ST 语言编程手册

目录

1. ST　基本原理

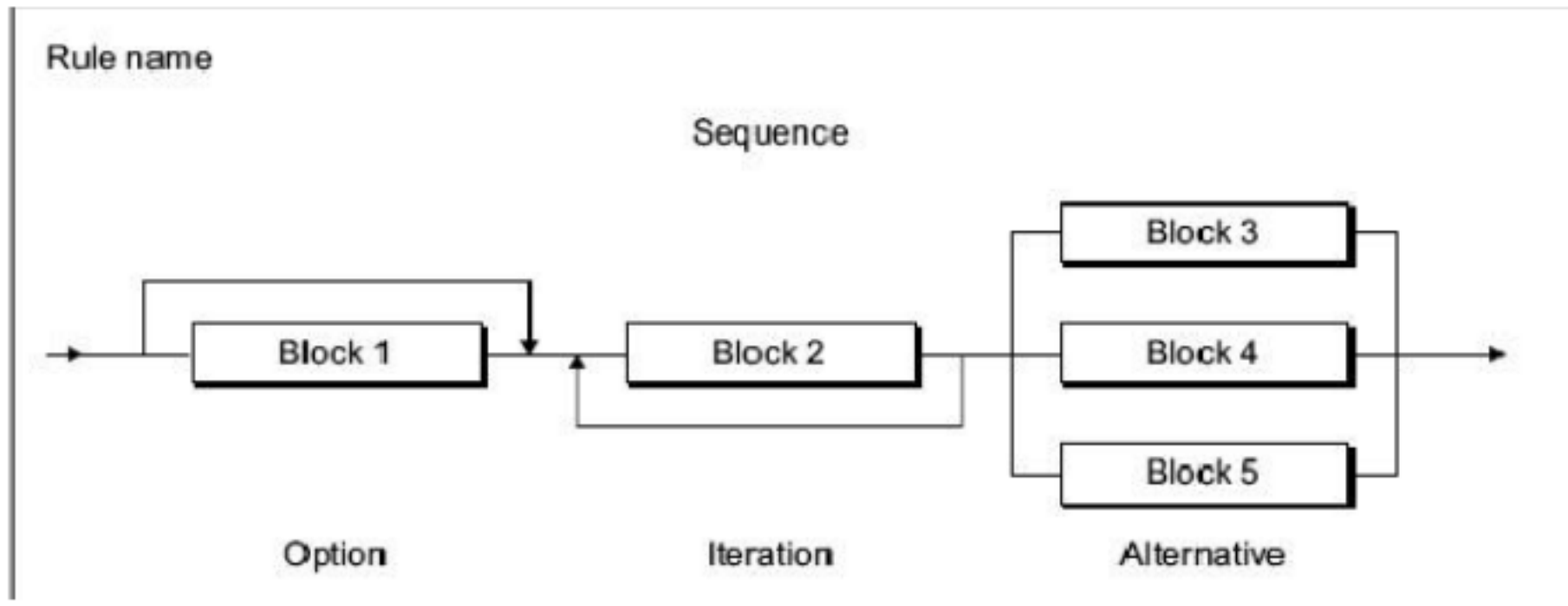
此章节描述了　ST中的语言资源和使用方法。请注意此章节中描述了功能、功能块和任务控制系统。一个完整正式的语言描述包括语法图。

1.1 语言描述

在本手册的下列章节中语法图用作语言描述的基础，　　为你提供了对　ST语法结构的新认识。

1.1.1　语法图

语法图是对语法结构的图形式的阐述。结构是由一系列的规则描述组成。可以基于现有的规则生成新的规则。



上图中的语法图需从左至右读。需要注意下列规则结构：

●序列：块的序列

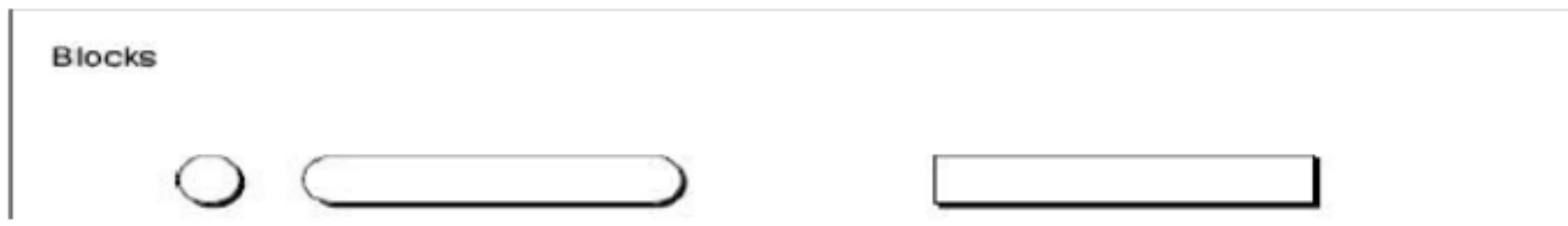
●选项：可以跳过的语句

●迭代：一个或多个语句的重复

●替代： Branch

1.1.2　语法图中的块

块是基本元素。下图显示了代表块而使用的符号类型。



要更多解释的基本元素　　由更多的语法图解释的复合单元

当输入源文本时需要注意格式化规则和非格式化规则。比如：把语法图中的块或元素转化为源文本（见帮助中的语言描述，第　291 页）

1.1.3　规则的意义（语义）

规则能仅代表语言的格式结构。意义（语义）一般不明显。由于这个原因，如果意义很关键，则需要在规则旁边写上额外的信息。

●如果同类型的元素意义不同，需要附上额外的命名。例如，额外指定每个十进制字符串元素—年，月或日（见　308 页）。名称表示了用途

● 重要的限制也列出。 如：整数规则中对于— （负号），标明负号只能在　SINT, INT, and DINT 数据类型之前出现（见　308 页）

1.2 基本元素的语言

ST语言的基本元素包括　ST字符组，根据 ST字符组（如：语言命令）创建的预留标识符，自定义的标识符和数字。

ST 字符组合预留标识符都是基本的元素（　terminals），因为是无需另外的规则来描述。自定义的标识符和数字不是　terminals，因为它们是由另外的规则来描述。

在语法图中，终端是由圆形或者椭圆形符号来表示，但是复合单元由长方形来表示（见语法图中的块，第　72 页）。下列内容是主要终端的节选，作为全局预览，参考基本元素（terminals，第 294 页）

1.2.1 ST　字符组

ST使用 ASCII字符组中的下列字母和数字

●从 A 到 Z 的大小字母

●从 0 到 9 的阿拉伯数字

字母和数字是最常用的字符。例如，标识符是字母，数字和下划线的组合。下划线是特殊的字符之一。

特殊字符在 ST中有特定的含义（见第 291 页的正式语言描述和第 294 页的基本元素）

1.2.2 ST 中的标识符

标识符的 ST 中的名称。这些名称可以根据系统来定义，例如语言命令等。但是，名称可以是用户定义，比如常量、变量或功能。

1.2.2.1 标识符的规则

标识符是由字母（ A 到 Z，a 到 z）、数字（ 0 到 9）或单独的下划线随意组成，但是首字符必须是字母或者下划线。大小写字母没有区分（比如，在编辑器中 Anna 和 AnNa 是一样的）。

一个正式的标识符可以由以下的语法图表示：

命名时，最好选用唯一的、有意义的命名，以便解释程序。

图表中的语法图声明了一个标识符的首字符必须是字母或者下划线。下划线必须跟着字母或数字。如：不允许连续有两根或以上的下划线。下划线可以跟着任意或者一系列的数字，下划线或字母。在这唯一的例外是两条下划线可能不会同时出现。

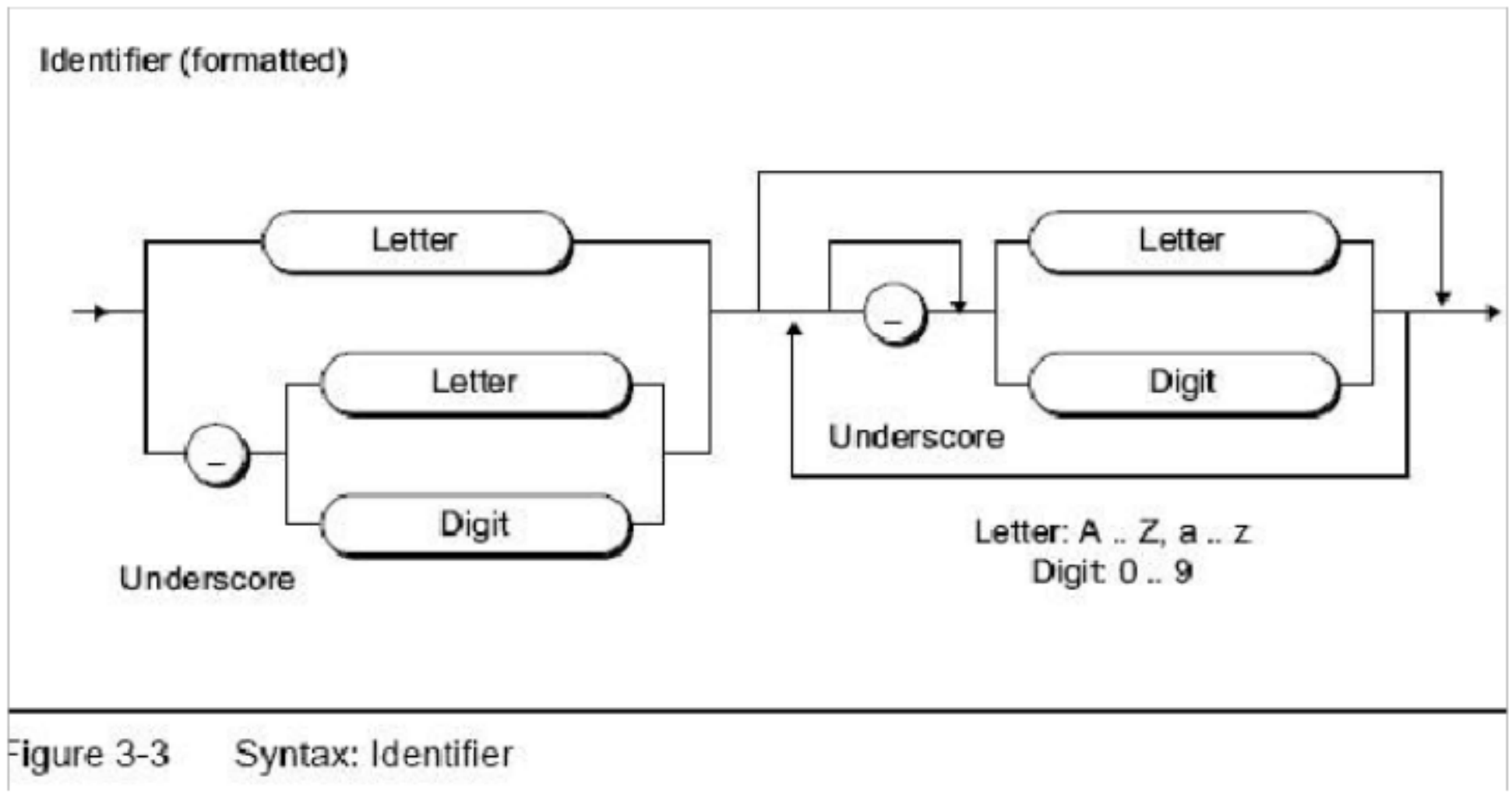


Figure 3-3 Syntax: Identifier

1.2.2.2 标识符举例

有效的标识符

```
x                y12              _sum             temperature      R_CONTROLLER3
name             area             myFB             table
```

无效的标识符

无效标识符	原因
4ter	第一次字符必须为一个字母或者下划线
*#AB	不允许特殊字符（除了下划线）
RR__20	不允许有两个下划线
S value	不允许出现空格，因为是特殊字符
Array	虽然 ARRAY是一个正式有效的标识符，但是它是一个预留标识符。 ，只能做预先定义使用。这意味着你不能使用这个名称， 比如：变量

标识符不能用作
绝不定义标识符：
●与预留标识符一样
●与任务命名相匹配

注意：
如有可能，避免定义由下划线， struct,enum,或者 command 开始的标识符

虽然这些是有效的标识符，当你下载技术包时可能导致出现错误。在基础系统和在技术包中命令词语，参数或数据类型以这些字符开始。

1.2.3 预留标识符

预留标识符可能和预先定义的用途不一样。你不能用预留标识符的名称来命名一个变量或者数据类型。

符号的大小写无区别。

所有标识符的预先定义的意义可以在 SIMOTION基本功能功能手册中找到：

- ST编程语言中的保护或者预留标识符，欲知更多，请见 76 页和 81 页
- 标准功能和功能定义的数据类型，欲知更多，请见 251 页
- 系统的常规功能块
- SIMOTION设备系统功能、系统变量和数据类型
- 技术目标的系统功能、系统变量和数据类型

1.2.3.1 保护标识符

ST语言中的保护标识符全列在了下表中。
欲见所有预留词的简短描述, 请见附录预留词语 （第 299 页）和规则附录中的语法图 （第 307 页）

Table 3-1 Protected identifiers in ST programming language

A	
ABS	ANYTYPE_TO_LITTLEBYTEARRAY
ACOS	ARRAY
AND	AS
ANYOBJECT	ASIN
ANYOBJECT_TO_OBJECT	AT
ANYTYPE_TO_BIGBYTEARRAY	ATAN
B	
BIGBYTEARRAY_TO_ANYTYPE	BY
BOOL	BYTE
BOOL_TO_BYTE	BYTE_TO_BOOL
BOOL_TO_DWORD	BYTE_TO_DINT
BOOL_TO_WORD	BYTE_TO_DWORD
BOOL_VALUE_TO_DINT	BYTE_TO_INT
BOOL_VALUE_TO_INT	BYTE_TO_SINT
BOOL_VALUE_TO_LREAL	BYTE_TO_UDINT
BOOL_VALUE_TO_REAL	BYTE_TO_UINT
BOOL_VALUE_TO_SINT	BYTE_TO_USINT
BOOL_VALUE_TO_UDINT	BYTE_TO_WORD
BOOL_VALUE_TO_UINT	BYTE_VALUE_TO_LREAL
BOOL_VALUE_TO_USINT	BYTE_VALUE_TO_REAL
C	
CASE	CTD_UDINT
CONCAT	CTU
CONCAT_DATE_TOD	CTU_DINT
CONSTANT	CTU_UDINT
COS	CTUD
CTD	CTUD_DINT
CTD_DINT	CTUD_UDINT

D	
DATE	DO
DATE_AND_TIME	DT
DATE_AND_TIME_TO_DATE	DT_TO_DATE
DATE_AND_TIME_TO_TIME_OF_DAY	DT_TO_TOD
DELETE	DWORD
DINT	DWORD_TO_BOOL
DINT_TO_BYTE	DWORD_TO_BYTE
DINT_TO_DWORD	DWORD_TO_DINT
DINT_TO_INT	DWORD_TO_INT
DINT_TO_LREAL	DWORD_TO_REAL
DINT_TO_REAL	DWORD_TO_SINT
DINT_TO_SINT	DWORD_TO_UDINT
DINT_TO_STRING	DWORD_TO_UINT
DINT_TO_UDINT	DWORD_TO_USINT
DINT_TO_UINT	DWORD_TO_WORD
DINT_TO_USINT	DWORD_VALUE_TO_LREAL
DINT_TO_WORD	DWORD_VALUE_TO_REAL
DINT_VALUE_TO_BOOL	
E	
ELSE	END_REPEAT
ELSIF	END_STRUCT
END_CASE	END_TYPE
END_EXPRESSION	END_VAR
END_FOR	END_WAITFORCONDITION
END_FUNCTION	END_WHILE
END_FUNCTION_BLOCK	ENUM_TO_DINT
END_IF	EXIT
END_IMPLEMENTATION	EXP
END_INTERFACE	EXPD
END_LABEL	EXPRESSION
END_PROGRAM	EXPT
F	
F_TRIG	FOR
FALSE	FUNCTION
FIND	FUNCTION_BLOCK
G	
GOTO	

I	
IF	INT_TO_SINT
IMPLEMENTATION	INT_TO_TIME
INSERT	INT_TO_UDINT
INT	INT_TO_UINT
INT_TO_BYTE	INT_TO_USINT
INT_TO_DINT	INT_TO_WORD
INT_TO_DWORD	INT_VALUE_TO_BOOL
INT_TO_LREAL	INTERFACE
INT_TO_REAL	
L	
LABEL	LREAL_TO_REAL
LEFT	LREAL_TO_SINT
LEN	LREAL_TO_STRING
LIMIT	LREAL_TO_UDINT
LITTLEBYTEARRAY_TO_ANYTYPE	LREAL_TO_UINT
LN	LREAL_TO_USINT
LOG	LREAL_VALUE_TO_BOOL
LREAL	LREAL_VALUE_TO_BYTE
LREAL_TO_DINT	LREAL_VALUE_TO_DWORD
LREAL_TO_INT	LREAL_VALUE_TO_WORD
G	
MAX	MOD
MID	MUX
MIN	
N	
NOT	
O	
OF	OR
P	
PROGRAM	

R	
R_TRIG	REAL_VALUE_TO_BYTE
REAL	REAL_VALUE_TO_DWORD
REAL_TO_DINT	REAL_VALUE_TO_WORD
REAL_TO_DWORD	REPEAT
REAL_TO_INT	REPLACE
REAL_TO_LREAL	RETAIN
REAL_TO_SINT	RETURN
REAL_TO_STRING	RIGHT
REAL_TO_TIME	ROL
REAL_TO_UDINT	ROR
REAL_TO_UINT	RS
REAL_TO_USINT	RTC
REAL_VALUE_TO_BOOL	
S	
SEL	SINT_TO_WORD
SHL	SINT_VALUE_TO_BOOL
SHR	SQRT
SIN	SR
SINT	STRING
SINT_TO_BYTE	STRING_TO_DINT
SINT_TO_DINT	STRING_TO_LREAL
SINT_TO_DWORD	STRING_TO_REAL
SINT_TO_INT	STRING_TO_UDINT
SINT_TO_LREAL	STRUCT
SINT_TO_REAL	StructAlarmId
SINT_TO_UDINT	STRUCTALARMID_TO_DINT
SINT_TO_UINT	StructTaskId
SINT_TO_USINT	
T	
TAN	TOD
THEN	TOF
TIME	TON
TIME_OF_DAY	TP
TIME_TO_INT	TRUE
TIME_TO_REAL	TRUNC
TO	TYPE

U	
UDINT	UINT_TO_UDINT
UDINT_TO_BYTE	UINT_TO_USINT
UDINT_TO_DINT	UINT_TO_WORD
UDINT_TO_DWORD	UINT_VALUE_TO_BOOL
UDINT_TO_INT	UNIT
UDINT_TO_LREAL	UNTIL
UDINT_TO_REAL	USELIB
UDINT_TO_SINT	USEPACKAGE
UDINT_TO_STRING	USES
UDINT_TO_UINT	USINT
UDINT_TO_USINT	USINT_TO_BYTE
UDINT_TO_WORD	USINT_TO_DINT
UDINT_VALUE_TO_BOOL	USINT_TO_DWORD
UINT	USINT_TO_INT
UINT_TO_BYTE	USINT_TO_LREAL
UINT_TO_DINT	USINT_TO_REAL
UINT_TO_DWORD	USINT_TO_SINT
UINT_TO_INT	USINT_TO_UDINT
UINT_TO_LREAL	USINT_TO_UINT
UINT_TO_REAL	USINT_TO_WORD
UINT_TO_SINT	USINT_VALUE_TO_BOOL
V	
VAR	VAR_OUTPUT
VAR_GLOBAL	VAR_TEMP
VAR_IN_OUT	VOID
VAR_INPUT	
W	
WAITFORCONDITION	WORD_TO_INT
WHILE	WORD_TO_SINT
WITH	WORD_TO_UDINT
WORD	WORD_TO_UINT
WORD_TO_BOOL	WORD_TO_USINT
WORD_TO_BYTE	WORD_VALUE_TO_LREAL
WORD_TO_DINT	WORD_VALUE_TO_REAL
WORD_TO_DWORD	
X	
XOR	

3.2.3.2 额外的预留标识符

下表包含了将来扩展所有的预留标识符。

Table 3-2 Additional reserved identifiers of the ST language

A	
ACTION ADD ADD_DT_TIME	ADD_TIME ADD_TOD_TIME
B	
BCD_TO_BYTE BCD_TO_DINT BCD_TO_DWORD BCD_TO_INT	BCD_TO_LWORD BCD_TO_SINT BCD_TO_WORD BYTE_TO_BCD
C	
CONFIGURATION CTD_LINT CTD_ULINT CTU_LINT	CTU_ULINT CTUD_LINT CTUD_ULINT
D	
DINT_TO_BCD DIV	DIVTIME DWORD_TO_BCD
E	
EN END_ACTION END_CONFIGURATION END_RESOURCE	END_STEP END_TRANSITION ENO EQ
F	
F_EDGE	FROM
G	
GE	GT
I	
INITIAL_STEP	INT_TO_BCD
L	
LE LINT PM	LWORD LWORD_TO_BCD
G	
MUL	MULTIME
N	
MS	
R	
R_EDGE	RESOURCE

S	
SEMA SINT_TO_BCD STEP SUB SUB_DATE_DATE	SUB_DT_DT SUB_DT_TIME SUB_TIME SUB_TOD_TIME SUB_TOD_TOD
T	
TRANSITION	
U	
ULINT	
V	
VAR_ACCESS VAR_ALIAS	VAR_EXTERNAL VAR_OBJECT
W	
WORD_TO_BCD	

1.2.4 数字和布尔值

在 ST中可以通过多种方式编写数字。 数字可以包括一个符号、 一个小数点或者一个指数。

下列规则适用于所有的数字：

●数字中不能出现逗号和空格

●下划线允许作为视觉上的分隔线

●数字可以冠以正号（ +）或者负号（ —），如果没有使用符号则认为数字为正。

●数字不能超过确定的最大值或最小值

1.2.4.1 整数

整数既不包括小数点也不包括指数。一个整数是一系列的数字，也可以在前面加上一个符号。

下列为有效的整数：

0　　1　　+1　　-1

743　-5280　60 000　-32 211 321

下列整数为无效，并且已经列出原因：

123,456　不允许有逗号

36.　整数不能包括小数点

10　20　30　不允许有空格

在 ST中，你可以使用不同的编号体系来表示整数。通过插入编号系统的关键字前缀来实

现。

●2#为二进制

●8#为八进制

●16#为十六进制

十进制小数 15 的有效表示方式为：

2#1111 8#17 16#F

1.2.4.2 浮点数

下列为有效的浮点数

0.0 1.3 -0.2 827.602

0000.0 +0.000743 60_000.15 -315.0066

下列为无效的浮点数

1. 数字必须出现在小数点之前或之后

1,000.0 不允许有逗号

1.333.333 不允许有两个小数点

1.2.4.3 指数

指数可以用来定义小数点的位置。如果没有出现小数点，我们就假设位于数字的右侧。指数必须为正整数或负整数。基数 10 用字母 E来表示。

3×10^8 在 ST中可以通过下列正确的浮点数来表示：

3.0E+8 3.0E8 3e+8 3E8 0.3E+9

0.3e9 30.0E+7 30e7

下列浮点数是无效的：

3.E+8 数字必须出现在小数点之前或之后

8e2.3 指数必须为一个整数

.333e-3 数字必须出现在小数点之前或之后

30 E8 不允许有空格

1.2.4.4 布尔值

布尔值是恒定常量。必须通过 0 或者 1， TRUE或者 FALSE来表示。

Example:

```
a := 1;          // is equivalent a := TRUE
b := FALSE; // is equivalent to b := 0
```

1.2.4.5 数字的数据类型

编辑器根据数值和使用来自动选择适合数字的基本数据类型。

也可以直接指定数据类型。在数字前面输入数据类型（数字数据类型）和字符“　　#”。

Examples:

```
INT#255                INT#16#FF              INT#8#377
WORD#255               WORD#16#FF             WORD#8#377
REAL#255               REAL#16#FF             REAL#8#377
REAL#255.0             REAL#2.55E2            LREAL#255.0
```

1.2.5　字符串

什么是字符串？

一个字符串是许多的　0 或在最前面或者最后面带撇号的多字符。每个字符在串中

一个字符可以如下输入：

●可打印的字符（ ASCII code $20 to $7E, $80 to $FF），除了美元符号 (ASCII code $24)和撇号 (ASCII code $27)之外，因为这些符号在字符串中有着特殊的含义。

●美元符号 ($)之后的相关字符的　2 位的十六进制　ASCII code

●根据下表组成的两个字符的组合

Table 3-3 2-character combinations for special characters in strings

Character combination			Meaning
$$			Dollar sign $ ($24)
$'			Apostrophe ' ($27)
$L	or	$l	Line Feed LF ($0A)
$N	or	$n	Carriage Return + Line Feed CR + LF ($0D$0A)
$P	or	$p	Form Feed FF ($0C)
$R	or	$r	Carriage Return CR ($0D)
$T	or	$t	Horizontal tab (HT) ($09)

Examples:

''	Empty string (length 0).
'A'	String of length 1 containing the letter A.
' '	String of length 1 containing a blank.
'$''	String of length 1 containing an apostrophe.
'RL' '$0D$0A'	Two equivalent representations for a string of length 2 containing the characters CR and LF.
'$$1.00'	String of length 5 containing $1.00.
'Text$R$L'	String of length 6 containing the word Text followed by the characters CR and LF.
'ÄÖÜ' '$C4$D6$FC'	Two equivalent representations for a string of length 3 containing the German umlauts ÄÖü (A, O, u with diaresis).

1.3 ST 源文件的结构

一个 ST源包含连续的文本，通过划分为逻辑块形成文本。详细的规则见源文件章节（第 169 页）。

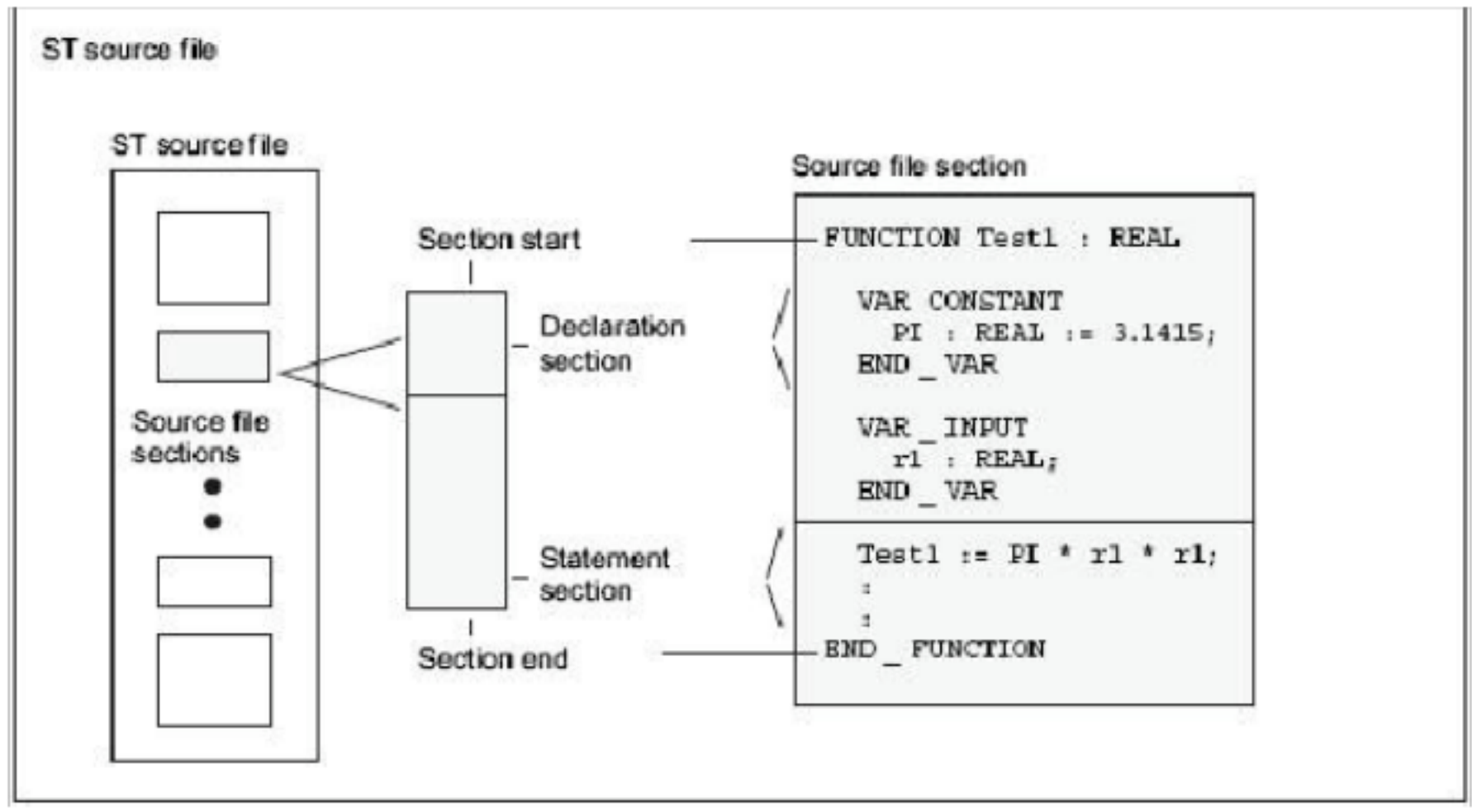
简单的总结如下：

●一个 ST 源文件是可以在项目中创建的一个逻辑单元，可以出现多次。通常被称为一个单元。

●一个 ST源文件的逻辑部分被称为 Section（见表格）

●一个用户程序是所有程序源的集合（如： ST 源文件， MCC 单元）

每个 ST源文件的逻辑部分的开头和结尾都有特定关键词



没有必要自己对每个功能编程，　可以使用 SIMOTION系统组件。这些是预先编程好的部分，如系统功能或技术目标的功能。

源文件部分	描述
单元语句（可选的）	包括 ST的名称
interface 部分	包括导入和导出变量的语句，类型和　POU
implementation 部分	包括 ST源文件的执行部分
POU（程序组织单元）	ST源文件（程序， 功能， 功能块） 单独的可执行部分
声明部分	包含声明（如变量和类型），可以被包含在 interface 部分和 implementation 部分，以及 POU中
语句部分	包含一个 POU的可执行语句

注意：
在线帮助中有很多可用的示范单元的模板。你可以使用作为一个新　　ST源文件的模板。

调用 ST编辑器帮助，点击相关链接。复制文本到 ST编辑器窗口，根据你的需求修改模板。

示范单元的模板包括此模板的复制件

1.3.1　语句

一个 POU 的语句部分包括重复的单独语句。跟着　　POU 的声明部分，以　POU 的结束而结束。首尾没有明显的关键词。

在 ST中有三种基本语句：

- 赋值：从表达式到变量的赋值，见 105 页变量声明
- 控制语句：语句的分支的重复，见 130 页控制
- 子程序执行：功能和功能块，见 147 页功能，功能块和程序

Table 3-5 Examples of statements

```
...
// Value assignment
 Status := 17;

// Control statement
 IF a = b THEN
     FOR c := 1 TO 10 DO
         b := b + c;
     END_FOR;
 END_IF;

// Function call
 retVal := Test1(10.0);
...
```

1.3.2 注释

注释用作编制文件，同时也帮助用户理解源文件部分。在编制后，对于程序执行没有任何意义。

注释有两种：

- 线注释
- 块注释

线注释由 // 开始。编辑器将进行跟随的文本，知道线注释结束

你可以在很多线之后输入一个块注释，如果 (首尾都是 *）先于它。

当插入注释时注意：

- 在注释中可以使用完整的扩展 ASCII字符集
- 在线型注释中可以忽略字符组 (* 和 *)
- 不允许块注释的嵌套。但是，在块注释中你可以嵌套线注释。
- 可以在任意位置插入注释，但是不能在保持的规则中插入，如标识符的名称中。欲知更多规则信息，见 291 页语言描述源。

Table 3-6 Examples of comments

```
// This is a one-line comment.
a := 5;

// This is an example of a one-line comment
// used several times in succession.
b := 23;

(* The above example is easier to edit as a
   multi-line comment.
 *)
c := 87;
```

1.4 数据类型

使用数据类型来定义如何在程序源中使用变量或常量值。

下列数据类型对用户是可用的：

- 基本数据类型
- 用户定义的数据类型（ UDT）
  - 简单的导数
  - 阵列
  - 枚举
  - 结构
- 技术目标数据类型
- 系统数据类型

见 基本数据类型（第 90 页）
技术目标数据类型（第 101 页）
系统数据类型（第 104 页）

1.4.1 基本数据类型

1.4.1.1 基本数据类型

基本数据类型定义了不能分成更小的单元的数据结构。一个基本数据类型描述了有固定长度的存储区域，代表了数字数据、整数、浮点值、时间、日期和字符串。

所有的基本数据类型在下表中列出：

类型	预留词	位宽度	值的范围
位数据类型： 此类型的数据使用 1 位、8 位、16 位或 32 位。此数据类型的变量初始值为 0			
位	BOOL	1	0, 1 or FALSE, TRUE
字节	BYTE	8	16#0 to 16#FF
字	WORD	16	16#0 to 16#FFFF
双字	DWORD	32	16#0to 16#FFFF_FFFF
数字类型： 此类数据类型可用于处理数值。此数据类型的变量初始值为 0（全为整数）或 0.0（全为浮点数）			
短整数	SINT	8	-128 to 127 (-2**7 to 2**7-1)
无符号短整数	U SINT	8	0 to 255 (0 to 2**8-1)
整数	INT	16	-32_768 to 32_767 (-2**15 to 2**15-1)
无符号整数	UINT	16	0 to 65_535 (0 to 2**16-1)
双整数	DINT	32	-2_147_483_648 to 2_147_483_647 (-2**31 to 2**31-1)
无符号双整数	UDINT	32	0 to 4_294_96_7295 (0 to 2**32-1)
浮点数 (per IEEE-754)	REAL	32	-3.402_823_466E+38to 1.175_494_351E-38, 0.0, +1.175_494_351E - 38 to +3.402_823_466E+38 精度：23 位尾数（对应 6 位小数），8 位指数， 1 位字符
长浮点数（IEEE-754）	LREAL	64	-1.797_693_134_862_315_8E+308 to -2.225_073_858_507_201_4E -308, 0.0, +2.225_073_858_507_201_4E- 308 to+1.797_693_134_862_315_8E+308 精度：52 位尾数（对应 15 位小数），11 位指数， 1 位字符
时间类型：此类数据用于表示不同的时间或日期			
1 分钟的增量的持续时间	TIME	32	T#0d_0h_0m_0s_0msto T#49d_17h_2m_47s_295ms 天，小时，分钟的数值最多为 2 位。里程碑最多为 3 位。初始为 T#0d_0h_0m_0s_0ms
1 天的增量的日期	DATE	32	D#1992-01-01 to D#2200-12-31 需要考虑闰年，年份为 4 位，月份和天数为 2 位。初始为 D#0001-01-01
分钟为单位的当日时间	TIME_OF_DAY(TOD)	64	TOD#0:0:0.0 to TOD#23:59:59.999 天，小时，分钟的数值最多为 2 位。里程碑最多为 3 位。初始为

			TOD#0:0:0.0
日期和时间	DATE_AND_TIME(DT)	64	DT#1992-01-01-0:0:0.0to DT#2200-12-31-23:59:59.999 日期和时间包括日期和时间的类型。初始为 DT#0001-01-01-0:0:0.0
串类型： 此类数据代表字符串，每个字符使用特定字节的数字来编码。 串的长度可以在声明中定义。用 "[" 和 "]" 来表示长度。如： STRING[100]。默认的设置包含 80 个字符。 指定（初始）字符的数字可以少于声明的长度。			
1 字节 / 字符的串	STRING	8	ASCIIcode $00 到 $F 的所有字符都是允许的。 默认 ’ 空字(符串)
注意：当导出变量到其他系统时，需要考虑对应的目标系统的数据类型的值的范围			

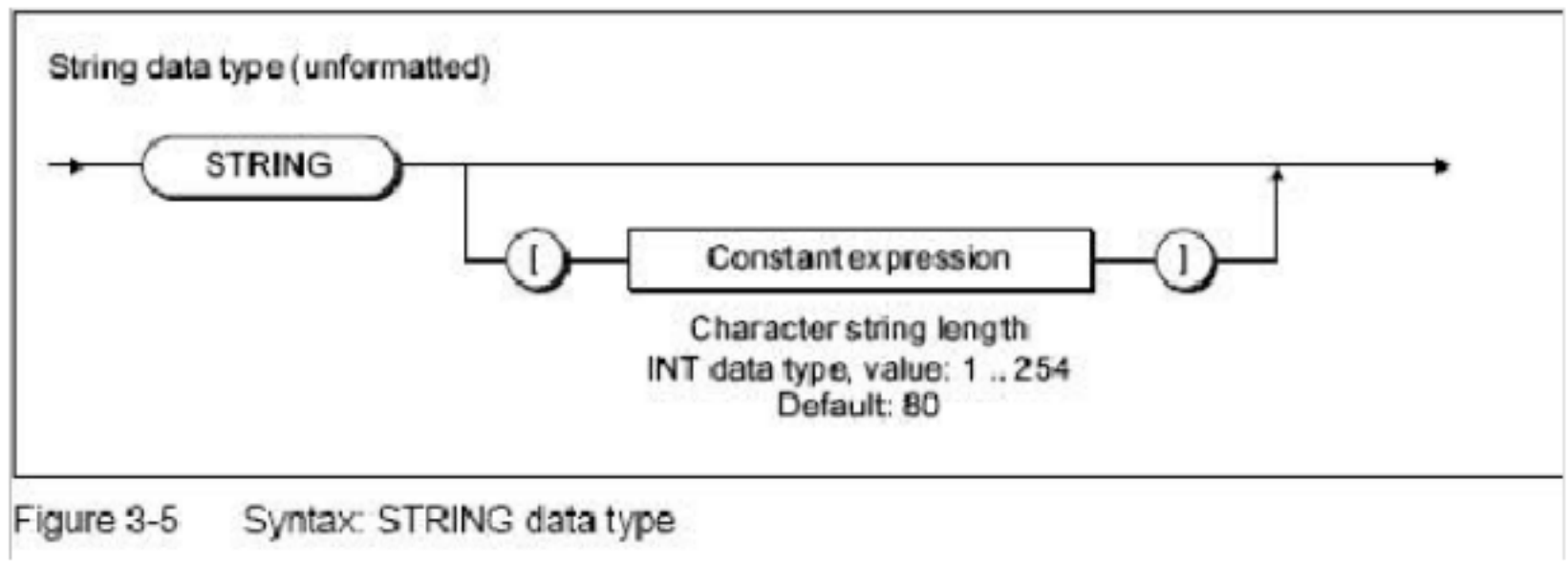


Figure 3-5 Syntax: STRING data type

1.4.1.2 基础数据类型的值的范围限制

基础数据类型的值的范围作为常量可用。

Table 3-8 Symbolic constants for the value range limits of elementary data types

Symbolic constant	Data type	Value	Hex notation
SINT#MIN	SINT	-128	16#80
SINT#MAX	SINT	127	16#7F
INT#MIN	INT	-32768	16#8000
INT#MAX	INT	32767	16#7FFF
DINT#MIN	DINT	-2147483648	16#8000_0000
DINT#MAX	DINT	2147483647	16#7FFF_FFFF
USINT#MIN	USINT	0	16#00
USINT#MAX	USINT	255	16#FF
UINT#MIN	UINT	0	16#0000
UINT#MAX	UINT	65535	16#FFFF
UDINT#MIN	UDINT	0	16#0000_0000
UDINT#MAX	UDINT	4294967295	16#FFFF_FFFF
T#MIN TIME#MIN	TIME	T#0ms	16#0000_0000[1]
T#MAX TIME#MAX	TIME	T#49d_17h_2m_47s_295ms	16#FFFF_FFFF[1]
TOD#MIN TIME_OF_DAY#MIN	TOD	TOD#00:00:00.000	16#0000_0000[1]
TOD#MAX TIME_OF_DAY#MAX	TOD	TOD#23:59:59.999	16#0526_5BFF[1]
[1] Internal display only			

