

变量：

输入：In0：短整型（SINT），输入为短整型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明：

- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：将短整数-128保存为BYTE变量。

```
si := SINT_TO_BYTE(-128);  // 结果：128（十六进制的16#80）
```

无符号短型转换USINT_TO_

从无符号短整型转换为其它数据类型。

指令外观：

指令	实例
USINT_TO_BOOL	OUT0 := USINT_TO_BOOL(In0);
USINT_TO_BYTE	OUT0 := USINT_TO_BYTE(In0);
USINT_TO_WORD	OUT0 := USINT_TO_WORD(In0);
USINT_TO_DWORD	OUT0 := USINT_TO_DWORD(In0);
USINT_TO_SINT	OUT0 := USINT_TO_SINT(In0);
USINT_TO_UINT	OUT0:= USINT_TO_UINT(In0);
USINT_TO_INT	OUT0:= UINT_TO_INT(In0);
USINT_TO_DINT	OUT0:= USINT_TO_DINT(In0);
USINT_TO_UDINT	OUT0:= USINT_TO_UDINT(In0);
USINT_TO_LINT	OUT0:= USINT_TO_LINT(In0);
USINT_TO_ULINT	OUT0:= USINT_TO_ULINT(In0);
USINT_TO_REAL	OUT0:= USINT_TO_REAL(In0);
USINT_TO_LREAL	OUT0:= USINT_TO_LREAL(In0);
USINT_TO_STRING	OUT0:= USINT_TO_STRING(In0);
USINT_TO_TIME	OUT0:= USINT_TO_TIME(In0);
USINT_TO_DATE	OUT0:= USINT_TO_DATE(In0);
USINT_TO_DT	OUT0:= USINT_TO_DT(In0);
USINT_TO_TOD	OUT0:= USINT_TO_TOD(In0);

变量：

输入：In0：无符号短整型（USINT），输入为无符号短整型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明：

- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：将无符号短整数128保存为SINT变量。

```
si := USINT_TO_SINT(128);  // 结果：-128
```

32位整数类型转换DINT_TO_

从32位整数类型转换为其它数据类型。

指令外观：

指令	实例
DINT_TO_BOOL	OUT0 := DINT_TO_BOOL(In0);
DINT_TO_BYTE	OUT0 := DINT_TO_BYTE(In0);
DINT_TO_WORD	OUT0 := DINT_TO_WORD(In0);
DINT_TO_DWORD	OUT0 := DINT_TO_DWORD(In0);
DINT_TO_SINT	OUT0 := DINT_TO_SINT(In0);
DINT_TO_USINT	OUT0:= DINT_TO_USINT(In0);
DINT_TO_INT	OUT0:= DINT_TO_INT(In0);
DINT_TO_UINT	OUT0:= DINT_TO_UINT(In0);
DINT_TO_UDINT	OUT0:= DINT_TO_UDINT(In0);
DINT_TO_LINT	OUT0:= DINT_TO_LINT(In0);
DINT_TO_ULINT	OUT0:= DINT_TO_ULINT(In0);
DINT_TO_REAL	OUT0:= DINT_TO_REAL(In0);
DINT_TO_LREAL	OUT0:= DINT_TO_LREAL(In0);
DINT_TO_STRING	OUT0:= DINT_TO_STRING(In0);
DINT_TO_TIME	OUT0:= DINT_TO_TIME(In0);
DINT_TO_DATE	OUT0:= DINT_TO_DATE(In0);
DINT_TO_DT	OUT0:= DINT_TO_DT(In0);
DINT_TO_TOD	OUT0:= DINT_TO_TOD(In0);

变量：

输入：In0：32位整型（DINT），输入为整型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明：

- ✧ 如果有符号数据类型向无符号数据类型转换时，先把有符号数据类型转换为补码的形式，再进行数据转换。
- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：将32位整数-2147483648保存为UDINT变量。

```
si := DINT_TO_UDINT( -2147483648); //  结果：2147483648
```

32位无符号整数类型转换UDINT_TO_

从32位无符号整数类型转换为其它数据类型。

指令外观：

指令	结构文本
UDINT_TO_BOOL	OUT0 := UDINT_TO_BOOL(In0);
UDINT_TO_BYTE	OUT0 := UDINT_TO_BYTE(In0);
UDINT_TO_WORD	OUT0 := UDINT_TO_WORD(In0);
UDINT_TO_DWORD	OUT0 := UDINT_TO_DWORD(In0);
UDINT_TO_SINT	OUT0 := UDINT_TO_SINT(In0);
UDINT_TO_USINT	OUT0:= UDINT_TO_USINT(In0);
UDINT_TO_UINT	OUT0:= UDINT_TO_UINT(In0);
UDINT_TO_DINT	OUT0:= UDINT_TO_DINT(In0);
UDINT_TO_LINT	OUT0:= UDINT_TO_LINT(In0);
UDINT_TO_ULINT	OUT0:= UDINT_TO_ULINT(In0);
UDINT_TO_REAL	OUT0:= UDINT_TO_REAL(In0);
UDINT_TO_LREAL	OUT0:= UDINT_TO_LREAL(In0);
UDINT_TO_STRING	OUT0:= UDINT_TO_STRING(In0);
UDINT_TO_TIME	OUT0:= UDINT_TO_TIME(In0);
UDINT_TO_DATE	OUT0:= UDINT_TO_DATE(In0);
UDINT_TO_DT	OUT0:= UDINT_TO_DT(In0);
UDINT_TO_TOD	OUT0:= UDINT_TO_TOD(In0);

变量：

输入：In0：32位无符号整型（UDINT），输入为无符号整型值。

输出：OUT0：任意类型，转换后对应类型的值。

功能说明 ：

- ✧ 从较大数据类型转为为较小的数据类型时，有可能丢失部分信息。
- ✧ 如果需转换的数超过了数值范围，结果取相应类型的字节数，忽略超出数值范围的高字节数。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：将32位无符号整数65536保存为UINT变量。

```
si := UDINT_TO_UINT(65536);   // 结果：0
```

实数/长实数类型转换REAL/LREAL_TO_

从实数/长实数变量类型转换为其它数据类型。

指令外观：

指令	实例
REAL_TO_BOOL	OUT0 := REAL_TO_BOOL(In0);
REAL_TO_INT	OUT0 := REAL_TO_INT(In0);
REAL_TO_UINT	OUT0 := REAL_TO_UINT(In0);
REAL_TO_SINT	OUT0 := REAL_TO_SINT(In0);
REAL_TO_USINT	OUT0 := REAL_TO_USINT(In0);
REAL_TO_DINT	OUT0 := REAL_TO_DINT(In0);
REAL_TO_UDINT	OUT0 := REAL_TO_UDINT(In0);
REAL_TO_LINT	OUT0 := REAL_TO_LINT(In0);
REAL_TO_ULINT	OUT0 := REAL_TO_ULINT(In0);
REAL_TO_LREAL	OUT0 := REAL_TO_LREAL(In0);
REAL_TO_STRING	OUT0 := REAL_TO_STRING(In0);
REAL_TO_TIME	OUT0:= REAL_TO_TIME(In0);
REAL_TO_DATE	OUT0:= REAL_TO_DATE(In0);
REAL_TO_DT	OUT0:= REAL_TO_DT(In0);
REAL_TO_TOD	OUT0:= REAL_TO_TOD(In0);

LREAL型转换为其它类型，与上表相同，只需将REAL更换为LREAL即可。

变量：

输入：In0：实数类型（REAL），输入为实数值。

输出：OUT0：其它类型，转换后对应类型的值。

功能说明：

- ✧ 从实数/长实数变量类型转换为其它数据类型，数值将被四舍五入为近似的整数值，然后转换成新的变量类型。但转换为STRING、REAL和LREAL类型变量时例外。
- ✧ 如果REAL 或 LREAL 类型转换成SINT, USINT, INT, UINT, DINT, UDINT, LINT或ULINT 类型，且实型数据的值超出了整形的范围，结果将会是不确定的并且该值取决于目标系统。这种情况即使产生一个异常也是可能的! 为了获取与目标无关的代码，应由应用程序进行值域越界处理。如果REAL/LREAL 型数据在整型的值域范围内，他们之间的转换在所有系统上都可以进行。
- ✧ 请注意转换为字符串类型时，（长）实数的位数不能超过16位。如果位数太多，那么第十六位将被四舍五入。
- ✧ 从较大的数据类型转换为较小的数据类型时，有可能丢失部分信息。
- ✧ 当操作数不等于0时，转换为布尔类型则结果为TRUE；当操作数为0时，结果为FALSE。

例程：

```
i := REAL_TO_INT(1.5);   (* 结果为2 *)
j := REAL_TO_INT(1.4);   (* 结果为1 *)
i := REAL_TO_INT(-1.5);   (* 结果为-2 *)
j := REAL_TO_INT(-1.4);   (* 结果为-1 *)
```

时间/时刻类型转换<时间>_TO_

把时间（TIME）或时刻（TIME_OF_DAY）类型变量转换为其他类型变量。

指令外观：

指令	实例
TIME_TO_BOOL	OUT0 := TIME_TO_BOOL(In0);
TIME_TO_BYTE	OUT0 := TIME_TO_BYTE(In0);
TIME_TO_WORD	OUT0 := TIME_TO_WORD(In0);
TIME_TO_DWORD	OUT0 := TIME_TO_DWORD(In0);
TIME_TO_INT	OUT0 := TIME_TO_INT(In0);
TIME_TO_UINT	OUT0 := TIME_TO_UINT(In0);
TIME_TO_SINT	OUT0 := TIME_TO_SINT(In0);
TIME_TO_USINT	OUT0 := TIME_TO_USINT(In0);
TIME_TO_DINT	OUT0 := TIME_TO_DINT(In0);
TIME_TO_UDINT	OUT0 := TIME_TO_UDINT(In0);
TIME_TO_REAL	OUT0 := TIME_TO_REAL(In0);
TIME_TO_LREAL	OUT0 := TIME_TO_LREAL(In0);
TIME_TO_LINT	OUT0 := TIME_TO_LINT(In0);
TIME_TO_ULINT	OUT0 := TIME_TO_ULINT(In0);
TIME_TO_STRING	OUT0 := TIME_TO_STRING(In0);
TIME_TO_DATE	OUT0 := TIME_TO_DATE(In0);
TIME_TO_DT	OUT0 := TIME_TO_DT(In0);
TIME_TO_TOD	OUT0 := TIME_TO_TOD(In0);
TOD_TO_BOOL	OUT0 := TOD_TO_BOOL(In0);
TOD_TO_BYTE	OUT0 := TOD _TO_BYTE(In0);
TOD_TO_WORD	OUT0 := TOD _TO_WORD(In0);
TOD_TO_DWORD	OUT0 := TOD _TO_DWORD(In0);
TOD_TO_INT	OUT0 := TOD _TO_INT(In0);
TOD_TO_UINT	OUT0 := TOD _TO_UINT(In0);
TOD_TO_SINT	OUT0 := TOD _TO_SINT(In0);
TOD_TO_USINT	OUT0 := TOD _TO_USINT(In0);
TOD_TO_DINT	OUT0 := TOD _TO_DINT(In0);
TOD_TO_UDINT	OUT0 := TOD _TO_UDINT(In0);
TOD_TO_LINT	OUT0 := TOD _TO_LINT(In0);
TOD_TO_ULINT	OUT0 := TOD _TO_ULINT(In0);
TOD_TO_REAL	OUT0 := TOD _TO_REAL(In0);
TOD_TO_LREAL	OUT0 := TOD _TO_LREAL(In0);
TOD_TO_STRING	OUT0 := TOD _TO_STRING(In0);
TOD_TO_TIME	OUT0 := TOD _TO_TIME(In0);
TOD_TO_DATE	OUT0 := TOD _TO_DATE(In0);
TOD_TO_DT	OUT0 := TOD _TO_DT(In0);

变量：

输入：In0：时间类型（TIME）或时间日期类型（TOD），输入为时间或时间日期。

输出：OUT0：其它类型，转换后对应类型的值。

功能说明：

- ✧ 把时间（TIME）或时刻（TIME_OF_DAY）类型变量转换为其他类型变量。时间将以毫秒为单位以DWORD类型（对于TIME_OF_DAY变量从00:00开始）存储在内部，然后再进行转化。
- ✧ 从较大的数据类型转换为较小的数据类型时，有可能丢失部分信息。
- ✧ 转换为STRING类型变量时，转换结果为对应的时间常数。

例程：

```
str :=TIME_TO_STRING(T#12ms);          (* 结果为T#12ms *)
dw:=TIME_TO_DWORD(T#5m);               (* 结果为300000 *)
si:=TOD_TO_SINT(TOD#00:00:00.012);     (* 结果为12 *)
```

日期/日期时间类型转换DATE/DT_TO_

从日期和日期时间类型变量转换为其他类型变量。

指令外观：

指令	实例
DATE_TO_BOOL	OUT0 := DATE_TO_BOOL(In0);
DATE_TO_BYTE	OUT0 := DATE_TO_BYTE(In0);
DATE_TO_WORD	OUT0 := DATE_TO_WORD(In0);
DATE_TO_DWORD	OUT0 := DATE_TO_DWORD(In0);
DATE_TO_INT	OUT0 := DATE_TO_INT(In0);
DATE_TO_UINT	OUT0 := DATE_TO_UINT(In0);
DATE_TO_SINT	OUT0 := DATE_TO_SINT(In0);
DATE_TO_USINT	OUT0 := DATE_TO_USINT(In0);
DATE_TO_DINT	OUT0 := DATE_TO_DINT(In0);
DATE_TO_UDINT	OUT0 := DATE_TO_UDINT(In0);
DATE_TO_LINT	OUT0 := DATE_TO_LINT(In0);
DATE_TO_ULINT	OUT0 := DATE_TO_ULINT(In0);
DATE_TO_REAL	OUT0 := DATE_TO_REAL(In0);
DATE_TO_LREAL	OUT0 := DATE_TO_LREAL(In0);
DATE_TO_STRING	OUT0 := DATE_TO_STRING(In0);
DATE_TO_TIME	OUT0 := DATE_TO_TIME(In0);
DATE_TO_DT	OUT0 := DATE_TO_DT(In0);
DATE_TO_TOD	OUT0 := DATE_TO_TOD(In0);
DT_TO_BOOL	OUT0 := DT_TO_BOOL(In0);
DT_TO_BYTE	OUT0 := DT_TO_BYTE(In0);
DT_TO_WORD	OUT0 := DT_TO_WORD(In0);
DT_TO_DWORD	OUT0 := DT_TO_DWORD(In0);

DT_TO_INT	OUT0 := DT_TO_INT(In0);
DT_TO_UINT	OUT0 := DT_TO_UINT(In0);
DT_TO_SINT	OUT0 := DT_TO_SINT(In0);
DT_TO_USINT	OUT0 := DT_TO_USINT(In0);
DT_TO_DINT	OUT0 := DT_TO_DINT(In0);
DT_TO_UDINT	OUT0 := DT_TO_UDINT(In0);
DT_TO_LINT	OUT0 := DT_TO_LINT(In0);
DT_TO_ULINT	OUT0 := DT_TO_ULINT(In0);
DT_TO_REAL	OUT0 := DT_TO_REAL(In0);
DT_TO_LREAL	OUT0 := DT_TO_LREAL(In0);
DT_TO_STRING	OUT0 := DT_TO_STRING(In0);
DT_TO_TIME	OUT0 := DT_TO_TIME(In0);
DT_TO_DATE	OUT0 := DT_TO_DATE(In0);
DT_TO_TOD	OUT0 := DT_TO_TOD(In0);

变量：

输入：In0：日期类型（DATE）或日期时间类型（DT），输入为日期或日期时间。

输出：OUT0：其它类型，转换后对应类型的值。

功能说明：

- ✧ 从日期和日期时间类型变量转换为其他类型变量。日期以秒为单位，用DWORD数据类型从1970年1月1日起存储在内部。然后再进行转化。
- ✧ 从较大的数据类型转换为较小的数据类型时，有可能丢失部分信息。转换为字符串类型变量时，转换结果为日期常量。
- ✧ 转换为STRING类型变量时，转换结果为对应的日期常数。

例程：

```
b :=DATE_TO_BOOL(D#1970-01-01);                (* 结果为FALSE *)
i :=DATE_TO_INT(D#1970-01-15);                 (* 结果为29952 *)
byt :=DT_TO_BYTE(DT#1970-01-15-05:05:05);      (* 结果为129 *)
str:=DT_TO_STRING(DT#1998-02-13-14:20);        (* 结果为“DT#1998-02-13-14:20” *)
```

字符串类型转换STRING_TO_

把字符串类型变量转换为其它类型。

指令外观：

指令	实例
STRING_TO_BOOL	OUT0 :=STRING_TO_BOOL(In0);
STRING_TO_BYTE	OUT0 :=STRING_TO_BYTE(In0);
STRING_TO_WORD	OUT0 :=STRING_TO_WORD(In0);
STRING_TO_DWORD	OUT0 :=STRING_TO_DWORD(In0);
STRING_TO_INT	OUT0 :=STRING_TO_INT(In0);
STRING_TO_UINT	OUT0 :=STRING_TO_UINT(In0);
STRING_TO_DINT	OUT0 :=STRING_TO_DINT(In0);
STRING_TO_UDINT	OUT0 :=STRING_TO_UDINT(In0);
STRING_TO_SINT	OUT0 :=STRING_TO_SINT(In0);
STRING_TO_USINT	OUT0 :=STRING_TO_USINT(In0);
STRING_TO_LINT	OUT0 :=STRING_TO_LINT(In0);
STRING_TO_ULINT	OUT0 :=STRING_TO_ULINT(In0);
STRING_TO_REAL	OUT0 :=STRING_TO_REAL(In0);
STRING_TO_LREAL	OUT0 :=STRING_TO_LREAL(In0);
STRING_TO_DATE	OUT0 :=STRING_TO_DATE(In0);
STRING_TO_TIME	OUT0 :=STRING_TO_TIME(In0);
STRING_TO_TOD	OUT0 :=STRING_TO_TOD(In0);
STRING_TO_DT	OUT0 :=STRING_TO_DT(In0);

变量：

In0：字符串型（STRING），输入为字符串。

输出：OUT0：其它类型，转换后对应类型的值。

功能说明：

- ✧ 把字符串类型变量转换为其它类型，先把STRING转换为INT类型变量，然后把INT转换为其它类型。
- ✧ 当转换为BYTE类型时，由于高字节将被截去，因此结果将介于0-255之间。
- ✧ STRING类型变量的操作数中必须包含一个在目标类型变量里有效的值，否则转换的结果为0。
- ✧ 转换为BOOL类型时，当操作数为“TRUE”，转换为布尔类型变量的结果为TRUE，否则结果为FALSE。

例程：

```
b :=STRING_TO_BOOL('TRUE');      (* 结果为TRUE *)
w :=STRING_TO_WORD('abc34');     (* 结果为0 *)
t :=STRING_TO_TIME('T#127ms');   (* 结果为T#127ms *)
bv :=STRING:TO_BYTE('500');      (* 结果为244 *)
```

取整TRUNC_INT

该函数用于将REAL类型转化为INT类型。

指令外观： OUT0 := TRUNC_INT(In0);

变量 ：

输入： In0：实数类型（REAL），输入为实数。

输出： OUT0：整数类型（INT），取整后的结果。

功能说明：

- ✧ 将REAL类型转化为INT类型，实现对输入数值的取整操作。
- ✧ TRUNC_INT 对应于 CoDeSys V2.3版本中的 TRUNC， 并且在导入V2.3项目时会自动替换。请注意 TRUNC 函数的改变。

例程：

```
iVar:=TRUNC_INT(1.9);     (* Result is 1 *)
iVar:=TRUNC_INT(-1.4);    (* Result is -1 *)
```

截尾取整TRUNC

该函数用于将REAL类型转化为DINT类型。

指令外观： OUT0 := TRUNC(In0);

变量：

输入： In0：实数类型（REAL），输入为实数。

输出： OUT0：双整数类型（DINT），取整后的结果。

例程：

```
diVar:=TRUNC(1.9);   (*  结果为1 *)
diVar:=TRUNC(-1.4);  (*  结果为-1 *)
```

任意类型转换TO_

把任意类型、或任意数字类型的数据转化为另一种类型。

指令外观：

指令	实例
TO_BOOL	OUT0 := TO_BOOL(In0);
TO_BYTE	OUT0 := TO_BYTE(In0);
TO_WORD	OUT0 := TO_WORD(In0);
TO_INT	OUT0 := TO_INT(In0);
TO_REAL	OUT0 := TO_REAL (In0);
TO_STRING	OUT0 := TO_STRING (In0);
TO_TIME	OUT0 := TO_TIME (In0);
TO_DATE	OUT0 := TO_DATE (In0);
TO_DT	OUT0 := TO_DT (In0);
TO_TOD	OUT0 := TO_TOD (In0);

变量：

输入：In0：任意类型，待转换的数据。

输出：OUT0：任意类型，转换后的结果。

功能说明：从较大的数据类型转换为较小的数据类型时，有可能丢失部分信息。

例程：

```
VAR
    iVar:INT;
    bVar:BOOL;
    strVar:STRING;
    wVar:WORD;
END_VAR
wVar:=TO_WORD('123')      (* 结果:123 *)
bVar:=TO_BOOL(1);         (* 结果:TRUE *)
strVar:=TO_STRING(342);   (* 结果:'342' *)
iVar_=TO_INT(4.22);       (* 结果:4 *)
```

3.3.2 位/字节转换指令

位抽取EXTRACT

提取指定的双字中的某一位的状态，将该位的状态TRUE或FALSE输出，由应用库UTIL.Library提供。

指令外观：EXTRACT(X:= （参数）， N:= （参数）);

变量：

输入：X：双字型（DWORD），要提取的数据；

N：字节型（BYTE），指定的提取位；

输出：布尔型（BOOL）；提取出的位状态。

功能说明：

- ✧ 提取指定的双字X中的某一位N的状态，将该位的状态TRUE（1）或FALSE（0）输出。函数的输入是一个 DWORD 类型的 X，以及一个 BYTE 类型的 N。输出为一个 BOOL 值，是输入 X 的第 N位的值，函数从第零位开始计数。
- ✧ 指定位N的有效数据范围是0~31，N值在32到255之间时，N/32的余数是N的有效数据，例如N=33时相当于N=1。

例程：

```
FLAG:=EXTRACT(X:=81, N:=4); (* 结果: TRUE，因为81的二进制数是1010001，所以第4位是1 *)
FLAG:=EXTRACT(X:=33, N:=0); (* 结果: TRUE，因为33的二进制数是100001, 所以第0位是 1 *)
```

位整合PACK

该函数是将8个布尔类型的输入位B0, B1, ..., B7整合为1个字节，是UNPACK的逆操作，由应用库UTIL.Library提供。

指令外观： PACK(B0:=(参数),B1:=(参数),B2:=(参数),B3:=(参数),
B4:=(参数),B5:=(参数),B6:=(参数), B7:=(参数));

变量:

输入： B0：布尔型（BOOL）；整合前的位。
B1到B7的定义同上，B0~B7管脚对应输出字节PACK的0~7位

输出： 字节型（BYTE）；整合的字节值。

功能说明： 输入B0~B7不使用的引脚也需要填写，否则会报错。

例程：

```
PackOut:=PACK(0,0,0,1,0,0,0,1);
(* 结果: 136，因为输入的二进制数是10001000, 所以输出为136 *)
```

设置位值PUTBIT

设置X（DWORD类型）中的第N（BYTE类型）位为B（BOOL类型），从X的第0位开始算起。由应用库UTIL.Library提供。

指令外观： PUTBIT(X:= （参数）, N:= （参数）, B:= （参数）);

变量：

输入： X：双字型（DWORD），源数据；
N：字节型（BYTE），指定的赋值位；
B：布尔型（BOOL），赋值位的状态。

输出： 双字型（DWORD），指定位赋值后的新数据。

功能说明： 指定位N的有效数据范围是0~31，N值在32到255之间时，N/32的余数是N的有效数据，例如N=33时相当于N=1。

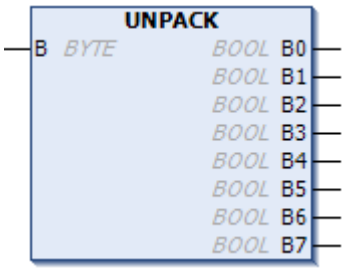
例程：

```
A:=38;                          (* 二进制数 100110 *)
B:=PUTBIT(A,4,TRUE);      (* 结果：54 = 2#110110 *)
C:=PUTBIT(A,1,FALSE);    (* 结果：36 = 2#100100 *)
```

位拆分UNPACK

该函数将输入B由BYTE类型转换为8个BOOL类型的输出变量B0,...,B7，是PACK的逆操作，由应用库UTIL.Library提供。

指令外观 ：

指令	FB/FUN	图形模块	结构文本
UNPACK	FB		UNPACK(B:= （参数）， B0=> （参数）， B1=> （参数）， B2=> （参数）， B3=> （参数）， B4=> （参数）， B5=> （参数）， B6=> （参数）， B7=> （参数）)；

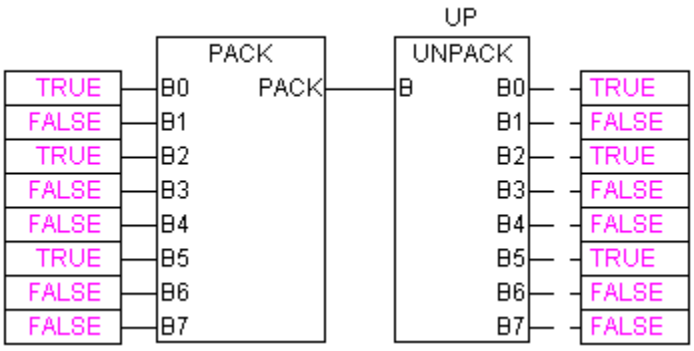
变量：

输入：B：字节型（BYTE），待拆分的源数据；

输出：B0：布尔型（BOOL），拆分后的位。

B1到B7的定义同上，B0~B7管脚对应输入字节B的0~7位。

例程：



位取反SWITCHBIT

该函数将双字X中的某一位N的开关状态取反后，输出取反后的双字值。由应用库UTIL.Library提供。

指令外观： SWITCHBIT(X:= （参数）, N:= （参数）);

变量：

输入： X：双字型（DWORD），源数据；

N：字节型（BYTE），指定的翻转位。

输出： 双字型（DWORD），指定位取反后的新数据。

功能说明： 指定位N的有效数据范围是0~31，N值在32到255之间时，N/32的余数是N的有效数据，例如N=33时相当于N=1。

例程：

```
A:=38;                    (* 二进制数 100110 *)
B:=SWITCHBIT(A,4);        (* 结果：54 = 2#110110 *)
C:= SWITCHBIT (A,1);      (* 结果：36 = 2#100100 *)
```

字拆分WORD_AS_BIT

该函数将输入字型W拆分为BOOL位B00~B15，由应用库UTIL.Library提供。

指令外观： WORD_AS_BIT(W:= （参数）, B00 => （参数）, B01 => （参数）, ……, B15 => （参数）);

变量 ：

输入： W：单字型（WORD）,待拆分的数据。

输出： B00：布尔型（BOOL）,拆分后的位。

B01~B15的定义同上B00到B15管脚对应输入字W的0~15位。

功能说明： 不使用的输出引脚B00~B15可删除或不填入变量，不影响正常使用。

例程：

```
A:=38;   (* 二进制数 100110 *)
B:=WORD_AS_BIT_0(W:=A,B00=>b0,B01=>b1,B02=>b2,B03=>b3,B04=>b4,B05=>b5);
(* 结果：b5=TRUE,b4=FALSE,b3=FALSE,b2=TRUE,b1=TRUE,b0=FALSE *)
```

位整合字BIT_AS_WORD

该函数是将16个布尔类型的输入位B00, B01, ..., B15整合为1个单字型W ，B00~B15分别对应W的0~15位，由应用库UTIL.Library提供。

指令外观： BIT_AS_WORD(B00:=（参数）,B01:=（参数）, ……, B15:=（参数）);

变量：

输入： B00：布尔型（BOOL），整合前的位；

B01到B15的定义同上，B00~B15管脚对应输出单字W的0~15位。

输出： W：单字型（WORD），整合的单字值。

功能说明：不使用的输入引脚B00~B15可删除或不填入变量，不影响正常使用。

例程：

```
BIT_AS_WORD_0(B00:=1,B01:=1,W=>d0); (* d0结果 : 3*)
```

3.3.3 BCD码转换指令

一个BCD码格式的字节包含0到99之间的整数。每4位表示一个十进制数。其中十位数存储在位4~7。所以BCD码格式和16进制表达方式类似，唯一的区别是BCD字节值是0~99，而16进制字节值是0~FF。

例如：整数51转换为BCD码格式。5的二进制表示是0101，1的二进制表示是0001，所以BCD码字节是01010001，这个值相当于16进制的51，即十进制的81。

BCD码转换成INT值BCD_TO_INT

该函数将BCD码转换成INT值，由应用库UTIL.Library提供。

指令外观： OUT0 := BCD_TO_INT(B);

变量：

输入：B：字节（BYTE）；该输入为Byte类型的BCD码。

输出：有符号整数型（INT）；BCD对应的整数型值。

功能说明：如果输入值不是BCD码（16进制数A、B、C、D、E、F），输出值是-1。

例程：

```
i:=BCD_TO_INT(73); (* 结果是 49 *)
k:=BCD_TO_INT(151); (* 结果是 97 *)
l:=BCD_TO_INT(15); (* 输出-1，因为输入值不是BCD码格式 *)
```

BCD码转换成BYTE值BCD_TO_BYTE

该函数将BCD码转换成BYTE值，由应用库UTIL.Library提供。

指令外观：OUT0 := BCD_TO_BYTE(B);

变量：

输入：B：字节（BYTE）；该输入为Byte类型的BCD码

输出：BYTE类型（INT）；输出值为字节的二进制值

功能说明：

根据输入的BCD码转换成对应的BYTE型数据，函数的输入值是BYTE类型，输出值是BYTE类型。

例程：

```
i:=BCD_TO_BYTE(73); (* 结果是 49 *)
k:=BCD_TO_ BYTE (151); (* 结果是 97 *)
l:=BCD_TO_ BYTE (15); (*结果是 15 *)
```

BCD码转换成WORD值BCD_TO_WORD

该函数将BCD码转换成WORD值，由应用库UTIL.Library提供。

指令外观： OUT0 := BCD_TO_WORD(W);

变量 ：

输入： W：单字（WORD）；该输入为单字的BCD码。

输出： 单字类型（WORD）；输出值为WORD值。

例程：

```
i:=BCD_TO_WORD (73);        (* 结果是 49 *)
k:=BCD_TO_ WORD (151);      (* 结果是 97 *)
l:=BCD_TO_ WORD (15);       (*结果是 15 *)
```

BCD码转换成DWORD值BCD_TO_DWORD

该函数将BCD码转换成DWORD值，由应用库UTIL.Library提供。

指令外观： OUT0 := BCD_TO_DWORD(X);

变量：

输入： X：双字类型（DWORD）；该输入为双字的BCD码。

输出： 双字类型（DWORD）；输出值为DWORD值。

例程：

```
i:=BCD_TO_DWORD (73);       (* 结果是 49 *)
k:=BCD_TO_DWORD (151);      (* 结果是 97 *)
l:=BCD_TO_DWORD (15);       (*结果是 15 *)
```

整数值转换成BCD码INT_TO_BCD

该函数将整数值转换成BCD码格式，由应用库UTIL.Library提供。

指令外观： OUT0 := INT_TO_BCD(I);

变量：

输入： I：整数型（INT）；该输入为INT类型的整数值。

输出： 字节型（BYTE）；返回值为字节类型的BCD码。

功能说明： 如果输入值小于0或大于99，即不能转换为BCD码，则输出为255.

例程：

```
i:=INT_TO_BCD(49);          (* 结果是 73 *)
k:=INT_TO_BCD(97);          (* 结果是 151 *)
l:=INT_TO_BCD(100);         (* 错误！输出： 255 *)
```

字节转换成BCD码BYTE_TO_BCD

该函数将字节类型的值转换成BCD码格式，由应用库UTIL.Library提供。

指令外观： OUT0 := BYTE_TO_BCD(B);

变量：

输入：B：字节型（BYTE）；该输入为范围0~99的字节类型的值。

输出：字节型（BYTE）；返回值为字节类型的BCD码。

功能说明：将字节类型的值转换成BCD码，函数的输入值是BYTE类型，输出值是BYTE类型。输入端的字节值范围0~99，如果超过99，则循环到0~99的范围输出。

例程：

```
i:=BYTE_TO_BCD(49);          (* 结果是 73 *)
k:= BYTE _TO_BCD(97);        (* 结果是 151 *)
l:= BYTE _TO_BCD(100);       (* 循环到0，结果为0 *)
```

单字转换成BCD码WORD_TO_BCD

该函数将单字类型的值转换成BCD码格式，由应用库UTIL.Library提供。

指令外观： OUTO := WORD_TO_BCD(W);

变量：

输入： W：单字型（WORD）；该输入为范围0~9999的单字类型的值。

输出： 单字型（WORD）；返回值为单字类型的BCD码。

功能说明： 将单字类型的值转换成BCD码，函数的输入值是WORD类型，输出值是WORD类型。输入值范围0~9999，如果超过9999，则循环到0~9999的范围输出。

例程：

```
i:=WORD_TO_BCD(49);          (* 结果是 73 *)
k:= WORD_TO_BCD(97);         (* 结果是 151 *)
l:= WORD_TO_BCD(10001);      (* 循环到1，结果为1 *)
```

双字转换成BCD码DWORD_TO_BCD

该函数将双字类型的值转换成BCD码格式，由应用库UTIL.Library提供。

指令外观： OUTO := DWORD_TO_BCD(X);

变量：

输入： X：双字型（DWORD）；该输入为范围0~99999999的双字类型的值。

输出： 双字型（DWORD）；返回值为双字类型的BCD码。

功能说明： 输入值范围0~99999999，如果超过99999999，则循环到0~99999999的范围输出。

例程：

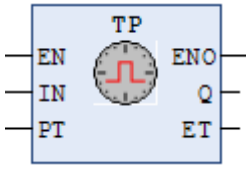
```
i:=DWORD_TO_BCD(49);             (* 结果是 73 *)
k:=DWORD_TO_BCD(97);             (* 结果是 151 *)
l:= DWORD_TO_BCD(100000002);     (* 循环到3，结果为3 *)
```

3.4定时与计数指令

3.4.1 定时器指令

启动后，时间不断增加，直至其达到指定时间。在计时期间，输出为TRUE，其他时候为FALSE。

指令外观：

指令	FB/FUN	图形模块	结构文本
TP	FB		TP(IN:= （参数）, PT:= （参数）, Q=> （参数）, ET=> （参数）);

变量：

输入： EN：功能块使能；当其为高电平时，TP功能块被激活。

IN：BOOL型，该输入端的上升沿触发定时器。

PT：TIME型，时间常量，用来设置输出ET计时上限。

输出：

ENO：辅助输出；一旦EN为高电平时，其值为高电平。

Q：BOOL型，定时器状态输出，当输出ET到达计数上限PT时，则Q输出FALSE。

ET：TIME型，输出计时实时值，从IN上升沿开始计时的时间值。

当EN为TRUE时，TP功能块被激活，ENO输出为TRUE。

功能说明：

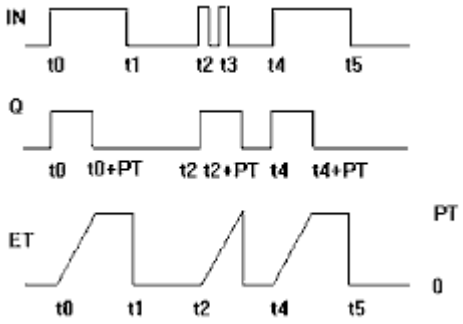
启动后，仅在设定时间内输出TRUE的定时器。

当检测到IN上升沿后，不论IN是否保持为TRUE，输出ET以毫秒精度开始计时，定时器输出Q变为TRUE，经过时间ET随着时间不断增加。

ET到达设定时间PT后，定时器输出Q变为FALSE。此时，ET停止增加。

如果IN没有检测到上升沿，Q输出为FALSE。

TP时序图：



注意事项：

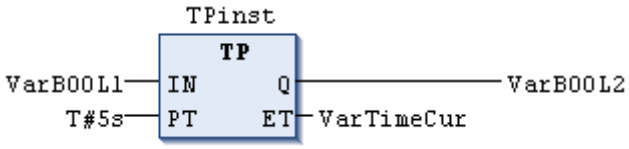
- 计时中（Q的值为TRUE），可变更PT的值。若变更后的PT>=ET，继续计时，ET的值达到变更后PT的值时，Q的值变为FALSE，ET停止增加。若变更后的PT<ET，Q的值立即变为FALSE，ET也立即停止增加。
- 计时精度为t#1ms。

例程：

变量声明示例：

```
TPinst : TP ;
```

FBD语言示例：



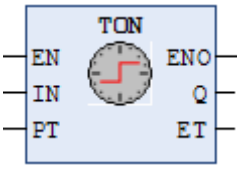
ST语言示例：

```
TPinst(IN := VarBOOL1, PT:= T#5s);
VarBOOL2 :=TPinst.Q;
```

通电延时定时器TON

当定时器的输入端变为TRUE时，经过设定的时间后，定时器的输出端才变为TRUE。

指令外观：

指令	FB/FUN	图形模块	结构文本
TON	FB		TON(IN:= （参数）， PT:= （参数）， Q=> （参数）， ET=> （参数）)；

变量：

输入： EN：功能块使能；当其为高电平时，TON功能块被激活。

IN：BOOL型，该输入端的上升沿触发通电延时定时器。

PT：TIME型，时间常量，用来设置通电延时时间。

输出： ENO：辅助输出；一旦EN为高电平时，其值为高电平。

Q：BOOL型，定时器状态输出，当输出ET到达延时时间PT时，则Q输出TRUE。

ET：TIME型，输出延时实时值，从IN上升沿开始计时的时间值。

当EN为TRUE时，TON功能块被激活，ENO输出为TRUE。

功能说明：

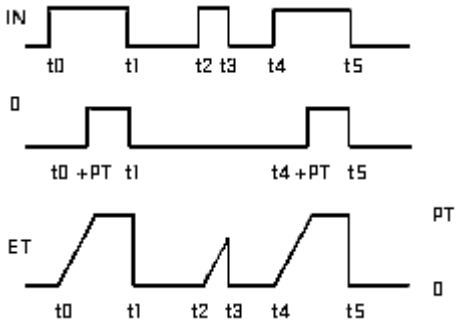
从启动起经过设定时间后，输出TRUE的定时器。

当IN为FALSE时,Q为FALSE，ET为0。

当检测到IN上升沿后，输出端ET以精确到毫秒级别开始计时，如果IN持续保持为TRUE，继续正常计时，到达设定时间PT后，定时器状态输出Q为TRUE。

如果计时到达设定时间PT之前，输入IN由TRUE变为FALSE，则定时器状态输出Q为FALSE，此时ET变为0。

TON时序图：



注意事项：

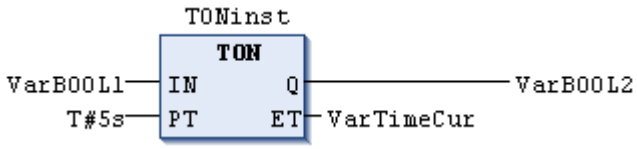
- 计时中，可变更PT的值。若变更后的PT>=ET，继续计时，ET的值达到变更后PT的值时，Q的值变为TRUE，ET停止增加。若变更后的PT<ET，Q的值立即变为TRUE，ET也立即停止增加。
- 计时精度为t#1ms。

例程：

变量声明示例：

```
TONInst : TON ;
```

FBD语言示例：



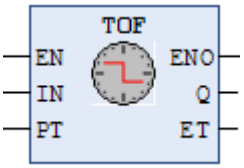
ST语言示例：

```
TONInst(IN := VarBOOL1, PT:= T#5s);
```

断电延时定时器TOF

当定时器的输入端由TRUE变为FALSE时（下降沿），等过了一段时间后，定时器的输出端才变为FALSE。

指令外观：

指令	FB/FUN	图形模块	结构文本
TOF	FB		TOF(IN:= （参数）， PT:= （参数）， Q=> （参数）， ET=> （参数） ）;

变量：

输入： EN：功能块使能；当其为高电平时，TOF功能块被激活。

IN：BOOL型，该输入端的下降沿触发断电延时定时器。

PT：TIME型，时间常量，用来设置断电延时时间。

输出： ENO：辅助输出；一旦EN为高电平时，其值为高电平。

Q：BOOL型，定时器状态输出，当输出ET到达延时时间PT时，则Q输出FALSE。

ET：TIME型，输出延时实时值，从IN下降沿开始计时的时间值。

当EN为TRUE时，TOF功能块被激活，ENO输出为TRUE。

功能说明：

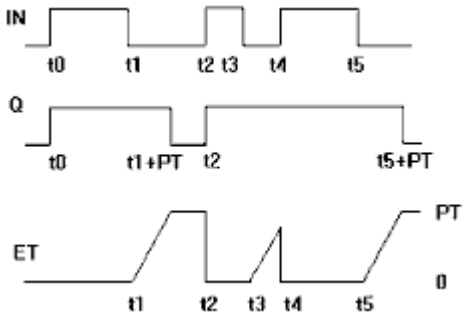
从启动起经过设定时间后，输出FALSE的定时器。

当IN为TRUE时,Q为TRUE，ET为0。

当检测到IN下降沿后，输出端ET以精确到毫秒级别开始计时，如果IN持续保持为FALSE，继续正常计时，到达设定时间PT后，定时器状态输出Q为TRUE。

如果计时到达设定时间PT之前，输入IN由FALSE变为TRUE，则定时器状态输出Q还是为TRUE，此时ET变为0。

TOF时序图：



注意事项：

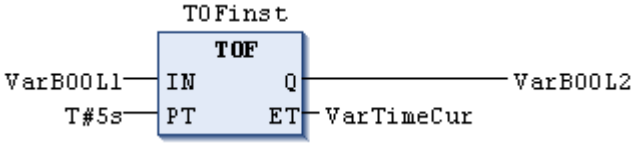
- 计时中，可变更PT的值。若变更后的PT>=ET，继续计时，ET的值达到变更后PT的值时，Q的值变为FALSE，ET停止增加。若变更后的PT<ET，Q的值立即变为FALSE，ET也立即停止增加。
- 计时精度为t#1ms。

例程：

变量声明示例：

```
TOFinst : TOF ;
```

FBD语言示例：



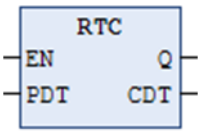
ST语言示例：

```
TOFinst(IN := VarBOOL1, PT:= T#5s);
VarBOOL2 :=TOFInst.Q;
```

