

**综述**            本章给出了本书中 M218 编程语言的介绍

**本章内容**        本章包含一下章节内容：

| 章节  | 章节内容      | 页码 |
|-----|-----------|----|
| 2.1 | 梯形图—LD 语言 |    |
| 2.2 | ST 语言     |    |
| 2.3 | FBD 功能块语言 |    |
| 2.4 | CFC 连续功能图 |    |
| 2.5 | SFC 顺序功能图 |    |

---

# 梯形图—LD

## 综述            本章节描述梯形图语言的编程方法

梯形图是用得最多的 PLC 编程语言，它与继电器控制系统的电路图相似，直观易懂，易熟悉继电器控制电路的电气人员掌握，适用于开关量逻辑控制。梯形图由触点、线圈和用方框表示的功能块组成。触点代表逻辑输入条件，如开关、按钮和内部条件等；线圈通常表示逻辑运算输出结果，用来控制外部的指示灯、接触器和内部的输出条件等；功能块用来表示定时器、计数器或者数学运算等特殊指令。

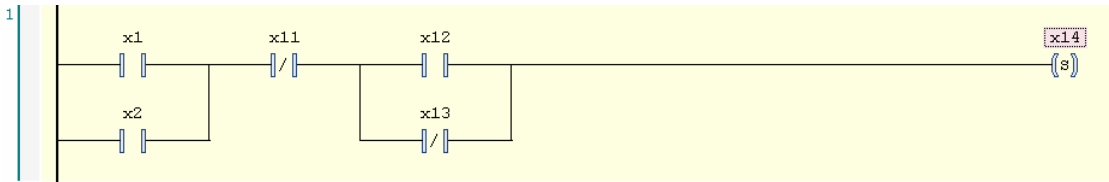
在分析梯形图中的逻辑关系时，可以想象两条垂直母线之间有从左向右流动的直流电。

## somachine 梯形图菜单

梯形图不但很适用于逻辑的转换，并且它也能创建类似于 FBD 中的节，所以用梯形图调用程序组织单元也是很方便的。在 somachine 软件中，当使用梯形图，可在 FBD/LD/IL 菜单选择。如下图：

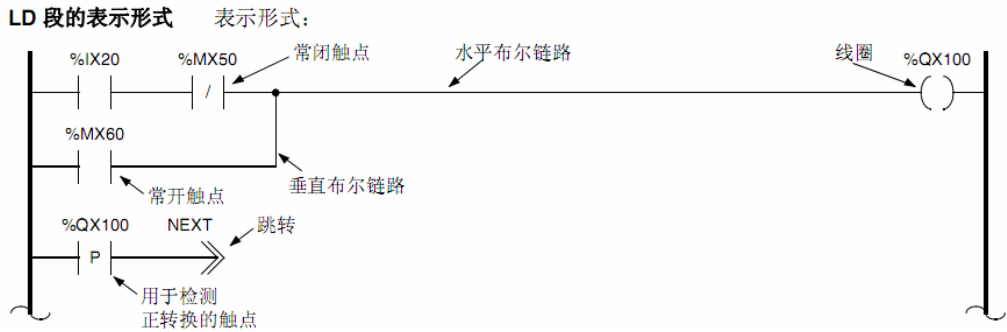


梯形图包含了一系列的**节**，左右两边各有一个垂直的电流线（能量线）限制其范围，在中间是由**触点**、**线圈**、连接线组成的电路图。如下图



每一个节的左边有一系列触点，这些触点根据布尔变量值的 TRUE 和 FALSE 来传递从左到右的开和关的状态。每一个触点是一个布尔变量，如变量值为 TRUE，通过连接线从左到右传递状态。否则传递“关”的状态。在节最右边的线圈，根据左边的状态获得一个开或关的值，并相应地赋给一个布尔变量真或假值。

### 梯形图 LD 代码段的演示



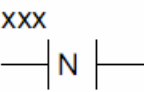
### 梯形图 LD 的各种编程元素

#### 触点

- 触点简介：触点是 LD 元素，可把水平链路状态传输到其右侧。此状态是对左侧的水平链路的状态与相关的布尔型实际参数的状态进行布尔 AND 运算的结果。触点并不更改相关实际参数的值，触点占用一个单元格。  
以下参数可作为实际参数：  
布尔变量；布尔常量；布尔地址（拓扑地址或符号地址）。

- 触点类型

| 名称        | 演示         | 描述  |
|-----------|------------|---|
| 常开        | <p>xxx</p> | 在常开触点的情况下,如果相关的布尔型实际参数（用 xxx 表示）的状态为 ON，那么左侧链路的状态会被传输到右侧链路。否则，右侧链路状态为 OFF。  |
| 常闭        | <p>xxx</p> | 在常闭触点的情况下,如果相关的布尔型实际参数（用 xxx 表示）的状态为 OFF，那么左侧链路的状态会被传输到右侧链路。否则，右侧链路状态为 OFF。 |
| 用来检测上升沿触点 | <p>xxx</p> | 使用可检测正转换的触点,当相关的实际参数（标记为 xxx）从 OFF 转换为 ON 且左侧链路的状态为 ON 时,程序循环的右侧链路的状态       |

|           |   |   |
|-----------|---|---|
|           |   | 态为 ON。否则，右侧链路状态为 0。   |
| 用来检测下降沿触点 |  | 使用可检测负转换的触点,当相关的实际参数（标记为 xxx）从 ON 转换为 OFF 且左侧链路的状态为 ON 时,程序循环的右侧链路的状态为 ON。否则，右侧链路状态为 0。 |

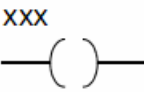
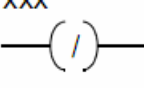
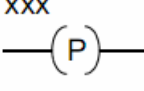
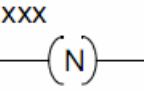
## 线圈

- 线圈简介：线圈是一个 LD 元素，它将左侧的水平链路的状态传输到右侧的水平链路，状态保持不变。此状态存储在相应的布尔类型的实际参数中。通常情况下，线圈在触点或 FFB 之后，但线圈后面还可以有触点。线圈占用一个单元格。