1.4.1.3 普通的数据类型

普通的数据类型被用作系统功能和系统功能块的输入和输出参数。子程序被称做包含在普通数据类型中的每个数据类型的变量。

下表列出了可用的普通数据类型：

Table 3-9 General data types

General data type	Data types contained
ANY_BIT	BOOL, BYTE, WORD, DWORD
ANY_INT	SINT, INT, DINT, USINT, UINT, UDINT
ANY_REAL	REAL, LREAL
ANY_NUM	ANY_INT, ANY_REAL
ANY_DATE	DATE, TIME_OF_DAY (TOD), DATE_AND_TIME (DT)
ANY_ELEMENTARY	ANY_BIT, ANY_NUM, ANY_DATE, TIME, STRING
ANY	ANY_ELEMENTARY, user-defined data types (UDT), system data types, data types of the technology objects

1.4.1.4 基础系统数据类型

在 SIMOTION系统中， 表格中指出的数据类型使用与基础数据类型类似。 和许多系统功能一起使用。

标识符	位宽度	用途
StructAlarmId	32	alarmId 的数据类型用于识别唯一的信息。 alarmId 用于生成信息。见功能手册 SIMOTION基本功能。初始为 STRUCTALARMID#NIL
StructTaskId	32	taskId 的数据类型是在执行系统中识别唯一的任务。见功能手册 SIMOTION 基本功能。初始为 STRUCTTASKID#NIL

无效的基础数据类型数值的符号常量

符号常量	数据类型	意义
STRUCTALARMID#NIL	StructAlarmId	无效 AlarmId
STRUCTTASKID#NIL	StructTaskId	无效 TaskId

1.4.2 用户定义的数据类型

1.4.2.1 用户定义的数据类型

用户定义的数据类型（ UDT）通过在声明章节中随后的源文件部分和以下内容来创建 TYPE/END_TYPE

- interface 部分
- implementation 部分
- 程序组织单元（ POU）

可以使用在声明章节中创建的数据类型。源文件部分决定了类型声明的范围。

1.4.2.2 用户定义的数据类型的语法（类型声明）

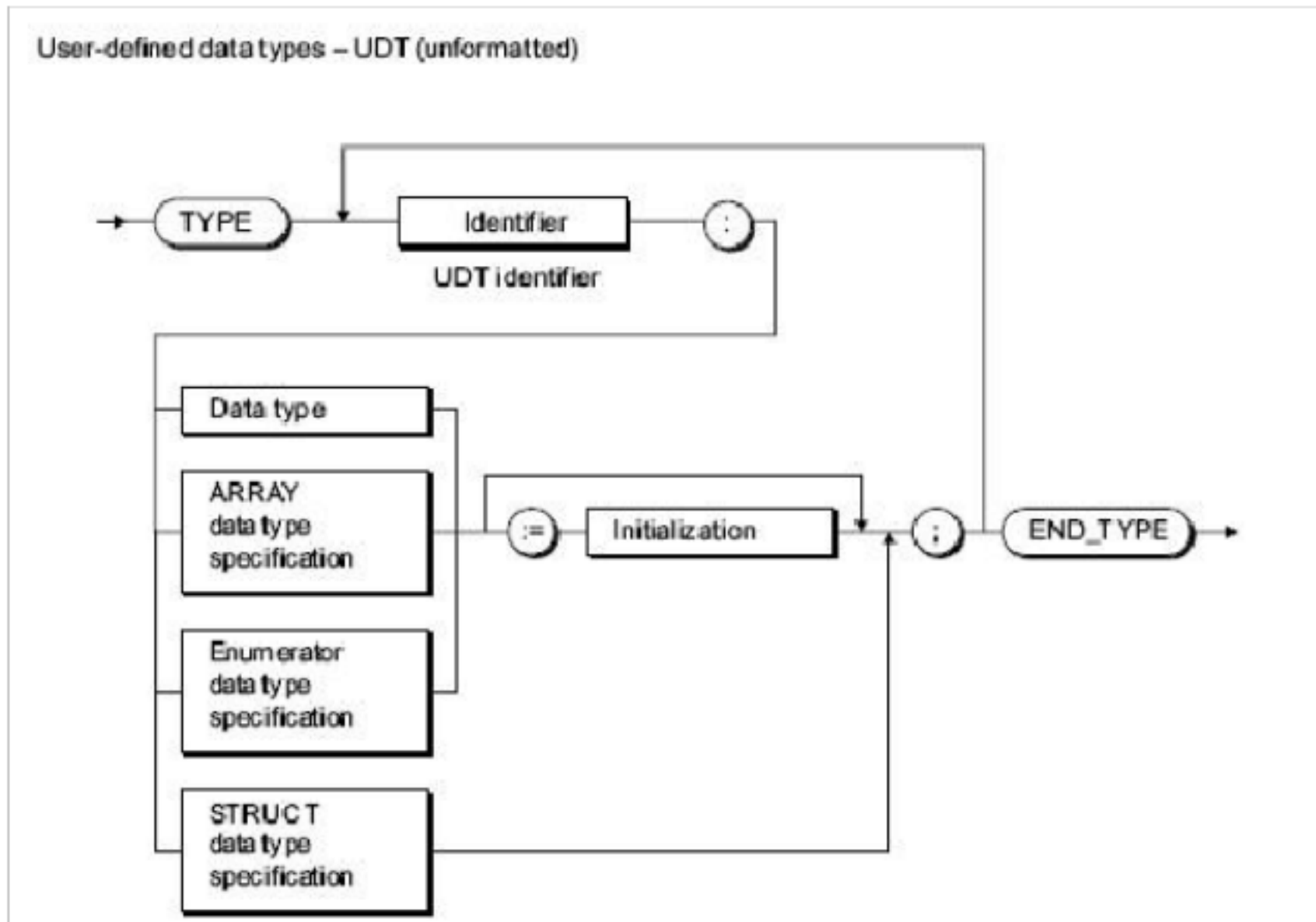


Figure 3-6 Syntax: User-defined data type

对于每个声明的数据类型，需要遵循以下：

1.名称：数据类型的名称必须遵循标识符的规则。

2.数据类型声明

数据类型包括（见第 96 页，基础派生或派生数据类型）

-基本数据类型

-之前声明的 UDT

-TO 数据类型

-系统数据类型

下列数据类型声明也是可能的：

-阵列数据类型声明（见派 97 页生数据类型 ARRAY）

-枚举数据类型声明（见派 99 页生数据类型 Enumerator）

-STRUCT 数据类型声明（见派 100 页生数据类型 STRUCT）

括号中的内容只可以参考的章节，相关的数据类型声明详细描述在这些章节中。

3.可选的初始值

你可以为每个数据类型指定初始值。如果你声明一个数据类型的变量，初始值被指定为变量。

例外：在 STRUCT 数据类型中，每个单独的组件的初始值范围规定在数据类型声明中

见变量或数据类型的初始化（第 107 页）

完整的 UDT 声明用 END_TYPE 关键词来结束。你可以使用 TYPE/END_TYPE 结构来创建任意数据类型的数字。你可以使用定义的数据类型来声明变量或参数。

只要语法在图标中可见， UDT 可以用任意方式嵌套。例如：你可以使用之前定义的 UDT 或嵌套结构作为一个数据类型声明。类型声明仅可以连续使用，并不是以嵌套结构的形式。

注意：你可以学习如何在所有变量声明的概述（见 106 页）中声明变量和参数，以及如何在语法中使用 UDT 来数值指定（见 113 页）。

1.4.2.3 基础派生或派生的数据类型

在数据类型的派生中， 一个基础或用户定义的数据类型 （UDT）使用 TYPE/END_TYPE 结构来定义。

TYPE 标识符：基本数据类型 { := initialization } ; END_TYPE
TYPE 标识符：用户定义数据类型 { := initialization } ; END_TYPE

一旦你已声明数据类型， 你可以定义派生的数据类型标识符的变量。 这等同于声明变量。

Table 3-12 Examples of derivation of elementary data types

```
TYPE
    I1: INT;      // Elementary data type
    R1: REAL;     // Elementary data type
    R2: R1;       // Derived data type (UDT)
END_TYPE
VAR
    // These variables can be used wherever
    // variables of type INT can be used.
    myI1 : I1;
    myI2 : INT;   // No derived data type!

    // These variables can be used wherever
    // variables of type REAL can be used.
    myR1 : R1;
    myR2 : R2;
END_VAR
myI1 := 1;
myI2 := 2;
myR1 := 2.22;
myR2 := 3.33;
```

1.4.2.4 派生数据类型 ARRAY

ARRAY派生数据类型包括用 TYPE/END_TYPE结构来定义的同种数据类型。下图的语法图展示了这种数据类型，在预留标识符 OF之后这种数据类型声明得更加详细。

TYPE标识符： ARRAY数据类型声明 { := initialization } ; END_TYPE

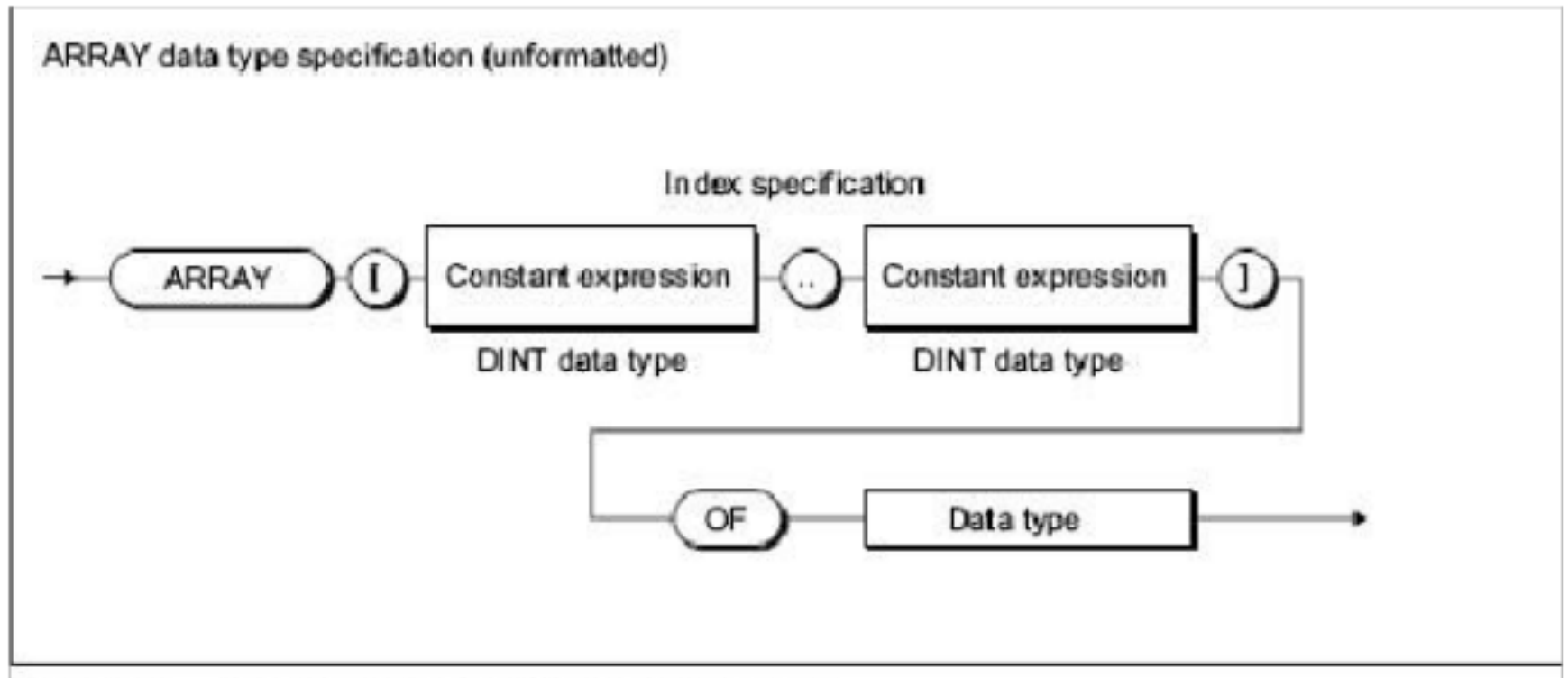


Figure 3-7 Syntax: ARRAY data type specification

索引声明书描述了阵列的限制：

- 阵列限制声明了索引值的最大和最小值。 可以使用常量或常量表达式。 数据类型为 DINT（或转化为 DINT-详见 141 页基础数据类型转化）
- 阵列限制必须用两个句号隔开
- 全部的索引声明须在方括号中
- 索引可以为一个数据类型 DINT（或转化为 DINT-详见 141 页基础数据类型转化）的整数值

注意：如果运行时阵列超限，程序会产生过程故障（见 SIMOTION基本功能功能手册）

用数据类型声明来声明阵列组件的数据类型。本章节中所描述的所有的选项都可以用作数据类型，甚至是用户定义数据（ UDT）。

有多种不同的阵列类型：

- 一维阵列类型为升序排列的一列数据元素
- 二维阵列是一个包括行和列的数据表。第一维是指行数，第二维是指列数。
- 更高维的阵列类型是二维阵列类型的扩展

Table 3-13 Examples of one-dimensional arrays

```
TYPE
    x : ARRAY[0..9] OF REAL;
    y : ARRAY[1..10] OF C1;
END_TYPE
```

二维阵列可以与带行和列的表格比较。可以通过多层次的类型声明来创建二维或多维阵列。

Table 3-14 Examples of multi-dimensional arrays

```
TYPE
    a : ARRAY[1..3] OF INT;     // one-dimensional array (3 columns):
    matrix1: ARRAY[1..4] OF a;      // two-dimensional Field
                                    // (4 lines with 3 columns)
    b: ARRAY[4..8] OF INT;     // one-dimensional array (5 columns):
    matrix2: ARRAY[10..16] OF b;    // two-dimensional Field
                                    // (7 lines with 5 columns)
END_TYPE

VAR
    m: matrix1;         // Variable m of data type two-dim. Field
    n: matrix2;         // Variable m of data type two-dim. Field
END_VAR

m[4][3] := 9;           // Write to Matrix1 at line 4, column 3
n[16][8] := 10;         // Write to Matrix2 at line 7, column 5
```

在例子中，你可以定义：

●表格列 a[1] 到 a[3]作为一维阵列，将包含整数

●表格行矩阵 1[1] 到矩阵 2[4]作为一个阵列， 但是作为使用表格中的列创建的阵列的数据类型声明。

当你在数据类型声明中声明阵列时，创建了一个第二维度。可以使用此方式创建更多的维度。

使用创建此表格的数据类型来声明一个变量。使用方括号为表格中的每个维度寻址。

1.4.2.5 派生的数据类型—枚举

在枚举的数据类型中，使用受限的标识符或名称来定义 TYPE/END_TYPE 结构。

TYPE 标识符：枚举数据类型声明 { := initialization } ; END_TYPE

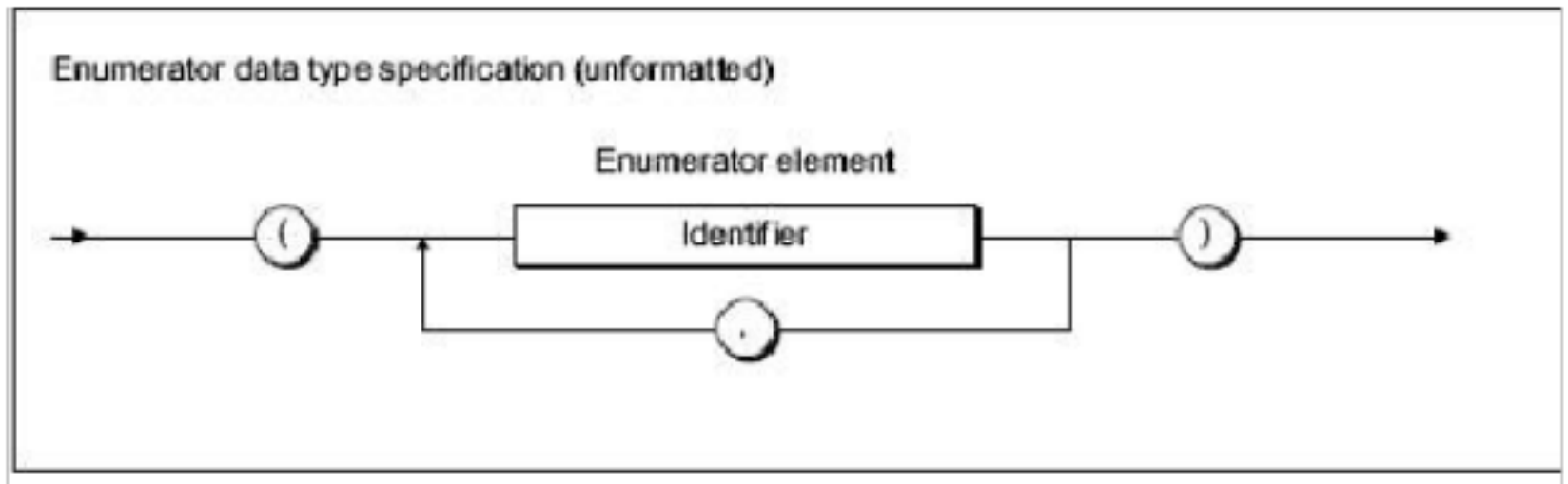


Figure 3-8 Syntax: Enumerator data type specification

一旦你已声明了标识符的数据类型，可以在枚举中定义变量。在语句部分，可以从这些变量的定义标识符（枚举元素）表单中数值指定。

可以直接声明数据类型：把枚举数据类型标识符和“ # ”放在枚举前面。

可以包含带 enum_type#MIN 和 enum_type#MAX 结构的枚举数据类型的第一和最后一个值， enum_type 为枚举数据类型标识符。

可以包含带 ENUM_TO_DINT转化功能的枚举元素的数值。

Table 3-15 Examples of enumerator data types

```
TYPE
    C1: (RED, GREEN, BLUE);
END_TYPE

VAR
    myC11, myC12, myC13 : C1;
END_VAR

myC11 := GREEN;
myC11 := C1#GREEN;
myC12 := C1#MIN;          // RED
myC13 := C1#MAX;          // BLUE
```

注意：你将会得知枚举数据类型为系统数据类型。

枚举数据类型可以为一个结构的部分，意味着在用户定义的数据结构中可以在任意的低级别中被找到。

1.4.2.6 派生的数据类型 STRUCT（结构）

派生的数据类型 STRUCT，或者结构包括 TYPE/END_TYPE 结构的固定数量的组件。这些组件的数据类型可以变化：

TYPE 标识符： STRUCT 数据类型声明； END_TYPE

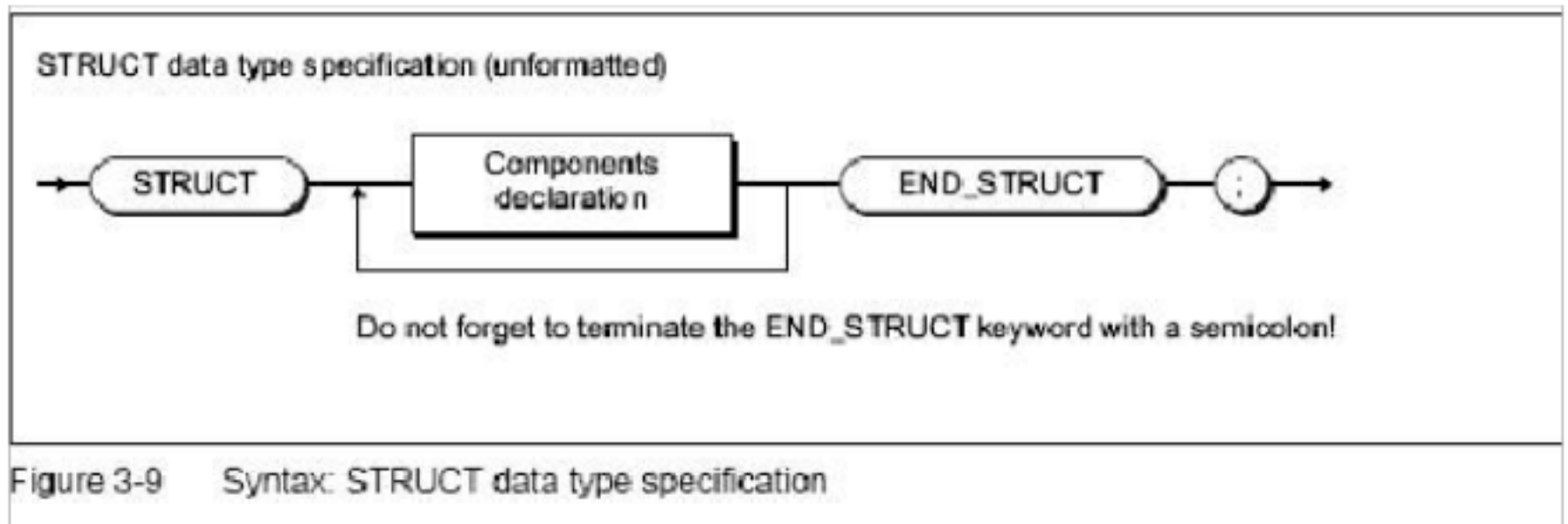


Figure 3-9 Syntax: STRUCT data type specification

The syntax of the component declaration is shown in the following figure.

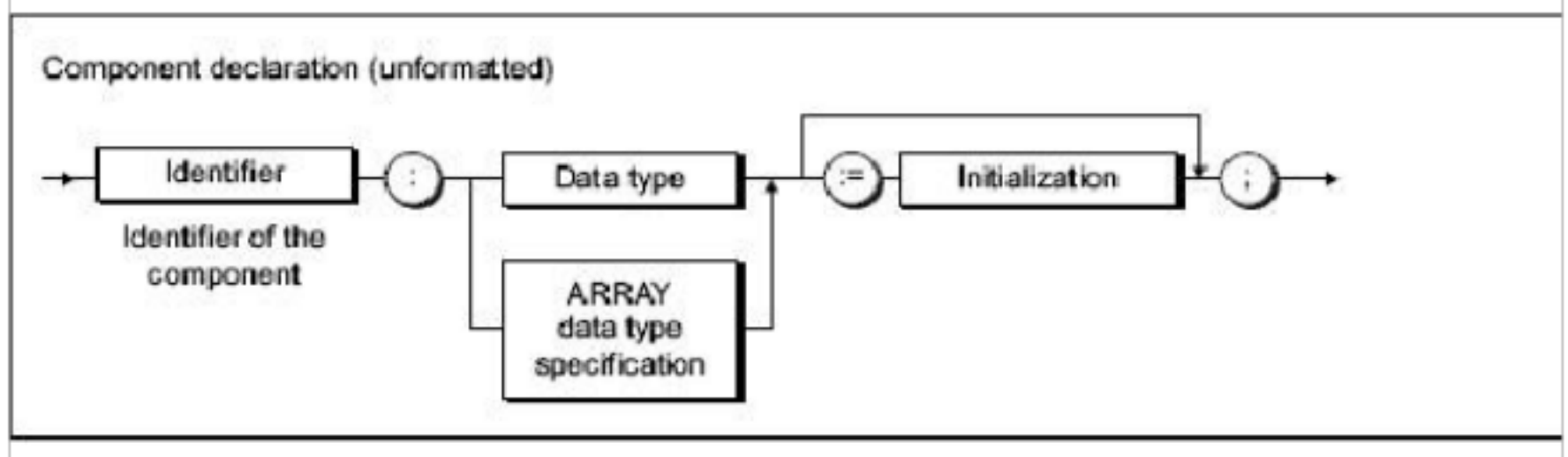


Figure 3-10 Syntax: Component declaration

下列为数据类型：

基础数据类型

●之前声明过的 UDT

●系统数据类型

●TO 数据类型

●ARRAY数据类型声明

你可以选择给组件赋初始值。继续关于变量初始值或数据类型（见 107 页变量或数据类型的初始值）

注意：

下列数据声明不可以在组件声明中直接使用

●STRUCT数据类型声明

●枚举数据类型声明

解决方案：用之前提到的声明在组件声明中预先声明 UDT（用户定义数据类型）

这允许嵌套 STRUCT数据类型

同样可能会发现 STRUCT数据类型为系统数据类型

这个例子说了了一个 UDT 是如何定义的，在变量声明中又是如何使用的这种数据类型

Table 3-16 Examples of derived data type STRUCT

```
TYPE      // UDT definition
    S1: STRUCT
        var1 : INT;
        var2 : WORD := 16#AFA1;
        var3 : BYTE := 16#FF;
        var4 : TIME := T#1d_1h_10m_22s_2ms;
    END_STRUCT;
END_TYPE

VAR
    myS1 : S1;
END_VAR

myS1.var1 := -4;
myS1.var4 := T#2d_2h_20m_33s_2ms;
```

1.4.3 技术目标数据类型

1.4.3.1 技术目标数据类型的描述

可以用技术目标来声明变量。下表给出了可用的技术目标的数据类型。

例如，可以用数据类型 posaxis 来声明一个变量，然后指配一个是适合的位置轴。

Table 3-17 Data types of technology objects (TO data type)

Technology object	Data type	Contained in the technology package
Drive axis	driveAxis	CAM[1 2], PATH, CAM_EXT
External encoder	externalEncoderType	CAM[1 2], PATH, CAM_EXT
Measuring input	measuringInputType	CAM[1 2], PATH, CAM_EXT
Output cam	outputCamType	CAM[1 2], PATH, CAM_EXT
Cam track (as of V3.2)	_camTrackType	CAM, PATH, CAM_EXT
Position axis	posAxis	CAM[1 3], PATH, CAM_EXT
Following axis	followingAxis	CAM[1 4], PATH, CAM_EXT
Following object	followingObjectType	CAM[1 4], PATH, CAM_EXT
Cam	camType	CAM, PATH, CAM_EXT
Path axis (as of V4.1)	_pathAxis	PATH, CAM_EXT
Path object (as of V4.1)	_pathObjectType	PATH, CAM_EXT
Fixed gear (as of V3.2)	_fixedGearType	CAM_EXT
Addition object (as of V3.2)	_additionObjectType	CAM_EXT
Formula object (as of V3.2)	_formulaObjectType	CAM_EXT
Sensor (as of V3.2)	_sensorType	CAM_EXT
Controller object (as of V3.2)	_controllerObjectType	CAM_EXT
Temperature channel	temperatureControllerType	TControl
General data type, to which every TO can be assigned	ANYOBJECT	

1) As of Version V3.1, the BasicMC, Position and Gear technology packages are no longer contained.

2) For Version V3.0, also contained in the BasicMC, Position and Gear technology packages.

3) For Version V3.0, also contained in the Position and Gear technology packages.

4) For Version V3.0, also contained in the Gear technology package.

可以通过结构访问技术目标的元素。

Table 3-18 Symbolic constants for invalid values of technology object data types

Symbolic constant	Data type	Meaning
TO#NIL	ANYOBJECT	Invalid technology object

1.4.3.2 轴属性的继承

轴的继承声明所有的数据类型，系统变量和 TO driveAxis 的功能都全部包含在 TO positionAxis 中。同样，位置轴页包含在 TO followingAxis 中，下列轴是在 TO pathAxis 中，有以下影响：

●如果一个功能或功能块，除了 driveAxis 数据类型的输入参数之外，调用时你可以使用 position axis 或者 following axis 或 path axis。

●如果有一个功能或功能块，除了 posAxis 数据类型的输入参数之外，调用时你可以使用 following axis 或 path axis。

1.4.3.3 技术目标数据类型的例子

以下，你将看见可选的技术目标数据类型 （你将在 SIMOTION基本功能功能手册中找到强制使用 TO 数据类型的变量）的变量使用。第二个例子介绍了无需使用 TO 数据类型的变量的替代方法。

TO 功能将被用为在程序的主要部分启用一个轴，因此轴能定位。在定位操作结束后，现行的轴的位置将使用结构访问来进行记录。

第一个例子使用了 TO 数据类型的变量来演示用途。

Table 3-19 Example of the use of a data type for technology objects

```
VAR
    myAxis : posAxis;    // Declaration variable for axis
    myPos : LREAL;       // Variable for position of axis
    retVal: DINT;       // Variable for return value of the
                // TO function
END_VAR
myAxis := Axis1;         // The name Axis1 was defined when the axis
                         // was configured in the project navigator.

// Call of function with variables of TO data type:
retVal := _enableAxis(axis := myAxis, commandId := _getCommandId());

// Axis is positioned.
retVal := _pos(axis := myAxis,
               position := 100,
               commandId:= _getCommandId() );

// Scan the position using structure access
myPos := myAxis.positioningState.actualPosition;
```

第二个例子未使用 TO 数据类型的变量。

Table 3-20 Example of using a technology object

```
VAR
    myPos : LREAL;       // Variable for position of axis
    retVal: DINT;  // Variable for return value of TO function
END_VAR

// Call of function without variable of TO data type
// The name Axis1 was defined when the axis
// was configured in the project navigator.
retVal := _enableAxis(axis := Axis1,
                      commandId:= _getCommandId() );

// Axis is positioned.
retVal := _pos(axis := Axis1
               position := 100,
               commandId:= _getCommandId() );

// Scan the position using structure access
myPos := Axis1.positioningState.actualPosition;
```

你将在 SIMOTION运动控制功能描述中招待技术目标的组态和配置细节。

1.4.4 系统数据类型

有很多可用的喜悦数据类型，无需预先声明就可以使用。每个导入的技术包提供了一库的系统数据类型。

额外的系统数据类型可以找到

- 在普通标准功能中的参数（见 SIMOTION基本功能功能手册）
- 在普通标准功能模块中的参数（见 SIMOTION基本功能功能手册）
- 在 SIMOTION设备的系统变量中（见相关的参数手册）
- 在 SIMOTION设备的系统功能参数中（见相关的参数手册）
- 在技术目标的系统变量和组态（见相关的参数手册）
- 在技术目标的系统功能参数中（见相关的参数手册）

1.5 变量声明

一个变量定义了可在 ST源文件中使用的变量内容的数据项。一个变量包括一个可以自由选择的标识符（如 myVar1)）和一个数据类型（如 BOOL）。预留的标识符（见 75 页预留标识符）不能作为标识符使用。

1.5.1 变量声明的语法

变量通常是根据源文件声明部分的相同模式来创建的。

- 通过合适的关键词来开始一个声明块 (如 VAR, VAR_GLOBAL- 见 106 页所有变量声明的概述
- 遵循实际的变量声明（见图表），也可以如你所愿尽可能的创建，顺序是任意的。
- 以 END_VAR来结束一个声明块
- 你可以创建更多的声明块（要有同样的关键词）

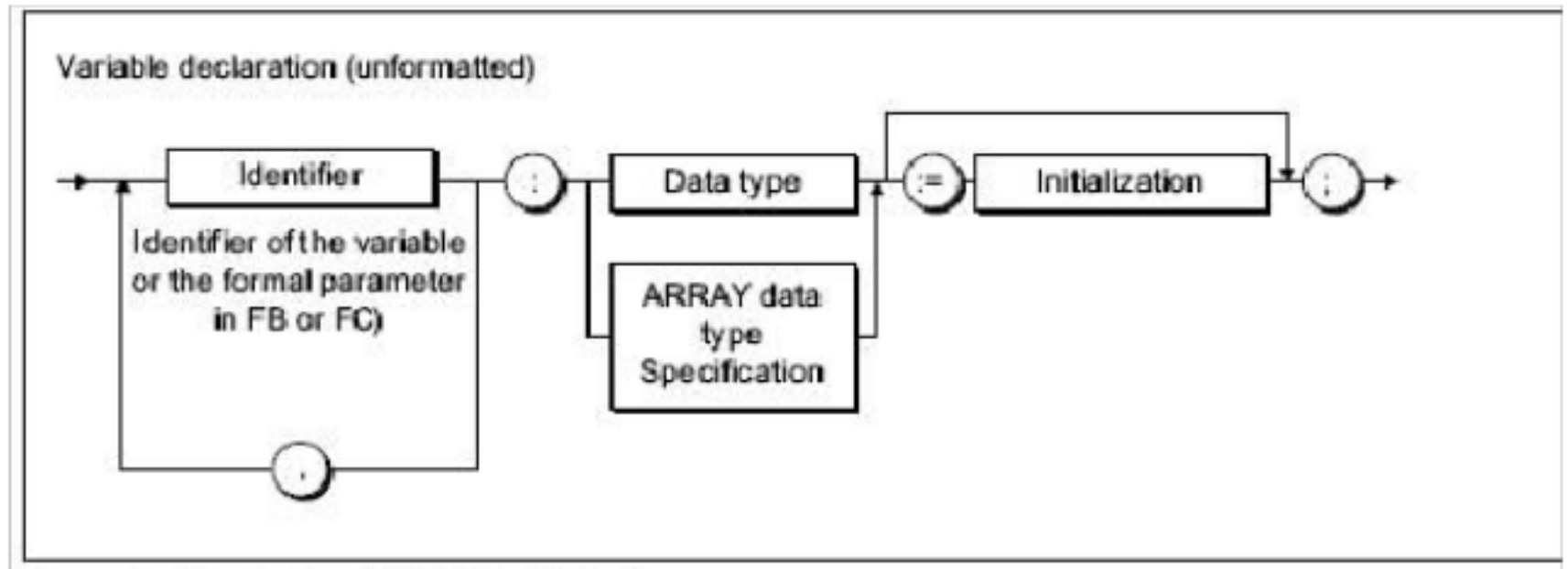


Figure 3-11 Syntax: Variable declaration

注意

●变量的名称必须是一个标识符，如，只能包含字母，数字或下划线，但是不能包含特殊字符。

●允许作为数据类型的下列项：

基础数据类型

—UDT（用户定义数据类型）

—系统数据类型

—TO 数据类型

—ARRAY数据类型声明

—功能块的设计

● 可以在声明语句中直接给变量赋初始值，这叫做初始化

从格式来的派生如下：

●常量声明（常量必须有初始值，见 111 页常量）

●过程图像访问（见 106 页所有变量声明的概述）

—不需要一个变量声明作为绝对过程图像访问

—不允许初始化作为符号过程访问控制

Table 3-21 Examples of variable declarations

```
VAR CONSTANT
    PI : REAL := 3.1415;
END_VAR

VAR
    // Declaration of a variable ...
    var1 : REAL;
    // ... or if there are several variables of the same type:
    var2, var3, var4 : INT;
    // Declaration of a one-dimensional array:
    a1 : ARRAY[1..100] OF REAL;
    // Declaration of a character string (string):
    str1 : STRING[40];
END_VAR
```

1.5.2 所有变量声明的概述

在变量和参数声明中声明名称、数据类型和变量的初始值。在下列源文件的声明部分执行这些声明：

- interface 部分
- 执行部分
- POU(程序、功能、功能块、表达式)

源文件部分也决定了你可以声明的变量和范围

欲知源文件部分的更多信息，参见 86 页的 ST源文件结构个 169 页的源文件部分。

关键词	意义	声明
VAR	暂时或静态变量的声明，见 184 页变量模型	任意 POU
VAR_GLOBAL	单元变量声明，见 184 页变量模型	interface 部分 implementation 部分
VAR_IN_OUT	输入 / 出参数的变量声明； POU直接访问变量（使用参考），也可直接修改。 见 148 页定义功能， 149 页定义功能块	功能 功能块 表达式
VAR_INPUT	输入参数的变量声明；数值是外部赋予， 不能在 POU里面改变这个数值。 见 148 页定义功能， 149 页定义功能块	功能 功能块 表达式
VAR-OUTPUT	输出参数的变量声明；数值是从功能块传递的。 见 148 页定义功能， 149 页定义功能块	功能块
VAR_TEMP	临时变量的声明 见 184 页变量模型	程序 功能块
RETAIN	保留变量的声明 见 184 页的变量模型	仅作为补充 interface 和 implementation 部分的 VAR_GLOBAL
CONSTANT	常量的声明，见 111 页常量	仅作为补充 在 FB,FC或程序中的 VAR 在 interface 或 implementation 部分的 VAR_GLOBAL

1.5.3　变量或数据类型的初始值

在声明中指定初始值给变量或数据类型是可选的（见图表语法：变量声明或语法：用户定义数据类型）

●如果在变量声明中没有特别指定初始值，　编译程序自动给变量指定在数据类型声明中声明的初始值

●如果在数据类型声明中也没有特别指定初始值，　编译程序自动给变量或数据类型指定值为 0。例外：

—时间数据类型：初始值

—枚举数据类型：　1.枚举值

预先给变量或用户定义的数据类型指定初始值在数据类型声明之后（见语法图表：变量初始化）

●根据语法图表：常量表达式来给基础数据类型（或从基础数据类型派生的数据类型）赋常量表达式

●根据语法图：域初始化给域（阵列）指定域初始化列表

●根据语法图：结构初始化列表来给单独的结构成分来指定结构初始化列表

●给枚举数据类型指定枚举元素

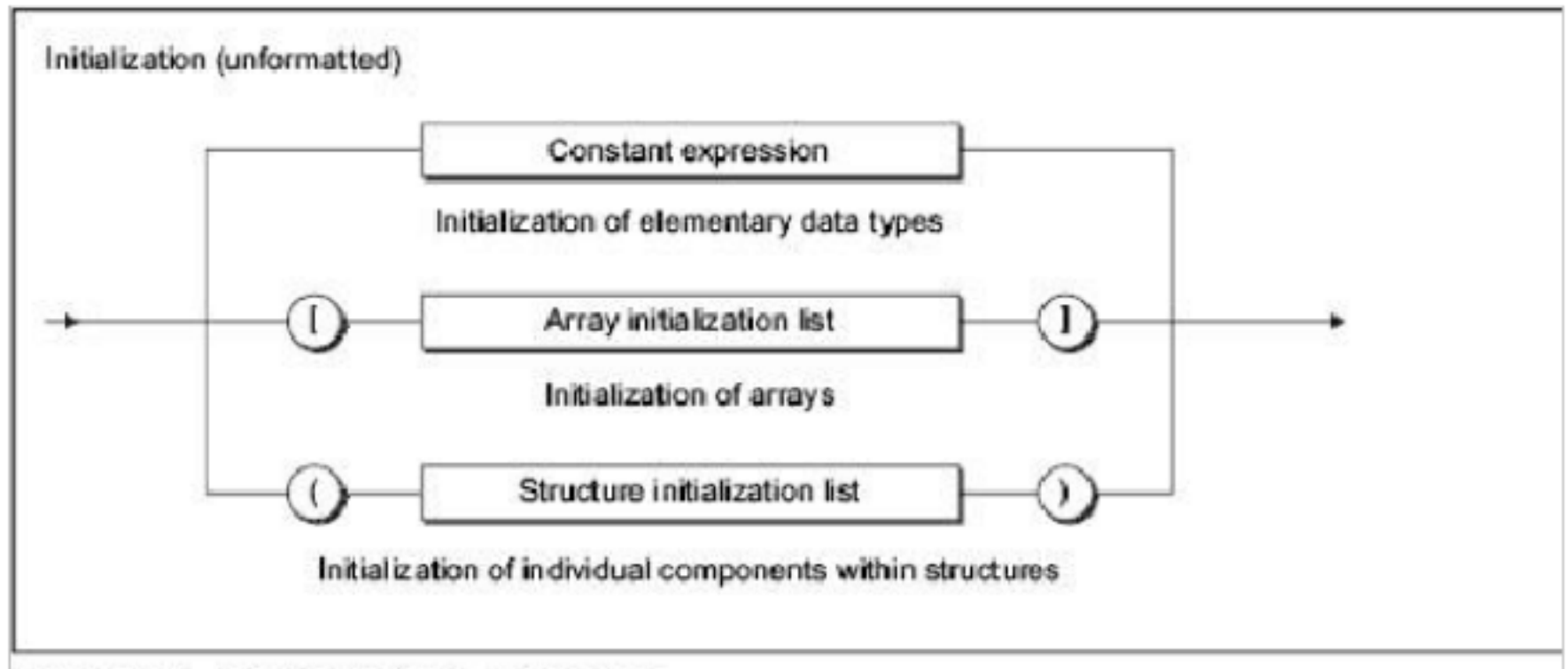


Figure 3-12　Syntax: Variable initialization

在编译程序时给变量赋的初始值是从常量表达式计算得知的。见语法图。欲知更多常量表达式的信息，见常量表达式的语法图。

注意一个变量列表　(a1, a2, a3, .. : INT := ..)可以是由普通数值初始化的。在这种情况下，不需要单独初始化变量　(a1 : INT := .. ; a2 : INT := .. ; etc.)