实时时钟RTC

实时时钟定时器功能块，返回从给定时间开始计时的当前日期和时间。

指令外观：

指令	FB/FUN	图形模块	结构文本
RTC	FB		RTC(EN:= （参数）， PDT:= （参数）， Q=> （参数）， CDT=> （参数）)；

变量：

输入： EN：BOOL型；当其为高电平时，该输入端的上升沿设置PDT值为当前实时时钟。

PDT：DATE_AND_TIME型，日期时间常量，用来设置日期和时间。

输出： Q：BOOL型，实时时钟状态输出，当EN输入为TRUE后CDT开始计时，则Q输出TRUE。

CDT：DATE_AND_TIME型，日期时间常量，输出实时时钟显示

功能说明：

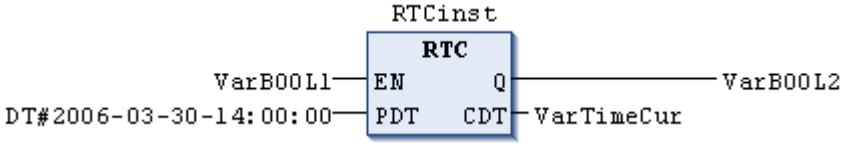
实时时钟刷新，返回从给定时间开始计时的当前日期和时间。

当EN为FALSE时,输出变量Q为FASLE，CDT输出将被复位成初始日期和时间：DT#1970-01-01-00:00:00。

一旦EN变为TRUE（上升沿），只要EN一直保持为TRUE，CDT端以PDT值作为初始值，开始以秒为精度递增，Q输出为TRUE。只要EN重新变为FALSE，Q输出位FALSE，CDT被复位为初始值DT#1970-01-01-00:00:00。

例程：

FBD语言示例：



ST语言示例：

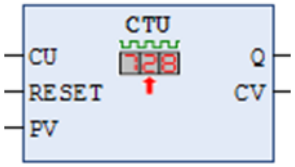
```
RTC(EN:=VarBOOL1, PDT:=DT#2006-03-30-14:00:00, Q=>VarBOOL2, CDT=>VarTimeCur);
```

3.4.2 计数器指令

递增计数器CTU

该功能块用作一个递增计数器。

指令外观：

指令	FB/FUN	图形模块	结构文本
CTU	FB		CTU(CU:= （参数）, RESET:= （参数）, PV:= （参数）, Q=> （参数）, CV=> （参数）);

变量：

输入： CU：布尔型（BOOL）；该输入端的上升沿触发CV的递增计数

RESET：布尔型（BOOL），用来复位计数器；当其为TRUE时，CV被复位为0

PV：字型（WORD）；用来设置输出CV递增计数的上限

输出： Q：布尔型（BOOL）；计数器状态输出，一旦CV达到其上限PV时，其值为TRUE

CV：字型（WORD）；该输出为递增计数实时值，按照CU上升沿依次显示从0到PV递增的数值，每次加1，直至其达到PV

功能说明：

递增计数器，每输入一次计数器输入信号，不断增加的计数器。

若RESET为TRUE时，将忽略CU，计数变量CV将变为0，计数器输出Q为FALSE。

若RESET为FALSE时，当CU端有一个从FALSE变为TRUE的上升沿时，CV将加1。当CV递增到上限PV时，Q输出为TRUE，此时CU继续输入，CV也会继续加1。

注意事项：

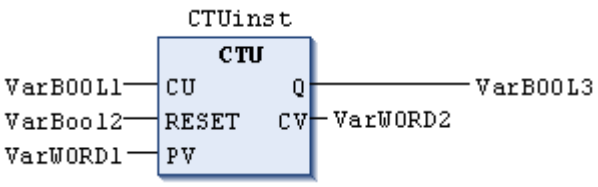
- ✧ 若要重新启动计数器，请先将RSET的值设为TRUE，再设为FALSE。
- ✧ 在RESET的值为FALSE的状态下，若PV的值发生变更，PV>CV，继续计数，PV<CV，结束计数，Q的值变为TRUE。

例程：

变量声明示例：

```
CTUinst : CTU ;
```

FBD语言示例：



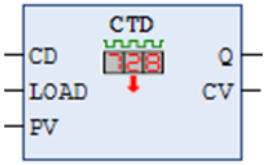
ST语言示例：

```
CTUinst(CU:= VarBOOL1, RESET:=VarBOOL2 , PV:= VarINT1);
VarBOOL3 := CTUinst.Q ;
VarWORD2 := CTUinst.CV;
```

递减计数器CTD

该功能块用作一个递减计数器。

指令外观：

指令	FB/FUN	图形模块	结构文本
CTD	FB		CTD(CD:= （参数）， LOAD:= （参数）， PV:= （参数）， Q=> （参数）， CV=> （参数）)；

变量：

输入：CD：布尔型（BOOL）；该输入端的上升沿触发CV的递减计数。

LOAD：布尔型（BOOL），用来复位计数器；当其为TRUE时CV复位为上限值PV。

PV：字型（WORD）；用来设置输出CV递减计数的初始值。

输出：Q：布尔型（BOOL）；计数器状态输出，一旦CV减到0，其值为TRUE。

CV：字型（WORD）；该输出为递增计数实时值，按照CD上升沿依次显示从PV到0递减的数值，每次减1，直至其达到0。

功能说明：

每输入一次计数器输入信号，不断减少的计数器。

当LOAD为TRUE时，预设值PV的值将载入计数值CV中，计数器输出Q变为FALSE。LOAD为TRUE期间，将忽略CD，CV不减少。

当LOAD为FALSE时，当CD端有一个从FALSE变为TRUE的上升沿时，若CV大于0时，它将减1，当CV等于0时，Q返回TRUE。

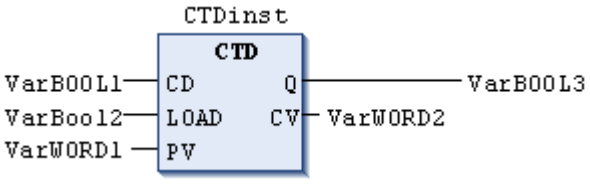
注意事项：若要重新启动计数器，请先将Load的值设为TRUE，再设为FALSE。

例程：

变量声明示例：

```
CTDinst : CTD ;
```

FBD语言示例：



ST语言示例：

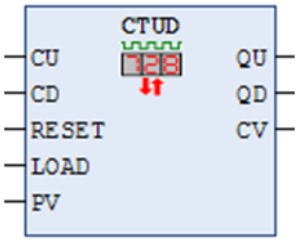
```
CTDinst(CD:= VarBOOL1, LOAD:=VarBOOL2 , PV:= VarINT1);
```

```
VarBOOL3 := CTDinst.Q ;
VarWORD2 := CTDinst.CV;
```

递增递减计数器CTUD

该功能块用作一个递增和递减计数器。

指令外观：

指令	FB/FUN	图形模块	结构文本
CTUD	FB		CTUD(CU:= （参数）, CD:= （参数）, RESET:= （参数）, LOAD:= （参数）, PV:= （参数）, QU=> （参数）, QD=> （参数）, CV=> （参数）);

变量：

输入：CU：布尔型（BOOL）；当CU端有上升沿时，触发CV的递增计数；

CD：布尔型（BOOL）；当CD端有上升沿时，触发CV的递减计数；

RESET：布尔型（BOOL）；用来复位递增计数器，当其为TRUE时，CV复位为0；

LOAD：布尔型（BOOL）；用来复位递减计数器，当其为TRUE时，CV被置为PV；

PV：字型（WORD）；用来设置输出CV递减或递增的计数上限。

输出：QU：布尔型（BOOL）；计数器状态输出，CV递增到计数上限PV时，其值为TRUE；

QD：布尔型（BOOL）；计数器状态输出，CV递减到0时，其值为TRUE；

CV：字型（WORD）；该输出为递增或递减的计数实时值。

功能说明：

根据加法计数器输入及减法计数器输入进行加减运算的计数器。也可单独作为加法计数器或减法计数器使用，使用方式分别同递增计数器CTU和递减计数器CTD。

作为加法和减法计数器共用使用时：

LOAD或RESET为TRUE时，将忽略CU及CD，CV不增减。

CU和CD同时启动时，CV无变化。

RESET和LOAD均为TRUE时，优先RESET，CV的值变为0.

若RESET设为TRUE，不论CU或CD是否检测到上升沿，都不能触发计数器，CV保持为0，QD的值为TRUE。

若LOAD设为TRUE，不论CU或CD是否检测到上升沿，都不能触发计数器，CV等于计数上限PV，QU的值为TRUE。

注意事项：

✧ 若要重置加法计数器而将RESET的值设置为TRUE，QU的值将变为FALSE，同时请注意QD的值变为TRUE。

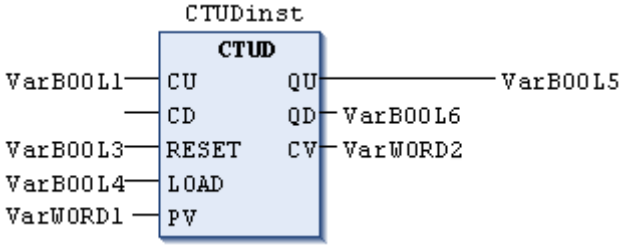
✧ 若要重置减法计数器而将LOAD的值设置为TRUE，QD的值将变为FALSE，同时请注意QU的值变为TRUE。

例程：

变量声明示例：

CTUDinst : CTUD ;

FBD语言示例：



ST语言示例：

```
CTUDinst(CU := VarBOOL1, CD:= VarBOOL2, RESET := VarBOOL3,
        LOAD:=VarBOOL4 , PV:= VarINT1);
 VarBOOL5 := CTUDinst.QU ;
 VarBOOL6 := CTUDinst.QD ;
 VarINT1 := CTUDinst.CV;
```

3.5辅助模块指令

3.5.1 地址操作指令

ADR和BITADR以及内容运算符 "^"是CoDeSys中有效的标准扩展地址运算符。

取地址ADR指令

ADR返回变量自身的地址，数据类型为DWORD。

指令外观： OUT0 := ADR(In0);

变量：

输入：操作数的数据类型：BYTE、WORD、DWORD、SINT、USINT、INT、UINT、DINT、UDINT、REAL、LREAL、TIME、DATE、TOD、DATE_AND_TIME、STRING和ARRAY

输出：指针地址：DWORD

功能说明：

ADR返回变量自身的地址，数据类型为DWORD。这个地址可以作为指针传递给操作

函数，也可以赋给工程内的某个指针。

注意事项：在线修改时，地址内容会改变，将导致POINTER变量指向不可用的内存区域。

例程：

```
dwVar:=ADR(bVAR);
```

取位地址BITADR指令

BITADR的返回值为DWORD中的位偏移量。

指令外观： `OUT0 := BITADR(In0);`

变量：

输入：操作数的数据类型：BOOL

输出：指针地址：DWORD

功能说明：

BITADR的返回值为DWORD中的位偏移量。请注意偏移量的值取决于目标设置中的字节编址是否被激活。

注意事项：使用指针的地址时，请注意申请一个可以转移地址内容的在线更改。

例程：

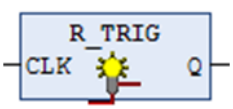
```
VAR
   var1 AT %IX2.3:BOOL;
   bitoffset: DWORD;
END_VAR

bitoffset:=BITADR(var1); (*如果字节地址=TRUE，结果为16#80000013*)
```

3.5.2 触发器指令

上升沿检测触发器R_TRIG

该功能块用作检测一个上升沿。

指令外观：

指令	FB/FUN	图形模块	结构文本
R_TRIG	FB		R_TRIG(CLK:= （参数）， Q=> （参数）);

变量：

输入：CLK：布尔型（BOOL）；被检测上升沿的输入信号。

输出：Q：布尔型（BOOL）；触发器状态输出，如果CLK检测到上升沿将输出一个扫描周期的TRUE。

功能说明：

上升沿检测，当CLK从FALSE变为TRUE时，Q输出先变为TRUE然后变为FALSE；如果CLK持续保持为FALSE或TRUE，Q输出一直保持为FALSE。

例程：

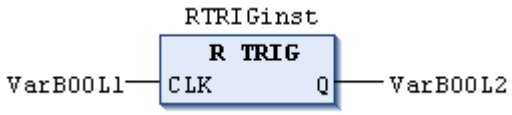
变量声明示例：

```
RTRIGinst : R_TRIG ;
```

ST 示例:

```
RTRIGinst(CLK:= VarBOOL1);
VarBOOL2 := RTRIGInst.Q;
```

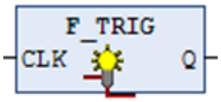
FBD语言示例：



下降沿检测触发器F_TRIG

该功能块检测一个下降沿。

指令外观：

指令	FB/FUN	图形模块	结构文本
F_TRIG	FB		F_TRIG(CLK:= （参数）， Q=> （参数）)；

变量：

输入：CLK：布尔型（BOOL）；被检测下降沿的输入信号。

输出：Q：布尔型（BOOL）；触发器状态输出，如果CLK检测到下降沿将输出一个扫描周期的TRUE。

功能说明：

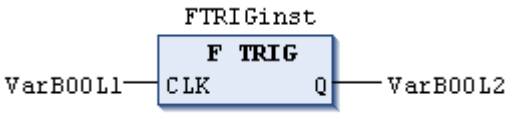
下降沿检测，当CLK从TRUE变为FALSE时，Q输出先变为TRUE然后变为FALSE；如果CLK持续保持为FALSE或TRUE，Q输出一直保持为FALSE。

例程：

变量声明示例：

```
FTRIGinst : F_TRIG ;
```

FBD语言示例：



ST语言示例：

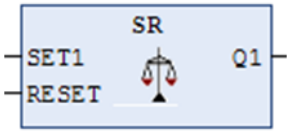
```
FTRIGinst(CLK:= VarBOOL1);
VarBOOL2 := FTRIGinst.Q;
```

3.5.3 双稳态指令

置位优先触发器SR

置位优先触发器。

指令外观：

指令	FB/FUN	图形模块	结构文本
SR	FB		SR(SET1:= （参数）， RESET:= （参数）， Q1=> （参数）)；

变量：

输入：SET1：布尔型（BOOL）；该输入端为置位输入，TRUE时置位输入；

RESET：布尔型（BOOL）；该输入端为复位输入，TRUE时复位输入。

输出：Q1：布尔型（BOOL）；触发器状态输出，一旦SET1置位，则Q1输出TRUE。

功能说明：

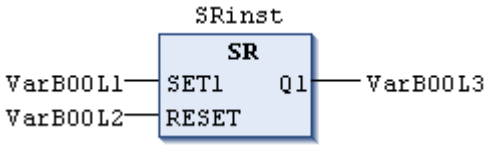
置位优先触发器。当SET1为TRUE时，不论RESET是否为TRUE，Q1输出都为TRUE；当SET1为FALSE时，如果Q1输出为TRUE，一旦RESET为TRUE，Q1输出立即复位为FALSE并保持。功能块逻辑表达式：Q1 = (NOT RESET AND Q1) OR SET1。

例程：

变量声明示例：

```
SRInst : SR ;
```

FBD语言示例：



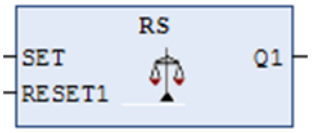
ST语言示例：

```
SRInst(SET1:= VarBOOL1 , RESET:=VarBOOL2 );
VarBOOL3 := SRInst.Q1 ;
```

复位优先触发器RS

复位优先功能块。

指令外观 ：

指令	FB/FUN	图形模块	结构文本
RS	FB		RS(SET:= （参数）， RESET1:= （参数）， Q1=> （参数）);

变量：

输入：SET：布尔型（BOOL）；该输入端为置位输入，TRUE时置位输入；

RESET1：布尔型（BOOL）；该输入端为复位输入，TRUE时复位输入。

输出：Q1：布尔型（BOOL）；触发器状态输出，一旦RESET1置位，则Q1输出FALSE。

功能说明：

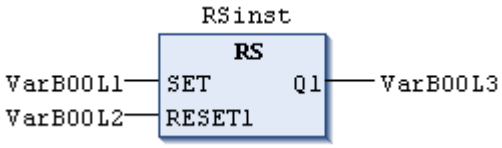
复位优先触发器。当RESET1为TRUE时，不论SET是否为TRUE，Q1输出都为FALSE；当RESET1为FALSE时，如果Q1输出为FALSE，一旦SET为TRUE，Q1输出立即置位为TRUE并保持。功能块逻辑表达式：Q1 = NOT RESET1 AND （Q1 OR SET）。

例程：

变量声明示例：

```
RSinst : RS ;
```

FBD语言示例：



ST语言示例：

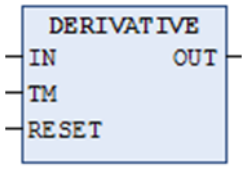
```
RSinst(SET:= VarBOOL1 , RESET1:=VarBOOL2 );
VarBOOL3 := RSinst.Q1 ;
```

3.5.4 数学辅助指令

DERIVATIVE指令

对连续输入的变量IN按照时间TM（ms）求导，结果输出到OUT。

指令外观：

指令	FB/FUN	图形模块	结构文本
DERIVATIVE	FB		DERIVATIVE(IN:= （参数）, TM:= （参数）, RESET:= （参数）, OUT=> （参数）);

变量：

输入： IN：REAL型；连续输入的变量；

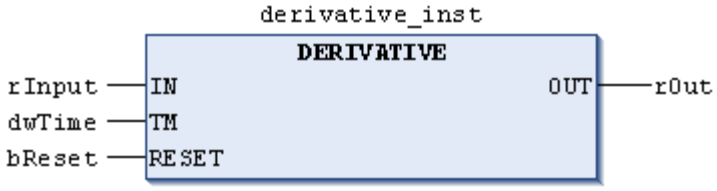
TM：DWORD型；微分时间（毫秒）。

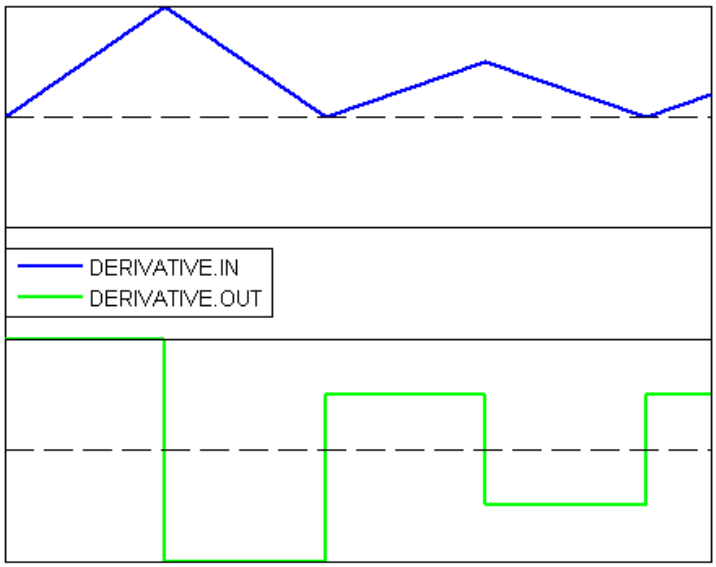
输出： OUT：REAL型；求导的结果。

功能说明：

对连续输入的变量IN按照时间TM（单位ms）求导，结果输出到OUT。为了获得最好的结果，DERIVATIVE近似使用最近的四个值，这样使输入参数不精确产生的误差尽可能小。

例程：

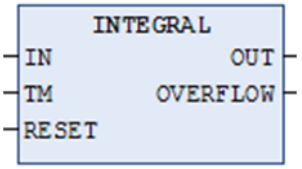




INTEGRAL指令

该功能块实现函数的积分功能，对连续输入的变量IN按照时间TM（ms）积分，结果输出到OUT。

指令外观：

指令	FB/FUN	图形模块	结构文本
INTEGRAL	FB		INTEGRAL(IN:= （参数）, TM:= （参数）, RESET:= （参数）, OUT=> （参数）, OVERFLOW=> （参数）);

变量：

输入：IN：REAL型；是被积函数的值；

TM：DWORD类型；是以毫秒为单位的时间，时间越大积分效果越强；

RESET：BOOL类型；其值为TRUE时，允许功能块重新启动。

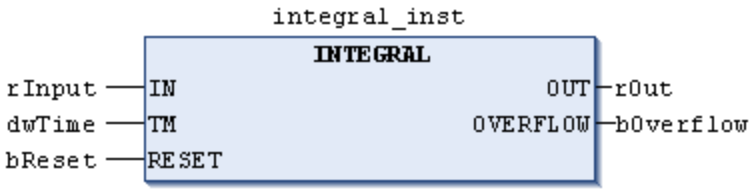
输出：

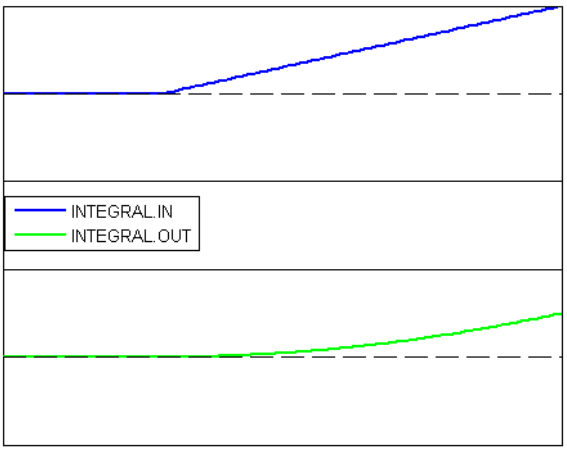
OUT：REAL型；积分的结果。

功能说明：

对连续输入的变量IN按照时间TM（ms）积分，结果输出到OUT。积分过程类似于阶跃函数，这些平均值被作为近似积分。如果积分的值超出REAL变量范围约. $\pm 10^{38}$，那么布尔输出OVERFLOW将会置TRUE，功能块被锁定直到输入RESET进行新的初始化。

例程：

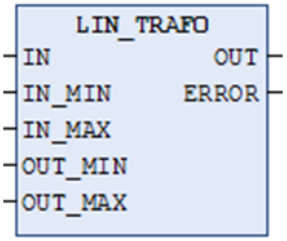




LIN_TRAFO指令

该功能块将在原始下限和上限值确定的范围内的实数，转换为由新的工程下限和上限值确定的范围内的实数。

指令外观：

指令	FB/FUN	图形模块	结构文本
LIN_TRAFO	FB		LIN_TRAFO(IN:= （参数）, IN_MIN:= （参数）, IN_MAX:= （参数）, OUT_MIN:= （参数）, OUT_MAX:= （参数）, OUT=> （参数）, ERROR=> （参数）);

变量：

输入：IN：REAL型；输入值

IN_MIN：REAL型；输入值下限

IN_MAX：REAL型；输入值上限

OUT_MIN：REAL型；输出值下限

OUT_MAX：REAL型；输出值上限

输出： OUT：REAL型；转换后输出值

ERROR：REAL型；错误输出，如果IN_MIN>=IN_MAX，或者输入值IN超出了设置值范围，即IN<IN_MIN或>IN_MAX，则发生错误，输出值为TRUE；

功能说明：

线性转换功能，将在原始下限和上限值确定的范围内的实数，转换为由新的工程下限和上限值确定的范围内的实数。转换公式为：

(IN - IN_MIN) : (IN_MAX - IN) = (OUT - OUT_MIN) : (OUT_MAX - OUT)

注意事项：

- ✧ IN_MAX必须>=IN_MIN，否则ERROR引脚会报错；
- ✧ IN需满足IN_MIN<=IN<=IN_MAX，否则ERROR引脚会报错。

例程：

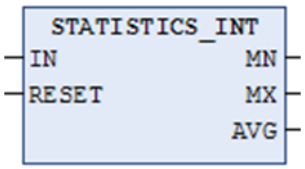
一个温度传感器提供电压值（输入值IN）。这里需要转换为以摄氏度表示的温度值（输出值OUT）。输入（电压）值的范围由IN_MIN=0和IN_MAX=10来确定。输出（摄氏度）值范围由OUT_MIN=-20 和 OUT_MAX=40来确定。

因此输入5V电压，将得到温度值为10摄氏度。

STATISTICS_INT指令

该功能块用作整型数据统计最大、最小、平均值。

指令外观：

指令	FB/FUN	图形模块	结构文本
STATISTICS_INT	FB		STATISTICS_INT(IN:= （参数）， RESET:= （参数）， MN=> （参数）， MX=> （参数）， AVG=> （参数） ）;

变量：

输入： IN：INT类型；输入的整数

RESET：BOOL类型；复位信号，其值为TRUE时，所有值重新初始化。

输出：MN：INT类型；最小值

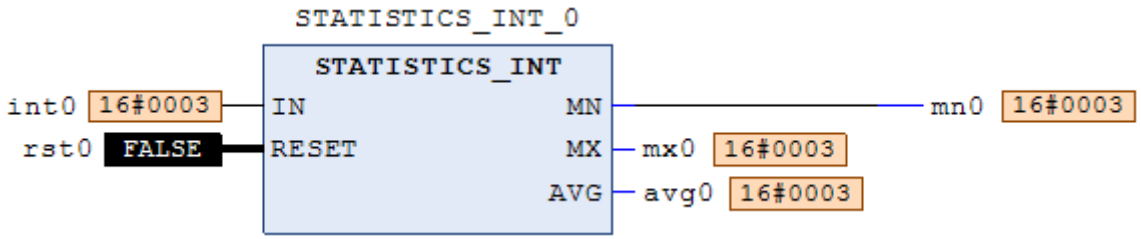
MX：INT类型；最大值

AVG：INT类型；平均值

功能说明：按统计方法计算输入整数的最大值、最小值和平均值。

注意事项：输出引脚AVG的值是以时间积分出的统计平均数，不是求和平均数

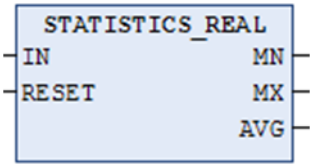
例程：



STATISTICS_REAL指令

该功能块与STATISTICS_INT相似，用作统计实数的最大、最小和平均值。

指令外观：

指令	FB/FUN	图形模块	结构文本
STATISTICS_REAL	FB		STATISTICS_REAL(IN:= （参数）， RESET:= （参数）， MN=> （参数）， MX=> （参数）， AVG=> （参数） ）;

变量：

输入：IN：REAL类型；输入的实数；

RESET：BOOL类型；其值为TRUE时，所有值重新初始化。

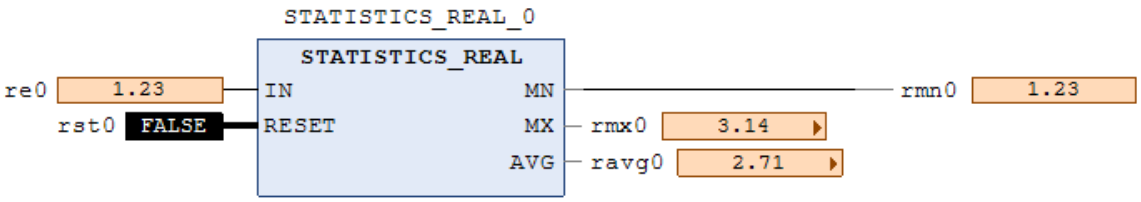
输出： MN：REAL类型，最小值；
MX：REAL类型，最大值；
AVG：REAL类型，平均值。

功能说明：

按统计方法计算输入实数的最大值、最小值和平均值。

注意事项： 输出引脚AVG的值是以时间积分出的统计平均数，不是求和平均数

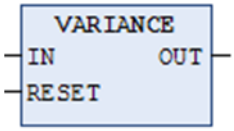
例程：



VARIANCE指令

VARIANCE用于计算输入值的方差。

指令外观：

指令	FB/FUN	图形模块	结构文本
VARIANCE	FB		VARIANCE(IN:= （参数）， RESET:= （参数）， OUT=> （参数）)；

变量：

输入： IN：REAL类型，输入的实数；
RESET：BOOL类型；复位信号，TRUE时OUT=0。

输出： OUT：REAL类型；平方方差值。

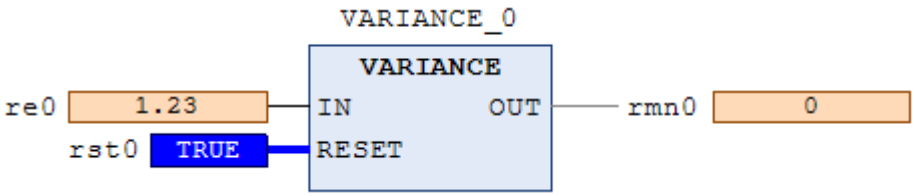
功能说明：

计算输入值的方差，方差是一个统计函数，其定义为一组样本数据的偏离程度，等于标准差的平方。

注意事项： 输入复位RESET=TRUE时，输出OUT的值为0。

例程：

当rst0=FALSE时，re0有变化时，rmn0输出连续变化的方差值，当rst0=TRUE时，输出rmn0=0。



3.5.5 调节器指令

PD比例微分控制器

指令外观：

指令	FB/FUN	图形模块	结构文本
PD	FB	PD ACTUAL Y SET_POINT LIMITS_ACTIVE KP TV Y_MANUAL Y_OFFSET Y_MIN Y_MAX MANUAL RESET	PD(ACTUAL:= （参数）， SET_POINT:= （参数）， KP:= （参数）， TV:= （参数）， Y_MANUAL:= （参数）， Y_OFFSET:= （参数）， Y_MIN:= （参数）， Y_MAX:= （参数）， MANUAL:= （参数）， RESET:= （参数）， Y=> （参数）， LIMITS_ACTIVE=> （参数）)；

变量：

输入： ACTUAL：REAL型；实际值，被控变量的当前值；

SET_POINT：REAL型；目标值，设定点；

KP：REAL型；比例系数（P）；

TV：REAL型；微分时间（D），以秒为单位；

Y_MANUAL：REAL型；当手动信号MANUAL = TRUE时Y的输出值；

Y_OFFSET：REAL型；操作变量Y的偏移量；

Y_MIN，Y_MAX：REAL型；操作变量Y的最小值和最大值。如果Y的值超限，输出LIMITS_ACTIVE将设置为TRUE，同时Y将保持在规定的范围内。这个控制只有在Y_MIN<Y_MAX的情况下起作用；

MANUAL：BOOL型；如果为TRUE，手动操作将被激活，Y=Y_MANUAL，相当于旁路PID功能；如果为FALSE，输出值Y由功能块运算得出；

RESET：BOOL型；为TRUE时复位功能块；输出Y = Y_OFFSET。

输出： Y：REAL型；功能块计算的输出值；

LIMITS_ACTIVE：BOOL型；为TRUE时，表示Y已经超出了给定的范围[Y_MIN, Y_MAX]。

功能说明：

比例微分控制器。Y_OFFSET，Y_MIN和Y_MAX用于在规定的范围之内转换操作值，MANUAL可以用于切换打开和关闭手动操作；RESET用于重新初始化控制器。

在正常操作时（MANUAL = RESET = LIMITS_ACTIVE = FALSE），控制器计算控制器偏差，也就是SET_POINT 与ACTUAL的差值，得到相对于时间的导数 dD/dt ，并且在内部保存这些值。除了当前值变化外，其他值的改变，如KP、TV等变化均在重新初始化后才能生效。

输出值Y，按照以下公式计算：$Y = KP \times (D + TV\ dD/dt) + Y\_OFFSET$

这里 D=SET_POINT-ACTUAL

所以除了P部分以外，当前控制器误差的改变（D部分）也影响操作值。

另外，Y被由 Y_MIN 和 Y_MAX.规定的范围所限制。如果Y超出范围，LIMITS_ACTIVE将为TRUE。如果不希望限制操作值， Y_MIN 和 Y_MAX必须设置为0。

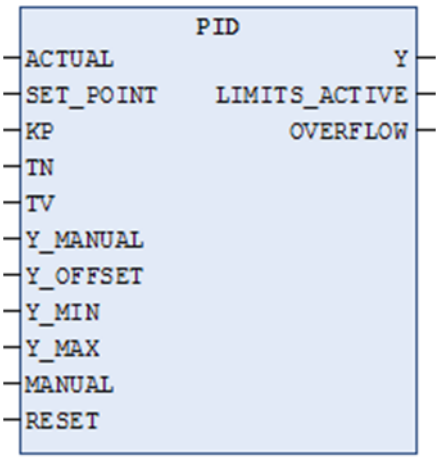
一旦MANUAL=TRUE，Y_MANUAL将赋给Y。

通过将TV设置为0，可以容易的创建一个P控制器。

注意事项：如果不希望限制操作值， Y_MIN 和 Y_MAX必须设置为0。

PID比例积分微分控制器

指令外观：

指令	FB/FUN	图形模块	结构文本
PID	FB		PID(ACTUAL:= （参数）， SET_POINT:= （参数）， KP:= （参数）， TN:= （参数）， TV:= （参数）， Y_MANUAL:=（参数）， Y_OFFSET:=（参数）， Y_MIN:= （参数）， Y_MAX:= （参数）， MANUAL:= （参数）， RESET:= （参数）， Y=> （参数）， LIMITS_ACTIVE=> （参数）， OVERFLOW=> （参数）);

变量：

输入：ACTUAL：REAL型；实际值，被控变量的当前值；

SET_POINT：REAL型；目标值，设定点；

KP：REAL型；比例系数（P）；

TN：REAL型；积分时间，I部分单位增益的倒数，以秒为单位；

TV：REAL型；微分时间（D），以秒为单位；

Y_MANUAL：REAL型；当手动信号MANUAL = TRUE时Y的输出值；

Y_OFFSET：REAL型；操作变量Y的偏移量；

Y_MIN，Y_MAX：REAL型；操作变量Y的最小值和最大值。如果Y的值超限，输出LIMITS_ACTIVE将设置为TRUE，同时Y将保持在规定的范围内。这个控制只有在Y_MIN<Y_MAX的情况下起作用；

MANUAL：BOOL型；如果为TRUE，手动操作将被激活，Y=Y_MANUAL，相当于旁路PID功能；如果为FALSE，输出值Y由功能块运算得出；

RESET：BOOL型；为TRUE时复位功能块；输出Y = Y_OFFSET，同时复位积分项；

输出： Y：REAL型；功能块计算的输出值；

LIMITS_ACTIVE：BOOL型；为TRUE时，表示Y已经超出了给定的范围[Y_MIN, Y_MAX]；

OVERFLOW：BOOL型，为TRUE时，表示积分项溢出。

功能说明：

比例积分微分控制器，和PD控制器不同，该功能块多一个积分项。

Y_OFFSET， Y_MIN和Y_MAX用于在规定的范围之内转换操作值，MANUAL可以用于切换到手动操作；RESET用于重新初始化控制器。

在正常操作时（MANUAL = RESET = LIMITS_ACTIVE = FALSE），控制器计算控制器偏差，也就是SET_POINT 与ACTUAL的差值，得到关于时间的导数de/dt，并且在内部保存这些值。

和PD控制器不同的是，输出操作值Y中包含了积分部分，按照以下公式计算：

$$Y = KP \times (D + 1/TN \int edt + TV\ dD/dt) + Y\_OFFSET$$

所以除了P部分以外，当前控制器误差的改变（D部分）和历史控制器误差（I部分）都影响操作值。

通过将TV设置为0，PID控制器可以容易的转换为一个PI控制器。

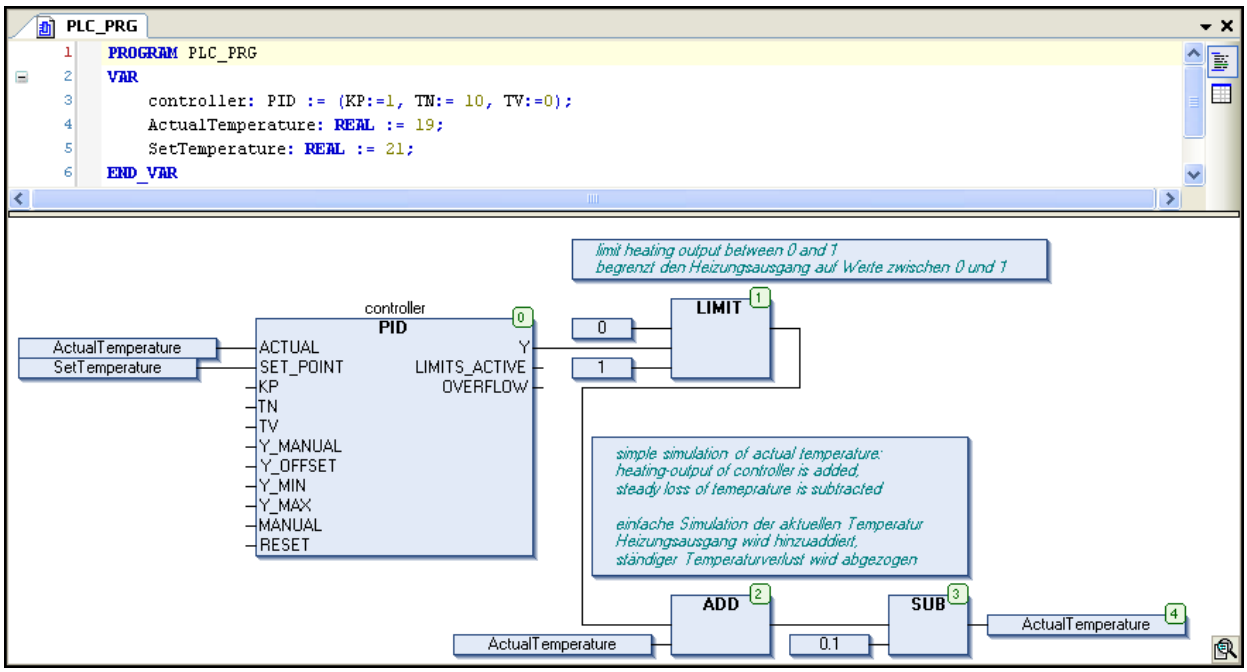
因为存在积分部分，如果误差D的积分变的过大，则可能由于控制器参数不正确而产生溢出。因此为了安全，提供一个BOOL型输出OVERFLOW，在溢出时该值为TRUE。这个仅仅发生在控制系统由于不正确的参数而不稳定时。此时，控制器将暂停，并且只有重新初始化才能再次运行。

注意事项：如果不希望限制操作值， Y_MIN 和 Y_MAX必须设置为0。

例程：

下图中使用PID模型和组合LIMIT操作符对温度控制的一个简单的例子。实际温度的输

入是通过给定一个恒定值来模拟的。



PID_FIXCYCLE比例积分微分控制器

比例积分微分控制器，与PID不同之处是采样周期是固定的，不是由功能块的内部计数器计算出来的。

指令外观：

指令	FB/FUN	图形模块	结构文本
PID_FIXCYCLE	FB	PID_FIXCYCLE ACTUAL — Y SET_POINT — LIMITS_ACTIVE KP — OVERFLOW TN TV Y_MANUAL Y_OFFSET Y_MIN Y_MAX MANUAL RESET CYCLE	PID_FIXCYCLE(ACTUAL:= （参数）， SET_POINT:= （参数）， KP:= （参数）， TN:= （参数）， TV:= （参数）， Y_MANUAL:= （参数）， Y_OFFSET:= （参数）， Y_MIN:= （参数）， Y_MAX:= （参数）， MANUAL:= （参数）， RESET:= （参数）， CYCLE:= （参数）， Y=> （参数）， LIMITS_ACTIVE=> （参数）， OVERFLOW=> （参数） ）;

变量：

输入：ACTUAL：REAL型；实际值，被控变量的当前值；

SET_POINT：REAL型；目标值，设定点；

KP：REAL型；比例系数（P）；

TN：REAL型；积分时间，I部分单位增益的倒数，以秒为单位；

TV：REAL型；微分时间（D），以秒为单位；

Y_MANUAL：REAL型；当手动信号MANUAL = TRUE时Y的输出值；

Y_OFFSET：REAL型；操作变量Y的偏移量；

Y_MIN，Y_MAX：REAL型；操作变量Y的最小值和最大值。如果Y的值超限，输出LIMITS_ACTIVE将设置为TRUE，同时Y将保持在规定的范围内。这个控制只有在Y_MIN<Y_MAX的情况下起作用；

MANUAL：BOOL型；如果为TRUE，手动操作将被激活，Y=Y_MANUAL，相当于旁路PID功能；如果为FALSE，输出值Y由功能块运算得出；

RESET：BOOL型；为TRUE时复位功能块；输出Y = Y_OFFSET，同时复位积分项；

CYCLE：REAL型；采样周期时间，单位为秒。

输出：

Y：REAL型；功能块计算的输出值；

LIMITS_ACTIVE：BOOL型；为TRUE时，表示Y已经超出了给定的范围[Y_MIN, Y_MAX]；

OVERFLOW：BOOL型，为TRUE时，表示积分项溢出。

功能说明：

该功能模块的功能和PID控制器一致，区别是其周期时间固定不变，由输入CYCLE（以秒为单位）设置，而不是通过一个内部函数自动测量。

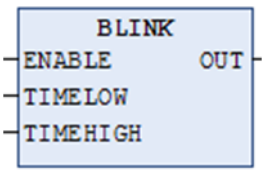
注意事项：如果不希望限制操作值， Y_MIN 和 Y_MAX必须设置为0。

3.5.6 信号发生器指令

BLINK

该功能块产生方波脉冲信号。

指令外观：

指令	FB/FUN	图形模块	结构文本
BLINK	FB		BLINK(ENABLE:= （参数）， TIMELOW:= （参数）， TIMEHIGH:= （参数）， OUT=> （参数）);

变量：

输入：ENABLE：BOOL型；功能块使能信号，高电平有效

TIMELOW：TIME型；输出脉冲信号一个周期中，低电平保持时间

TIMEHIGH：TIME型；输出脉冲信号一个周期中，高电平保持时间

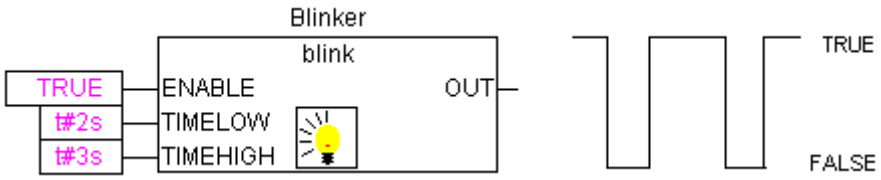
输出：OUT：BOOL型；输出脉冲信号

功能说明：

方波脉冲信号发生器。如果ENABLE为TRUE，在时间周期TIMEHIGH，BLINK设置输出为TRUE；然后在时间周期TIMELOW，设置输出为FALSE。

当ENABLE复位为FALSE，输出OUT将保持不变，也就是说不再产生脉冲。如果在ENABLE复位为FALSE时，需要准确的使OUT输出FALSE，可以使用OUT和ENABLE相与;（也就是说，还要再增加一个AND函数）。