以下参数可作为实际参数：

布尔变量；布尔地址（拓扑地址或符号地址）。

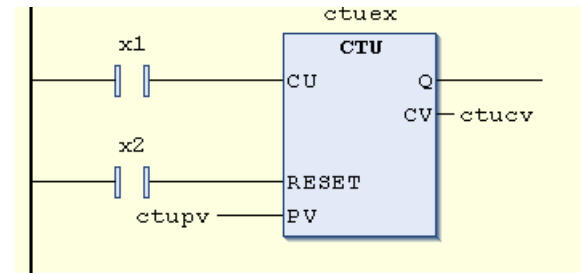
- 线圈类型

| 名称         | 表示形式  | 描述  |
|------------|---|---|
| 线圈         |  | 通过线圈,左侧链路的状态将传输到相应的布尔型实际参数（用 xxx 表示）以及右侧链路中。  |
| 反向线圈       |  | 通过反向线圈,左侧链路的状态将复制到右侧链路中。左侧链路的相反状态将复制到相应的布尔型实际参数（用 xxx 表示）中。如果左侧链路为 OFF，那么右侧链路也将为 OFF，相关的布尔型实际参数将为 ON。 |
| 用于检测正转换的线圈 |  | 使用可检测正转换的线圈,左侧链路的状态将复制到右侧链路。如果左侧链路的状态从 0 转换为 1,则程序循环中数据类型为 EBOOL 的相关实际参数（用 xxx 表示）为 1。                |
| 用于检测负转换的线圈 |  | 使用可检测负转换的线圈,左侧链路的状态将复制到右侧链路。如果左侧链路的状态从 1 转换为 0,则程序循环中相关的布尔型实际参数（用 xxx 表示）为 1。                         |

|      |   |   |
|------|---|---|
| 置位线圈 | <div> <div>xxx</div> <div> <div>(S)</div> </div> </div> | 使用置位线圈,左侧链路的状态将复制到右侧链路。如果左侧链路的状态为 ON, 则相关的布尔型实际参数 (用 xxx 表示) 被设定为 ON, 否则, 实际参数保持不变。使用复位线圈可将相应的布尔类型实际参数复位。   |
| 复位线圈 | <div> <div>xxx</div> <div> <div>(R)</div> </div> </div> | 使用复位线圈,左侧链路的状态将复制到右侧链路中。如果左侧链路的状态为 ON, 则相关的布尔型实际参数 (用 xxx 表示) 被设定为 OFF, 否则, 实际参数保持不变。通过置位线圈可以置位相应的布尔类型实际参数。 |

# 指令块

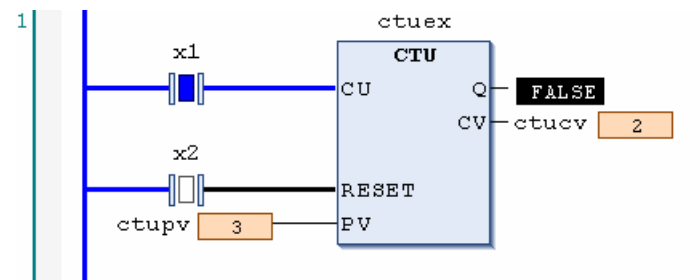
基本指令块具有内部状态。每次调用该功能时,即使输入值相同,输出值也可能不同,例如对于计数器,输出值是递增的。



在图形表示中,基本指令块用包含多个输入和多个输出的块结构表示。输入始终表示在块结构的左侧,而输出始终表示在块结构的右侧。指令块的名称(即功能块类型)显示在块结构的中央。实例名称显示在块结构的上方。功能块的快捷方式如下图所示。



在图形表示中,基本指令块用包含多个输入和一个输出的块结构表示。输入始终表示在块结构的左侧,而输出始终表示在块结构的右侧。功能的名称(即功能类型)显示在块结构的中央,即 ctuex 是 CTU 块的名称。



## ST 表达式

### 综述 本章节描述 ST 语言的编程方法

表达式是计算之后获得返回值的结构。在[指令](#)中需要使用该返回值。表达式由[操作符](#)、[操作数](#)、[赋值](#)组成。操作数可以是常量、变量、函数调用的返回值或其它表达式。

举例：

|              |          |
|--------------|----------|
| 33           | (*常量*)   |
| ivar         | (*变量*)   |
| fct(a,b,c)   | (*函数调用*) |
| a AND b      | (*表达式*)  |
| (x*y) / z    | (*表达式*)  |
| real_var2 := | (*赋值*)   |
| int_var;     |          |

## ST 编程语言的各种元素

使用结构化文本（ST）的编程语言，可以执行多种操作，例如调用功能块、和赋值、有条件地执行指令和重复任务。（ST）的编程语言由各种元素组成，具体如下。

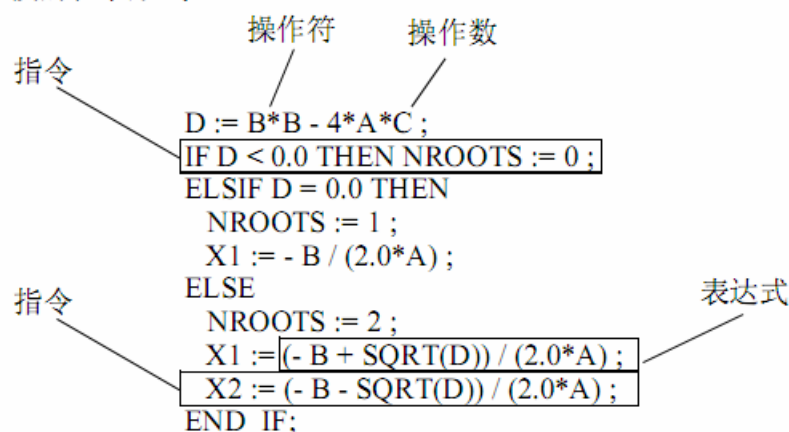
表达式：ST 编程语言使用" 表达式"。表达式是由操作符和操作数组成的结构，在执行表达式时会返回值。

操作数：操作数表示变量，数值，地址，功能块等。

操作符：操作符是执行运算过程中所用的符号。

指令：指令用于将表达式返回的值赋给实际参数，并构造和控制表达式。

ST 段的表示形式：



## 表达式

计算表达式时将根据操作符的优先级所定义的顺序将操作符应用于操作数表。首先执行表达式中具有最高优先级的操作符，接着执行具有次优先级的操作符，依此类推，直到完成整个计算过程。优先级相同的操作符将根据它们在表达式中的书写顺序从左至右执行。可使用括号更改此顺序。

例如，如果 A、B、C 和 D 的值分别为 1、2、3 和 4，并按以下方式计算：A+B-C\*D，结果则为 -9。(A+B-C)\*D，结果则为 0。

如果操作符包含两个操作数，则先执行左边的操作数，例如在表达式 SIN(A)\*COS(B) 中，先计算表达式 SIN(A)，后计算 COS(B)，然后计算它们的乘积。

## 操作数

操作数可以是：地址，数值，变量，多元素变量，多元素变量的元素，功能调用，功能块输出等。处理操作数的指令中的数据类型必须相同。如果需要处理不同类型的操作数，则必须预先执行类型转换。