Constant expression (unformatted)

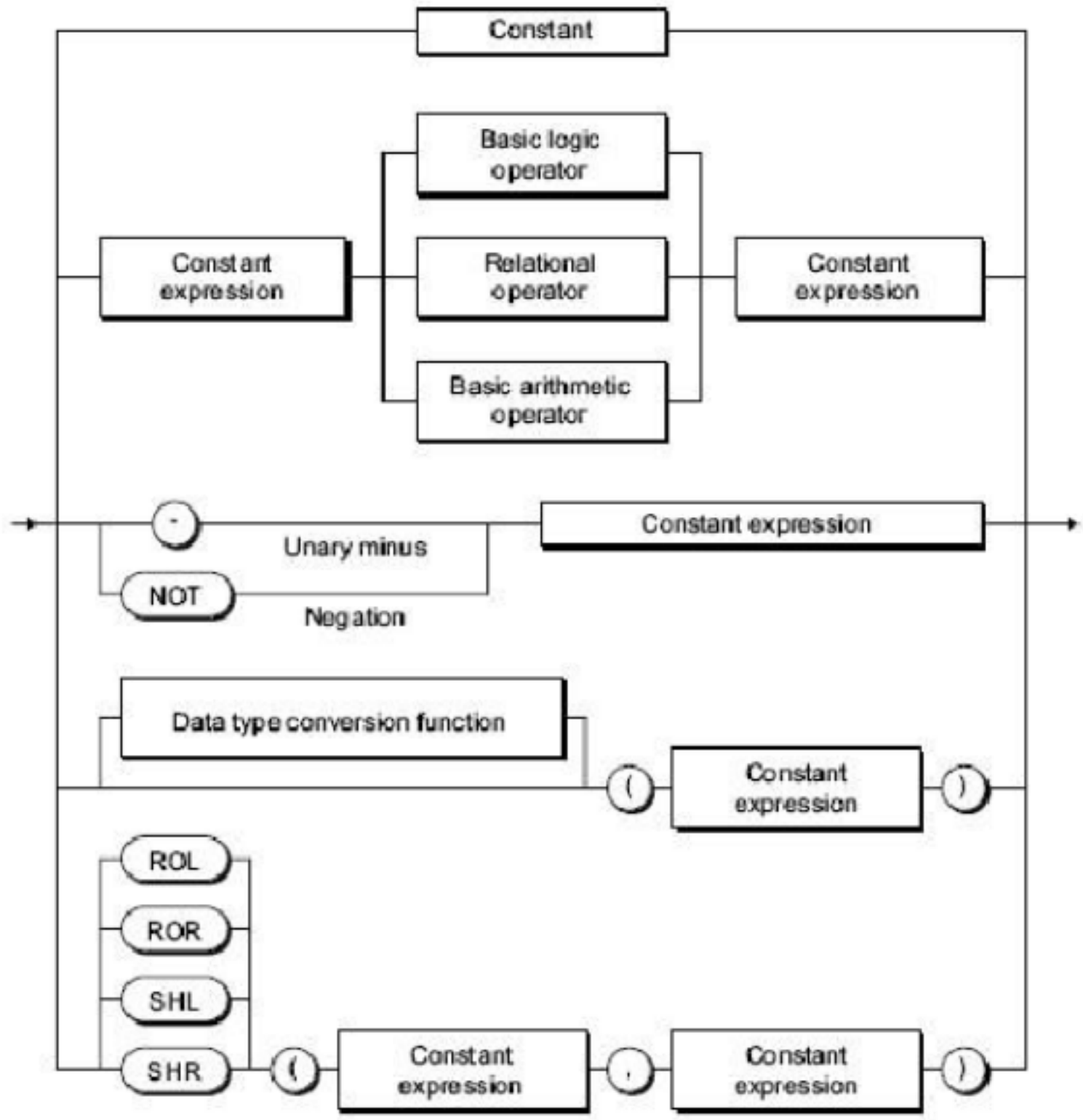


Figure 3-13 Syntax: Constant expression

Array initialization list (unformatted)

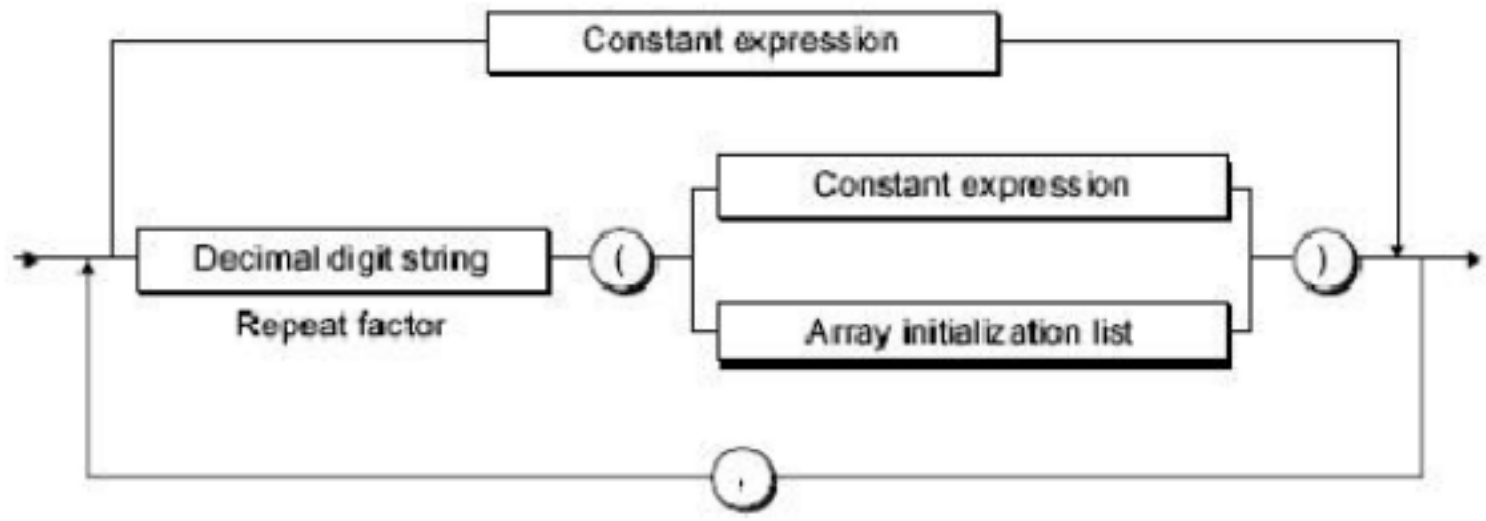


Figure 3-14 Syntax: Array initialization list

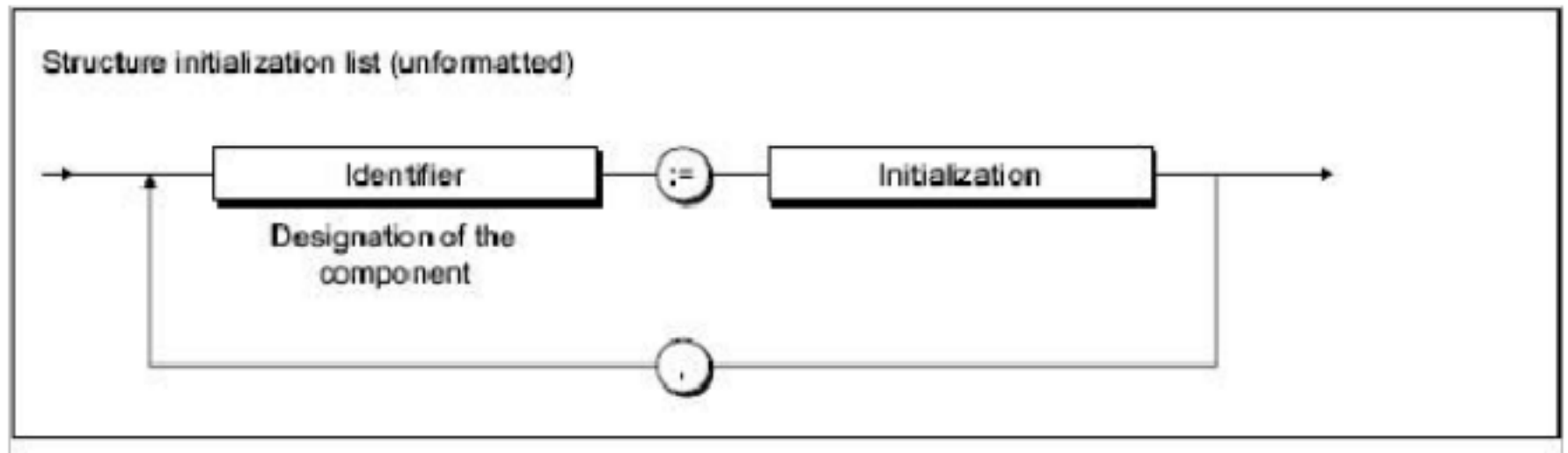


Figure 3-15 Syntax: Structure initialization list

Table 3-23 Examples of variable initialization

```
VAR
    // Declaration of a variable ...
    var1 : REAL := 100.0;
    // ... or if there are several variables of the same type:
    var2, var3, var4 : INT := 1;
    var5 : REAL := 3 / 2;
    var6 : INT  := 5 * SHL(1, 4)
    myC1 : C1 := GREEN;
    array1 : ARRAY [0..4] OF INT  := [1, 3, 8, 4, 0];
    array2 : ARRAY [0..5] OF DINT := [6 (7)];
    array3 : ARRAY [0..10] OF INT := [2 (2(3),3(1)),0];
                // is equivalent to [2(3),3(1),2(3),3(1)),0]
                // Initialization as follows:
                // Array elements 0, 1        with 3;
                // Array elements 2, 3, 4        with 1;
                // Array elements 5, 6        with 3;
                // Array elements 7, 8, 9        with 1;
                // Array element 10             with 0
    myAxis : PosAxis := TO#NIL;
END_VAR
```

Table 3-24 Examples of data type initialization

```
TYPE
    // Initialization of a derived data type
    type1 : REAL := 10.0;
    // Initialization of an enumeration data type
    cmyk_colour : (cyan, magenta, yellow, black) := yellow;
    // Initialization of structures
    var_rgb_colour : STRUCT
            red, green, blue : USINT := 255;// white
        END_STRUCT;
    new_colour : var_rgb_colour := (red := 0, blue := 0);//green
END_TYPE
```

技术目标（ TO）数据类型的变量由编辑器用 TO#NIL初始化。任务对变量初始化的影响在SIMOTION基本功能功能手册中描述。

3.5.4 常量

常量是带固定值的数据， 在程序运行时不能更改固定数值。 常量的声明方式与变量一样。

- 本地常量的 POU声明部分（见语法图： POU 中的常量块和语法图：常量声明）
- 在 ST 源文件单元常量是 interface 或 implementation 部分（见语法图：在 interface 或 implementation 部分的单元常量和语法图： 常量声明） 。可以在 interface 部分导入单元常量到其他 ST 源文件（见 184 页常量模型）

源文件部分也决定常量声明的范围。

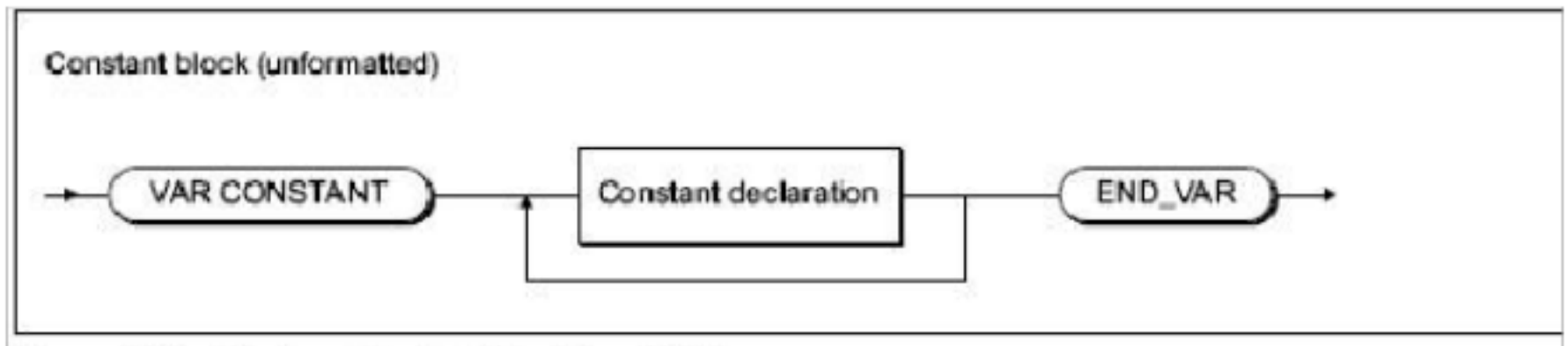


Figure 3-16 Syntax: Constant block in a POU

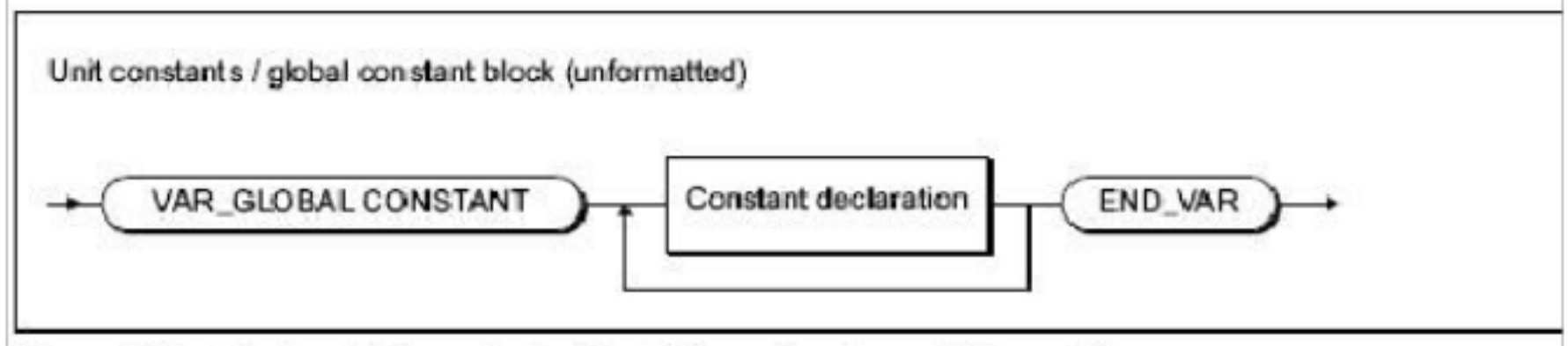


Figure 3-17 Syntax: Unit constants in interface or implementation section

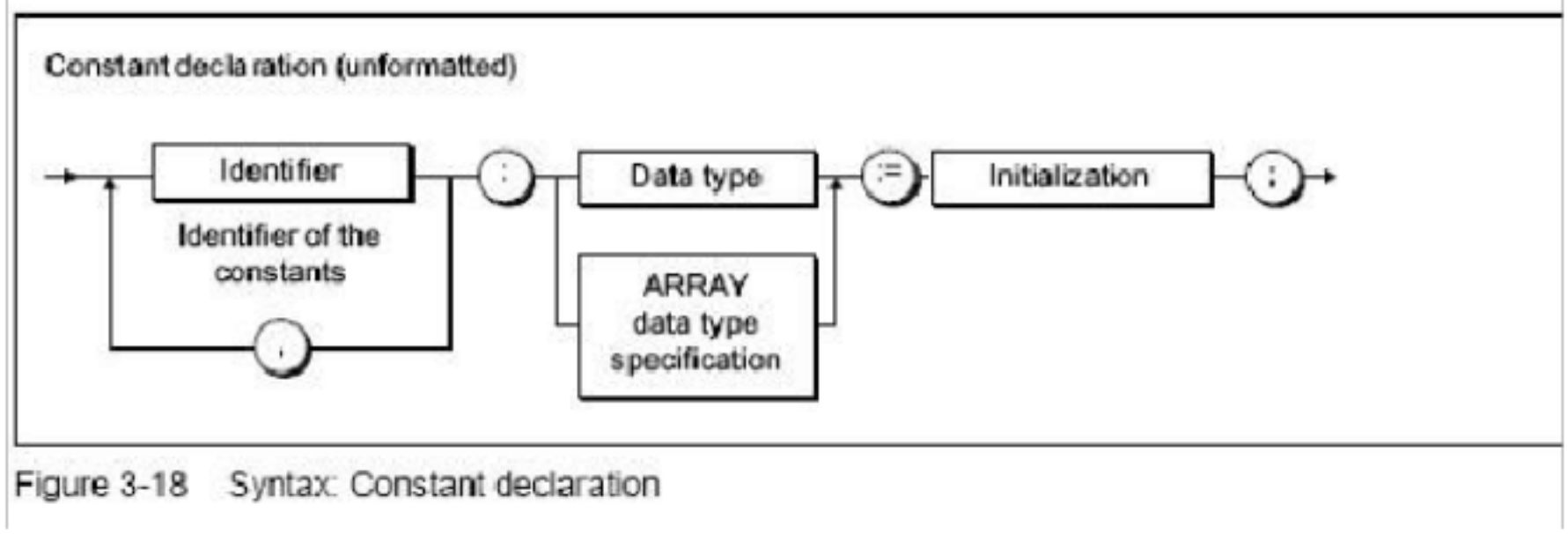


Figure 3-18 Syntax: Constant declaration

赋给常量的值是通过在编译时计算常量表达式得出的。欲知更多常量表达式的语法图和信息，见常量表达式的语法图。

Table 3-25 Examples of constants

```
VAR CONSTANT
    PI : REAL := 3.1415;
    intConst      : INT := 10;
    sintConst     : SINT := 0;
    dintConst     : DINT := 10_000;
    timeConst     : TIME := TIME#1h;
    strConst      : STRING[40] := 'Example of a string';
    Two_PI        : REAL := 2 * PI;
END_VAR
```

1.6 赋值和表达式

你已经用字符串来创建数值指定， 也许是一个作为部分例子的语句 （见 87 页语句示范表），或者是在源文件声明部分初始化变量的时候。

然而这只是对可用的公式化指定数值的小范围的选择。手册的本章节通过使用大量的例子详细描述了这个重要的话题。

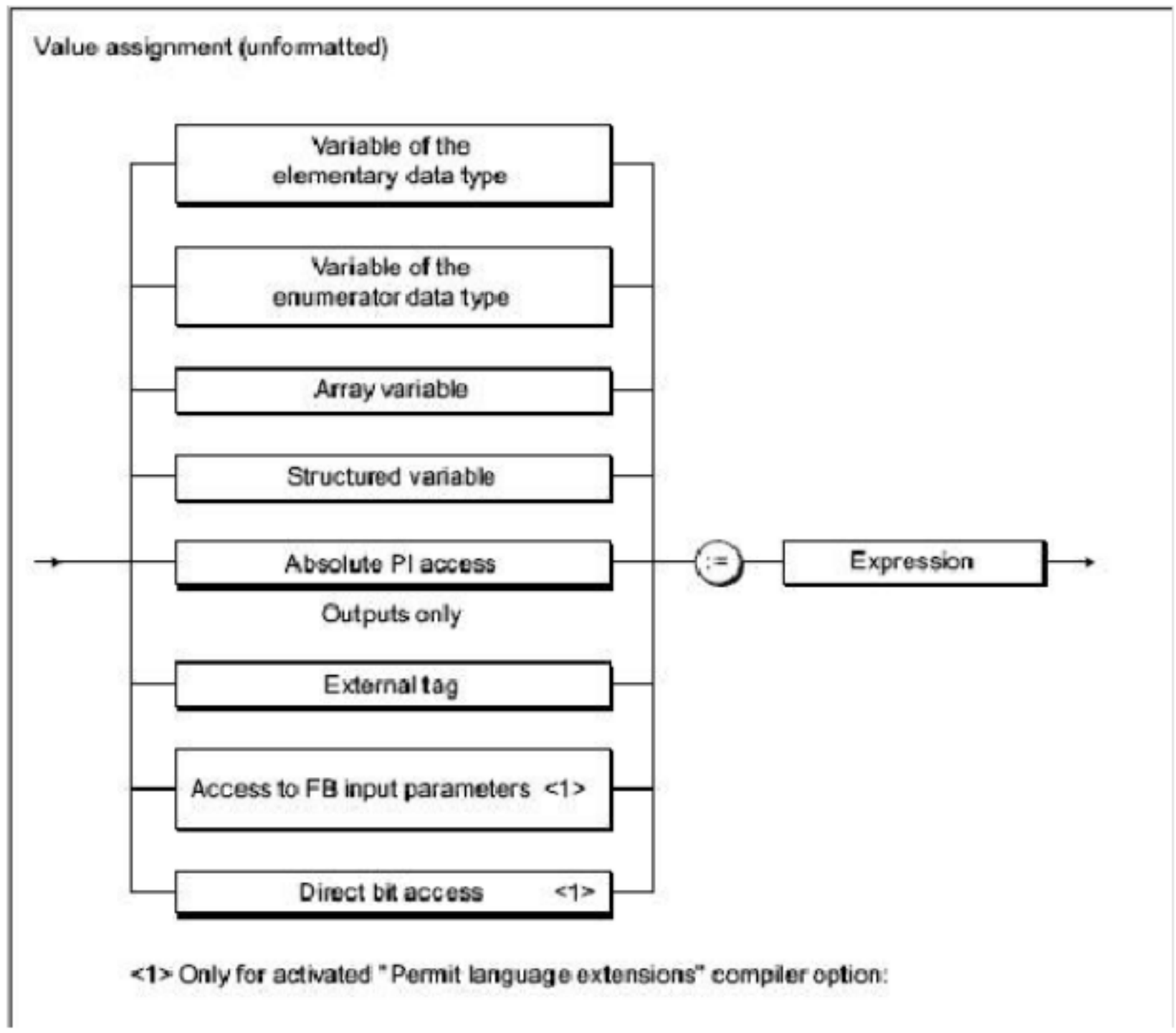
注意：在算法和逻辑表达式中，结果通常是通过最大数值格式化表达式和转化数据类型所得出的。隐式转化在数值指定时不总是可行的。欲知更多错误源文件和解决方法的信息，见 SIMOTION基本功能功能手册。

1.6.1 赋值

1.6.1.1 赋值的语法图

使用指定的数值来给变量数值指定。重写之前的数值。在一个数字可以正确指定前，在声明部分必须声明一个变量（见 105 页变量声明语法图）

如下面的语法图所示，表达式位于右侧。 结果保存在变量中，变量的名称列于左侧。所有的目标变量在图表中显示



下列包含了左侧的数值指定的解释和例子：

●基础数据类型的变量数值指定（ 114 页）

●派生的枚举数据类型的变量数值指定（ 117 页）
●派生的阵列数据类型的变量数值指定（ 118 页）
●派生的结构数据类型的变量数值指定（ 118 页）
●绝对的 PI 访问的数值指定（给过程图像寻址），见 221 页绝对 PI 访问

1.6.1.2 基础数据类型的变量的数值指定

当下列情况之一满足时，基础数据类型（ 90 页）的表达式可以指定给一个变量：

●表达式和目标变量是一样的数据类型

注意关于字符串数据类型 (114 页)的下列信息

●表达式的数据类型可以隐式转化为目标变量的数据类型

Examples

```
elemVar         := 3*3;
elemVar         := elemVar1;
```

1.6.1.3 串基础数据类型的变量数值指定

串数据类型的变量之间的数值指定

不同长度的串基础数据类型的变量之间的数值指定没有限制，如果声明的目标变量的长度要短于现在指定的字符串的长度，字符串被截取成目标变量的长度。

例外：

in/out 数值指定的应用 （参数转化为 in/out 参数）：指定变量的声明长度必须长于或者等同于目标变量（正式 in/out 参数）的声明长度。见 154 页参数转化为 in/out 参数。

例子：

```
string20        := 'ABCDEFG';
string20        := string30;
```

一个字符串元素的访问

一个字符串的单独元素可以用阵列 [1..n]的元素的相同方式寻址。这些元素隐式转化成基础数据元素 BYTE。通过这种方式， 字符串元素和 BYTE数据类型的变量之间的指定是可行的。

例子：

```
byteVar       := string20[5];
string20[10]          := byteVar;
```

需要考虑到下面的特殊情况：

1.当把 BYTE数据类型的变量指定给一个字符串元素时： (e.g. stringVar[n:] := byteVar):
—给字符串元素的数值超出了声明的字符串的长度
字符串保持不变， TSI#ERRNO被设为 1

—给字符串元素的数值超出了指定的字符串的长度 (n > LEN(stringVar))但是在声明的长度之内）
字符串的长度被更改，在 LEN(stringvar)和 N 之间的字符串元素被设为 $00

2.当给 BYTE数据类型的一个变量指定一个字符串元素时 (by teVar := stringVar[n:])：

—给字符串元素的数值超出了指定的字符串的长度 (n > LEN(stringVar))
变量设为 16#00, TSI#ERRNO设为 2

编辑字符串
不同的系统功能对于编辑字符串都是可用的，比如插入字符串，替换和字符等。见 SIMOTION基本功能功能手册

数字和字符串之间的转换
不同的系统功能对于数字数据类型的变量和字符串之间的转换是可用的，见 141 页基础数据类型转化和 SIMOTION基本功能功能手册

1.6.1.4 位数据类型的变量的数值指定

访问单独的位数据类型的变量的二进制数字
访问单独的 BYTE,WORD或 DWORD数据类型的变量的二进制数字：

- 通过标准功能
  - 可以通过使用 _getBit, _setBit 和 _toggleBit 功能来读，写或转化任意位字符串
  - 可以通过变量来指定位数
- 直接的位访问
  - 可以定义你需要访问的变量的位，通过变量后面的一个隔开的点
  - 可以通过一个常量来指定位数

必须先开启编辑器功能选项“允许语言扩展”才能使用（见 45 全局编辑器设置和 46 页本地编辑器设置）

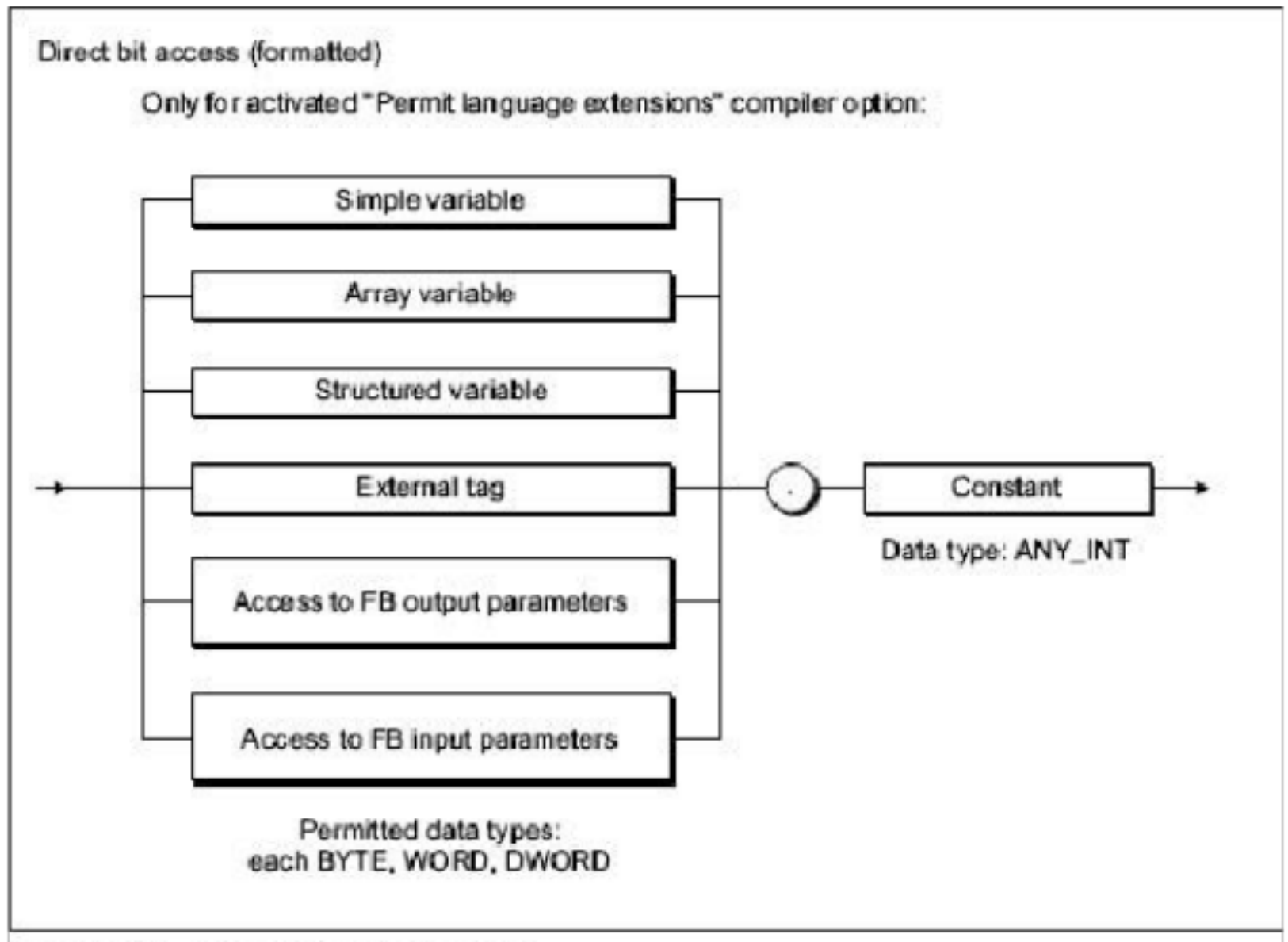


Figure 3-20 Syntax: Direct bit access

Table 3-26 Example of direct bit access

```
// Only with compiler option "Permit language extensions"
FUNCTION f : VOID
    VAR CONSTANT
        BIT_7 : INT := 7;
    END_VAR
    VAR
        dw : DWORD;
        b: BOOL;
    END_VAR
    b := dw.BIT_7;   // Access to bit 7
    b := dw.3;       // Access to bit 3
//  b := dw.33;      // Compilation error;
                     // Bit 33 not permitted.
END_FUNCTION
```

注意：访问 I/O 变量或者系统变量的二进制数字可以被其他任务干扰。 所以不能保证一致性。

编辑位数据类型的变量

你能够：

1.可以把同数据类型的不同变量合为一种高级别的数据类型的变量（例如： BYTE 数据类型的两个变量变为一种 WORD 数据类型）。不同的系统功能都可以使用此功能，如 WORD_FROM_2BYTE

2.把一个变量拆分成低级别的几种数据类型的变量（如： SWORD 数据类型的一个变量分为 4 个 BYTE数据类型的变量）。不同的系统功能都可以使用此功能， 如 DWORD_TO_4BYTE

3.在一个变量中旋转或移位字节。位字符串标准功能　　ROL, ROR, SHL和 SHR都可使用此功能

这些系统功能和系统功能块在　　SIMOTION基本功能功能手册中有详细描述。

逻辑运算符
位数据类型的变量可以和逻辑运算符一起结合，见　　127 页逻辑表达式和位串行表达式。

1.6.1.5 派生的枚举数据类型的变量的数值指定

每个表达式和派生的枚举数据类型的每个变量（见　　99 页派生数据类型—枚举可以指定同数据类型的其他变量）

1.6.1.6 派生的阵列数据类型的变量的数值指定

一个阵列包含不同的维度和阵列元素，所有相同类型（见　　97 页派生数据类型—阵列）

有许多不同的方法来给变量指定阵列。你可以指定完整阵列，单独元素或部分阵列：
如果部件和阵列限制（最小和最大可能的数组下标）的数据类型相同，一个完整的阵列可以指定给其他的阵列。有效的指定为：

```
array_1　　:= array_2;
```

●一个单独的阵列元素通过阵列名称（方括号中的索引值）来寻址。索引必须是数据类型 SINT,USINT,INT,UINT,DINT的算术表达式

```
elem1　　:= array　　[i];
array_1　　[2]　　:= array_2　　[5];
array　　[j]　　:=　　14;
```

●可以通过省略右边开始的方括号中阵列的每个维度来获得有效子阵列的赋值。　　这样解决了与剩余目录数目相同的维度数的阵列的部分区域（见下例）

所以你可以引用矩阵中的行和独立部分，但是不能引用关闭的列（关闭了意义上的从　　...到...）。有效的指定为：

```
matrix1[i]　　:= matrix2[k];
array1　　:=　　matrix2　　[k];
```

1.6.1.7 派生的 STRUCT 数据类型的变量数值指定

包括 STRUCT数据类型声明的用户指定数据类型的变量称为结构变量。　　（见 100 页派生数据类型）。即可代表一个完整的结构也可代表结构的一个部件。

有效的结构变量参数如下：

struct1 //Identifier for a structure

struct1.elem1 //Identifier for a structure component

struct1.array1 //Identifier of a simple array
//within a structure

struct1.array1[5] //Identifier of an array component
//within a structure

有两种方法来给变量指定结构。可以参考完整的结构或结构成分。

●如果数据类型和结构成分名称相符，一个完整的结构仅可以被指定为其他结构。
有效的指定如下： struct1 := struct2;

●可以指定一个类型兼容的变量， 一个类型兼容的表达式或其他结构成分件给每个结构成分。
有效指定如下：

struct1.elem1 := Var1;

struct1.elem1 := 20;

struct1.elem1 := struct2.elem1;

struct1.array1 := struct2.array1;

struct1.array1[10] := 100;

注意：可以使用 FBInstanceName 和输出参数格式中的结构变量，如： myCircle.访问功能块的输出变量的环境。欲知功能块的更多信息，见 148 页定义功能描述和 149 页 定义功能块描述。

更多的结构变量的应用是为访问 TO变量和基础系统的变量。

1.6.2 表达式

一个表达式代表了当程序编译或执行时计算出的值。它包含运算对象（如：常量，变量或功能数值）和运算符 (如 . *, /, +, -)

运算对象的数据类型和相关的运算符决定了表达式的类型。 ST使用下列类型的表达式

●算术表达式
●关系表达式
●逻辑表达式
●位串行表达式

1.6.2.1 表达式结果

表达式的结果为：

- 指定给一个变量
- 用作一个控制语句的条件
- 用作一个功能或功能块调用的参数

注意：仅包含下列元素的表达式可用作变量初始化和阵列声明中的索引声明（见 107 页变量或数据类型中初始化的常量表达式）

常量

- 基本算术运算
- 逻辑和关系运算
- 位串行标准功能

1.6.2.2 表达式的解释顺序

表达式的解释顺序基于以下：

使用的运算符的优先级别

- 从左至右原则
- 括号的使用（对于同级别的运算符）

根据特定的规则处理表达式：

- 根据优先级别来执行运算符（见 129 页运算符优先级别表）
- 同级别的运算符从左至右执行
- 标识符前面的负号表示乘以— 1
- 一个算术运算符不能立即跟另外一个

表达式 a * -b 是无效的，但是 a * (-b)是允许的

- 括号重写运算符优先顺序，如：括号有最高优先级
- 带括号的表达式被视为单独的运算符，最先算
- 左括号的数量必须与右括号的数量一致
- 算术运算符不嫩滑用做字符或逻辑数据。所以表达式 (n<=0) + (n<0)是无效的

表达式的例子

```
testVar    // Operand
A AND   (B)    // Logic expression
A AND   (NOT B)   // Logic expression with negation
(C)   <   (D)   // Relational expression
3+3*4/2   // Arithmetic expression
```

1.6.3 运算对象

定义：运算对象是指可以用作写成表达式的对象。运算对象可以通过语法图表示：

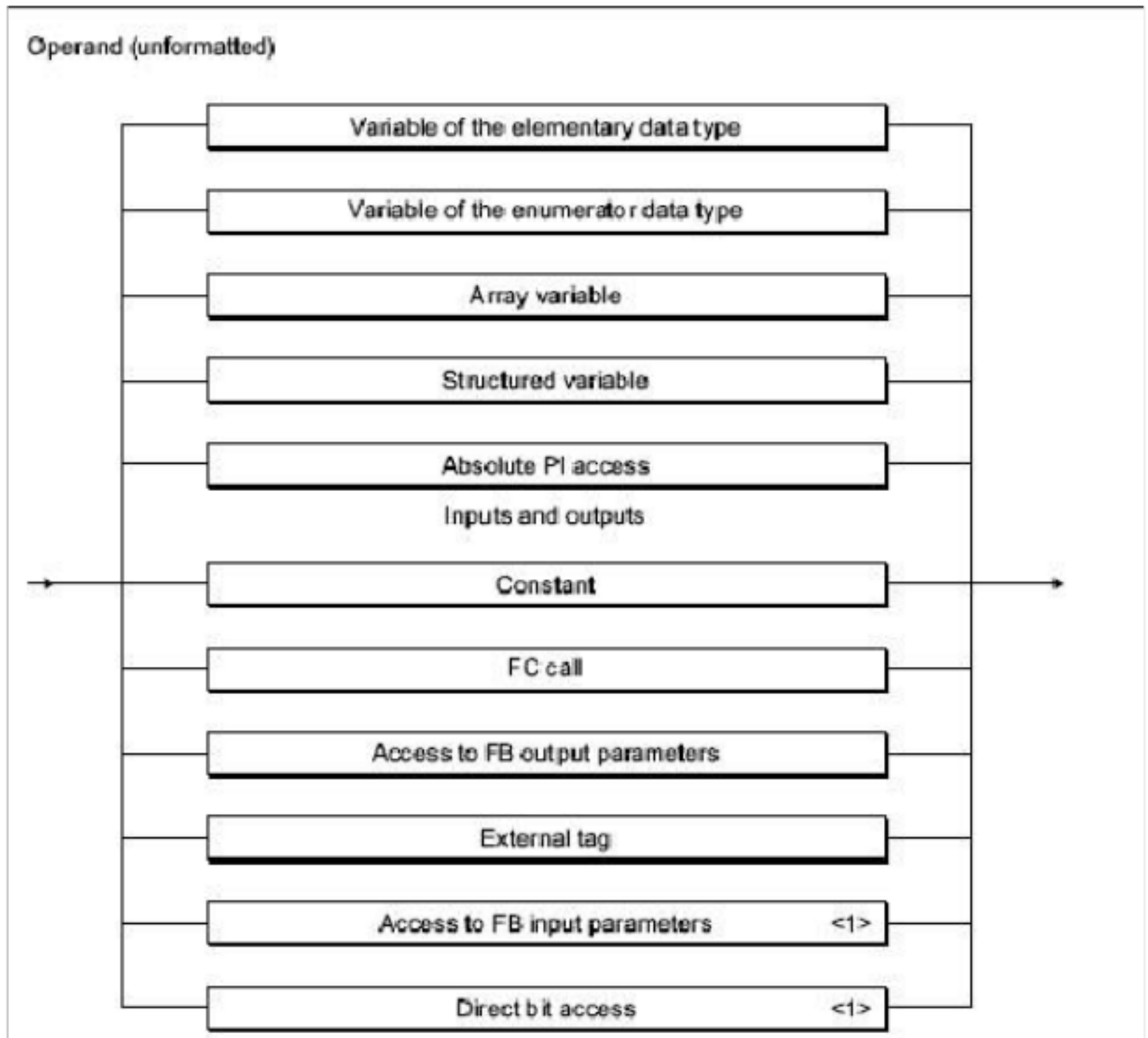


Figure 3-21 Syntax: Operand

Table 3-28 Examples of operands

intVar
5
%I4.0
PI
NOT TRUE
axis1.motionStateData.actualVelocity

1.6.4 算术表达式

一个算术表达式是指由算术运算符形成的一个表达式。这些表达式允许处理数字型的数据类型。

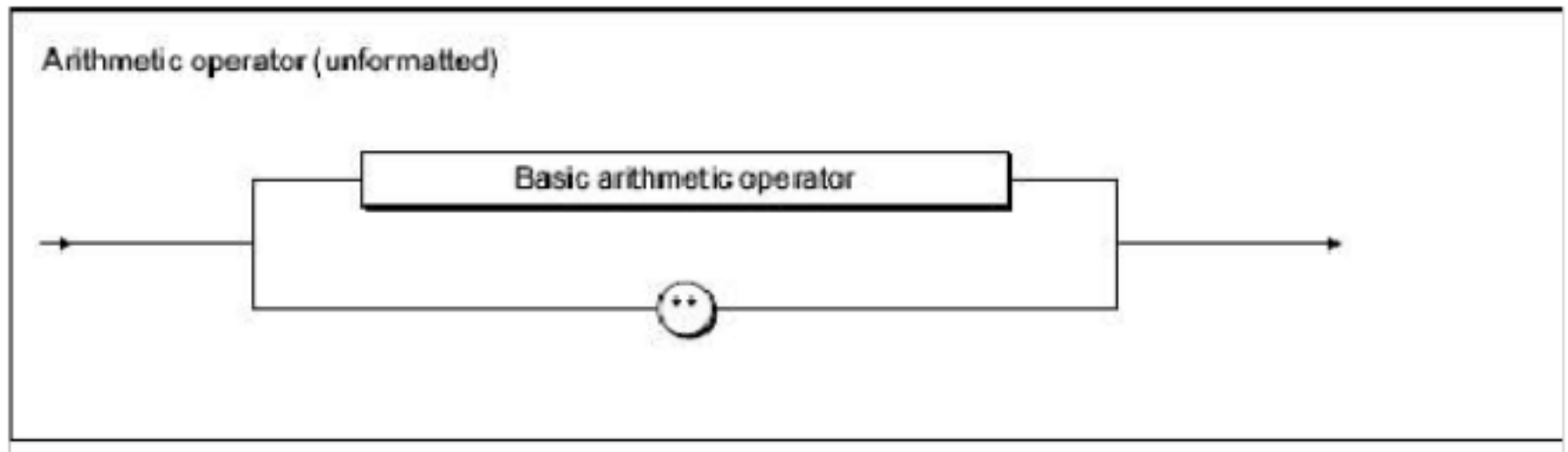


Figure 3-22 Syntax: Arithmetic operator

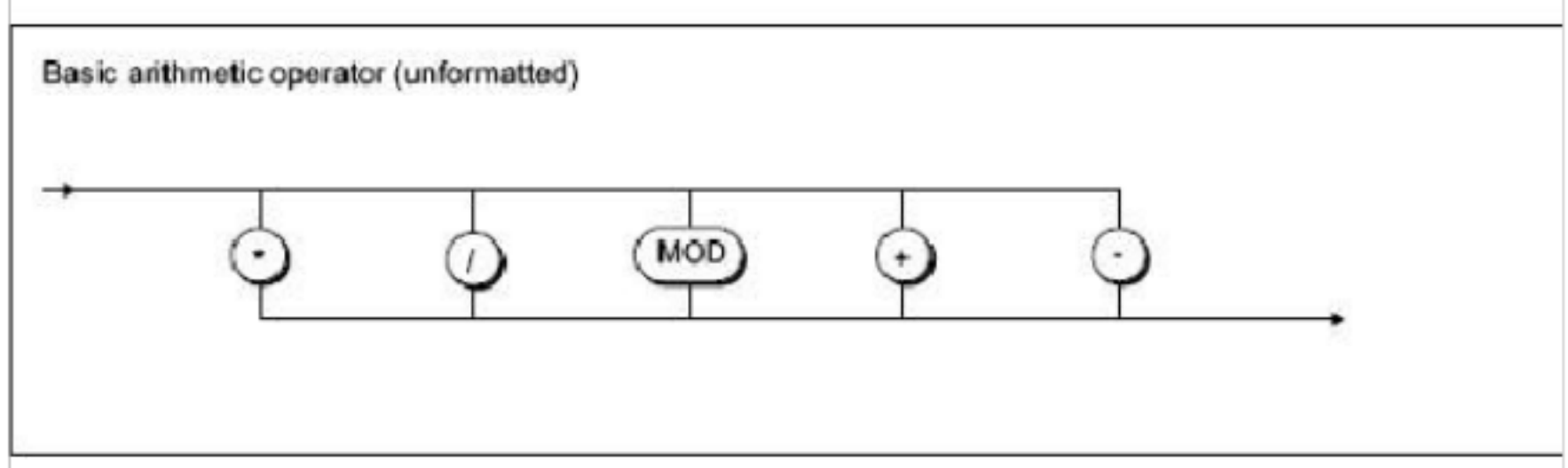


Figure 3-23 Syntax: Basic arithmetic operator

下表显示了每个算术运算符

●算术运算符

●允许的运算符的数据类型

●结果的数据类型

使用了一些通用的数据类型（第 92 页）

注意

使用数字功能进行更多的操作是可行的， 见 SIMOTION基本功能功能手册中标准数字功能

建议使用括号来表示附属，尽管有时不需要，但是是为了提高辨识性

算法运算符根据各自对应的级别（ 129 页）进行

	运算符	数据类型		
		第 1 个运算目标	第 2 个运算目标	第 3 个运算目标
Exponential	**	ANY_REAL2	ANY_REAL	ANY_REAL3
Unary minus	-	ANY_NUM	(None)	ANY_NUM
Multiplication	*	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT4	ANY_BIT4	ANY_BIT
		TIME	ANY_NUM	TIME
Division	/	ANY_NUM	ANY_NUM5	ANY_NUM
		ANY_BIT4 A	ANY_BIT4 5	NY_BIT
		TIME	ANY_NUM5	TIME
		TIME	TIME5	UDINT