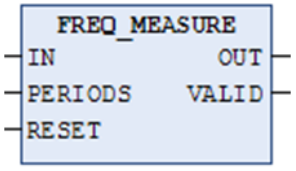
例程：



FREQ_MEASURE

该功能块测量一个布尔类型输入信号的（平均）频率（单位Hz）。

指令外观 ：

指令	FB/FUN	图形模块	结构文本
FREQ_MEASURE	FB		FREQ_MEASURE(IN:= （参数）， PERIODS:= （参数）， RESET:= （参数）， OUT=> （参数）， VALID=> （参数）);

变量：

输入： IN：BOOL型；输入信号，脉冲上升沿有效

PERIODS：INT型；输入信号采样周期，一个周期是指需要计算平均频率的输入信号的两个上升沿之间的间隔取值范围：1~10

RESET：BOOL型；复位所有参数到0，高电平有效

输出： OUT：REAL型；测量频率值，单位Hz

VALID：BOOL型；参数正确，功能块工作时为TRUE

功能说明：测量一个布尔类型输入信号的（平均）频率（单位Hz）。可以设定对多少个周期进行平均。一个周期是指输入信号的两个上升沿之间的时间间隔。

注意事项：

- ✧ PERIODS设置范围：1~10，否则功能块将不能正常工作；此参数意义为设置输入信号若干个周期测量频率的平均值，例如PERIODS=2，表示OUT输出值为2个上升沿周期测量频率平均值
- ✧ VALID正常工作时为TRUE，以下情况下，VALID为FALSE:

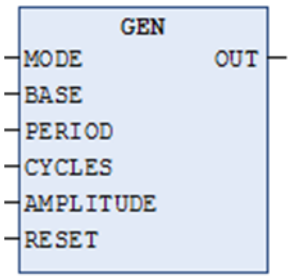
功能块参数设置错误

功能块工作时，输入信号频率为0（当输入信号2次上升沿间隔时间>3*OUT）

GEN

该功能块是函数发生器，可产生7种不同波形，并以INT变量形式输出。

指令外观：

指令	FB/FUN	图形模块	结构文本
GEN	FB		GEN(MODE:= （参数）， BASE:= （参数）， PERIOD:= （参数）， CYCLES:= （参数）， AMPLITUDE:= （参数）， RESET:= （参数）， OUT=> （参数）);

变量：

输入：

MODE：GEN_MODE枚举类型；函数波形选择，可选择的函数模式：

TRIANGLE：三角波函数，幅值-Amplitude~+Amplitude

TRIANGLE_POS：三角波函数，幅值0~+Amplitude

SAWTOOTH_RISE：上升锯齿函数

SAWTOOTH_FALL：下降锯齿函数

RECTANGLE：矩形函数

SINE：正弦函数

COSINE：余弦函数

BASE：BOOL类型；当BASE=TRUE时，函数波形的周期取决于PERIOD参数；当BASE=FALSE时，函数波形周期取决于CYCLE

PERIOD：TIME型；函数波形的周期，单位为时间

CYCLE：INT型；函数波形的周期，以GEN功能块被调用的次数表示

AMPLITUDE：INT型；函数波形的幅值

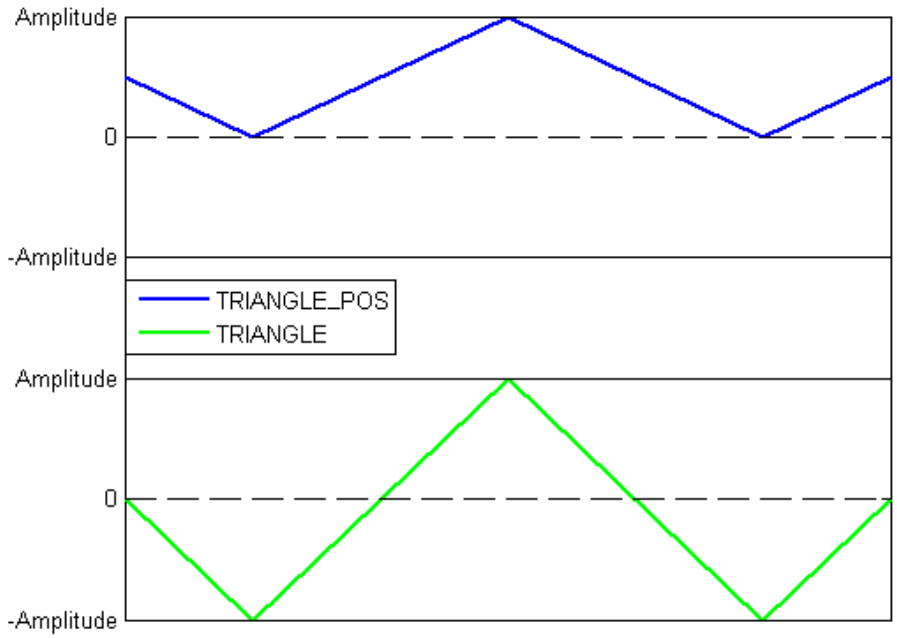
RESET：BOOL型；当RESET=TRUE时，函数发生器重新设置为0

输出：OUT：INT类型；函数发生器生成的波形输出

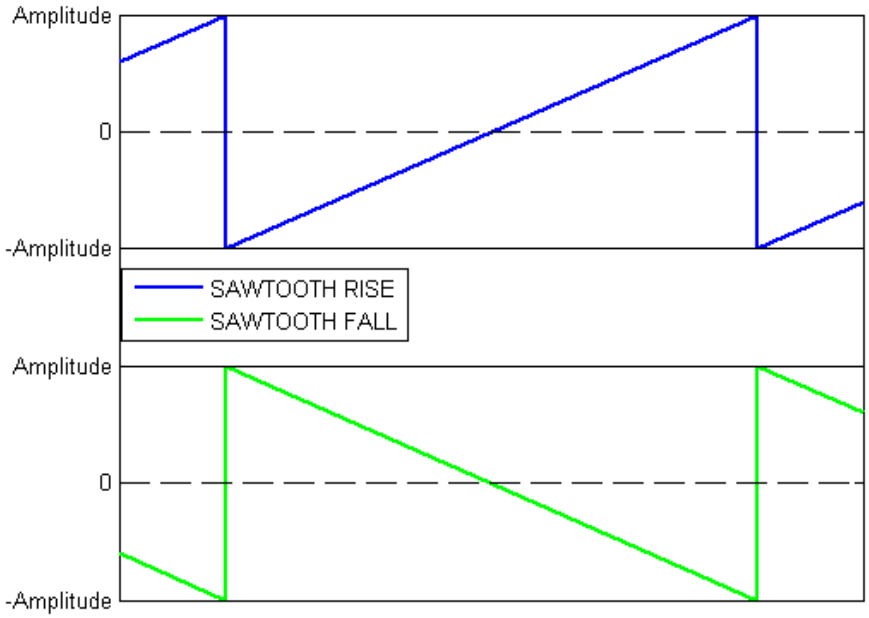
功能说明：TRIANGLE_POS生成三角函数，SAWTOOTH_RISE生成上升锯齿函数，SAWTOOTH_FALL生成下降锯齿函数，RECTANGLE生成矩形函数，SINE和COSINE分别生成正弦和余弦函数。

不同MODE类型的输出图示：

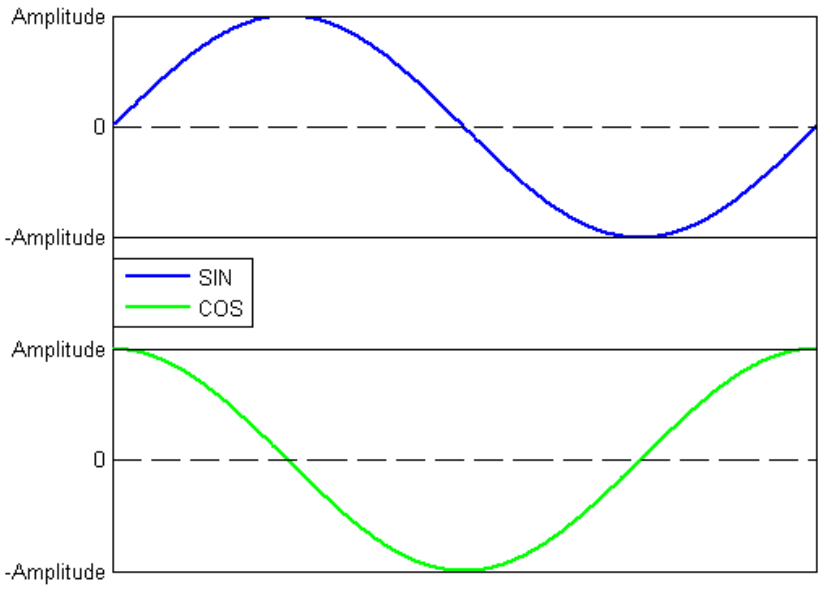
三角波TRIANGLE / TRIANGLE_POS：



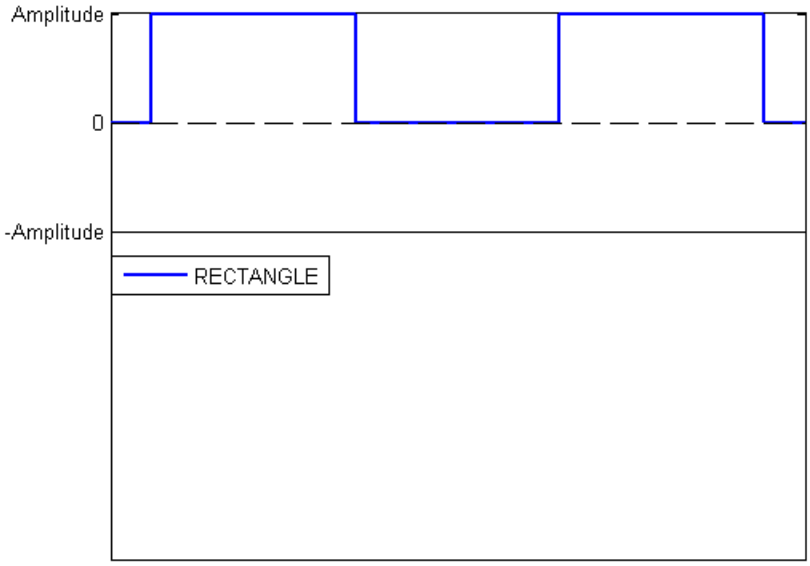
锯齿波SAWTOOTH_RISE / SAWTOOTH FALL：



正弦、余弦SINUS / COSINUS：



方波RECTANGLE：



注意事项：

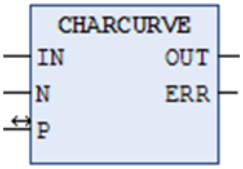
- ✧ 当BASE=FALSE时，函数周期取决于CYCLES。例如CYCLES=1000，则表示该函数波形经GEN功能块执行1000次后输出一个周期。
- ✧ 当BASE=TRUE时，函数周期取决于PERIOD值，PERIOD值不能为0，否则PLC会报程序异常错误，可在程序中对PERIOD赋予初始值。

3.5.7 函数操作指令

CHARCURVE

该功能块用于将给定的若干个数据点，分段进行线性化。

指令外观 ：

指令	FB/FUN	图形模块	结构文本
CHARCURVE	FB		CHARCURVE(IN:= （参数）， N:= （参数）， P:= （参数）， OUT=> （参数）， ERR=> （参数）);

变量：

输入：

IN：INT类型；线性化的点X坐标值

N：BYTE类型；线性化的点个数，范围[2,11]

P：ARRAY P[0..10] OF POINT；POINT类型是一个基于两个 INT 值 (X 和 Y) 的结构体，表示一个点

输出：

OUT：INT类型；点线性化后输出的Y

ERR：BYTE类型；错误代码，正常时为0

注意事项：

- ✧ 数组中的点 P[0]..P[N-1] 必须依照X值大小升序存储，否则ERR为1。
- ✧ 如果输入IN不在P[0].X 和 P[N-1].X之内，ERR=2，此时OUT=P[0]. Y或者P[N-1].Y。如果N超出了2~11之间的允许值，ERR=4。
- ✧ 输入N指定了使用P中的点的数量，如N=3，则P[0]~P[2]这3个点用于线性化。

例程：

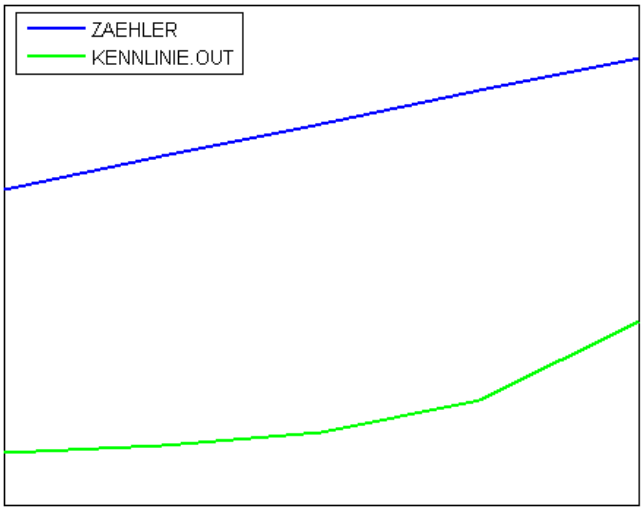
首先必须在头部定义数组P：

```
VAR
 ...
 CHARACTERISTIC_LINE:CHARCURVE;
 KL:ARRAY[0..10] OF POINT:=[(X:=0,Y:=0),(X:=250,Y:=50),
(X:=500,Y:=150),(X:=750,Y:=400),(X:=1000,Y:=1000)];
 COUNTER:INT;
 ...
END_VAR
```

然后用连续增加的值作为CHARCURVE的输入：

```
COUNTER:=COUNTER+10;
CHARACTERISTIC_LINE(IN:=COUNTER,N:=5,P:=KL);
```

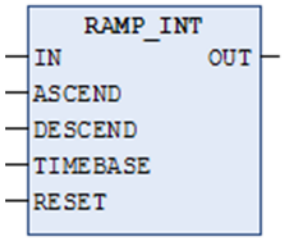
随后的跟踪显示效果：



RAMP_INT

该功能块根据输入IN值的变化，输出OUT斜坡上升或下降函数。

指令外观：

指令	FB/FUN	图形模块	结构文本
RAMP_INT	FB		RAMP_INT(IN:= （参数）， ASCEND:= （参数）， DESCEND:= （参数）， TIMEBASE:=（参数）， RESET:= （参数）， OUT=> （参数）);

变量：

输入：

IN：INT型；输入数值

ASCEND：INT型；输出OUT的上升斜坡

DESCEND：INT型；输出OUT的下降斜坡

TIMEBASE：TIME型；上升/下降斜坡的时间基准

RESET：BOOL型；功能块复位，高电平有效

输出：

OUT：INT型；输出数值

功能说明：

根据输入IN值的变化，输出OUT斜坡上升或下降函数。输入由以下三个INT类型数据组成：输入IN、在给定时间间隔内的最大增加值ASCEND和最大减小值DESCEND，该时间间隔由TIME类型的TIMEBASE来定义。RESET设置为TRUE，将使RAMP_INT重新初始化。输出OUT是INT类型，随着上升或下降斜率的变化的函数值。

当 TIMEBASE设置为t#0s时，ASCEND和DESCEND与时间间隔无关，而是保持不变。

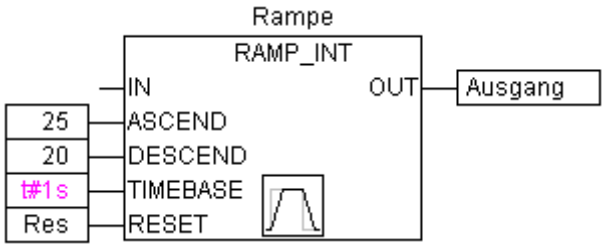
注意事项：

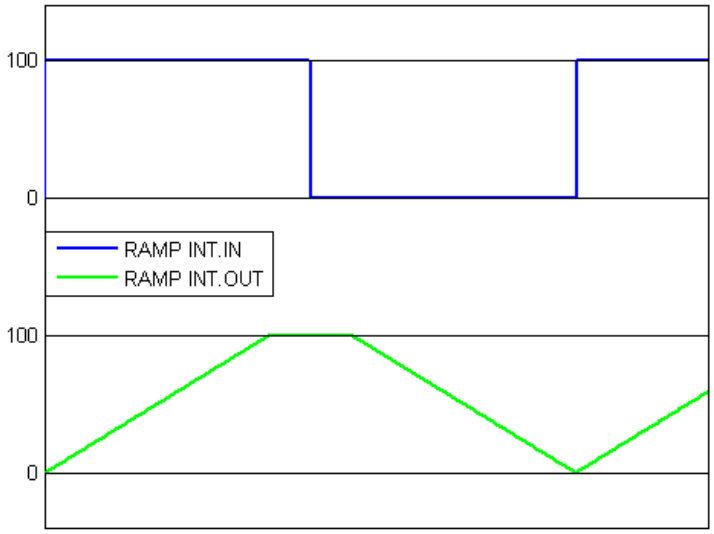
✧ ASCEND/TIMEBASE，即为上升斜率。

✧ DESCEND/TIMEBASE，即为下降斜率。

✧ 如果定义 TIMEBASE 时间比一个循环的持续时间要少那么将会发生错误。

例程：

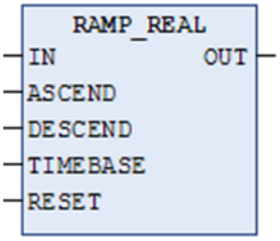




RAMP_REAL

RAMP_REAL函数和RAMP_INT相似，区别就是输入IN、ASCEND、DESCEND和输出OUT都是REAL类型。

指令外观：

指令	FB/FUN	图形模块	结构文本
RAMP_REAL	FB		RAMP_REAL(IN:= （参数）, ASCEND:= （参数）, DESCEND:= （参数）, TIMEBASE:= （参数）, RESET:= （参数）, OUT=> （参数）);

变量：

输入：

IN：REAL型；输入数值

ASCEND：REAL型；输出OUT的上升斜坡

DESCEND：REAL型；输出OUT的下降斜坡

TIMEBASE：TIME型；上升/下降斜坡的时间基准

RESET：BOOL型；功能块复位，高电平有效

输出：

OUT：REAL型；输出数值

功能说明：

根据输入IN值的变化，输出OUT斜坡上升或下降函数。输入由以下三个REAL类型数据组成：输入IN、在给定时间间隔内的最大增加值ASCEND和最大减小值DESCEND，该时间间隔由TIME类型的TIMEBASE来定义。RESET设置为TRUE，将使RAMP_INT重新初始化。输出OUT是REAL类型，随着上升或下降斜率的变化的函数值。

当 TIMEBASE设置为t#0s时，ASCEND和DESCEND与时间间隔无关，而是保持不变。

注意事项：

✧ ASCEND/TIMEBASE，即为上升斜率。

✧ DESCEND/TIMEBASE，即为下降斜率。

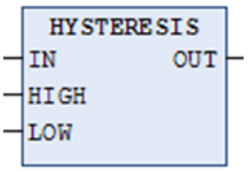
✧ 如果定义 TIMEBASE 时间比一个循环的持续时间要少那么将会发生错误。

3.5.8 模拟量监视指令

HYSTERESIS

该功能块监视输入模拟量IN的变化，并根据设定的上下限阈值，控制输出量OUT的状态。

指令外观

指令	FB/FUN	图形模块	结构文本
HYSTERESIS	FB		HYSTERESIS(IN:= （参数）, HIGH:= （参数）, LOW:= （参数）, OUT=> （参数）);

变量：

输入： IN：INT类型；输入模拟量数值

HIGH：INT类型；IN的上限阈值

LOW：INT类型；IN的下限阈值

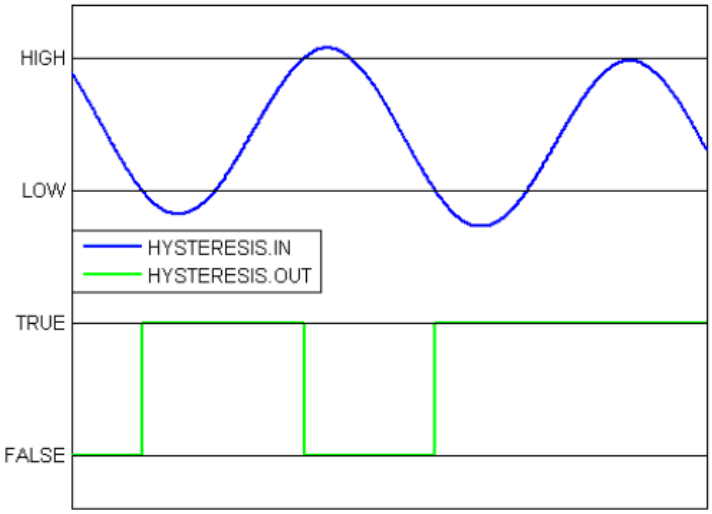
输出： OUT：BOOL类型；输出量

功能说明：

该功能块通过监视输入模拟量IN的变化，并根据设定的上下限阈值，控制输出量OUT的状态。

如果IN低于限值LOW，OUT变为TRUE。如果IN高于上限HIGH，OUT变为为FALSE。如果IN降到下限LOW,OUT将变为TRUE。下次运行IN超过上限时，输出将再次变为FALSE，直到IN再次降到LOW，因此，OUT再次获得TRUE。

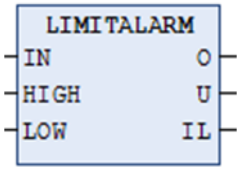
时间函数Hysteresis.IN和Hysteresis.OUT的图解比较：



LIMITALARM

该功能块根据设定的上下限阈值，输出BOOL型状态标志位，来表示输入模拟量IN当前所属的范围。

指令外观：

指令	FB/FUN	图形模块	结构文本
LIMITALARM	FB		LIMITALARM(IN:= （参数）, HIGH:= （参数）, LOW:= （参数）, O=> （参数）, U=> （参数）, IL=> （参数）);

变量：

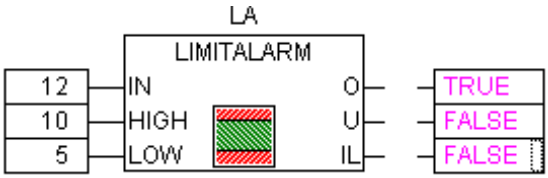
输入： IN：INT类型；输入模拟量数值
HIGH：INT类型；IN的上限阈值
LOW：INT类型；IN的下限阈值

输出： O：BOOL类型；当IN>HIGH时，O=TRUE
U：BOOL类型；当IN<LOW时，U=TRUE
IL：BOOL类型；当LOW<=IN<=HIGH时，IL=TRUE

功能说明：

根据设定的上下限阈值，输出BOOL型状态标志位，来表示输入模拟量IN当前所属的范围。如果IN高于HIGH，O为TRUE。当IN低于LOW，U为TRUE。IN位于LOW和HIGH之间时，IL为TRUE。

例程：



3.5.9 IEC扩展指令

__NEW

操作符 “__NEW”为功能块实例或标准数据类型数组分配内存。此操作符返回一个指向对象的非零指针。如果在分配时未使用该操作符，将会弹出错误信息。如果__NEW 操作不成功，将返回 0。

指令外观：

声明：pOUT: POINTER TO 数据类型

语法：pOUT:=__NEW (数据类型,长度);

功能说明 ：该操作符将创建一个指定类型的新对象并返回指向该对象的指针。在创建之后调用对象的初始化。如果返回0，表示创建失败。

要求：需在**Application**的 **属性** 对话框中，在 **应用程序生成信息** 标签中选择 **使用动态内存分配** 复选框。

如果 <类型>是标量，必须设置可选操作数<长度>，操作符以该长度创建标量类型的数组。

示例: pScalarType := __New(ScalarType, length);

注意事项：.

✧ 在线修改动态创建的对象时不能复制代码!

因此不含库(因为它们不能改变)的功能块 和具有‘enable_dynamic_creation’属性的功能块允许使用 __New操作符

如果对有此标志的功能块进行修改，拷贝代码时将产生错误信息。

✧ 内存分配的代码应该是不可重入的。

信号量 (SysSemEnter) 用于避免两个任务同时分配内存. 因此, __New 的扩展使用将会引起较高的抖动。

例程：

标量类型的示例:

```
TYPE DUT :
STRUCT
a,b,c,d,e,f : INT;
END_STRUCT
END_TYPE

PROGRAM PLC_PRG
VAR
    pDut : POINTER TO DUT;
    bInit: BOOL := TRUE;
    bDelete: BOOL;
END_VAR

IF (bInit) THEN
    pDut := __NEW(DUT);
    bInit := FALSE;
END_IF
IF (bDelete) THEN
```

```
    __DELETE(pDut);
END_IF
```

功能块示例：

```
FBDynamic(FP)
{attribute 'enable_dynamic_creation'}
FUNCTION_BLOCK FBDynamic
VAR_INPUT
    in1, in2 : INT;
END_VAR
VAR_OUTPUT
    out : INT;
END_VAR
VAR
    test1 : INT := 1234;
    _inc : INT := 0;
    _dut : POINTER TO DUT;
    neu : BOOL;
END_VAR
out := in1 + in2;

PLC_PRG(PRG)
VAR
    pFB : POINTER TO FBDynamic;
    loc : INT;
    bInit: BOOL := TRUE;
    bDelete: BOOL;
END_VAR
IF (pFB <> 0) THEN
    pFB^(in1 := 1, in2 := loc, out => loc);
    pFB^.INC();
END_IF
IF (bDelete) THEN
    __DELETE(pFB);
END_IF
```

数组示例：

```
PLC_PRG(PRG)
VAR
    bInit: BOOL := TRUE;
    bDelete: BOOL;
```

```
    pArrayBytes : POINTER TO BYTE;
    pArrayDuts : POINTER TO BYTE;
    test: INT;
    parr : POINTER TO BYTE;
END_VAR
IF (bInit) THEN
    pArrayBytes := __NEW(BYTE, 25);
    bInit := FALSE;
END_IF
IF (pArrayBytes <> 0) THEN
    pArrayBytes[24] := 125;
    test := pArrayBytes[24];
END_IF
IF (bDelete) THEN
    __DELETE(pArrayBytes);
END_IF
```

__DELETE

操作符_DELETE释放由__NEW分配的内存。__DELETE 没有返回值，其操作数执行之后将被设置为0。分配内存请使用__NEW。

指令外观：

__DELETE(<指针>)

如果指针是指向功能块，在指针被设置为NULL之前将调用专用方法 FB_Exit。

例程：

```
FUNCTION_BLOCK FBDynamic
VAR_INPUT
in1, in2 : INT;
END_VAR
VAR_OUTPUT
out : INT;
END_VAR
VAR
test1 : INT := 1234;
_inc : INT := 0;
_dut : POINTER TO DUT;
neu : BOOL;
END_VAR
out := in1 + in2;

METHOD FB_Exit : BOOL
```

```
VAR_INPUT
bInCopyCode : BOOL;
END_VAR
__Delete(_dut);

METHOD FB_Init : BOOL
VAR_INPUT
bInitRetains : BOOL;
bInCopyCode : BOOL;
END_VAR
_dut := __NEW(DUT);

METHOD INC : INT
VAR_INPUT
END_VAR
_inc := _inc + 1;
INC := _inc;

PLC_PRG(PRG)
VAR
pFB : POINTER TO FBDynamic;
bInit: BOOL := TRUE;
bDelete: BOOL;
loc : INT;
END_VAR
IF (bInit) THEN
pFB := __NEW(FBDynamic);
bInit := FALSE;
END_IF
IF (pFB <> 0) THEN
pFB^(in1 := 1, in2 := loc, out => loc);
pFB^.INC();
END_IF
IF (bDelete) THEN
__DELETE(pFB);
END_IF
```

__ISVALIDREF

用于检查一个引用是否指向一个不等于0的有效值。

指令外观：

<boolean variable> := __ISVALIDREF(identifier, declared with type <REFERENCE TO <datatype>);

例程：

```
Declaration:

ivar : INT;
ref_int : REFERENCE TO INT;
ref_int0: REFERENCE TO INT;
testref: BOOL := FALSE;

Implementation

ivar := ivar + 1;
ref_int REF= hugo;
ref_int0 REF= 0;
testref := __ISVALIDREF(ref_int); // will be TRUE, because ref_int points to ivar, which is unequal 0
testref0 := __ISVALIDREF(ref_int0); // will be FALSE, because ref_int is set to 0
```

__QUERYINTERFACE

在运行时__QUERYINTERFACE允许接口引用的类型转换。该运算符返回一个BOOL类型的结果。TRUE意味着转换成功执行。

指令外观:

__QUERYINTERFACE(<ITF_Source>, < ITF_Dest>)

这个运算符的第一个参数是一个接口引用或者预期类型的功能块实例，第二个参数是一个接口引用。如果从ITF_Source引用的对象实现了这个接口，那么在执行__QUERYINTERFACE之后，ITF_Dest保留了一个期望接口的引用。此时转换成功，运算符返回结果TRUE。否则返回FALSE。

显式转换的先决条件是，ITF_Source和ITF_Dest都必须是接口__System.IQueryInterface的扩展。该接口是隐式提供的，不需任何库。

例程:

```
INTERFACE ItfBase EXTENDS __System.IQueryInterface
METHOD mbase : BOOL
END_METHOD

INTERFACE ItfDerived1 EXTENDS   __System.IQueryInterface
```

```
METHOD mderived1 : BOOL
END_METHOD

INTERFACE ItfDerived2 EXTENDS  __System.IQueryInterface
METHOD mderived2 : BOOL
END_METHOD

PROGRAMM POU
VAR
    itfderived1 : ItfDerived1;
    itfderived2 : ItfDerived2;
    bTest1, bTest2, xResult1, xResult2: BOOL;
END_VAR

xResult1 := __QUERYINTERFACE(itfbase, itfderived1); // ItfBase类型的变量，也就是ItfDerived1
xResult2 := __QUERYINTERFACE(itfbase, itfderived2); // ItfBase类型的变量，也就是ItfDerived2
IF (xResult1 = TRUE) THEN
    bTest1 := itfderived1.mderived1();
ELSIF xResult2 THEN
    bTest2 := itfderived2.mderived2();
END_IF
```

__QUERYPOINTER

在运行时__QUERYPOINTER分配了一个无类型指针的接口引用。该运算符返回一个BOOL类型的结果。TRUE意味着转换成功执行。

指令外观：

_QUERYPOINTER(,)(<ITF_Source>, < ITF_Dest>)

这个运算符的第一个参数是一个接口引用或者期望类型的功能块实例，第二个参数是无类型指针。在执行__QUERYPOINTER之后，Pointer_Dest保留了一个期望接口的引用地址。此时转换成功，运算符返回结果TRUE。否则返回FALSE。Pointer_Dest是无类型的，并且能被转成任何类型。实际类型应由程序员确定。例如，接口可能提供一个返回类型码的方法。

显式转换的先决条件是，ITF_Source和ITF_Dest都必须是接口__System.IQueryInterface的扩展。该接口是隐式提供的，不需任何库。

例程：

```
INTERFACE ItfBase EXTENDS __System.IQueryInterface
METHOD mbase : BOOL
END_METHOD

INTERFACE ItfDerived EXTENDS ItfBase
```

```
METHOD mderived1 : BOOL
END_METHOD

FUNCTION_BLOCK FBVariante IMPLEMENTS ITFDerived
PROGRAMM POU
VAR
    itfderived : ItfDerived;
    insV : FBVariante;
    xResult, xTest : BOOL;
    pVar: POINTER TO DWORD;
END_VAR

itfderived := insV;
xResult := __QUERYPOINTER(itfderived, pVar);
IF xResult THEN
    xTest := pVar.mderived();
END_IF
```

第4章　雷赛专用指令

本章主要是介绍PMC600系列产品封装库（PMC_Controller库、文件操作库FileManage、Communication通讯库）的指令，以便于用户理解和使用。

4.1　PMC_Controller库的指令

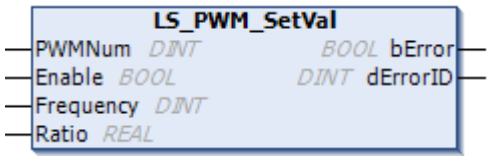
4.1.1　PWM输出指令

PWM相关指令包含在PMC_Controller库中。程序中若想使用这些功能，必须确认库中是否存在PMC_Controller文件，如不存在，需在工程中添加PMC_Controller库。

设置PWM输出 LS_PWM_SetVal

该功能块实现PWM波形输出。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_PWM_SetVal	FUN		LS_PWM_SetVal(PWMNum:= , Enable:= , Frequency:= , Ratio:= , bError=> , dErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	PWMNum	DINT	[0,3]	0	PWM通道：0~3
	Enable	BOOL	TRUE/FALSE	FALSE	False-禁止PWM；True-使能PWM，状态需要保持
	Frequency	DINT	[1,500000]	1	PWM的频率1~500000Hz
	Ratio	REAL	[0,1]	0	PWM的占空比0%~100%
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误；True-输入参数越界
	dErrorID	DINT	遵照数据类型	0	错误码

功能说明：

按设置的PWM端口号、频率输出及占空比输出PWM波形。

PMC610运动控制器提供了4路PWM（脉冲宽度调制）输出信号，且有光电隔离电路，

频率和占空比可调，输出频率范围：1HZ~500KHZ，其输出端口及接线方式见硬件手册。其输出波形如图4.1所示，PWM波形的周期为t2（频率即为1/t2），占空比为t1/t2，幅值为V1 = 5V，**一般用于控制变频器等**。

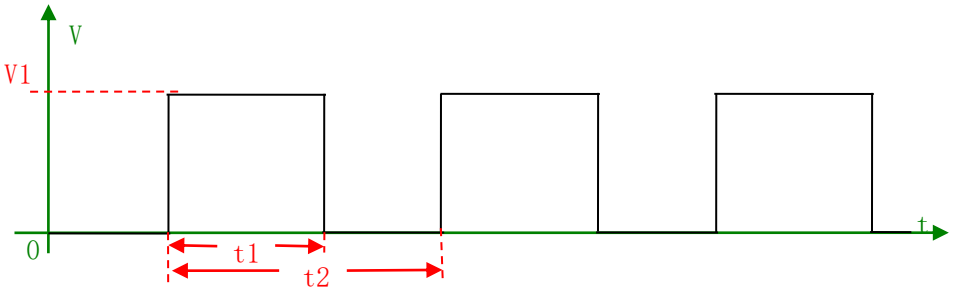
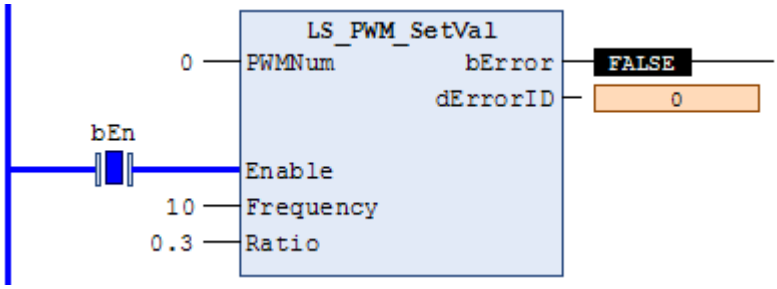


图4.1　PWM输出示意图

例程：

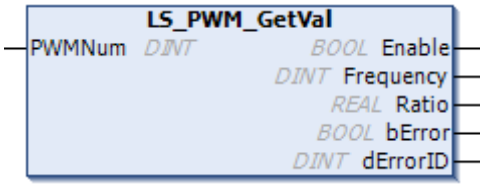
设置PWM0输出频率10HZ，占空比为30%的信号。



获取PWM参数 LS_PWM_GetVal

该功能块用作回读设置的PWM参数信息。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_PWM_GetVal	FUN		LS_PWM_GetVal(PWMNum:= , Enable=> , Frequency=> , Ratio=> , bError=> , dErrorID=>);

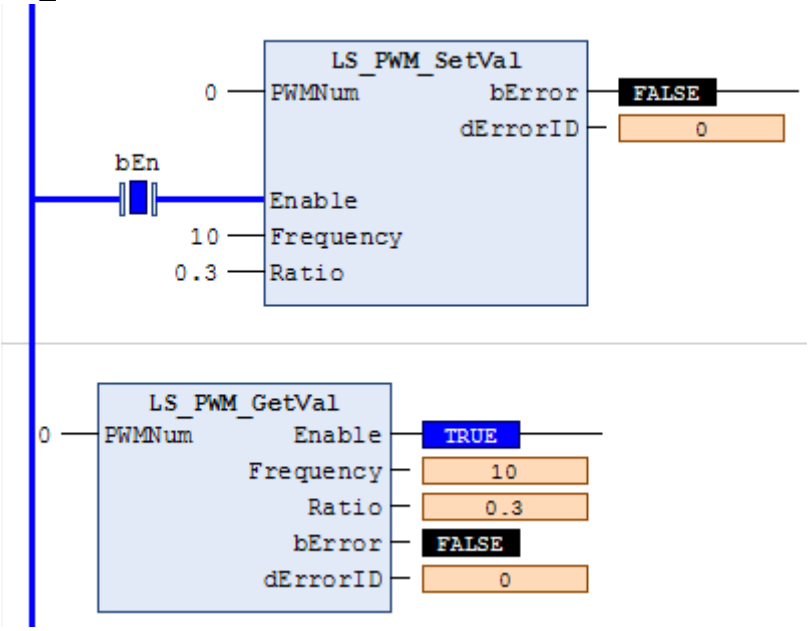
变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	PWMNum	DINT	[0,3]	0	PWM通道：0-3
输出	Enable	BOOL	TRUE/FALSE	False	False-禁止PWM；True-使能PWM，状态需要保持
	Frequency	DINT	[1,500000]	1	PWM的频率1~500000Hz

	Ratio	REAL	[0,1]	0	PWM的占空比0%~100%
	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误；True-输入参数越界
	dErrorID	DINT	遵照数据类型	0	错误码

例程：

通过LS_PWM_SetVal功能块设置PWM0输出频率10HZ，占空比为30%的信号，然后通过功能块LS_PWM_GetVal回读此PWM参数信息。



4.1.2 系统时间指令

控制器系统自带系统时间，掉电后不会丢失。系统时间相关指令在包含在PMC_Controller库中，必须确认库中是否存在PMC_Controller文件，如不存在，需在工程中添加PMC_Controller库，相关指令如表4.1所示。

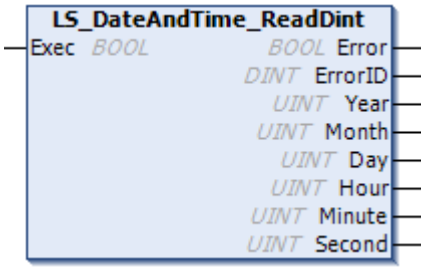
表4.1 系统时间指令

名称	功能
LS_DateAndTime_ReadDint	获取控制器系统时间
LS_DateAndTime_ReadString	获取控制器系统时间
LS_DateAndTime_SetDint	设置控制器系统时间
LS_DateAndTime_ SetString	设置控制器系统时间

读取系统时间LS_DateAndTime_ReadDint

该功能块用作读取系统时间。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_DateAndTime_ReadDint	FB		LS_DateAndTime_ReadDint (Exec:= , Error=> , ErrorID=> , Year=> , Month=> , Day=> , Hour=> , Minute=> , Second=>);

变量：

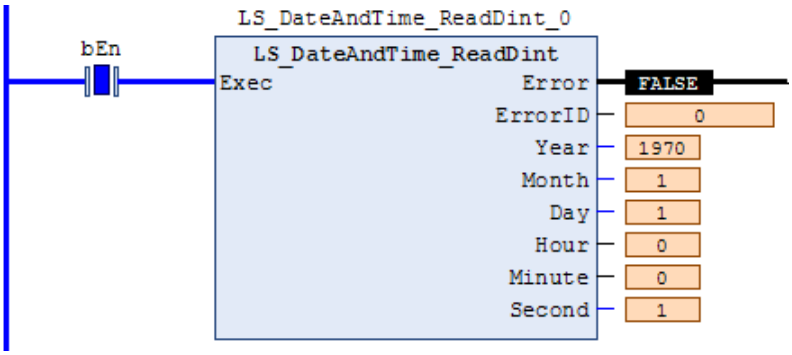
输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	TRUE时使能功能块，需要保持
输出	Error	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误；
	ErrorID	DINT	遵照数据类型	0	错误码
	Year	UINT	遵照数据类型	0	年
	Month	UINT	遵照数据类型	0	月
	Day	UINT	遵照数据类型	0	日
	Hour	UINT	遵照数据类型	0	时
	Minute	UINT	遵照数据类型	0	分
	Second	UINT	遵照数据类型	0	秒

功能说明：

读取控制器系统时间，将年月日时分秒拆分读取。

注意事项：系统时间需在有电池供电情况下断电才不会丢失。使用该功能时只需直接调用该指令即可，注意指令中参数的数据类型。

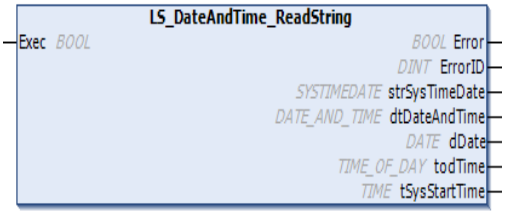
例程：



读取系统时间LS_DateAndTime_ReadString

该功能块用作读取系统时间。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_DateAndTime_ReadString	FB		LS_DateAndTime_ReadString (Exec:= , Error=> , ErrorID=> , strSysTimeDate=> , dtDateAndTime=> , dDate=> , todTime=> , tSysStartTime=>);

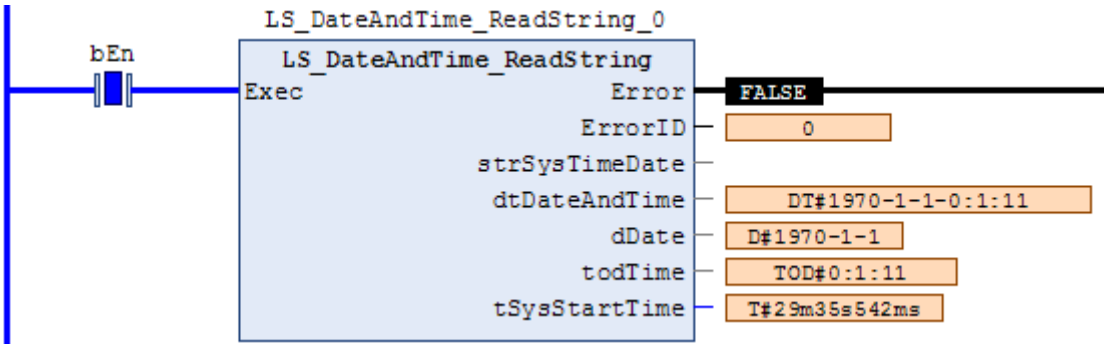
变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	TRUE时使能功能块，需要保持
输出	Error	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误；
	ErrorID	DINT	遵照数据类型	0	错误码
	strSysTimeDate	SYSTIMEDATE	遵照数据类型	-	控制器日期时间
	dtDateAndTime	DATE_AND_TIME	遵照数据类型	DT#1970-1-1-0:0:0	日期和时间
	dDate	DATE	遵照数据类型	D#1970-1-1	日期
	todTime	TOD	遵照数据类型	TOD#0:0:0	时间
	tSysStartTime	TIME	遵照数据类型	T#0ms	开机所需时间

