

TM246
Structured Text



预备课程和要求

培训手册	TM210 - Working with Automation Studio TM213 - Automation Runtime TM223 - Automation Studio Diagnostics
软件	Automation Studio 3.0.90 或更高版本
硬件	无

目录

1 简介.....	4
1.1 学习目标.....	4
2 概况.....	5
2.1 结构化文本的特征.....	5
2.2 编辑功能.....	6
3 基本要素.....	7
3.1 表达式.....	7
3.2 赋值.....	7
3.3 源代码文档：注释.....	8
3.4 操作符的优先级顺序.....	9
3.5 保留关键字.....	10
4 命令组.....	11
4.1 布尔运算.....	11
4.2 算术运算.....	12
4.3 数据类型转换.....	13
4.4 比较和判断.....	15
4.5 状态机 - CASE 语句.....	18
4.6 循环.....	20
5 Function, function block和action.....	26
5.1 调用Function和function block.....	26
5.2 调用actions.....	29
6 附加functions, 辅助functions.....	30
6.1 Pointers和references.....	30
6.2 IEC程序的预处理器.....	30
7 诊断功能.....	31
8 练习.....	32
8.1 练习-电梯.....	32
8.2 练习-分散搅拌机.....	33
9 总结.....	35
10 附录.....	36
10.1 数组.....	36
10.2 练习题参考解答.....	39

1 简介

结构化文本（ST）是一种高级编程语言。其基本元素概念源自BASIC、Pascal和ANSI C语言。由于其易于理解的标准结构，结构化文本（ST）是自动化现场编程的一种快速且有效的方式。



下面的章节将为你介绍在结构化文本中使用的命令、关键字和语法。本手册会提供一些简单的例子，让你有机会把这些概念付诸实践，使你更容易了解他们。

1.1 学习目标

该培训模块用来选择练习，以帮助你使用结构化文本（ST）来学习高级语言编程的基础知识。

- 你将了解高级语言编辑器和Automation Studio的SmartEdit功能。
- 您将学习使用高级语言编程中的基本要素，以及如何使用ST命令。
- 您将学习如何使用命令组和算术函数。
- 您将学习如何使用比较和布尔运算符。
- 您将学习如何调用用于控制程序流程的元素。
- 您将学习如何使用保留的关键字工作。
- 您将学习actions、functions 和function blocks之间的区别，以及如何使用它们。
- 您将学习如何使用高级语言编程的诊断功能

2 概况

2.1 结构化文本的特征

概况

ST是针对自动化系统的一种基于文本编程的高级语言。其简单的标准结构，使编程更快速、高效。使用多种高级语言的传统特征，包括变量、操作符、函数和元素来控制程序流程。

ST编程语言与IEC标准一致¹

特性

结构化文本具有以下特点：

- 基于文本编程的高级编程语言
- 结构化程序设计
- 易于使用的标准结构
- 快速高效的编程
- 自我解释和灵活
- 类似pascal
- 符合IEC 61131-3标准

功能

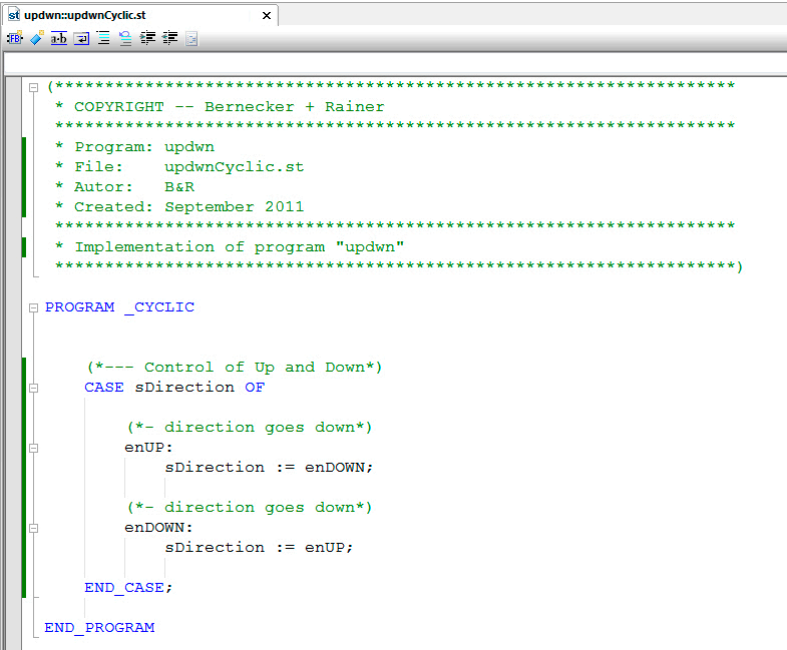
Automation Studio 支持以下功能：

- 数字量和模拟量输入和输出
- 逻辑运算
- 逻辑比较表达式
- 算术运算
- 判断
- Step切换机制
- 循环
- 功能块
- 动态变量
- 调用actions
- 综合诊断

¹ IEC 61131-3是全球适用的用于可编程逻辑控制器编程语言的标准。除了结构化的文本之外，其他的编程语言，如顺序功能图，梯形图，指令列表，和功能块图也被定义。

2.2 编辑功能

编辑器是一个具有许多附加功能的文本编辑器。命令和关键字以颜色区分。区域可以扩大和缩小。可对变量和结构体进行自动补全（SmartEdit）。



图片 1: ST编辑器中的用户程序

编辑器具有以下功能和特性:

- 大写字母和小写字母之间的区别（区分大小写）
- 自动补全（SmartEdit, <CTRL> + <SPACE>, <TAB>）
- 插入和管理代码片段（<Ctrl> + Q, K）
- 相应的对圆括号的识别
- 扩大和缩小结构（概述）
- 插入块注释
- URL识别
- 行修改标记

[Programming \ Editors \ Text editors](#)
[Programming \ Editors \ General operations \ SmartEdit](#)
[Programming \ Editors \ General operations \ SmartEdit \ Code snippets](#)
[Programming \ Programs \ Structured Text \(ST\)](#)

变量声明表中支持变量、常量、用户数据类型和结构体的初始化。另外，用户也可以给变量也可以添加注释。声明表也支持SmartEdit。

[Programming \ Editors \ Table editors \ Declaration editors](#)

3 基本要素

下面的部分更详细地介绍了ST的基本要素。 也介绍了表达式，赋值，在程序代码和保留关键字中使用注释，以及其他的东西。

3.1 表达式

表达式是一个结构，它计算完成后，返回一个值。 表达式由运算符和操作数组成。 运算对象可以是一个变量，常量或函数。 运算符用于连接操作数（3.4 “操作符的优先级顺序”）。 无论它是函数调用或者赋值，每个表达式必须以分号结束（“；”）。



```
b + c;
(a - b + c) * COS(b);
SIN(a) + COS(b);
```

表格 1: 表达式的示例

3.2 赋值

赋值是指左边的一个变量，在右边使用赋值操作符“:=”，分配其一个计算或表达式的结果。 所有的赋值必须以分号“;”结束。



```
Result := ProcessValue * 2;    (*Result ← (ProcessValue * 2) *)
```

表格 2: 赋值语句的执行顺序是从右到左的

当这行代码执行完成，“Result”变量的值是“ProcessValue”变量的两倍。

访问变量的位

在赋值时，也有可能只处理特定的位。 要做到这样，只需在变量名之后放置一个点（“.”）。 然后使用位号进行读取，从0开始。 位号的位置也可以使用常量。



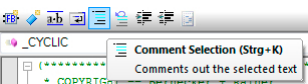
```
Result := ProcessValue.1;
```

表格 3: 访问变量值的第二位


3.3 源代码文档：注释

注释是源代码的一个重要组成部分。他们对代码进行了描述，使得代码更加易读易懂。注释使你或者其他能够理解，哪怕是在程序完成很久之后。注释不会被编译，不会对程序执行造成影响。注释必须放置在一对括号和星号之间，如（*注释*）。

另外注释也可由 “//”进行标识。可以在编辑器中选择几行代码，使用工具栏中的一个按钮，将其一起注释掉。这种变化是扩展现有的IEC标准。



图片 2：注释一个文本块

 单行注释	(*This is a single line of comment.*)
多行注释	(* These are several lines of comment. *)
使用“//”注释	// This is a general // text block. // It contains comment.