Modulo division	MOD	ANY_INT	ANY_INT5	ANY_INT
		ANY_BIT4	ANY_BIT4 5	ANY_BIT
Addition	+	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT4	ANY_BIT4	ANY_BIT
		TIME	TIME	TIME6
		TOD	TIME	TOD6)
		DT	TIME	DT7
Subtraction	-	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT4	ANY_BIT4	ANY_BIT
		TIME	TIME	TIME
		TOD	TIME8	TOD
		DATE	DATE	TIME9
		TOD	TOD	TIME9
		DT	TIME	DT
		DT	DT	TIME9

1.结果的数据类型由运算目标的最强数据类型决定。

2.第一个运算目标必须大于 0

除了 SIMOTION Kernel4.1版中

—如果第二个运算对象是整数，第一个运算对象必须小于 0

—如果第二个运算对象是整数，第一个运算对象可以等于 0

下列 SIMOTION Kernel4.1版中的应用：如果第一个运算对象等于 0，会得到一个带执行错误任务的错误信息

3.第一个运算目标的数据类型

4.除了布尔数据类型。使用同等字宽不带符号的整数来计算。

5.第二个运算目标必须不等于 0

6.加法：也有可能溢出

7.带日期纠正的加法

8.在计算前从 TIME 到 TOD的限制

9.这些操作是基于 TIME 数据类型的最大值的模数

注意：如果在运算一般的 ANY_REAL数据类型的变量时超出数值范围， 结果将根据 IEEE754 包含相同的位模式。

为了检查是否超出数值范围， 可以通过使用功能—— finite（见 SIMOTION基本功能功能手册）来检测结果

1.6.4.1 算术表达式的例子

带数字的算术表达式的例子

假设 i 和 j 是整数变量（ INT 数据类型），值为 11 和-3，一些整数表达式和对应的数值如下所示：

表达式	值
i + j	8
i - j	14
i * j	-33
i MOD j	-2
i / j	-3

有时间说明的有效的算术表达式例子

变量	内容	数据类型
t1	T#1D_1H_1M_1S_1MS	TIME
t2	T#2D_2H_2M_2S_2MS	TIME
d1	D#2004-01-11	DATE
d2	D#2004-02-12	DATE
tod1	TOD#11:11:11.11	TIME_OF_DAY
tod2	TOD#12:12:12.12	TIME_OF_DAY
dt1	DT#2004-01-11-11:11:11.11	DATE_AND_TIME
dt2	DT#2004-02-12-12:12:12.12	DATE_AND_TIME

下面示范了一些带变量和值的表达式

表达式	值
t1 + t2	T#3D_3H_3M_3S_3MS
dt1 + t1	DT#2004-01-12-12:12:12.111
t1 - t2	T#48D_16H_1M_46S_295MS
t1 * 2	T#2D_2H_2M_2S_2MS
t1 / 2	T#12H_30M_30S_500MS
DATE_AND_TIME_TO_TIME_OF_DAY(dt1)	TOD#11:11:11.110
DATE_AND_TIME_TO_DATE(dt1)	D#2004-01-11

1.6.5 关系表达式

定义：关系表达式是指带关系运算符的 BOOL数据类型的表达式

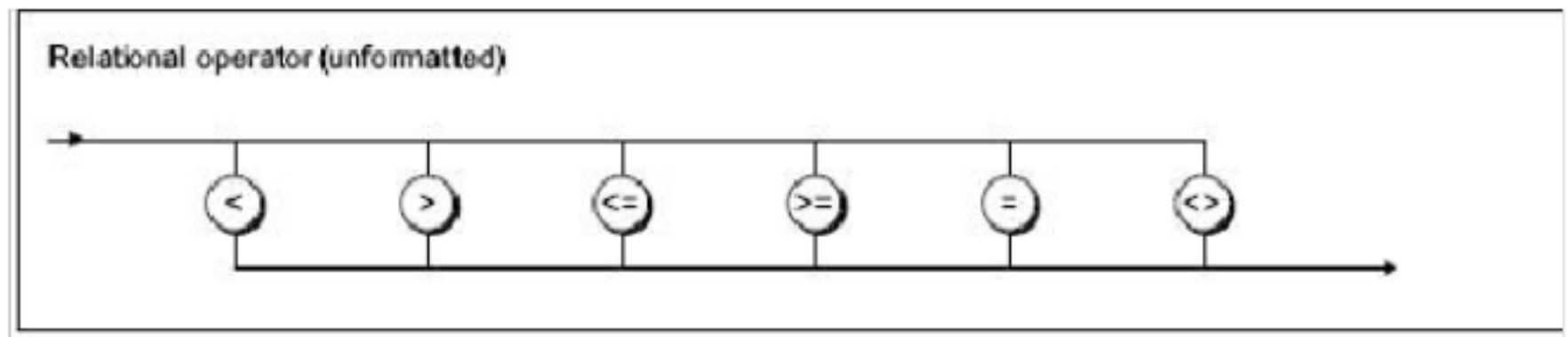


Figure 3-24 Syntax: Relational operators

关系运算符比较两种运算目标的值然后得出布尔值作为结果

第一个运算对象　运算符　第二个运算对象　->布尔值

运算符	意义
>	第一个运算目标大于第二个运算目标
<	第一个运算目标小于第二个运算目标
>=	第一个运算目标大于或等于第二个运算目标
<=	第一个运算目标小于或等于第二个运算目标
=	第一个运算目标等于第二个运算目标
<>	第一个运算目标不等于第二个运算目标

关系表达式的结果为：

- 1（TRUE）：当满足对比时
- 0（FALSE）：当不满足对比时

下表列出了允许的两个运算对象和关系运算符的数据类型的组合

数据类型		允许的关系运算符
第一个运算对象	第二个运算对象	
ANY_NUM	ANY_NUM1	<, >, <=, >=, =, <>
ANY_BIT	ANY_BIT	<, >, <=, >=, =, <>
DATE	DATE	<, >, <=, >=, =, <>
TIME_OF_DAY (TOD)	TIME_OF_DAY (TOD)	<, >, <=, >=, =, <>
DATE_AND_TIME (DT)	DATE_AND_TIME (DT)	<, >, <=, >=, =, <>
TIME	TIME	<, >, <=, >=, =, <>
STRING	STRING2	<, >, <=, >=, =, <>
Enumerator data type	Enumerator data type3	=, <>
ARRAY	ARRAY3	=, <>
Structure (STRUCT)	Structure (STRUCT)3	=, <>
1.两种运算对象必须通过隐式转化为最强的数据类型（见 141 页基础数据类型转化和SIMOTION基本功能功能手册的数字数据类型和位数据类型的转化功能） 2.不考虑声明的字符串长度，字符串数据类型的变量可以进行比较		

为了比较两种不同长度的字符串数据类型的变量，　　短的字符串通过在右侧插入　$00 扩展为长字符串的长度。比较基于　ASCII编码中对应的字符串从左至右进行。
例如：’ABC’ < ’AZ’ < ’Z’ < ’abc’ < ’az’ < ’z’
3.第一个运算目标的数据类型

关系标法是和　BOOL 数据类型的变量或常量可以一起组合成带逻辑运算符的逻辑表达式 (见 127 页逻辑表达式)。 这启用了查询的实运用，如　　If a < b and b < c, then ….

注意：关系表达式运算符在表达式中比逻辑表达式级别要高。因此如果它们自身为逻辑表达式或位串行表达式，关系表达式的运算目标必须放于括号中。

注意当比较　REAL或 LREAL变量时会产生错误

```
IF A   =    2 THEN
//...
END_IF;
var_1    := B    < C; // var_1 of BOOL data type
IF D   < E OR var_2 THEN    // var_2 of BOOL data type
       //     ...
END_IF;
```

1.6.6　逻辑表达式和位串行表达式

定义：带有逻辑运算符　AND, &, XOR和　OR, 可以把运算目标和一般的数据类型　　ANY_BIT (BOOL, BYTE, WORD, or DWORD)的表达式组合。如果是带有逻辑运算符　　NOT, 那么可以否定运算目标和　ANY_BIT数据类型的表达式

下表提供了可用的运算符的信息

命令	运算符	第 1 个运算目标	第 2 个运算目标	结果
Negation	NOT	ANY_BIT	-	ANY_BIT
Conjunction	AND or &	ANY_BIT	ANY_BIT	ANY_BIT
Exclusive disjunction	XOR	ANY_BIT	ANY_BIT	ANY_BIT
Disjunction	OR	ANY_BIT	ANY_BIT	ANY_BIT
结果的数据类型由最强的运算目标的数据类型决定				

表达式被指定为

●一个逻辑表达式，如果只使用了　　BOOL数据类型的运算目标
运算符对下列表格中的运算目标有影响。逻辑表达式的结果为　　1（TRUE）或者 0(FALSE)

●位串行表达式，如果使用了　BYTE, WORD, 或 DWORD数据类型的运算目标。
运算符对下表中的单独的字节运算目标有影响

运算目标 (数据类型	结果 (数据类型　BOOL)

BOOL)						
a	b	NOT a	NOT b	a AND b a & b	a XOR b	a OR b
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1

例子：

表达式 (let n = 10)	值
(n>0) AND (n<20)	TRUE
(n>0) AND (n<5)	FALSE
(n>0) OR (n<5)	TRUE
(n>0) XOR (n<20)	FALSE
NOT ((n>0) AND n<20))	FALSE

位串行表达式

表达式	值
2#01010101 AND 2#11110000	2#01010000
2#01010101 OR 2#11110000	2#11110101
2#01010101 XOR 2#11110000	2#10100101
NOT 2#01010101	2#10101010

查询中的表达式 (let value1 be 2#01, let value2 be 2#11)

IF (value1 AND value2) = 2#01 THEN...

条件为 TRUE因为位串行变为 2#01.

1.6.7 运算符的优先级

表达式的公式的一些基本的规则描述在 119 页表达式中。下表列出了在表达式中的单独运算符的优先级别。

命令	符号	优先级别
Parentheses	(Expression)	最高级
Function evaluation	标识符 (argument list) 如. LN(a), EXPT (a,b) etc	
Negation Component	- NOT	
Exponentiation	**	

Multiplication Division Modulo	* / MOD	
Addition Subtraction	+ -	
Comparison	<, >, <=, >=	
Equal Not Equal	= <>	
Boolean AND	&, AND	
Boolean EXCLUSIVE OR	XOR	
Boolean OR	OR	最低级

1.7 控制语句

源文件部分不能被编程，所有的语句以开始到结束的顺序被执行。通常，只有当条件为TRUE时才会执行一些语句，某些语句会被重复执行（循环）。在源文件部分中的程序控制语句是完成这个的方式。

1.7.1 IF　　语句

IF 语句是条件语句。如果指定了一个或者多个选项，选择语句部分的一个（或无）来执行。

当执行条件语句时，估算指定的逻辑表达式。如果表达式的值为　　TRUE, 条件实现。如果只为 FALSE,　条件没实现。

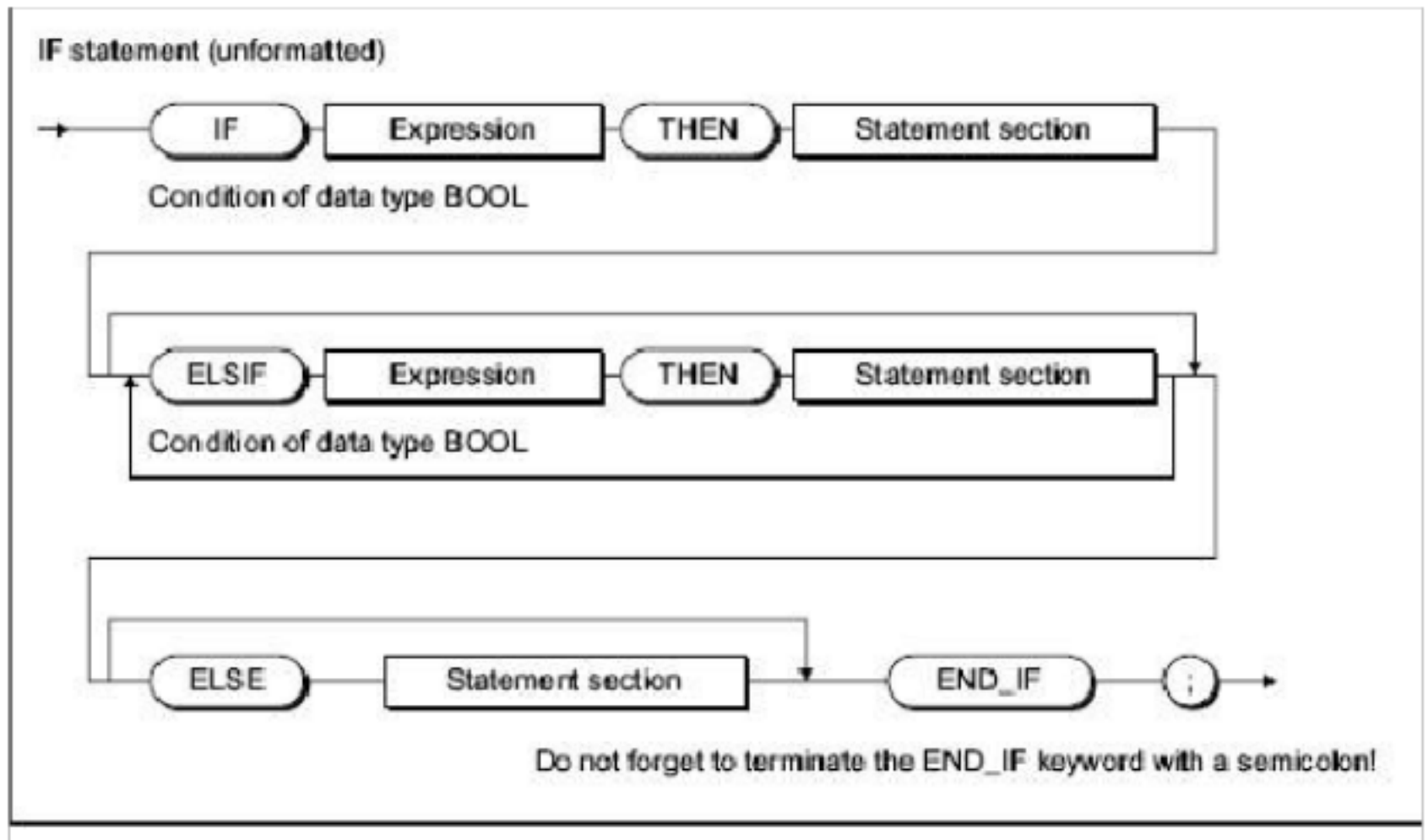


Figure 3-25 Syntax: IF statement

根据下列规则处理 IF 语句：

1.如果第一个表达式的值为 TRUE，执行在 THEN之后的语句
在 END_IF之后恢复程序。

2.如果第一个表达式的值为 FALSE，ELSIF分支的表达式将估算。如果 ELSIF分支的一个布尔表达式为 TRUE, 执行在 THEN之后的语句
在 END_IF之后恢复程序。

3. 如果没有 ELSIF分支的布尔表达式为 TRUE，执行在 ELSE之后的语句（或，如果没有 ELSE 分支，无更多的语句可执行）
在 END_IF之后恢复程序。

ELSIF语句的任意数字都可以被编程。

注意可以没有任意的 ELSIF分支和 /或 ELSE分支。可以用同样的方式解释犹如没有语句证明分支的存在。

使用一个或多个 ELSIF分支的优势， 而不是 IF 语句的顺序， 优势在于遵循有效表达式的逻辑表达式不再需要估算。这将缩短程序要求的处理时间，阻止执行不需要的程序路径。

```
IF A=B THEN
n:=  0;
END_IF;
IF temperature  <  5.0 THEN
      %Q0.0  := TRUE;
ELSIF temperature  >  10.0 THEN
      %Q0.2  := TRUE;
```

```
ELSE
%Q0.1    := TRUE;
END_IF;
```

1.7.2 CASE　　语句

CASE语句被用作从　N 中程序选项中选择　1

这种选择决定了一个选择表达式（选择器）

●一般数据类型　ANY_INT的表达式

●枚举数据类型的变量

选择来自于一列表的数值（数值列表），程序的一部分被指定为每个数值或数组。

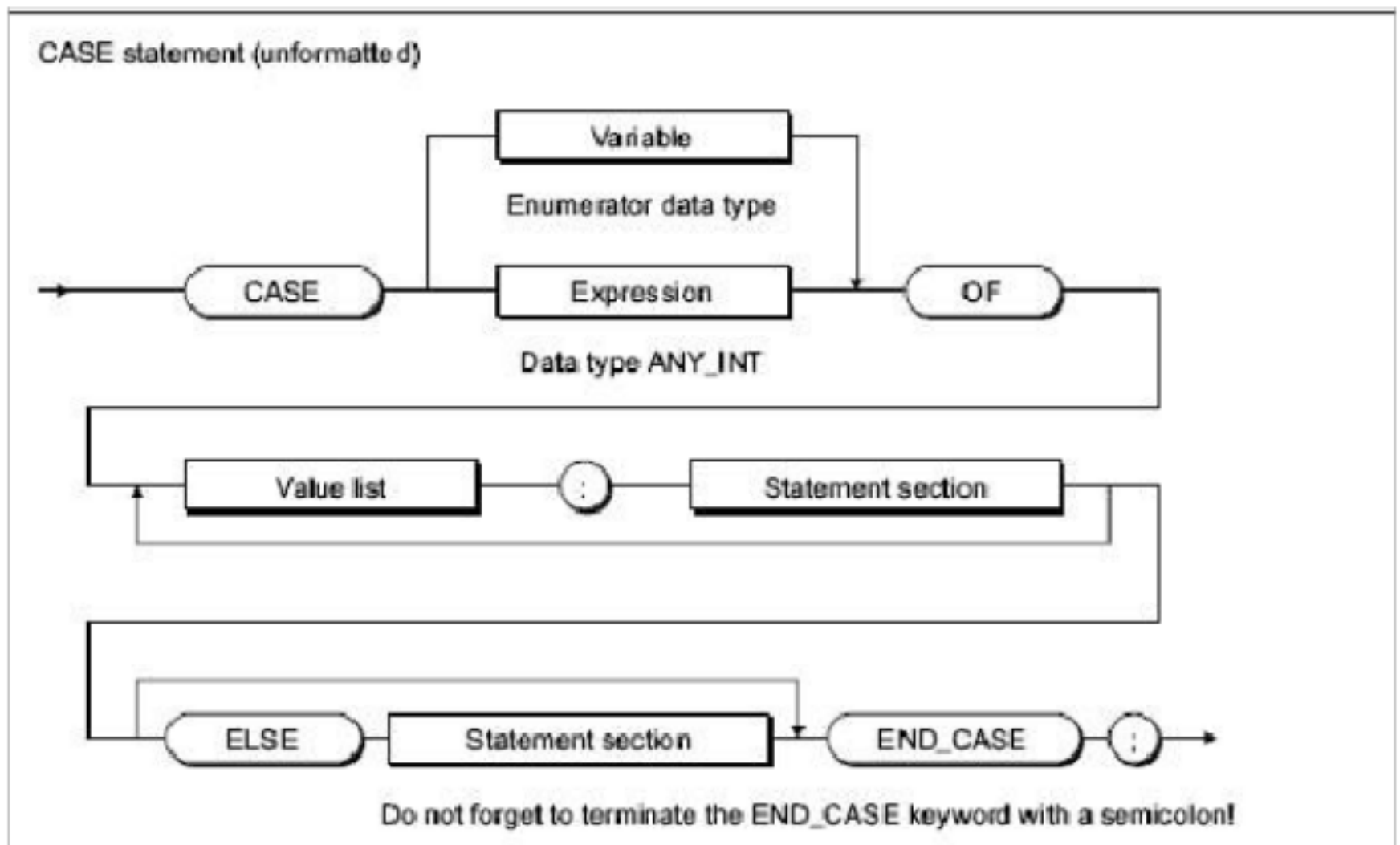


Figure 3-26　Syntax: CASE statement

CASE语句根据下列规则被执行。

1.计算选择表达式（选择器）。必须回送一般数据类型　　ANY_INT（整数）或枚举数据类型的数值。

2.检查以决定选择器数值是否包含在数值列表中。　　每个列表中的数值代表了选择表达式允许的数值之一。

3.如果发现匹配，执行列表中被指定的程序部分

4.ELSE分支是可选的。如果没有发现匹配就执行。

5.如果 ELSE分支确实，没有发现匹配，在　END_CASE之后恢复程序。

数值列表包含选择表达式允许的数值。

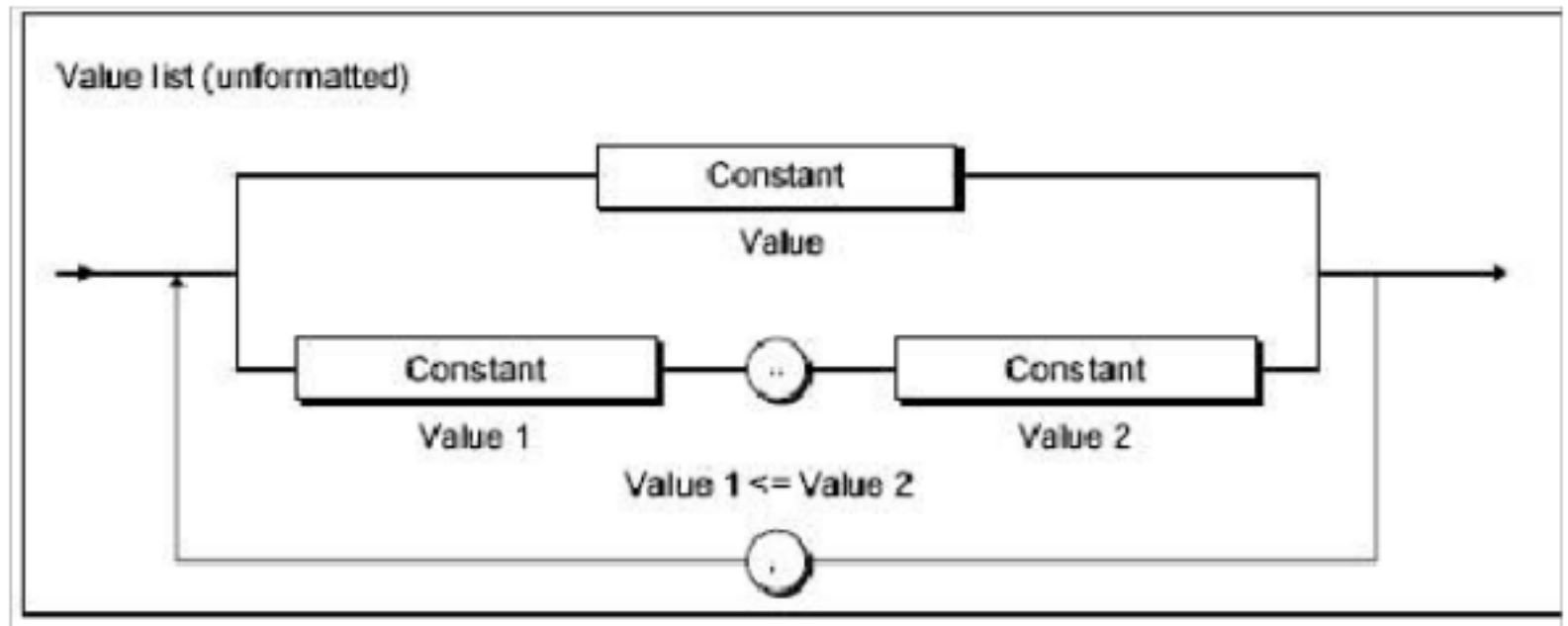


Figure 3-27 Syntax: Value list

当用公式表达数值列表时，注意以下：

●每个数值列表可以由一个常量（值），一个常量列表（值 1，值 2，值 3 等），或一个常量范围（值 1 到值 2）开始。

●数值列表中的数值必须为整数常量或选择器中枚举数据类型的常量 / 元素

注意：

一个数值应该在 CASE语句中的数值列表中仅出现一次。

如果一个数值多次出现，编辑器将产生报警，仅仅与首次产生数值所在的数值列表一直的语句部分会被执行。

下面的例子说明了 CASE语句的使用

```
CASE intVar OF
1     : a   :=   1;
2,3   : b   :=   1;
4..9   : c   :=    1; d:=2;
ELSE
e   :=   5;
END_CASE;
```

1.7.3 FOR 语句

一个 FOR 语句或者一个重复语句在一个循环执行一系列的语句，此时数值会在每次通过时被指定给一个变量（一个计数变量）。计数变量必须为 SINT,INT或 DINT 类型的本地变量。

带 FOR的一个循环的定义包括起始值和结束值的说明。两个变量必须为同计数变量一样的数据类型

注意：

在编程阶段循环通过的次数可知时， 你可以使用 FOR语句。如果循环通过次数未知， WHILE 或 REPEAT语句更适合（见 136 页 WHILE语句和 137 页 REPEAT语句）

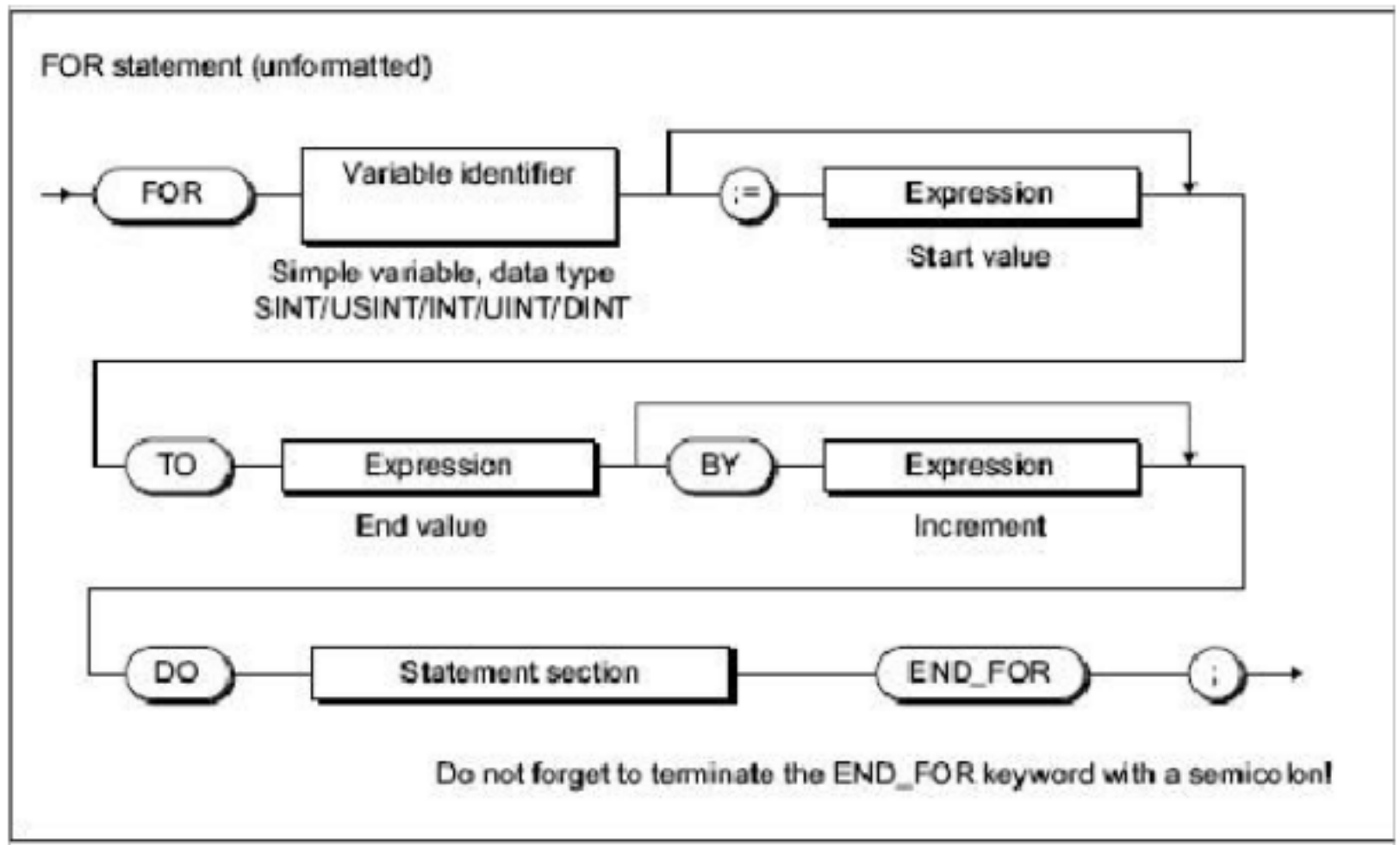


Figure 3-28 Syntax: FOR statement

1.7.3.1 处理 FOR 语句

根据下列规则处理 FOR语句

1.在循环开始时，计数变量被设为起始值，在每次循环通过后会以指定的增量增大（正增量）或减小（负增量），知道达到结束值。在通过第一次循环之后，起始值被认为是当前值。

2.在每次通过时，系统将检查下列条件是否为 TRUE
起始值或当前值 <= 结束值 (正增量) 或者
起始值或当前值 >= 结束值 (负增量)
如果条件满足，执行语句
如果条件不满足，跳过循环和语句顺序，在 END_FOR之后恢复程序。

3.因为第 2 不步如果没有执行 FOR循环，计数变量保持当前值。

1.7.3.2 FOR 语句规则

下列规则适用于 FOR语句

- 可以省略 BY（增量）说明。如果没有指定增量，那么默认为 +1
- 起始值，结束值和增量为表达式 （见 119 页表达式）。在开始 FOR语句时会估算表达式。
- 如果起始值和结束值为 DINT 数据类型，数值（结束值—起始值）必须小于双整数的最

大值的范围，即小于　2**31-1

●只执行选择器中第一个选择语句
●计数变量包含引起退出循环的数值，如：在退出循环前增加
●不允许在执行循环时改变结束值和增加值

1.7.3.3FOR 语句例子

```
FOR k  :=   1 TO   10 BY  2 DO
l:=l+1;
//    ...
END_FOR;
```

1.7.4WHILE　语句

在控制迭代条件的情况下，　WHILE语句允许重复执行语句顺序。　迭代条件根据逻辑表达式的规定被写成公式。

注意：在编程阶段，当循环通过次数未知时，可以使用　　WHILE语句。如果次数已知，　FOR语句更为适合（见　134 页 FOR语句）。

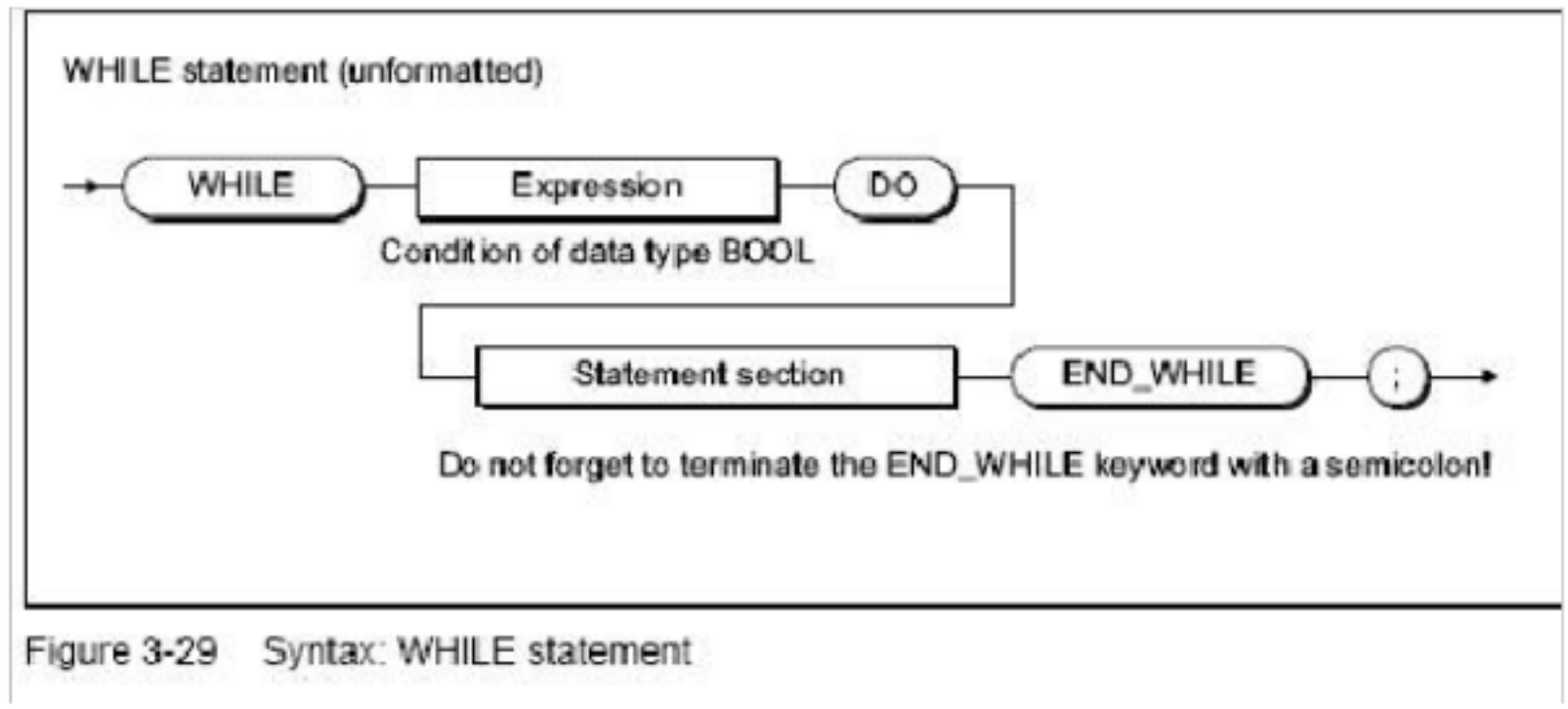


Figure 3-29　Syntax: WHILE statement

重复 DO 之后的语句部分直到迭代条件中有　　TRUE. 根据下列规则处理　WHILE语句：

1.每次在执行语句部分之前估算迭代条件
2.如果值为　TRUE，执行语句部分
3.如果值为　FALSE,结束　WHILE语句（第一次出现在估算条件时），在　　END_WHILE之后恢复程序。

```
WHILE Index  <=   50 DO
Index:= Index   +   2;
END_WHILE;
```

1.7.5REPEAT　语句

REPEAT语句引起重复执行　REPEAT到 UNTIL之间顺序的语句，直到结束条件为　TRUE。结束条件根据逻辑表达式的规定被写成公式

注意：在编程阶段，　当循环通过次数未知时，　可以使用 REPEAT语句。如果次数已知，　FOR语句更为适合（见　134 页 FOR语句）。

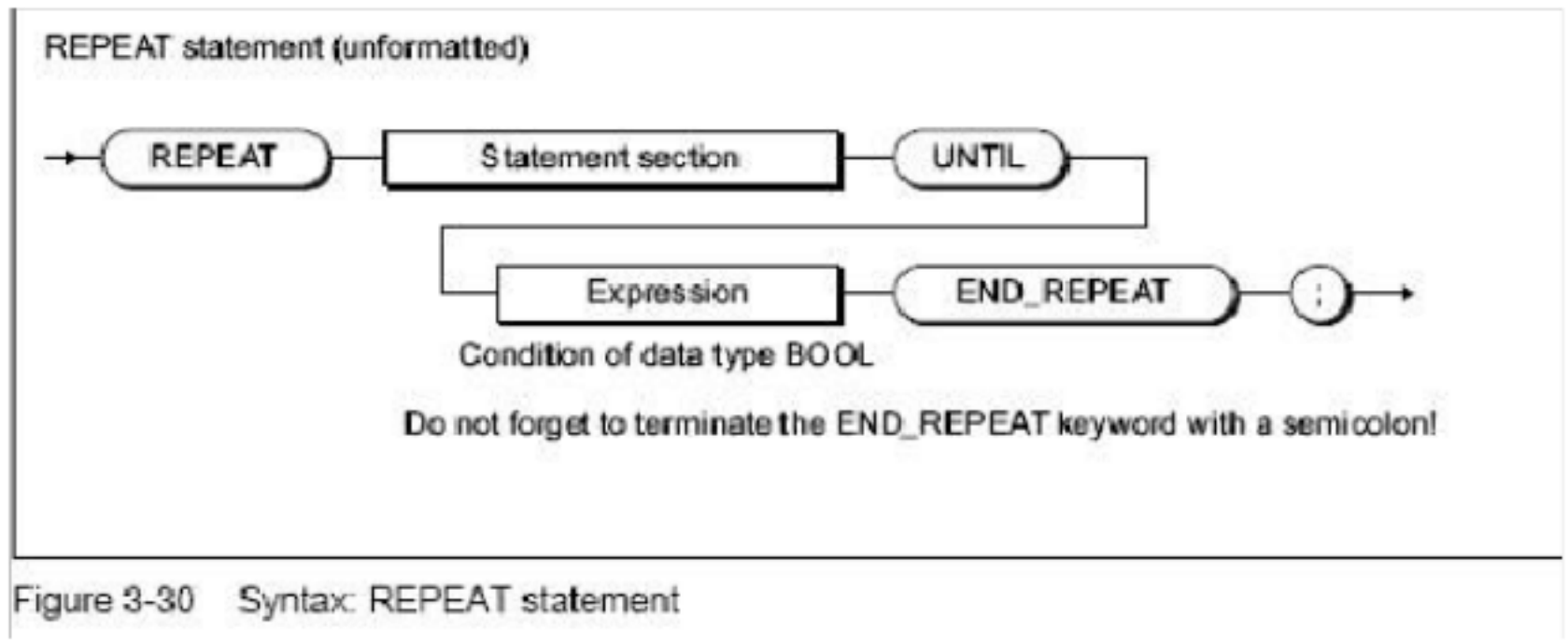


Figure 3-30　Syntax: REPEAT statement

在执行完语句部分之后检查条件。这意味着语句部分至少要执行　　1 次，即便在开始时结束语句为 TRUE.