功能说明：读取控制器系统时间，将日期时间直接读取。

注意事项：系统时间需在有电池供电情况下断电才不会丢失。使用该功能时只需直接调用该指令即可，注意指令中参数的数据类型。

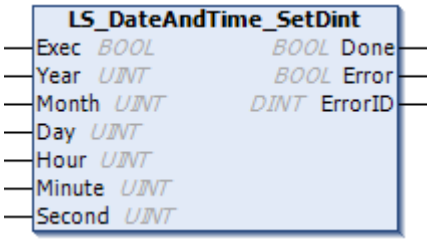
例程：



设置系统时间LS_DateAndTime_SetDint

该功能块用作设置系统时间。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_DateAndTime_SetDint	FB		LS_DateAndTime_SetDint (Exec:= , Year:= , Month:= , Day:= , Hour:= , Minute:= , Second:= Done=> , Error=> , ErrorID=>);

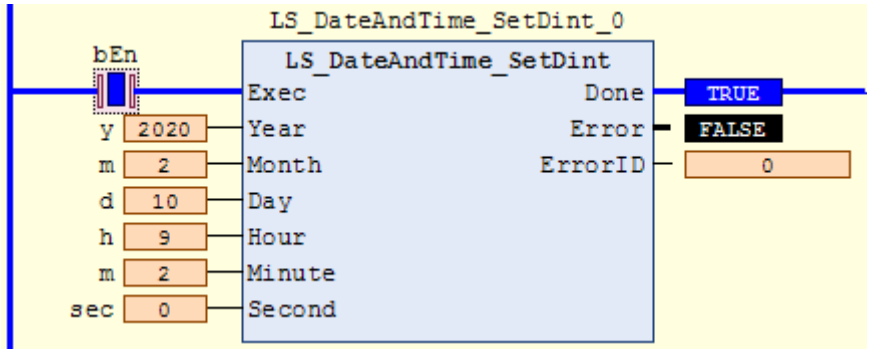
变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	TRUE时使能功能块，上升沿有效
	Year	UINT	遵照数据类型	0	年
	Month	UINT	[1,12]	0	月
	Day	UINT	[1,31]	0	日
	Hour	UINT	[0,23]	0	时
	Minute	UINT	[0,59]	0	分
	Second	UINT	[0,59]	0	秒
输出	Done	BOOL	TRUE/FALSE	FALSE	完成，False-没有完成；
	Error	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误；
	ErrorID	DINT	遵照数据类型	0	错误码

功能说明：按年月日时分秒字段设置系统时间。

注意事项：系统时间需在有电池供电情况下断电才不会丢失。使用该功能时只需直接调用该指令即可，注意指令中参数的数据类型。

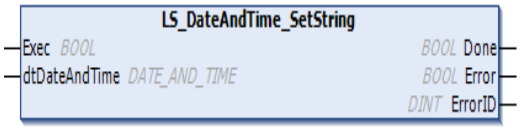
例程：



设置系统时间LS_DateAndTime_SetString

该功能块用作设置系统时间。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_DateAndTime_SetString	FB		LS_DateAndTime_SetString (Exec:= , dtDateAndTime:= , Done=> , Error=> , ErrorID=>);

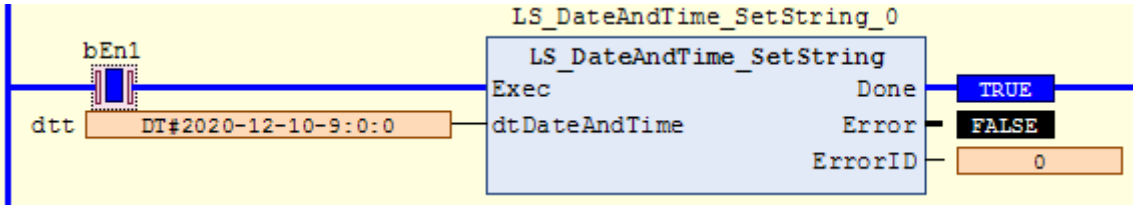
变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	TRUE时使能功能块，需要保持
输出	Error	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误；
	ErrorID	DINT	遵照数据类型	0	错误码
	strSysTimeDate	SYSTIMEDATE	遵照数据类型	0	
	dtDateAndTime	DATE_AND_TIME	遵照数据类型	DT#1970-1-1-0:0:0	日期和时间
	dDate	DATE	遵照数据类型	D#1970-1-1	日期
	todTime	TOD	遵照数据类型	TOD#0:0:0	时间
	tSysStartTime	TIME	遵照数据类型	T#0ms	累计时间

功能说明：按日期时间设置系统时间。

注意事项：系统时间需在有电池供电情况下断电才不会丢失。使用该功能时只需直接调用该指令即可，注意指令中参数的数据类型。

例程：



4.1.3 掉电保持变量指令

控制器系统自带掉电保存功能，它提供了64K的掉电保护存储区。在程序中可将重要数据保存于掉电保持型变量中，数据将不会丢失，断电重启控制器后，可将重要数据读取出来，表4.2为程序常用变量数据类型。

表4.2 常用变量数据类型

类型	类型名称	数据下限	数据上限	数据宽度	备注
BOOL	布尔型	0	1	1bit	
BYTE	字节型	0	255	8bit	
WORD	字型	0	65535	16bit	
DWORD	双字型	0	4294967295	32bit	
SINT	短整型	-128	127	8bit	
USINT	无符号短整型	0	255	8bit	
INT	整型	-32768	32767	16bit	
UINT	无符号整型	0	65535	16bit	
DINT	长整型	-2147483648	2147483647	32bit	
UDINT	无符号长整型	0	4294967295	32bit	
REAL	实数型	1.175494351e-38	3.402823466e+38	32bit	单精度浮点数
LREAL	实数型	2.225073855072014e-308	1.7976931348623158e+308	64bit	双精度浮点数
TIME	时间型			32bit	示例： Time1 : TIME := t#3s;
TOD	时刻型				示例：Tod1：TOD：= TOD#00:00:00
DATE	日期型				示例：Date1：DATE：= D#2008-8-8;
DT	日期时刻型				示例：DT1： DT： = dt#2008-08-08-20:08:08
STRING	字符串型				示例： Str:STRING(35):=‘hi’;
ARRAY	数组				示例： Arr1:ARRAY[1..5]OF BYTE:=1,2,3,4,5;

掉电保持型变量可直接在编程软件中定义，步骤如下：

1) 在工程设备栏中选择“Application”，右击选择“添加对象”-“持续变量”，在弹出的窗口命名这个全局掉电保持型变量列表PersistentVars(可任意命名)；

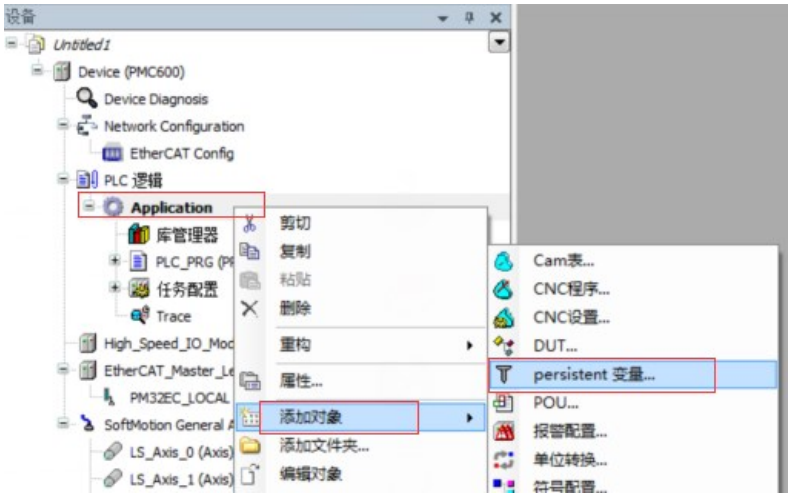


图4.2　添加掉电保持型变量列表

2)　在变量列表中添加你需要定义的全局掉电保持型变量即可(注意其数据类型)。

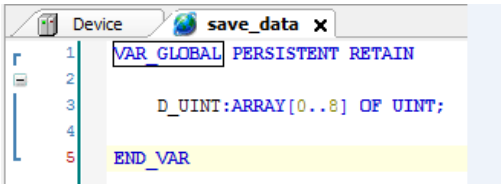


图4.3　定义掉电保持型变量

由图4.3所示，定义掉电保持型变量有其固有的格式，VAR GLOBAL表示全局变量，PERSISTENT RETAIN表示持续保持变量，也就是掉电保持型变量必须是全局变量，另外普通变量可以通过热复位变成设定的初始值或者标准初始值，**但掉电保持型变量只能通过初始值复位(设备中的任何应用、引导工程和剩余变量都将被删除)才能将其复位，这时需重新将程序下载到控制器中执行。**

4.1.4　固件版本指令

PMC系列总线运动控制器支持控制器相关版本信息的读取，包括控制器版本信息、CPU版本信息、FPGA版本信息以及ARM版本信息。控制器进行各种版本信息读取的相关指令包含在PMC_Controller库，程序若用到该功能，必须确认库中是否存在PMC_Controller文件，如不存在，须在工程中添加PMC_Controller库，相关指令如表4.3所示。

表4.3　固件版本指令

名称	功能
LS_PLC_ControllerVer	获取控制器硬件版本信息
LS_PLC_CPUID	获取控制器CPU ID号
LS_PLC_HardwareVerInformation	获取固件FPGA版本信息
LS_PLC_SoftwareVerInformation	获取固件Arm版本信息
LS_PLC_VerInformation	获取控制器版本信息

获取版本信息LS_PLC_ControllerVer

该函数获取控制器的硬件版本信息。

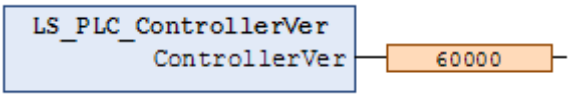
指令外观：

指令	FB/FUN	图形模块	结构文本
LS_PLC_ControllerVer	FC		LS_PLC_ControllerVer(ControllerVer=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输出	ControllerVer	DINT	遵照数据类型	0	控制器硬件版本

例程：



获取CPU的ID号LS_ PLC_CPUID

该函数获取控制器的CPU ID号。

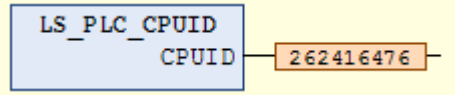
指令外观：

指令	FB/FUN	图形模块	结构文本
LS_PLC_CPUID	FC		LS_PLC_CPUID(ControllerVer=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输出	CPUID	UDINT	遵照数据类型	0	控制器CPU ID号

例程：



获取固件FPGA信息LS_ PLC_ HardwareVerInformation

该函数获取控制器的固件FPGA版本信息。

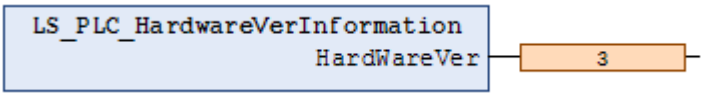
指令外观：

指令	FB/ FUN	图形模块	结构文本
LS_PLC_ HardwareVerInf ormation	FC		LS_PLC_ HardwareVerInformation(HardWareVer=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输出	HardWareVer	DINT	遵照数据类型	0	控制器固件FPGA版本

例程：



获取固件ARM信息LS_ PLC_ SoftwareVerInformation

该函数获取控制器的固件ARM版本信息。

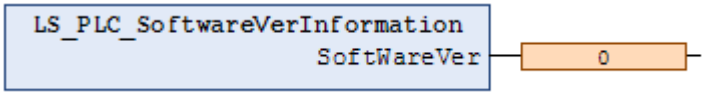
指令外观：

指令	FB/ FUN	图形模块	结构文本
LS_PLC_ SoftwareVerInf ormation	FC		LS_PLC_ SoftwareVerInformation(SoftWareVer=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输出	SoftWareVer	DINT	遵照数据类型	0	控制器固件ARM版本

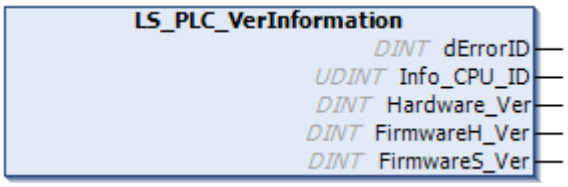
例程：



获取控制器的版本信息LS_ PLC_ VerInformation

该函数获取控制器的所有版本信息。

指令外观：

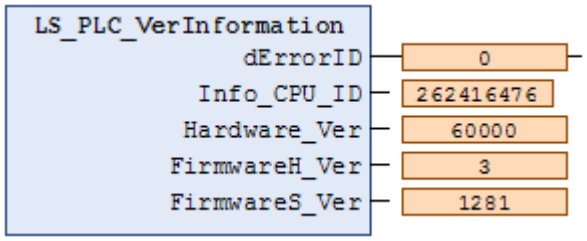
指令	FB/FUN	图形模块	结构文本
LS_PLC_VerInformation	FC		LS_PLC_VerInformation(dErrorID=> , Info_CPU_ID=> , Hardware_Ver=> , FirmwareH_Ver=> , FirmwareS_Ver=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输出	dErrorID	DINT	遵照数据类型	0	错误码
	Info_CPU_ID	UDINT	遵照数据类型	0	控制器CPU ID号
	Hardware_Ver	DINT	遵照数据类型	0	控制器硬件版本
	FirmwareH_Ver	DINT	遵照数据类型	0	控制器固件FPGA版本
	FirmwareS_Ver	DINT	遵照数据类型	0	控制器固件ARM版本

功能说明：用作读取控制器的版本信息，包括ID号、硬件、固件等信息。

例程：



4.2 文件管理指令

PMC600运动控制器支持内部文件操作、U盘文件传输、用户参数配置文件传输，方便用户更好地实现上传下载文件功能。文件操作相关指令在包含在PMC_FileManage库中，程序中若想使用文件操作功能，必须确认库中是否存在PMC_FileManage文件，如不存在，须在工程中添加PMC_FileManage库。

PMC_FileManage库中用户用到的指令包含两部分：

UseData_FileManage：对控制器本地“UsrData”文件夹内的文件进行读写；

UsrConfig_FileManag：对控制器本地“UsrConfig”文件夹内的文件进行读写。

一般情况下建议用户将系统类的参数文件存储在“UsrConfig”文件夹内，工艺类的文件存储在“UsrData”内。

注意：文件存取在UsrData和UsrConfig固定路径下，用户不能修改路径。

用户在读取文件信息时，会使用到File_Information结构体，具体参数如表4.4所示。

表4.4　File_Information结构体参数

名称	类型	初始化	注释
FileName	STRING(32)	‘’	文件名
CreatTime	STRING(32)	‘’	文件创建时间
ModifyTime	STRING(32)	‘’	文件最近修改时间
FileSize	STRING(32)	‘’	文件大小，Byte

注意：

1) 控制器能够识别FAT32文件系统的U盘，对于NTFS文件系统的U盘控制器是无法识别的；

2) 控制器能识别的U盘最大32G。

3) 支持U盘热插拔，但是在读写U盘文件的时候不要插拔U盘，否则可能会造成文件或者U盘损坏。

4) 文件操作比较耗费时间，我们在编写系统程序的时候，需要特别注意：文件操作指令和运动指令不要放在同一任务中。常规用法是：文件操作单独放在一个任务中，该任务的**优先级参数设置为20或更大一个数，任务周期时间大于200ms**（文件大小在1M以内，推荐值为200~400ms）。

4.2.1 UsrData内文件操作指令

UsrData内文件操作指令见表4.5。

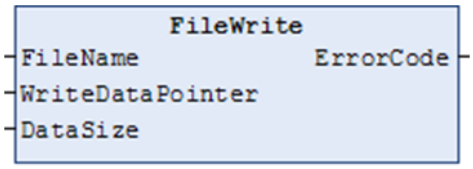
表4.5 UsrData内文件操作指令表

名称	功能
FileWrite	在本地路径“UsrData”内写文件内容
FileWriteAppend	在本地路径“UsrData”内追加方式写文件内容
FileRead	在本地路径“UsrData”内读取文件
FileCopy	在本地路径“UsrData”内拷贝文件
FileRename	在本地路径“UsrData”内对文件进行重命名
FileDelete	在本地路径“UsrData”内删除文件
FileDeleteAll	在本地路径“UsrData”内批量删除文件
GetDirectoryFile	获取本地路径“UsrData”内所有文件名及相关文件信息
GetFileInformation	获取本地路径“UsrData”内特定文件的信息
UDisk_CopyFromUDisk	从U盘拷贝文件至本地“UsrData”文件夹内
UDisk_CopyToUDisk	将本地“UsrData”文件夹内文件拷贝至U盘
UDisk_GetDirectoyFile	获取U盘目录下的文件名及相关文件信息

文件写入 FileWrite

在控制器“UsrData”文件夹内创建新文件，写文件内容。若该文件名存在，该文件内容会被覆盖。

指令外观：

指令	FB/FUN	图形模块	结构文本
FileWrite	FB		FileWrite(FileName:= , WriteDataPointer:= , DataSize:= , ErrorCode=>);

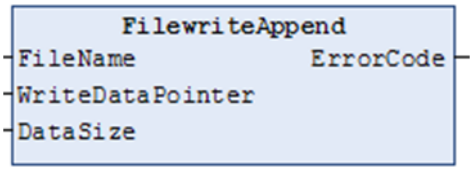
变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	默认的保存文件名
	WriteData Pointer	POINT TO BYTE	遵照数据类型		需要写入文件的数据指针
	DataSize	DWORD	遵照数据类型	0	写入的数据大小
输出	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误，2：创建文件错误

文件追加写入FilewriteAppend

在控制器“UsrData”文件夹内采用追加的方式写文件内容。

指令外观：

指令	FB/FUN	图形模块	结构文本
FileWriteAppend	FB		FileWriteAppend(FileName:= , WriteDataPointer:= , DataSize:= , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	待写入内容的文件名
	WriteData Pointer	Pointer to byte	遵照数据类型		追加写入文件的数据指针
	Datasize	DWORD	遵照数据类型	0	写入数据大小
输出	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误，非0：有错误

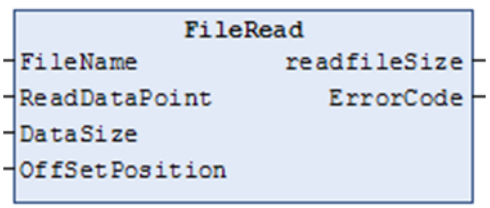
功能说明：

在控制器“UsrData”文件夹内，对FileName指定的文件名以追加的方式写入WriteDataPointer指定的文件内容。

文件读取 FileRead

在控制器“UsrData”文件夹内读取文件。

指令外观：

指令	FB/FUN	图形模块	结构文本
FileRead	FB		FileRead(FileName:= , ReadDataPoint:= , DataSize:= , OffSetPosition:= , readfileSize=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	默认的文件名
	ReadDataPointer	POINT TO BYTE	遵照数据类型		数据接受指针

	DataSize	DWORD	遵照数据类型	0	读取的数据大小
	OffSetPosition	DWORD	遵照数据类型	0	读取的位置偏移
输出	ReadfileSize	DWORD	遵照数据类型	0	读取的文件大小
	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误，2：打开源文件错误，4：文件位置错误

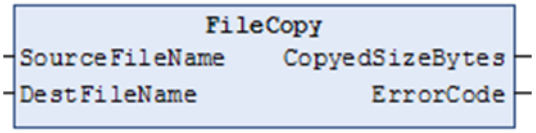
功能说明：

在控制器“UsrData”文件夹内读取FileName所指定的文件名中的文件内容，读取的文件内容存储在ReadDataPointer中。

文件拷贝FileCopy

在控制器“UsrData”文件夹内拷贝文件。

指令外观：

指令	FB/FUN	图形模块	结构文本
FileCopy	FB		FileCopy(SourceFileName:= , DestFileName:= , CopyedSizeBytes=>, ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	SourceFileName	STRING（32）	遵照数据类型	‘tmp.data’	源文件名
	DestFileName	STRING（32）	遵照数据类型	‘tmp2.data’,	目标文件名
输出	CopyedSizeBytes	DWORD	遵照数据类型	0	拷贝的字节数
	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误，2：打开源文件错误，3：创建目标文件错误

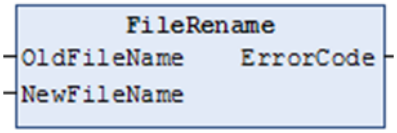
功能说明：

在控制器“UsrData”文件夹内将SourceFileName指定的源文件拷贝到DestFileName所指定的目标文件中，拷贝的字节数有CopyedSizeBytes所指定。

文件重命名 FileRename

在控制器“UsrData”文件夹内对文件进行重命名操作。

指令外观：

指令	FB/FUN	图形模块	结构文本
FileRename	FB		FileRename(OldFileName:= , NewFileName:= , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	OldFileName	STRING（32）	遵照数据类型	‘tmp.data’	原文件名
	NewFileName	STRING（32）	遵照数据类型	‘temp2.data’	新文件名
输出	ErrorCode	UDINT	遵照数据类型	0	0：无错误，1：源文件不存在

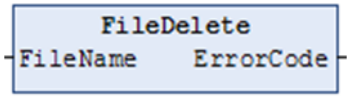
功能说明：

在控制器“UsrData”文件夹内对OldFileName所指定的文件名称进行重命名操作，新文件名由NewFileName指定。

文件删除FileDelete

在控制器“UsrData”文件夹内删除文件操作。

指令外观：

指令	FB/FUN	图形模块	结构文本
FileDelete	FB		FileDelete(FileName:= , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	删除的文件名
输出	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误，1：删除错误，2：文件不存在

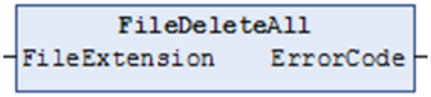
功能说明：

在控制器“UsrData”文件夹内删除FileName所指定名称的文件。

文件批量删除FileDeleteAll

在控制器“UsrData”文件夹内批量删除文件操作。

指令外观：

指令	FB/FUN	图形模块	结构文本
FileDeleteAll	FB		FileDeleteAll(FileExtension:= , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileExtension	STRING（8）	遵照数据类型		文件扩展名，如'.data'
输出	ErrorCode	UDINT	遵照数据类型	0	错误码： 0：无错误，非0：有错误

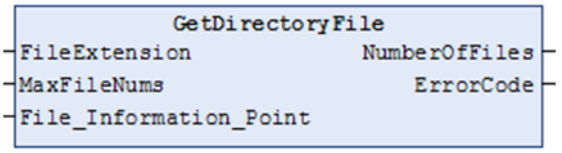
功能说明：

批量删除“UsrData”文件夹内FileExtension所指定扩展名的多个文件，若扩展名为空，删除路径下所有文件。

获取目录内文件信息GetDirectoryFile

从控制器“UsrData”文件夹内获取该目录下的所有文件名及相关文件信息。

指令外观：

指令	FB/FUN	图形模块	结构文本
GetDirectoryFile	FB		GetDirectoryFile(FileExtension:= , MaxFileNums:= , File_Information_Point:= , NumberOfFiles=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileExtension	STRING（8）	遵照数据类型		文件名包含指定的字符串，不输入字符表示获取所有文件名
	MaxFileNums	INT	遵照数据类型	20	需要读取的文件个数
	File_Infomation_Point	Pointer To File_Information	遵照数据类型		文件信息存储的指针
输出	NumberOfFiles	INT	遵照数据类型	0	读取到的文件个数
	ErrorCode	UDINT	遵照数据类型	0	错误码： 0：无错误，非0：有错误

功能说明：

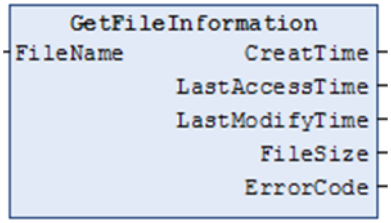
从控制器“UsrData”文件夹内获取该目录下文件名称中包含FileExtension所指定的字

符串的文件信息。

获取指定文件信息GetFileInformation

在控制器“UsrData”文件夹内获取文件信息。

指令外观：

指令	FB/FUN	图形模块	结构文本
GetFileInformation	FB		GetFileInformation(FileName:= , CreatTime=> , LastAccessTime=> , LastModifyTime=> , FileSize=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Filename	STRING（32）	遵照数据类型	‘tmp.data’	文件名
输出	CreatTime	STRING（31）	遵照数据类型	‘’	创建时间
	LastAcessTime	STRING（31）	遵照数据类型	‘’	最近访问时间
	LastModifyTime	STRING（31）	遵照数据类型	‘’	最近修改时间
	FileSize	UDINT	遵照数据类型	0	文件大小（Byte）
	ErrorCode	UDINT	遵照数据类型	0	错误码： 0：无错误，1：源文件不存在

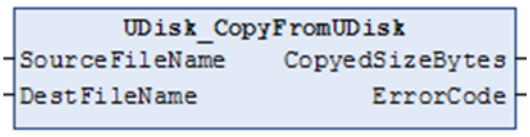
功能说明：

在控制器“UsrData”文件夹内获取Filename所指定的文件名称的文件信息，包括创建时间、最近访问、修改时间等。

从U盘拷贝文件UDisk_CopyFromUDisk

从U盘拷贝文件至控制器“UsrData”文件夹内。

指令外观：

指令	FB/FUN	图形模块	结构文本
UDisk_CopyFromUDisk	FB		UDisk_CopyFromUDisk(SourceFileName:= , DestFileName:= , CopyedSizeBytes=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始化	注释
输入	SourceFileName	STRING（32）	遵照数据类型	'tmp.data'	源文件名
	DestFileName	STRING（32）	遵照数据类型	'tmp2.data'	目标文件名
输出	CopyedSizeBytes	DWORD	遵照数据类型	0	拷贝的字节数
	ErrorCode	UDINT	遵照数据类型	0	错误码： 0表示没有错误，1表示找不到U盘， 2表示创建本地文件错误， 3表示打开U盘文件错误

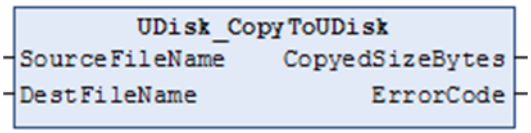
功能说明：

从U盘拷贝文件SourceFileName至控制器“UsrData”文件夹内，文件名称为DestFileName。

拷贝文件至U盘UDisk_CopyToUDisk

从控制器“UsrData”文件夹内将文件拷贝至U盘。

指令外观：

指令	FB/FUN	图形模块	结构文本
UDisk_CopyToUDisk	FB		UDisk_CopyToUDisk(SourceFileName:= , DestFileName:= , CopyedSizeBytes=>, ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始化	注释
输入	SourceFileName	STRING（32）	遵照数据类型	'tmp.data'	源文件名
	DestFileName	STRING（32）	遵照数据类型	'tmp2.data'	目标文件名
输出	CopyedSizeBytes	DWORD	遵照数据类型	0	拷贝的字节数
	ErrorCode	UDINT	遵照数据类型	0	错误码： 0表示没有错误，1表示找不到U盘， 2表示打开本地文件错误， 3表示U盘创建文件错误

功能说明：

用于将控制器“UsrData”文件夹内的SourceFileName文件拷贝至U盘，文件名称为DestFileName。

获取U盘目录内文件UDisk_GetDirectoyFile

获取U盘目录下的文件名及相关文件信息。

指令外观：

指令	FB/FUN	图形模块	结构文本
UDisk_GetDirectoyFile	FB	UDisk_GetDirectoyFile FileExtension NumberOfFiles MaxFileNums ErrorCode File_Information_Point	UDisk_GetDirectoyFile(FileExtension:= , MaxFileNums:= , File_Information_Point:= , NumberOfFiles=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始化	注释
输入	FileExtension	STRING（8）	遵照数据类型		文件扩展名 '.data'
	MaxFileNums	INT	遵照数据类型	20	允许读取最多文件数
	File_Information_Point	POINTER TO File_Information	遵照数据类型		文件信息指针
输出	NumberOfFiles	INT	遵照数据类型	0	读取的文件数
	ErrorCode	UDINT	遵照数据类型	0	错误码： 0表示没有错误， 1表示找不到U盘

功能说明：

用于获取U盘目录下的文件及相关文件信息，U盘目录下文件名称中需包含FileExtension所指定的扩展名信息。

4.2.2 UsrData文件操作例程

程序功能：实现在控制器目录UsrData文件夹内文件写入、文件读取、文件拷贝、文件删除、文件读取、U盘文件操作等功能。

程序编写步骤：

1) 新建工程并命名FileManage_UsrData，编程语言为结构化文本ST，确认库中是否存在PMC_FileManage文件，如不存在，须在工程中添加PMC_FileManage库，然后双击左侧设备栏“库管理器”添加“PMC_FileManage”文件管理库。默认已存在此库文件，用户无需添加。

2) 右击左侧设备栏“Application”，选择“添加对象”-“程序组织单元”，命名为POU_FileManage，选择类型为程序，实现语言为结构化文件，点击打开即可，如图4.4所示。

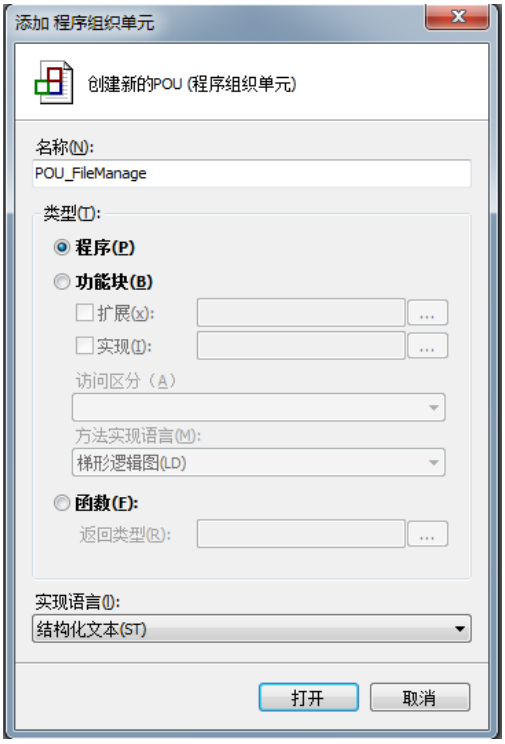


图4.4　添加POU

图4.5　定义全局变量列表

3)　在POU_FileManage中完成各模块程序功能主体的编写，鉴于要在主程序中来控制POU_FileManage中各模块功能，定义一个全局变量列表来实现(右击左侧设备栏“Application”，选择“添加对象”-“全局变量列表”，命名为GVL_File)，如图4.5所示；

4)　在主程序PLC_PRG中控制文件操作使能变量状态来实现文件具体操作，并加入相关IO操作，如图4.6所示；其中定义的几个比较重要的常量和数组如图4.7所示。主程序见图4.8。

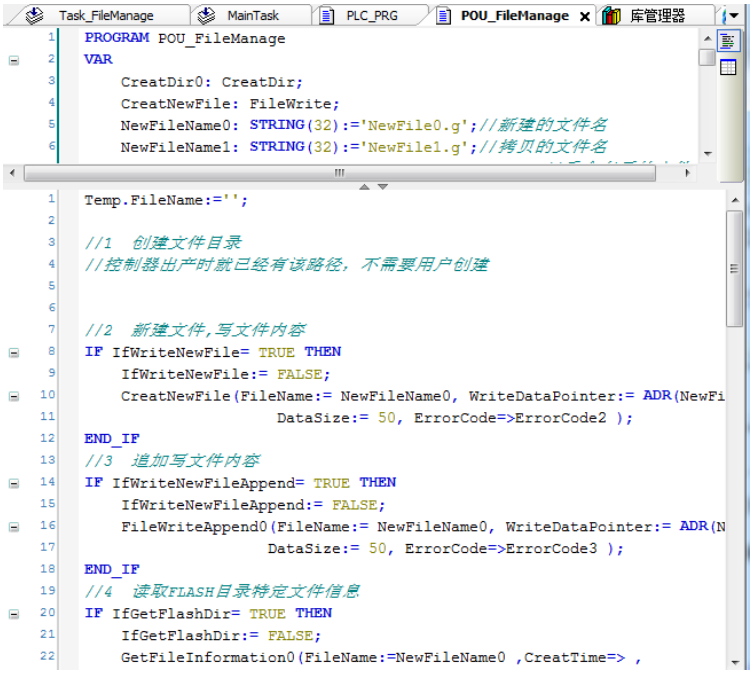


图4.6 文件操作模块主体

```
NewFileName0: STRING(32):='NewFile0.g';//新建的文件名
NewFileName1: STRING(32):='NewFile1.g';//拷贝的文件名
NewFileName2: STRING(32):='NewFile1_Rename.g';//重命名后的文件名
NewFile0: STRING(50):='Welcome to LeadShine !';//NewFileName0的内容
NewFile1: STRING(50):='Append NewFile !';//追加NewFileName0的内容
FileInfo2: ARRAY[0..100] OF File_Information;//定义的文件信息结构体数组
```

图4.7 重要的常量和数组

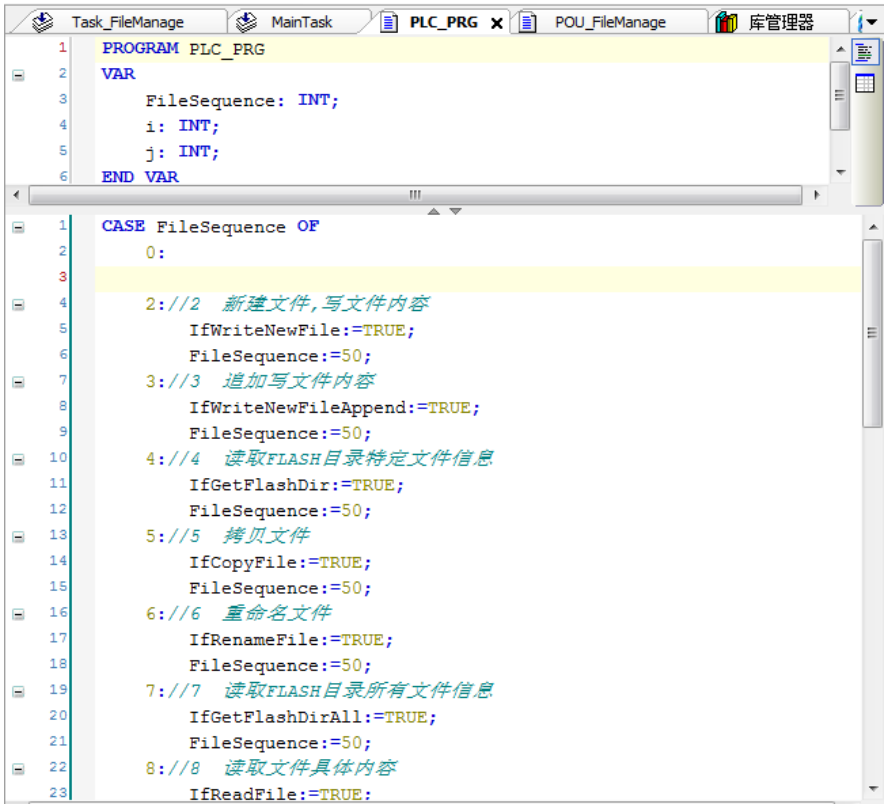


图4.8 文件操作主程序

5) 新增加一个任务，用于处理文件的操作。配置步骤：右击左侧设备栏“任务配置”=> 选择“添加对象” => “任务” => 命名为Task_FileManage => 将POU_FileManage加进扫描区 => 设置任务优先级为20，设置扫描周期为250ms，如图4.9所示。

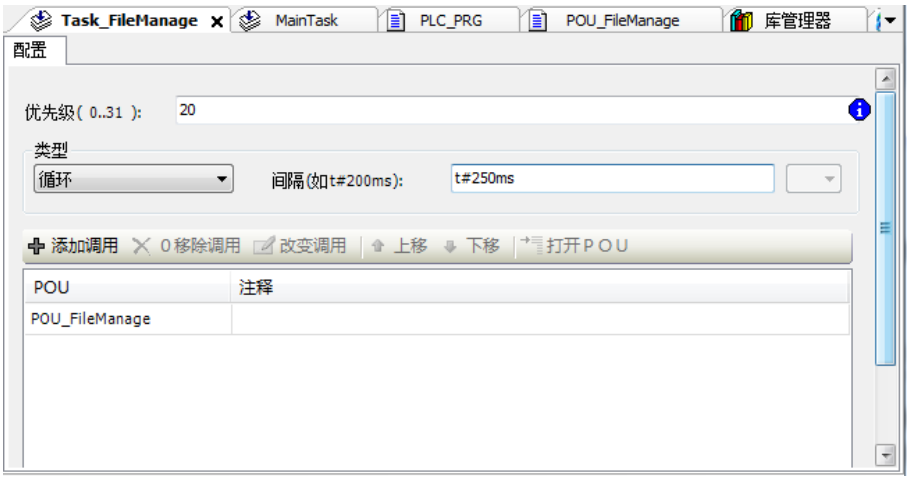


图4.9 添加文件处理任务

6) 完成编程，编译OK后连接控制器，将程序下载到控制器中，点击运行按钮或者按F5，运行程序。然后根据功能需求情况相应地强制使能条件为TRUE来调试。

运行结果：

1) 写文件内容：当新建文件使能被强制为TRUE后，设置的内容被写入文件NewFileName0中，读取该文件可返回该文件创建时间、上次访问时间、上次修改时间、文件大小等参数，如图4.10、4.11所示。

```
//2  新建文件,写文件内容
IF IfWriteNewFile FALSE = TRUE THEN
    IfWriteNewFile FALSE := FALSE;
    CreatNewFile(FileName 'NewFile0.g' := NewFileName0 'NewFile0.g', WriteDataPointer 16#4053B6A7 := ADR(NewFile0 'Welcome to ),
                 DataSize 50 := 50, ErrorCode 0 =>ErrorCode2 0 );
```

图4.10 写文件

```
//4  读取FLASH目录特定文件信息
IF IfGetFlashDir FALSE = TRUE THEN
    IfGetFlashDir FALSE := FALSE;
    GetFileInformation0(FileName 'NewFile0.g' :=NewFileName0 'NewFile0.g' ,CreatTime DT#2015-12-17-14:52:44 =>time0 DT#2015-12-17-14:52:44 ,
    LastAccessTime DT#2015-12-17-14:52:44 =>time1 DT#2015-12-17-14:52:44 , LastModifyTime DT#2015-12-17-14:52:44 =>time2 DT#2015-12-17-14:52:44 ,
        FileSize 50 =>NewFileName0_FileSize 50 ,ErrorCode 0 =>ErrorCode4 0 );
END_IF
```

图4.11 读文件信息

2) 追加写文件内容：当追加写文件使能被强制为TRUE后，设置的内容被追加写入文件NewFileName0中，再次读取该文件可返回该文件创建时间、上次访问时间、上次修改时间、文件大小等参数部分已变动，如图4.11、4.12所示。

```
//3  追加写文件内容
IF IfWriteNewFileAppend FALSE = TRUE THEN
    IfWriteNewFileAppend FALSE := FALSE;
    FileWriteAppend0(FileName 'NewFile0.g' := NewFileName0 'NewFile0.g', WriteDataPointer 16#405616DA := ADR(NewFile1 'Append New '),
                     DataSize 50 := 50, ErrorCode 0 =>ErrorCode3 0 );
```

图4.12　追加写文件

```
//4  读取FLASH目录特定文件信息
IF IfGetFlashDir FALSE = TRUE THEN
    IfGetFlashDir FALSE := FALSE;
    GetFileInformation0(FileName 'NewFile0.g' :=NewFileName0 'NewFile0.g' ,CreatTime DT#2015-12-17-14:59:58 =>time0 DT#2015-12-17-14:59:58 ,
    LastAccessTime DT#2015-12-17-14:52:44 =>time1 DT#2015-12-17-14:52:44 ,  LastModifyTime DT#2015-12-17-14:59:58 =>time2 DT#2015-12-17-14:59:58 ,
        FileSize 100 =>NewFileName0_FileSize 100 ,ErrorCode 0 =>ErrorCode4 0 );
```

图4.13　读文件信息

3)　拷贝文件：当拷贝文件使能被强制为TRUE后，文件NewFileName0内容被拷贝到NewFileName1中，读取FLASH文件数已变为2，如图4.14、4.15所示。

```
//5  拷贝文件
IF IfCopyFile FALSE = TRUE THEN
    IfCopyFile FALSE := FALSE;
    FileCopy0(SourceFileName 'NewFile0.g' :=NewFileName0 'NewFile0.g' , DestFileName 'NewFile1.g' :=NewFileName1 'NewFile1.g' ,
            CopyedSizeBytes 100 =>NewFileName0_FileSize 100 , ErrorCode 0 =>ErrorCode5 0 );
```

图4.14　拷贝文件

```
//7  读取FLASH目录所有文件信息
IF IfGetFlashDirAll FALSE = TRUE THEN
    IfGetFlashDirAll FALSE := FALSE;
    GetDirectoryFile0(FileExtension '.g' :=FileExtension0 '.g' ,MaxFileNums 10 :=10 ,File_Information_Point 16#4054F460 :=ADR(FileInfo2) ,
                NumberOfFiles 2 =>Number_AllFiles 2 ,ErrorCode 0 =>ErrorCode7 0 );
```

图4.15　读文件个数

4) 重命名文件：当重命名文件使能被强制为TRUE后，文件NewFileName1名称由'NewFile1.g'被改成到'NewFile2.g'中，读取FLASH文件信息'NewFile1.g'文件已经被重命名，'NewFile1.g'文件已不复存在，如图4.16、4.17。

```
//6  重命名文件
IF IfRenameFile FALSE = TRUE THEN
    IfRenameFile FALSE := FALSE;
    FileRename0(OldFileName 'NewFile1.g' :=NewFileName1 'NewFile1.g' , NewFileName 'NewFile1_R' :=NewFileName2 'NewFile1_R' ,
                                ErrorCode 0 =>ErrorCode6 0 );
```