表格 4: Comment variations

	Programming \ Editors \ Text editors \ (Un)commenting selections
	Programming \ Structured software development \ Program layout

3.4 操作符的优先级顺序

使用不同的操作符带来的优先级的问题。 运算符的优先级在求解表达式时是很重要的。

求解表达式时，高优先级的操作符优先进行。 在表达式中出现相同优先级的操作符时，从左到右执行。

运算符	语句	
Braces	()	最高优先级
Function call	Call (Argument)	
Exponent	**	
Negation	NOT	
Multiplication, division, modulo operator	*, /, MOD	
Addition, subtraction	+, -	
Comparisons	<, >, <=, >=	
Equality comparisons	=, <>	
Boolean AND	AND	
Boolean XOR	XOR	
Boolean OR	OR	最低优先级

表达式的分解由编译器执行。 下面的例子表明，可以通过使用括号来实现不同的结果。



无括号时表达式的求解：

```
Result := 6 + 7 * 5 - 3;           (* ##### 38 *)
```


先进行乘法，然后为加法运算。 最后进行减法运算。

用圆括号求解表达式：

```
Result := (6 + 7) * (5 - 3);       (* ##### 26 *)
```

表达式从左到右执行。 在括号中的操作是在乘法之前执行的，因为括号有更高的优先级。 你可以看到，使用圆括号会导致不同的结果。


3.5 保留关键字



在编程中，所有的变量必须遵循一定的命名规则。 此外，已被编辑器确认的保留的关键字，会在编辑器中以不同的颜色显示。 这些不能作为变量使用。

OPERATOR和AsIecCon库是一个新项目中的标准部分。 它们包含的功能是IEC功能，被解释为关键词。

此外，该标准还定义了数字和字符串。 这使得有可能以不同的格式表示数字。



Programming \ Structured software development \ Naming conventions

Programming \ Standards \ Literals in IEC languages

Programming \ Programs \ Structured Text (ST) \ Keywords

Programming \ Libraries \ IEC 61131-3 functions

4 命令组

下面的命令组是高级语言编程的基本结构。 这些结构可以灵活地组合和嵌套在一起。

它们可以根据以下的命令组类别进行分类：

- [4.1 “布尔运算”](#)
- [4.2 “算术运算”](#)
- [4.4 “比较和判断”](#)
- [4.5 “状态机 - CASE 语句”](#)
- [4.6 “循环”](#)

4.1 布尔运算

布尔运算符可以用于在二进制级别上链接变量值。 NOT、AND、OR和XOR的区别。 操作数不一定是BOOL类型。 然而，应考虑操作数的顺序。 可以使用括号。

符号	逻辑操作	举例
NOT	Binary negation	<code>a := NOT b;</code>
AND	Logical AND	<code>a := b AND c;</code>
OR	Logical OR	<code>a := b OR c;</code>
XOR	Exclusive OR	<code>a := b XOR c;</code>

表格 5: 布尔连接词概述

这些操作的真值表是这样的：

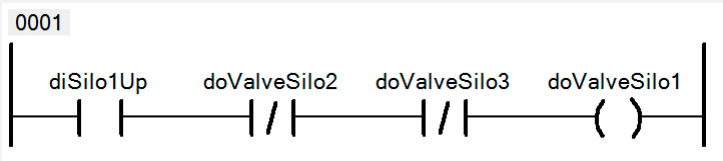
Input		AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

表格 6: 布尔运算的真值表

布尔运算可以以任何方式合并。 额外的一组圆括号增加程序的可读性，并确保正确地求解表达式。 表达式的唯一可能的结果是真的（逻辑1）或错误（逻辑0）。



布尔连接 - 梯形图和ST语言的比较



图片 3: 连接常开和常闭触点
doValveSilo1 := diSilo1Up AND NOT doValveSilo2 AND NOT doValveSilo3;
表格 7: 布尔连接的实现

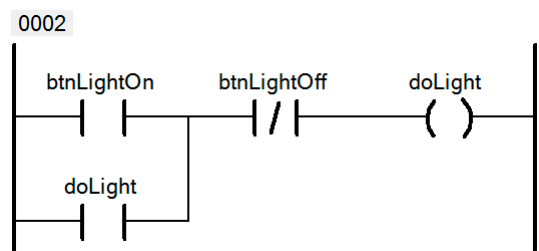
不需要有一个比AND更高的优先级。 然而，使用圆括号来使程序更清晰是一个好主意。

```
doValveSilo1 := (diSilo1Up) AND (NOT doValveSilo2) AND (NOT doValveSilo3);
```

表格 8: 使用括号实现布尔连接词

练习：灯光控制

在 “BtnLightOn”按钮被按下时，输出 “DoLight”应该是ON，并且应该保持ON直到 “BtnLightOff” 按钮被按下。 使用布尔运算来完成这个练习。



图片 4: 开关，带锁存器的继电器

4.2 算术运算

高级编程语言的一个关键优点是它们易于操作的算术运算。

算术运算概述

结构化文本为应用程序提供基本的算术运算。 同样的，在使用操作符时注意他们的顺序。 例如，乘法是在加法之前进行的。 使用括号可以实现所需的顺序。

符号	算术运算	举例
:=	Assignment	a := b;
+	Addition	a := b + c;
-	Subtraction	a := b - c;
*	Multiplication	a:= b *c;

表格 9: 算术运算概述

符号	算术运算	举例
/	Division	<code>a := b / c;</code>
MOD	Modulo, 整数除法的余数	<code>a := b MOD c;</code>

表格 9: 算术运算概述

变量的值和数据类型始终是计算的关键。计算右侧的表达式的结果，然后分配给相应的变量。其结果取决于所使用的数据类型和语法（符号）。下表对此进行了说明。

Expression / Syntax	数据类型			Result
	Result	Operand1	Operand2	
<code>Result := 8 / 3;</code>	INT	INT	INT	2
<code>Result := 8 / 3;</code>	REAL	INT	INT	2.0
<code>Result := 8.0 / 3;</code>	REAL	REAL	INT	2.66667
<code>Result := 8.0 / 3;</code>	INT	REAL	INT	*Error

表格 10: 由编译器执行的隐式转换

结果当中的“*Error”代表下列编译器错误信息：“**Error 1140: Incompatible data types: Cannot convert REAL to INT.**”。这是因为它不能将表达式的结果分配到这个特定的数据类型。

4.3 数据类型转换

用户在编程时，不可避免地面临着不同的数据类型。原则上，一个程序中能够混合不同的数据类型。不同的数据类型的分配也是可能的。不过，你应该小心使用。

隐式数据类型转换


每一次执行程序代码，编译器会检查数据类型。在一个语句中，赋值总是由右到左进行的。因此，变量必须有足够的空间来保存值。从一个较小的数据类型转换到一个更大的的是由编译器隐式地进行，而不需要任何用户所需的任何动作。

试图将一个较大的数据类型的值赋给一个较小的数据类型的变量，将导致编译器错误。在这种情况下，必须要进行显式数据类型转换。



即使进行了隐式数据类型转换，它仍然可能是一个不正确的结果，这取决于不同的编译器和转换类型的不同。将一个无符号值赋值给有符号的数据类型就是这里所提到的一种危险操作。


当你进行加法和乘法时，将导致溢出错误。然而这取决于所使用的平台。在这种情况下，编译器不会产生警告。

	表达式	数据类型	备注
	Result := Value; (*UINT USINT*)	UINT, USINT	无需额外操作实现变量类型转换。
	Result := Value; (*INT UINT*)	INT, UINT	用到负数的时候要小心！
	Result := value1 + value2; (*UINT USINT USINT*)	UINT, USINT, USINT	相加时会有范围溢出的危险


表格 11: 隐式数据类型转换的例子

显式数据类型转换

虽然隐式数据类型转换往往是比较方便的方法，但它不应该永远是第一选择。 清晰的编程，必须正确使用明确的数据类型转换处理类型。 下面的例子突出一些必须采用显式转换的情况。



所有的变量声明按照IEC格式列出。 变量声明可以在相应程序的 .var文件中的文本视图中按格式输入。

	当下列情况发生时，会有发生溢出错误的危险：
声明：	<pre>VAR TotalWeight: INT; Weight1: INT; Weight2: INT; END_VAR</pre>
程序代码：	<pre>TotalWeight := Weight1 + Weight2;</pre>


表格 12: 溢出错误可以直接发生在赋值操作符的右边。

在这种情况下，结果的数据类型必须有足够的空间来保存加法运算的和。 因此，有一个更大的数据类型是必要的。 当做加法是，必须将至少一个操作数转换为更大的数据类型。 第二个操作数的转换，由编译器隐式处理。

声明：	<pre>VAR TotalWeight: DINT; Weight1: INT; Weight2: INT; END_VAR</pre>
程序代码：	<pre>TotalWeight := INT_TO_DINT (Weight1) + Weight2;</pre>