在下面的示例中，整数变量 i1 在添加到实数变量 r4 中之前会先转换为实数变量。

```
r3 := r4 + SIN(INT_TO_REAL(i1)) ;
```

使用其他数据类型，要将其他数据类型指定为地址的缺省数据类型，必须通过显式声明进行。使用变量编辑器可方便地完成变量的声明。地址的数据类型不能在 ST 段中直接声明（例如，不允许声明 AT %MW1:UINT）。

例如，在变量编辑器中声明以下变量：

```
UnlocV1: ARRAY [1..10] OF INT;
```

```
LocV1:  ARRAY [1..10] OF INT AT %MW100;
```

```
LocV2:  TIME AT %MW100;
```

下面的调用具有正确的语法：

```
%MW200 := 5;
```

```
UnlocV1[2] := LocV1[%MW200];
```

```
LocV2      := t#3s;
```

## 操作符

操作符是一种符号，它表示：

要执行的算术运算，或要执行的逻辑运算，功能编辑调用。

操作符是泛型的，即它们自动适应操作数的数据类型。

ST 语言的操作符

| 操作符                         | 含义           | 优先级   | 适用的操作数   | 描述  |
|-----------------------------|--------------|-------|--|---|
| ( )                         | 使用括号：        | 1（最高） | 表达式  | 括号用于改变操作符的执行顺序。示例：如果操作数 A、B、C 和 D 的值分别为 1、2、3 和 4，A+B-C*D 的结果则为 -9，而 (A+B-C)*D 的结果则为 0。 |
| FUNCNAME<br>(实际参数<br>-list) | 功能处理<br>(调用) | 2     | 表达式、数值、变量、地址（所有数据类型）                               | 功能处理用于执行功能（请参见 <a href="#">调用基本功能</a> ）。  |
| -                           | 取反           | 3     | 数据类型为 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 | 取反（-）时，操作数值的符号会反转。示例：本示例中，如果 IN1 为 4，则  |

|     |    |   |   |   |
|-----|----|---|---|---|
|     |    |   | <a href="#">UINT</a> 、 <a href="#">UDINT</a> 或 <a href="#">REAL</a> 的表达式、数值、变量或地址   | OUT 为 -4。<br>OUT := - IN1 ;   |
| NOT | 反码 | 3 | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">WORD</a> 或 <a href="#">DWORD</a> 的表达式、数值、变量或地址  | 进行 NOT 运算时，操作数将逐位反转。<br>示例：本示例中，如果 IN1 为 1100110011，则 OUT 为 0011001100。<br>OUT := NOT IN1 ;   |
| **  | 幂  | 4 | 数据类型为 <a href="#">REAL</a> （底数）和 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 或 <a href="#">REAL</a> （指数）的表达式、数值、变量或地址 | 求幂（**）运算时，将以第一个操作数为底数，第二个操作数为指数进行求幂。<br>示例：该示例中，如果 IN1 为 5.0，IN2 为 4.0，则 OUT 为 625.0。<br>OUT := IN1 ** IN2 ;                                      |
| *   | 乘法 | 5 | 数据类型为 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 或 <a href="#">REAL</a> 的表达式、数值、变量或地址                                | 乘法（*）运算时，将用第一个操作数的值乘以第二个操作数（指数）的值。<br>示例：该示例中，如果 IN1 为 5.0，IN2 为 4.0，则 OUT 为 20.0。<br>OUT := IN1 * IN2 ;<br>注：先期库中的 MULTIME 函数可用于涉及数据类型 Time 的乘法。 |
| /   | 除法 | 5 | 数据类型为 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 或 <a href="#">REAL</a> 的表达式、数值、变量或地址                                | 除法（/）运算时，将用第一个操作数的值除以第二个操作数的值。<br>示例：该示例中，如果 IN1 为 20.0，IN2 为 5.0，则 OUT 为 4.0。<br>OUT := IN1 / IN2 ;<br>注：先期库中的 DIVTIME 函数可用于涉及数据类型 Time 的除法。     |
| MOD | 模数 | 5 | 数据类型为 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 或 <a href="#">UDINT</a> 的   | 执行 MOD 时，将用第一个操作数的值除以第二个操作数的值，除法的余数（模数）显示为结果。<br>示例：本示例中  |



|   |      |   |   |  |
|---|------|---|---|--|
|   |      |   | 表达式、数值、变量或地址  | <ul style="list-style-type: none"> <li>• 如果 IN1 为 7, IN2 为 2, 则 OUT 为 1。</li> <li>• 如果 IN1 为 7, IN2 为 -2, 则 OUT 为 1。</li> <li>• 如果 IN1 为 -7, IN2 为 2, 则 OUT 为 -1。</li> <li>• 如果 IN1 为 -7, IN2 为 -2, 则 OUT 为 -1。</li> </ul> <p>OUT := IN1 MOD IN2 ;</p> |
| + | 加法   | 6 | 数据类型为 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 、 <a href="#">REAL</a> 或 <a href="#">TIME</a> 的表达式、数值、变量或地址   | <p>加法 (+) 运算时, 将用第一个操作数的值加上第二个操作数的值。</p> <p>示例: 本示例中如果 IN1 为 7, IN2 为 2, 则 OUT 为 9</p> <p>OUT := IN1 + IN2 ;</p>   |
| - | 减法   | 6 | 数据类型为 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 、 <a href="#">REAL</a> 或 <a href="#">TIME</a> 的表达式、数值、变量或地址   | <p>减法 (-) 运算时, 将用第一个操作数的值减去第二个操作数的值。</p> <p>示例: 该示例中, 如果 IN1 为 10, IN2 为 4, 则 OUT 为 6。</p> <p>OUT := IN1 - IN2 ;</p>   |
| < | 小于比较 | 7 | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 、 <a href="#">REAL</a> 、 <a href="#">TIME</a> 、 <a href="#">WORD</a> 、 <a href="#">DWORD</a> 、 <a href="#">STRING</a> 、 <a href="#">DT</a> 、 <a href="#">DATE</a> 或 <a href="#">TOD</a> 的表达式、数值、变量或地址 | <p>使用 &lt; 将第一个操作数的值与第二个操作数的值进行比较。如果第一个操作数的值小于第二个操作数的值, 则结果为布尔 1。如果第一个操作数的值大于或等于第二个操作数的值, 则结果为布尔 0。</p> <p>示例: 本示例中, 如果 IN1 小于 10, 则 OUT 为 1, 否则为 0。</p> <p>OUT := IN1 &lt; 10 ;</p>   |