图4.16　重命名文件

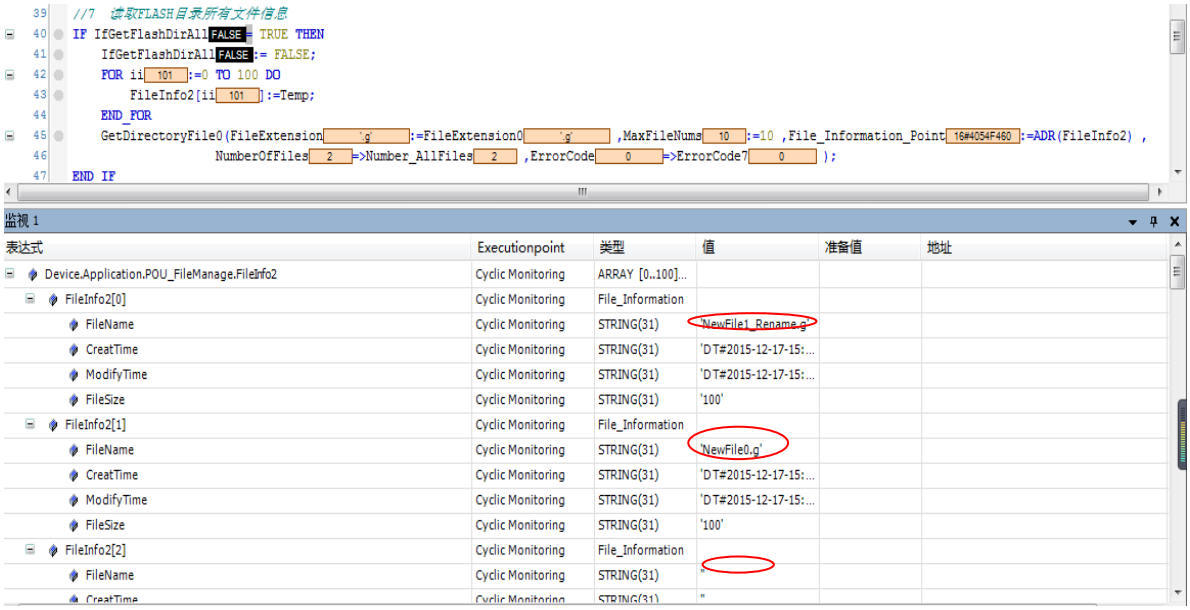


图4.17　读文件信息

5) 向U盘拷贝文件：当向U盘拷贝文件使能被强制为TRUE后，文件NewFileName2名被拷贝到U盘中，并读取U盘文件信息，如图4.18、4.19所示。

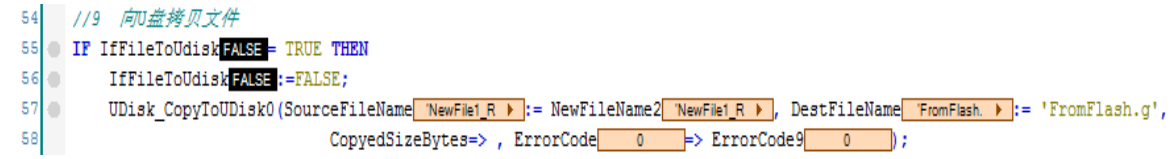


图4.18　向U盘拷贝文件

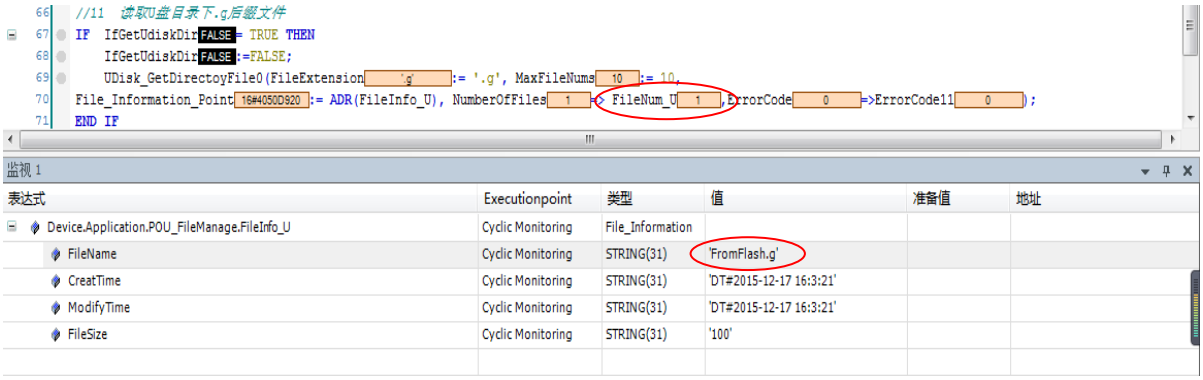


图4.19　读U盘文件信息

由图4.19可知：拷贝到U盘的文件名和读取U盘文件信息中的文件名是一致的；打开U盘的文件可以看到结果2中追加写入文件的内容已成功写入，如图4.20所示。

6) 删除单个文件和批量删除文件：当删除文件使能被强制为TRUE后，文件NewFileName0被被删除；当批量删除文件使能被强制为TRUE后，文件NewFileName2也被删除，如图4.21~4.23所示。

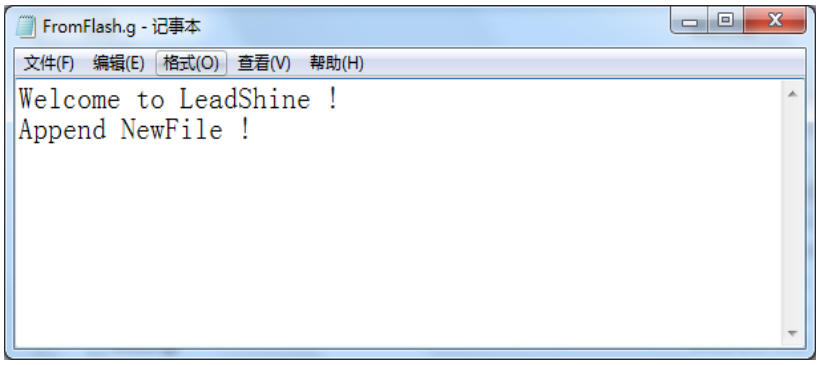


图4.20　写入成功

```
//7  读取FLASH目录所有文件信息
IF IfGetFlashDirAll = TRUE THEN
    IfGetFlashDirAll := FALSE;
    FOR ii:=0 TO 100 DO
        FileInfo2[ii]:=Temp;
    END_FOR
    GetDirectoryFile0(FileExtension:='g' :=FileExtension0 'g' ,MaxFileNums 10 :=10 ,File_Information_Point 16#4054F460 :=ADR(FileInfo2) ,
                      NumberOfFiles 2 =>Number_AllFiles 2 ,ErrorCode 0 =>ErrorCode7 0 );
```

图4.21　删除前文件数

```
//12  删除单个文件
IF IfDelateFile = TRUE THEN
    IfDelateFile := FALSE;
    FileDelete0(FileName 'NewFile0.g' :=NewFileName0 'NewFile0.g' , ErrorCode 0 =>ErrorCode12 0 );
END_IF
```

图4.22　删除单个文件

```
//7  读取FLASH目录所有文件信息
IF IfGetFlashDirAll = TRUE THEN
    IfGetFlashDirAll := FALSE;
    FOR ii:=0 TO 100 DO
        FileInfo2[ii]:=Temp;
    END_FOR
    GetDirectoryFile0(FileExtension:='g' :=FileExtension0 'g' ,MaxFileNums 10 :=10 ,File_Information_Point 16#4054F3E0 :=ADR(FileInfo2) ,
                      NumberOfFiles 1 =>Number_AllFiles 1 ,ErrorCode 0 =>ErrorCode7 0 );
```

图4.23　删除后文件数

由于前面删除了NewFileName0文件，只剩NewFileName2文件，现在再新建一个文件来说明批量删除文件的作用。

如图4.24所示，新建文件；如图4.25所示，批量删除前的文件数；如图4.26所示，批量删除文件；批量删除后文件数，如图4.27所示。

```
//2  新建文件,写文件内容
IF IfWriteNewFile = TRUE THEN
    IfWriteNewFile := FALSE;
    CreatNewFile(FileName 'NewFile0.g' := NewFileName0 'NewFile0.g' , WriteDataPointer 16#4050D6A7 := ADR(NewFile0 'Welcome to ' ),
                 DataSize 50 := 50, ErrorCode 0 =>ErrorCode2 0 );
END_IF
```

图4.24　新建文件

```
//7  读取FLASH目录所有文件信息
IF IfGetFlashDirAll FALSE = TRUE THEN
    IfGetFlashDirAll FALSE := FALSE;
    FOR ii 101 :=0 TO 100 DO
        FileInfo2[ii 101 ]:=Temp;
    END_FOR
    GetDirectoryFile0(FileExtension  'g'  :=FileExtension0  'g'  ,MaxFileNums 10 :=10 ,File_Information_Point 16#4054F460 :=ADR(FileInfo2) ,
                      NumberOfFiles 2 =>Number_AllFiles 2 ,ErrorCode 0 =>ErrorCode7 0 );
END_IF
```

图4.25　批量删除前文件数

```
//13  批量删除文件
IF IfDelateAllFile FALSE = TRUE THEN
    IfDelateAllFile FALSE := FALSE;
    FileDeleteAll0(FileExtension:= , ErrorCode 0 =>ErrorCode13 0 );
END_IF
```

图4.26　批量删除文件

```
//7  读取FLASH目录所有文件信息
IF IfGetFlashDirAll FALSE = TRUE THEN
    IfGetFlashDirAll FALSE := FALSE;
    FOR ii 101 :=0 TO 100 DO
        FileInfo2[ii 101 ]:=Temp;
    END_FOR
    GetDirectoryFile0(FileExtension  'g'  :=FileExtension0  'g'  ,MaxFileNums 10 :=10 ,File_Information_Point 16#4054F380 :=ADR(FileInfo2) ,
                      NumberOfFiles 0 =>Number_AllFiles 0 ,ErrorCode 0 =>ErrorCode7 0 );
```

图4.27　批量删除后文件数

由图4.27可知，在进行批量删除后，FLASH目录下“UsrData”文件夹内的文件已经全部被删除。

注意：在此例程中，读取控制器FLASH系统目录下UsrData文件夹内的文件时，由于读到的是多个文件，定义的文件信息指针必须是指向文件信息结构体数组的，而不是一个文件信息结构体。

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**文件操作功能-FileManage_UsrData**”。

4.2.3　UsrConfig内文件操作指令

“UsrConfig”文件夹内文件操作和“UsrData”内文件操作是一样的，只是换了路径而已。两个文件夹存储区给用户存储不一样的文件，防止误删除了所有文件，不影响另一个文件夹的文件。

一般情况下建议用户将系统类的参数文件存储在“UsrConfig”文件夹内，工艺类的文件存储在“UsrData”内。

UsrConfig内文件操作指令如表4.6所示。

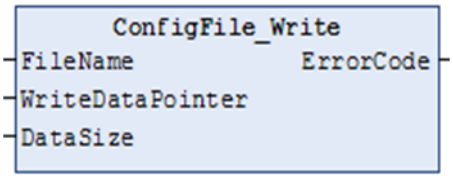
表4.6　UsrConfig内文件操作指令列表

名称	功能
ConfigFile_Write	在本地路径“UsrConfig”内写文件内容
ConfigFile_WriteAppend	在本地路径“UsrConfig”内追加方式写文件内容
ConfigFile_Read	在本地路径“UsrConfig”内读取文件
ConfigFile_Del	在本地路径“UsrConfig”内删除文件
ConfigFile_GetDirectoryFile	获取本地路径“UsrConfig”内所有文件名及相关文件信息
ConfigFile_CopyFromUDisk	从U盘拷贝文件至本地“UsrConfig”文件夹内
ConfigFile_CopyToUDisk	将本地“UsrConfig”文件夹内文件拷贝至U盘

文件写入ConfigFile_Write

在控制器 “UsrConfig”文件夹内创建新文件，写文件内容。若该文件名存在，则该文件内容会被覆盖。

指令外观：

指令	FB/FUN	图形模块	结构文本
ConfigFile_Write	FB		ConfigFile_Write(FileName:= , WriteDataPointer:= , DataSize:= , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	默认的保存文件名
	WriteDataPointer	POINT TO BYTE	遵照数据类型		需要写入文件数据的指针
	DataSize	DWORD	遵照数据类型	0	写入的数据大小
输出	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误， 3：创建文件错误

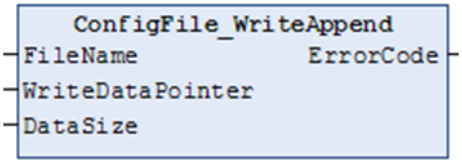
功能说明：

在控制器“UsrConfig”文件夹内创建新文件，写入文件内容。若该文件名存在，则该文件内容会被覆盖。新文件名称由FileName指定，写入的内容由WriteDataPointer指定。

文件追加写入ConfigFile_WriteAppend

在控制器“UsrConfig”文件夹内采用追加的方式写入文件内容。

指令外观：

指令	FB/FUN	图形模块	结构文本
ConfigFile_WriteAppend	FB		ConfigFile_WriteAppend (FileName:= , WriteDataPointer:= , DataSize:= , ErrorCode=>);

变量 ：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	追加写入文件名
	WriteDataPointer	Pointer to byte	遵照数据类型		追加写入文件的数据指针
	Datasize	DWORD	遵照数据类型	0	写入数据大小
输出	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误， 3：创建文件错误

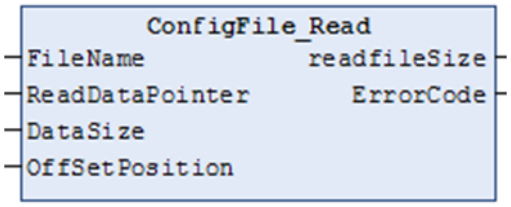
功能说明：

在控制器“UsrConfig”文件夹内对FileName指定的文件名以追加的方式写入WriteDataPointer指定的文件内容。

文件读取ConfigFile_Read

在控制器“UsrConfig”文件夹内读取文件内容。

指令外观 ：

指令	FB/FUN	图形模块	结构文本
ConfigFile_Read	FB		ConfigFile_Read(FileName:= , ReadDataPointer:=, DataSize:= , OffSetPosition:= , readfileSize=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	读取的文件名
	ReadDataPointer	POINT TO BYTE	遵照数据类型		读取的数据指针
	DataSize	DWORD	遵照数据类型	0	需要读取的数据大小
	OffSetPosition	DWORD	遵照数据类型	0	读取的位置偏移
输出	ReadfileSize	DWORD	遵照数据类型	0	实际读取的数据大小
	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误，3：打开文件有错误，4：文件位置错误

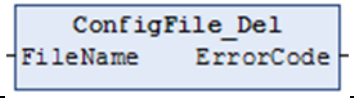
功能说明：

在控制器“UsrConfig”文件夹内读取FileName的文件内容，读取的文件内容存储在ReadDataPointer中。

文件删除ConfigFile_Del

在控制器“UsrConfig”文件夹内删除文件。

指令外观：

指令	FB/FUN	图形模块	结构文本
ConfigFile_Del	FB		ConfigFile_Del(FileName:= , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	FileName	STRING（32）	遵照数据类型	‘tmp.data’	删除的文件名
输出	ErrorCode	UDINT	遵照数据类型	0	错误码，0：无错误，非0：有错误

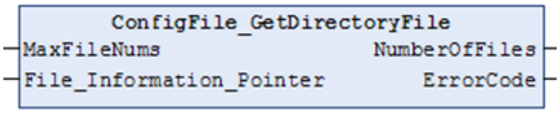
功能说明：

在控制器“UsrConfig”文件夹内删除FileName所指定名称的文件。

获取目录下文件信息ConfigFile_GetDirectoryFile

从控制器“UsrConfig”文件夹内获取该目录下的所有文件名及相关文件信息。

指令外观：

指令	FB/FUN	图形模块	结构文本
ConfigFile_GetDirectoryFile	FB		ConfigFile_GetDirectoryFile(MaxFileNums:= , File_Information_Pointer:= , NumberOfFiles=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	MaxFileNums	INT	遵照数据类型	20	需要读取的文件个数
	File_Infomation_Point	Pointer To File_Information	遵照数据类型		文件信息的指针
输出	NumberOfFiles	INT	遵照数据类型	0	读取到的文件个数
	ErrorCode	UDINT	遵照数据类型	0	0：无错误，非0：有错误

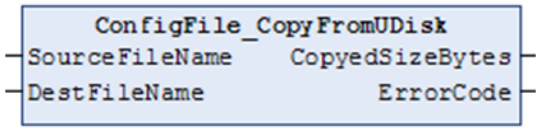
功能说明：

从控制器“UsrConfig”文件夹内获取该目录下的所有文件名及相关文件信息，如创建时间、修改时间、文件大小等。

从U盘拷贝文件ConfigFile_CopyFromUDisk

从U盘拷贝文件至控制器“UsrConfig”文件夹内。

指令外观：

指令	FB/FUN	图形模块	结构文本
ConfigFile_CopyFromUDisk	FB		ConfigFile_CopyFromUDisk(SourceFileName:= , DestFileName:= , CopyedSizeBytes=> , ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始化	注释
输入	SourceFileName	STRING（32）	遵照数据类型	'tmp.data'	源文件名
	DestFileName	STRING（32）	遵照数据类型	'tmp2.data'	目标文件名
输出	CopyedSizeBytes	DWORD	遵照数据类型	0	拷贝的字节数
	ErrorCode	UDINT	遵照数据类型	0	错误码： 0表示没有错误，1表示找不到U盘， 2表示创建本地文件错误， 3表示打开U盘文件错误

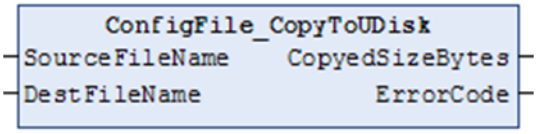
功能说明：

从U盘拷贝SourceFileName文件至控制器“UsrConfig”文件夹内，文件名称为DestFileName。

将文件拷贝至U盘ConfigFile_CopyToUDisk

从控制器“UsrConfig”文件夹内将文件拷贝至U盘。

指令外观：

指令	FB/FUN	图形模块	结构文本
ConfigFile_CopyToUDisk	FB		ConfigFile_CopyToUDisk(SourceFileName:= , DestFileName:= , CopyedSizeBytes=>, ErrorCode=>);

变量：

输入输出	名称	类型	有效范围	初始化	注释
输入	SourceFileName	STRING（32）	遵照数据类型	'tmp.data'	源文件名
	DestFileName	STRING（32）	遵照数据类型	'tmp2.data'	目标文件名
输出	CopyedSizeBytes	DWORD	遵照数据类型	0	拷贝的字节数
	ErrorCode	UDINT	遵照数据类型	0	错误码： 0表示没有错误，1表示找不到U盘， 2表示打开本地文件错误， 3表示U盘创建文件错误

功能说明：

从控制器“UsrConfig”文件夹内将SourceFileName文件拷贝至U盘，U盘中文件名称为DestFileName。

4.2.4 UsrConfig文件操作例程

程序功能：实现在控制器“UsrConfig”下写入文件、读取文件、删除文件、读取文件数及文件信息、U盘文件操作等功能。

程序编写步骤：

1) 新建工程并命名FileManage_UsrConfig，编程语言为结构化文本ST，然后双击左侧设备栏“库管理器”，添加“FileManage”文件管理库。

2) 右击左侧设备栏“Application”，选择“添加对象”-“程序组织单元”，命名为POU_FileManage，选择类型为程序，实现语言为结构化文本。

3) 在POU_FileManage中完成各模块程序功能主体的编写，并定义一个全局变量列表来控制POU_FileManage中各模块功能的启动，如图4.28和4.29所示；

```
VAR_GLOBAL
    IfWriteNewFile: BOOL;//新建配置文件使能
    IfWriteNewFileAppend: BOOL;//追加写配置文件使能
    IfReadFile:BOOL;//读取配置文件使能
    IfGetFlashDirAll: BOOL;//读FLASH所有配置文件使能
    IfDelateFile:BOOL;//删除配置文件使能
    IfFileToUdisk: BOOL;//向U盘拷贝配置文件使能
    IfFileFromUdisk: BOOL;//从U盘拷贝配置文件使能
END_VAR
```

图4.28　定义全局变量列表

```
PROGRAM POU_FileManage
VAR
    NewFileName0: STRING(32):='NewConfigFile0.g';//新建的配置文件名
    ConfigFile:ARRAY[0..10] OF BYTE:=[0,10,13,1,8,15,6,50,32,20,65];//NewFileName0的内容
    ConfigFileAppend:ARRAY[0..2] OF BYTE:=[1,2,3];//追加NewFileName0的内容
    FileInfo1: ARRAY[0..100] OF BYTE;   //读取到的配置文件数据
    ErrorCode1: UDINT;
    ConfigFile_Write0: ConfigFile_Write;
    ErrorCode2: UDINT;
    ReadfileSize: DWORD;
```

```
Temp.FileName:='';
//1  新建配置文件,写文件内容
IF IfWriteNewFile= TRUE THEN
    IfWriteNewFile:= FALSE;
    ConfigFile_Write0(FileName:=NewFileName0 , WriteDataPointer:=ADR(ConfigFile) ,
                DataSize:=11 , ErrorCode=>ErrorCode1 );
END_IF
//2  追加写配置文件内容
IF IfWriteNewFileAppend= TRUE THEN
    IfWriteNewFileAppend:= FALSE;
    ConfigFile_WriteAppend0(FileName:=NewFileName0 , WriteDataPointer:=ADR(ConfigFileAppend) ,
                            DataSize:=3 , ErrorCode=>ErrorCode2 );
END_IF
//3  读取配置文件具体信息
IF IfReadFile= TRUE THEN
    IfReadFile:= FALSE;
    //FOR ii:=0 TO 100 DO
        //FileInfo1[ii]:=0;
    //END_FOR
    ConfigFile_Read0(FileName:=NewFileName0 ,ReadDataPointer:=ADR(FileInfo1) ,DataSize:=100 ,
        OffSetPosition:= 11,readfileSize=>ReadfileSize ,ErrorCode=>ErrorCode3 );
END_IF
//4  读取UsrConfig目录下所有文件信息
IF IfGetFlashDirAll= TRUE THEN
```

图4.29　UsrConfig文件操作模块主体

其中定义的几个比较重要的常量和数组如图4.30所示：

```
PROGRAM POU_FileManage
VAR
    NewFileName0: STRING(32):='NewConfigFile0.g';//新建的配置文件名
    ConfigFile:ARRAY[0..100] OF BYTE:=[0,10,13,1,8,15,6,50,32,20,65];//NewFileName0的内容
    ConfigFileAppend:ARRAY[0..2] OF BYTE:=[1,2,3];//追加NewFileName0的内容
    FileInfo1: ARRAY[0..100] OF BYTE;   //读取到的配置文件数据
```

图4.30　重要的常量和数组

4) 在主程序PLC_PRG中控制文件操作使能变量状态来实现文件具体操作，并加入相关IO操作，如图4.31所示。

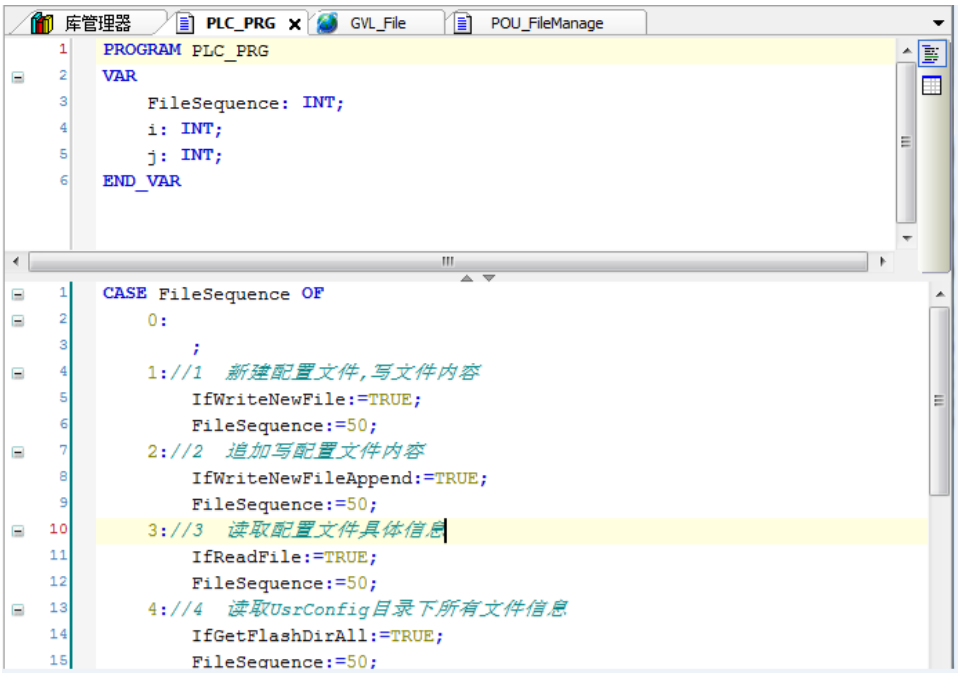


图4.31 UsrConfig文件操作主程序

5) 新增加一个任务，用于处理文件的操作。配置步骤：右击左侧设备栏“任务配置”=> 选择“添加对象” => “任务” => 命名为Task_FileManage => 将POU_FileManage加进扫描区 => 设置任务优先级为20，设置扫描周期为250ms，如图4.32所示。

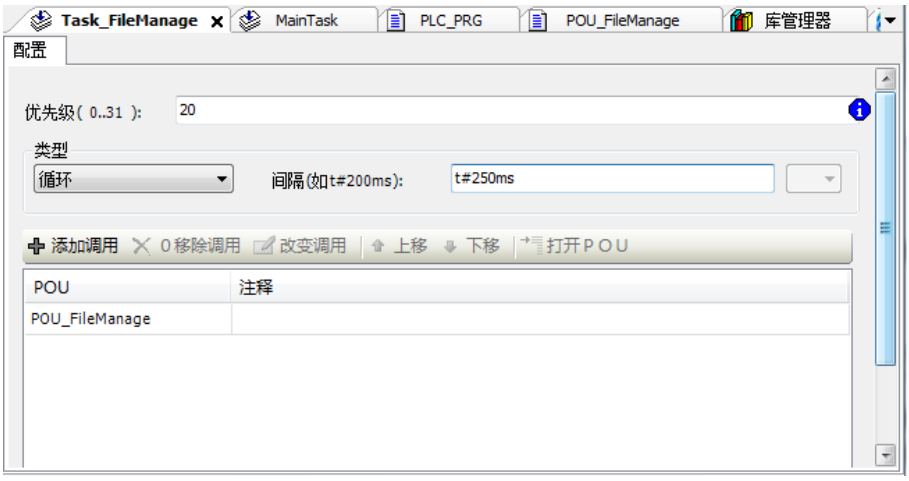


图4.32 添加文件处理任务

6) 完成编程，编译OK后连接控制器，将程序下载到控制器中，点击运行按钮或者按F5，运行程序。然后根据功能需求情况相应地强制使能条件为TRUE来调试。

运行结果：

1) 写文件内容：当新建文件使能被强制为TRUE后，设置的内容被写入文件NewFileName0中，读取该文件可返回配置文件具体信息参数，如图4.33、4.34所示。

```
//1  新建配置文件,写文件内容
IF IfWriteNewFile FALSE = TRUE THEN
    IfWriteNewFile FALSE := FALSE;
    ConfigFile_Write0(FileName 'NewConfigF' :=NewFileName0 'NewConfigF' , WriteDataPointer 16#405964CB :=ADR(ConfigFile) ,
                DataSize 11 :=11 , ErrorCode 0 =>ErrorCode1 0 );
```

图4.33　写文件

```
//3  读取配置文件具体信息
IF IfReadFile FALSE = TRUE THEN
    IfReadFile FALSE := FALSE;
    ConfigFile_Read0(FileName 'NewConfigF' :=NewFileName0 'NewConfigF' ,ReadDataPointer 16#4050D4CB :=ADR(FileInfo1) ,DataSize 100 :=100 ,
        OffSetPosition:= ,readfileSize 100 =>ReadfileSize 100 ,ErrorCode 0 =>ErrorCode3 0 );
END_IF
//4  读取UsrConfig目录下所有文件信息
```

监视 1

表达式	Executionpoint	类型	值	准备值	地址
Device.Application.POU_FileManage.FileInfo1	Cyclic Monitoring	ARRAY [0..100]...			
FileInfo1[0]	Cyclic Monitoring	BYTE	0		
FileInfo1[1]	Cyclic Monitoring	BYTE	10		
FileInfo1[2]	Cyclic Monitoring	BYTE	13		
FileInfo1[3]	Cyclic Monitoring	BYTE	1		
FileInfo1[4]	Cyclic Monitoring	BYTE	8		
FileInfo1[5]	Cyclic Monitoring	BYTE	15		
FileInfo1[6]	Cyclic Monitoring	BYTE	6		
FileInfo1[7]	Cyclic Monitoring	BYTE	50		
FileInfo1[8]	Cyclic Monitoring	BYTE	32		
FileInfo1[9]	Cyclic Monitoring	BYTE	20		
FileInfo1[10]	Cyclic Monitoring	BYTE	65		

图4.34　读文件内容

由图4.34可知，程序读取到的结果和写入的文件内容是一致的。需要注意的是，写入配置文件的内容必须是字节类型的数据，也就是写文件内容指令中参数WriteDataPointer必须是指向字节类型的变量或者数组。

2) 追加写文件内容：当追加写文件使能被强制为TRUE后，追加的内容被写入文件NewFileName0中，再次读取该文件可返回其数据已变动，如图4.35、4.36所示。

```
//2  追加写配置文件内容
IF IfWriteNewFileAppend FALSE = TRUE THEN
    IfWriteNewFileAppend FALSE := FALSE;
    ConfigFile_WriteAppend0(FileName 'NewConfigF' :=NewFileName0 'NewConfigF' , WriteDataPointer 16#40596530 :=ADR(ConfigFileAppend) ,
                DataSize 3 :=3 , ErrorCode 0 =>ErrorCode2 0 );
END_IF
```

图4.35　追加写文件

```
//3  读取配置文件具体信息
IF IfReadFile FALSE = TRUE THEN
    IfReadFile FALSE := FALSE;
    FOR ii 101 :=0 TO 100 DO
        FileInfo1[ii 101 ]??? :=0;
    END_FOR
    ConfigFile_Read0(FileName 'NewConfigF' :=NewFileName0 'NewConfigF' ,ReadDataPointer 16#405B03A0 :=ADR(FileInfo1) ,DataSize 100 :=100 ,
        OffSetPosition:= ,readfileSize 14 =>ReadfileSize 14 ,ErrorCode 0 =>ErrorCode3 0 );
END_IF
```

监视 1

表达式	Executionpoint	类型	值	准备值	地址
Device.Application.POU_FileManage.FileInfo1	Cyclic Monitoring	ARRAY [0..100]...			
FileInfo1[0]	Cyclic Monitoring	BYTE	0		
FileInfo1[1]	Cyclic Monitoring	BYTE	10		
FileInfo1[2]	Cyclic Monitoring	BYTE	13		
FileInfo1[3]	Cyclic Monitoring	BYTE	1		
FileInfo1[4]	Cyclic Monitoring	BYTE	8		
FileInfo1[5]	Cyclic Monitoring	BYTE	15		
FileInfo1[6]	Cyclic Monitoring	BYTE	6		
FileInfo1[7]	Cyclic Monitoring	BYTE	50		
FileInfo1[8]	Cyclic Monitoring	BYTE	32		
FileInfo1[9]	Cyclic Monitoring	BYTE	20		
FileInfo1[10]	Cyclic Monitoring	BYTE	65		
FileInfo1[11]	Cyclic Monitoring	BYTE	1		
FileInfo1[12]	Cyclic Monitoring	BYTE	2		
FileInfo1[13]	Cyclic Monitoring	BYTE	3		
FileInfo1[14]	Cyclic Monitoring	BYTE	0		
FileInfo1[15]	Cyclic Monitoring	BYTE	0		

图4.36　读文件内容

由4.35图可知，程序在追加数据到指定文件中后，读取该文件得到的数据如图4.36所示，配置文件有14个数据元素，数据已成功追加到该配置文件中。

需要注意的是，读取配置文件信息指令有一个关键参数OffSetPosition(读取位置偏移量)，若不设置，则默认为0不偏移，读取该文件全部数据；在程序中第一次新建配置文件写入11个数据元素，后追加3个数据元素，共14个数据元素，若设置偏移量为10，则读取到的结果为新建配置文件的最后一个数据元素和后三个追加的数据元素，若设置偏移量为11，则读取到的结果为后三个追加的数据元素，偏移量设置数不可大于数据元素总数量，否则读到的数据会是空的。

设置偏移量为11读文件内容，如图4.37所示：

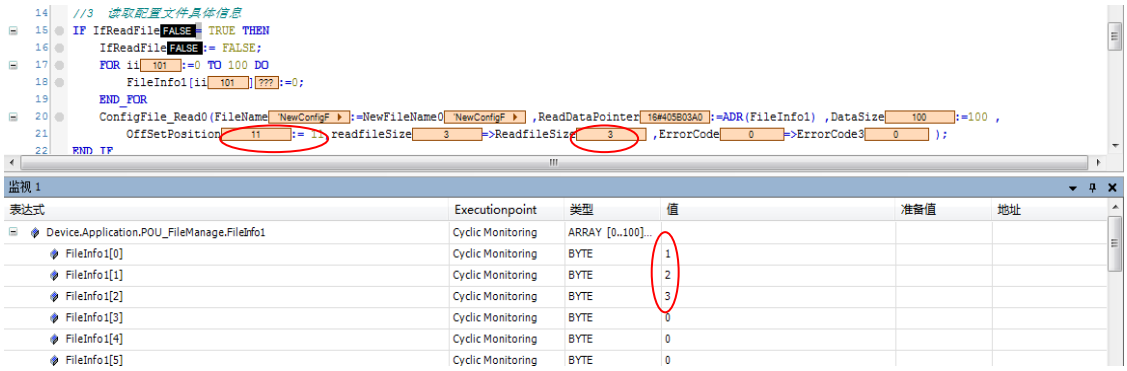


图4.37　读文件内容

由图4.37可知，程序在设置读取偏移量为11时，读取得到的数据只有后追加的数据，如上图所示，实际读到的数据个数只有3个。

注意：在读取文件时，由于读到的数据是多个，因此定义读取数据指针必须是指向BYTE类型的数组，而不是一个BYTE型变量。

3) 读目录内文件信息：当读目录内文件信息使能被强制为TRUE后，该指令执行，返回读取到的文件数、文件名、文件创建时间、文件修改时间以及文件实际大小，如图4.38所示。

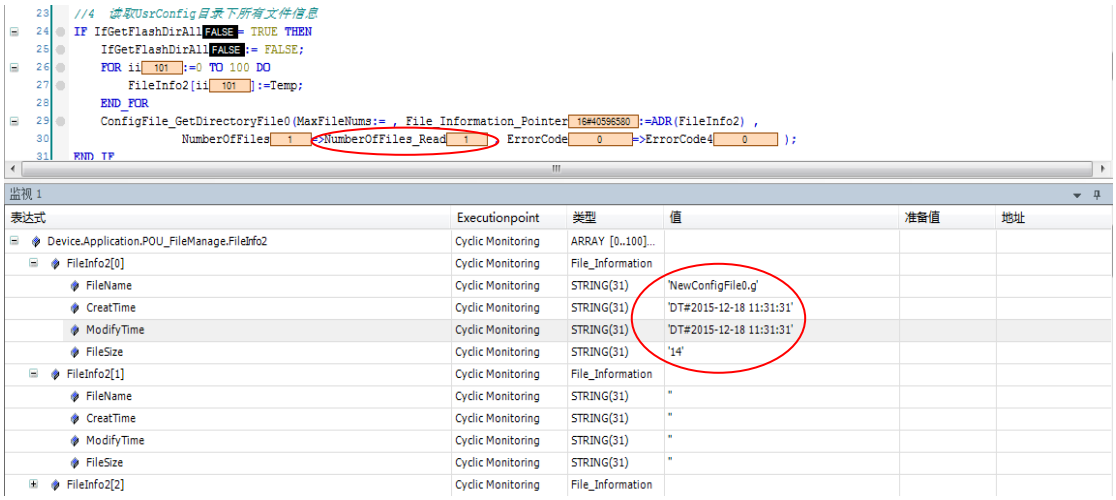


图4.38　读目录内文件信息

4) 向U盘拷贝文件：当向U盘拷贝文件使能被强制为TRUE后，指令执行拷贝文件到U盘的

功能，如图4.39所示。

```
//5  向U盘拷贝文件
IF IfFileToUdisk FALSE = TRUE THEN
    IfFileToUdisk FALSE :=FALSE;
    ConfigFile_CopyToUDisk0(SourceFileName 'NewConfigF' :=NewFileName0 'NewConfigF' , DestFileName 'ConfigFile' :='ConfigFileFromFlash.g' ,
            CopyedSizeBytes 14 =>CopyedSizeBytes_ToUpan 14 , ErrorCode 0 =>ErrorCode5 0 );
```

图4.39　向U盘拷贝文件

由图4.39可知，将“UsrConfig”文件夹内文件拷贝至U盘时，CopyedSizeBytes记录实际拷贝的数据大小。

5) 从U盘拷贝文件到控制器：当从U盘拷贝文件使能被强制为TRUE后，指令执行从U盘拷贝文件到控制器，如图4.40、4.41所示。

```
//6  从U盘向Flash拷贝文件
IF IfFileFromUdisk FALSE = TRUE THEN
    IfFileFromUdisk FALSE :=FALSE;
    ConfigFile_CopyFromUDisk0(SourceFileName 'ConfigFile' :='ConfigFileFromFlash.g' , DestFileName 'ConfigFile' :='ConfigFileFromUpan.g' ,
            CopyedSizeBytes 14 =>CopyedSizeBytes_FromUpan 14 , ErrorCode 0 =>ErrorCode6 0 );
```

图4.40　向U盘拷贝文件

```
//4  读取UsrConfig目录下所有文件信息
IF IfGetFlashDirAll FALSE = TRUE THEN
    IfGetFlashDirAll FALSE := FALSE;
    FOR ii 101 :=0 TO 100 DO
        FileInfo2[ii 101 ]:=Temp;
    END_FOR
    ConfigFile_GetDirectoryFile0(MaxFileNums:= , File_Information_Pointer 16#40596580 :=ADR(FileInfo2) ,
            NumberOfFiles 2 =>NumberOfFiles_Read 2 , ErrorCode 0 =>ErrorCode4 0 );
END_IF
```

监视 1

表达式	Executionpoint	类型	值	准备值	地址
Device.Application.POU_FileManage.FileInfo2	Cyclic Monitoring	ARRAY [0..100]...			
FileInfo2[0]	Cyclic Monitoring	File_Information			
FileName	Cyclic Monitoring	STRING(31)	'ConfigFileFromUpan.g'		
CreatTime	Cyclic Monitoring	STRING(31)	'DT#2015-12-18 13:53:51'		
ModifyTime	Cyclic Monitoring	STRING(31)	'DT#2015-12-18 13:53:51'		
FileSize	Cyclic Monitoring	STRING(31)	'14'		
FileInfo2[1]	Cyclic Monitoring	File_Information			
FileName	Cyclic Monitoring	STRING(31)	'NewConfigFile0.g'		
CreatTime	Cyclic Monitoring	STRING(31)	'DT#2015-12-18 11:31:31'		
ModifyTime	Cyclic Monitoring	STRING(31)	'DT#2015-12-18 11:31:31'		
FileSize	Cyclic Monitoring	STRING(31)	'14'		
FileInfo2[2]	Cyclic Monitoring	File_Information			
FileName	Cyclic Monitoring	STRING(31)	''		

图4.41　读目录内所有文件

由图4.41可知，执行将U盘文件拷贝至控制器后，再次读取控制器“UsrConfig”下文件信息时，读取到两个文件：之前新建的文件“NewConfigFile0.g”'和从U盘拷贝到控制器的文件“ConfigFileFromUpan.g”。

6) 删除文件：当删除文件使能被强制为TRUE后，指令执行删除文件的功能，如图4.42、4.43、4.44所示。

```
//4  读取UsrConfig目录下所有文件信息
IF IfGetFlashDirAll FALSE = TRUE THEN
    IfGetFlashDirAll FALSE := FALSE;
    FOR ii 101 :=0 TO 100 DO
        FileInfo2[ii 101 ]:=Temp;
    END_FOR
    ConfigFile_GetDirectoryFile0(MaxFileNums:= , File_Information_Pointer 16#40596580 :=ADR(FileInfo2) ,
            NumberOfFiles 2 =>NumberOfFiles_Read 2 , ErrorCode 0 =>ErrorCode4 0 );
```

图4.42　删除前文件数

```
//7  删除单个配置文件
IF IfDelateFile FALSE = TRUE THEN
    IfDelateFile FALSE :=FALSE;
    ConfigFile_Del0(FileName 'NewConfigF' :=NewFileName0 'NewConfigF' , ErrorCode 0 =>ErrorCode7 0 );
END_IF RETURN
```

图4.43　删除文件

```
//4  读取UsrConfig目录下所有文件信息
IF IfGetFlashDirAll FALSE = TRUE THEN
    IfGetFlashDirAll FALSE := FALSE;
    FOR ii 101 :=0 TO 100 DO
        FileInfo2[ii 101 ]:=Temp;
    END_FOR
    ConfigFile_GetDirectoryFile0(MaxFileNums:= , File_Information_Pointer 16#40596580 :=ADR(FileInfo2) ,
                NumberOfFiles 1 =>NumberOfFiles_Read 1 , ErrorCode 0 =>ErrorCode4 0 );
END_IF
```

监视 1

表达式	Executionpoint	类型	值
Device.Application.POU_FileManage.FileInfo2	Cyclic Monitoring	ARRAY [0..100]...	
FileInfo2[0]	Cyclic Monitoring	File_Information	
FileName	Cyclic Monitoring	STRING(31)	'ConfigFileFromUpan.g'
CreatTime	Cyclic Monitoring	STRING(31)	'DT#2015-12-18 13:53:51'
ModifyTime	Cyclic Monitoring	STRING(31)	'DT#2015-12-18 13:53:51'
FileSize	Cyclic Monitoring	STRING(31)	'14'
FileInfo2[1]	Cyclic Monitoring	File_Information	
FileName	Cyclic Monitoring	STRING(31)	''

图4.44　删除后文件数

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**文件操作功能-FileManage_UsrConfig**”。

4.3 通讯指令

4.3.1 HMI库相关指令

1. 变量定义及Modbus地址

大多数人机界面（HMI）的数据地址从1开始，而雷赛PMC运动控制器内存地址则从0开始。因此，当使用此类人机界面，在填入数据寄存器地址时，需要在控制器内存地址的基础上加1。例如：某数据的控制器内存地址为4x_100，则其对应的人机界面地址（HMI）为4x_101。