1.每次在执行语句部分之后估算迭代条件

2.如果值为 FALSE，再次执行语句部分

3.如果值为 TRUE, 结束 REPEAT语句的执行，在　END_REPEAT之后恢复程序执行。

```
Index:=　1;
REPEAT
Index:= Index　+　2;
UNTIL Index　>　50
END_REPEAT;
```

1.7.6EXIT　　语句

EXIT语句被用作在任何点退出一个循环（　FOR, WHILE 或 REPEAT循环），不考虑结束条件是否为 TRUE或 FALSE。

这个语句对在　EXIT语句直接跳出循环油影响。

在结束循环之后恢复程序。（如　:END_FOR之后）

```
Index:=    1;
FOR Index  :=    1 to    51 BY   2 DO
       IF   %I0.0 THEN
EXIT;
END_IF;
END_FOR;
// The following value assignment is made after the execution of EXIT // or after the regular
end of the FOR loop
// For the execution:
Index_find    := Index_2;
```

1.7.7RETURN　语句

RETURN语句引起结束现在运行（程序，功能，功能块）的　　POU。

当结束一个功能或功能块时，在调用功能或功能块所在的位置之后，在高级别的　　POU将继续执行程序。

```
Index:=    1;
FOR Index  :=    1 to    51 BY   2 DO
       IF   %I0.0 THEN
RETURN;
END_IF;
END_FOR;
// The following value assignment is made after the regular end
// of the FOR loop for the execution, however, not after the execution // of RETURN:
Index_find    := Index_2;
```

1.7.8WAIFORCONDITION　语句

可以使用　WAIFORCONDITION语句来等待可编程事件或在　　MOTIONTask 中的条件。调用 MotionTask 的执行语句被暂停，直到条件为　　TRUE可以在表达式（ 166 页）中对此条件进行编程。更多的　WAIFORCONDITION和表达式的信息，见　SIMOTION MOTION 控制基本功能功能手册。

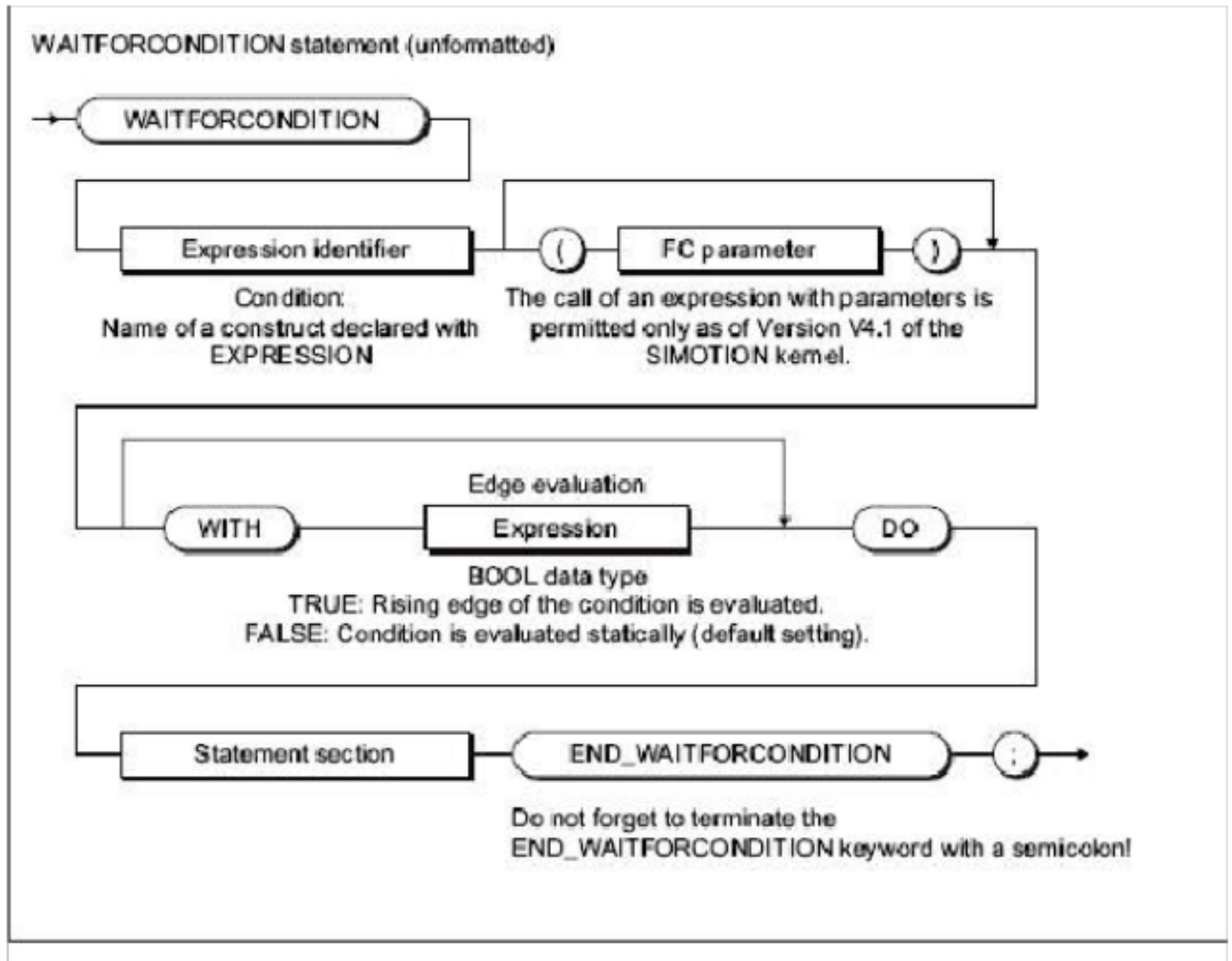


Figure 3-31 Syntax: WAITFORCONDITION statement

表达式标识符使用 EXPRESSION 声明的结构：它的数值定义（如有需要，同 WITH edge 计算，如有必要）是否满足条件。

WITH edge evaluation 顺序是可选的。 Edge evaluation 是 BOOL 数据类型的表达式， 决定了如何干扰表达式标识符。

- Edge evaluation=TRUE: 干扰表达式标识符的上升边缘。如：当表达式标识符从 FALSE 改为 TRUE 时，条件满足
- Edge evaluation=FALSE: 干扰表达式标识符的静态值。如：当表达式标识符为 TRUE 时，满足条件。

如果没有说明 WITH edge evaluation，默认设置为 FALSE。如：估算表达式标识符的静态值

语句部分必须包括至少一个语句（空语句也算）

```
//    ...
// Call the statement with name of expression
WAITFORCONDITION myExpression WITH TRUE DO
// Here, at least one statement will be executed with higher priority,
e.g.
%Q0.0   := TRUE;
END_WAITFORCONDITION;
//    ...
```

完整的例子，见表达式的描述（ 166 页）

1.7.9GOTO 语句

GOTO语句将引起至语句说明的跳转标签的跳转。（见 250 页跳转语句和标签）

使用 GOTO 语句编写跳转语句，指定跳转标签至你想要跳转的位置。跳转只允许在 POU 之内。

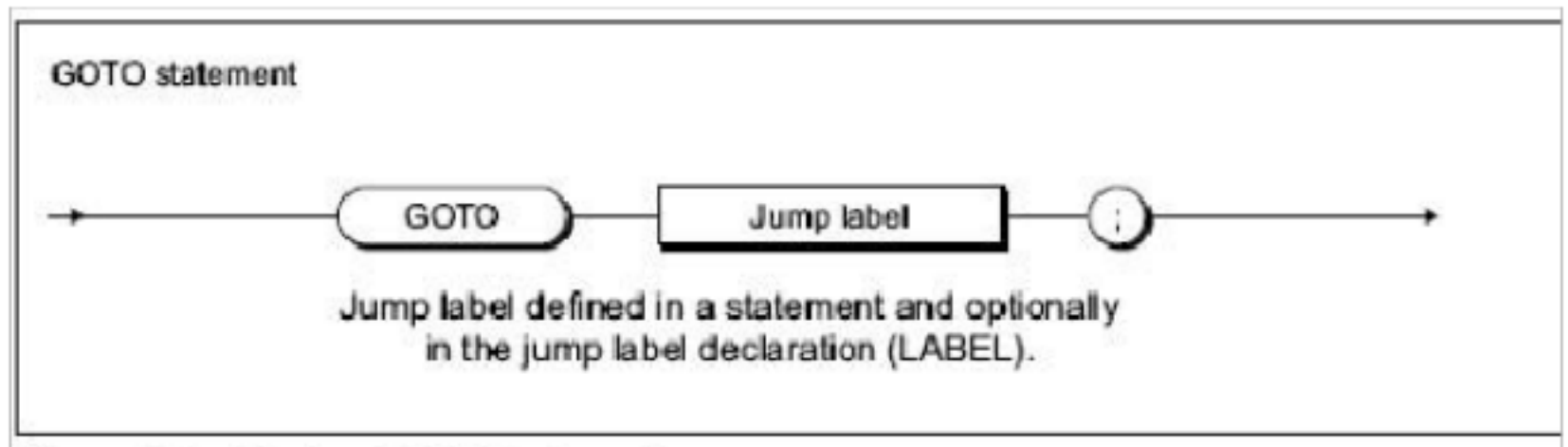


Figure 3-32 Syntax: GOTO statement

注意：
在特定的情况 (如：调试检修)中只能使用 GOTO语句。根据结构编程的规则不能使用 GOTO 语句。

跳转只允许在 POU之内。

下列的跳转是非法的

- 跳转至下属的控制结构（ WHILE,FOR 等）
- 从 WAITFORCONDITION结构来的跳转
- 在 CASE语句中的跳转

跳转标签只能在使用的 POU中被声明。如果声明了跳转标签， 只能使用声明的跳转标签。

1.8 数据类型转换

此节描述了在基础数据类型之间的显式和隐式转换。也包括额外的转换可能的概述。

1.8.1 基础数据类型转换

下表显示在数字数据类型和字节数据类型之间的转换选项。下列是明显的转换选项。

●隐式转换：当在一个表达式中使用不同的数据类型或编辑器指定数值时自，自动转换
●显式转换：当用户调用一个转换功能时执行转换（见　　SIMOTION基本功能功能手册）

Source data type	Target data type												
	BOOL	BYTE	WORD	DWORD	USINT	UINT	UDINT	SINT	INT	DINT	REAL	LREAL	STRING
BOOL	-	Im/Ex	Im/Ex	Im/Ex	Val	Val	Val	Val	Val	Val	Val	Val	
BYTE	Ex	-	Im/Ex	Im/Ex	Ex	Ex	Ex	Ex	Ex	Ex	Val	Val	Elem
WORD	Ex	Ex	-	Im/Ex	Ex	Ex	Ex	Ex	Ex	Ex	Val	Val	
DWORD	Ex	Ex	Ex	-	Ex	Ex	Ex	Ex	Ex	Ex	Ex/Val	Val	
USINT	Val	Ex	Ex	Ex	-	Im/Ex	Im/Ex	Ex	Im/Ex	Im/Ex	Im/Ex	Im/Ex	
UINT	Val	Ex	Ex	Ex	Ex	-	Im/Ex	Ex	Ex	Im/Ex	Im/Ex	Im/Ex	
UDINT	Val	Ex	Ex	Ex	Ex	Ex	-	Ex	Ex	Ex	Ex	Ex	Ex
SINT	Val	Ex	Ex	Ex	Ex	Ex	Ex	-	Im/Ex	Im/Ex	Im/Ex	Im/Ex	
INT	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	-	Im/Ex	Im/Ex	Im/Ex	
DINT	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	Ex	-	Ex	Im/Ex	Ex
REAL	Val	Val	Val	Ex/Val	Ex	Ex	Ex	Ex	Ex	Ex	-	Im/Ex	Ex
LREAL	Val	Val	Val	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	-	Ex
STRING	-	Elem	-	-	-	-	Ex	-	-	Ex	Ex	Ex	

Im：可能的隐式数据类型转换
Ex：使用 Quelldatentyp_TO_Zieldatentyp 类型转换功能的可能的显式数据类型转换
　　使用 Quelldatentyp_VALUE_TO_Zieldatentyp 类型转换功能的可能的显式数据类型转换
Elem：使用字符串数据类型元素的隐式数据类型转换

1.8.1.1 隐式数据类型转换

如果扩大的数值范围没有引起任何数值漏失，　　隐式类型的转换是可能的。　如：从 REAL 到 LREAL或 从 INT 到 REAL。结果通常被定义。

下表提供了所有隐式转换链的基于图标的视图。在类型转换链中的每个阶段（从左至右或从上至下读）通常代表数值范围的扩大。

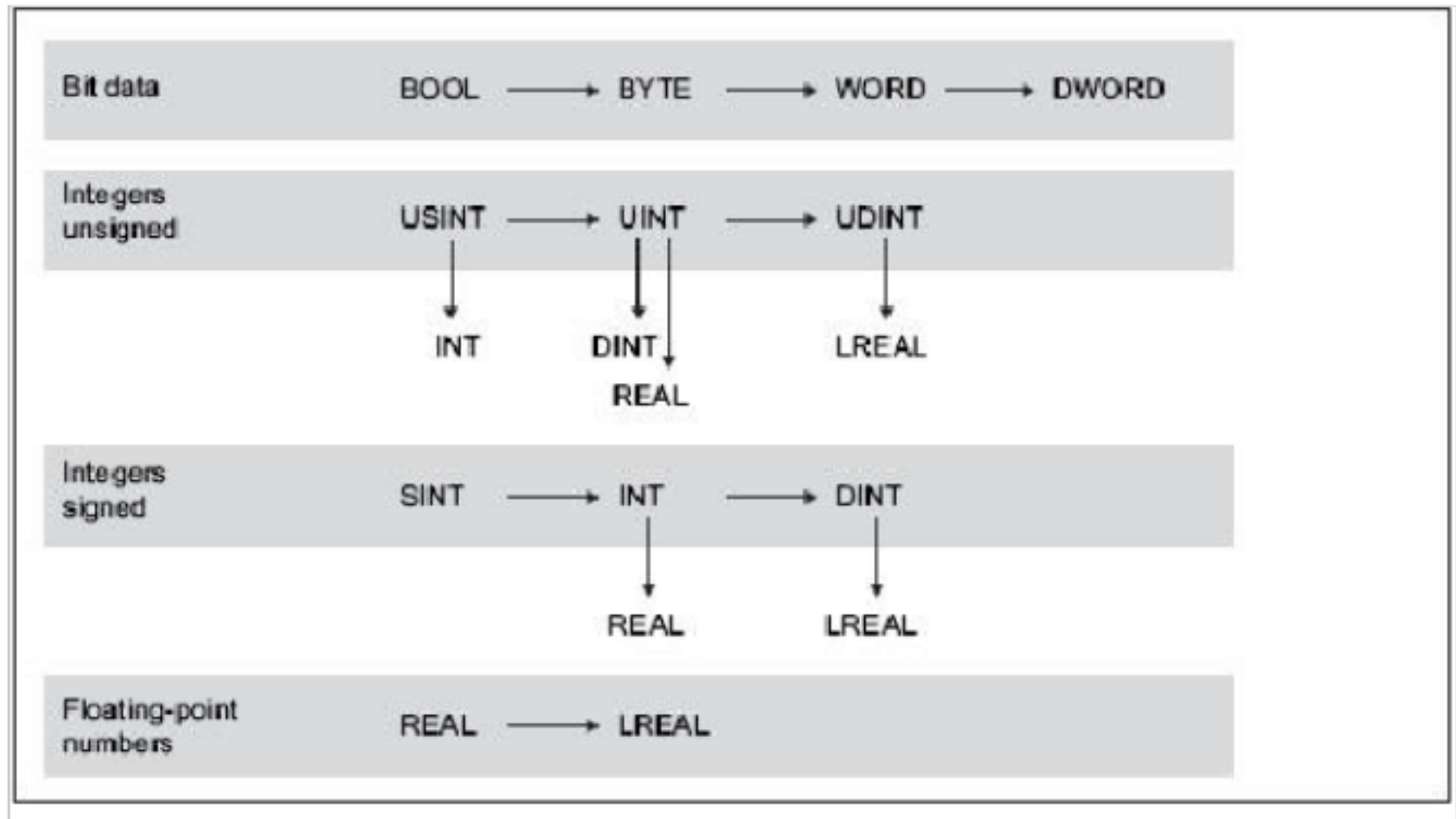


Figure 3-33 Implicit type conversion chains (one or more levels from left to right or one level from top to bottom)

下列隐式类型转换支持：

1.超过一级或多级的水平式（从左至右） (e.g. USINT to UDINT)

2.超过一级的垂直式（从上到下） (e.g. UINT to REAL)

隐式类型转换可以通过下列顺序组合 (e.g. INT to LREAL).

所有其他的类型转换不能隐式执行 (e.g. UDINT to REAL) 意味着必须使用显式功能

注意：

在算法表达式中，结果通常以表达式中包含的最大数值公式计算。

数值仅可以被指定给表达式，如果：

- 计算的表达式和指定的变量是同种数据类型
- 计算的表达式的数据类型可以被隐式转换为指定的变量的数据类型

欲知更多错误源和解决方案的信息，参考 SIMOTION功能手册。

```
VAR
    usint var    : USINT;
    real var     : REAL;
    byte var     : BYTE;
    string var   : STRING[80]   := 'example for string';
END VAR

usint var   :=  234  /  10;          // Expression data type: USINT
                                     // Result   =  23

real var    :=  234  /  10;          // Expression data type: USINT //
                                     Implicit conversion possible //
                                     Result   =  23.0

usint var   :=  234  / SINT#10;      // Expression data type: INT ↓
                                     // Implicit conversion and ↓
                                     // value assignment not possible

real var    :=  234  /  10.0;        // Expression data type: REAL //
                                     Result   =  23.4

usint var   :=  234  /  10.0;        // Expression data type: REAL ↓
                                     // Implicit conversion and ↓
                                     // value assignment not possible

byte var    := string var[5];        // Implicit conversion possible //
                                     Result   =  16#70   ('p')

string var[10]   := byte var;        //  Implicit conversion possible ↓
                                     // Result   = 'example fpr string'
```

注意：如适用，显式指定数字的数据类型（如： UINT#127, 如果数字 127 i 为数据类型 UINT 而不是 USINT).

1.8.1.2 显式数据类型转换

如果信息可能丢失， 需要显式转换。 例如：如果数值范围减小或精度减小， 就如同从 LREAL 到 REAL的转换一样。

数字数据类型的转换功能在 SIMOTION基本功能功能手册中列出。

当检测到精度缺失的转换时编辑器输出警告。

注意：在程序运行时的类型转换将引起错误，这将引起任务组态中的错误回复。

当转换 DWORD到 REAL时，需要特别注意。 从 DWORD得来的位字符串视为 REAL数值没有检查。必须确保对应标准化的浮点值的位格式在 DWORD 中的位字符串与 IEEE一致。可以使用 _finite 和 _isNaN 功能达到此目标。

否则，一旦 REAL值首次被用作算术运算时，会触发一个错误（例如：在程序中或监视符号浏览窗口）

注意：如果在转换 LREAL到 REAL时超出数值限制范围时，应用以下：

●下溢（ LREAL数字的绝对值小于最小 LREAL正数）
结果为 0.0

●上溢（ LREAL数字的绝对值大于最大 LREAL正数）
触发任务组态中的指定的错误回复

1.8.2 补充的转换

ST系统功能允许下列转换

●组合的比特串类型
这些功能把一个位串行数据类型的多个变量组合成一个高级别数据类型的变量

●拆分的比特串数据类型
这些功能把一个比特串数据类型的一个变量拆分为一个高级别数据类型的多个变量

●在任意数据类型和比特阵列之间的转换
通常用作在不同设备之间为数据交换创建定义转移公式
欲知更多信息（如：比特阵列的安排），见 SIMOTION基本功能功能手册

●技术目标数据类型的转换
把分层的 TO 数据类型 (driveAxis, posAxis, 或 followingAxis)的变量或普通的 ANYOBJEC类型转换为兼容的 TO 数据类型

欲知更多的例子和信息，请见 SIMOTION基本功能功能手册

2. 功能，功能块和程序

此章描述了如何创建和调用用户定义的功能和功能块。标准功能在系统中的类型转换，三角法，和比特字符串处理是可用的。 SIMOTION基本功能功能手册描述了如何使用系统功能和技术目标的功能。

一个功能（ FC）是指无静态数据的编码块。当退出功能时所有的本地变量丢失数值，下次再调用功能时再重新启动。

一个功能块（ FB）是指带静态数据的编码块。 FB 有内存，所以在用户程序中可以随时访问输出参数。本地变量在调用之间保持原值。

程序与 FB 类似，但是没有参数。然而，却可以指定执行级别和任务（见 SIMOTION基础功能手册）

FC和 FB有着可以重新使用的优势，因为他们是被可以指定参数的源文件部分所包含。

功能，功能块和程序为可编程组织单元（ POU），如：它们是可执行的源文件部分。你将在 169 页找到源文件部分的概述。

2.1 创建和调用功能和功能块

下列描述解释了如何创建和调用功能（ FC）和功能块（ FB）。一个完整的例子表明了 FC 和 FB的区别（见 161 页功能和功能块的比较）

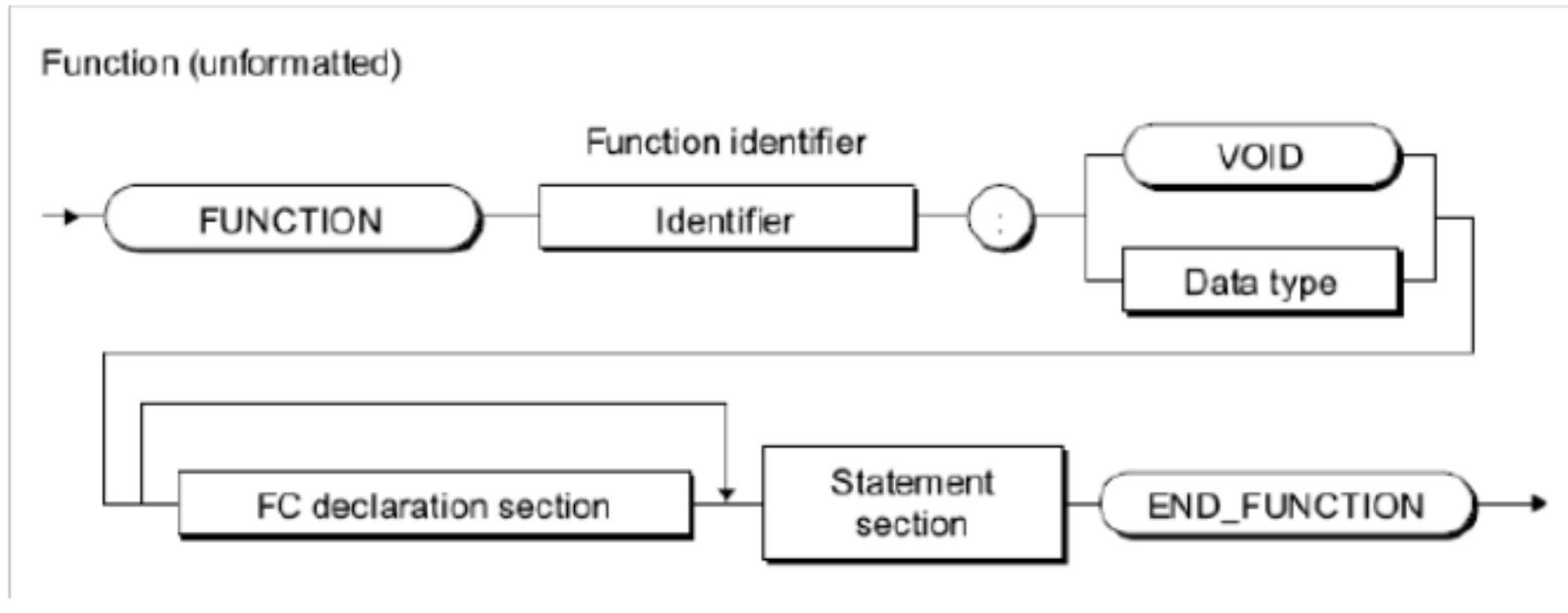
必须用来定义和调用规定的源文件部分的顺序在 169 页源文件部分的使用中给出。

如何导出和导入功能和功能块在 179 页在 ST源文件部分的导入和导出部分中解释。

2.1.1 定义功能

在调用源文件部分（程序， FB,FC）之前，在 implementation 部分的声明部分定义一个功能。

使用下面的语法图：



功能关键词由如同 FC 名称和回送数值的数据类型的标识符跟随。如果 FC 没有回送值，输入 VOID 为数据类型。

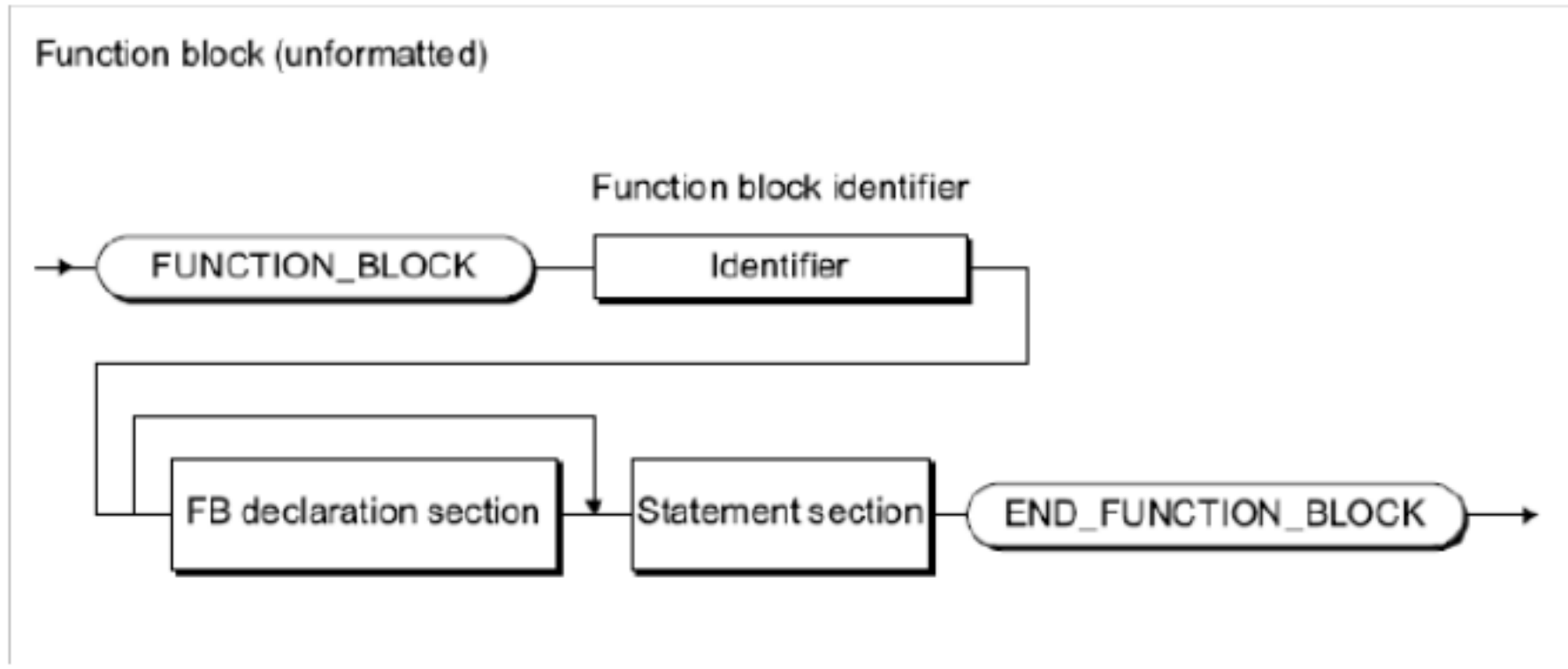
然后输入（见 162 页带注释的源文件例子）

- 可选的声明部分
- 语句部分
- END_FUNCTION关键词

2.1.2　定义功能块

在调用源文件部分（程序，　FB，或 FC）中的 FB之前，可以在　implementation　部分的声明部分定义一个功能块。

使用以下的语法图：



在 FUNCTION_BLOCK关键词之后输入一个标识符作为　FB名称。

然后输入（ 162 页带注释的源文件例子）

- 可选的声明部分
- 语句部分
- END_FUNCTION关键词

2.1.3FC　和 FB 的声明部分

一个声明可以被细分为不同的声明块，每个声明块由单独的一双关键词来标识。每个块包含带类似数据的声明列表，如常量，本地变量和参数。块的每个类型只能出现一次，可以出现在任意顺序中。

下列的选项对　FC和 FB的声明部分都是可用的

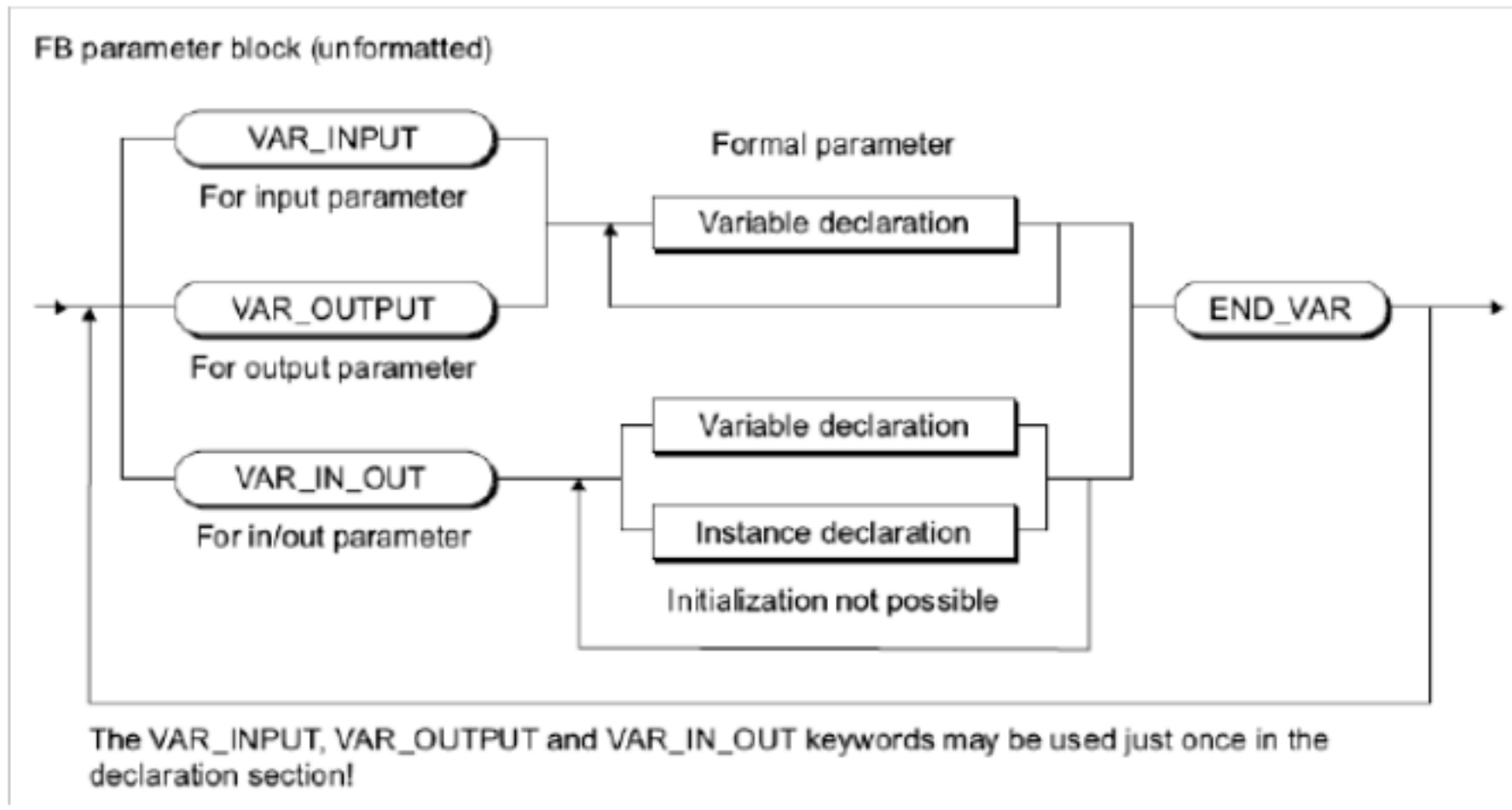
数据	语法	FB	FC
常量	VAR CONSTANT 声明列表 END_VAR	X	X
输入参数	VAR_INPUT 声明列表 END_VAR	X	X
in/out　参数	VAR_IN_OUT 声明列表	X	X

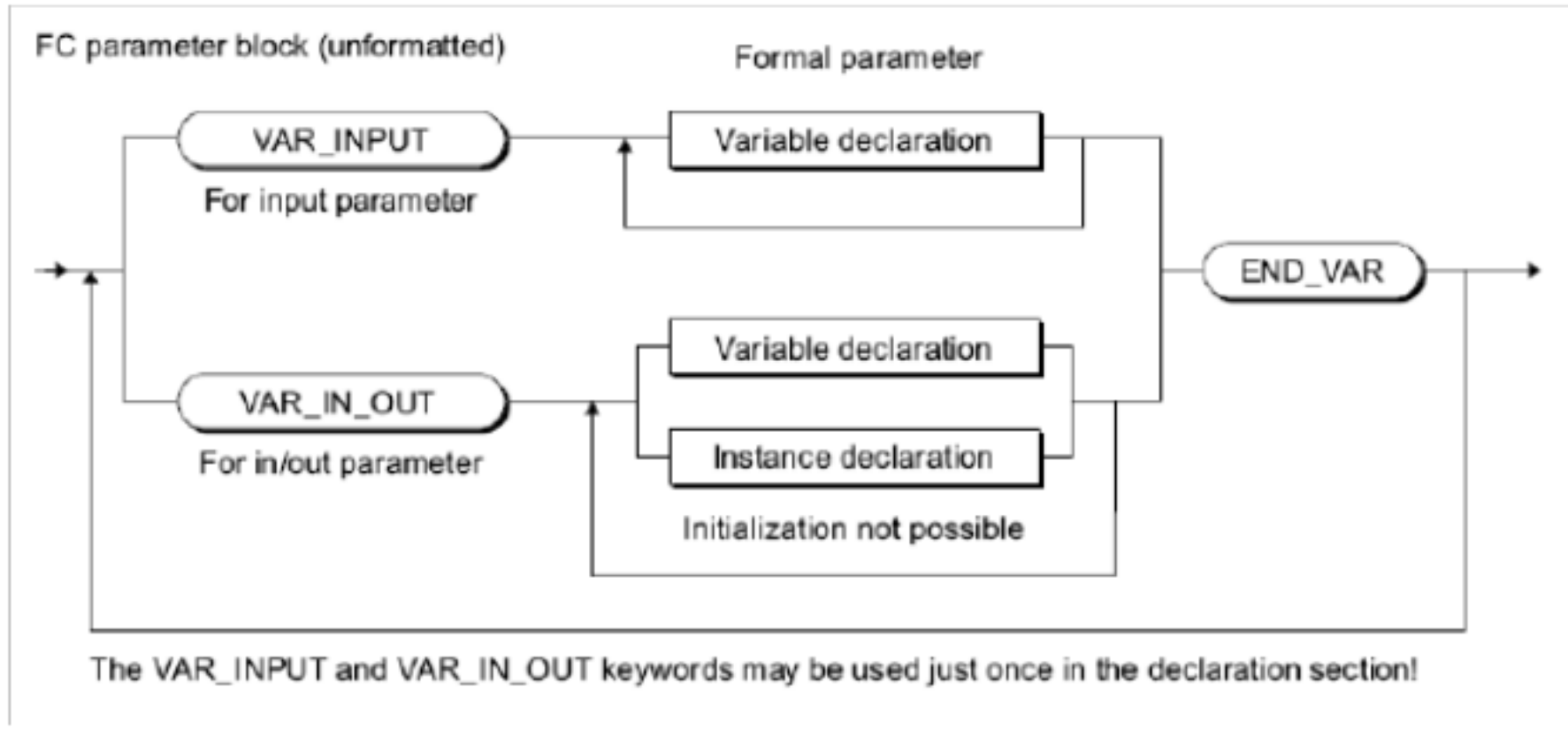
	END_VAR		
输出参数	VAR_OUTPUT 声明列表 END_VAR	X	-
本地变量 (FC和 FB)	VAR 声明列表 END_VAR	X（静态）	X（临时）
本地变量 (FB)	VAR_TEMP 声明列表 END_VAR	X（临时）	
声明列表：声明类型标识符的列表			

参数为本地数据和功能或功能块的形式参数。当调用　　FC或 FB 时，形式参数由实际参数所替代，因此提供了在被调用和调用源文件部分之间的交换信息的方式。

- 形式输入参数接收实际输入参数（向内数据流）
- 形式输出参数（只指　FB）被用作传输输出数值（向外数据流）
- 作为输入和输出参数的形式　 in/out 参数行为

下表为 FB 和 FC的参数说明的语法图：





可以如同其他带 FB 或 FC 的变量一样使用声明的参数，以下例外，不能给输入参数指定数值：

从 FB或 FC的外部，你可以访问：

●通过结构变量（见 94 页用户定义数据类型）访问 FB的输入和输出参数

只有当激活编辑器中“ 允许语言扩展 ”选项时（见 45 页全局编辑器设置和 46 页本地编辑器设置），可以访问输入参数。

输出参数的数据访问功能的标准的。

●通过使用表达式的功能和指定的 FC 的回送值，如：到一个变量（功能名称的说明调用功能，同时回送一个结果）

2.1.4FB 和 FC 部分的语句

FC和 FB的语句部分包含调用 FC或 FB时所执行的语句。 与创建语句部分的正式规则相比，是一样的。但是，你必须注意下表中的信息。

注意：如何有效使用参数，请见 SIMOTION基础功能手册中的优化运行时间的编程。

参数/变量	用途
输入参数	调用 FC或 FB，指定当前值给输入参数。使用这些数值来处理 FC或 FB 的数据。 如：对于计算而言， 但是不能自行修改。 只针对激活编辑器中的“ 允许语言扩展 ”选项（见 45 页全局编辑器设置和 46 页本地编辑器设置）。 FB 的输入参数可以使用结构变量来读和写， FB 之外也如此。（如调用源文件部分）