表格 13: 使用显式转换可以防止溢出错误。

在32位平台上，被计算的操作数会被转换为32位的值。 在这种情况下，加法不会导致溢出错误。



Programming \ Variable and data types \ Data types \ Basic data types

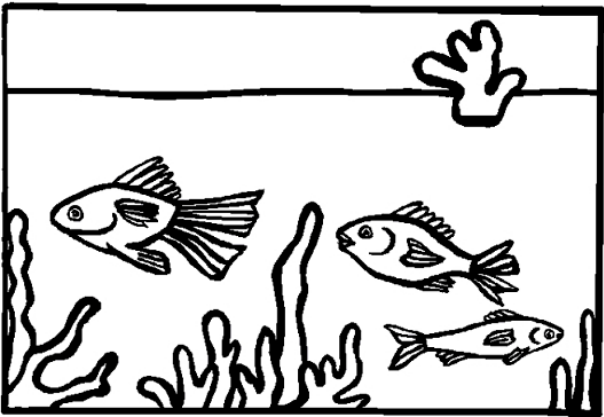
Programming \ Libraries \ IEC 61131-3 functions \ CONVERT

Programming \ Editors \ Text editors

Programming \ Editors \ Table editors \ Declaration editors \ Variable declaration

练习：水族箱

水族箱的温度是在两个不同的地方测量的。 创建一个程序，计算平均温度，并将其传递到一个模拟输出。 不要忘记模拟输入和输出的数据类型必须为 int。



图片 5：水族箱

声明：

```
VAR
    aiTemp1: INT;
    aiTemp2: INT;
    aoAvgTemp: INT;
END_VAR
```

表格 14：建议的变量声明

4.4 比较和判断

在结构化文本中，提供了简单的结构来比较变量。 他们返回的值为TRUE 或FALSE。 比较运算符和逻辑运算的主要用于条件语句如，IF、ELSIF、WHILE和UNTIL。

符号	比较类型	举例
=	等于	IF a = b THEN
<>	不等于	IF a <> b THEN
>	大于	IF a > b THEN
>=	大于或等于	IF a >= b THEN
<	小于	IF a < b THEN
<=	小于或等于	IF a <= b THEN

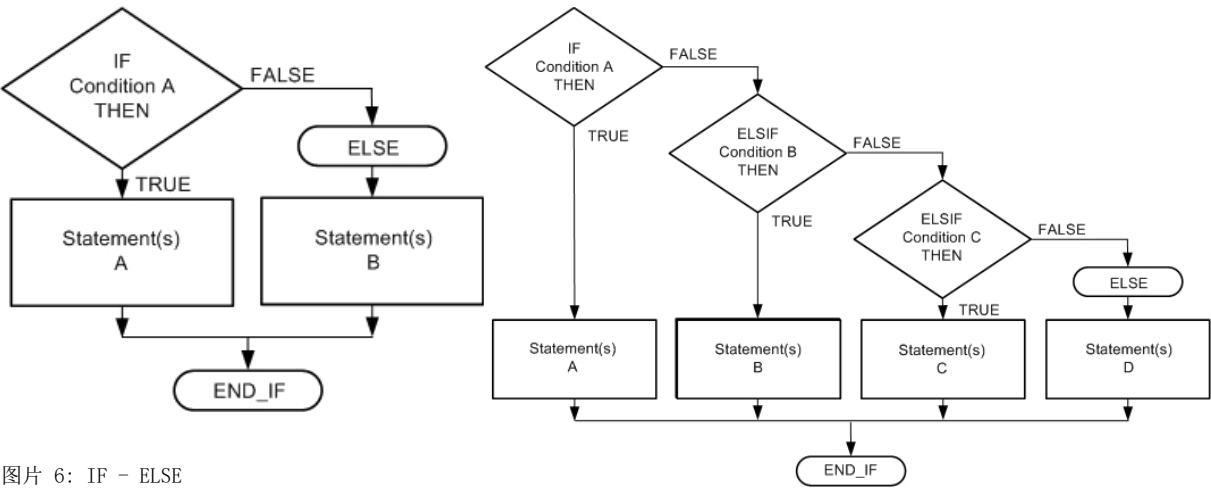
表格 15：逻辑比较运算符的概述

判断

IF语句用于处理程序中的决策。 你已经知道比较操作符。 可以在这里使用。

关键词	语句	描述
IF .. THEN	IF a > b THEN	1. 比较
	Result := 1;	如果第一个比较结果为TRUE，那么执行左边的语句
ELSIF .. THEN	ELSIF a > c THEN	2. 比较
	Result := 2;	如果第二个比较结果为TRUE，那么执行左边的语句
ELSE	ELSE	另一个分支，上述比较没有一个是TRUE
	Result := 3;	如果执行另一个分支，那么执行左边内容
END_IF	END_IF	判断结束

表格 16: IF语句的语法




图片 6: IF - ELSE

图片 7: IF - ELSIF - ELSE

表格 17: IF 状态概览

比较表达式可以与布尔运算符结合，从而可以同时判断多个条件。

	Explanation:	如果“a”大于“b”并且“a”小于“c”，那么“Result”等于100。
	程序代码:	<pre>IF (a > b) AND (a < c) THEN Result := 100; END_IF;</pre>

表格 18: 使用多个比较表达式

IF语句也可以嵌套额外的IF语句。 不过不要使用太多的嵌套层次，否则会使得程序变得混乱。



编辑器的SmartEdit 功能可以使输入信息更容易。 如果你想插入一个if语句，只需键入“IF”然后按<<TAB>>键。 编辑器中会自动添加“IF”语句的基本结构。



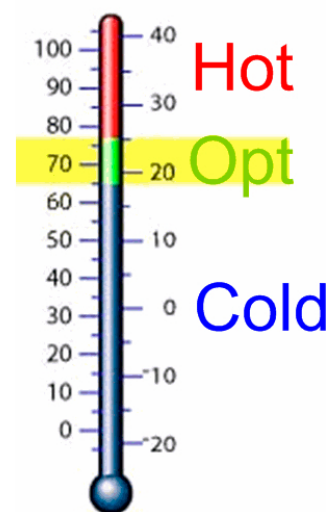
Programming \ Programs \ Structured Text (ST) \ IF statement
Programming \ Editors \ General operations \ SmartEdit

练习：气象站-第一部分

使用温度传感器测量外界温度。 此温度使用一个模拟输入量读取，作为内部的文本信息进行输出。

- 1) 如果温度低于18° C，应显示Cold”。
- 2) 如果温度介于18° C和25° C之间，显示应读“OPT”（最佳）。
- 3) 如果温度超过25° C，应显示“Hot”。

使用IF、ELSIF和ELSE语句创建一个解决方案。



图片 8：温度计



输出文本所需的字符串类型的变量。 可以像这样赋值： `sShowText := 'COLD';`

练习：气象站-第二部分

除了温度之外，再考虑湿度。

当湿度在40%到75%之间，并且温度介于18° C到25° C之间时，显示“Opt”文本。 否则显示“TempOK”。

使用嵌套的IF语句解决此任务。



如果几个IF语句检查的是相同的变量，那么应该考虑使用一个CASE语句，这可能是一个更好的，更清晰的解决方案。

比起IF语句，CASE语句有一个显著的优势，那就是他只需要比较一次，是的程序代码更高效。

4.5 状态机 - CASE 语句

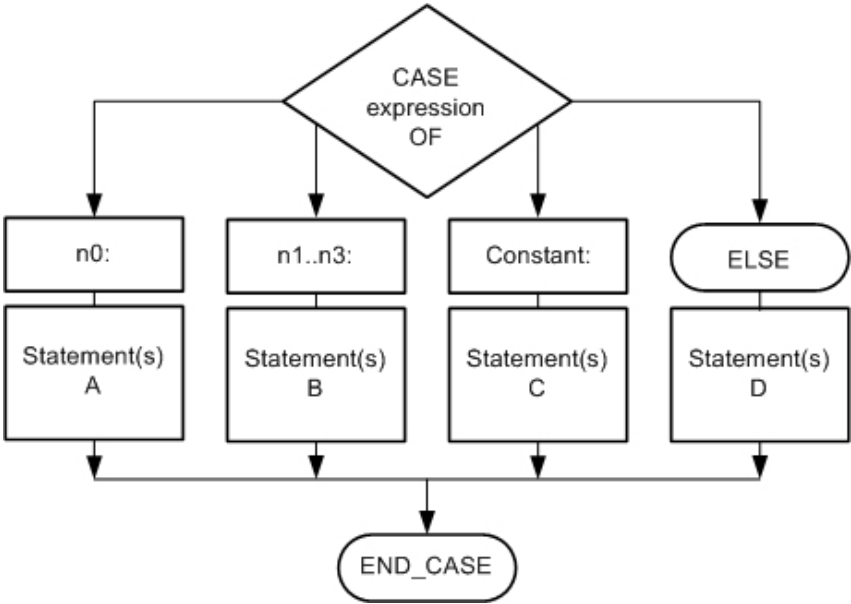
CASE语句比较了一个具有多个值的step变量。 如果其中一个比较匹配，则执行与该步骤相关联的语句。 和IF语句一样，如果没有一个比较匹配，那么在ELSE语句下的程序代码将被执行。