|    |         |   |  |   |
|----|---------|---|--|---|
| >  | 大于比较    | 7 | 数据类型为<br><a href="#">BOOL</a> 、<br><a href="#">BYTE</a> 、<br><a href="#">INT</a> 、 <a href="#">DINT</a> 、<br><a href="#">UINT</a> 、<br><a href="#">UDINT</a> 、<br><a href="#">REAL</a> 、<br><a href="#">TIME</a> 、<br><a href="#">WORD</a> 、<br><a href="#">DWORD</a> 、<br><a href="#">STRING</a> 、<br><a href="#">DT</a> 、 <a href="#">DATE</a><br>或 <a href="#">TOD</a><br>的表达式、<br>数值、变量<br>或地址 | 使用 > 将第一个操作数的值与第二个操作数的值进行比较。如果第一个操作数的值大于第二个操作数的值，则结果为布尔 1。如果第一个操作数的值小于或等于第二个操作数的值，则结果为布尔 0。<br>示例：本示例中，如果 IN1 大于 10，则 OUT 为 1，如果 IN1 小于 10 则为 0。<br>OUT := IN1 > 10 ; |
| <= | 小于或等于比较 | 7 | 数据类型为<br><a href="#">BOOL</a> 、<br><a href="#">BYTE</a> 、<br><a href="#">INT</a> 、 <a href="#">DINT</a> 、<br><a href="#">UINT</a> 、<br><a href="#">UDINT</a> 、<br><a href="#">REAL</a> 、<br><a href="#">TIME</a> 、<br><a href="#">WORD</a> 、<br><a href="#">DWORD</a> 、<br><a href="#">STRING</a> 、<br><a href="#">DT</a> 、 <a href="#">DATE</a><br>或 <a href="#">TOD</a><br>的表达式、<br>数值、变量<br>或地址 | 使用 <= 将第一个操作数的值与第二个操作数的值进行比较。如果第一个操作数的值小于或等于第二个操作数的值，则结果为布尔 1。如果第一个操作数的值大于第二个操作数的值，则结果为布尔 0。<br>示例：本示例中，如果 IN1 小于或等于 10，则 OUT 为 1，否则为 0。<br>OUT := IN1 <= 10 ;        |
| >= | 大于或等于比较 | 7 | 数据类型为<br><a href="#">BOOL</a> 、<br><a href="#">BYTE</a> 、<br><a href="#">INT</a> 、 <a href="#">DINT</a> 、<br><a href="#">UINT</a> 、<br><a href="#">UDINT</a> 、<br><a href="#">REAL</a> 、<br><a href="#">TIME</a> 、<br><a href="#">WORD</a> 、<br><a href="#">DWORD</a> 、<br><a href="#">STRING</a> 、<br><a href="#">DT</a> 、 <a href="#">DATE</a>   | 使用 >= 将第一个操作数的值与第二个操作数的值进行比较。如果第一个操作数的值大于或等于第二个操作数的值，则结果为布尔 1。如果第一个操作数的值小于第二个操作数的值，则结果为布尔 0。<br>示例：本示例中，如果 IN1 大于或等于 10，则 OUT 为 1，否则为 0。<br>OUT := IN1 >= 10 ;        |

|    |     |   |   |  |
|----|-----|---|---|--|
|    |     |   | 或 <a href="#">TOD</a> 的表达式、数值、变量或地址   |  |
| =  | 等于  | 8 | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 、 <a href="#">REAL</a> 、 <a href="#">TIME</a> 、 <a href="#">WORD</a> 、 <a href="#">DWORD</a> 、 <a href="#">STRING</a> 、 <a href="#">DT</a> 、 <a href="#">DATE</a> 或 <a href="#">TOD</a> 的表达式、数值、变量或地址 | 使用 = 将第一个操作数的值与第二个操作数的值进行比较。如果第一个操作数的值等于第二个操作数的值，则结果为布尔 1。如果第一个操作数的值不等于第二个操作数的值，则结果为布尔 0。示例：本示例中，如果 IN1 等于 10，则 OUT 为 1，否则为 0。<br>OUT := IN1 = 10 ;    |
| <> | 不等于 | 8 | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">INT</a> 、 <a href="#">DINT</a> 、 <a href="#">UINT</a> 、 <a href="#">UDINT</a> 、 <a href="#">REAL</a> 、 <a href="#">TIME</a> 、 <a href="#">WORD</a> 、 <a href="#">DWORD</a> 、 <a href="#">STRING</a> 、 <a href="#">DT</a> 、 <a href="#">DATE</a> 或 <a href="#">TOD</a> 的表达式、数值、变量或地址 | 使用 <> 将第一个操作数的值与第二个操作数的值进行比较。如果第一个操作数的值不等于第二个操作数的值，则结果为布尔 1。如果第一个操作数的值等于第二个操作数的值，则结果为布尔 0。示例：本示例中，如果 IN1 不等于 10，则 OUT 为 1，否则为 0。<br>OUT := IN1 <> 10 ; |
| &  | 逻辑与 | 9 | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">WORD</a> 或 <a href="#">DWORD</a> 的表达式、数值、变量或地址  | 对于 &，操作数之间存在逻辑与关联。对于 BYTE、WORD 和 DWORD 数据类型，此关联是逐位进行的。示例：本示例中，如果 IN1、IN2 和 IN3 均为 1，则 OUT 为 1。<br>OUT := IN1 & IN2 & IN3 ;                             |

|     |      |         |  |  |
|-----|------|---------|--|--|
| AND | 逻辑与  | 9       | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">WORD</a> 或 <a href="#">DWORD</a> 的表达式、数值、变量或地址 | 对于 AND，操作数之间存在逻辑与关联。对于 BYTE、WORD 和 DWORD 数据类型，此关联是逐位进行的。<br>示例：本示例中，如果 IN1、IN2 和 IN3 均为 1，则 OUT 为 1。<br>OUT := IN1 AND IN2 AND IN3 ；   |
| XOR | 逻辑异或 | 10      | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">WORD</a> 或 <a href="#">DWORD</a> 的表达式、数值、变量或地址 | 对于 XOR，操作数之间存在逻辑异或关联。对于 BYTE、WORD 和 DWORD 数据类型，此关联是逐位进行的。<br>示例：本示例中，如果 IN1 和 IN2 不相等，则 OUT 为 1。如果 A 和 B 的状态相同(均为 0 或均为 1)，则 D 为 0。<br>OUT := IN1 XOR IN2 ；<br>如果将两个以上的操作数进行关联，当状态为 1 的操作数个数不是偶数时结果为 1，而当状态为 1 的操作数个数是偶数时结果为 0。<br>示例：本示例中，如果有 1 个或 3 个操作数为 1，则 OUT 为 1，如果有 0、2 或 4 个操作数为 1，则 OUT 为 0。<br>OUT := IN1 XOR IN2 XOR IN3 XOR IN4 ； |
| OR  | 逻辑或  | 11 (最低) | 数据类型为 <a href="#">BOOL</a> 、 <a href="#">BYTE</a> 、 <a href="#">WORD</a> 或 <a href="#">DWORD</a> 的表达式、数值、变量或地址 | 对于 OR，操作数之间存在逻辑或关联。对于 BYTE、WORD 和 DWORD 数据类型，此关联是逐位进行的。<br>示例：本示例中，如果 IN1、IN2 或 IN3 为 1，则 OUT 为 1。<br>OUT := IN1 OR IN2 OR IN3 ；   |