类似的人机界面还有Eview、MCGS、Weinview等触摸屏，以及组态王、三维力控等组态软件。**在下面的所有说明中，默认为人机界面的数据起始地址是从1开始**；在库中对每个变量的注释，表示的是该变量人机界面（HMI）对应的Modbus地址，如图4.45所示。

注意，并不是所有人机界面的数据地址都是从1开始的，设计时请严格参照其说明书或相关手册。

用户只需关注定义的变量与Modbus地址的对应关系，而不用关心变量在控制器内部的地址（%M）。

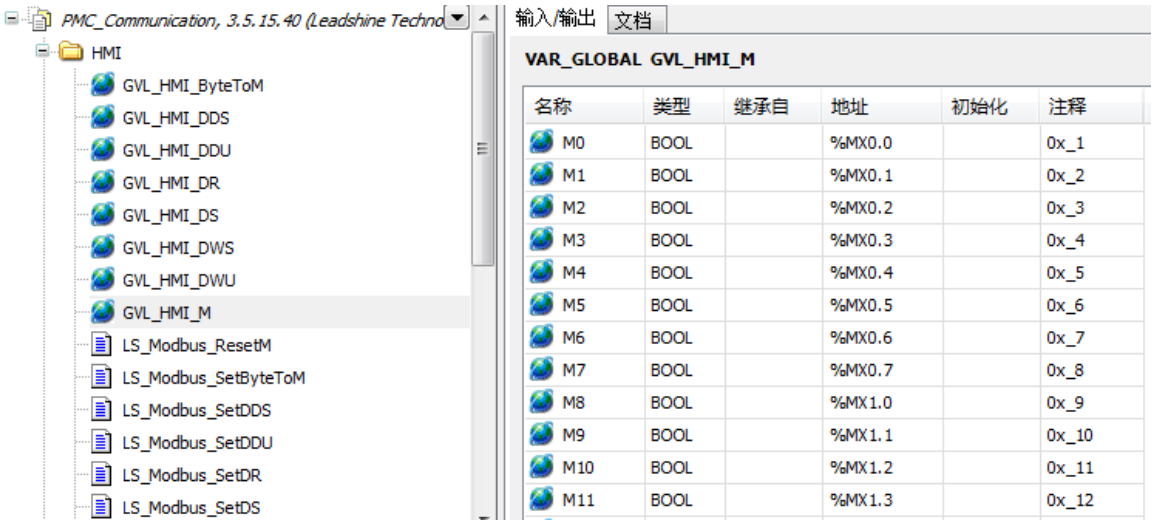


图4.45 Modbus地址对应

在PMC_Communication库文件的HMI文件中，定义了如表4.7所示的数据类型及数量。

表4.7　HMI文件中的数据类型及数量

数据类型	数据长度	类型名称	数量
BOOL	1位	M	4000
BYTE	8位	ByteToM	500
INT	16位	DWS	5000
UINT	16位	DWU	5000
DINT	32位	DDS	5000
UDINT	32位	DDU	5000
REAL	32位	DR	2500
String	31个字符	DS	150

备注：DWS-Data Word Signed；DWU-Data Word Unsigned；DDS-Data Double Signed；DDU-Data Double Unsigned；DR-Data Real；DS-Data String；ByteToM-Byte。

（1）BOOL变量的Modbus地址

BOOL变量M的Modbus地址映射公式为：**M i = 0x_ j**

i的范围0~3999；j的范围1~4000

例如：M0的Modbus地址为0x_1，M200的Modbus地址为0x_201。表4.8列出了变量M对应的Modbus地址。

表4.8 变量M对应的Modbus地址

M	人机界面（HMI）Modbus地址	控制器内存地址
M0	0x_1	%MX0.0
M1	0x_2	%MX0.1
M2	0x_3	%MX0.2
M3	0x_4	%MX0.3
…	…	…
M3997	0x_3998	%MX499.5
M3998	0x_3999	%MX499.6
M3999	0x_4000	%MX499.7

（2）ByteToM变量的Modbus地址

ByteToM变量的Modbus地址映射公式为：**ByteToM[i].k = 0x_ j**

i的范围0~499；k的范围0~7；j的范围4001+8*i+k。

这种变量按位访问的写法：ByteToM[i].k 。

例如：ByteToM[0].0的Modbus地址为0x_4001，ByteToM[10].5 的Modbus地址为4x_(4001+8*10+5)=0x_4086。

表4.9列出了变量ByteToM对应的Modbus地址。

表4.9　变量ByteToM对应的Modbus地址

ByteToM	人机界面（HMI）Modbus地址	控制器内存地址
ByteToM[0].0	0x _4001	%MB500.0
ByteToM[0].1	0x _4002	%MB500.1
ByteToM[0].2	0x _4003	%MB500.2
ByteToM[0].3	0x _4004	%MB500.3
…	…	…
ByteToM[499].5	0x _7998	%MB999.5
ByteToM[499].6	0x _7999	%MB999.6
ByteToM[499].7	0x _8000	%MB999.7

（3）INT变量的Modbus地址

INT变量DWS的Modbus地址映射公式为：**DWS [i] = 4x_ j**

i的范围0~4999；j的对应范围1+i。

例如：DWS[0]的Modbus地址为4x_1，DWS[200]的Modbus地址为4x_201。

表4.10列出了变量DWS对应的Modbus地址。

表4.10　变量DWS对应的Modbus地址

DWS	人机界面（HMI）Modbus地址	控制器内存地址
DWS[0]	4x _1	%MB1000
DWS[1]	4x _2	%MB1002
DWS[2]	4x _3	%MB1004
DWS[3]	4x _4	%MB1006
…	…	…
DWS[4997]	4x _4998	%MB10994
DWS[4998]	4x _4999	%MB10996
DWS[4999]	4x _5000	%MB10998

（4）UINT变量的Modbus地址

UINT变量DWU的Modbus地址映射公式为：**DWU [i]= 4x_ j**

i的范围0~4999；j=5001+i；

例如：DWU[0]的Modbus地址为4x_5001，DWU[200]的Modbus地址为4x_5201。

表4.11列出了变量DWU对应的Modbus地址。

表4.11　变量DWU对应的Modbus地址

DWS	人机界面（HMI）Modbus地址	控制器内存地址
DWU[0]	4x _5001	%MB11000
DWU[1]	4x _5002	%MB11002
DWU[2]	4x _5003	%MB11004
DWU[3]	4x _5004	%MB11006
…	…	…

DWU[4997]	4x _9998	%MB20994
DWU[4998]	4x _9999	%MB20996
DWSU[4999]	4x _10000	%MB20998

（5）DINT变量的Modbus地址

DINT变量DDS的Modbus地址映射公式为：**DDS[i]= 4x_ j**

i的范围0~4999；j的范围10001+2*i；

例如：DDS[0]的Modbus地址为4x_10001，DDS[200]的Modbus地址为4x_10401。

表4.12列出了变量DDS对应的Modbus地址。

表4.12　变量DDS对应的Modbus地址

DDS	人机界面（HMI）Modbus地址	控制器内存地址
DDS[0]	4x _10001	%MB21000
DDS[1]	4x _10003	%MB21004
DDS[2]	4x _10005	%MB21008
DDS[3]	4x _10007	%MB21016
…	…	…
DDS[4997]	4x _19995	%MB40988
DDS[4998]	4x _19997	%MB40992
DDS[4999]	4x _19999	%MB40996

（6）UDINT变量的Modbus地址

UDINT变量DDU的Modbus地址映射公式为：**DDU[i]= 4x_ j**

i的范围0~4999；j的范围20001+2*i。

例如：DDU[0]的Modbus地址为4x_20001，DDU[200]的Modbus地址为4x_20401。

表4.13列出了变量DDU对应的Modbus地址。

表4.13　变量DDU对应的Modbus地址

DDU	人机界面（HMI）Modbus地址	控制器内存地址
DDU[0]	4x _20001	%MB41000
DDU[1]	4x _20003	%MB41004
DDU[2]	4x _20005	%MB41008
DDU[3]	4x _20007	%MB41016
…	…	…
DDU[4997]	4x _29995	%MB60988
DDU[4998]	4x _29997	%MB40992
DDU[4999]	4x _29999	%MB40996

（7）REAL变量的Modbus地址

REAL变量DR的Modbus地址映射公式为：**DR[i]= 4x_ j**

i的范围0~2499；j的范围30001+2*i。

例如：DR[0]的Modbus地址为4x_30001，DR[200]的Modbus地址为4x_30401。

表4.14列出了变量DR对应的Modbus地址。

表4.14　变量DR对应的Modbus地址

DR	人机界面（HMI）Modbus地址	控制器内存地址
DR[0]	4x _30001	%MB61000
DR[1]	4x _30003	%MB61004
DR[2]	4x _30005	%MB61008
DR[3]	4x _30007	%MB61016
…	…	…
DR[2497]	4x _34995	%MB70988
DR[2498]	4x _34997	%MB70992
DR[2499]	4x _34999	%MB70996

（8）STRING变量的Modbus地址

STRING变量DS的Modbus地址映射公式为：**DS[i]= 4x_ j**

i的范围0~149；j的范围35001+16*i。

例如：DS[0]的Modbus地址为4x_35001，DS[20] 的Modbus地址为4x_35321。

表4.15列出了变量DS对应的Modbus地址。

表4.15　变量DS对应的Modbus地址

DS	人机界面（HMI）Modbus地址	控制器内存地址
DS[0]	4x _35001	%MB71000
DS[1]	4x _35017	%MB71032
DS[2]	4x _35033	%MB71064
DS[3]	4x _35049	%MB71096
…	…	…
DS[147]	4x _37353	%MB75704
DS[148]	4x _37369	%MB75736
DS[149]	4x _37385	%MB75768

2. 批量变量操作功能块

为了方便用户使用雷赛定义的HMI变量，在PMC_Communication库文件的HMI文件中定义了变量的批量赋值操作功能块，程序中对于单个变量操作可以直接赋值，对多个变量进行操作就可以用功能块来实现，如：

将M5变量赋值为TRUE：M5:=true;

将ByteToM[4]的第3位置为True：ByteToM[4].3:=True;

将DDS[23]变量赋值为123：DDS[23]:=123;

将DR[35]变量赋值为-9.34：DR[35]:=-9.34;

将DDU[12]的值赋给i：i:= DDU[12];

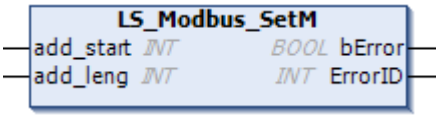
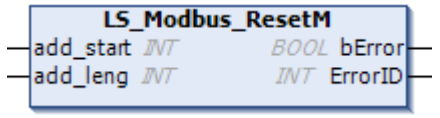
M变量的操作指令

M（BOOL型）变量操作提供了两个功能块，相关指令如表4.16所示。

表4.16　M变量操作指令

名称	功能
LS_Modbus_Set_M	用于为M变量置位，置位后为1(TRUE)
LS_Modbus_ResetM	用于为M变量复位，复位后变量值为0(FALSE)

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetM	FB		LS_Modbus_SetM (add_start:= , Add_leng:= , bError=> , ErrorID=>);
LS_Modbus_ResetM	FB		LS_Modbus_ResetM (add_start:= , Add_leng:= , bError=> , ErrorID=>);

变量：

指令LS_Modbus_SetM和LS_Modbus_ResetM

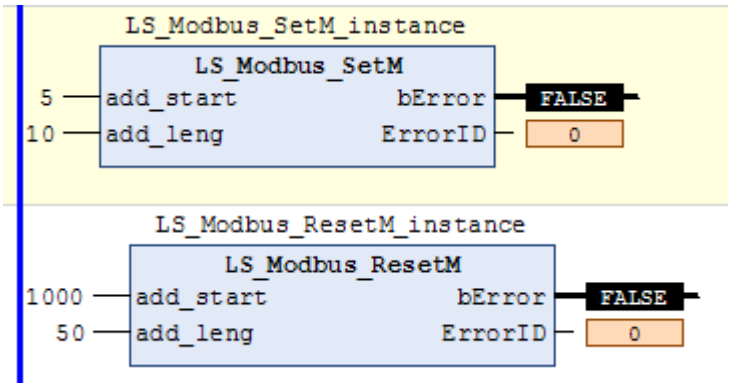
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,3999]	0	变量起始地址
	add_leng	INT	[1,4000]	0	变量个数
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

批量置位或复位BOOL型变量M0~M3999的值，对应Modbus地址0x_1~0x_4000。

例程：

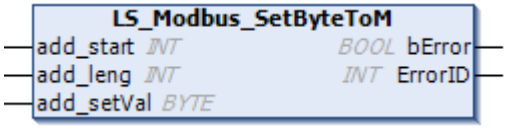
将M5至M14的值设置为true，M1000至M1049的值设置为FALSE。程序如下：



LS_Modbus_SetByteToM指令

该功能块用于设置ByteToM(BYTE型)变量的值。

指令外观 ：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetByteToM	FB		LS_Modbus_SetByteToM (add_start:= , add_leng:= , add_setVal:= , bError=> , ErrorID=>);

变量：

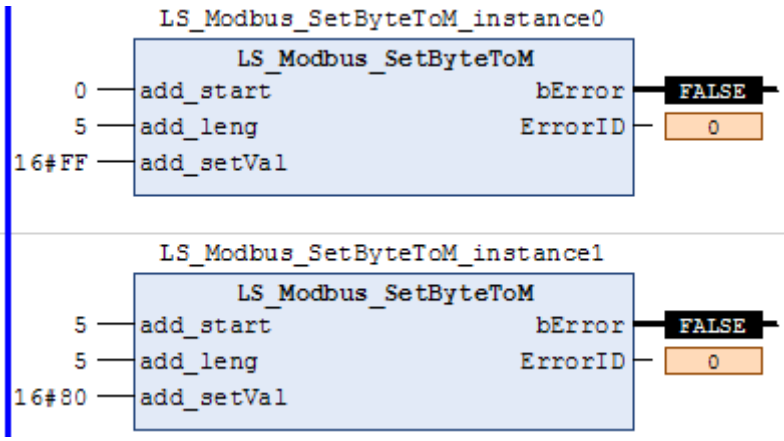
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,499]	0	变量起始地址
	add_leng	INT	[1,500]	0	变量个数
	add_setVal	BYTE	遵照数据类型	0	设置的值
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

按位设置字节型变量ByteToM[0]~ ByteToM[499]的值，对应Modbus地址0x_4001~0x_8000。

例程：

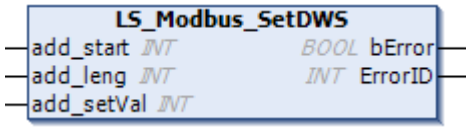
将ByteToM[0]至ByteToM[4]的每位设置为TRUE，ByteToM[5]至ByteToM[9]每个字节的最高位设置为TRUE。程序如下：



LS_Modbus_SetDWS指令

该功能块用于设置DWS(INT型)变量的值。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetDWS	FB		LS_Modbus_SetDWS (add_start:= , add_leng:= , add_setVal:= , bError=> , ErrorID=>);

变量 ：

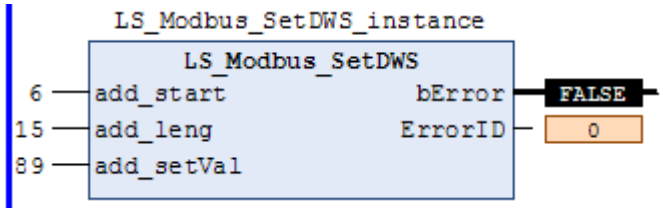
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,4999]	0	变量起始地址
	add_leng	INT	[1,5000]	0	变量个数
	add_setVal	INT	遵照数据类型	0	设置的值
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

设置INT型变量DWS [0]~ DWS[4999]的值，对应Modbus地址4x _1~4x_5000。

例程：

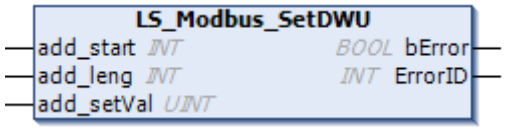
将DWS[6]至DWS[15]的值设置为89。程序如下：



LS_ Modbus_SetDWU指令

该功能块用于设置DWU(UINT型)变量的值。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetDWU	FB		LS_Modbus_SetDWU (add_start:= , add_leng:= , add_setVal:= , bError=> , ErrorID=>);

变量：

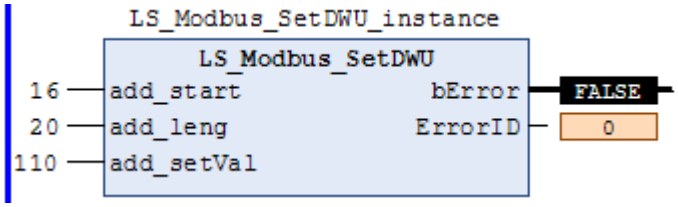
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,4999]	0	变量起始地址
	add_leng	INT	[1,5000]	0	变量个数
	add_setVal	UINT	遵照数据类型	0	设置的值
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

设置UINT型变量DWU [0]~ DWU [4999]的值，对应Modbus地址4x _5001~4x_10000。

例程：

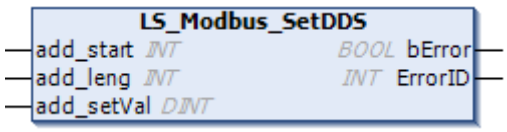
将DWU[16]至DWU[35]的值设置为110。程序如下：



LS_Modbus_SetDDS指令

该功能块用于设置DDS(DINT型)变量的值。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetDDS	FB		LS_Modbus_SetDDS (add_start:= , add_leng:= , add_setVal:= , bError=> , ErrorID=>);

变量：

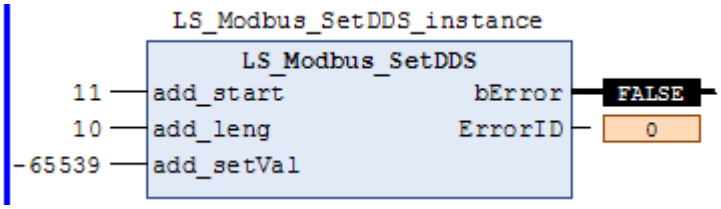
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,4999]	0	变量起始地址
	add_leng	INT	[1,5000]	0	变量个数
	add_setVal	DINT	遵照数据类型	0	设置的值
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

设置DINT型变量DDS [0]~ DDS [4999]的值，对应Modbus地址4x _10001~4x_19999。

例程：

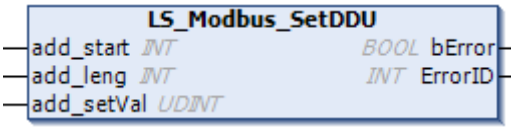
将DDS[11]至DDS[20]的值设置为-65539。程序如下：



DDU变量的操作

该功能块用于设置DDU(UDINT型)变量的值。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetDDU	FB		LS_Modbus_SetDDU (add_start:= , add_leng:= , add_setVal:= , bError=> , ErrorID=>);

变量：

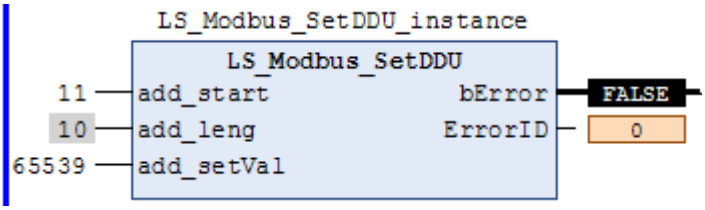
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,4999]	0	变量起始地址
	add_leng	INT	[1,5000]	0	变量个数
	add_setVal	UDINT	遵照数据类型	0	设置的值
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

设置UDINT型变量DDU [0]~ DDU [4999]的值，对应Modbus地址4x _20001~4x_29999。

例程：

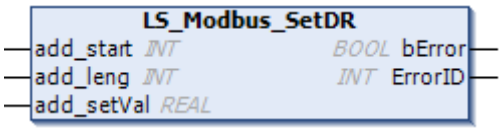
将DDU[11]至DDU[20]的值设置为65539。程序如下：



LS_Modbus_SetDR指令

该功能块用于设置DR(REAL型)变量的值。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetDR	FB		LS_Modbus_SetDR (add_start:= , add_leng:= , add_setVal:= , bError=> , ErrorID=>);

变量：

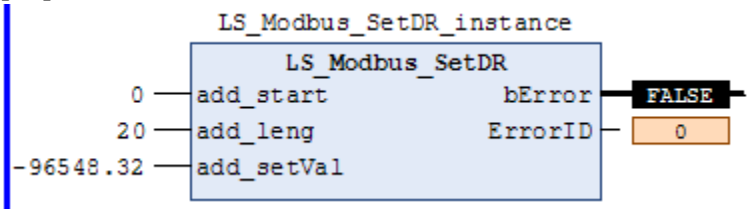
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,2499]	0	变量起始地址
	add_leng	INT	[1,2500]	0	变量个数
	add_setVal	REAL	遵照数据类型	0	设置的值
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

设置REAL型变量DR [0]~ DR [2499]的值，对应Modbus地址4x _30001~4x_34999。

例程：

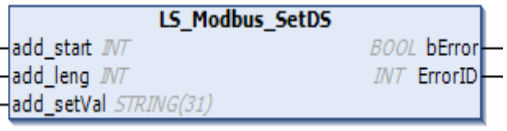
将DR[0]至DR[19]的值设置为-96548.32。程序如下：



LS_Modbus_SetDS指令

该功能块用于设置DS(STRING型)变量的值。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_Modbus_SetDS	FB		LS_Modbus_SetDS (add_start:= , add_leng:= , add_setVal:= , bError=> , ErrorID=>);

变量：

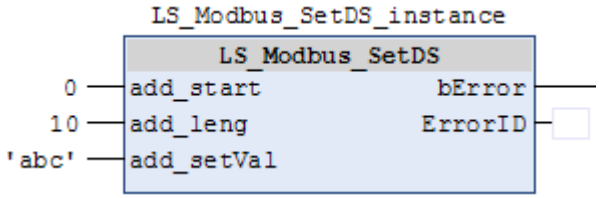
输入输出	名称	类型	有效范围	初始值	描述
输入	add_start	INT	[0,149]	0	变量起始地址
	add_leng	INT	[1,150]	0	变量个数
	add_setVal	STRING	遵照数据类型	0	设置的值
输出	bError	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	INT	遵照数据类型	0	错误码，1-变量起始地址值不正确，2-变量个数超出范围

功能：

设置STRING型变量DS [0]~ DS [149]的值，对应Modbus地址4x _35001~4x_37385。

例程：

将DS[0]至DS[9]的值设置为'abc'。程序如下：



3. 通过Modbus RTU与HMI通信的例程

PMC600运动控制器的RS232和RS485硬件端口均支持ModbusRTU协议通信，均可用于与HMI进行通信。此时控制器作为Modbus从站，HMI作为Modbus主站。

例程：通过TK6070iH触摸屏来控制电机的启动和停止，并将电机的实时位置和速度显示在触摸屏上，运动控制器与触摸屏通过RS232口进行连接。

1) 编写触摸屏程序：设置显示0~3轴实时位置的地址分别为4x_10001、4x_10003、4x_10005、4x_10007，设置显示0~3轴实时速度的地址分别为4x_10013、4x_10015、4x_10017、4x_10019；设置启动、停止运动的地址为0x_1、0x_2，清除计数的地址为0x_3，选择轴号的地址为4x_1，显示运动状态的指示灯地址为0x_4，如图4.46所示。

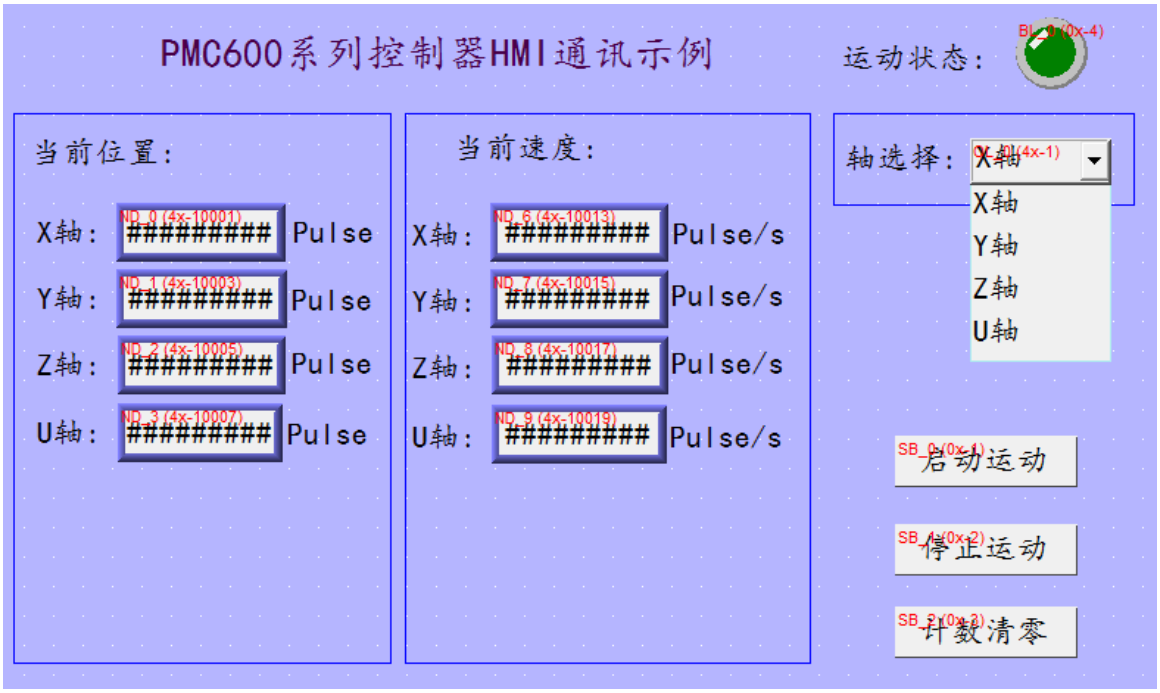


图4.46　触摸屏界面

2)　设置运动控制器为Modbus从站。双击编程工具iStudio左侧的设备树“Network Configuration”，打开网络配置界面，鼠标单击控制器，使能RS232下的Modbus从站，此时左侧设备树中会增加一个“Modbus_Slave_RS232_COM”，如图4.47所示。

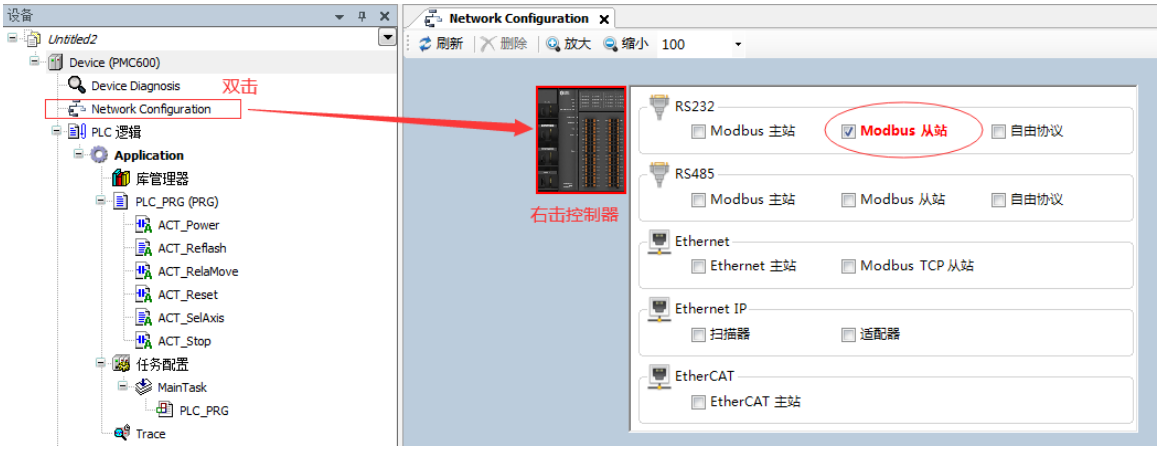


图4.47　Modbus从站启用

3)　双击设备树下的“Modbus_Slave_RS232_COM”，进入Modbus从站配置界面，配置串口波特率115200，偶校验、数据位8，停止为1及从站站号8。如图4.48所示。

4)　编写控制器端程序，按照之前的例程的方法在主程序下新建下面几大模块：控制器轴上电模块ACT_Power、实时位置及速度显示模块ACT_Reflash、相对运动模块ACT_RelaMove、位置清零模块ACT_Reset、停止模块ACT_Stop、选择轴模块ACT_SelAxis，然后在主程序中分别根据条件调用。

图4.48　Modbus从站配置

5)　完成控制器程序编程，编译OK后连接控制器，将程序下载到控制器中，点击运行按钮或者按F5，运行程序，如图4.49所示。

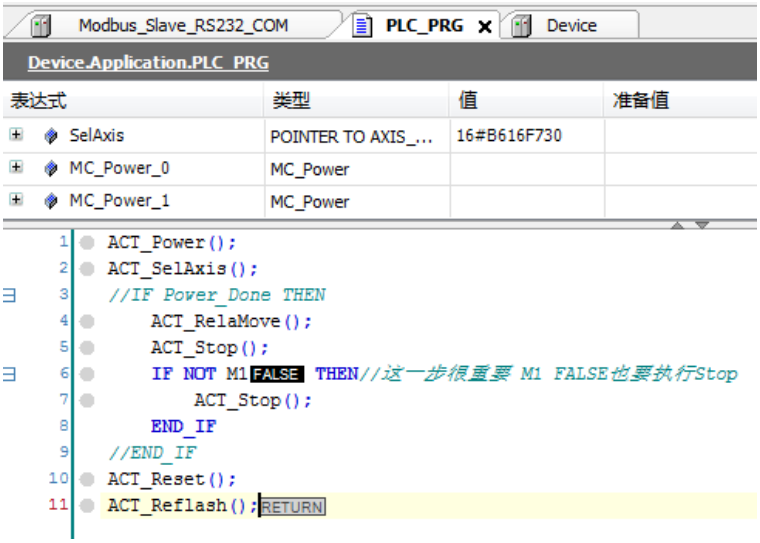


图4.49　程序运行结果

6)　设置触摸屏通讯参数，以TK6070IH为例，在触摸屏编程软件里菜单栏点击“编辑”-“系统参数设置”，设置控制器类型为MODBUS RTU，通讯接口类型RS-232，相关参数为波特率115200、数据位8、停止位1、校验位偶校验，控制器预设站号为8（与控制器Modbus从站站号一致），其他的默认即可，如图4.50所示。

7)　保存触摸屏程序，编译OK后，接好PC和控制器串口连接线(用交叉串口线，具体接法见用户手册)，选择在线模拟就可以实时控制控制器0~3轴运动、停止、清零计数器并监控每个轴的位置和速度信息，显示器运动状态如图4.51所示。

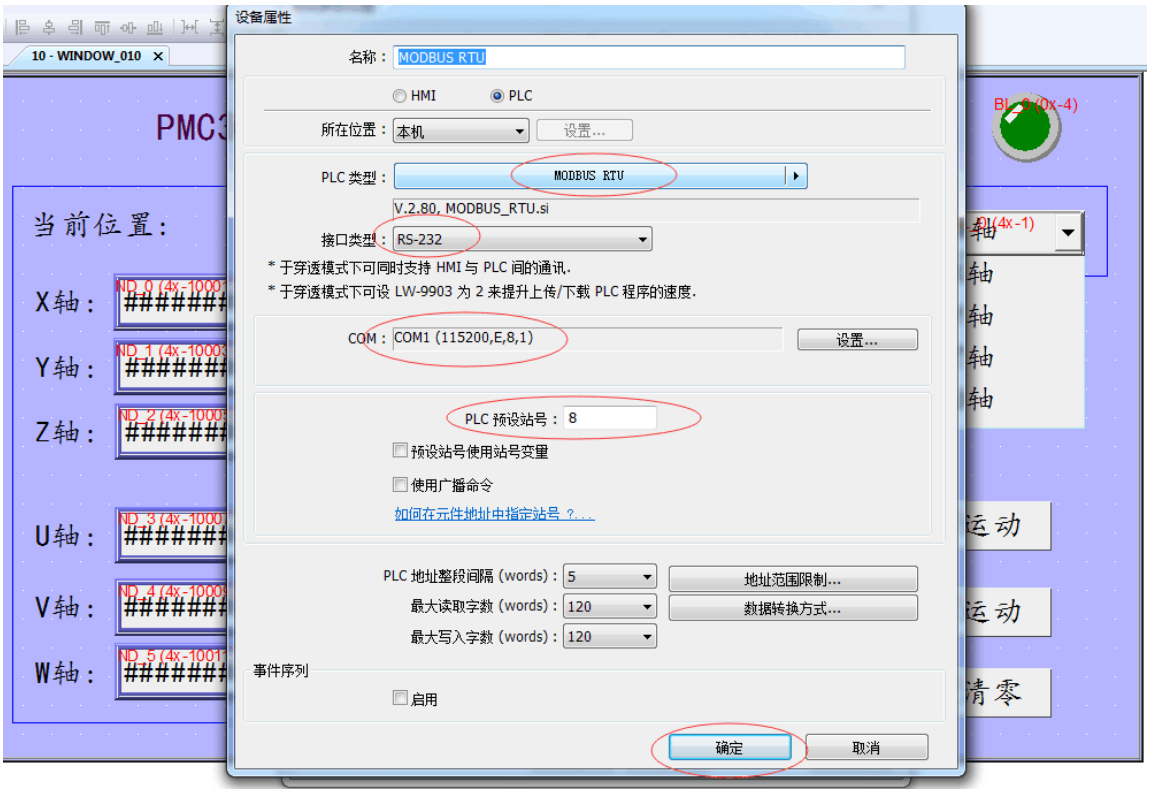


图4.50　触摸屏通讯参数设置

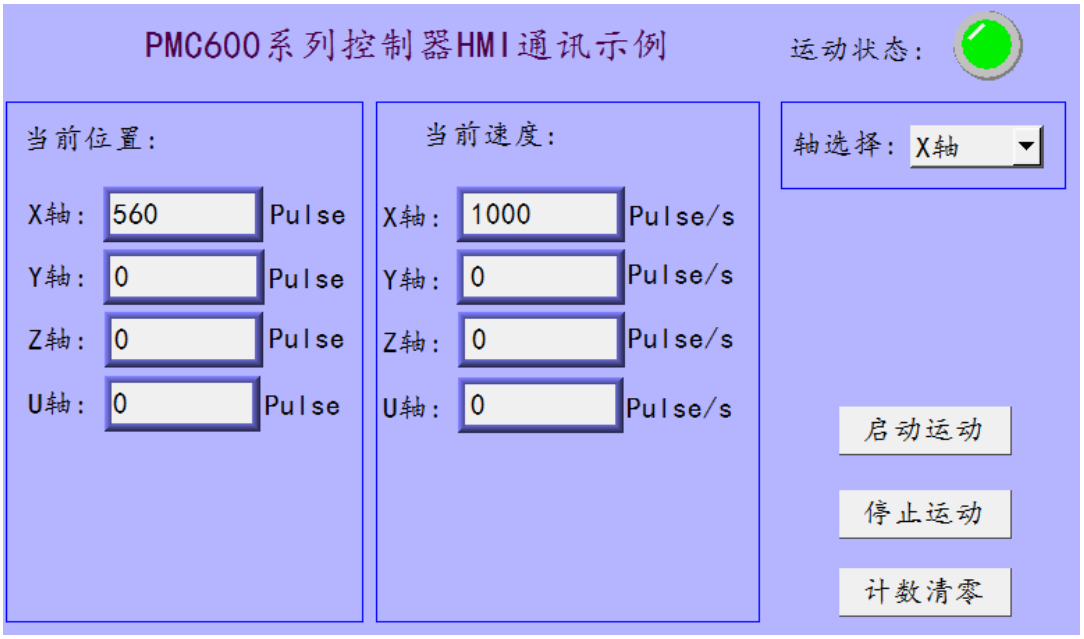


图4.51　X轴运动

整个程序需要注意的是：控制器程序运动模块只有一个，但程序中配置有4个轴，于是定义了一个指向轴类型的指针变量SelAxis:POINTER TO AXIS_REF_VIRTUAL_SM3，当触摸屏改变轴号时，指针变量SelAxis的值也跟着变化，具体可在上述选择轴模块ACT_SelAxis中看到，运动模块的轴号引用这个指针便是触摸屏设置的轴号了(引用格式SelAxis^)，停止模块和复位模块轴号引用也是如此。

通过RS485口与HMI连接与上述操作类似，只需在步骤2中使能控制器RS485下的

Modbus从站，同时将步骤6中的触摸屏接口类型选择RS485，串口参数设置一致即可。

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**HMI-RS232功能-Modbus-RS232**”。

触摸屏源代码参见PMC600软件资料中的“例程”文件夹中的“**HMI-RS232功能-HMI**”。

4. 通过Modbus TCP/IP与HMI通信的例程

运动控制器以太网运行标准Modbus TCP/IP协议时，可以与HMI通信，此时控制器作为Modbus从站，HMI作为Modbus主站。

例程：编写程序，设置控制器网口工作在Modbus协议，将触摸屏的通讯参数修改成Modbus TCPIP。

1) 新建工程并命名ModbusTCP，编程语言为功能块。
2) 设置控制器为Modbus TCP从站。双击编程工具iStudio左侧的设备树“Network Configuration”，打开网络配置界面，鼠标单击控制器，使能Ethernet下的Modbus TCP从站，此时左侧设备树中会增加一个“Modbus_TCP_Slave”，如图4.52所示。

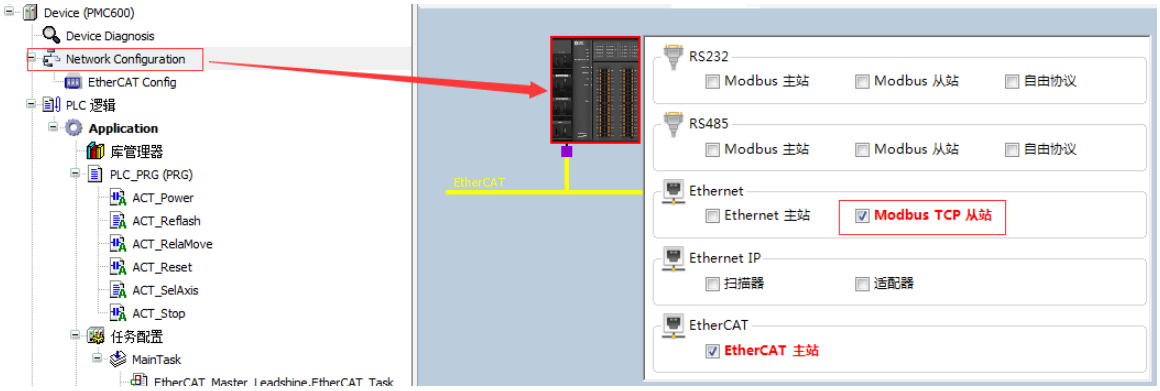


图4.52　Modbus TCP启用

3) 双击设备树下的“Modbus_TCP_Slave”，进入Modbus TCP从站配置界面，配置从站站号为8。如图4.53所示。
4) 编写控制器端程序，按照之前RS232 Modbus例程的方法在主程序下新建下面几大模块：控制器轴上电模块ACT_Power、实时位置及速度显示模块ACT_Reflash、相对运动模块ACT_RelaMove、位置清零模块ACT_Reset、停止模块ACT_Stop、选择轴模块ACT_SelAxis，然后在主程序中分别根据条件调用。如图4.54所示。

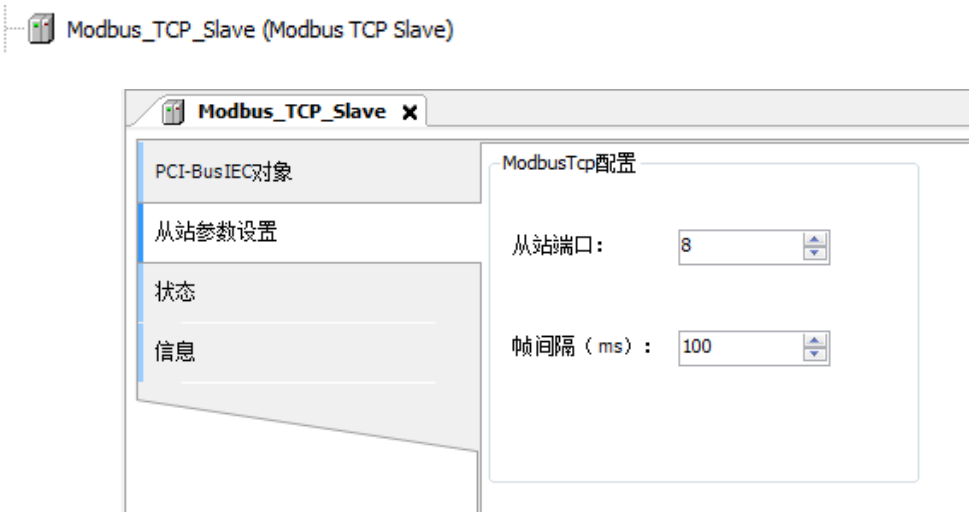


图4.53　Modbus TCP从站配置

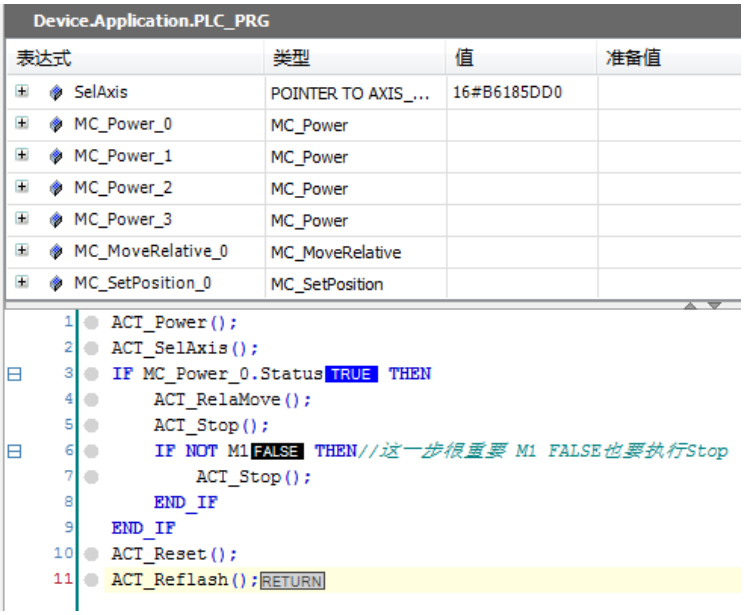


图4.54　运行程序

5) 编写HMI界面。将HMI的通讯端口参数按照图4.55所示配置：
6) 将控制器程序和HMI程序下载到相应的设备。控制器和HMI通过网线连接，通讯连接正常后，就可以实时控制控制器0~3轴运动、停止、清零计数器并监控每个轴的位置和速度信息，显示器运动状态如图4.56所示。