根据应用程序，可以使CASE语句来设置状态观测或自动检测。


关键词	语句	描述
CASE .. OF	CASE sStep OF	CASE语句的开头
	1, 5: Show := MATERIAL;	如果sStep为1或者5
	2: Show := TEMP;	如果sStep为 2
	3, 4, 6..10: Show := OPERATION;	如果sStep为3, 4, 6, 7, 8, 9 或者10
ELSE	ELSE	其他
	(*...*)	
END_CASE	END_CASE	CASE的结束

每一个程序周期只有一个CASE语句的一个step。


step变量必须是一个整数数据类型。



图片 9: CASE语句概述



比起固定的数值，在程序代码中更应该使用常量或者枚举数据类型。 使用文本替换一个数值，会使程序代码更加容易阅读。 此外，如果这些值需要在程序中更改，那么只需要在声明的更改一次，而用在程序代码中更改。



Programming \ Programs \ Structured Text (ST) \ CASE statement

Programming \ Variables and data types \ Variables \ Constants

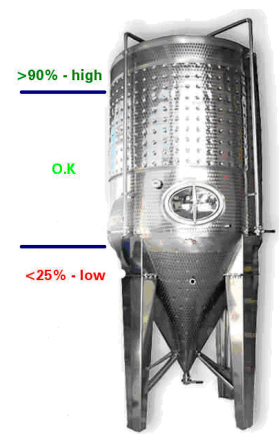
Programming \ Variables and data types \ Data types \ Derived data types \ Enumerations

练习：水位控制

容器中的水位应监测三个区域：low, high 和 OK。
low, high 和 OK每一个都作为一个输出。

罐内液位的输入为模拟量（0 - 32767），并转化为一个百分比值（0-100%）。

如果罐内液位低于1%时，应发出警告音。 使用CASE语句创建一个解决方案。



图片 10: 测量容器内的液位

声明:

```
VAR
    aiLevel : INT;
    PercentLevel : UINT;
    doLow : BOOL;
    doOk : BOOL;
    doHigh : BOOL;
    doAlarm : BOOL;
END_VAR
```

表格 19: 建议的变量声明

4.6 循环

在许多应用程序中，在同一周期内重复执行代码段是有必要的。这种类型的处理被称为一个循环。循环中的代码被一直执行，直到满足定义的终止条件。

循环使程序看起来更清晰、更短。扩展程序功能的能力也是它的一个问题。

这取决于程序的结构，有可能产生错误，阻止程序跳出循环，直到CPU的时间监控机制进行干预。为了避免这样的死循环，总要提供一种在循环指定次数之后终止循环的方法。

有两种主要类型的循环：一种循环控制开始于顶部，而另一种在底部开始。

FOR和WHILE语句会在一开始先检查是否满足循环开始条件，如果满足再开始循环操作。REPEAT语句会在一开始先执行循环操作，在语句最后判断是否已满足终止条件，如果满足则终止循环。这将会被循环执行至少一次。

4.6.1 FOR 语句

FOR语句用于重复执行有限次数的程序部分。WHILE和REPEAT循环用于预先未知循环次数的程序部分。


关键词	语句
FOR .. TO .. BY ² .. DO	FOR i:= StartVal TO StopVal BY Step DO
	Res := Res + 1;
END_FOR	END_FOR

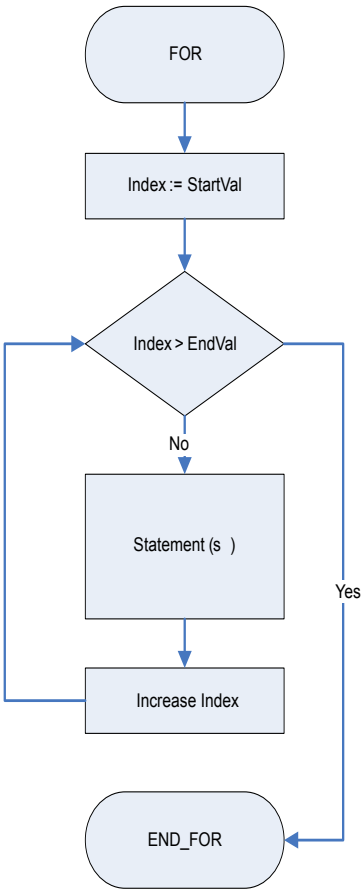
表格 20: FOR语句的元素

² Specifying the keyword "BY" is optional.

循环计数器的“index”是被开始值”startval”进行初始化。循环重复执行，直到循环计数器’Index’的值达到变量“stopval”。在这里，循环计数器总是增加了1，被称为“BY step”。如果一个负值被用于“step”增量，则循环将向后计数。

循环计数器，开始值和结束值都必须具有相同的数据类型。这可以通过显式数据转换来实现（4.3 “数据类型转换”）。

 如果开始和结束值在开始时是相同的，这种类型的循环将至少执行一次！（例如，如果起始值和结束值都等于0）

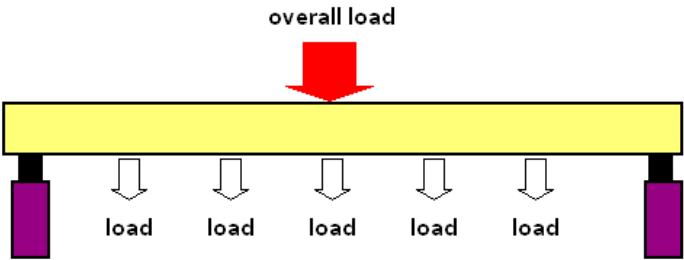


图片 11: FOR语句概述

 Programming \ Programs \ Structured Text (ST) \ FOR statement

练习：起重机总负载

一个起重机一次性能举起5个负载。负载接收器每个连接到模拟量输入并返回范围在0到32767的值。为计算总的负载和平均值，单一负载必须第一时间被合计并且除以负载传感器的数量。练习用FOR循环创建一个解决方案。



图片 12: 5个负载的起重机

声明:

```
VAR
    aWeights : ARRAY [0..4] OF INT;
    iCnt : USINT;
    sumWeight : DINT;
    avgWeight : INT;
END_VAR
```

表格 21: 建议的变量声明



需要数组解决这个练习。 深远的关于数组的信息能在附录或者AS里找到。
在可能的地方，使用负载的数组声明和常量去限制循环结束值。 这个相当地改善了声明的可读性和程序。 之后做出改变也变得更加容易了。



Programming \ Programs \ Variables and data types \ Data types \ Composite data types \ Arrays
See [10.1 “数组” on page 36](#).