## ST 指令

- 赋值指令

描述：执行赋值时，单元素或多元素变量的当前值会替换为表达式的计算结果。赋值表达式的结构为：左边是变量名称，之后是赋值操作符 :=，然后是要求值的表达式。两个变量（分别位于赋值操作符的左侧和右侧）的数据类型必须相同。数组是个特例。显式启用后，也可对长度不同的两个数组执行赋值操作。将一个变量的值赋给另一个变量赋值用于将一个变量的值赋给另一个变量。

例如，指令

A := B ；

用于将变量 A 的值替换为变量 B 的当前值。如果 A 和 B 是基本数据类型，则 B

的单个值会传递给 A。如果 A 和 B 是导出的数据类型，则所有 B 元素的值都传递给 A。

将数值赋给变量

赋值用于将数值赋给变量。

指令

C := 25 ;

用于将值 25 赋给变量 C。

将运算值赋给变量

赋值用于将运算结果赋给变量。

例如，指令

X := (A+B-C)\*D ;

用于将 (A+B-C)\*D 的运算结果赋给变量 X。

将 FFB 的值赋给变量

赋值用于将功能或功能块返回的值赋给变量。

例如，指令

B := MOD(C,A) ;

用于调用 MOD（模数）功能并将计算结果赋给变量 B。

例如，指令

A := MY\_TON.Q ;

用于将 MY\_TON 功能块（TON 功能块的实例）的 Q 输出值赋给变量 A。（这不是功能块调用）

- 选择指令 IF...THEN...END\_IF

描述：IF 指令只有确定其相关布尔表达式的值为 1（真）时，才会执行指令或一组指令。如果条件为 0（假），将不会执行该指令或指令组。THEN 指令标识条件的结尾和指令的开头。END\_IF 指令标记指令的结尾。

注意：可以嵌套任何数量的 IF...THEN...END\_IF 指令，以生成复杂的选择指令。

示例 IF...THEN...END\_IF

该条件可以使用布尔变量表达。

如果 FLAG 为 1，将执行指令；如果 FLAG 为 0，则不会执行。

IF FLAG THEN

C:=SIN(A) \* COS(B) ;

B:=C - A ;

END\_IF ;

该条件可使用返回布尔结果的操作表达。如果 A 大于 B，将会执行指令；如果 A 小于或等于 B，则不会执行。

示例 IF NOT...THEN...END\_IF

该条件可使用 NOT 反转（为 0 时执行这两个指令）。

IF NOT FLAG THEN

C:=SIN\_REAL(A) \* COS\_REAL(B) ;

B:=C - A ;

END\_IF ;

- 选择指令 ELSE

描述： ELSE 指令始终出现在 IF...THEN、ELSIF...THEN 或 CASE 指令后面。

如果 ELSE 指令出现在 IF 或 ELSIF 指令后面，则仅当 IF 和 ELSIF 指令的关联布尔表达式为 0（假）时，才会执行该指令或指令组。如果 IF 或 ELSIF 指令的条件为 1（真），则不会执行该指令或指令组。

如果 ELSE 指令出现在 CASE 后面，则仅当所有标签都不包含选择器的值时，才会执行该指令或指令组。如果某个标识包含选择器的值，则不会执行该指令或指令组。

注意： 可以嵌套任何数量的 IF...THEN...ELSE...END\_IF 指令，以生成复杂的选择指令。

示例 ELSE

```
IF A>B THEN
    C:=SIN(A) * COS(B) ;
    B:=C - A ;
ELSE
    C:=A + B ;
    B:=C * A ;
END_IF ;
```

- 选择指令 ELSIF...THEN

描述： ELSE 指令始终出现在 IF...THEN 指令后面。ELSIF 指令确定仅当 IF 指令的关联布尔表达式的值为 0（假）并且 ELSIF 指令的关联布尔表达式的值为 1（真）时，才会执行指令或指令组。如果 IF 指令的条件为 1（真）或者 ELSIF 指令的条件为 0（假），则不会执行该命令或命令组。THEN 指令标识 ELSIF 条件的结尾和指令的开头。

注意： 可以嵌套任何数量的 IF...THEN...ELSIF...THEN...END\_IF 指令，以生成复杂的选择指令。

示例 ELSIF...THEN

```
IF A>B THEN
    C:=SIN(A) * COS(B) ;
    B:=SUB(C,A) ;
ELSIF A=B THEN
    C:=ADD(A,B) ;
    B:=MUL(C,A) ;
END_IF ;
```

例如嵌套指令

```
IF A>B THEN
    IF B=C THEN
        C:=SIN(A) * COS(B) ;
    ELSE
        B:=SUB(C,A) ;
    END_IF ;
ELSIF A=B THEN
```

```

        C:=ADD(A,B) ;
        B:=MUL(C,A) ;
ELSE
        C:=DIV(A,B) ;
END_IF ;

```

- 选择指令 CASE...OF...END\_CASE

描述: CASE 指令包含一个 INT 数据类型的表达式 (选择器) 和一个指令组列表。每组都具有一个包含一个或多个整数 (INT、DINT、UINT 或 UDINT) 或整数值范围的标签。将执行的指令为其标签中包含选择器计算出的值的第一组指令。否则, 将不执行任何标签对应的指令。

OF 指令指示标签的开头。

所有标签都不包含选择器的值时, 才会在 CASE 指令内执行 ELSE 指令。

END\_CASE 指令标记指令的结尾。

- 重复指令 FOR...TO...BY...DO...END\_FOR

描述

FOR 指令用于在发生次数可预先确定的情况下。否则可使用 WHILE 或 REPEAT。FOR 指令会重复执行指令序列, 直到遇到 END\_FOR 指令为止。发生次数由起始值、结束值和控制变量决定。