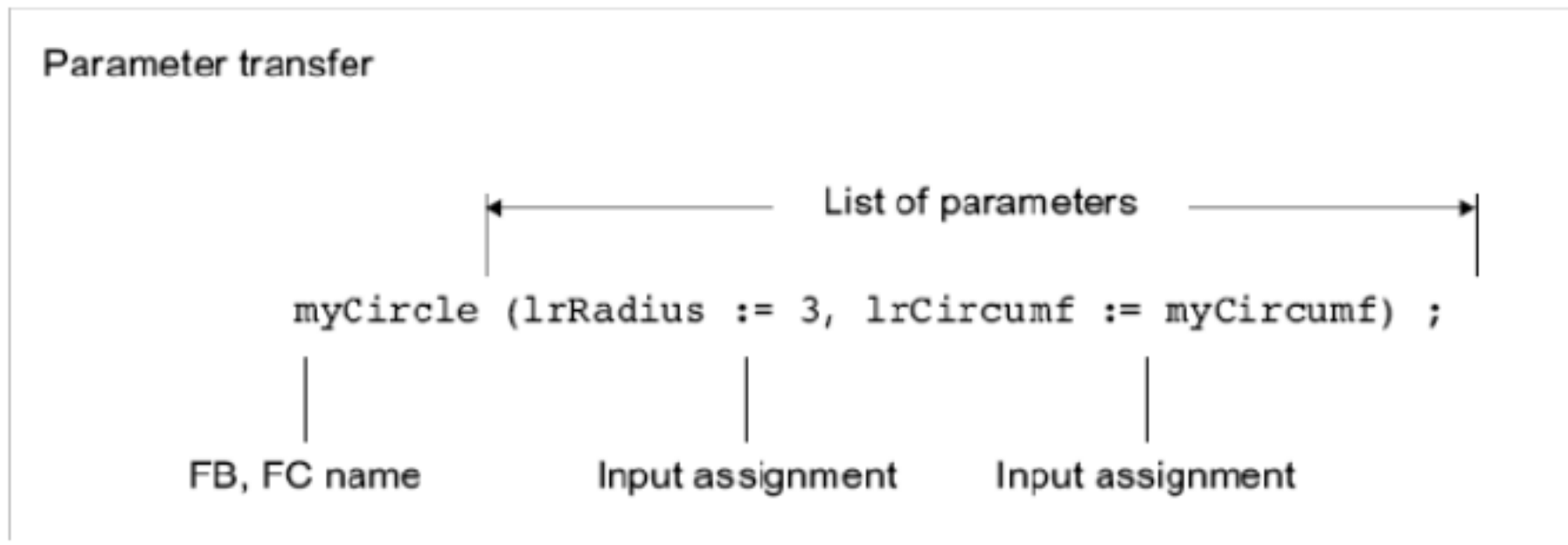
in/out 参数	为了调用 FB或 FC指定一个变量给 in/out 参数。FB或 FC直接访问此变量，也可立即对此进行更改。不支持类型转换。 被指定给 in/out 参数的变量必须可以直接被读或写。因此系统变量（ SIMOTION设备或技术目标）， I/O 变量或过程图像访问不能被指定给一个 in/out 参数
输出参数（只指 FB）	使用 =>运算符调用 FB 时可以指定 一个变量给一个 in/out 参数。当关闭 FB时输出参数（结果）的值被转移给变量。 FB 输出参数可以通过使用结构变量来读， FB 以外的也一样。（如：在调用源文件部分） FC 没有形式输出参数，因为功能名称接收回送数值。某种程度来说功能名称自己就是一个输出参数。
本地变量	本地变量为仅在块中声明和使用的变量。 所有的本地变量 (VAR ... END_VAR) 在 FC中都是临时的。如：当 FC 结束时，它们的数值将丢失。下次调用 FC时，将重新启动。 静态和临时的本地数据的不同在于 FB： 关闭 FB时，静态数据 (VAR ... END_VAR) 保持原值 关闭 FB 时，临时变量 (VAR_TEMP ... END_VAR) 数值丢失。下次调用 FC时，将重新启动。 本地变量的数值不能通过调用块直接查询。 只能使用输出参数查询。

2.1.5 功能和功能块的调用

这将给功能和功能块的调用提供一个概述。

2.1.5.1 参数转移的原则

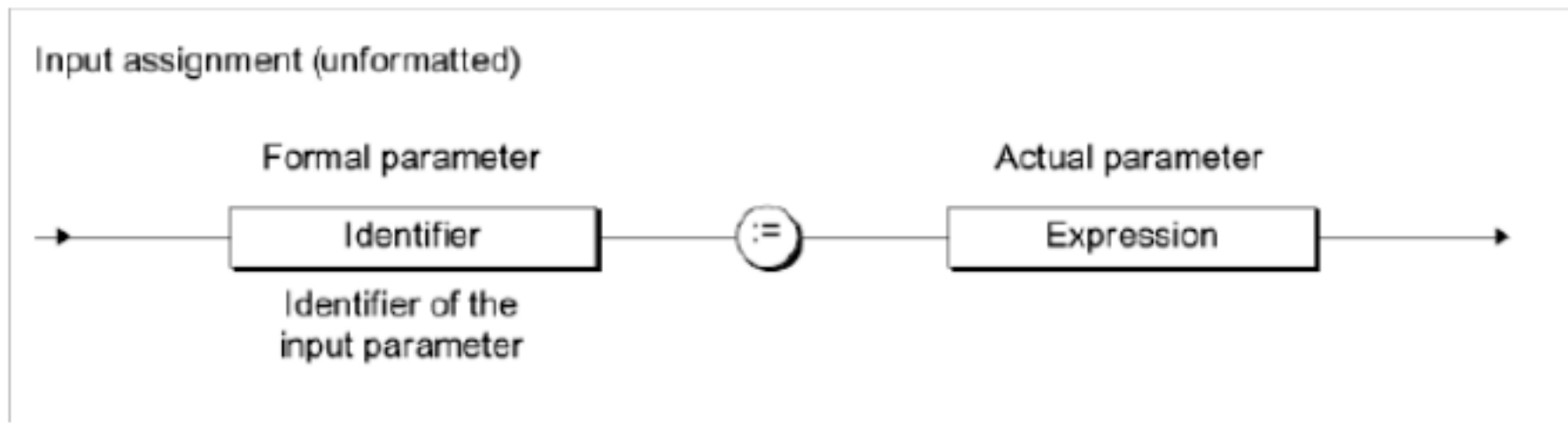
当你调用一个 FC或 FB 时，数据交换在调用和被调用的块之间发生。转移的数据必须指定为调用里的一个参数列表。参数编写在括号中。多个参数由逗号隔开。



输入和 in/out 参数作为一个数值被指定。指定数值（实际参数）給被调用的块（形式参数）声明部分中定义的参数。

使用 =>运算符来指定输出参数。指定数值（实际参数）給 被调用的块（形式参数）声明部分中定义的输出参数。

2.1.5.2 转移给输入参数的参数



通过输入指定，转移数据（实际参数）给 FC 或 FB 的形式输入参数。可以通过表达式指定实际参数。 FB 或 FC 中语句的形式输入参数也可使用，但是你不能修改数值。

支持简易格式的参数转移， 但是不能用于用户定义 FB的兼算。简易格式只使用于某些 FC，见 SIMOTION基础功能手册。

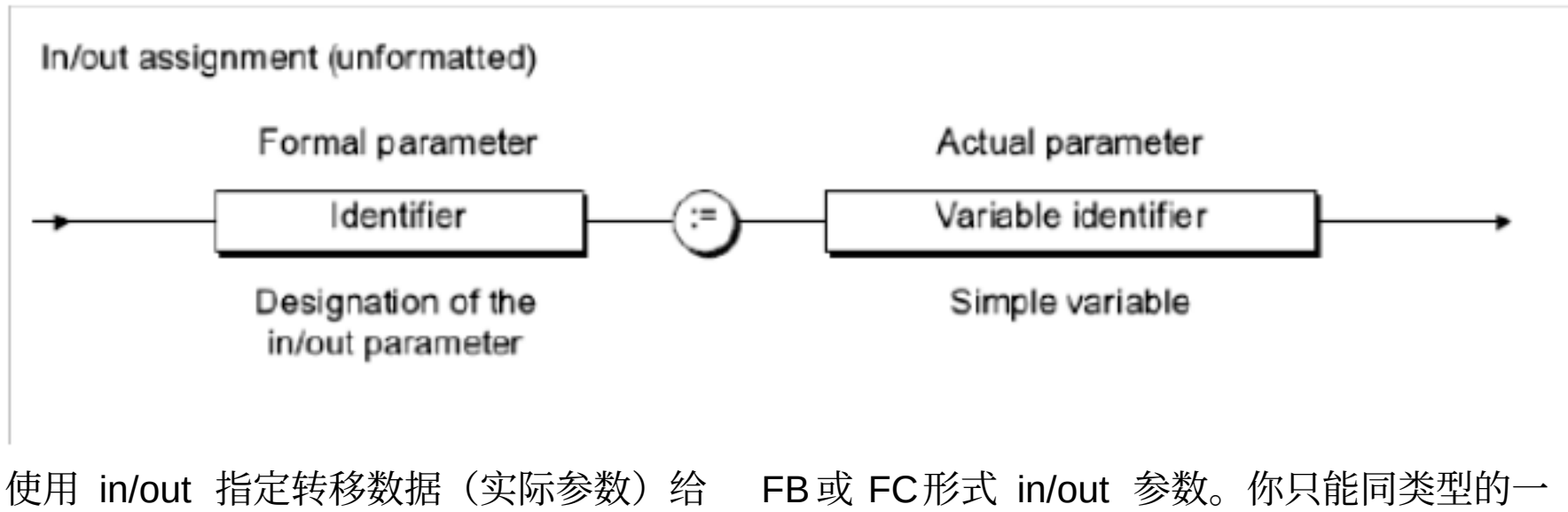
实际参数的指定对 FB 是可选的。如果没有说明输入指定，最后调用的数值将保留，因为 FB 是带内存的源文件部分。

当表达式的初始化被指定为形式参数的说明时，一个实际参数的指定对 FC是可选的。

同时参考 156 页调用功能和 157 页调用功能块。

任何时间在 FB 外也可以读和写一个 FB 输入参数。欲知更多信息，见： 159 页在 FB外访问 FB输入参数

2.1.5.3 参数转移给 in/out 参数



使用 in/out 指定转移数据（实际参数）给 FB 或 FC 形式 in/out 参数。你只能同类型的一个变量给形式 in/out 参数，不可以转换数据类型。

你可以在 FC 后 FB 中使用和改变形式 in/out 参数。FC 或 FB 直接访问实际参数的变量， 也可以立即更改。

见 156 页调用功能和 157 页调用功能块

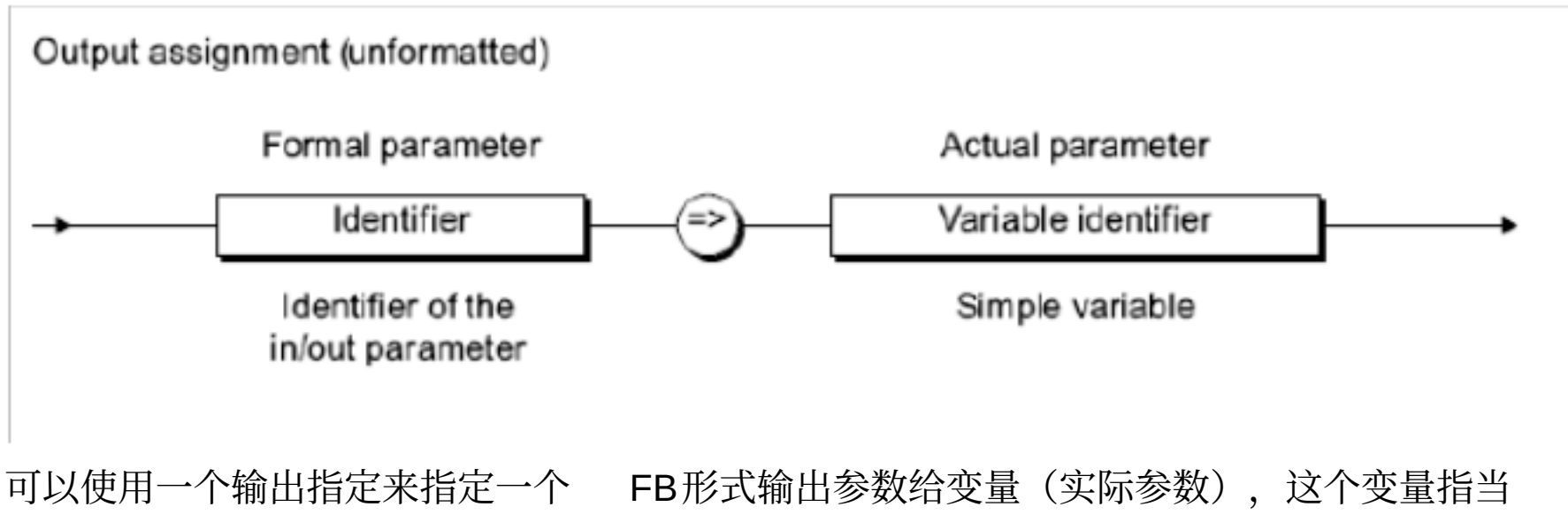
in/out 指定使用字符串数据类型时， 实际参数的声明长度必须大于或等于形式 in/out 参数的长度（见下例）

```
FUNCTION BLOCK REF STRING
    VAR IN OUT
        io  : STRING[80];
    END VAR
;    // Statements
END FUNCTION BLOCK

FUNCTION BLOCK test
    VAR
        my_fb  : REF STRING;
        str1   : STRING[100];
        str2   : STRING[50];
     END VAR
    my_fb(io  := str1);     // Permitted call
    my_fb(io  := str2);     // Not permitted call,
                            // compiler error message

END FUNCTION BLOCK
```

指定给一个 in/out 参数的变量必须能够直接读和写。因此，系统变量（ SIMOTION设备或技术目标）， I/O 变量或过程图像访问不能被指定给一个 in/out 参数。

请注意不同的参数访问时间！

2.1.5.4 参数转移到输出参数（仅对 FB）



可以使用一个输出指定来指定一个 FB形式输出参数给变量（实际参数），这个变量指当关闭 FB时接收形式输出参数的数值

你可以在 FB 的语句中使用和更改形式输出参数。

见 157 页调用功能块的例子

输出指定对于参数转移是可选的。 你在任何时间可以读和写一个 FB输出参数， 甚至在 FB外也行。欲知更多，见 159 页访问 FB外的输出参数。

2.1.5.5 参数访问时间

访问的类型和参数访问时间是不同的：

- 就输入指定而言，实际参数的值复制给形式参数。如果大结构（如阵列）被复制，而且 FC或 FB被频繁调用，这将限制性能。
- 值没有复制到 in/out 指定。反而在形式参数的存储地址和实际参数之间创建了一个链接。因此转移变量要快于输入指定（特别是涉及到大量数据时）。但是，从 FB的访问变量较之可能会慢些。
- 如果你正在使用单元变量， 没有内容会复制给 FB或 FC。因为这些变量在全部的 ST源文件中是有效的。（见 184 页变量模型）

注意：
如果大量数据要传到功能块，使用 in/out 参数代替输入参数只会更快。

如果主要使用单元变量，而不是参数，产生的程序结构将很复杂和混乱：目标导向，数据封装，变量名称的多重使用（有效性范围的封装）等，这些都不再可能。

2.1.5.6 调用一个功能

如下一个功能被调用：

- 带回送值的功能（除了 VOID 的数据类型）

功能位于一个数值指定的右侧。也可在表达式中作为运算对象出现。在调用功能之后，在合适的点使用回送值来计算表达式。

```
y:=sin(x);
y  := sin(in   := x);
y  := sqrt   (1   - cos(x)   * cos(x));

```

●无回送值（ VOID 数据类型）的功能

指定仅包含功能调用。

下面的例子是有效的， 假如已经定义含有 in1 和 in2 输入参数的 funct1 功能和 inout in/out 参数。

```
funct1      (in1      := var11, in2          := var12, inout1          := var13);
```

注意：在功能中，结果（回送值）被指定为功能名称（除了 VOID 数据类型）

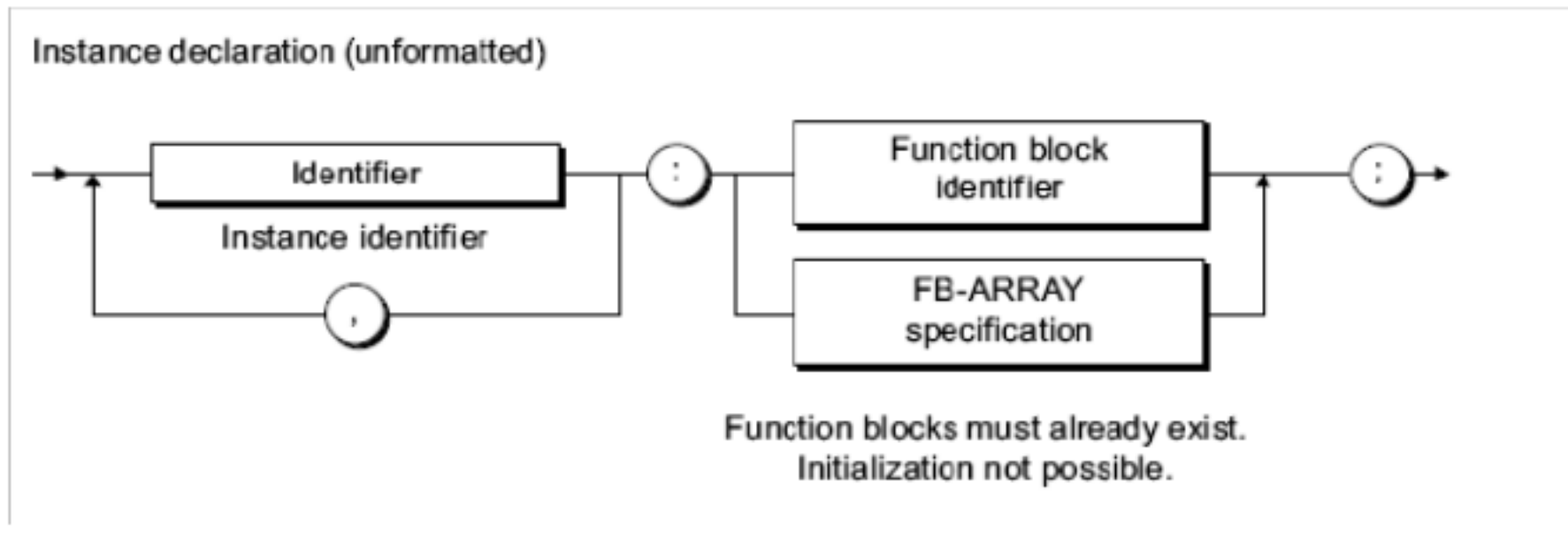
2.1.5.7 调用功能块（实例调用）

在调用一个功能块（ FB）之前，你必须声明一个实例。你声明一个变量，输入功能块的名称作为数据类型。声明这个实例：

●本地（在程序或功能块的声明部分 VAR/END_VAR）

●全局（在 implementation 部分的 interface 中的 VAR_GLOBAL/END_VAR）

●作为一个 in/out 参数（在功能或功能块的声明部分中的 VAR IN OUT / END VAR）



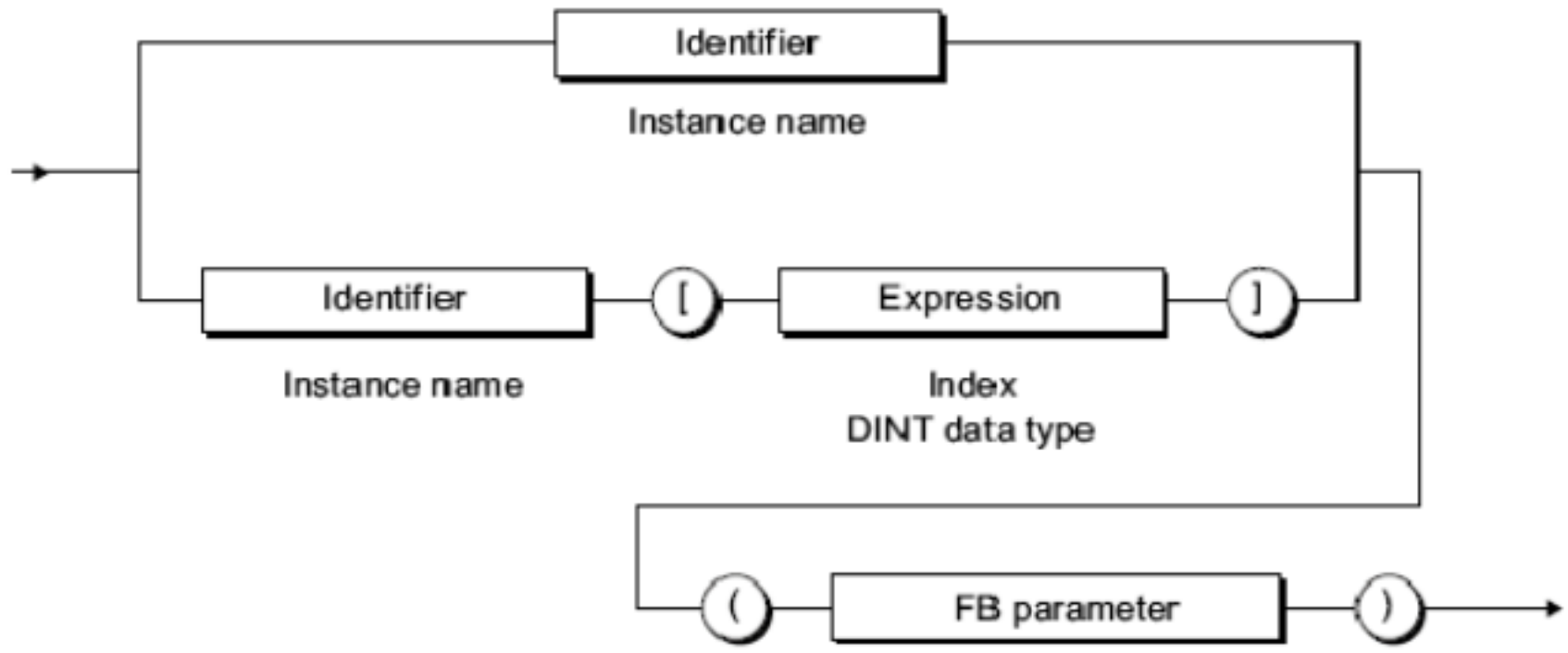
实例声明也可以是一个阵列，如：

```
FB_inst      : ARRAY     [1..2] OF FB_name.
```

注意：不同的变量类型的不同初始化

在 POU的语句部分调用一个功能块。 FB参数是通过逗号隔开的输入和 in/out 指定。

FB call (unformatted)



在下表中的例子是适用的，假设已经定义了 supply 和 motor 的功能块

●FB Supply

输入参数 in1, in2; in/out 参数 inout; 输出参数 out

●FB motor

in/out 参数 inout1, inout2; 输出参数 out1, out2

```
VAR ↵
    Supply1, Supply2: Supply; ↓
    Motor1  : Motor; ↵
END VAR ↵
↵
Supply1  (in1  := var11, in2  := expr12, inout  := var13, out  => var14)  ; ↵
Supply2  (in1  := var21, in2  := expr22, inout  := var23, out  => var24)  ; ↵
Motor1 (inout1 := var31, inout2 := var32, out1 => var33, out2 => var34); // ... ↵
var15  := PowerSupply1.out; ↓
var25  := PowerSupply2.out; ↓
var35  := Motor1.out1; ↓
var36  := Motor1.out2; ↵
var41  := Motor1.out1  * Motor1.out2  *  (Supply1.out  + Supply2.out); ↵
                                                                        ↵
```

2.1.5.8 在 FB 外访问 FB 输出参数

除了输出指定来调用一个 FB之外，在 FB 外访问一个 FB输出参数也是可能的。

如要这样做，使用 FB 实例名称的结构变量（ 100 页），， 输出参数格式，如 Supply1.out

FB的实例名称不能在数值指定中使用。

2.1.5.9 在 FB 外访问 FB 输入参数

除了输入指定调用 FB 之外，在 FB 外读写一个 FB 输入参数也是可能的。

如要这样做，使用 FB 实例名称的结构变量（ 100 页），，输入参数格式，如 Supply1.in1

注意：要使用此功能，必须激活编辑器中的允许语言扩展选项。

FB的实例名称不能在数值指定中使用。

Table 4-5 Example of assignment to input parameter

```
// Only with compiler option "Permit language extensions" activated
VAR
    var_fb   : _WORD_TO_2BYTE;
    var_word : WORD;
END_VAR
var_fb.wordin := var_word;
// ..
var_fb();
```

2.1.5.10FB 调用时的错误源

当调用一个功能块实例时注意：

●只指定带直接存储在内存中的变量的 in/out 参数

只有下列的变量为允许的实际参数

—全局参数（单元变量和全局设备用户变量）

—本地参数

—TO 数据类型（ TO 实例）的变量

特别是下列内容，是被允许的

系统变量（ TO 变量）

—从设计系统来的技术目标的名称

—I/O 变量

—绝对和符号过程图像访问

●不使用功能作为 in/out 参数

FC 回送值，如 FC 调用，在 in/out 指定中不能为实际参数。你必须先存储在本地变量中FC的结果，然后使用此变量作为 in/out 指定中的实际参数。

●不使用变量作为 in/out 参数

只能使用变量作为 in/out 指定的实际参数，因为会回写数值。

●不能初始化 in/out 参数

2.2 功能和功能块的比较

用户定义的功能和功能块的区别在下面用详细的例子来说明。

2.2.1 例子说明

下面的例子说明了 FB 和 FC之间的区别。 为了简单起见， ，每个类型的参数只使用一次，但是在现实的实际情况中，你可以定义多个参数。使用的术语在详细的描述 148 页定义功能和 149 页定义功能块中被定义。

一个块将在 implementation 部分的声明中作为一个 FB和 FC被创建，用作计算圆周和半径输入变量的圆圈面积。

●为半径定义输入参数

●为圆周定义 in/out 参数，如：在调用 FC或 FB 时，直接指定转移变量的数值

●有多种方法定义 FB和 FC的圆圈面积

—对 FB而言，定义输出参数

—对 FC而言，使用其回送值。合适定义回送值的数据类型

●每个 FB和 FC会被记录在计数器中（本地变量）。对例子的解释为：我们认为这个数值将持续仅在 FB中被计数。

●在编程部分， FB或 FC被调用，实际参数被指定给下列形式参数：

—对 FB而言，输入， in/out 和输出参数

—对 FC而言，输入和 in/out 参数

在调用 FB 或 FC之后，对圆周和面积的数值都是可用的。

—对 FB而言，在 in/out 和输出参数的实际参数里。

输出参数甚至可以在 FB外被读出。

—对 FC而言，在功能的回送值和 in/out 参数的实际参数值里。

2.2.2 带注释的源文件

Table 4-6 Example of differences between FB and FC

Function block (FB)

```
INTERFACE
  PROGRAM CircleCalc1;
END_INTERFACE
IMPLEMENTATION
  FUNCTION_BLOCK Circle1
    //Constant declaration
    VAR CONSTANT
      PI : LREAL := 3.1415 ;
    END_VAR
    //Input parameter
    VAR_INPUT
      Radius : LREAL;
    END_VAR
    //In/out parameter
    VAR_IN_OUT
      circumference : LREAL;
    END_VAR
    //Output parameter
    VAR_OUTPUT
      Area : LREAL;
    END_VAR
    // Local variables, static
    VAR
      Counter : DINT;
      (* Variable retains its value
      between calls *)
    END_VAR
    //Call counter
    Counter := counter + 1 ;
    Circumference := 2 * PI * Radius ;
    Area := PI * Radius**2 ;
  END_FUNCTION_BLOCK
  PROGRAM CircleCalc1
    VAR
      myCircle1         : Circle1 ;
      myArea1, myArea2 : LREAL;
      myCircf           : LREAL;
    END_VAR;
    myCircle1(Radius := 3
      , Circumference := myCircf
      , Area => myArea1) ;
    myArea2 := myCircle1.Area ;
    // myCircf has the value 18,849
    // myArea1 has the value 28,274
    // myArea2 has the value 28,274
  END_PROGRAM
END_IMPLEMENTATION
```

Function (FC)

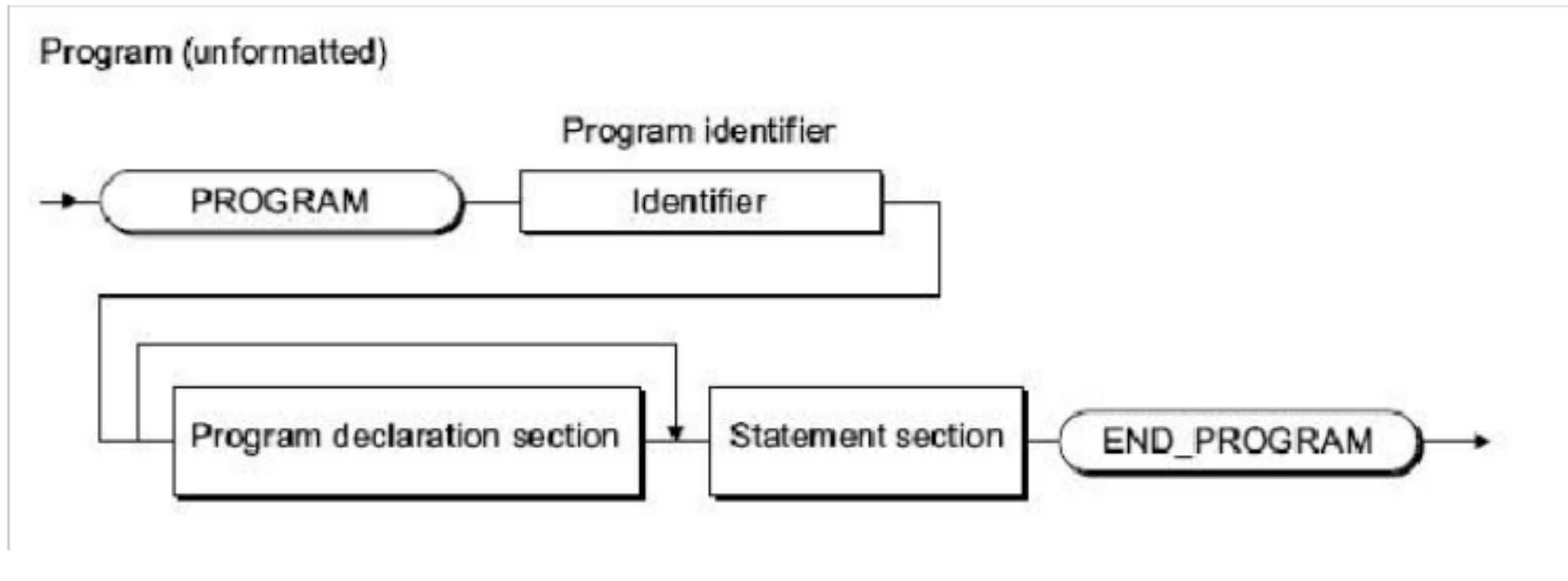
```
INTERFACE
  PROGRAM CircleCalc2;
END_INTERFACE
IMPLEMENTATION
  FUNCTION Circle2 : LREAL
    //Constant declaration
    VAR CONSTANT
      PI : LREAL := 3.1415 ;
    END_VAR
    //Input parameter
    VAR_INPUT
      Radius : LREAL;
    END_VAR
    //In/out parameter
    VAR_IN_OUT
      circumference : LREAL;
    END_VAR
    //Output parameter
    // Not possible


    // Local variables, temporary
    VAR
      Counter : DINT;
      (* Variable will be initialized
      with 0 for each call *)
    END_VAR
    //Call counter
    Counter := Counter + 1 ;
    Circumference := 2 * PI * Radius ;
    Circle2 := PI * Radius**2 ;
  END_FUNCTION
  PROGRAM CircleCalc2
    VAR

      myArea  : LREAL;
      myCircf          : LREAL;
    END_VAR;
    myArea := Circle2(Radius := 3
      , Circumference := myCircf);


    // myCircf has the value 18,849
    // myArea has the value 28,274

  END_PROGRAM
END_IMPLEMENTATION
```

功能块 FB	功能 FC
注释	
定义的预留词： FUNCTION_BLOCK and END_FUNCTION_BLOCK	定义的预留词： FUNCTION and END_FUNCTION

不允许回送值	回送值的数据类型必须在名称（ VOID 数据类型，如果没有回送值）后说明
可以使用输入参数来转移数值到 FB	可以使用输入参数来转移数值到 FC
可以使用 In/out 参数来读写 FB中的转移变量	可以使用 In/out 参数来读写 FC中的转移变量
可以使用输出参数从 FB回送数值	不允许输出参数
本地变量是静态的，如：在 FB 调用中保留数值 计数器本地变量增加；当 FB 结束时保留其数值。因此每次调用 FB时变量增加。 为了看见这种行为：指定本地变量数值到 FB 的全局变量。在重复 FB 调用后监视全局变量数值	本地变量是临时的， 如：当功能结束后数值丢失。 虽然计数器本地变量增加，当 FC 退出时数值丢失。下次调用 FC 时变量重新恢复（例子中变为 0） 为了看见这种行为：指定本地变量数值到 FC 的全局变量。在重复 FC调用后保持全局变量不变。
在语句部分，结果（回送值）被指定为输出或 in/out 参数	在语句部分，结果（回送值）被指定为功能名称（除了特别指定 VOID数据类型之外）
在执行调用的块的声明部分，声明 FB 的一个实例：声明一个变量，指定 FB 名称为它的数据类型。使用声明的实例名称来调用 FB，以访问输出参数。 FB的名称 不能使用在语句部分。	
●在 FB 实例被调用时，指定一个变量到 in/out 参数 ●调用时，可以指定输出参数给一个变量 ●可以读出一个 FB 的输出参数，甚至在 FB 外。 为此，你必须使用下列格式的结构变量： FB-instancename.outputparameter	●在 FB 实例被调用时，指定一个变量到 in/out 参数 ●为获得 FC的回送值 —指定一个功能到一个变量 —使用表达式右侧数值指定的的功能
执行调用的程序不能访问变量，除了 in/out 变量和 FB 的输出参数。 例外：激活编辑器的允许语言扩展功能。， 调用的程序可以访问 FB的输入参数。为此，使用下列格式的结构变量： FB-instancename.inputparameter.	执行调用的程序不能访问变量， 除了回送值

2.3 程序

程序是在 PROGRAM和 END_PROGRAM之间放置的一系列语句。



程序在 ST 源文件的 implementation 部分被声明，并与 FB 比较。静态的本地变量 (VAR...END_VAR)或临时变量 (VAR_TEMP...END_VAR)可以被创建。 例如：它们没有任何的形式参数，所以不能不能通过参数调用。程序的例子见 162 页带注释的源文件和 64 页程序示范。

在执行系统中指定一个程序

默认执行系统中的程序被指定给一个任务。程序的执行动作，例如：相关的任务决定变量的初始化。欲知更多，见 SIMOTION 基础功能手册。这需要程序在源文件的 interface 部分必须指定为导出的程序组织单元。

在程序中调用一个程序（“ 程序中的程序 ”）

在不同的程序或功能块中可以调用一个程序。这需要激活编辑器的以下功能。（见 45 页编辑器的全局设置和 46 页编辑器的本地设置）：

1. 调用程序或功能块程序源的 “ 允许语言扩展 ” 功能

2.调用程序的程序源的 “ 仅一次创建程序实例数据 “ 功能。调用作为带参数和回送值的功能被执行，见下面的例子。

注意：编译器的 “ 仅一次创建程序实例数据 “ 功能将引起：

- 程序（程序实例数据）的静态变量将存储在不同的区域（ 194 页）。这将改变初始化的动作（ 204 页）。
- 带相同名称的所有被调用的程序使用相同的程序实例数据。

Table 4-8 Example for calling a program in a program

```
PROGRAM my_prog
    ; // ...
END_PROGRAM

PROGRAM main_prog
    ; // ...
    my_prog();
    ; // ...
END_PROGRAM
```

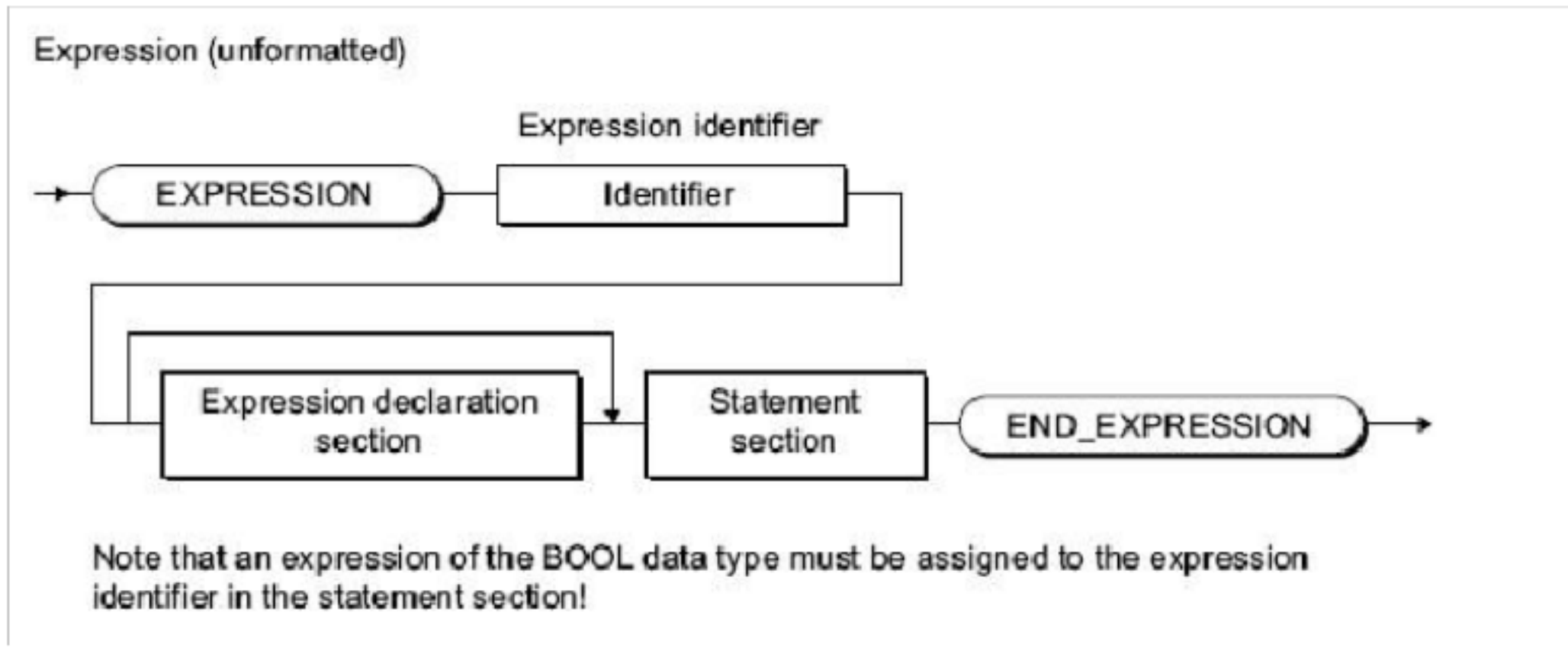
这可以被用作执行指定程序到任务的大部分的程序。在执行系统中，只有每个相关的调用程序需要被指定给任务。