练习: 起重机总负载-改善程序代码

由先前的任务得出结论，单一负载总数能在一个循环的帮助下生成。 直到现在，确定的数值位于变量声明以及程序代码中。 这个任务的目的是去用常量从声明和程序代码里替换尽可能多的确定数值。

声明:

```
VAR CONSTANT
    MAX_INDEX : INT := 4;
END_VAR
```

表格 22: 建议的变量声明

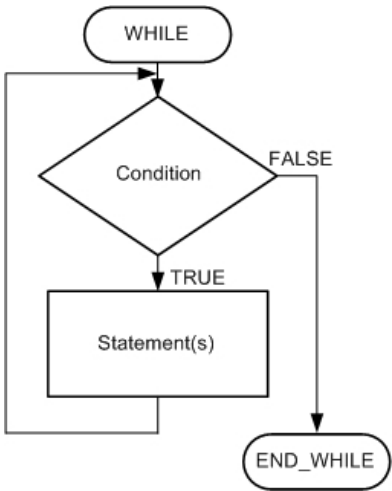
4.6.2 WHILE 语句

不像FOR语句，WHILE循环不会依附于一个循环计数器。 只要一个状态或表达式为TRUE，此类型的循环就被执行。 确定循环有一个结束以至于一个周期时间妨碍不会发生在runtime里是重要的。


关键词	语句
WHILE .. DO	WHILE i < 4 DO
	Res := value + 1;
	i := i + 1;
END_WHILE	END_WHILE

表格 23: 执行一个WHILE语句

只要状态为TRUE，语句就能被重复执行。 若第一时间状态返回FALSE，它就被评估，然后语句从不被执行。



图片 13: WHILE语句的概览

 Programming \ Programs \ Structured Text (ST) \ WHILE statement


4.6.3 REPEAT语句

一个REPEAT循环跟WHILE循环不同，因为最终状态只在循环执行后被检查。 这表明循环运行至少一次，不管最终状态如何。

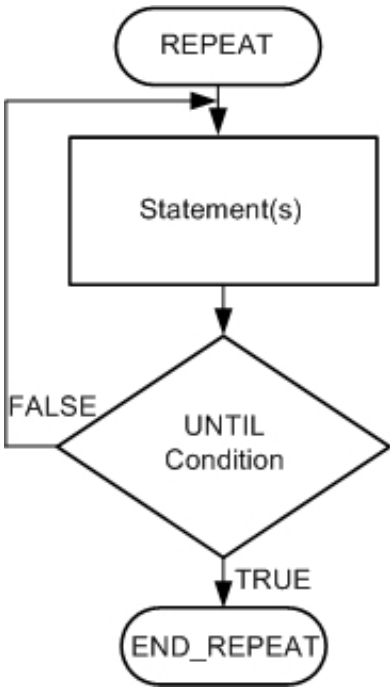
关键词	语句
REPEAT	REPEAT
	(*program code*)
	i := i + 1;
UNTIL	UNTIL i > 4
END_REPEAT	END_REPEAT

表格 24: 执行一个REPEAT语句

语句被重复执行直到UNTIL状态为TRUE。 若在最开始的时候UNTIL状态为真，语句只被执行一次。



若UNTIL状态从不呈现这个值TRUE，语句就无限地重复，导致了一个runtime错误。



图片 14: REPEAT语句概览



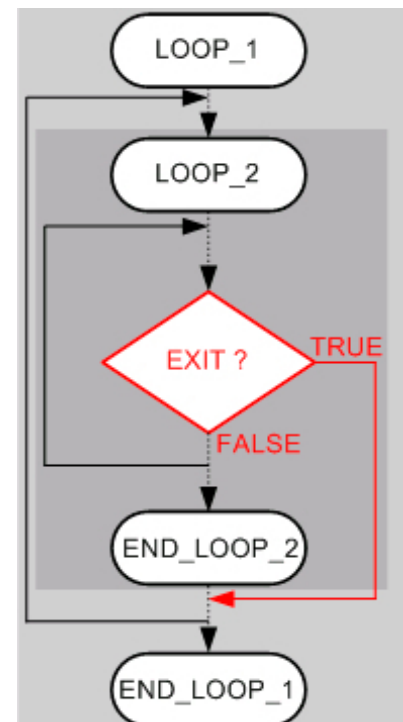
Programming \ Programs \ Structured Text (ST) \ REPEAT statement

4.6.4 EXIT语句

在最终状态达到之前，EXIT语句能被用在所有不同类型的循环 若EXIT被执行，循环就终止。

关键词	语句
	REPEAT
	IF setExit = TRUE THEN
EXIT	EXIT;
	END_IF
	UNTIL i > 5
	END_REPEAT

一旦在循环中的EXIT语句已经被执行，循环就终止，无论是否最终状态或者循环结束值已达到。在嵌套循环里，只有包括了EXIT语句的循环被终止。



图片 15: EXIT语句只终止嵌套循环。



Programming \ Programs \ Structured Text (ST) \ EXIT statement

练习：终值搜寻

100个数字的一个列表应被搜索去找到一个特殊的数字。此表包括随机数。若数字10被发现，搜索过程应被终止。然而，这个数字不在列表里是可能的。

在你的解决方案里用REPEAT和EXIT语句。记住2个终止状态。

声明：

```

VAR
    aValues : ARRAY[0..99] OF INT;
END_VAR
  
```



单一的数组元素也能用在程序代码或变量声明窗口中被预初始化。



Programming \ Editors \ Table editors \ Declaration editors \ Variable declaration

5 Function, function block和action

多样的function和 function block扩展一个编程语言的功能。 Actions被用于更加有效地构建程序。 Function和function block能从工具栏中插入。



图片 16: 从工具栏中插入Functions 和function block

5.1 调用Function和function block

Function

Function是当被调用时能返回一个特殊值的子程序。 一个function能被调用的一种方法是一个表达式。 传递parameter， 也被称为parameter， 是能被传递到一个function的值。 它们被逗号隔开。

声明:	<pre>VAR sinResult : REAL; xValue : REAL; END_VAR</pre>
程序代码:	<pre>xValue := 3.14159265; sinResult := SIN(xValue);</pre>

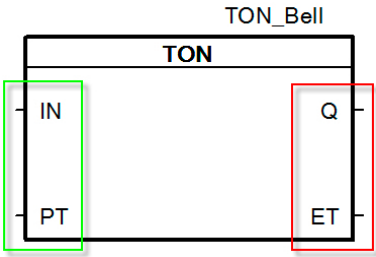
表格 25: 用一个传递parameter 调用一个function

 Programming \ Programs \ Structured Text (ST) \ Calling functions

功能块

一个function block具有能呈现几个传递parameter 并返回多种结果的事实特征。

不像一个function， 声明一个跟function block有相同数据类型的实例变量是有必要的。 在此的优势是一个function block能通过计算一个在有几周期的一段时间的结果来处理更复杂的任务。 通过使用不同的例子， 几个多样的相同类型的function blocks能用不同传递parameter调用。



图片 17: 用于TON () function block的传递parameter (绿) 和结果 (红)

当调用一个function block， 传递所有parameter 或者仅仅他们中的一些是有可能的。 使用实例变量中的元素， parameter 和结果能被存取进程序代码里。

声明:	<pre>VAR diButton : BOOL; doBell : BOOL; TON_Bell : TON; END_VAR</pre>
调用方式1:	<pre>TON_Bell(IN := diButton, PT := T#1s); doBell := TON_Bell.Q;</pre>
调用方式2:	<pre>(*-- parameters*) TON_Bell.IN := diButton; TON_Bell.PT := T#1s; (*-- call functionblock*) TON_Bell(); (*-- read results*) doBell := TON_Bell.Q;</pre>

表格 26: 用TON () function block消除一个按钮的抖动

声明:	<pre>VAR diButton : BOOL; doBell : BOOL; TON_Bell : TON; END_VAR</pre>
调用功能块方法1:	<pre>TON_Bell(IN := diButton, PT := T#1s); doBell := TON_Bell.Q;</pre>
调用功能块方法2:	<pre>(*--####*) TON_Bell.IN := diButton; TON_Bell.PT := T#1s; (*-- #####*) TON_Bell(); (*-- #####*) doBell := TON_Bell.Q;</pre>

表格 27: 使用TON () 功能块创建声明一个按钮

在变量1的调用中，当function block被调用的时候所有parameter 都被直接传递。 在变量2的调用中，parameter 被分配到例子变量的元素中。 在两个情况下，在调用已经发生后期望结果必须从例子变量中读到。

 Programming \ Programs \ Structured Text (ST) \ Calling function blocks

练习: Function block

调用STANDARD库中的一些功能块。在执行此操作之前，请查看“AS帮助”中的功能和参数说明。

- 1) 调用延时功能块 TON 。
将变量 “diSwitch” 关联到引脚 “IN” 上，用于开启延时器功能。
- 2) 调用计数器 CTU 功能块，
将变量 “diCountImpuls” 关联到引脚 “CU” 上，其上升沿用于开启计数功能。将变量 “diReset” 关联到引脚 “ RESET ” 上，用于给计数器清零。

声明:

```
VAR
    TON_Test : TON;
    CTU_Test : CTU;
    diSwitch : BOOL;
    diCountImpuls : BOOL;
    diReset : BOOL;
END_VAR
```

表格 28: 变量声明建议



在AS帮助中可以找到库功能的更详细描述。按<F1> 键可以打开光标所在功能块的帮助文档
许多库也包括不同的应用例子。



Programming \ Libraries \ IEC 61131-3 functions \ STANDARD
Programming \ Examples