控制变量、起始值和结束值必须具有相同的数据类型 (DINT 或 INT)。

控制变量、起始值和结束值可由重复指令进行更改。这是对 IEC 61131-3 的补充。

FOR 指令以控制变量值为步幅递增起始值, 直到达到结束值。增量值的缺省值为 1。

如果要使用其他值, 则可以指定显式增量值 (变量或常量)。每个新的循环之前都要检查控制变量值。如果它位于起始值和结束值的范围之外, 则将离开循环。

首次运行循环之前, 会进行检查以确定从初始值开始的控制变量递增是否是朝着结束值的方向。如果不是 (例如, 起始值  $\leq$  结束值并且增量为负值), 则不会对循环进行处理。控制变量值不是在循环外定义的。

DO 指令标识重复定义的结尾和指令的开头。

可以使用 EXIT 提前终止循环。END\_FOR 指令标记指令的结尾。

- 重复指令 WHILE...DO...END\_WHILE

说明: WHILE 指令可使一个指令序列重复执行, 直到其相关布尔表达式为 0 (假)。

如果从一开始该表达式就为假, 则根本不会执行该指令组。

DO 指令标识重复定义的结尾和指令的开头。可以使用 EXIT 提前终止循环。

END\_WHILE 指令标记指令的结尾。

下列情况下不应使用 WHILE, 因为它可能导致无限循环, 从而造成程序崩溃:

- WHILE 不能用于过程之间的同步, 例如, 不能用作具有外部定义的结束条件的"等待循环"。
- WHILE 不能用在算法中, 因为无法确保完成循环结束条件或执行 EXIT 指令。

示例 WHILE...DO...END\_WHILE

```

x := 1; WHILE x <= 100 DO x := x + 4; END_WHILE ;

```

- 重复指令 REPEAT...UNTIL...END\_REPEAT

描述: REPEAT 指令可使一个指令序列重复执行(至少执行一次),直到相关布尔条件为 1(真)。UNTIL 指令标记结束条件。可以使用 EXIT 提前终止循环。

END\_REPEAT 指令标记指令的结尾。

下列情况下不应使用 REPEAT,因为它可能导致无限循环,从而造成程序崩溃:

- REPEAT 不能用于过程之间的同步,例如,不能用作具有外部定义的结束条件的"等待循环"。
- REPEAT 不能用在算法中,因为无法确保完成循环结束条件或执行 EXIT 指令。

示例 REPEAT...UNTIL...END\_REPEAT

```
x := -1
REPEAT
    x := x + 2
UNTIL x >= 101
END_REPEAT ;
```

- 重复指令 EXIT

描述: EXIT 指令用于在满足结束条件前终止重复指令(FOR、WHILE 或 REPEAT)。如果 EXIT 指令位于嵌套的重复指令内,则会离开最里面的循环(EXIT 所在的循环)。接下来,将执行循环结尾(END\_FOR、END\_WHILE 或 END\_REPEAT)后的第一个指令。

示例 EXIT

如果 FLAG 的值为 0,执行指令后 SUM 将为 15。

如果 FLAG 的值为 1,执行指令后 SUM 将为 6。

```
SUM := 0 ;
FOR I := 1 TO 3 DO
    FOR J := 1 TO 2 DO
        IF FLAG=1 THEN EXIT;
    END_IF ;
    SUM := SUM + J ;
END_FOR ;
SUM := SUM + I ;
END_FOR
```



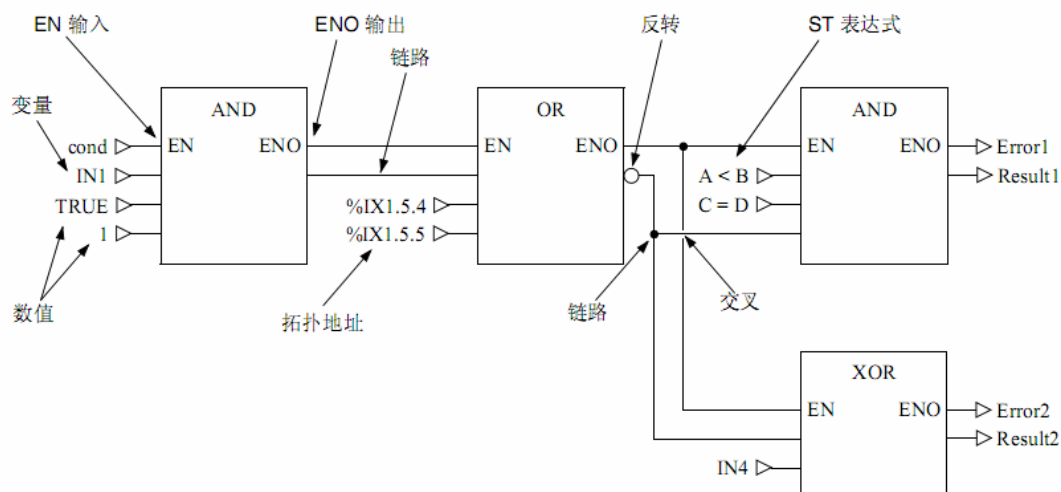
## 综述 本章节描述 FBD 语言的编程方法

### FBD 功能块指令

对象 FBD 编程语言（功能块图）的对象可帮助将一个段分成若干个 EF 和 EFB 基本功能和基本功能块；DFB（导出的功能块）；过程和控制元素。

这些对象按 FFB 的名称组合在一起，它们可以通过链路或实际参数方式相互链接。

**FBD 段的表示形式** 表示形式：



基本功能块（EFB）具有内部状态。每次调用该功能时，即使输入值相同，输出值也可能不同，例如对于计数器，输出值是递增的。

在图形表示中，基本功能块用包含多个输入和多个输出的块结构表示。输入始终表示在块结构的左侧，而输出始终表示在块结构的右侧。

功能块可以有多个输出。

功能块的名称（即功能块类型）显示在块结构的中央。

功能块的执行编号（见 FFB 的执行顺序页 314）显示在功能块类型的右侧。

实例名称显示在块结构的上方。

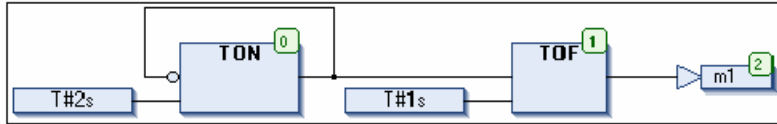
实例名称用作项目中的功能块的唯一标识。

可以修改这个自动生成的名称，以便标识实例。实例名称（最多 32 个字符）在整个项目中必须是唯一的，并且不区分大小写。实例名称必须符合一般命名约定。

## 综述 本章节描述 CFC 语言的编程方法


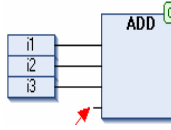

### CFC—连续功能图语言

CFC连续功能图是IEC61131-3 标准编程语言的扩展，是基于[功能块图](#)的图形化编程语言，但它没有网络限制，可任意放置元素，例如允许插入反馈回路，如下图。