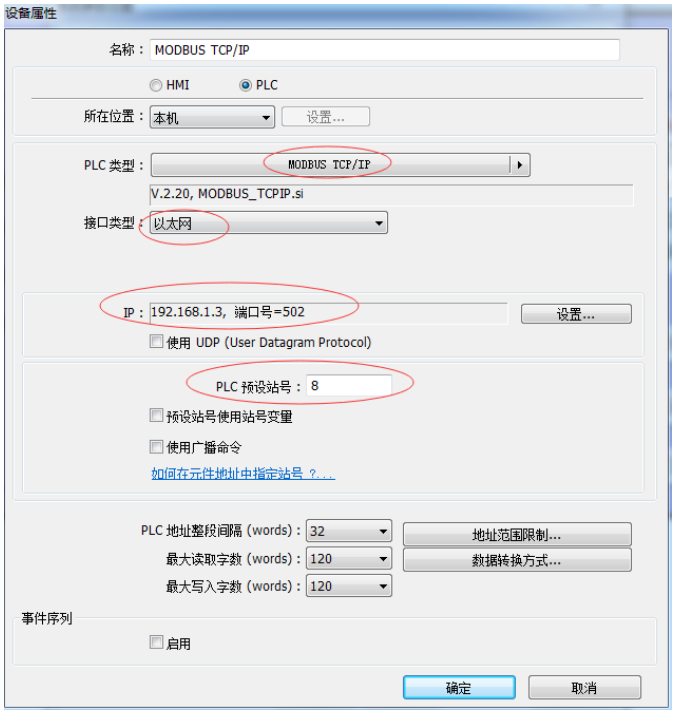


图4.55　HMI设置

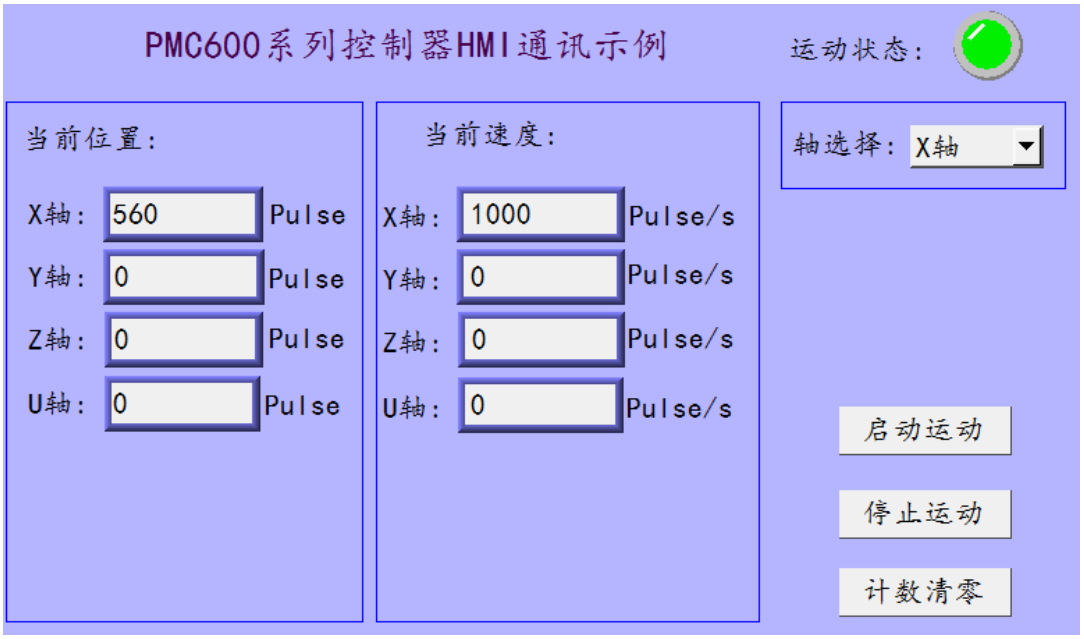


图4.56　X轴运动

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**HMI-ModbusTCP**”。

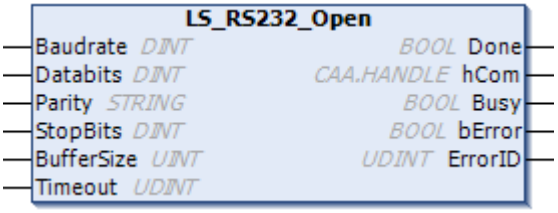
4.3.2 RS232无协议通信指令

设置RS232接口为无协议状态时，用户可直接对RS232口进行读写数据。雷赛为用户提供了RS232无协议功能块，定义了用户在RS232接口为自由协议状态时通讯的读写功能块。**该功能块由库文件“PMC_Communication”提供，程序中若想使用RS232无协议通信，必须确认库中是否存在PMC_ Communication文件，如不存在，需在工程中添加PMC_ Communication库。**

打开串口 LS_RS232_Open

打开串行端口。

指令外观：

指令	FB/ FUN	图形模块	结构文本
LS_RS232_Open	FB		LS_RS232_Open(Baudrate:= , Databits:= , Parity:= , StopBits:= , BufferSize:= , Timeout:= , Done=> , hCom=> , Busy=> , bError=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Baudrate	DINT	遵照描述	115200	波特率: 115200,38400,19200,9600,4800,2400,1200
	Databits	DINT	[7,8]	8	数据位:8,7
	Parity	STRING	遵照描述	'e'	校验: 校验:'e'或者'E','o'或者'O','n'或者'N'
	StopBits	DINT	[0,2]	1	0或1: 1stop bit; 2: 2stop bits
	BufferSize	UINT	遵照数据类型	1024	保持默认值，不需要修改
	Timeout	UDINT	遵照数据类型	0	串口打开超时时间
输出	Done	BOOL	TRUE/FALSE	FALSE	完成信号
	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
	Busy	BOOL	TRUE/FALSE	FALSE	正在执行
	bError	BOOL	TRUE/FALSE	FALSE	错误
	ErrorID	UDINT	遵照数据类型	0	错误号。1：参数不正确；2：数据位参数不正确；3：效验参数不正确；4：停止位参数不正确

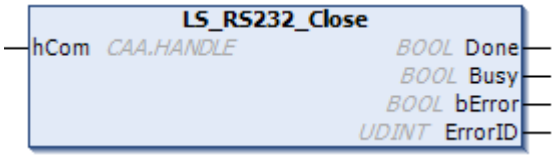
功能说明：

打开连接句柄hCom指定的串行端口。该模块Done为True以后，不要反复调用。

关闭串口LS_RS232_Close

功能：关闭串行端口。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_RS232_Close	FB		LS_RS232_Close(hCom:= , Done=> , Busy=> , bError=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
输出	Done	BOOL	TRUE/FALSE	FALSE	完成信号
	Busy	BOOL	TRUE/FALSE	FALSE	正在执行
	bError	BOOL	TRUE/FALSE	FALSE	错误
	ErrorID	UDINT	遵照数据类型	0	错误号。

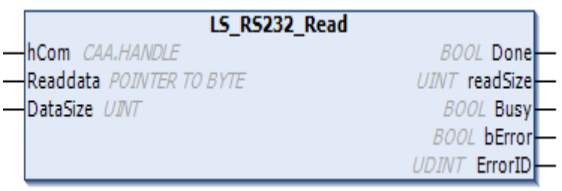
功能说明：

关闭连接句柄hCom指定的串行端口。

读串口数据 LS_RS232_Read

功能：从串行端口读取无协议的接收数据。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_RS232_Read	FB		LS_RS232_Read(hCom:= , Readdata:= , DataSize:= , Done=> , readSize=> , Busy=> , bError=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
	Readdata	POINTER TO BYTE	遵照数据类型	-	读取数据指针
	DataSize	UINT	遵照数据类型	0	需要读取数据大小
输出	Done	BOOL	TRUE/FALSE	FALSE	完成信号
	readSize	UINT	遵照数据类型	0	读取到的数据大小
	Busy	BOOL	TRUE/FALSE	FALSE	正在执行
	bError	BOOL	TRUE/FALSE	FALSE	错误
	ErrorID	UDINT	遵照数据类型	0	错误号

功能说明：

从连接句柄hCom指定的串行端口接收无协议的通信数据，接收的数据首先保存在串口端口的接收缓存区中。本指令将按接收数据大小DataSize，把接收缓存的数据传输到读取数据Readdata指向的内存中。

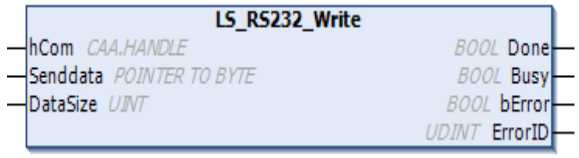
注意事项 ：

- ✧ 本指令仅在端口可使用时可以执行，仅可用于无协议模式。
- ✧ DataSize为0时，则不会将接收缓存区的数据传送到Readdata指向的内存中。
- ✧ 读取数据指针参数必须是指向字节类型的数组，否则如果只是单个变量的话，当读取的数据不止一个的时候会造成数据丢失；而且如果本次读到的数据长度小于上一次，则保存数据的数组的前面的数据会被覆盖掉，后面的还会保存上一次的数据，这时候需要看情况清掉对应的数据了。

发串口数据 LS_RS232_Write

功能：从串行端口进行无协议数据发送。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_RS232_Write	FB		LS_RS232_Write(hCom:= , Senddata:= , DataSize:= , Done=> , Busy=> , bError=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
	Senddata	POINTER TO BYTE	遵照数据类型	-	需要发送的数据指针
	DataSize	UINT	遵照数据类型	0	发送数据大小
输出	Done	BOOL	TRUE/FALSE	FALSE	完成信号
	Busy	BOOL	TRUE/FALSE	FALSE	正在执行
	bError	BOOL	TRUE/FALSE	FALSE	错误
	ErrorID	UDINT	遵照数据类型	0	错误号

功能说明：

从连接句柄hCom指定的串行端口进行无协议数据发送。发送的数据为Senddata指向的内容，发送的数据大小在DataSize中指定。数据发送完成Done置为TRUE。

注意事项：

- ✧ 本指令仅在端口可使用时可以执行，仅可用于无协议模式。
- ✧ DataSize为0时，不发送。

自由协议读写RS232接口数据方法：

1) 调用LS_RS232_Open指令配置RS232接口参数(包括通讯波特率、数据位、停止位、校验位等)并打开RS232接口；

2) 调用LS_RS232_Write、LS_RS232_Read指令实现RS232端口对外发送数据或者读取RS232接收到的数据；

3) 在使用完RS232接口后调用LS_RS232_Close指令关闭RS232接口。

RS232端口无协议例程

编写程序实现控制器232接口向PC发送数据”0123456789”，并接受PC发送给控制器的数据。

例程源码如下：

```
//声明
VAR
    LS_RS232_Open_instance: LS_RS232_Open;
    LS_RS232_Close_instance: LS_RS232_Close;
    LS_RS232_Write_instance: LS_RS232_Write;
    LS_RS232_Read_instance: LS_RS232_Read;
    iState: INT;
    Comhandle: PMC_Communication.CAA.HANDLE;
    dataBuf: ARRAY[0..127] OF BYTE;
    readByte: UINT;
    i: INT;
    Size: UINT;
END_VAR
```

```
//程序
CASE iState OF
0:
;
1:   //open

     LS_RS232_Open_instance(Baudrate:= 115200, Databits:= 8, Parity:= 'E', StopBits:= 1,
BufferSize:= 1024,
        Timeout:= ,    Done=> , hCom=>Comhandle ,Busy=>, bError=> , ErrorID=> );
     IF LS_RS232_Open_instance.Done THEN
        iState:=2;
     END_IF
2: //write
     LS_RS232_Write_instance(hCom:=Comhandle , Senddata:= ADR('0123456789'),
DataSize:= 10,
        Done=> , Busy=> , bError=> , ErrorID=> );
   IF LS_RS232_Write_instance.Done THEN
        iState:=3;
   END_IF
3: //read
   FOR i:=UINT_TO_INT(Size) TO 127 DO//如果当次读取字节数小于上一次，则清掉多余的
数据
            dataBuf[i]:=0;
   END_FOR
     LS_RS232_Read_instance(hCom:=Comhandle , Readdata:=ADR(dataBuf) , DataSize:=
128,
        Done=> , readSize=> readByte, Busy=> , bError=> , ErrorID=> );
   IF readByte>0 THEN
        Size:=readByte;
        iState:=2;
   END_IF
4:
   LS_RS232_Close_instance(hCom:= Comhandle,Done=> ,Busy=> ,bError=> ,ErrorID=> );
   IF LS_RS232_Close_instance.Done THEN
        iState:=5;
   END_IF
5:
   ;
END_CASE
```

例程实现：

1) 新建工程并命名RS-232，编程语言为结构化文本ST。

2) 在主程序中编写程序，依次配置232接口参数并打开232接口，RS232接口先向PC发送数据”0123456789”，发送完再等待读取232接收到的数据，读取完数据后关闭232接口，

如图4.57所示；

```
PLC_PRG
PROGRAM PLC_PRG
VAR
    LS_RS232_Open_instance: LS_RS232_Open;
    LS_RS232_Close_instance: LS_RS232_Close;
    LS_RS232_Write_instance: LS_RS232_Write;
    LS_RS232_Read_instance: LS_RS232_Read;

CASE iState OF
0:
;
1:  //open
    LS_RS232_Open_instance(Baudrate:= 115200, Databits:= 8, Parity:= 'E',
                           StopBits:= 1, BufferSize:= 1024, Timeout:= ,
                           Done=> , hCom=>Comhandle ,Busy=>bBusy_232,
                           bError=>bError_232 , ErrorID=>dErrorID_232 );

    IF LS_RS232_Open_instance.Done THEN
        iState:=2;
    END_IF

2: //write
    LS_RS232_Write_instance(hCom:=Comhandle , Senddata:= ADR('0123456789'),
                            DataSize:= 10, Done=> , Busy=> , bError=> ,
                            ErrorID=> );

    IF LS_RS232_Write_instance.Done THEN
        iState:=3;
    END_IF

    FOR i:=UINT_TO_INT(Size) TO 127 DO//当次读取字节数小于上一次，则清掉多余
        dataBuf[i]:=0;
    END_FOR
3: //read
    LS_RS232_Read_instance(hCom:=Comhandle , Readdata:=ADR(dataBuf) ,
                           DataSize:= 128, Done=> , readSize=> readByte,
                           Busy=> , bError=> , ErrorID=> );
```

图4.57　RS232通讯主程序

3)　接线连接PC和控制器232接口，打开串口助手并按照控制器232的参数打开对应的串口；如图4.58所示。

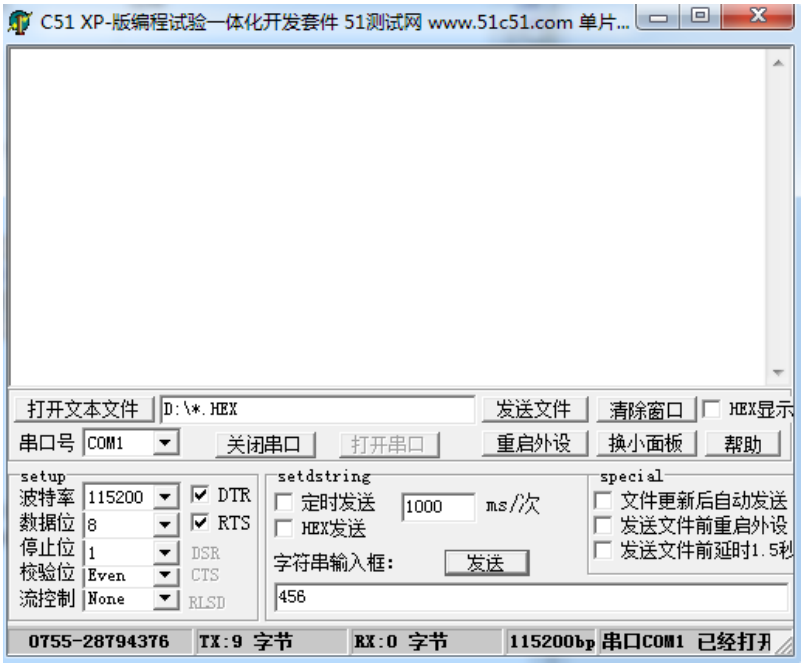


图4.58　连接串口助手

4)　完成控制器程序编程，编译OK后连接控制器，将程序下载到控制器中，点击运行按钮或者按F5，运行程序，并将iState的值强制为1，如图4.59所示。

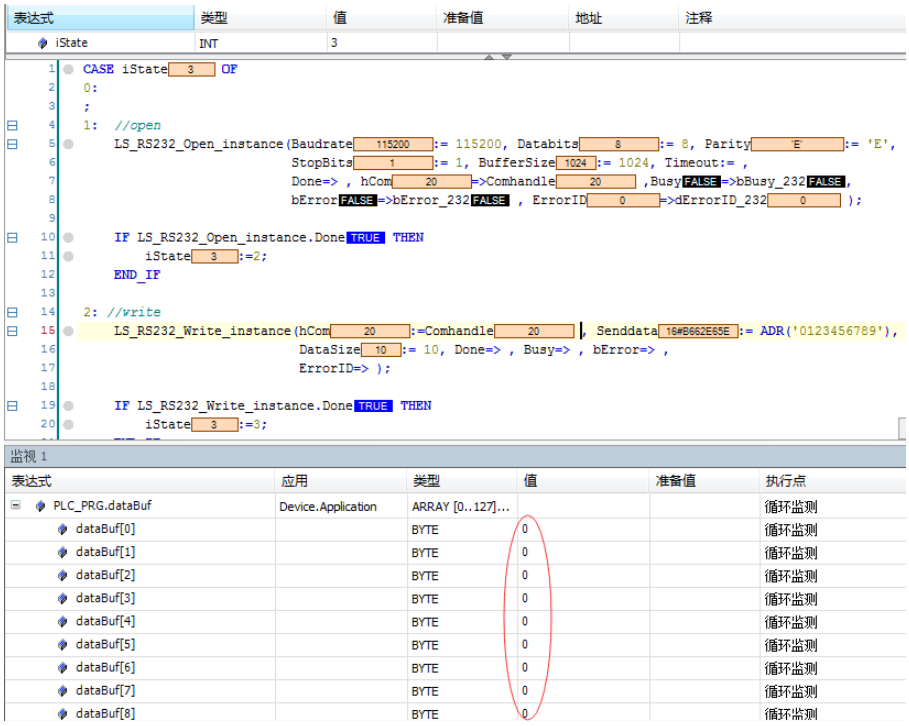


图4.59　RS232通讯控制器程序启动

5)　在串口助手数据接收区就会显示控制器发送来的数据，通过串口助手向控制器发送数据abcdef(ASCII码模式发送)，在控制器程序中监控232接收到的数据，结果如图4.60、4.61所示。

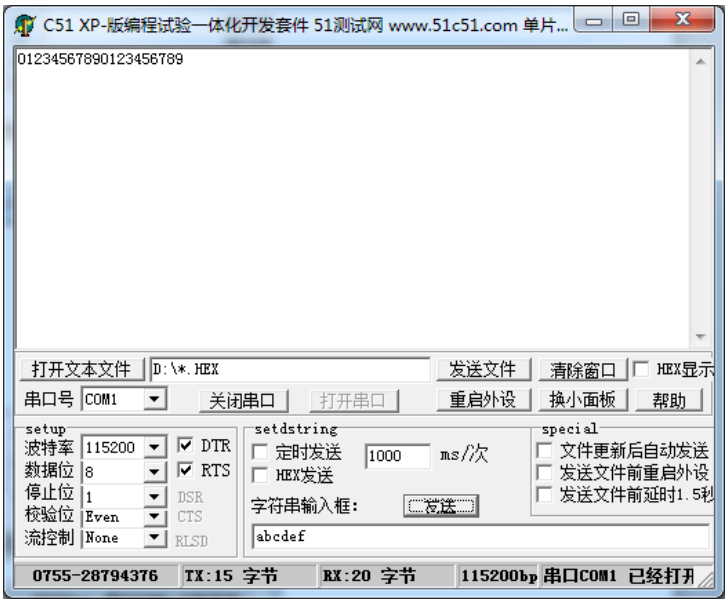


图4.60　RS232通讯PC端发送数据

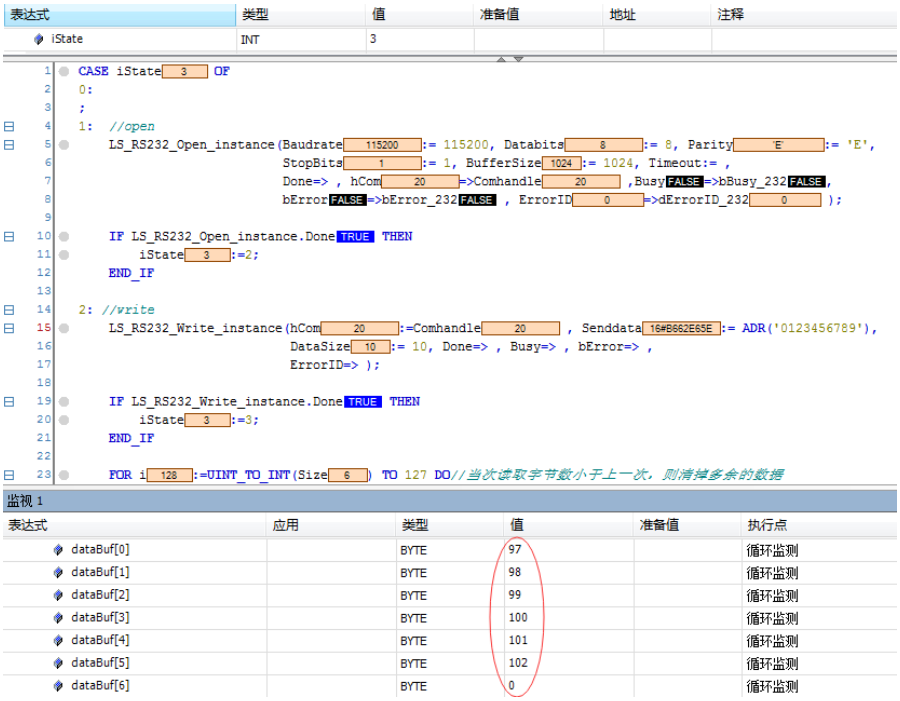


图4.61　RS232通讯控制器端接收数据

串口助手再次发送数据‘ertt’，监控控制器程序时，接收到的数据已发生变化，如图4.62、4.63所示。

图4.62　RS232通讯PC端发送数据

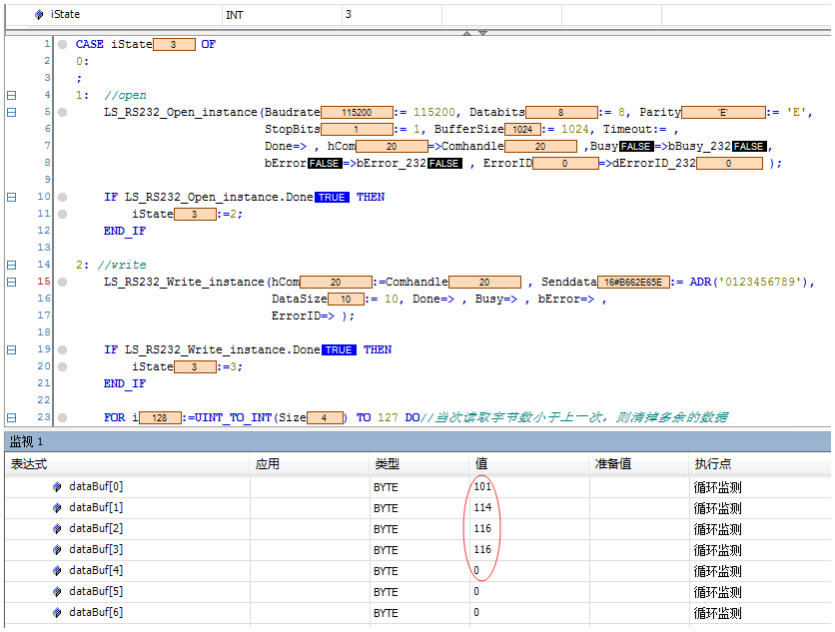


图4.63　RS232通讯控制器端接收数据

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**232通讯功能-RS232-NoProtocol**”。

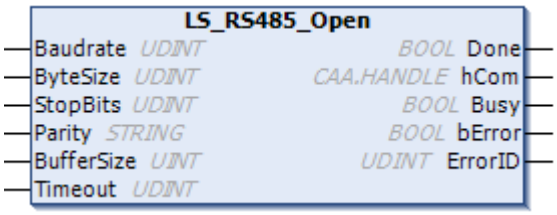
4.3.3　RS485无协议通信指令

设置RS485接口为无协议状态时，用户可直接对RS485接口进行读写数据。雷赛为用户提供了RS485无协议通信功能块，定义了用户在RS485接口为自由协议状态时通讯的读写功能块。**该功能块由库文件“PMC_Communication”提供，程序中若想使用RS485无协议通信，必须确认库中是否存在PMC_ Communication文件，如不存在，需在工程中添加PMC_ Communication库。**

打开RS485端口 LS_RS485_Open

功能：打开RS485端口。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_RS485_Open	FB		LS_RS485_Open(Baudrate:= , ByteSize:= , StopBits:= , Parity:= , BufferSize:= , Timeout:= , Done=> , hCom=> , Busy=> , bError=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Baudrate	DINT	遵照描述	9600	波特率:115200,38400,19200,9600,4800,2400,1200
	Databits	DINT	[7,8]	8	数据位:8,7
	Parity	STRING	遵照描述	'e'	效验:效验:'e'或者'E','o'或者'O','n'或者'N'
	StopBits	DINT	[0,2]	0	0或1: 1stop bit; 2: 2stop bits
	BufferSize	UINT	遵照数据类型	1024	保持默认值，不需要修改
	Timeout	UDINT	遵照数据类型	0	超时时间
输出	Done	BOOL	TRUE/FALSE	FALSE	完成信号
	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
	Busy	BOOL	TRUE/FALSE	FALSE	打开状态中
	bError	BOOL	TRUE/FALSE	FALSE	错误
	ErrorID	UDINT	遵照数据类型	0	错误号。1：参数不正确；2：数据位参数不正确；3：效验参数不正确；4：停止位参数不正确

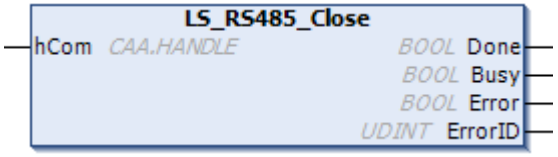
功能说明：

打开连接句柄hCom指定的RS485端口。该模块Done为True以后，不要反复调用。

关闭RS485端口 LS_RS485_Close

功能：关闭RS485端口。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_RS485_Close	FB		LS_RS485_Close(hCom:= , Done=> , Busy=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
输出	Done	BOOL	TRUE/FALSE	FALSE	完成
	Busy	BOOL	TRUE/FALSE	FALSE	执行中
	Error	BOOL	TRUE/FALSE	FALSE	错误状态，False-没有错误
	ErrorID	UDINT	遵照数据类型	0	错误码

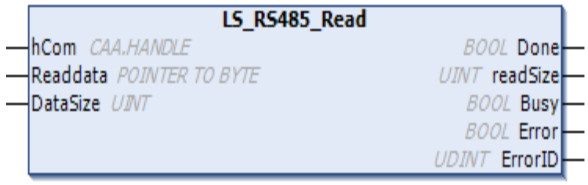
功能说明：

关闭连接句柄hCom指定的RS485端口。

接收RS485数据 LS_RS485_Read

功能：从RS485端口读取无协议的接收数据。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_RS485_Read	FB		LS_RS485_Read(hCom:= , Readdata:= , DataSize:= , Done=> , readSize=> , Busy=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
	Readdata	POINTER TO BYTE	遵照数据类型	-	读取数据指针
	DataSize	UINT	遵照数据类型	0	需要读取数据大小
输出	Done	BOOL	TRUE/FALSE	FALSE	完成信号
	readSize	UINT	遵照数据类型	0	读取到的数据大小

	Busy	BOOL	TRUE/FALSE	FALSE	正在执行
	Error	BOOL	TRUE/FALSE	FALSE	错误
	ErrorID	UDINT	遵照数据类型	0	错误号

功能说明：

从连接句柄hCom指定的RS485端口接收无协议的通信数据，接收的数据首先保存在RS485端口的接收缓存区中。本指令将按接收数据大小DataSize，把接收缓存的数据传输到读取数据Readdata指向的内存中。

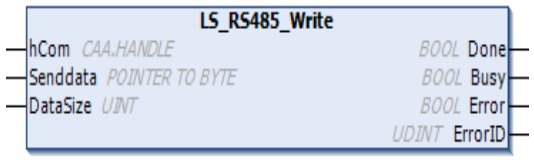
注意事项：

- ✧ 本指令仅在端口可使用时可以执行，仅可用于无协议模式。
- ✧ DataSize为0时，则不会将接收缓存区的数据传送到Readdata指向的内存中。
- ✧ 读取数据指针参数必须是指向字节类型的数组，否则如果只是单个变量的话，当读取的数据不止一个的时候会造成数据丢失；而且如果本次读到的数据长度小于上一次，则保存数据的数组的前面的数据会被覆盖掉，后面的还会保存上一次的数据，这时候需要看情况清掉对应的数据了。

发送RS485数据 LS_RS485_Write

功能：从RS485口发送无协议数据。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_RS485_Write	FB		LS_RS485_Write(hCom:= , Senddata:= , DataSize:= , Done=> , Busy=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	hCom	CAA.HANDLE	遵照数据类型	0	连接句柄
	Senddata	POINTER TO BYTE	遵照数据类型	-	需要发送的数据指针
	DataSize	UINT	遵照数据类型	0	发送数据大小
输出	Done	BOOL	TRUE/FALSE	FALSE	完成信号
	Busy	BOOL	TRUE/FALSE	FALSE	数据发送中
	Error	BOOL	TRUE/FALSE	FALSE	错误
	ErrorID	UDINT	遵照数据类型	0	错误号

功能说明：

从连接句柄hCom指定的RS485端口进行无协议数据发送。发送的数据为Senddata指向的内容，发送的数据大小在DataSize中指定。数据发送完成Done置为TRUE。

注意事项：

- ✧ 本指令仅在端口可使用时可以执行，仅可用于无协议模式。
- ✧ DataSize为0时，不发送。

自由协议读写RS485接口数据方法：

1) 调用LS_RS485_Open指令配置RS485接口参数(包括通讯波特率、数据位、停止位、校验位等)并打开485接口；

2) 调用LS_RS485_Write、LS_RS485_Read指令实现RS485端口对外发送数据或者读取RS485接收到的数据；

3) 在使用完RS485接口后调用LS_RS485_Close指令关闭RS485接口。

RS485端口无协议例程

编写程序实现控制器485接口向PC发送数据”0123456789”，并接受PC发送给控制器的数据”abc123”。

```
//声明
VAR
LS_RS485_Open_instance: LS_RS485_Open;
   LS_RS485_Close_instance: LS_RS485_Close;
   LS_RS485_Write_instance: LS_RS485_Write;
   LS_RS485_Read_instance: LS_RS485_Read;
   iState: INT;
   Comhandle: PMC_Communication.CAA.HANDLE;
   dataBuf: ARRAY[0..127] OF BYTE;
   readByte: UINT;
   i: INT;
   Size: UINT;
END_VAR

//程序
CASE iState OF
0:
   ;
1:   //open
      LS_RS485_Open_instance(Baudrate:= 9600, ByteSize:= 8, StopBits:= 0, Parity:= 'E', BufferSize:= ,
            Timeout:= ,    Done=> , hCom=>Comhandle , bError=> , ErrorID=> );
      IF LS_RS485_Open_instance.Done THEN
            iState:=2;
      END_IF
2: //write
      LS_RS485_Write_instance(hCom:=Comhandle , Senddata:= ADR('0123456789'), DataSize:= 10,
```

```
        Done=> , Busy=> , Error=> , ErrorID=> );
    IF LS_RS485_Write_instance.Done THEN
        iState:=3;
    END_IF
3: //read
    FOR i:=UINT_TO_INT(Size) TO 127 DO//当次读取字节数小于上一次，则清掉多余的数据
        dataBuf[i]:=0;
    END_FOR
    LS_RS485_Read_instance(hCom:=Comhandle , Readdata:=ADR(dataBuf) , DataSize:= 128,
        Done=> , readSize=> readByte, Busy=> , Error=> , ErrorID=> );
    IF readByte>0 THEN
        Size:=readByte;
        iState:=2;
    END_IF
4:
    LS_RS485_Close_instance(hCom:= Comhandle,Done=> ,Busy=> ,Error=> ,ErrorID=> );
    IF LS_RS485_Close_instance.Done THEN
        iState:=5;
    END_IF
5:
  ;
END_CASE
```

4.3.4 以太网通信指令

控制器Ethernet网口支持标准TCP/IP协议、UDP/IP协议以及Modbus TCP/IP协议。控制器以太网运行TCP/IP协议、UDP/IP协议可实现多台控制器组网，在上位机编写程序解析数据来监控每台控制器的状态。以太网运行标准TCP/IP、UDP/IP协议的对应操作，相关指令包含在“**PMC_Communication**”库中。**程序中若要使用TCP/IP、UDP/IP协议通讯，必须确认库中是否存在PMC_ Communication文件，如不存在，需要在工程中添加PMC_Communication库。**

TCP/IP协议通讯的相关指令

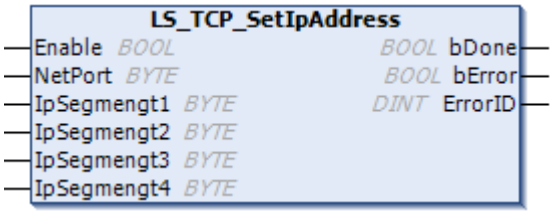
表4.17 TCP/IP协议通讯的相关指令

名称	功能
LS_TCP_GetIpAddress	获取控制器网口的IP地址
LS_TCP_SetIpAddress	设置控制器网口的IP地址
LS_TCP_SetClient	将本控制器设置为Client
LS_TCP_SetServer	将本控制器设置为Server
LS_TCP_WriteData	发送数据
LS_TCP_ReadData	读取数据

LS_TCP_SetIPAddress指令

功能：该功能块用作设置本机内置网口的IP地址。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_TCP_SetIp Address	FB		LS_TCP_SetIpAddress(Enable:= , NetPort:= , IpSegmengt1:= , IpSegmengt2:= , IpSegmengt3:= , IpSegmengt4:= , bDone=> , bError=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Enable	BOOL	TRUE/FALSE	FALSE	上升沿有效，状态从False切换到True，配置IP地址
	NetPort	BYTE	[1,2]	0	网口编号，从1号开始
	IpSegmengt1	BYTE	[0,255]	0	Ip地址第1段
	IpSegmengt2	BYTE	[0,255]	0	Ip地址第2段
	IpSegmengt3	BYTE	[0,255]	0	Ip地址第3段
	IpSegmengt4	BYTE	[0,255]	0	Ip地址第4段
输出	bDone	BOOL	TRUE/FALSE	FALSE	操作完成
	bError	BOOL	TRUE/FALSE	FALSE	FALSE-没有错误；True-错误
	ErrorID	DINT	遵照数据类型	0	错误号。1.IP地址第1段数据值不能为0；2.NetPort值超出范围

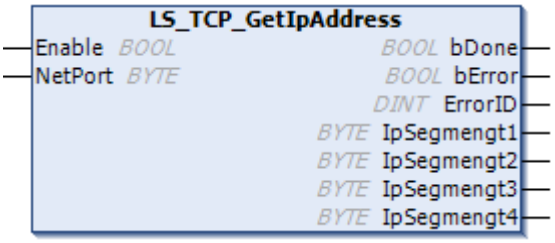
功能说明：

配置控制器内置网口的IP地址，硬件网口编号由NetPort指定，从1开始编号，1对应eth1网口，2对应eth2和3网口，0为EtherCAT口不可设置，IP地址由IpSegmengt参数指定，该功能块Enable上升沿生效。

LS_TCP_GetIPAddress指令

功能：该功能块用作获取内置网口的IP地址。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_TCP_GetIpAddress	FB		LS_TCP_GetIpAddress(Enable:= , NetPort:= , bDone=> , bError=> , ErrorID=> , IpSegmengt1=> , IpSegmengt2=> , IpSegmengt3=> , IpSegmengt4=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Enable	BOOL	TRUE/FALSE	FALSE	上升沿有效，状态从False切换到True，读取IP地址
	NetPort	BYTE	[1,2]	0	网口编号，从1号开始
输出	bDone	BOOL	TRUE/FALSE	FALSE	操作完成
	bError	BOOL	TRUE/FALSE	FALSE	FALSE-没有错误；True-错误
	ErrorID	DINT	遵照数据类型	0	错误号。
	IpSegmengt1	BYTE	[0,255]	0	Ip地址第1段
	IpSegmengt2	BYTE	[0,255]	0	Ip地址第2段
	IpSegmengt3	BYTE	[0,255]	0	Ip地址第3段
	IpSegmengt4	BYTE	[0,255]	0	Ip地址第4段

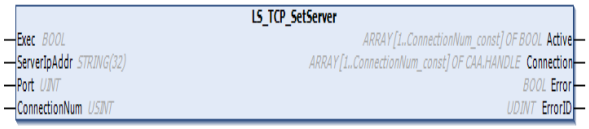
功能说明：

获取NetPort所指定编号的网口IP地址，该功能块Enable上升沿有效。

LS_TCP_SetServer指令

该功能块用作将本控制器设置为TCP服务器，可与一个客户端点对点通信。

指令外观：

指令	FB/ FUN	图形模块	结构文本
LS_TCP_Set Server	FB		LS_TCP_SetServer(Exec:= , ServerIpAddr:= , Port:= , ConnectionNum, Active=> , Connection=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	该信号必须保持
	ServerIpAddr	STRING(32)	遵照数据类型	192.168.1.3	设置服务器的IP地址，即本机的IP地址
	Port	UINT	遵照数据类型	5000	设置服务器的端口号
	ConnectionNum	USINT	遵照数据类型	1	设置可连接客户端个数，最多不能超过50个
输出	Active	ARRAY[1..ConnectionNum] OF BOOL	遵照数据类型	[50(FALSE)]	服务器连接激活
	Connection	ARRAY[1..ConnectionNum] OF CAA.HANDLE	遵照数据类型	[50(1)]	服务器连接句柄
	Error	BOOL	TRUE/FALSE	FALSE	服务器错误
	ErrorID	UDINT	遵照数据类型	0	错误码。

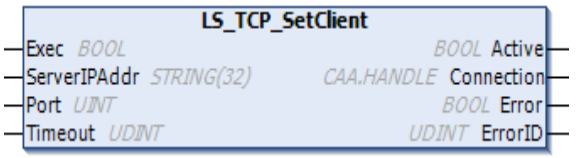
功能说明：

将IP地址为ServerIpAddr的网口设置为服务器，服务器端口号由Port指定。

LS_TCP_SetClient指令

该功能块用作将本控制器设置为客户端。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_TCP_SetClient	FB		LS_TCP_SetClient(Exec:= , ServerIPAddr:= , Port:= , Timeout:= , Active=> , Connection=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	该信号必须保持
	ServerIpAddr	STRING(32)	遵照数据类型	192.168.1.112	设置需要连接的服务器IP地址
	Port	UINT	遵照数据类型	5000	设置需要连接的服务器端口号
	Timeout	UDINT	遵照数据类型	8000	设置错误超时时间，单位us
输出	Active	BOOL	遵照数据类型	FALSE	连接激活
	Connection	CAA.HANDLE	遵照数据类型	0	连接句柄
	Error	BOOL	TRUE/FALSE	FALSE	连接错误
	ErrorID	UDINT	遵照数据类型	0	错误码。

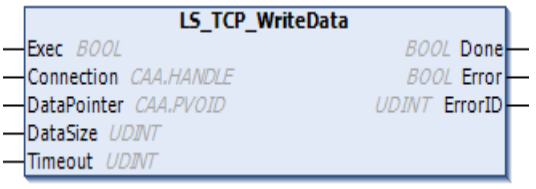
功能说明：

设置本控制器为TCP通信的客户端，并连接IP地址为ServerIpAddr，端口号为Port的TP通信的服务器。

LS_TCP_WriteData指令

该功能块用作TCP通信发送数据。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_TCP_WriteData	FB		LS_TCP_WriteData(Exec:= , Connection:= , DataPointer:= , DataSize:= , Timeout:= , Done=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	执行
	Connection	CAA.HANDLE	遵照数据类型	0	连接句柄
	DataPointer	CAA.PVOID	遵照数据类型	-	设置需要发送的数据指针
	DataSize	UDINT	遵照数据类型	0	发送数据大小
	Timeout	UDINT	遵照数据类型	8000	发送超时时间，单位us
输出	Done	BOOL	TRUE/FALSE	FALSE	发送完成
	Error	BOOL	TRUE/FALSE	FALSE	发送数据错误
	ErrorID	UDINT	遵照数据类型	0	错误码。

功能说明：

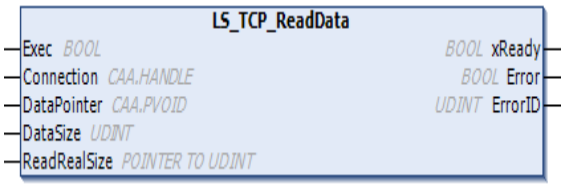
由Connection指定的连接句柄将发送数据DataPointer发送出去，发送数据的大小在DataSize中指定。

即使Exec变为FALSE，本指令也将一直执行到最后。处理是否正常结束，可通过Done的值是否为TRUE来确认。DataSzie的值为0，不发送数据。

LS_TCP_ReadData指令

该功能块用作TCP通信接收数据。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_TCP_ReadData	FB		LS_TCP_ReadData(Exec:= , Connection:= , DataPointer:= , DataSize:= , ReadRealSize:= , xReady=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	使能执行
	Connection	CAA.HANDLE	遵照数据类型	0	连接句柄
	DataPointer	CAA.PVOID	遵照数据类型	-	读取数据的指针
	DataSize	UDINT	遵照数据类型	0	数据指针指向的数组大小
	ReadRealSize	POINTE	遵照数据类型	-	实际读取的数据大小

		R TO UDINT			
输出	xReady	BOOL	TRUE/FALSE	FALSE	True时表示接收到数据，False表示未接收到数据
	Error	BOOL	TRUE/FALSE	FALSE	发送数据错误
	ErrorID	UDINT	遵照数据类型	0	错误码。