5.2 调用actions

action是能被添加进程序和库的程序代码。 Actions为构建程序提供的另一种方法，并且能够成为编程程序中的一部分。 Actions通过它们的名字加以标识。

调用一个action与调用function相似。 不同点是不存在传递parameters和不返回的值。



若你用一个CASE语句去控制一个复杂的序列，举个例子，单一的CASE步骤内容能被传递到actions。 这个保持了主程序的紧凑。 若相同的功能在其他地方需要，然后再次调用action是很轻松的。



```

CASE Sequence OF
  WAIT:
    IF cmdStartProc = 1 THEN
      Sequence := STARTPROCESS;
    END_IF

    STARTPROCESS:
      acStartProc; (*machine startup*)

      IF Process_finished = 1 THEN
        Sequence := ENDPROCESS;
      END_IF

      ENDPROCESS:
        acEndProc; (*machine shutdown*)
        (*...*)
      END_CASE

  ACTION acStartProc:
    (*add your sequence code here*)
    Process_finished := 1;
  END_ACTION

```

程序:

Action:

表格 29: 在主程序调用actions



Actions

6 附加functions，辅助functions


6.1 Pointers和references

为扩展目前的IEC标准功能，贝加莱也提供在ST语言下的pointers。 这个允许在runtime期间一个模拟变量被分配一个特殊的存储地址。 这个程序被称为referencing或一个模拟变量的初始化。

一旦模拟变量被初始化了，它能被用去访问它“指向”的存储地址的内容。 为了此操作，关键词ACCESS被使用。

声明:	<pre>VAR iSource : INT; pDynamic : REFERENCE TO INT; END_VAR</pre>
程序代码:	<pre>pDynamic ACCESS ADR(iSource); (*pDynamic references to iSource*)</pre>

表格 30: 引用一个指针

 延伸的IEC标准功能能在AS项目的设置中使能。


6.2 IEC程序的预处理器

在基于文本的编程语言中，使用预处理器指令是可能的。 至于语法，被广泛实施的语句符合那些ANSI C预处理器。

这些预处理器命令是一个IEC的扩张。 他们必须在项目设置中使能。

预处理器指令被用来程序的条件编译或者整个配置。 当启动时，编译器设置使能影响编译器处理如何发生的功能。

可以在Automation Studio帮助中找到所有可用函数/功能块的说明和完整列表。



Project management \ The workspace \ General project settings \ Settings for IEC compliance

Programming \ Programs \ Preprocessor for IEC programs

7 诊断功能

理解的诊断工具使得整个程序进程更加高效。 AS包含了几个用于排除高级编程语言错误的工具。

- 监控模式
- 监控窗口
- 线路覆盖
- 工具提示
- 调试器
- 交叉索引



Diagnostics and service \ Diagnostics tool \ Debugger

Diagnostics and service \ Diagnostics tool \ Variable watch

Diagnostics and service \ Diagnostics tool \ Monitors \ Programming languages in monitor mode \ Line coverage

Diagnostics and Service \ Diagnostic tools \ Monitors \ Programming languages in monitor mode \ Powerflow

Project management \ The workspace \ Output window \ Cross reference

8 练习

8.1 练习-电梯

练习：电梯

两传送带 (doConvTop, doConvBottom) 正被用去传送箱子到一个电梯。

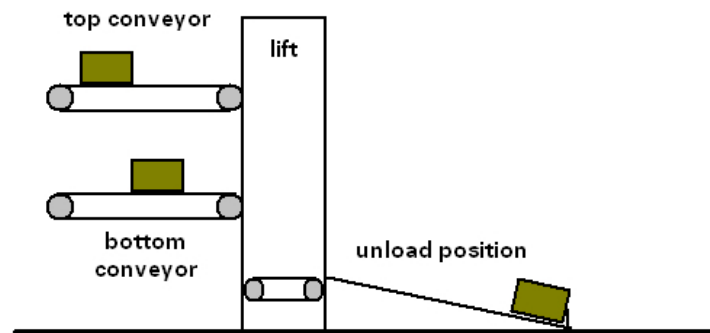
若光电池 (diConvTop or diConvBottom) 被激活，相应的传送带停止并且电梯被请求。

若电梯没有正在被请求，它就返回到适当的位置 (doLiftTop, doLiftBottom)。

当电梯在被要求的位置时 (diLiftTop, diLiftBottom)，电梯传送带 (doConvLift) 被启动直到箱子被完整地放在电梯上 (diBoxLift)。

电梯之后移动到卸货位置 (doLiftUnload)。 当它达到这个位置时 (diLiftUnload)，箱子就被放在卸货传送带上。

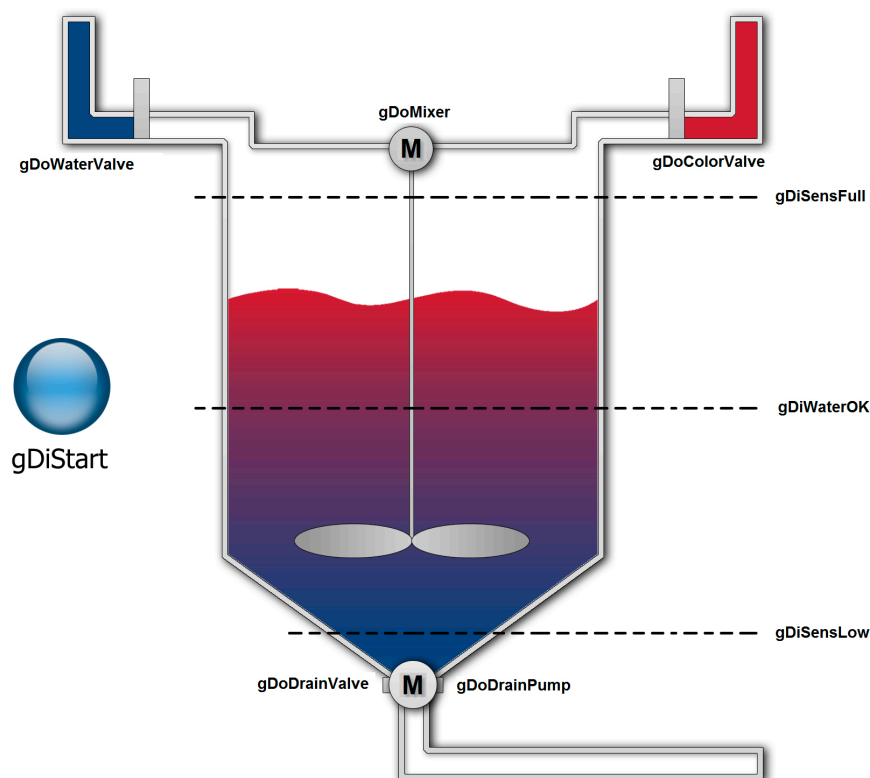
一旦箱子离开了电梯，电梯就等待下一次请求。



图片 18: 电梯

8.2 练习-分散搅拌机

这里我们将配置一个能混合水和颜料成分散的混合系统。



图片 19: 调漆系统图表

混合系统将依照以下程序运行：

- 混合系统等待直到开启按钮被按下(gDiStart)。
- 水(gDoWaterValve)被添加进容器里直到“gDiWaterOK”传感器被触发。
- 混合单元(gDoMixer)启动并且颜料(gDoColorValve)被添加直到“gDiSensFull”传感器被触发。
- 混合时间应花30秒。
- 打开排水阀(gDoDrainValve)和排水泵gDoDrainPump)来 清空容器。
- 当“gDiSensLow”信号被触发时，排水过程完成了。
- 开始状态被还原。

练习：实施分散混合器

即将到来的任务是去分析分散混合器进程然后去一步一步用ST语言编码它。 为了这样做，要实施一下步骤：

- 1) 草拟这个过程。

- 定义步骤。
 - 定义传递。
(从一步到接下来的一步传递状态)
 - 定义actions。
(在一步中被直接调用的程序段)
- 2) 【可选】决定哪一步能被传递到一个IEC action。
- 3) 实现了ST程序的需求



一个从STANDARD库的计时块应被用于实施混合时间。 确保在时间到期后计时器被重置。

9 总结

ST语言是一门提供一个广泛的功能的高级编程语言。它包括了每件事都有必要去创建一个处理特殊任务的应用。你现在有个结构的概览和运用ST语言的能力。



Automation Studio帮助文档包含所有这些结构体的说明。在使用算术功能和计算数学公式时，这种编程语言特别有用。

10 附录

10.1 数组

和原生数据类型不同的是，数组中包含了属于同一种类型的若干个变量。数组中的每个元素都可以通过数组名加上下标来访问。

使用数组下标访问数组元素时，下标的值一定不能超过该数组的实际大小。一个数组的大小是在定义变量的时候确定的。


在程序中，索引是一个确定值，一个变量，一个恒定的或枚举的元素。

Name	Typ	Wert
aPressure	INT[0..9]	
aPressure[0]	INT	123
aPressure[1]	INT	555
aPressure[2]	INT	0
aPressure[3]	INT	552
aPressure[4]	INT	32767
aPressure[5]	INT	9700
aPressure[6]	INT	0
aPressure[7]	INT	9
aPressure[8]	INT	0
aPressure[9]	INT	13