#### 2.4.1 CFC 元素

|  |     |  |  |
|--|-----|--|--|
|  | 输入  |  | 选中‘???’文本，然后修改为变量或者常量。通过输入助手可以选择输入一个有效标识符。   |
|  | 输出  |  |  |
|  | 运算块 |  | <p>运算块可用来表示操作符，函数，功能块和程序。选中运算块的‘???’文本框，修改为一个操作符名，函数名，功能块名或者程序名。通过输入助手可以选择输入一个有效的对象。在例子中，当插入一个功能块，随即运算块上出现另一个‘???’，这时要把“???”修改为功能块实例名。</p> <p>若运算块被修改为另一个运算块（通过修改运算块名），而且新运算块的最大输入或输出引脚数，或者最小输入或输出引脚数与前者不同。运算块的引脚会自动做相应的调整。若要删除引脚，则首先删除最下面的引脚。</p> |
|  | 跳转  |  | 跳转用来指示程序下一步执行到哪里，这个位置是由标签定义的（见下）。插入一个新标签后，要用标签名替代“???”。  |
|  | 标签  |  | <p>标签标识程序跳转的位置（见上文“跳转”）</p> <p>在在线模式下，标识 POU 结束的返回标签会自动插入。</p>   |
|  | 返回  |  | 注意：在线模式下，return 自动插入到编辑器第一列的最后那个元素之后。在单步调试中，在离开该 POU 之前，会自动跳转到该 return。  |
|  | 编排器 |  | 编排器用于结构体类型的运算块输入。编排器会显示结构体的所有成员，以方便编程人员使用它们。使用方法是：先增加一个编排器到编辑器中，修改“???”为要使用的结构体名字，然后连接编排器的输出引脚和运算块的输入引脚。   |
|  | 选择器 |  | 选择器用于结构体类型的运算块输出。选择器会显示结构体的所有成员，以方便编程人员使用它们。使用方法是：先增加一个选择器到编辑器中，修改“???”为要使用的结构体名字，然后连接选择器的输出引脚和运算块的输出引脚。   |
|  | 注释  |  | 用该元素可以为图表添加注释。选中文本，即可以输入注释。用户可以用<ctrl>+<enter>在注释中换行。  |

|   |      |   |   |
|---|------|---|---|
|  | 输入引脚 |  | 有些运算块可以增加输入引脚。首先在工具箱中选中 Input Pin，然后拖放到在 CFC 编辑器中的算法块上，该运算块就会增加一个输入引脚。  |
|  | 输出引脚 |   | 有些运算块可以增加输出引脚。首先在工具箱中选中 Output Pin，然后拖放到在 CFC 编辑器中的算法块上，该运算块就会增加一个输出引脚。 |

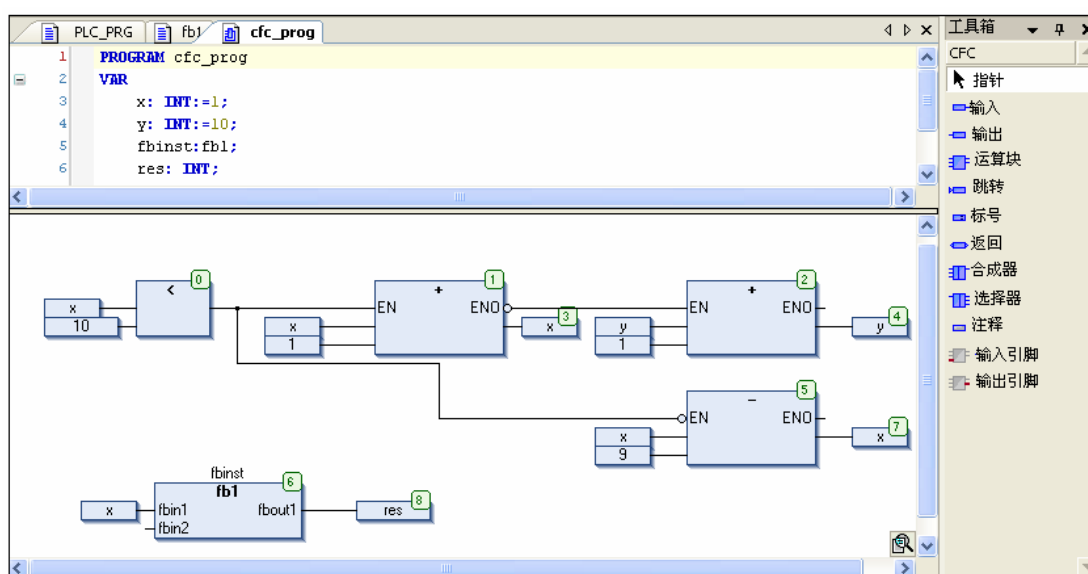
## CFC 编辑器

CFC编辑器由“CFC编辑器”插件提供，用于编写CFC（连续功能图）。CFC语言是对IEC 61131-3 编程语言的扩展。点击“工程”菜单下的“[添加对象](#)”，可以增加以CFC为编程语言的POU。 CFC编辑器是一个图形编辑器。

在编写CFC POU时，窗口的上半部分是[声明编辑器](#)，下半部分是CFC编辑器。

与网络编辑器不同，CFC编辑器允许把元素放在任何[位置](#)，例如，允许直接插入反馈回路。CFC 编辑器内部有一个链表，包含了所有已经插入的元素，链表的顺序决定了CFC元素的执行顺序，但是用户可以改变元素的执行顺序。