2.4 表达式

表达式是功能说明的特例：

回送值的数据类型被定义为 BOOL，并且不是明确指出。 连同 WAITFORCONDITION 语句一起使用。

一个表达式只能在源文件的 implementation 部分被声明。



在声明部分可以声明下列内容：

- 本地（临时）变量
- 本地常量
- 用户定义数据类型（ UDT）
- 输入和 in/out 参数

在语句部分可以访问下列内容

- 本地变量的表达式
- 输入和 in/out 参数（假如声明是被允许的）
- 单元变量
- 全局设备变量， I/O 变量和过程图像

数据类型 BOOL的表达式必须被指定给在表达式语句部分的表达式名称。

注意：表达式的语句部分不能包含任何的功能调用或循环。

下面的例子假设支程序在 MOTION TASK里面运行。 在 Startup Task之后选择功能激活。 在 SIMOTION SCOUT里面运行从程序到任务的指定。

```
INTERFACE
    USEPACKAGE cam;
    PROGRAM feeder; // in MotionTask_1
END_INTERFACE

IMPLEMENTATION
    // Condition for WAITFORCONDITION statement
    EXPRESSION automaticExpr
        automaticExpr := IOfeedCam; // Digital input
    END_EXPRESSION

    PROGRAM feeder
        VAR
            retVal : DINT ;
        END_VAR ;
        retVal := _enableAxis (axis := realAxis,
            enableMode := ALL,
            servoCommandToActualMode := INACTIVE,
            nextCommand := WHEN_COMMAND_DONE,
            commandId := _getCommandId() );

        // Wait until the start condition is satisfied
        WAITFORCONDITION automaticExpr WITH TRUE DO
            // High-priority execution of all statements
            // to the END_WAITFORCONDITION command
            retVal := _pos (axis := realAxis,
                positioningMode := RELATIVE,
                position := 500,
                velocityType := DIRECT,
                velocity := 300,
                velocityProfile := TRAPEZOIDAL,
                mergeMode := IMMEDIATELY,
                nextCommand := WHEN_MOTION_DONE,
                commandId:= _getCommandId() );
    END_WAITFORCONDITION;

    retVal := _disableAxis (axis := realAxis,
        disableMode := ALL,
        servoCommandToActualMode := INACTIVE,
        nextCommand := WHEN_COMMAND_DONE,
        commandId := _getCommandId() );
    END_PROGRAM
END_IMPLEMENTATION
```

更多的例子包含在 SIMOTION运动控制基础功能手册中。 特别是手册描述了使用带参数的表达式在 WAITFORCONDITION语句里编写一个时间监控的程序。

3. 在 SIMOTION 中 ST 的集成

本章描述了 ST程序和 SIMOTION SCOUNT的互操作性。

3.1 源文件部分的使用

源文件部分意义的概述在 86 页中的 ST源文件结构中有描述。 此章描述了细节， 如 section 的语法，和如何使用在许多的 ST源文件中以导入和导出数据。

3.1.1 源文件部分的使用

在源文件部分（模块）中，你必须遵循一些结构和语法规则，以便可以编译 ST 源文件。下面列出了一些基本的指南，详细内容在本章节后半部分。

●当创建源文件时，你应该注意源文件部分的顺序。即将调用的部分必须先于正在调用的部分。否则即将调用的部分不能识别正在调用的部分。

例如：变量必须在使用之前声明，功能在调用之前必须被定义。

●对于最普通的源文件部分的源文本—程序，功能或功能块—包含以下：
—从带预留词语和标识符的部分开始
声明部分（可选的）
—语句部分
—以预留词语结束的部分
●源文件部分的标识符—因此是指名称或遵循一般标识符语法规则的名称列表

注意：在线帮助中有可用的源文件部分的模板。

3.1.1.1interface 部分

interface 部分包含导入和导出数据（数据类型，变量，功能块，功能和程序）的语句。页可以下载技术包和库。

interface 有着如下的语法：

语法	INTERFACE // Interface statements (optional) END_INTERFACE 不能指定 section 的一个单独的标识符。

	但是， interface 语句存在于预留词语 INTERFACE and END_INTERFACE 的之间的顺序： 1.使用的技术包的说明，语法： USEPACKAGE tp-name[AS namespace] 欲知更多详情，见 SIMOTION基础功能手册 2.使用的库的说明，语法： USELIB library-name-list [AS namespace] 欲知更多详情， 见 230 页从库中使用数据类型，功能和功能块。 3.参考其他单元以便使用其导出部分，语法： USES unit_name-list; 欲知更多详情， 见 181 页在导入单元使用的语句 4.对导出的声明和说明 —数据类型定义（ 176 页） 用户定义的数据类型在所有的 ST 源文件是有效的，需要被导出 —变量说明（ 177 页） 单元变量和单元常量在所有的 ST 源文件中是有效的和导出。 允许的关键词见 177 页变量声明 —关于要导出的程序组织单元的信息 语法： FUNCTION fc_name; FUNCTION_BLOCK fb_name; PROGRAM program_name; 列在 interface 部分的所有的技术包， 库，导入单元， 数据类型声明， 变量说明和程序组织单元都将被导出。欲知更多的导出信息，见 179 页 interface 部分导出单元。
顺序	interface 部分是 ST 源文件部分的第一个部分。 interface 语句的顺序 1 到 4 都是固定的 在 1 到 4 中，任意的顺序都是允许的。 单数据类型定义和变量定义的单独的声明块可以出现不只一次。
频次	源文件 / 每次
强制部分	YES
单元语句可以选择先于 interface 部分（见 179 页单元的标识符）	

3.1.1.2implementation　部分

implementation 部分包含执行部分，是　ST源文件的主要组成部分。

implementation 部分语法如下：

语法	IMPLEMENTATION // Implementation statements (optional) END_IMPLEMENTATION 不能指定部分的一个单独的标识符
	implementation 部分（ST源文件的主要部分）可以选择存在于预留词　IMPLEMENTATION 和 END_IMPLEMENTATION之间的顺序中 1.参考其他单元以便使用其导出部分 USES unit_name-list; 欲知更多详情，见 181 页在导入单元使用的语句 2.声明 —数据类型定义（ 176 页） 用户定义的数据类型在所有的　ST 源文件是有效的 —变量说明（ 177 页） 单元变量和单元常量在所有的　ST 源文件中是有效的和导出。 允许的关键词见　177 页变量声明 3.程序组织单元
顺序	总是遵循　interface 部分 上述 implementation 部分的顺序是强制的。在顺序 2 到 3 之间，任意顺序都是允许的
频次	源文件 / 每次
强制部分	YES

3.1.1.3 程序组织单元（ POU）

POU是可以被执行的源文件部分

功能（ FC）（ 172 页）

●表达式（ 174 页）

●功能块（ 173 页）

程序（ 174 页）

注意：已调用的　POU先于正在调用的　POU，所以它们通过后者被辨识。

3.1.1.4 功能（FC）

功能（FC）被列为程序组织单元（ POU）。功能是从程序和功能块调用的临时数据的参数化的源文件部分。当退出功能，下次调用此功能时并且重新恢复时，所有的内部变量失去数值。

FC语法如下：

语法	FUNCTION name : function_data_type
	// Declaration section
	// Statement section
	END_FUNCTION
	名称代表功能的标识符， 而功能数据类型代表回送值的数据类型
	在声明部分的变量说明的允许词语中，见 177 页变量声明
	注意带功能数据类型 <> VOID 的下列功能：
	在语句部分， 功能数据类型的表达式必须被指定给功能标识符。

顺序	FC只能在 implementation 部分被定义。 注意顺序： FC必须在被调用的 POU 之前 声明部分（ 175 页）必须先于 176 页的语句部分。
频次	每个 ST源文件任意次
强制部分	NO

欲知 FC更多信息，见 147 页创建和调用功能及功能块。

3.1.1.5 功能块（ FB）

功能块被列为 POU。它们是带可以从程序和被指定的参数（在调用间内部变量保持数值不变）调用静态数据的源文件部分。。因为 FB有内存，可以任何时间或从用户程序的任意点访问其输出参数。

FB 语法如下：

语法	FUNCTION_BLOCK name
	// Declaration section
	// Statement section
	END_FUNCTION_BLOCK
	名称代表功能块的标识符
	在声明部分的变量说明的允许词语中，见 177 页变量声明

特殊特征	在你调用一个功能块之前， 你必须声明一个实例：你声明一个变量， 然后输入功能块的标识符作为数据类型。 你可以本地声明实例（在一个程序或功能块的声明部分 VAR / END_VAR中） 你可以全局声明实例（在 interface 或 implementation 部分的 VAR_GLOBAL / END_VAR中）。但是，不能使用在同样的ST源文件中定义的功能块。 只有在功能块通过导入程序源文件和库可用的情况下才可行。
顺序	FB只能在 implementation 部分被定义。 注意顺序： FB 必须先于一个实例被作为本地变量声明的 POE
频次	每个 ST源文件任意次
强制部分	NO

欲知 FB更多信息，见 147 页创建和调用功能及功能块。

3.1.1.6 程序

程序被列为 POU。在目标系统中根据任务指定 （见 SIMOTION基础功能手册中的配置执行系统）被调用，和调用功能及功能块。

程序的语法如下：

语法	PROGRAM name
	// Declaration section
	// Statement section
	END_PROGRAM
	名称代表程序的标识符
	在声明部分的变量说明的允许词语中，见 177 页变量声明

顺序	程序只能在 implementation 部分被定义。 把程序放在表达式， 功能和功能块之后是由优势的。这使得程序辨别和使用源文件部分。 声明部分（175 页）必须先于语句部分 （176 页）
频次	每个 ST源文件任意次
强制部分	NO

欲知程序更多信息，见 164 页程序。

3.1.1.7 表达式

表达式是带说明的 BOOL 数据类型回送值的功能说明的特列。计算在 EXPRESSION（表达式标识符）到指定给功能名称的 END_EXPRESSION 预留值的表达式。

你可以使用 WAITFORCONDITION 结构来直接等待 MOTION TASK 里面可编程的事件或条件。语句暂停调用的任务直到条件（表达式）为 TRUE

语法	EXPRESSION name
	// Declaration section
	// Statement section
	END_EXPRESSION
	名称代表表达式的标识符
	在声明部分的变量说明的允许词语中，见 177 页变量声明。
	注意：在语句部分， BOOL 数据类型的表达式必须指定给表达式标识符

顺序	表达式只能在 ST 源文件 implementation 部分被定义。 因此，表达式先于被调用的 WAITFORCONDITION 结构的程序。 声明部分（175 页）必须先于语句部分 （176 页）。
频次	每个 ST源文件任意次
强制部分	NO

欲知表达式更多信息，见 166 页表达式。对于 WAITFORCONDITION 语句，见 SIMOTION 基础功能手册。

3.1.1.8 声明部分

程序组织单元的声明部分包含 POU的数据类型定义和变量说明。

声明部分结构如下：

结构	// Data type definition // Variable declaration
顺序	声明部分在首末没有明显的关键词。 在各自的 POU 的关键词之后开始声明部分，在语句部分的第一个执行语句之后结束。 包含以下顺序：

	数据类型定义（ 176 页） ●用户定义数据类型，在 POU 中本地有效 ●变量声明（ 177 页） 变量和常量在 POU 中本地有效 根据各自的 POU 得出的允许关键词： 见 177 页变量声明列表 注意：标识符必须在使用前声明
频次	每个 POU一次
强制部分	NO

3.1.1.9 语句部分

POU的语句部分包含独立的（可执行的）语句。

语句部分结构如下：

结构	// Statements
顺序	语句部分在首末没有明显的关键词。 在声明部分之后开始，在各自的 POU 关键词之后结束。
频次	每个 POU一次
强制部分	NO

欲知更多语句信息，见

●数值指定和表达式（ 112 页）
●控制语句（ 130 页）
●调用功能和功能块（ 153 页）

3.1.1.10 数据类型定义

对于定义数据类型， 你指定用户定义数据类型 （UDT）。你可以使用它作为变量说明。 UDT 可以在 interface 部分， implementation 部分和 FC,FB及程序部分被说明。

数据类型定义语法如下：

语法	TYPE name : data_type_specification; // ... END_TYPE 名称代表你使用的变量说明的单独的数据类型的名称。 数据类型说明代表任意的数据类型或结构。在单个的数据类型 TYPE 和 END_TYPE之间可以出现任意数字。
顺序	可以按照以下内容定义 UDT：

	●在 interface 部分 UDT 在 ST源文件中被识别然后导出 可以在单元变量的声明中的 interface 和 implementation 部分使用，或者是在所有 POU的本地变量声明中使用。 除此之外，还可以使用在所有单元， 导入 ST 源文件 ●在 implementation 部分 UDT 在 ST 源文件中被识别 可以使用于单元变量的声明部分，和所有 POU 的本地变量声明中。 ●在一个 POU（FC,FB，程序，表达式）的声明部分 UST 在 POU 被本地识别 可以使用于 POU 的本地变量声明中 UDT 必须在使用于变量声明之前被使用
频次	TYPE/END_VAR声明块在源文件部分出现不止一次：在一个声明块中 UDT 的数量是可能的
强制部分	NO

欲知更多 UDT 信息，见用户定义数据类型（ 94 页）

3.1.1.11 变量声明

声明部分包含变量声明，也可以被包含在 FC,FB和程序（ POU）之中， interface 部分和 implementation 部分同样如此。

变量声明语法如下：

语法	variable_type name_list : data_type; // ... END_VAR variable_type 代表将要声明的变量类型的关键词。 允许的关键词基于源文件部分。

	• In the Interface section or Implementation section of an ST source file: VAR_GLOBAL: Non-retentive unit variable VAR GLOBAL CONSTANT: Unit constant VAR GLOBAL RETAIN: Retentive unit variable • In the Declaration section of a function: VAR: Local variable VAR CONSTANT: Local constant VAR_INPUT: Input parameter VAR_IN_OUT: In/out parameter • In the Declaration section of a function block: VAR: Local variable VAR CONSTANT: Local constant VAR_TEMP: Temporary variable VAR_INPUT: Input parameter VAR_OUTPUT: Output parameter VAR_IN_OUT: In/out parameter • In the Declaration section of a program: VAR: Local variable VAR CONSTANT: Local constant VAR_TEMP: Temporary variable • In the Declaration section of an expression: VAR: Local variable VAR CONSTANT: Local constant VAR_INPUT: Input parameter (as of Version 4.1 of the SIMOTION kernel) VAR_IN_OUT: In/out parameter (as of Version 4.1 of the SIMOTION kernel) name_list 是将要声明的 data_type 数据类型的标识符。 .
顺序	声明变量 ●在 ST源文件的 interface 部分 允许的关键词：见 Field 语法图表 单元变量在 ST源文件中被识别，将被导出 可以在 ST源文件的所有 POU中使用 除此之外，还可以在导入 ST源文件的所有单元中使用 ●在 ST源文件的 implementation 部分 允许关键词：见 Field 语法图表 单元变量在 ST源文件中被识别 可以在 ST源文件的所有 POU中使用 ●在 POU（FC,FB程序，表达式）的声明部分 允许关键词是根据 POU类型定义的，见 CELL语法图表 变量在 POU中被本地识别 只能在 POU的本地变量的说明部分中使用

	例外： 可以在 FB外访问一个功能块的输出参数 可以在 FB外访问一个功能块的输入参数，假如 ： 编辑器的允许语言扩展选项被激活。见 45 页编辑器全局设置和 46 页编辑器本地设置 变量必须在使用之前被声明
频次	相关的源文件部分将定义一个指定的变量类型的variable_type / END_VAR声明块出现的次数。 ●在 ST源文件的 interface 和 implementation 部分说明块可以出现不止一次 ●在一个 POU（FC,FB程序和表达式）的声明部分每个说明块 (除了 VAR CONSTANT/ END_VAR)在说明部分只能出现一次 允许声明块和关键词基本相关的源文件部分：见 CELL语法图表 在一个说明中允许多个变量声明。
强制部分	NO

欲知更多变量声明的信息，见 105 页变量声明。

3.1.2 在 ST 源文件之间的导入和导出

ST 应用单元的概念，在 ST中你可以访问全局变量，数据类型，功能，功能块，和程序。
因此你可以编辑重复可用的子程序，使得它们为可用的程序。

3.1.2.1 单元标识符

单元是指一个程序源文件 （如：ST源文件，MCC源文件）。程序源文件的名称在 SIMOTION SCOUT中作为标识符被定义。

你可以设置单元语句为 ST源文件（先于 interface 部分）的第一语句。语法为：
UNIT name;

定义在 SIMOTION SCOUT中的 ST源文件名称对应的名称，见 21 页添加 ST源或 23 页 ST源文件属性的更改。

如果指定的名称不同于 ST源文件的名称，那么将忽略单元语句。

3.1.2.2 一个导出单元的 interface 部分

可以在一个导出单元的 interface 部分输入下列结构。结构的语法仅在此处显示，欲知更多，见 170 页 interface 部分。

●即将导出的类型说明

TYPE
带完整说明的用户定义数据类型

●即将导出的变量说明

VAR_GLOBAL, VAR_GLOBAL RETAIN, 或 VAR_GLOBAL CONSTANT

带说明的不保留和保留的单元变量和单元常量

●即将导出的 POU（功能，功能块和程序）
说明每个即将导出的 POU的相关关键词。输入一个分号来结束

- FUNCTION_BLOCK fb_name ;
- FUNCTION fc_name ;
- PROGRAM program_name ;

说明可以以任意顺序排列。 POU 在 ST源文件的执行部分自行编程。

注意：在 interface 部分下列更多的说明是可能的，位于导出类型，变量和 POU 之前

1.使用的技术包的说明 (USEPACKAGE…)
2.使用的库的说明 (USELIB…)
3.参考其他单元以使用导出单元 (USES…)

这些导入的技术包，库和单元也被导出。关于继承性，见 181 页在导入单元的 USES语句

你必须遵循单元 （ST源文件）的 interface 部分的说明中的顺序， 见 170 页 interface 部分。否则， ST源文件的无误编辑是不可能的。

一个 ST源文件的程序必须列在 interface 部分，以便可以被指定给执行系统的一个任务 （见在 SIMOTION基础功能手册配置执行系统）。如果程序不能在 ST 源文件的 interface 部分被导出，编辑器输出一个警告信息。

仅在 ST源文件的功能和功能块不能被列在 interface 部分。

3.1.2.3 一个导出单元的例子

下面是一个导出单元的例子。通过 myUnit_B 被导入（见 183 页一个导入单元的例子）

Table 5-11 Example of an exporting unit

```
UNIT myUnit A;      // Optional, name of the ST source file
INTERFACE
    //  ... USES statement also possible here
    TYPE          // Declaration of data types to be exported
        color  :  (RED, GREEN, BLUE);
    END TYPE
    VAR GLOBAL
        cycle  : INT  :=  1;   // Declaration of the
                               // unit variables to be exported
    END VAR
    FUNCTION myFC;             // Export statement of an FC
    FUNCTION BLOCK myFB;       // Export statement of an FB
    PROGRAM myProgram A;       // Export statement of a program
                               //  (to interface with the execution system)
END INTERFACE

IMPLEMENTATION
    Function myFC  : LREAL   // Function written out
        ;      //  ...  (Statements)
    END FUNCTION

    Function BLOCK myFB        // Function block written out
        ;      //  ...  (Statements)
    END FUNCTION BLOCK

    PROGRAM myProgram A        // Program written out
        ;      //  ...  (Statements)
    END PROGRAM
END IMPLEMENTATION
```

3.1.2.4 在一个导入单元的 USES语句

在一个导入单元的 interface 或 implementation 部分输入以下语句

USES unit_name-list

单元名称列表是一个从导入的模块中通过逗号隔开的单元的列表

例子： USES unit_1, unit_2, unit_3;

这使得你可以访问在导入单元（ ST源文件， MCC 源）的 interface 部分指定或声明的的下列元素：

- 用户定义数据类型（ UDT）
- 单元变量和单元常量
- 程序，功能和功能块
- 导入的技术包，库和单元

你仅可以使用导入元素犹如存在于现在的单元中。

注意：关键词 USES仅可以在一个单元的 interface 部分或 implementation 部分中出现一次。当导入多个单元时，输入时用逗号隔开，以列表的形式输入在关键词 USES之后。

USES语句可以在一个单元的 interface 部分或 implementation 部分中出现。这有着深远的含义：

影响	在 interface 部分的 USES语句	在 implementation 部分的 USES语句
继承性	现行的单元继续导出被导入的单元；导入单元通过其他可以访问现行单元的其他单元被继承 例子： 1. Unit B 在interface 部分导入 Unit A i 2. Unit C 依次导入 Unit B. 3. Then Unit C也可自动导入 Unit A A → B → C ? A → C 因为继承性, UnitA 不能明显被导入到 UnitC	继承被干扰 例子： 1. Unit B 在 implementation 部分导入 imports Unit A 2. Unit C依次导入 Unit B. 3. Then Unit C 不能自动访问 Unit A. UnitC 必须明显导入 UnitA, 如果它需要访问 UnitA
变量声明	一个导入的数据类型的单元变量声明可能在： ●nterface 部分 ●mplementation 部分	一个导入的数据类型的单元变量声明只可能在 implementation 部分

注意：你将找到在 SIMOTION功能手册中的单元变量使用的技巧：

3.1.2.5 一个导入单元的例子

以下是一个导入单元（ myUnit_B）的例子。导入从一个导出单元（ 181 页）的例子中的 myUnit_A

Table 5-13 Example of an importing unit

```
UNIT myUnit_B;          // Optional, name of the ST source file
INTERFACE
    //  ... if required, USES statement
    PROGRAM myProgram_B;
    // Specification of programs to be exported, FB, FC //
    Data types and unit variables
END_INTERFACE

IMPLEMENTATION
    USES myUnit_A;      // Specification of unit to be imported

    VAR_GLOBAL
        myInstance   : myFB;        // Declaration of an instance
                                    // of the imported FB
        mycolor   : color;          // Declaration of a variable
                                    // of the imported data type
    END_VAR

    PROGRAM myProgram_B

        mycolor   := GREEN;         // Value assignment to a variable of the
                                    // data type to be imported
        cycle   := cycle   +  1;    // Value assignment to
                                    // imported variable
    END_PROGRAM
END_IMPLEMENTATION

```

3.2 在 SIMOTION 中的变量

总结了 ST中可用的变量

3.2.1 变量模型

下表列出了用 ST编程的所有可用的变量类型

- SIMOTION设备和技术目标的系统变量
- 全局用户变量 (I/O 变量，设备全局变量，单元变量)
- 本地用户变量（在一个程序，功能或功能块中的变量）

系统变量

变量类型	意义
SIMOTION设备的系统变量	每个 SIMOTION设备和技术目标有特定的系统变量。可以如下被访问：
技术目标的系统变量	

	●在 SIMOTION设备的所有程序中 ●从 HMI 设备 你可以在符号浏览器中监视系统变量

全局用户变量

变量类型	意义
I/O 变量	你可以分配符号名称给 SIMOTION设备或者周边设备的 I/O 地址。这使得你有以下 I/O 直接访问或过程图像访问 ●在 SIMOTION设备的所有程序中 ●从 HMI 设备 在你选择项目导航中的 I/O 元素之后，在符号浏览器中创建这些变量 你可以在符号浏览器中监视 I/O 变量
全局设备变量	可以通过 SIMOTION设备的所有程序和 HMI 设备访问用户定义变量。 在你选择项目导航中的全局设备变量之后，在符号浏览器中创建这些变量 全局设备变量可以被定义为保留值。 这意味着即使 SIMOTION设备电源无连接时将保持存储。 你可以在符号浏览器中监视全局设备变量
单元变量	所有程序，功能块和功能（ ST 源，MCC 源，LAD/FBD源）可以在一个单元中访问用户定义变量。 在单元中声明这些变量： 在 interface 部分 你可以导入这些变量到其他的单元（ ST 源，MCC源, LAD/FBD源），对于 HMI 设备是作为标准可用的。 在 implementation 部分 仅可以在相关单元中访问这些变量 可以声明单元变量为保留值。 这意味着即使 SIMOTION 设备电源无连接时将保持存储。 可以在符号浏览器中监视单元变量

本地用户变量

变量类型	意义
	可以在被定义的程序 （功能或功能块） 中访问用户定义变量
一个程序的变量（程序变量）	变量在程序中被说明。 仅可以在程序中访问变量。静态和临时数据之间的比较如下： ●根据存储区域的大小初始化静态变量。通过编辑器选项来规定存储区域。 默认根据指

	定程序的任务来初始化静态变量。 可以在符号浏览器中监视静态变量 ●每次在任务中的程序被调用时初始化临时变量 不能在符号浏览器中监视临时变量
一个功能的变量（ FC变量）	变量在一个功能（ FC）中被声明。只能在功能中访问变量。 FC变量是临时的，每次调用 FC时被初始化。 不能在符号浏览器中监视临时变量。
一个功能块的变量（ FB变量）	变量在功能块源中被声明。 只能在功能块中访问变量。静态和临时数据之间的比较如下： ●当 FB结束时保留静态变量数值， 只有当初始化 FB 实例时才能初始化静态变量，这基于 FB 的实例声明的变量类型。 可以在符号浏览器中监视静态变量 ●当 FB结束时临时变量丢失数值。 下次调用FB时，重新恢复。 不能在符号浏览器中监视临时变量。

下列源有跟多的可用信息

●在想对应的列表手册中，你可以找到 SIMOTION 技术包和 SIMOTION 设备的所有系统变量的压缩信息

●欲知技术目标的系统变量使用的更多信息，参考 SIMOTION 运动控制技术目标功能手

●在 SIMOTION 基础功能手册中你可以找到如何访问系统变量和组态数据的信息

●文件包含下列信息：

—访问带 I/O 变量的 I/O 地址（见 214 页直接访问和循环任务过程图像）

—过程图像访问

—创建和使用全局设备变量（见 193 页全局设备变量使用）

—单元变量和本地变量的使用（静态和临时变量）

注意：下载 ST 源文件到目标系统和允许的任务会影响变量初始化， 因此变量的内容， 见 200 页变量初始化时间

同时，见 220 页背景任务的固定过程图像

3.2.1.1 单元变量

在整个 ST源文件中单元变量是有效的，如：在任何的源文件部分可以访问单元变量。

在 ST源文件的 interface 和/或 implementation 部分可以声明单元变量；声明的位置决定了单元变量的有效期：

●如果在 interface 部分声明变量，你可以创建能在其他程序源（如： ST源文件， MMC 单

元）中使用的变量。欲知更多在程序源文件中的导入和导出信息，见 179 页 ST 源文件中导入和导出

默认这些单元变量在 HMI 设备上也是可用的。可以被导出到 HMI 设备的单元变量总大小限制为 64KB/每个单元。

- 如果在 implementation 部分声明单元变量，你可以在现行的源文件中所有的程序组织单元（POU）中创建变量

可以使用在声明块中的一个 pragma 来更改 HMI 单元变量导出的默认设置， 见 208 页变量和 HMI 设备及 247 页带属性的的控制编辑器。

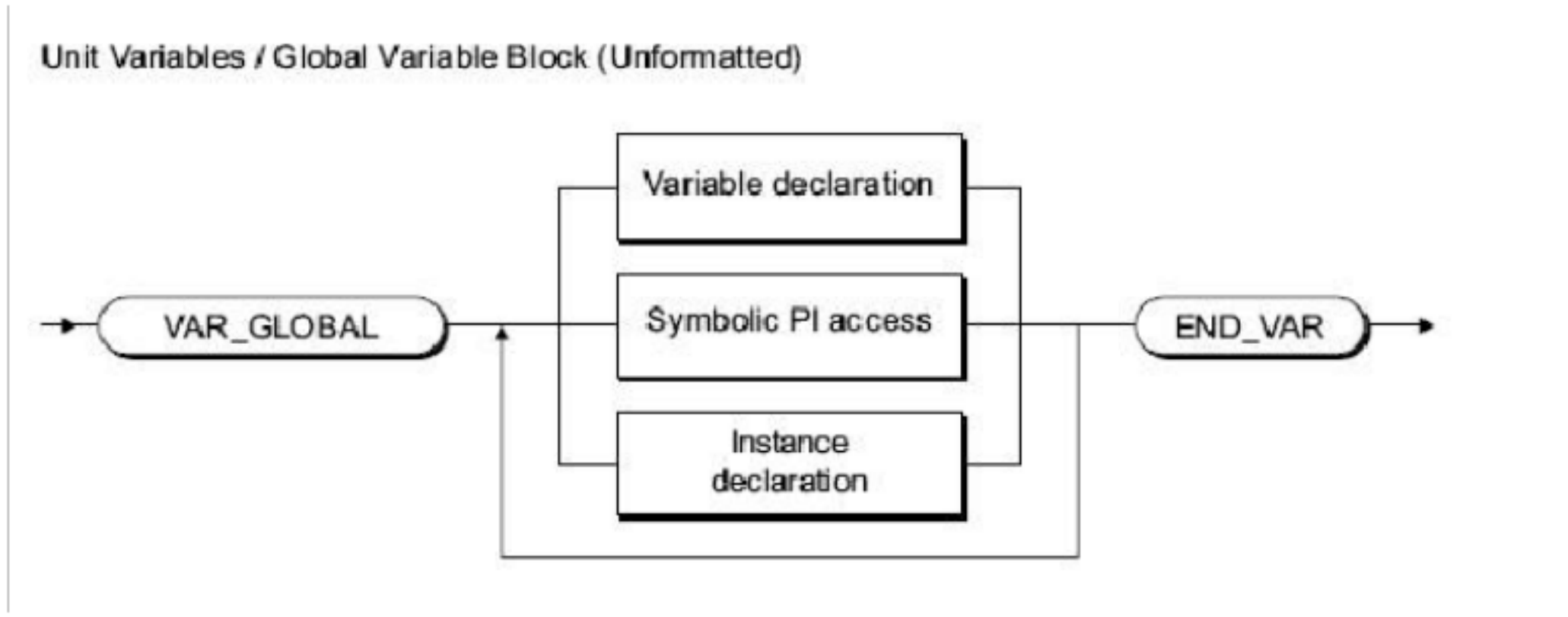
可以通过不同的行为来定义单元变量，如：电源故障

- 不保留的单元变量（关键词 VAR_GLOBAL）：数值在电源故障时丢失
- 保留的单元变量（关键词 VAR_GLOBAL RETAIN）：数值在电源故障时保留
- 单元常量（关键词 VAR_GLOBAL CONSTANT）：数值保持不变 (见 111 页常量)

你将在 SIMOTION 基础功能手册中找到单元变量有效使用的技巧。

3.2.1.2 不保留的单元变量

在电源故障时不保留的单元变量将失去数值。



说明块在一个 interface 或 implementation 部分中可能出现不止一次。指定变量名称和数据类型为变量说明 （见 106 页所有变量声明的概述和 107 页变量或数据类型的初始化） 。

声明的范围和 HMI 导出，见 186 页单元变量

注意：不保留单元变量的初始化

- 见 202 页不保留变量的初始化

●在下载时的动作可以被设置（选项　>设置菜单命令，项目下载标签，初始化所有的不保留设备全局变量和程序数据检查框）

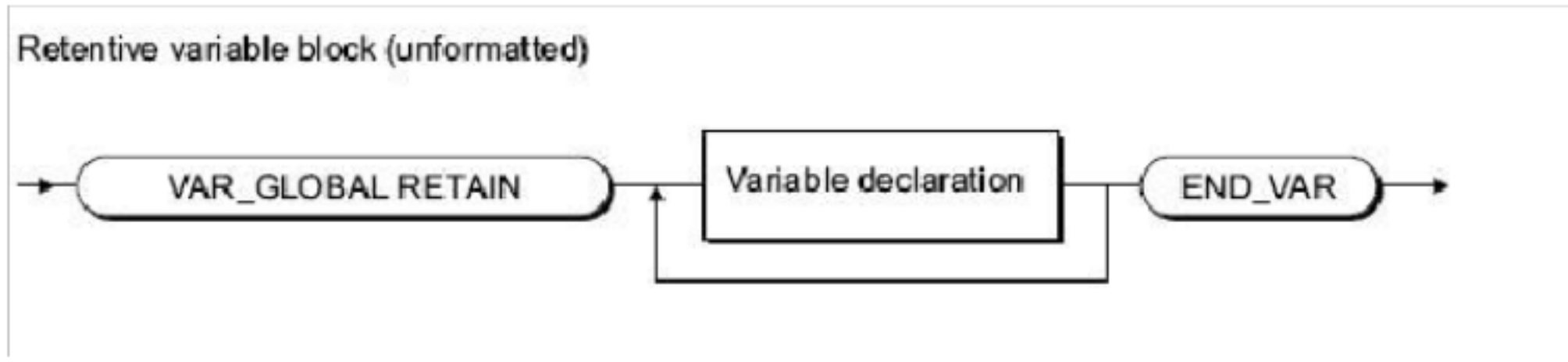
●版本 ID 的类型和下载的初始化动作都基于　SIMOTION Kernel版本。欲知更多，见　207 页全局变量的版本　ID 和下载时的初始化。

Table 5-14　Examples of non-retentive unit variables

```
INTERFACE
    VAR_GLOBAL     //These variables can be exported.
        rotation1          : INT;
        field1             : ARRAY [1..10] OF REAL;
        flag1              : BOOL;
        motor1             : motor;    // Instance declaration
    END_VAR
END_INTERFACE
IMPLEMENTATION
    VAR_GLOBAL     //These variables cannot be exported
                   //  MotionTask.
        rotation2          : INT;
        field2             : ARRAY [1..10] OF REAL;
        flag2              : BOOL;
        motor2             : motor;    // Instance declaration
    END_VAR
END_IMPLEMENTATION
```

3.2.1.3 保持单元变量

保留单元变量允许变量数值的永久存储，即使在电源故障时。



说明块在一个　interface 或 implementation　部分中可能出现不止一次。指定变量名称和数据类型为变量说明 （见 106 页所有变量声明的概述和　107 页变量或数据类型的初始化）　。

声明的范围和　HMI 导出，见　186 页单元变量

注意：保留单元变量的初始化

●见 201 页保留全局变量的初始化

●在下载时的动作可以被设置（选项　>设置菜单命令，项目下载标签，初始化所有的不保

留设备全局变量和程序数据检查框）

●版本 ID 的类型和下载的初始化动作都基于 SIMOTION Kerne版本。欲知更多，见 207 页全局变量的版本 ID 和下载时的初始化。

Table 5-15 Examples of retentive variables

```
VAR_GLOBAL RETAIN
    Measuring field : ARRAY[1.0.10] OF REAL;
    Pass : INT;
    Switch: BOOL;
END_VAR
```

3.2.1.4 本地变量（静态和临时变量）

本地变量仅在声明的源文件部分（如：程序, FC,FB）是有效的。我们在下列内容中加以区别：

●静态变量（ 192 页）
静态变量在整个源文件部分（块内存）保持数值

●临时变量（ 204 页）
临时变量在每次重新调用源文件部分的时候被初始化。见 204 页本地变量的初始化

注意：本地变量不能在声明的源文件部分之外被访问。

下表提供了静态和临时变量声明的概述。下表说明可在源文件部分中的哪些变量可以被声明金额使用来声明的关键词

Source file section	Keywords for the declaration	
	Static variables	Temporar
Function	–	VAR / EI c VAR_INPUT c VAR_IN_OUT
Expression	–	VAR / EI c VAR_INPUT c VAR_IN_OUT
Function block	VAR / END_VAR[1] or VAR_INPUT / END_VAR[1] or VAR_OUTPUT / END_VAR[1]	VAR_TEMP c VAR_IN_OUT
Program	VAR / END_VAR[3]	VAR_TEMP

1.变量的初始化基于声明实例的初始化，见　　205 页功能块的实例初始化。

2.转移变量的参考（指针）是临时的。

3.变量的初始化基于存储的内存区域。见　　204 页静态程序变量的初始化。

注意：请注意下载　ST 源文件到目标系统和允许的任务会影响变量初始化，　　因此变量的内容，见 200 页变量初始化时间。

Table 5-17 Examples of static and temporary variables

```
IMPLEMENTATION
    FUNCTION testFkt
        VAR             // Declaration of temporary variables
            flag : BOOL;
        END_VAR
    END_FUNCTION
    FUNCTION_BLOCK testFbst;
        VAR             // Declaration of static variables
            rotation1       : INT;
        END_VAR

        VAR_TEMP        // Declaration of temporary variables
            help1, help2 : REAL;
        END_VAR
    END_FUNCTION_BLOCK
    PROGRAM testPrg;
        VAR             // Declaration of static variables
            rotation2       : INT;
        END_VAR

        VAR_TEMP        // Declaration of temporary variables
            help1, help2 : REAL;
        END_VAR
    END_PROGRAM
END_IMPLEMENTATION
```

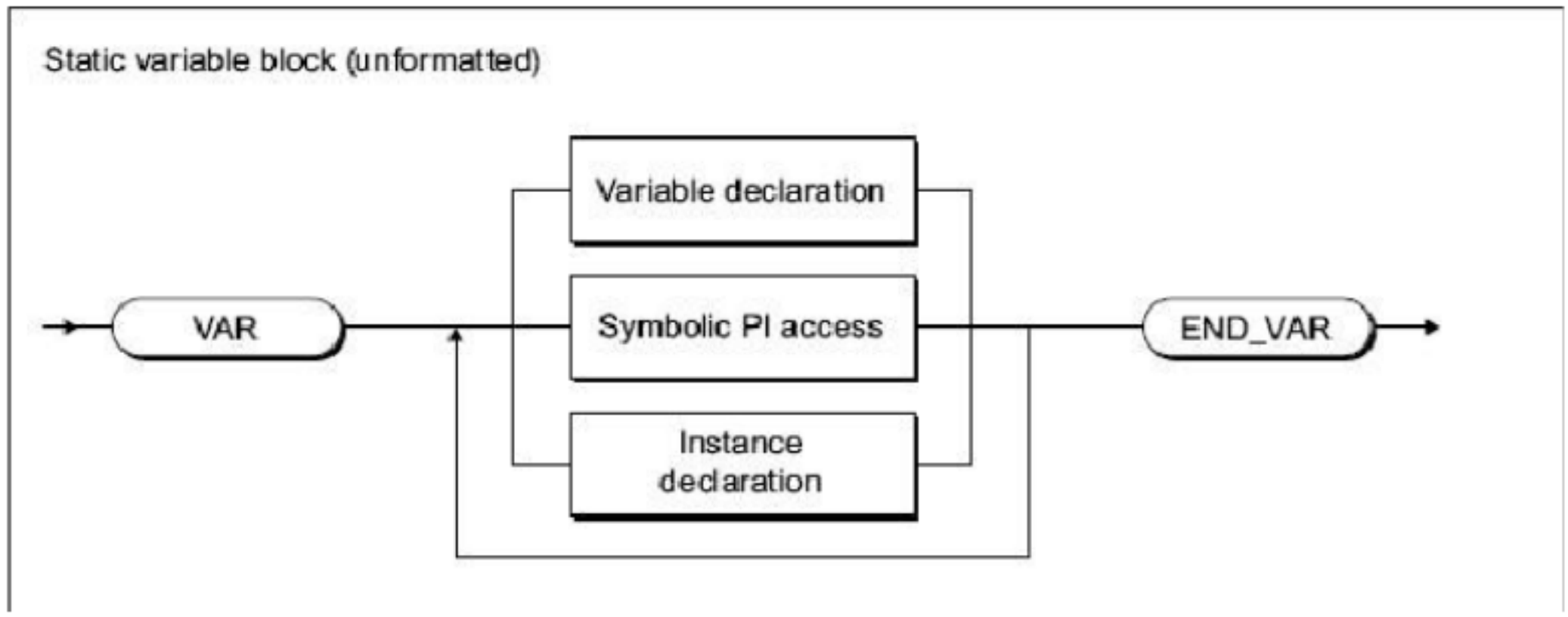
3.2.1.5 静态变量

当退出源文件部分时， 静态变量保留其最近的数值。 这个数值在下次调用时将重新使用。

下列源文件部分包含静态变量：

- 程序
- 功能块

静态变量在一个静态变量块中被说明



根据语法图，你可以在一个静态变量块中进行下列操作：

- 声明变量（名称和数据类型），选择性地初始化
- 声明到背景任务的过程图像的符号访问
- 声明功能块的实例

对于静态变量的初始化

- 在程序中：基于到指定程序的执行动作（见 SIMOTION 基础功能手册和 204 页静态程序变量的初始化）
- 在功能块中：基于声明实例的初始化（见 205 页功能块实例的初始化）