图片 20: 一个范围在0到9的数据类型为INT的数组相当于10个不同的数组元素。

声明以及使用数组

当一个数组被声明时，必须给予它一个数据类型和尺寸。 通常的惯例支持一个宿主的最小为0的索引值。 在这样的有10个元素的数组中最大索引值是9的情况下备注是重要的。


 声明	<pre>VAR aPressure : ARRAY[0..9] OF INT := [10(0)]; END_VAR</pre>
程序代码	<pre>#####123#####0####*# aPressure[0] := 123;</pre>

表格 31: 声明一个10元素数组，开始索引 = 0

若试图去访问一个带有索引10的数组元素，编译器输出一下错误信息：

程序代码	<pre>aPressure[10] := 75;</pre>
错误信息	<pre>Error 1230: The constant value '10' is not in range '0..9'.</pre>

表格 32: 给一个超过范围的数组范围

 如果要定义一个包含10个元素的数组，可以通过在编辑器中填入“USINT[0..9]亦或是USINT[10]来完成”。在使用这两种方式定义个数组时，都是为定义一个起始下标为0，结束下标为9的数组。

练习：创建“aPressure”数组

- 1) 在逻辑视图中添加新的“int_array”程序
- 2) 打开变量声明窗口。
- 3) 声明“aPressure”数组。


数组应包含10个元素。 最小的数组索引是0。 数据类型必须为INT。

- 4) 在程序代码中使用数组。

在程序代码中使用索引去访问数组。 可使用具体数字，常量和一个变量。

5) 在程序代码里强加一个无效的访问。

访问数组中为10的索引值然后分析在信息窗口的输出。



当分配aPressure[10] := 123;，编译器反馈以下的错误信息。

Error 1230: The constant value '10' is not in range '0..9'.


若用一个变量生成此分配，编译器不能检查数组访问。

```
index := 10;
aPressure[index] := 123;
```


用常量声明一个数组

自从在声明中使用具体的数值并且程序代码他自己通常通向难处理的以及难保持的编程，使用数字化的常量是很大的注意。

用三个常量能定义一个数组更高和更低的索引值。 这些常量之后能被用于程序代码中去限制变化的数组索引。

	<pre>VAR CONSTANT MAX_INDEX : USINT := 9; END_VAR VAR aPressure : ARRAY[0..MAX_INDEX] OF INT ; index : USINT := 0; END_VAR</pre>
程序代码	<pre>IF index > MAX_INDEX THEN index := MAX_INDEX; END_IF aPressure[index] := 75;</pre>

表格 33: 使用一个常量声明一个数组



程序代码现在已经被更新以至于被用来访问数组的索引值被限制到数组的最大索引值。 一个这个的优点是数组能调整其大小（更大或更小）而没必要在编程成宿代码中做出改变。



Programming \ Variables and data types \ Data types \ Derived data types \ Arrays

练习：计算总数和平均值。


应在“aPressure”数组的内容里计算平均值。 程序必须按如此的方式被创建，此方式就是当修正数组大小时有必要给予程序最少的改变。

1) 用一个循环计算总数。

确定的数值也许不能再程序代码里使用。

2) 计算平均值

此数据类型的平均值必须跟此数据类型的数组（INT）一样。



一个已经被使用去为一个数组变量声明而决定数组大小的常量也能作为循环结束值而使用。当生成此平均值时，相同的常量也能被用来做除法。当添加单一数组元素时，一个比输出数据类型（比如DINT）更大的数据类型必定被使用。使用一个明确的数据类型转换，结果值能被转换回INT数据类型。


多维数组

数组也由几个维度组成。既然这样声明和使用可以是这个样子：

	声明	<pre>## Array2Dim : ARRAY [0..6,0..6] OF INT; END_VAR</pre>	
	程序代码	<pre>(*Zählweise mit 0 beginnend*) Array2Dim[2,2] := 11;</pre>	

图片 21: 访问在第三行第三列的值

表格 34: 声明和访问一个7x7二维数组



用一个具体的数访问在程序代码的一个数组会成为无效的尝试，常量或枚举类型元素将被编译器探测以及阻止。

用一个变量去访问在程序代码的一个数组会成为无效的尝试，这不能被编译器探测到而且可能导致在runtime的存储错误。通过限制数组索引到有效范围能避免runtime错误。

IEC Check 库能被引入到一个AS项目有助于查找runtime错误。



Programming \ Libraries \ IEC Check library

10.2 练习题参考解答

练习：灯光控制

声明:	<pre>VAR btnLightOn: BOOL; btnLightOff: BOOL; doLight: BOOL; END_VAR</pre>
程序代码:	<pre>doLight := (btnLightOn OR doLight) AND NOT bntLightOff;</pre>

表格 35: 开关，带锁存器的继电器

练习：水族箱

声明:	<pre>VAR aiTemp1 : INT; aiTemp2 : INT; aoAvgTemp : INT; END_VAR</pre>
程序代码:	<pre>aoAvgTemp := DINT_TO_INT((INT_TO_DINT(aiTemp1) + aiTemp2) / 2);</pre>

表格 36: 在相加之前和除法之后要添加类型转换的处理

练习：气象站 - 第一部分

声明:	<pre>VAR aiOutside : INT; sShowText : STRING[80]; END_VAR</pre>
程序代码:	<pre>IF aiOutside < 18 THEN sShowText := 'Cold'; ELSIF (aiOutside >= 18) AND (aiOutside <= 25) THEN sShowText := 'Opt'; ELSE sShowText := 'Hot'; END_IF;</pre>

表格 37: IF statement

练习：气象站 - 第二部分

声明:	<pre>VAR aiOutside : INT; aiHumidity: INT; sShowText : STRING[80]; END_VAR</pre>
程序代码:	<pre>IF aiOutside < 18 THEN sShowText := 'Cold'; ELSIF (aiOutside >= 18) AND (aiOutside <= 25) THEN IF (aiHumid >= 40) AND (aiHumid <= 75) THEN sShowText := 'Opt'; ELSE sShowText := 'Temp. Ok'; END_IF ELSE sShowText := 'Hot'; END_IF;</pre>

表格 38: IF语句嵌套

练习：水箱

声明:	<pre>VAR aiLevel : INT; PercentLevel : UINT; doLow : BOOL; doOk : BOOL; doHigh : BOOL; doAlarm : BOOL; END_VAR</pre>
-----	--

表格 39: 用于查询值和范围的case状态

程序代码:

```
(*scaling the analog input to percent*)
PercentLevel := INT_TO_UINT(aiLevel / 327);
(*reset all outputs*)
doAlarm := FALSE;
doLow := FALSE;
doOk := FALSE;
doHigh := FALSE;

CASE PercentLevel OF
  0:      (*-- level alarm*)
    doAlarm := TRUE;
    doLow := TRUE;
  1..24: (*-- level is low*)
    doLow := TRUE;
  25..90: (*-- level is ok*)
    doOk := TRUE;
  ELSE    (*-- level is high*)
    doHigh := TRUE;
END_CASE
```

表格 39: 用于查询值和范围的case状态

练习: 终值搜寻

声明:

```
VAR CONSTANT
  MAXNUMBERS : UINT := 99;
END_VAR
VAR
  aNumbers : ARRAY[0..MAXNUMBERS] OF INT;
  nCnt : INT;
END_VAR
```

程序代码:

```
nCnt := 0;
REPEAT
  IF aNumbers[nCnt] = 10 THEN
    (*found the number 10*)
    EXIT;
  END_IF
  nCnt := nCnt + 1;
UNTIL nCnt > MAXNUMBERS
END_REPEAT
```

表格 40: REPEAT状态, 终止搜索结果和限制循环数

练习：起重机总负载

声明:	<pre>VAR CONSTANT MAX_INDEX: USINT := 4; END_VAR VAR aWeights : ARRAY[0..MAX_INDEX] OF INT; iCnt : USINT; sumWeight : DINT; avgWeight : INT; END_VAR</pre>
程序代码:	<pre>sumWeight := 0; FOR iCnt := 0 TO MAX_INDEX DO sumWeight := sumWeight + aWeights[iCnt]; END_FOR avgWeight := DINT_TO_INT (sumWeight / (MAX_INDEX + 1));</pre>