如下图所示

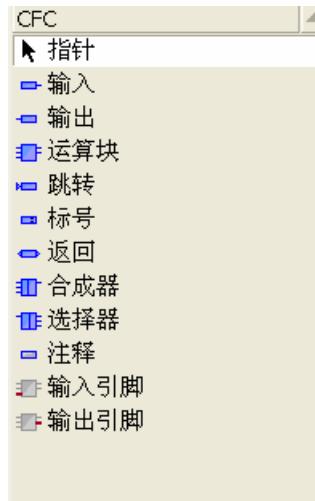


[工具箱](#)中包含下列元素：运算符（包括操作符、功能、功能块和程序），输入、输出、注释、标签、跳转、编排器和选择器；可以把这些元素插入到CFC编辑器中。

拖动鼠标，连接元素的输入和输出，会自动创建最短连接线。当元素移动的时候，连线也会自动调整。请参看：[插入和排列元素](#)。

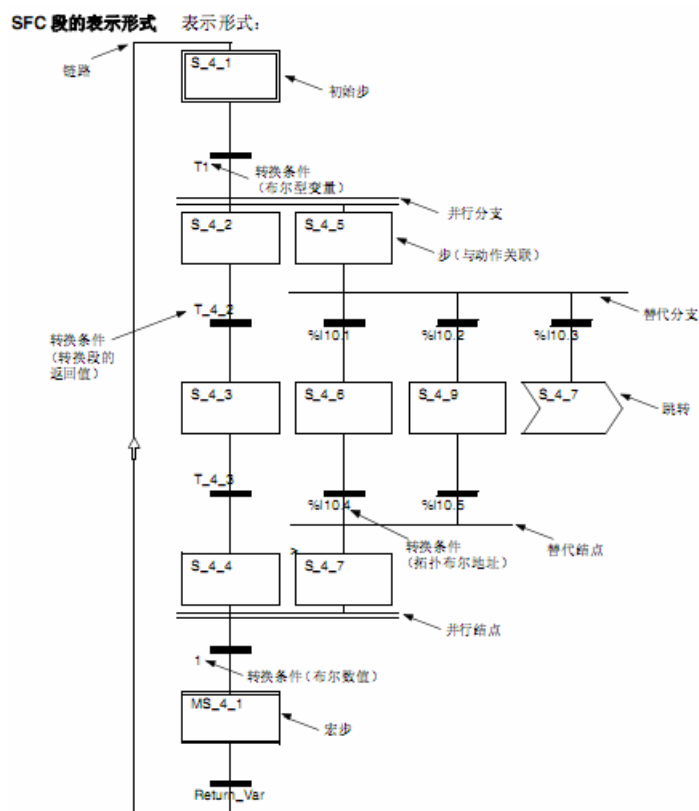
用户可以通过缩放工具改变编辑窗口的尺寸：点击编辑窗口右下角的按钮，在打开的菜单中选择一个缩放倍数；还有另外一个方法，在打开的菜单中选择 ... 打开一个对话框，然后可以输入任意缩放倍数。

当CFC编辑器处于激活状态时，通过右键菜单或者“CFC”菜单，可以使用CFC[命令](#)。



## 综述 本章节描述 SFC 语言的编程方法

顺序功能块图（SFC）是一种图形化语言，可以在一个程序内按照时间顺序对动作进行编辑描述。这些动作可以作为独立的编程对象，用任意编程语言进行编写。在SFC内，它们被分配到“步”元素，其处理顺序由“转移”元素进行控制。如下图



SFC 段是一个"状态机器", 即, 状态由活动步创建并且转换传递到切换/更改行为。步和转换通过方向链路相互链接在一起。两个步不得直接链接, 必须始终由转换分隔。活动信号状态沿方向链路的方向进行处理, 并通过切换转换进行触发。链处理的方向沿着方向链路的方向, 并从前一步的末端运行到下一步的顶端。分支从左向右进行处理。每一步可以没有操

作，也可以有多个操作。每个转换都需要有一个转换条件。链中的最后一个转换始终连接到链中的另一步（通过图形链路或跳转符号），以便形成一个闭环。因此，步链得到循环处理。  
SFC 内的处理顺序

在线模式下，一些类型的动作，可以根据定义的序列来执行，参见下表。

首先注意下述名词：

活动步：一步，它的步动作正在被执行，被叫做“活动”。在线模式下，活动步显示为蓝色。

初始步：在一个SFC POU被调用后的第一个周期内，初始步自动被激活，并且其相关联的“步动作”被执行。

IEC 动作：被至少执行两次：第一次执行是当它们被激活时，第二次执行是在下个周期，它们被禁止时。

选择分支：如果选择分支的水平起始线前的步被激活，则将从左至右计算每个特定分支的首个转移。从最左侧开始，第一个转移条件为 TRUE 的分支将被执行，即，此分支中后续的步将被激活。

并行分支：如果并行分支的起始双连线是活动的，并且前面的转换条件值为 TRUE，则在所有并行分支中的第一步都将被激活。这时这些分支会一个接一个的并行处理。当前面所有步都已激活，且双线后的转换条件值为 TRUE 时，分支结尾的双线后的步将被激活。

元素处理顺序（同 CoDeSysV2.3 的处理顺序）：

|               |  |
|---------------|--|
| 1. 复位         | IEC 动作的所有动作控制标志被复位（但是在动作内调用的 IEC 动作的标志则不会被复位！）。  |
| 2. 步退出动作      | 所有的步将按照流程图中定义的顺序进行检查（从上到下，从左到右），来判断步退出动作的执行条件是否满足，如果满足，则其将被执行。如果步马上要被禁止，则会执行一个退出动作，即，它的入口和步动作（如果存在）已经在上一个周期被执行了，并且下个步的转换条件为 TRUE。                                  |
| 3. 步入口动作      | 所有的步按照流程图中定义的顺序被测试，用来判断步的入口动作执行条件是否满足，如果满足，则其将被执行。如果步前的转移条件为 TRUE 并且步也已被激活，则一个入口动作将会被执行。   |
| 4. 超时检测，步激活动作 | 对于所有的步，下面内容将按照流程图中定义的顺序进行处理：<br>– （尚未实现）适用情况下，经过时间会被拷贝到对应的隐含步状态变量<stepname>.t 中<br>– （尚未实现）适用情况下，任何超时都可以被检测到，并且 SFC 错误标志会根据需要设定。<br>– 对于非 IEC 标准的步，相应的步活动动作现在被执行了。 |
| 5. IEC 动作     | 在流程图中使用的 IEC 动作，按照字母顺序执行。通过动作列表，有两种途径可以实现。第一种，所有在当前周期被禁止的 IEC 动作都将被执行。第二种，所有在当前周期被激活的 IEC 动作都将被执行。   |

|               |   |
|---------------|---|
| 6. 转移检测，激活下一步 | 转移被计算：如果当前周期的步是活动的，并且其后续转移返回 TRUE（并且如果已经超过了最小活动时间），则后续步被激活。 |
|---------------|---|

注意：

关于动作的实现，请务必注意下面的内容：

因为一个动作可以被分配给多个流程，因此其有可能在一个循环周期内被执行多次。（例如，一个 SFC 可以有两个 IEC 动作 A 和 B，它们都在 SFC 内实现，而且由都去调用 IEC 动作 C；这样在 IEC 中的动作 A 和 B 就有可能在同一个循环内被同时激活，而且同时 IEC 动作 C 也可以在这两个动作中被激活，这样 C 就有可能在同一循环内被调用了两次。）

如果同样的 IEC 动作被在一个 SFC 内的不同级别上同时使用，根据前面对处理顺序的说明可知，这可能会导致不可预期的结果。出于这样的原因，在这种情况下，会产生一个错误信息。在用老版本的编程系统创建工程时，可能会发生这中情况。