功能说明：

将Connection中指定的通信连接句柄中接收的数据保存在DataPointer中，保存数据的大小在DataSize中指定，保存数据后，将实际保存的数据大小代入ReadRealSize中。指令正常结束时（xReady的值变为TRUE），完成向DataPointer保存数据。

即使Exec变为FALSE，本指令也将一直执行到最后。处理是否正常结束，可通过xReady的值是否为TRUE来确认。DataSzie的值为0，不读取接收数据。

UDP/IP通讯协议相关指令

表4.18　UDP/IP通讯协议指令

名称	功能
LS_UDP_SetConnect	启动UDP/IP服务
LS_UDP_WriteData	发送数据
LS_UDP_ReadData	接收数据

LS_UDP_SetConnect指令

该功能块用作启动UDP/IP服务。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_UDP_SetConnect	FB		LS_UDP_SetConnect(Exec:= , LocalIpAddr:= , Port:= , Active=> , Connection=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	该信号必须保持
	LocalIpAddr	STRING (32)	遵照数据类型	192.168.1.3	本机IP地址
	Port	UINT	遵照数据类型	5000	本机端口号
输出	Active	BOOL	TRUE/FALSE	FALSE	UDP连接激活
	Connection	CAA.H ANDLE	遵照数据类型	0	连接句柄
	Error	BOOL	TRUE/FALSE	FALSE	连接错误
	ErrorID	UDINT	遵照数据类型	0	错误码。

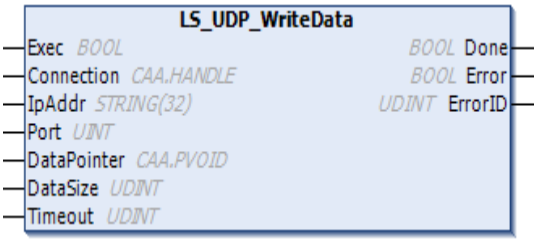
功能说明：

将LocalIpAddr中指定的IP地址和Port中指定的端口号启用UDP/IP服务。Active的值为TRUE后，UDP连接激活，生成的UDP连接句柄保存在Connection中，用作发送和接收数据。

LS_UDP_WriteData指令

该功能块用作UDP发送数据。

指令外观：

指令	FB/ FUN	图形模块	结构文本
LS_UDP_WriteData	FB		LS_UDP_WriteData(Exec:= , Connection:= , IpAddr:= , Port:= , DataPointer:= , DataSize:= , Timeout:= , Done=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始值	描述
输入	Exec	BOOL	TRUE/FALSE	FALSE	使能执行
	Connection	CAA.HAND LE	遵照数据类型	0	当前发送数据端句柄
	IpAddr	STRING(32)	遵照数据类型	192.168.1.5	设置接收数据端的IP地址
	Port	UINT	遵照数据类型	5000	设置接收数据端IP地址对应的端口号
	DataPointer	CAA.PVOID	遵照数据类型		需要发送的数据指针
	DataSize	UDINT	遵照数据类型	0	发送数据大小
	Timeout	UDINT	遵照数据类型	10000	发送超时时间，单位us

输出	Done	BOOL	TRUE/FALSE	FALSE	数据发送完成
	Error	BOOL	TRUE/FALSE	FALSE	发送数据错误
	ErrorID	UDINT	遵照数据类型	0	错误码。

功能说明：

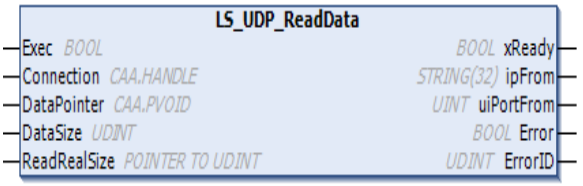
由启动UDP/IP服务生成的UDP连接句柄Connection将DataPointer数据发送出去，发送的数据大小在DataSize中指定，指令正常结束时（Done的值为TRUE），完成向发送缓存保存DataPointer数据。

即使Exec变为FALSE，本指令也将一直执行到最后。处理是否正常结束，可通过Done的值是否为TRUE来确认。DataSzie的值为0，不发送数据。

LS_UDP_ReadData指令

该功能块用作UDP接收数据。

指令外观：

指令	FB/FUN	图形模块	结构文本
LS_UDP_ReadData	FB		LS_UDP_ReadData(Exec:= , Connection:= , DataPointer:= , DataSize:= , ReadRealSize:= , xReady=> , ipFrom=> , uiPortFrom=> , Error=> , ErrorID=>);

变量：

输入输出	名称	类型	有效范围	初始化	注释
输入	Exec	BOOL	TRUE/FALSE	FALSE	使能执行
	Connection	CAA.HANDLE	遵照数据类型	0	接收端句柄
	DataPointer	CAA.PVOID	遵照数据类型		读取数据的指针
	DataSize	UDINT	遵照数据类型	0	数据指针指向的数组大小
	ReadRealSize	POINTER TO UDINT	遵照数据类型		实际读取的数据大小
输出	xReady	BOOL	TRUE/FALSE	FALSE	True表示接收到数据；False表示无数据接收
	ipFrom	STRING（32）	遵照数据类型		发送方的IP地址
	uiPortFrom	UINT	遵照数据类型		发送方的端口号
	Error	BOOL	TRUE/FALSE	FALSE	接收错误状态
	ErrorID	UDINT	遵照数据类型	0	错误码

功能说明：

执行启动UDP/IP服务生成的UDP连接句柄Connection的接收请求，将Connection中接收的数据保存在DataPointer中，保存数据的大小在DataSize中指定，保存数据后，将实际保存的数据大小代入ReadRealSize中。指令正常结束时（xReady的值变为TRUE），完成向DataPointer保存数据。

即使Exec变为FALSE，本指令也将一直执行到最后。处理是否正常结束，可通过xReady的值是否为TRUE来确认。DataSzie的值为0，不读取接收数据。

TCP/IP协议通信例程

运动控制器以太网运行标准TCP/IP协议的对应操作相关指令包含在“**PMC_Communication**”库中，如图4.64所示。

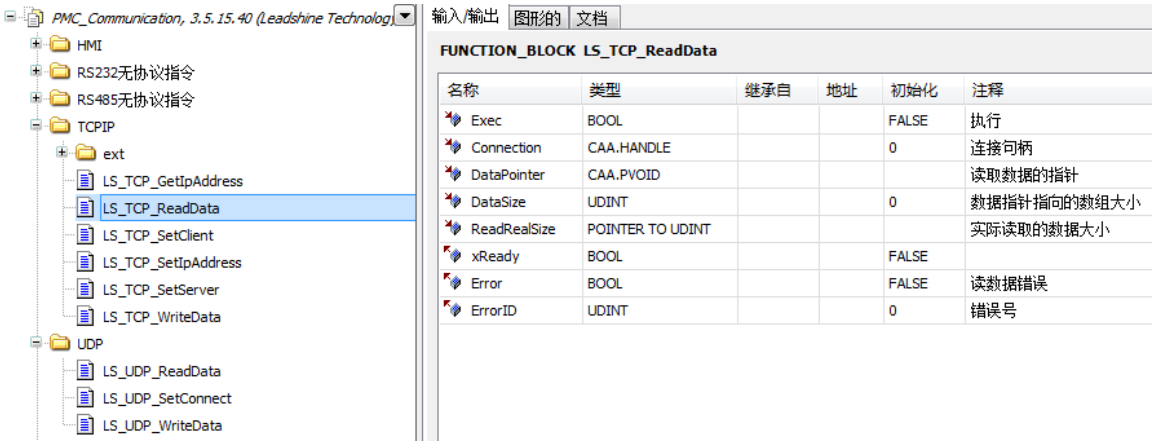


图4.64　PMC_Communication库中TCP/IP协议的相关操作指令

需要注意的是LS_TCP_SetIpAddress设置IP指令最好在使用前单独一个程序改IP，在功能程序中不要改，而且设置完成后立即生效，该参数会保存。控制器默认IP地址为192.168.1.3，用户可直接使用该IP地址，而无需调用LS_TCP_SetIpAddress设置IP。

TCP/IP通讯使用方法：

1) 配置控制器工作模式：调用LS_TCP_SetServer、LS_TCP_SetClient设置控制器工作在服务器模式还是客户端模式，并初始化连接；

2) 收发数据：在确认连接成功后，调用LS_TCP_ReadData和LS_TCP_WriteData指令读取控制器收到的数据并向外发送数据(参数连接句柄由设置控制器工作模式指令生成)；

3) 根据程序功能实现启动连接、收发数据、中断通讯、统计收发数据的大小等。

Server模式例程：设置控制器工作在Server模式，PC为Client模式，向PC发送数据“0123456789”，并接受PC发送给控制器的数据“abc123”。

1)　新建工程并命名TCPIP-Server，编程语言为结构化文本ST。

2)　在设备栏右击主程序“POU_TCPTest_Server”，新建一个动作ACT_Server，编程语言

为连续功能图(CFC)，负责设置控制器工作在Server模式-即服务器IP为192.168.1.3，端口号为5000，以及收发数据主体，并添加对应的参数，如图4.65所示；

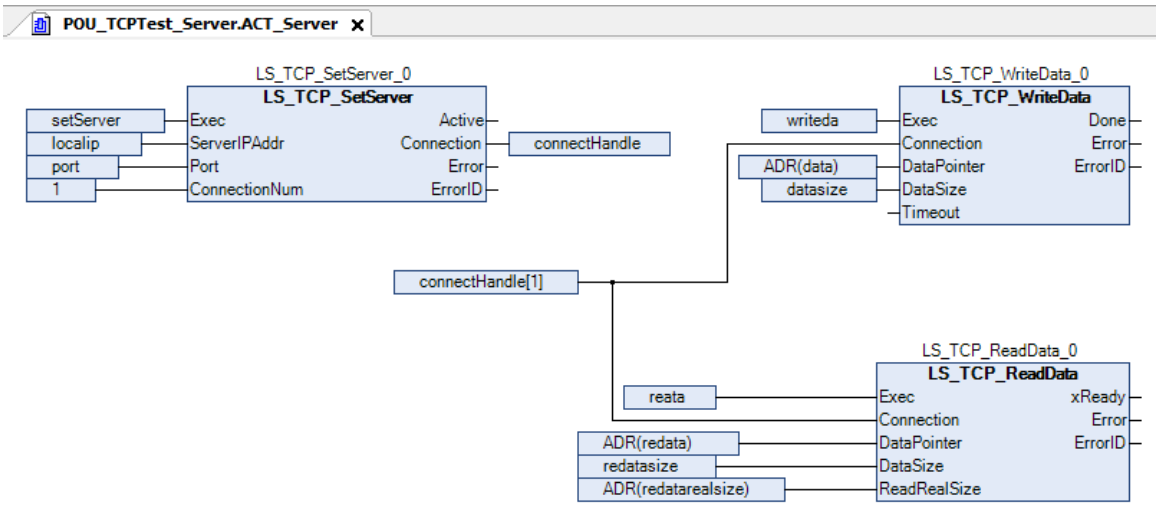


图4.65　新建动作ACT_Server

3)　在主程序中调用模块ACT_Server，并编写功能程序来实现收发数据的启动和停止，并统计实际发送和接收的字节总数，如图4.66所示；

POU_TCPTest_Server

```
PROGRAM POU_TCPTest_Server
VAR
    LS_TCP_WriteData_0: LS_TCP_WriteData;
    LS_TCP_ReadData_0: LS_TCP_ReadData;
    LS_TCP_SetServer_0: LS_TCP_SetServer;
    localip:STRING(32):='192.168.1.11';
    port:UINT:=5000;
    setServer:BOOL:=FALSE;
    writeda:BOOL:=FALSE;
    data:ARRAY[0..12] OF BYTE:=[0,1,2,3,4,5,6,7,8,9,10,11,12];
    datasize:UDINT:=13;
```

```
CASE istate OF
1://ini初始化
   setServer:=TRUE;
   senddataAll:=0;
   redatarealsize:=0;
   readdataAll := 0;
   istate:=10;
10: //等待连接成功
   IF NOT LS_TCP_SetServer_0.Active THEN
       reata:=FALSE;
    ELSE
       reata:=TRUE;//连接成功，启动读功能块
       istate:=2;
   END_IF
2://发送数据
   writeda:=TRUE;
   IF LS_TCP_WriteData_0.Done THEN
       writeda:=FALSE;
       istate:=3;
       senddataAll:=senddataAll+13;//统计发送数据的大小
   ELSIF LS_TCP_WriteData_0.Error THEN
      writeda:=FALSE;//发送数据报错则跳转
      istate := 100;
   END_IF
3:
   istate := 2;  //循环
```

图4.66　统计实际发送和接收的字节总数

4)　完成控制器程序编程，编译OK后连接控制器，将程序下载到控制器中，点击运行按钮或者按F5，运行程序，并将istate的值强制为1启动设置服务器连接模式，如图4.67所示；

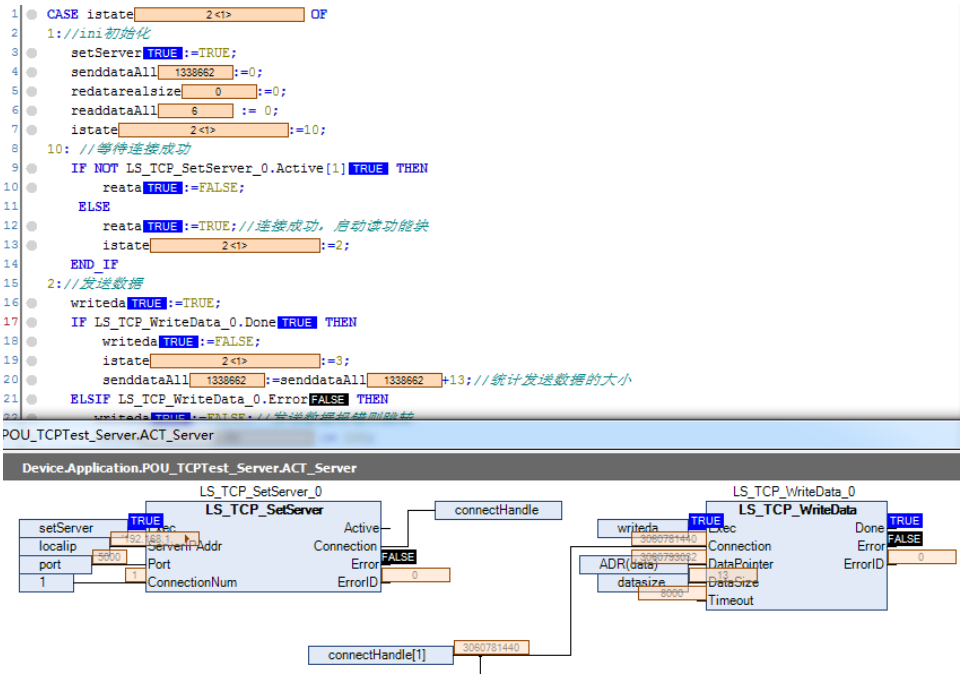


图4.67　设置服务器连接模式

5)　打开TCP&UDP测试工具，客户端模式新建连接，连接类型为TCP，目标IP为服务器IP(即控制器IP)，端口号5000，本机端口可随机也可指定，创建后连接，如图4.68所示。

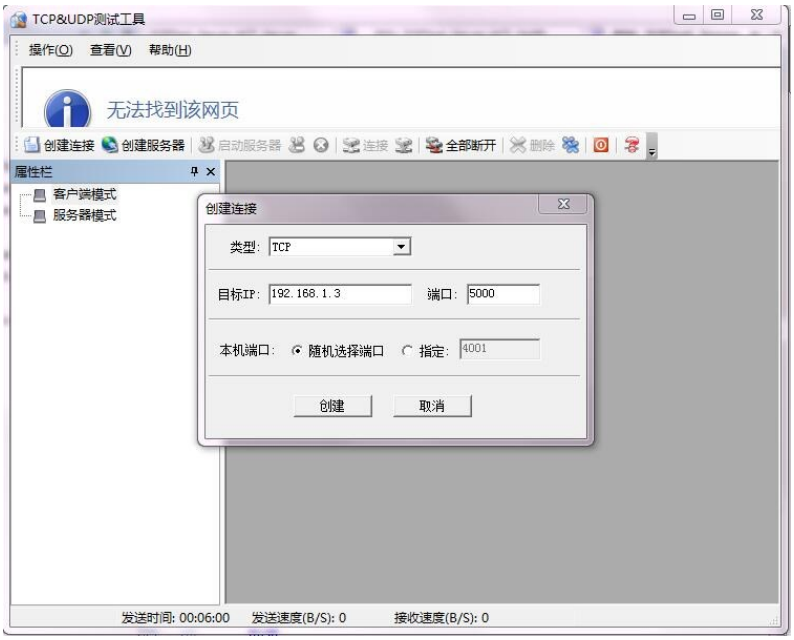


图4.68　TCP&UDP测试工具界面

6)　在TCP&UDP测试工具接收区可看到控制器发来的数据“00 01 02 03 04 05 06 07 08 09 0a 0b 0c”，发送区域输入“abc123”，发送间隔为100ms，选择自动发送，在控制器程序中可监测到接收的数据，结果如图4.69所示。

由图4.69可知，当控制器程序设定服务器模式连接时，TCPIP还未真正连接上，在PC端设定客户端连接后，整个TCPIP才连接成功，并且在TCP&UDP测试工具中可以观察到

控制器发送来的数据以及PC端收发数据的字节数；设定PC端发送数据，并选择自动发送，间隔100ms，在控制器程序中可监测到控制器收到的数据以及控制器端收发的字节数，如图4.70所示。当然PC端也可发送文件到控制器，这时控制器端需编写程序来解析数据。

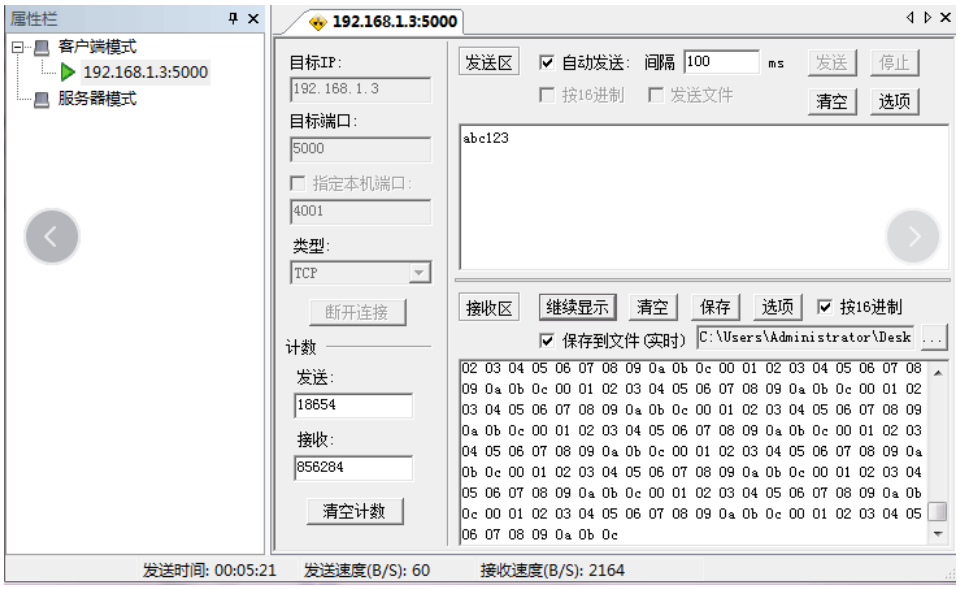


图4.69　监测接收的数据

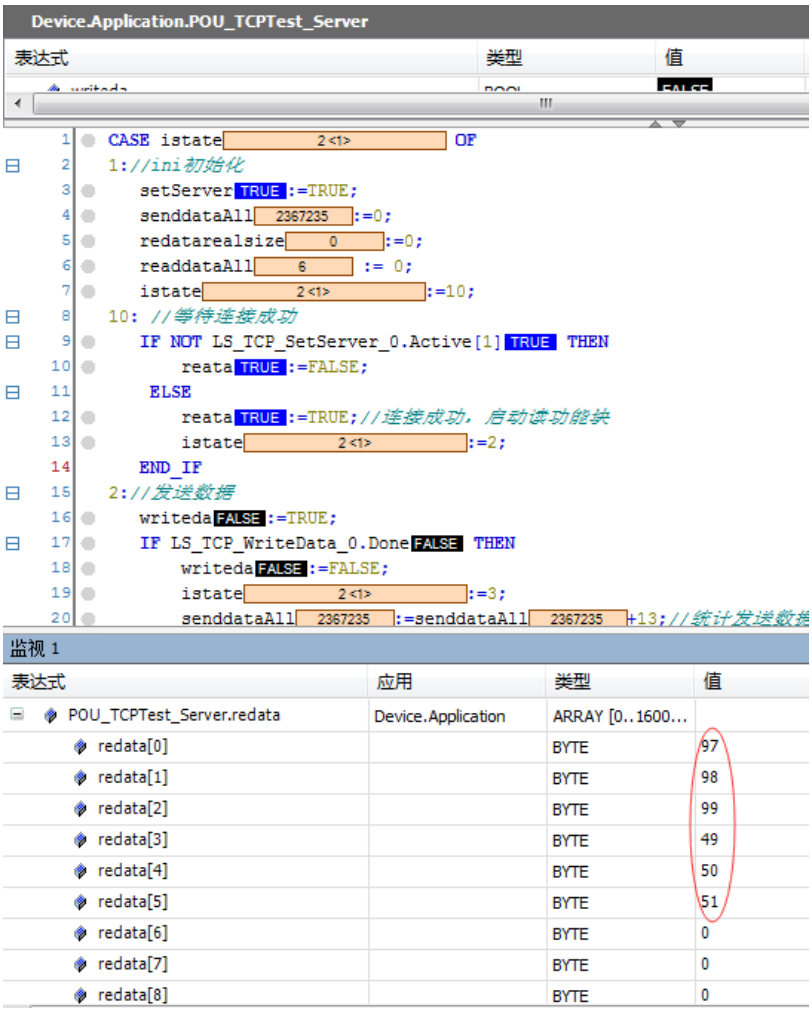


图4.70　解析数据程序

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**TCPIP-Server**”。

Client模式例程：编写程序，设置控制器工作在Client模式，PC为Server模式，向PC发送数据“1 2 3 4 5 6 7 8 9 10 11 12 13 14 15”，并接受PC发送给控制器的数据“abc123”。

Client模式例程和Server模式例程是一样的，只是在设置控制器工作模式不同而已，只需将设置服务器模式改成设置客户端模式即可，其他程序一样，如图4.71所示。

POU_TCPTest_Client

```
PROGRAM POU_TCPTest_Client
VAR
    LS_TCP_SetClient_0: LS_TCP_SetClient;
    LS_TCP_ReadData_0: LS_TCP_ReadData;
    LS_TCP_WriteData_0: LS_TCP_WriteData;

    writeda:BOOL:=FALSE;
    data:ARRAY[0..14] OF BYTE:=[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15];
```

```
CASE istate OF
1:  //ini初始化
   setClient:=TRUE;
   senddataAll:=0;
   redatarealsize:=0;
   readdataAll := 0;
   istate := 2;
2:
    TaskCycle:=TaskCycle+1;
    IF  LS_TCP_SetClient_0.Active THEN
        reata:=TRUE;
        istate := 3;
    ELSIF TaskCycle>10 THEN
        TaskCycle:=0;
        reata:=FALSE;
        setClient:=FALSE;
        istate := 1;
    END_IF
3:   //send data
   writeda:=TRUE;
   IF LS_TCP_WriteData_0.Done THEN
     writeda:=FALSE;
     senddataAll:=senddataAll+1024;
     istate :=4;
   ELSIF LS_TCP_WriteData_0.Error THEN
      writeda:=FALSE;
      reata:=FALSE;
      istate := 100;
   END_IF
```

图4.71　设置客户端模式

将程序下载到控制器中，点击运行按钮或者按F5，运行程序，并将istate的值强制为1启动设置客户端连接模式，如图4.72所示，启动设置客户端模式连接；

```
CASE istate 2 OF
1:  //ini初始化
   setClient TRUE :=TRUE;
   senddataAll 0 :=0;
   redatarealsize 0 :=0;
   readdataAll 0 := 0;
   istate 2 := 2;
2:
    TaskCycle 10 :=TaskCycle
    IF  LS_TCP_SetClient_0.Activ
        reata FALSE :=TRUE;
        istate 2 := 3;
    ELSIF TaskCycle 10 >10 THE
        TaskCycle 10 :=0;
        reata FALSE :=FALSE;
        setClient TRUE :=FALSE;
        istate 2 := 1;
    END_IF
3:   //send data
   writeda FALSE :=TRUE;
   IF LS_TCP_WriteData_0.Done FAL
     writeda FALSE :=FALSE;
     senddataAll 0 :=send
     istate 2 :=4;
   ELSIF LS_TCP_WriteData_0.Erro
      writeda FALSE :=FALSE;
      reata FALSE :=FALSE;
      istate 2 := 100;
   END_IF
4:
    istate 2 := 3;  //循环
    IF stopSend FALSE THEN
       stopSend FALSE :=FALSE;
       istate 2 := 120;  //退出
    END_IF
```

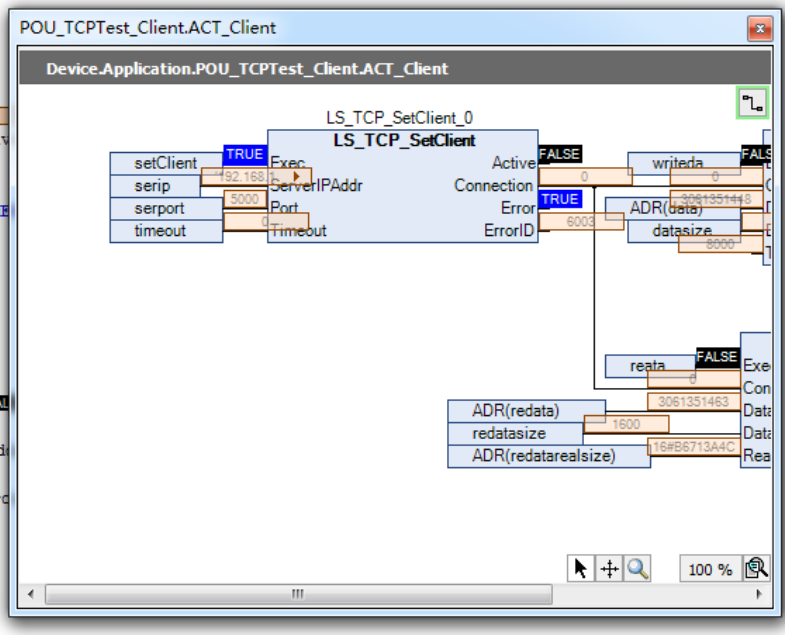


图4.72　设置客户端连接模式

打开TCP&UDP测试工具，服务器模式新建连接，本机端口号设定5000，创建后启动服务器，如图所示。在TCP&UDP测试工具接收区可看到控制器发来的数据“00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f”，在发送区域输入“123abc”，发送间隔为100ms，选择自动发送，在控制器程序中可监测到接收的数据，结果如图4.73所示；

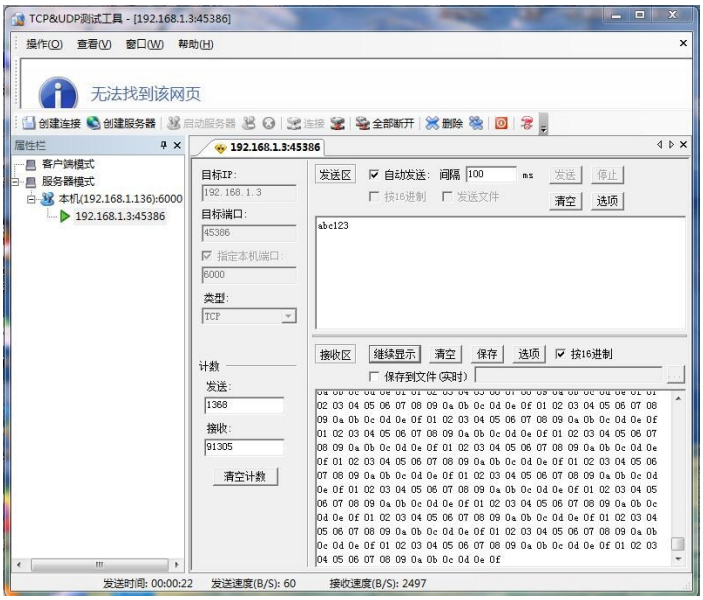


图4.73　启动服务器

POU_TCPTest_Client

Device.Application.POU_TCPTest_Client

```
1  CASE istate 2 OF
2  1:  //ini初始化
3     setClient TRUE :=TRUE;
4     senddataAll 0 :=0;
5     redatarealsize 0 :=0;
6     readdataAll 0 := 0;
7     istate 2 := 2;
8  2:
9      TaskCycle 9 :=TaskCycle 9 +1;
10     IF  LS_TCP_SetClient_0.Active FALSE THEN
11         reata FALSE :=TRUE;
12         istate 2 := 3;
13     ELSIF TaskCycle 9 >10 THEN
14         TaskCycle 9 :=0;
15         reata FALSE :=FALSE;
16         setClient TRUE :=FALSE;
17         istate 2 := 1;
18     END_IF
19 3:   //send data
20    writeda FALSE :=TRUE;
21    IF LS_TCP_WriteData_0.Done FALSE THEN
22      writeda FALSE :=FALSE;
23      senddataAll 0 :=senddataAll 0 +1024;
```

监视 1

表达式	应用	类型	值
POU_TCPTest_Client.redata	Device.Application	ARRAY [0..1600...	
redata[0]		BYTE	49
redata[1]		BYTE	50
redata[2]		BYTE	51
redata[3]		BYTE	97
redata[4]		BYTE	98
redata[5]		BYTE	99
redata[6]		BYTE	0
redata[7]		BYTE	0
redata[8]		BYTE	0

图4.74　控制器端收发数据(控制器Client模式)

当控制器程序设定Client模式连接时，TCPIP还未真正连接上，在PC端设定服务器启动后，整个TCPIP才连接成功，并且在TCP&UDP测试工具中可以观察到控制器发送来的数据以及PC端收发数据的字节数；设定PC端发送数据，并选择自动发送，间隔100ms，在控制器程序中可监测到控制器收到的数据以及控制器端收发的字节数。当然PC端也可发送文件到控制器，这时控制器端需编写程序来解析数据。

在设定控制器Client模式，PC为Server模式时，需要将PC的Windows防火墙关掉才能建立连接进行通讯。

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**TCPIP-Client**”。

UDP/IP协议通信例程

控制器Ethernet网口运行标准UDP/IP协议的对应操作相关指令包含在“PMC_Communication”库中。

UDP/IP协议相对TCP/IP协议来说是一种不可靠的协议，UDP是一个非连接的协议，传输数据之前源端和终端不建立连接，当它想传输时就简单地去抓取来自应用程序的数据；它不需要维护连接状态，包括收发状态等，因此一台服务机可同时向多个客户机传输相同的消息；对系统资源的要求少，程序结构相对较简单。

UDP/IP通讯使用方法：

1) 启动UDP/IP服务：调用UDP_SetConnect启动控制器UDP/IP服务；

2) 收发数据：在确认启动UDP/IP服务成功后，调用TCP_ReadData和TCP_WriteData指令读取控制器收到的数据并向外发送数据；

3) 根据程序功能实现启动连接、收发数据、中断通讯、统计收发数据的大小等。

UDP/IP例程：启动控制器UDP/IP服务，并向PC发送数据，并接受PC发送给控制器的数据。

1) 新建工程并命名UDPIP-Terminal，编程语言为结构化文本ST；

2) 在设备栏右击主程序“UDP_Terminal”，新建一个动作ACT_Connection，编程语言为连续功能图(CFC)，负责启动控制器UDP/IP服务(设定控制器IP为192.168.1.3，端口号为5000）以及收发数据主体(其中发送数据指令中目标IP和端口为PC机的IP以及测试工具设定的端口号)，并添加对应的参数，如图4.75所示；

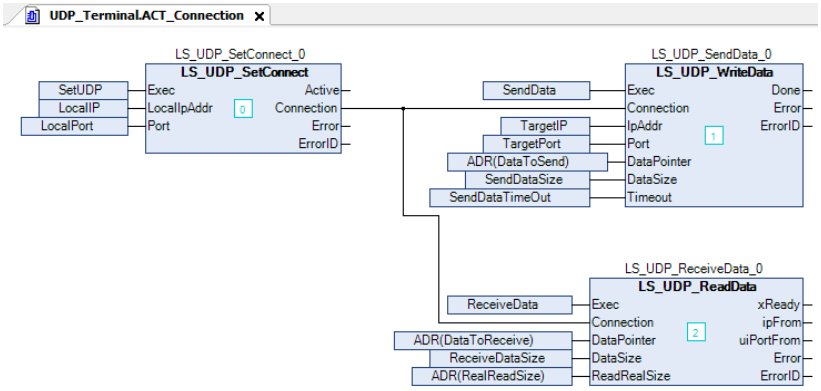


图4.75　启动控制器UDPIP服务主体程序

3) 在主程序中调用模块UDP_Terminal，并编写功能程序来实现收发数据的启动和停止，并统计实际发送和接收的字节总数；

4) 完成控制器程序编程，编译OK后连接控制器，将程序下载到控制器中，点击运行按钮或者按F5，运行程序，并将iState的值强制为1启动控制器UDPIP服务，如图4.76所示；

启动控制器UDP服务后，不管外部连接是否成功便开始收发数据。

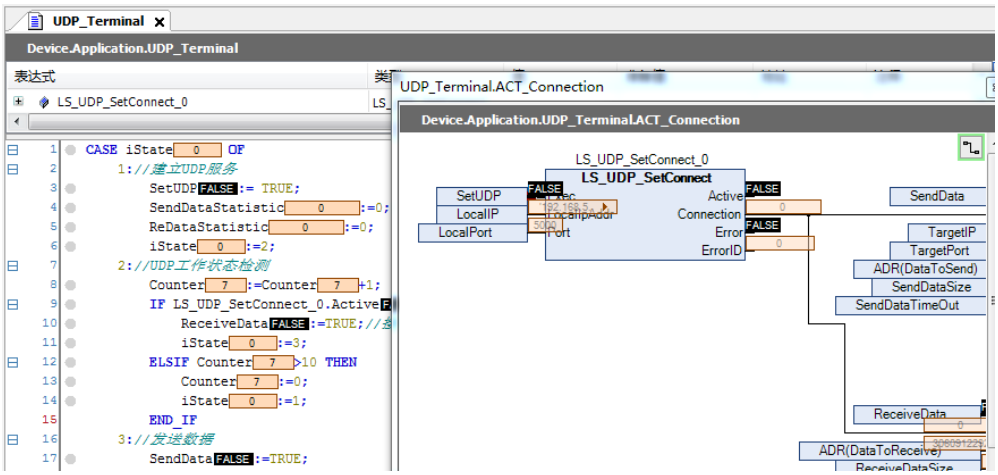


图4.76　启动控制器UDPIP服务整体程序

5)　打开TCP&UDP测试工具，客户端模式新建连接，连接类型为UDP，目标IP为控制器IP，端口号5000，本机端口指定为5010(需要和控制器发送数据指令的目标端口对应)，创建后连接，如图4.77所示。

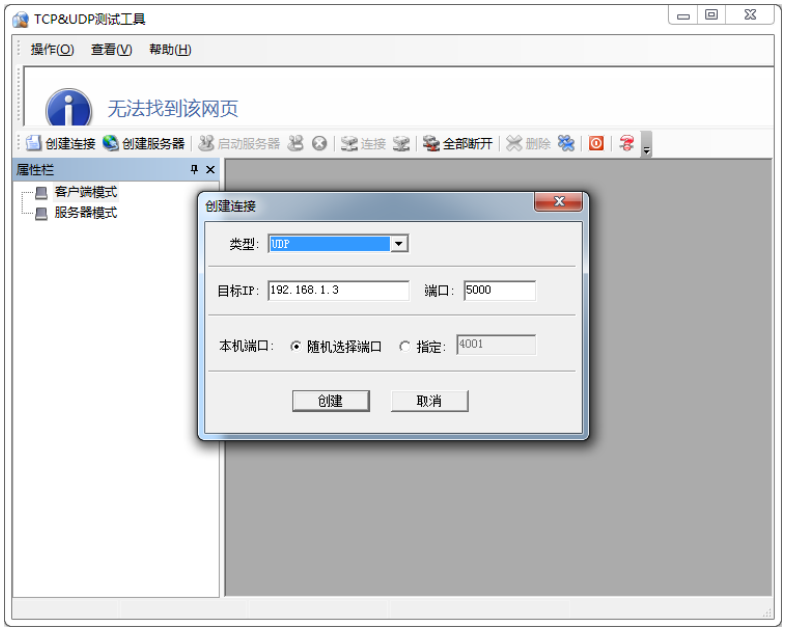


图4.77　PC端启动UDPIP服务

6)　在TCP&UDP测试工具接收区可看到控制器发来的数据“65 f5 7d f2 e1 4f 08 07 aa 0b”，发送区域输入“abc123”，发送间隔为100ms，选择自动发送，在控制器程序中可监测到接收的数据，结果如图4.78、4.79所示。

需要注意的是，在控制器和PC进行UDP通讯时，需要将PC的Windows防火墙关掉才能正常通讯。

本例中只是一对一的收发数据通讯，若要实现UDP广播式，则只将服务机程序中发送数据指令的目标IP改成255.255.255.0即可实现广播式发送数据。

本例程原代码参见PMC600软件资料中的“例程”文件夹中的“**TCPIP- Teminal**”。

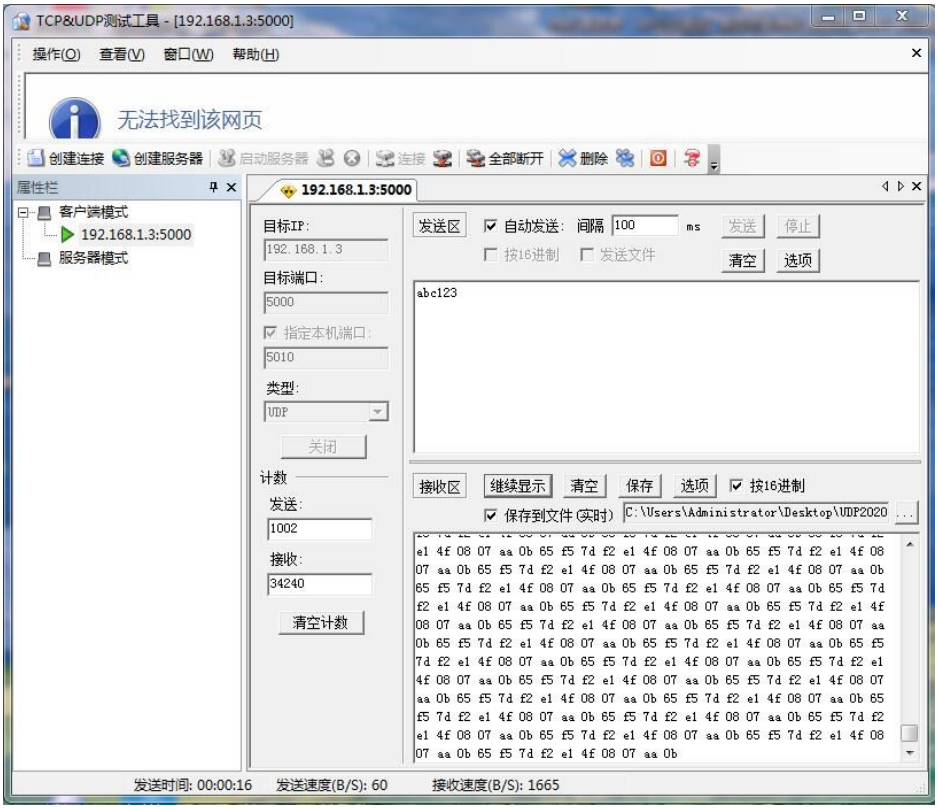


图4.78　PC端接收来自控制器的数据

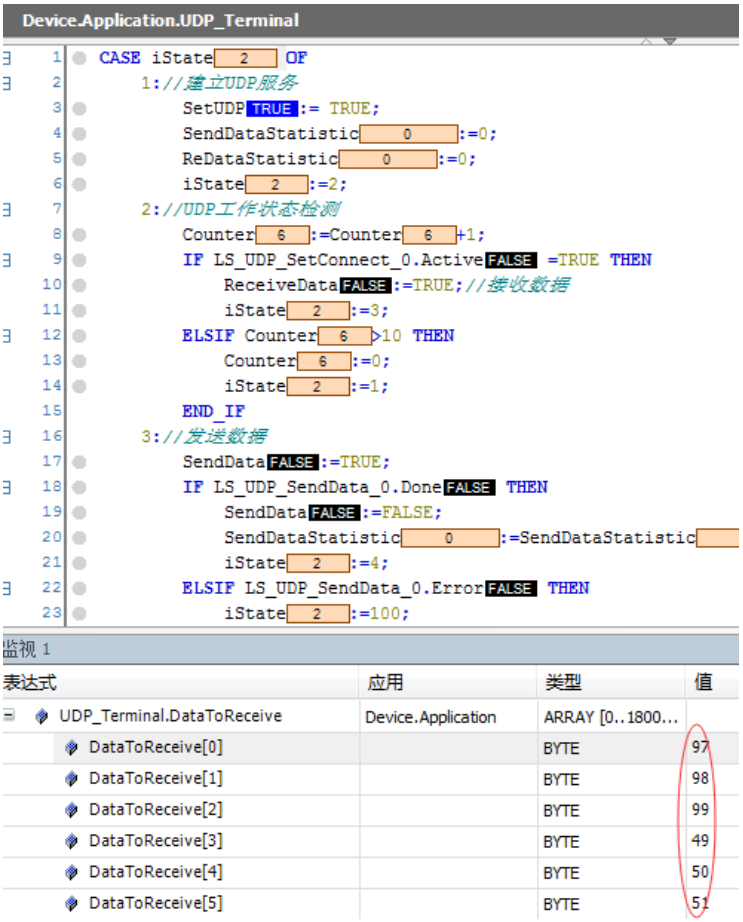


图4.79　控制器端接收来自PC端的数据

深圳市雷赛控制技术有限公司
SHENZHEN LEADSHINE CONTROL TECHNOLOGY CO.,LTD

深圳市雷赛控制技术有限公司
地　址：深圳市深圳市南山区学苑大道1001号南山智园A3栋9楼
邮　编：518052
电　话：0755-26415968
传　真：0755-26417609
Email：info@szleadtech.com.cn
网　址：http：//www.szleadtech.com.cn