表格 41: FOR 语句，相加重量

练习：Function block

声明:	<pre>VAR TON_Test : TON; CTU_Test : CTU; diSwitch : BOOL; diCountImpuls : BOOL; diReset : BOOL; END_VAR</pre>
程序代码:	<pre>TON_Test(IN := diSwitch, PT := T#5s); CTU_Test(CU := diCountImpuls, RESET := diReset);</pre>

表格 42: 调用TON 和CTU

练习：电梯

声明：

```

VAR CONSTANT
    WAIT : UINT := 0;
    TOP_POSITION : UINT := 1;
    BOTTOM_POSITION : UINT := 2;
    GETBOX : UINT := 3;
    UNLOAD_POSITION : UINT := 4;
    UNLOAD_BOX : UINT := 5;
END_VAR

VAR
    (*-- digital outputs*)
    doConvTop: BOOL;
    doConvBottom: BOOL;
    doConvLift: BOOL;
    doLiftTop: BOOL;
    doLiftBottom: BOOL;
    doLiftUnload: BOOL;
    (*-- digital inputs*)
    diConvTop: BOOL;
    diConvBottom: BOOL;
    diLiftTop: BOOL;
    diLiftBottom: BOOL;
    diLiftUnload: BOOL;
    diBoxLift: BOOL;
    (*-- status variables*)
    selectLift: UINT;
    ConvTopOn: BOOL;
    ConvBottomOn: BOOL;
END_VAR

```

表格 43: 练习举盒t的解决方案

程序代码:

```
doConvTop := NOT diConvTop OR ConvTopOn;
doConvBottom := NOT diConvBottom OR ConvBottomOn;
CASE selectLift OF
  (*-- wait for request*)
  WAIT:
    IF (diConvTop = TRUE) THEN
      selectLift := TOP_POSITION;
    ELSIF (diConvBottom = TRUE) THEN
      selectLift := BOTTOM_POSITION;
    END_IF

  (*-- move lift to top position*)
  TOP_POSITION:
    doLiftTop := TRUE;
    IF (diLiftTop = TRUE) THEN
      doLiftTop := FALSE;
      ConvTopOn := TRUE;
      selectLift := GETBOX;
    END_IF
```

表格 43: 练习举盒t的解决方案

```

(*-- move lift to bottom position*)
BOTTOM_POSITION:
    doLiftBottom := TRUE;
    IF (diLiftBottom = TRUE) THEN
        doLiftBottom := FALSE;
        ConvBottomOn := TRUE;
        selectLift := GETBOX;
    END_IF

(*-- move box to lift*)
GETBOX:
    doConvLift := TRUE;
    IF (diBoxLift = TRUE) THEN
        doConvLift := FALSE;
        ConvTopOn := FALSE;
        ConvBottomOn := FALSE;
        selectLift := UNLOAD_POSITION;
    END_IF

(*-- move lift to unload position*)
UNLOAD_POSITION:
    doLiftUnload := TRUE;
    IF (diLiftUnload = TRUE) THEN
        doLiftUnload := FALSE;
        selectLift := UNLOAD_BOX;
    END_IF

(*-- unload the box*)
UNLOAD_BOX:
    doConvLift := TRUE;
    IF (diBoxLift = FALSE) THEN
        doConvLift := FALSE;
        selectLift := WAIT;
    END_IF
END_CASE

```

表格 43: 练习举盒t的解决方案

练习：实施分散混合器

变量声明

```

(*****
* COPYRIGHT -- Bernecker + Rainer
*****
* Program: mixer
* File: mixer.var
* Author: Academy
* Created: February 04, 2015
*****
* Local variables of program mixer
*****)

```

```

(*state machine variables*)
VAR
    sMixerState : enMixerStates := enWAIT_FOR_START;
END_VAR
(*Digital input signals*)
VAR
    gDiStart : BOOL := FALSE;
    gDiWaterOK : BOOL := FALSE;
    gDiSensFull : BOOL := FALSE;
    gDiSensLow : BOOL := FALSE;
END_VAR
(*Digital output signals*)
VAR
    gDoWaterValve : BOOL := FALSE;
    gDoColorValve : BOOL := FALSE;
    gDoMixer : BOOL := FALSE;
    gDoDrainPump : BOOL := FALSE;
    gDoDrainValve : BOOL := FALSE;
END_VAR
(*Function block instance variables*)
VAR
    TON_Mixer : TON;
END_VAR

```

循环程序段

```

(*****
 * COPYRIGHT -- Bernecker + Rainer
 *****)
* Program: mixer
* File: mixerCyclic.st
* Author: Academy
* Created: February 04, 2015
*****
* Implementation of program mixer
*****
)

PROGRAM _CYCLIC

    (*reset all outputs - will be set in the individual states*)
    gDoWaterValve := FALSE;
    gDoColorValve := FALSE;
    gDoMixer := FALSE;
    gDoDrainValve := FALSE;
    gDoDrainPump := FALSE;
    TON_Mixer.IN := FALSE;

    (* implementation of dispersion mixer state machine *)
    CASE sMixerState OF
        (*wait until operator starts process by start button*)
        enWAIT_FOR_START:
            IF gDiStart = TRUE THEN
                sMixerState := enFILL_WATER;
            END IF;
    END CASE

```

```

        END_IF

        (*fill water into the reservoir until limit is reached*)
    enFILL_WATER:
        IF gDiWaterOK = TRUE THEN
            sMixerState := enFILL_COLOR;
        END_IF

        gDoWaterValve := TRUE;

        (*add color and mix dispersion until reservoir is full*)
    enFILL_COLOR:
        IF gDiSensFull = TRUE THEN
            sMixerState := enMIX_TIME;
        END_IF

        gDoColorValve := TRUE;
        gDoMixer := TRUE;

        (*mix color and water until time is elapsed*)
    enMIX_TIME:

        TON_Mixer.IN := TRUE;
        TON_Mixer.PT := T#30s;

        IF TON_Mixer.Q = TRUE THEN
            sMixerState := enBOTTLE_DISPERSION;
        END_IF

        gDoMixer := TRUE;

        (*continue mixing and bottle dispersion until reservoir is empty*)
    enBOTTLE_DISPERSION:
        IF gDiSensLow = TRUE THEN
            sMixerState := enWAIT_FOR_START;
        END_IF

        gDoDrainValve := TRUE;
        gDoDrainPump := TRUE;
        gDoMixer := TRUE;

    END_CASE

    (*call timer function block*)
    TON_Mixer();

END_PROGRAM

```

Offered by the Automation Academy

自动化学院为我们的客户以及我们自己的员工提供有针对性的培训课程。

在自动化学院中，你可以在短时间内发展你所需要的技能！

我们的培训可以使你提高在自动化工程设计方面的专业知识。

一旦完成培训，你将能够采用贝加莱技术实现高效的自动化解决方案。这将有可能使您取得决定性的竞争优势，让您和您的企业更快地应对不断变化的市场需求。



培训



质量和相关性是我们培训必不可少的组成部分。培训步伐是严格按照课程参与者所带来的经验进行并为他们的需求量身定制的。小组学习和自学相结合为最大限度地提高学习经验提供了所需的高灵活性。每一次培训都由我们经验丰富的培训师教授。

培训手册

我们的培训手册为培训课程和自学提供了基本学习资料。这些紧凑的手册都是基于各个不同的话题的。自下而上的结构使你能够快速高效的学习复杂的相关话题。他们提供了帮助系统以外的最好的学习方式。培训手册pdf版可以从官网下载，也可以向总部下单购买可以打印的版本。

话题目录：

- ⇒ 控制技术
- ⇒ 运动控制技术
- ⇒ 安全技术
- ⇒ 人机界面
- ⇒ 过程控制
- ⇒ 诊断与服务
- ⇒ POWERLINK 和 openSAFETY

ETA 系统



ETS系统为教育、职业培训和实验室提供了一个现实的构造。有两种不同的基本机械结构可供选择。轻便型ETA系统具有高移动性，节省空间，适合实验室使用。标准ETA系统具有坚固的机械结构，并且包含完全接线的传感器和执行机构。

发现更多！

你想要一个额外的培训吗？你有兴趣了解贝加莱培训学员还能提供什么吗？你来对地方了。

更多详情可以在如下链接找打：

www.br-automation.com/academy





TM246TRE.00-ENG

V1.0.8.11 ©2019/02/18 by B&R, All rights reserved.
All registered trademarks are the property of their respective owners.
We reserve the right to make technical changes.