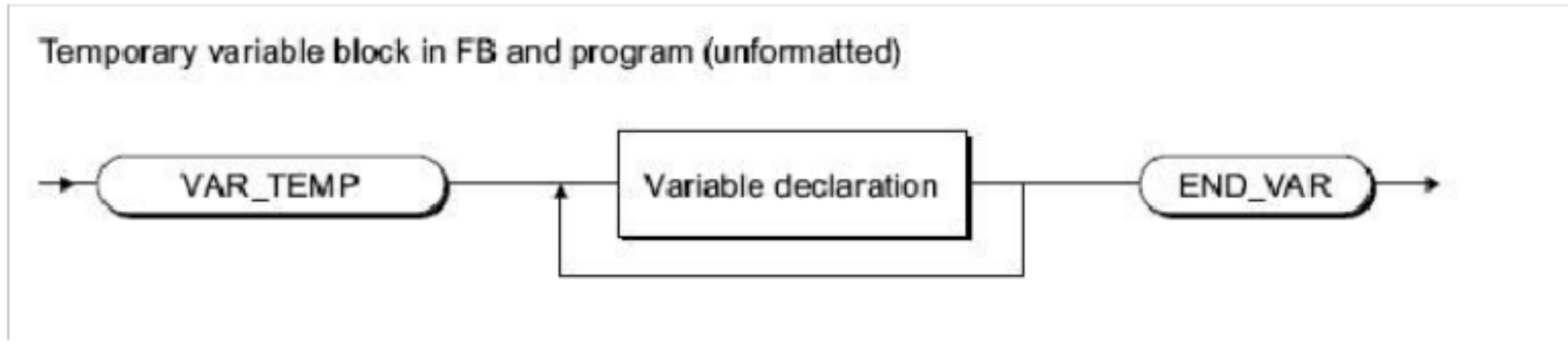
3.2.1.6 临时变量

在每次调用源文件时初始化临时变量。仅在执行源文件时数值保留。

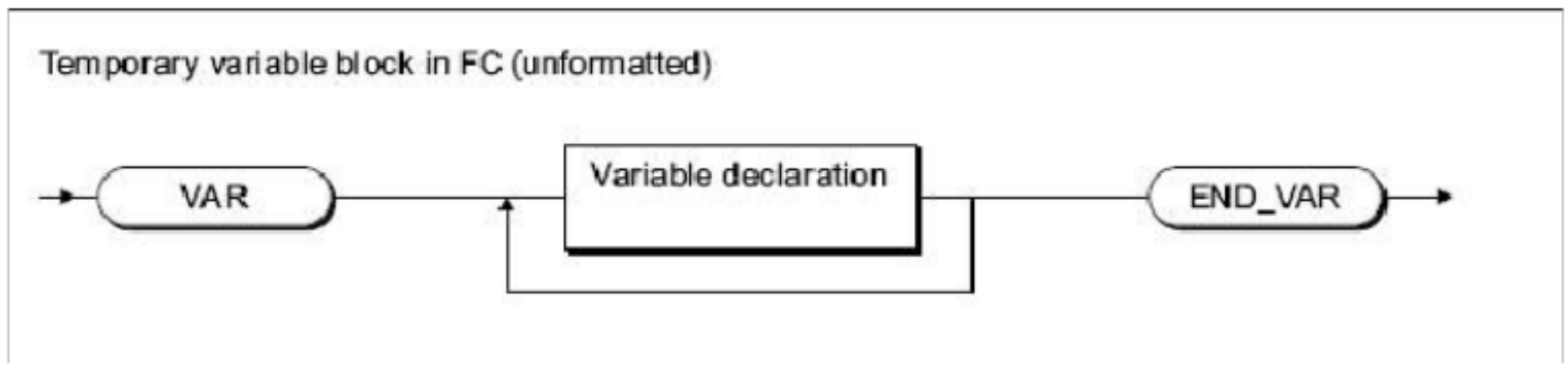
下列源文件部分包含临时变量：

- 功能块
- 功能
- 表达式

在功能和表达式中，你可以在 FB临时变量块中声明临时变量（见下图）



在功能和表达式中，你可以在 FC临时变量块中声明临时变量（见下图）



3.2.2 全局设备变量的使用

全局设备变量是你可以从一个 SIMOTION设备的所有的程序源 （如 ST源文件，MCC单元）访问的用户定义的数据

全局设备变量在符号浏览器标签的细节浏览部分被创建。要使用此功能，你必须在离线模式下进行。

下列是这个规程的简要描述

1.在 SIMOTION SCOUT的项目导航中，在 SIMOTION 设备子目录中选择 GLOBALDEVICE VARIABLES元素

2.在细节浏览中，选择符号浏览器标签，然后滚动到最末的变量表（空行）

3.在下表的最后一行，输入或选择以下：

—变量名称

—变量的数据类型（只允许基础数据类型）

4.或者你可以选择性地输入以下：

—选择保留检查框 (把变量作为保留值声明，以便在电源故障时保留数值)

—阵列长度（阵列大小）

—初始值（如果阵列，每个元素）

—显示格式（如果阵列，每个元素）

现在你可以使用符号浏览器或 SIMOTION设备的任意程序来访问变量。

在 ST源文件中，你可以使用一个全局设备变量，如同其他变量一样。

注意：

如果你已经声明了单元变量或本地变量为同样的名称 (如. var-name), 指定全局设备变量为 _device.var-name

另外一种全局设备变量为在隔开的单元中的单元变量的声明，隔开的单元被导入到其他单元中，这有着以下的优势：

1.可以使用变量结构

2.在停止—运行的转变时变量的初始化（通过启动任务的程序）

3.对于最新创建的全局单元变量，在运行时的一次下载是可以的。

请见 SIMOTION基础功能参数

3.2.3 变量类型的存储范围

不同的变量类型存储在不同的存储区域，在不同的时间被初始化，下表显示了：

●在 ST源文件中声明的变量类型的可用存储区域（基于 SIMOTION Kernel版本）

●每个存储区域的初始化时间

使用例子的解释包含在存储区域的例子中，作为 Kernel V3.1（见 196 页）有效

存储区	指定的变量类型	初始化
保留内存	保留单元变量	在下载时使用下载设置
单元的用户内存	●不保留的单元变量 ●带 VAR_GLOBAL 声明的功能块实例 ,包括相关的静态变量 (VAR, VAR_INPUT, VAR_OUTPUT) 激活编辑器中仅此一次创建程序实例数据（ 44 页） ●带 VAR 声明的单元程序的本地变量 ●带 VAR_GLOBAL 声明的功能块实例 ,包括相关的静态变量 (VAR, VAR_INPUT, VAR_OUTPUT)	●当设备开启时 ●在下载时使用下载设置 ●在 SIMOTION V4.1中，当相关的声明块指定下列 pragma{ BlockInit_OnDeviceRun := ALWAYS; }时的运行模式的转变 见 247 页控制带属性的编辑器
任务的用户内存	对于 44 页编辑器选项“仅此一次创建程序实例数据” ●带 VAR 的指定程序的本地变量 ●时带 VAR 的在指定的程序中的功能块实例，包括相关的静态变量 (VAR, VAR_INPUT, VAR_OUTPUT)	根据任务的执行动作： ●顺序任务 每次开始顺序时 ●循环任务 CPU转到运行模式时
任务中的本地数据 （关于于 SIMOTION KernelV3.1 版本） [2]	●参见（指针）在任务中调用的程序 ●带 VAR TEMP 的任务中调用的程序的声明的本地	在任务中每次调用程序时

	变量	
	●参见（指针）在任务中调用的功能块 ●带 VAR_TEMP 声明的功能块 ●带 VAR_IN_OUT 的声明的功能块的 in/out 参数	在任务中每次调用功能块实例时
	●带 VAR, VAR_INPUT或VAR_OUTPUT的声明的调用功能变量 ●调用功能的回送值	在任务中每次调用功能实例时
（高于 SIMOTION KernelV3.0 版本）[3]	●在任务中调用的程序的复制数据，包括所有相关的变量 (VAR, VAR_TEMP)	在任务中每次调用程序时
	●从被调用的功能块的实例中复制来的数据 (VAR, VAR_INPUT, VAR_OUTPUT, VAR_IN_OUT1, VAR_TEMP)	在任务中每次调用功能块实例时
	●带 VAR, VAR_INPUT or VAR_IN_OUT1 声明被调用的功能变量 ●调用的功能的回送值	在任务中每次调用功能实例时

1.参见转移的变量

2.参考 SIMOTION设备编辑的库的使用和相关的 SIMOTION Kernel版本（关于 V3.1 版本），见本地数据变量的存储要求（关于 V3.1 的 Kernel 版本）

3.基于设备编辑的库的使用（如：不参考 SIMOTION设备和 SIMOTION kernel 版本），见本地数据变量的存储要求（高于 V3.1 的 Kernel 版本）

4.单独的变量类型的初始化动作的详细描述，见 200 页变量初始化的时间

3.2.3.1 存储区域的例子，有效关于 KernelV3.1

Table 5-19 Example of memory ranges of the variable types, as of Kernel V3.1 (Part 1)

```
INTERFACE
// The statements in the interface section specify,
// what source content is exported.
    FUNCTION FC1;
    FUNCTION_BLOCK FB1;
    PROGRAM p1;

    // Unit variables of the interface section are also visible
    // on HMI devices.
    VAR_GLOBAL                  // Non-retentive unit variables
                                // are present in the UNIT user memory
        u1_if : INT;
    END_VAR
    VAR_GLOBAL CONSTANT         // Unit constants are located
                                // in the unit user memory
    END_VAR
    VAR_GLOBAL RETAIN           // Retentive unit variables are located
                                // in the retentive (power-fail-safe) memor
    END_VAR
END_INTERFACE

IMPLEMENTATION
// The implementation section contains the executable code sections
// in different program organization units (POU)
// A POU can be a program, FC, or FB.
    // Unit variables of the implementation section can only be used
    // within the source file.
    VAR_GLOBAL                  // Non-retentive unit variables are locatec
                                // in the unit user memory
        u1_glob : INT;
    END_VAR
    VAR_GLOBAL CONSTANT         // Unit constants are located
                                // in the unit user memory
    END_VAR
    VAR_GLOBAL RETAIN           // Retentive unit variables are located
                                // in the retentive (power-fail-safe) memor
    END_VAR
//-----------------------------------------------------------------------
```

Table 5-20 Example of memory ranges of the variable types, as of Kernel V3.1 (Part 2)

```
// Continuation
//-----------------------------------------------------------------------
    FUNCTION_BLOCK FB1 // Declaration of an instance
    // instance determines where its data are located:
    // - as VAR_GLOBAL in a unit:
    // in the unit user memory
    // - as VAR in a program:
    // in the user memory of the task (default)
    // - As VAR in a function block:
    //     in the user memory of the unit or task,
    //     depending on the instance declaration of the higher-level FB
    // When the instance is called, a pointer to the instance data
    // is placed on the stack of the calling task

     VAR_INPUT          // Input parameters
                        // are in the user memory
                        // are written when the instance is called
         fb_in      : INT;
     END_VAR
     VAR_OUTPUT         // Output parameters
                        // are in the user memory
         fb_out     : INT;
     END_VAR
     VAR_IN_OUT         // In/out parameter
                        // references are in the user memory
                        // are written when the instance is called
        fb_in_out : INT;
    END_VAR

    VAR                 // Static variables
                        // are in the user memory
                        // can be used locally in the FB
         fb_var1    : INT;
    END_VAR

    VAR_TEMP            // Temporary variables
                        // are on the stack of the calling task
                        // are initialized on each call
        fb_temp1   : INT;
    END_VAR

        // Code is in the user memory of the unit
        fb_var1    := fb_var1 + 1;
        fb_out     := fb_var1;
        fb_temp1   := fb_in_out;
        fb_in_out := fb_temp1 + fb_in;
    END_FUNCTION_BLOCK
//-----------------------------------------------------------------------
```

Table 5-21 Example of memory ranges of the variable types, as of Kernel V3.1 (Part 3)

```
// Continuation
//-------------------------------------------------------------------
    FUNCTION FC1 : INT     // The function data is on the
        // stack of the calling task; they are initialized each time
        // the function is called.
        // The return value is on the stack of the calling task

    VAR_INPUT              // Input parameters
                           // are on the stack of the calling task
                           // are written when the function is called
        fc_in   : INT;
    END_VAR

    VAR                    // Temporary variables
                           // are on the stack of the calling task
        fc_var  : INT;
    END_VAR
    // Code is in the user memory of the unit
    fc_var := 567;
    fc1 := fc_in + fc_var;
END_FUNCTION

PROGRAM p1
    VAR                    // By default, variables are located in the
                           // in the user memory of the task
        p_var   : INT;
        p_varFB : FB1;
    END_VAR

    VAR_TEMP               // Temporary variables
                           // are on the stack of the task,
                           // are initialized on each task pass
        p_temp  : INT;
    END_VAR

    // Code is in the user memory of the unit
    p_temp  := p_var;
    p_varFB (fb_in_out := p_temp);
    u1_glob := 4711;
END_PROGRAM
END_IMPLEMENTATION
```

3.2.3.2 本地数据栈变量的存储要求（ kernel V3.1 或更高）

存储在一个任务的本地数据栈的变量列在 194 页的变量类型的存储区域。在任务组态中

设置栈的大小。

注意下列本地栈中的存储要求：
●临时本地变量要求在栈中有各自的大小
●全局变量和静态变量不需要在栈中要求任何资源。　　如果作为一个功能的输入参数使用它们，要求有各自的栈的数据大小
●即使在一个任务重多次调用一个功能，也只使用一次栈的资源
●BOOL类型的变量在栈中需要　1 字节

注意：
对于参考 SIMOTION设备编辑的库的使用和相关的　SIMOTION Kernel版本（关于 V3.1 版本），下列细节也是　TRUE。

注意：
如果库是不基于设备的（如：不参考　　SIMOTION设备和 SIMOTION kernel版本）：这些库编译为与允许的　SIMOTION kernel版本相兼容。

从这些库中调用的　POU 变量占用本地数据栈。见　199 页在本地数据栈的变量的存储要求（高于 SIMOTION kernel V3.0）

3.2.3.3 本地数据栈变量的存储要求（　kernelV3.0　或更低）

存储在一个任务的本地数据栈的变量列在　　194 页的变量类型的存储区域。在任务组态中设置栈的大小。

注意下列本地栈中的存储要求：
●在程序中的静态变量要求在数据栈中加倍它的大小
●在 FB 中的静态数据变量在数据栈中需要多次，基于调用长度
●临时本地变量（在程序，　FB，FC）在栈中需要各自的大小
●全局变量不占用任何的栈存储空间
如果作为功能或功能块的输入参数来使用，将会占用数据栈上的通常大小
●即使在一个任务中多次调用一个功能，仅此一次使用栈的资源
●BOOL 类型的变量需要栈上　1 个字节的大小

注意：
当调用一个功能块实例时，所有的实例数据被复制到本地数据栈，即使实例作为 VAR_GLOBAL 实例被声明。

如果库是不基于设备（如：不参考　　SIMOTION设备和 SIMOTION kernel版本）：这些库编译为与允许的 SIMOTION kernel版本相兼容。从这些库中调用的　POU变量占用本地数据栈，在本章中描述。

在本地数据栈的存储要求大于　SIMOTION kernel版本（关于 kernel V3.1），见 199 页本地数据栈的变量的存储要求。在任务组态中设置栈大小时需要考虑这些问题。

3.2.4 变量初始化的时间

变量初始化时间由下列内容决定：

●变量被指定的存储区域
●算子运作（如：源文件下载到目标系统）
●指定到程序的任务的执行动作（顺序，循环）。所有的变量类型和变量初始化的时间在下表中列出。你将在 SIMOTION基础功能手册中找到更多任务的信息。

在下载时变量初始化的动作可以设置。要想如此，默认设置为 Options > 设置菜单和下载标签或定义现在下载时的设置

注意：
可以从 SIMOTION设备上传单元变量或全局设备变量数值给 SIMOTION SCOUT，以 XML 格式保存。
1.把需要的单元变量或全局设备变量数据段作为带功能 _saveUnitDataSet 的数据集
2.使用在 SIMOTION SCOUT 中的保存变量功能

你可以使用恢复的变量功能来下载这些数据集和变量到 SIMOTION设备。

欲知更多信息，参考 SIMOTION SCOUT 组态手册。

这允许一些可能性，如：即使不可使用或通过一个项目下载初始化时，也可以获得数据。（见 SIMOTION SCOUT 版本变更）

3.2.4.1 保留全局变量的初始化

保留变量在电源断开后任然保留其数值。当设备再次开启时，所有其他的数据重新启动。

保留全局变量被初始化：
●当保留数据备份或缓冲失败时
●当硬件更新时
●当执行存储重置（MRES）时
●SIMOTION P350中的重启功能（Del.SRAM）
●通过应用 _resetUnitData 功能 (关于 kernel V3.2)，有选择的保留数据的不同数据段
●根据下列描述执行下载时

变量类型	变量初始化的时间
保留全局设备变量	下载时的动作基于所有保留全局设备变量和程序数据设置[1]： ●YES[2]：所有保留全局变量被初始化 ●NO[3]：

	—关于版本 V3.2 的 SIMOTION kernel 隔开保留设备变量的版本 ID。如果版本 ID 改变，保留全局设备变量会被初始化。 —高于版本 V3.1 的 SIMOTION kernel 组合所有全局设备变量 （保留和不保留的） 的版本 ID。如果版本 ID 改变，保留全局设备变量会被初始化。 见 207 页全局变量的版本号和下载时的初始化
保留单元变量	下载时的动作基于所有保留全局设备变量和程序数据设置 [1]： ●YES [2] ：所有保留单元局变量（所有单元）被初始化 ●NO [3] ： —关于版本 V3.2 的 SIMOTION kernel 在 interface 或 implementation 部分隔开保留单元变量的每个单独数据块 (= declaration block)4 的版本 ID。如果版本标识符改变，只有相关的数据块会被初始化 [5]。 —高于版本 V3.1 的 SIMOTION kernel 所有单元变量（保留和不保留的，在 interface 和 implementation 部分）的常用的版本 ID。如果版本 ID 改变，单元的所有单元变量被初始化。 见 207 页全局变量的版本号和下载时的初始化

1.默认设置 Options > 设置菜单 ,下载标签 ,或下载时的现有设置
2.激活对应的检查框
3.关闭对应的检查框
4.许多在 interface 或 implementation 部分的保留单元变量的数据块只能在 SIMOTION ST 编程语言中被声明。关于 SIMOTION MCC 和 SIMOTION LAD/FBD 编程语言，只有一个保留单元变量的数据块在 implementation 或执行部分被创建
5.在运行时的下载，假设满足相关的条件，在相关的声明块中用一个 pragma(仅对 SIMOTION ST 编程语言)指定下列属性： { BlockInit_OnChange := TRUE; }.
对于在运行中的下载，见 SIMOTION基本功能手册

3.2.4.2 不保留的全局变量的初始化

在电源断开时不保留全局变量丢失数值。它们被初始化：

- 对于 201 页保留全局变量的初始化，如：在固件跟新或普通重置时（ MRES）
- 在上电时
- 通过应用 _resetUnitData 功能 (关于 kernel V3.2)，有选择的不保留数据的不同数据段
- 在下载时与下面中的描述保持一致
- 仅对于 SIMOTION kernelV4.1和不保留单元变量

对于转移到运行模式， 相关的含 PRAGMA的声明模块指定下列属性 （仅对于 SIMOTION ST 编程语言）： { BlockInit_OnDeviceRun := ALWAYS; }

变量类型	变量初始化的时间

不保留的全局设备变量	下载时的动作基于所有不保留全局设备变量和程序数据设置 [1]： ●YES[2] ：所有不保留全局变量被初始化 ●NO[3] ： —关于版本 V3.2 的 SIMOTION kernel 隔开不保留设备变量的版本 ID。如果版本 ID 改变，不保留全局设备变量会被初始化。 —高于版本 V3.1 的 SIMOTION kernel 组合所有全局设备变量（保留和不保留的）的版本 ID。如果版本 ID 改变，保留全局设备变量会被初始化。 见 207 页全局变量的版本号和下载时的初始化
不保留的单元变量	下载时的动作基于所有不保留全局设备变量和程序数据设置 [1]： ●YES[2] ：所有不保留单元局变量（所有单元）被初始化 ●NO[3] ： —关于版本 V3.2 的 SIMOTION kernel 在 interface 或 implementation 部分隔开保留单元变量的每个单独数据块(= declaration block)4 的版本 ID。如果版本标识符改变， 只有相关的数据块会被初始化 [5]。 —高于版本 V3.1 的 SIMOTION kernel 所有单元变量（保留和不保留的，在 interface 和 implementation 部分）的常用的版本 ID。如果版本 ID 改变，单元的所有单元变量被初始化。 见 207 页全局变量的版本号和下载时的初始化

1.默认设置 Options > 设置菜单 ,下载标签 ,或下载时的现有设置

2.激活对应的检查框

3.关闭对应的检查框

4.许多在 interface 或 implementation 部分的不保留单元变量的数据块只能在 SIMOTION ST 编程语言中被声明。关于 SIMOTION MCC 和 SIMOTION LAD/FBD 编程语言，只有一个不保留单元变量的数据块在 implementation 或执行部分被创建

5.在运行时的下载，假设满足相关的条件，在相关的声明块中用一个 pragma(仅对 SIMOTION ST 编程语言)指定下列属性： { BlockInit_OnChange := TRUE; }.
对于在运行中的下载，见 SIMOTION基本功能手册

3.2.4.3 本地变量的初始化

本地变量被初始化

●对于保留单元变量的初始化（ 201 页）

●对于不保留单元变量的初始化（ 202 页）

还有根据下面的描述：

变量类型	变量初始化的时间
本地程序变量	程序的本地变量初始化不同： ●静态变量（ VAR）根据存储区域初始化， 见 204 页静态程序变量的初始化 ●临时变量（VAR_TEMP）在每次调用程序的任务时初始化
功能块的本地变量	功能块的本地变量初始化不同： ●静态变量（ VAR, VAR_IN, VAR_OUT）仅在 FB 实例被初始化时初始化，见 205 页功能块实例的初始化 ●临时变量（ VAR_TEMP）在每次调用 FB时初始化
功能的本地变量	功能的本地变量是临时的，在每次调用功能时初始化

注意：你可以获得关于在本地数据栈中使用编程结构功能（见 239 页）的 POU 的存储要求

3.2.4.4 静态编程变量的初始化

下列版本影响下列静态变量：

●一个带 VAR声明的单元程序的本地变量

●在一个单元程序中带 VAR 声明的功能块实例，包括相关的静态变量 (VAR, VAR_INPUT, VAR_OUTPUT).

初始化动作由存储静态变量的存储区域决定。这是由编辑器中“ 仅此一次创建程序实例数据 ”（ 44 页）。

●对于未激活的编辑器中“ 仅此一次创建程序实例数据 ”（默认）：静态变量存储在每个指定给程序的任务中的用户内存中。

变量的初始化基于指定给程序的任务的执行动作（见 SIMOTION基本功能手册）

—顺序任务 (MotionTasks, UserInterruptTasks, SystemInterruptTasks, StartupTask, ShutdownTask)每次开启任务时初始化静态变量

—循环任务 (BackgroundTask,SynchronousTasks,TimerInterruptTasks):每次转到运行模式时初始化静态变量

●对于激活的编辑器中“ 仅此一次创建程序实例数据 ”：设置是必要的，例如：如果从程序内调用一个程序，静态变量仅此一次存储在任务的用户内存中。

—与不保留单元变量一起被初始化，见 202 页不保留全局变量的初始化

—只关于 SIMOTION kernel版本 V4.1

可以在转到运行模式时被初始化。要想如此，下列属性必须在相关的带 pragma（仅 SIMOTION ST编程语言）声明块中被指定： { BlockInit_OnDeviceRun := ALWAYS; }

3.2.4.5 功能块实例的初始化

功能块实例（ 157 页）的初始化由声明的位置决定：

- 全局声明（在 implementation 部分的 interface 的 VAR_GLOBAL/END_VAR 中）
  作为一个不保留单元变量初始化，见 202 页不保留全局变量的初始化
- 在程序中的本地声明（带 VAR / END_VAR）
  程序静态变量的初始化，见 204 页程序静态变量的初始化
- 在一个功能块中的本地声明（带 VAR / END_VAR）
  作为功能块的实例初始化
- 在一个功能块或功能中作为 in/out 参数声明（ VAR_IN_OUT / END_VAR）
  对于 POU的初始化，只有参考（指针）被初始化，功能块的实例保持不变

注意：你可以在本地数据栈中使用程序结构 （239 页）功能获得 POU的存储要求的信息。

3.2.4.6 技术目标的系统变量的初始化

技术目标的系统变量通常是不保留的。基于技术目标，一些系统变量存储在保留的存储区域（如：绝对编码器校验）

保留和不保留全局变量的初始化动作（除了下载时）是一样的。见 201 页保留全局变量的初始化和 202 页不保留全局变量的初始化。

在下载时的动作说明如下：

- 不保留系统变量
- 保留系统变量

变量类型	变量初始化的时间
不保留系统变量	下载时的动作基于技术目标设置 [1] 的所有不保留数据： ● YES [2] ：所有技术目标被初始化 —所有的技术目标重组，所有不保留系统变量被初始化 —所有技术报警被清除 ● NO [3] ：只有在 SIMOTION SCOUT 中更改的技术目标被初始化 —在问句中的技术目标重组，所有不保留系统变量被初始化 —清除所有搁置技术目标的报警 —如果一个报警只能在上电时被确认，搁置了技术目标，那么不会初始化，放弃下载
保留系统变量	只有当技术目标在 SIMOTION SCOUT 中更改时，保留系统变量才会被初始化。 所有其他的技术目标的保留系统变量被保存（如：绝对编码器校准）

1.默认设置 Options > 设置菜单 ,下载标签 ,或下载时的现有设置 2.激活对应的检查框 3.关闭对应的检查框

3.2.4.7 全局变量的版本 ID 和下载时的初始化

数据段			高于 SIMOTION kernel 版本 V3.1
全局设备变量			
	保留全局设备变量	●隔开每个数据段的全局设备变量的版本 ID ●数据段的版本标识符变更： —在数据段中添加或删除一个变量 —在数据段中标识符的变更或一个变量的数据类型 ●版本 ID 不会变更 —其他数据段中的更改 —初始化数值的更改 ●在下载时 [2]，规则是：仅版本 ID 更改时初始化数据段 ●使用功能做数据备份和初始化是可能的	●全局设备变量的所有数据段的普通版本 ID ●当数据段中的变量声明更改时，更改版本 ID ●在下载时 [2]，规则是：仅版本 ID 更改时初始化数据段 ●使用功能来备份数据是可能的
	不保留全局设备变量		
一个单元的单元变量			
	在 interface 部分的保留单元变量	●在每个数据段中有一些数据块（=声明块）[3] 是可能的 ●每个数据块的版本 ID ●数据块的版本标识符更改 —在相关的声明块中添加或删除一个变量 —在相关的声明块中更改标识符或更改一个变量的数据类型 —更改在相关的声明块中使用的数据类型定义（从隔开的或导入 [4] 的单元） —在相同的数据段中，相关的声明块之前添加或删除声明块 ●版本 ID 不会更改 —在其他数据段添加或删除声明块 —在相同的数据段，相关的声明块之后添加或删除声明块 在其他数据块的更改	●在每个数据段（也指多个声明块）[3] 中一个数据块 ●在一个单元中所有全局变量的普通版本 ID ●版本 ID 的改变要对应以下： —在一个数据段中的变量声明 —在单元中的全局数据类型的声明 —在 interface 部分的导入 [4] 单元的声明 ●在下载时 [2]，规则是：仅版本 ID 更改时初始化数据段 ●使用功能来备份数据仅可能在：在 interface 部分不保留单元变量
	在 implementation 部分的保留单元变量		
	在 interface 部分的不保留单元变量		
	在 implementation 部分的不保留单元变量		

		—初始化数值的更改 —不使用在相关的数据块中的数据类型定义的更改 ●在下载时 [2]，规则是：仅版本 ID 更改时初始化数据段 ●数据备份和初始化时考虑数据块的版本 ID	

1.只有初始化问题的数据块或数据段，更改的初始化数值才能生效。

2.如果所有保留全局变量和程序数据的初始化 =No 和所有不保留全局设备变量的初始化和程序数据 =NO

在其他设置中，见 201 页保留全局变量的初始化和 202 页不保留全局变量的初始化

3.每个数据段的多个声明块仅在 SIMOTION ST编程语言中可能。对于 SIMOTION MCC和 SIMOTION LAD/FBD编程语言，每个数据段仅能创建一个声明块。

4.单元的导入基于编程语言，见 181 页相关章节

5.对于运行时的下载，假设满足相关的条件，在相关的声明块中用 pragma 指定下列属性（仅指 SIMOTION ST编程语言）：{ BlockInit_OnChange := TRUE; }

对于在运行时的下载，见 SIMOTION 基础功能手册

3.2.5 变量和 HMI 设备

下列变量被导出到可用的 HMI 设备：

●SIMOTION 设备的系统变量

●技术目标的系统变量

●I/O 变量

●全局设备变量

●nterface 部分的保留和不保留单元变量（默认设置）。每个声明块的默认设置可以使用 pragma 改变：

{ HMI_Export := FALSE; }

带这种标识符声明块的单元变量不会被导出到 HMI 设备。HMI 一致性检查在下载时也被省略。

见 247 页带属性的控制编辑器

下列变量不会被导出到 HMI 涉笔，在下列情况也不可用：

●nterface 部分的保留和不保留单元变量（默认设置）。每个声明块的默认设置可以使用 pragma 改变：

{ HMI_Export := TRUE; }

带这种标识符声明块的单元变量会被导出到 HMI 设备。在下载时需要接受 HMI 一致性检查。

见 247 页带属性的控制编辑器。

●一个 POU 的本地变量

注意：

可以导出到 HMI 设备的单元变量的大小被限制为 64KB每单元。

pragma { HMI_Export := FALSE; } 和{ HMI_Export := TRUE; }基于 SIMOTIONkernel 的版本

●Pragma 影响从对应的声明块到 HMI 设备和 HMI 地址空间结构的导出：

—在声明块中导出到 HMI 设备的变量占用 HMI 地址空间结构

—在 HMI 地址空间中，变量根据声明的顺序安排

●高于 SIMOTION kernelV3.2 或 V4.0 的版本

Pragma 仅仅影响对应声明块到 HMI 设备的导出。

HMI 地址空间被没有指定给 HMI 设备的声明块的 interface 部分的单元变量占用。

在 HMI 地址空间中，变量以下列顺序分类：

—interface 部分的保留单元变量（导出和不导出的）

—implementation 部分的保留单元变量（导出的）

—interface 部分的不保留单元变量（导出和不导出的）

—implementation 部分的不保留单元变量（导出的）

在这些段中，变量根据声明的顺序排列

●高于 SIMOTION kernel 版本 V3.1

Table 5-27 Example for the control of the HMI export with the corresponding pragma

```
INTERFACE
    VAR_GLOBAL
        // HMI export
        x1 : DINT;
    END_VAR
    VAR_GLOBAL
        { HMI_Export := FALSE; }
        // No HMI export
        x2 : DINT;
    END_VAR
    // ...
END_INTERFACE

IMPLEMENTATION
    VAR_GLOBAL
        // No HMI export
        y1 : DINT;
    END_VAR
    VAR_GLOBAL
        { HMI_Export := TRUE; }
        // HMI export
        y2 : DINT;
    END_VAR
    // ...
END_IMPLEMENTATION
```

3.3 访问输入和输出（过程图像，　I/O 变量）

3.3.1　访问输入和输出的概述

SIMOTION提供访问 SIMOTION设备输入和输出的多种可能性，中央和分散　I/O 也是入如此：

●通过直接访问 I/O 变量

使用直接访问来访问对应的　I/O 地址。

不指定任务来定义 I/O 变量。SIMOTION 设备的整个地址空间都是可用使用的。

使用带顺序编程（在 SIMOTION 任务中）的直接访问：在特定的时间访问现行输入和输出值尤为重要。

更多信息，见 214 页直接访问和循环任务的过程图像

●使用 I/O 变量的循环任务的过程图像

循环任务的过程图像是 SIMOTION 设备 RAM 的存储区域，SIMOTION 设备的全部 I/O 地址空间反射到这个区域。 每个 I/O 地址的镜像会被指定一个循环任务，　然后使用此任务更新。在整个周期中任务保持一致。当编程指定任务（循环编程）时会优先使用过程图像。

定义一个 I/O 变量（名称和 I/O 地址），然后给其指定一个任务。 SIMOTION 设备的全部地址区域都是可使用的。

直接访问 I/O 变量也是可能的：直接用 _direct.varname 指定直接访问。

更多信息见 214 页直接访问和循环任务的过程图像

●使用背景任务的固定过程图像

背景任务的过程图像是 SIMOTION 设备 RAM 的存储区域，SIMOTION 设备的子集 I/O 地址空间反射到这个区域。每个镜像会随着背景任务一起刷新，并在整个周期中任务保持一致。当编程背景任务（循环编程）时会优先使用过程图像。

可以使用地址空间 0...63 。执行使用循环任务的过程图像访问　I/O 地址。

更多信息见 220 页访问背景任务的过程图像

最重要的属性的比较包含在　212 页直接访问和过程图像的属性。

你可以像其他变量一样使用　I/O 变量，见 226 页访问 I/O 变量

注意：

通过过程图像访问比直接访问更有效。

3.3.2 直接访问和过程图像访问的重要特征

	直接访问	循环任务过程图像的访问	背景任务的固定过程图像的访问
允许的地址范围	SIMOTION设备的全部地址 例外：组成大于一个字节的 I/O 变量必须不能连续包含地址 63 和 64（如：不允许 PIW63 或 PQD62） 使用的地址必须显示在 I/O，并合理配置。		0...63 除了在循环任务的过程图像中使用的地址。 不显示在 I/O 的地址和没有组态的地址也是可以使用的
指定的任务	无	供选择的循环任务： ●SynchronousTasks, ●TimerInterruptTasks, ●BackgroundTask.	●BackgroundTask.
更新	● SIMOTION 设备 C230-2，C240，和 P350 Onboard I/O：在 125 μs 时钟脉冲周期更新 ●D4xx SIMOTION 设备的 PROFIBUS DP，PROFINET，P-Bus，DRIVE-CLiQ Onboard I/O：在位置控制的时钟脉冲周期更新	更新随着指定任务发生 ●在指定任务开始前读取输入，然后传给过程输出图像 ●在完成指定任务之后，把过程输出图像写给输出。	随着背景任务更新 ●在背景任务开始前读取输入， 然后传给过程输出图像 ●在完成背景任务之后，把过程输出图像写给输出
一致性	—	在指定任务的整个周期 例外：直接访问输出	在背景任务的整个周期 例外： 直接访问输出
	仅基础数据类型的一致性能确保。 当使用阵列时， 用户要负责确保数据一致性		
使用	在 MOTION TASK中优先	在指定任务中优先	在背景任务中优先
使用绝对地址	不支持		可能，使用下列语法 ，如%IW62, %Q63.3
作为变量声明	必须，在符号浏览器中整个设备作为一个 I/O 变量		可能， 但是不是必要的： ●在符号浏览器中整个设备作为一个 I/O 变量 ●作为一个单元变量

			●在程序中作为一个本地变量
给输出写保护	可能，可以选择只读状态	不支持	不支持
阵列说明	可能		不支持
更多信息	见 214 页直接访问和循环任务的过程图像		见 220 页访问背景任务的固定过程图像
对错误的反应	从用户程序访问出错时，替代反应可用： ●CPU 停止 [1] ●代替数值 ●最终值	在生成过程图像时出错，替代反应可用： ●CPU 停止 [2] ●代替数值 ●最终值	在生成过程图像时出错，替代反应可用： ●CPU 停止 [2] 例外：如果在相同的地址创建一个直接访问，应用此处的动作
	参见 SIMOTION基础功能描述		
访问			
●RUN 模式	无限制	无限制	无限制
●启动任务时	可能限制： ●可以读取输入 ●直到完成启动任务才写输出	可能限制： ●在启动任务时最先读取输入 ●直到完成启动任务才写输出	可能限制： ●在启动任务时最先读取输入 ●直到完成启动任务才写输出
●关闭任务时	无限制	可能限制： ●输入保持最近更新的状态 ●不再写输出	可能限制： ●输入保持最近更新的状态 ●不再写输出
1调用 ExecutionFaultTask. 2.调用 PeripheralFaultTask			

3.3.3 直接访问和循环任务的过程图像

属性：

直接访问输入和输出、访问循环任务的过程图像通常是通过 I/O 变量实现的。 SIMOTION 设备的所有地址范围（见下表）都是可用的。

最重要的属性的比较包含在 220 页访问背景任务的过程图像和 212 页直接访问和过程图像的属性

直接访问：

使用直接访问来访问对应的 I/O 地址。首先使用直接访问顺序编程（在 SIMOTION TASK 中）。在特定的时间访问输入和输出的现行值是尤为重要的。

对于直接访问，不用指定一个任务即可定义　　I/O 变量（217 页）

注意：通过过程图像的访问比直接访问更有效。

循环任务的过程图像
循环任务的过程图像是　SIMOTION　设备 RAM 的存储区域，　SIMOTION　设备的全部　I/O 地址空间反射到这个区域。　每个 I/O 地址的镜像会被指定一个循环任务，　　然后使用此任务更新。在整个周期中任务保持一致。当编程指定任务（循环编程）时会优先使用过程图像。在任务的完整周期中保持一致性是尤为重要的。

对于循环任务的过程图像，你可以定义一个　　I/O 变量，然后指定一个任务。

直接访问 I/O 变量也是可能的：直接用　_direct.var-name　指定直接访问。

SIMOTION设备的地址范围
基于 SIMOTION kernel版本的 SIMOTION设备的地址范围在下表中显示。可以使用完整的地址范围来直接访问和循环任务的过程图像。

SIMOTION device	Address range for SIMOTION Kernel version		
	V3.0	V3.1, V3.2	As of V4.0
C230-2	0 .. 1023	0 .. 2047 [3]	0 .. 2047 [3]
C240	–	–	0 .. 4096 [3]
D410 [1]	–	–	0 .. 16383 [3 4]
D425 [2]	–	0 .. 4095 [3]	0 .. 16383 [3 4]
D435	0 .. 1023	0 .. 4095 [3]	0 .. 16383 [3 4]
D445 [2]	–	0 .. 4095 [3]	0 .. 16383 [3 4]
P350	0 .. 1023	0 .. 2047 [3]	0 .. 4095 [3]

[1] Available as of V4.1.

[2] Available as of V3.2.

[34] For distributed I/O (over PROFIBUS DP), the transmission volume is restricted to 1024 bytes per PROFIBUS DP line.

[5] For distributed I/O (over PROFINET), the transmission volume is restricted to 4096 bytes per PROFINET segment.

注意：见 216 页直接访问和循环任务的过程图像的　　I/O 地址

3.3.3.1　直接访问和循环任务的过程图像的　I/O 地址的规则

注意：

在检查 SIMOTION项目（如：下载时）的一致性时检查规则的符合性

1.用于 I/O 变量的地址必须显示在 I/O 中，然后在 HW Config中合理配置

2. 组成大于一个字节的 I/O 变量必须不能连续包含地址 63 和 64

下列 I/O 地址是不允许的：

—输入： PIW63, PID61, PID62, PID63

—输出： PQW63, PQD61, PQD62, PQD63

3. 组成大于一个字节的 I/O 变量地址必须在用 HW-config 配置的地址区域

4.通过单独 I/O 变量的数据类型 BYTE,WORD 或 DWORD，或这些数据类型的阵列来使用一个 I/O 地址（输入或输出）。通过 BOOL数据类型的 I/O 变量访问各自的比特也是可能的。

5.如果多个过程（如： I/O 变量，技术目标， PROFIdrive telegram）访问一个 I/O 变量，应用下列：

—只有一个单独的过程可以写访问到一个输出的（）的 I/O 地址

使用读访问到一个 I/O 变量的输出时可能的

—所有过程必须使用同样的数据类型 (BYTE, WORD, DWORD 或 这些数据类型的阵列)来访问 I/O 地址。访问各自的比特也是可能的。

比如：如果你想使用 I/O 变量来读取转移的驱动 PROFIBUS电报：I/O 变量的长度必须与电报的长度匹配。

—从多个过程写访问到一个地址的不同位是可能的，但是，访问带 BYTE, WORD 或 DWORD的数据类型是不可能的。

注意：

这些规则不适用于访问背景任务的固定过程图像。在项目（如：下载时）一致性检查时不需要考虑这些访问。

3.3.3.2 为直接访问和循环任务的过程图像创建一个 I/O 变量

在符号浏览器的细节部分创建一个 I/O 变量，为此，你必须使用离线模式：

下列是这个规程的简要描述

1.在 SIMOTION SCOUT的项目导航中，在 SIMOTION设备子目录中选择 I/O 元素

2.在细节浏览中，选择符号浏览器标签，然后滚动到最末的变量表（空行）

3.在表格的最后一行，输入或选择以下：

—变量名称

—根据输入 I/O 地址的语法图的 I/O 地址（ 219 页）

—输出的选择：

如果你只要读输入到输出，激活只读检查框

然后你可以读取正在由其他过程（一个输出 CAM 的输出， PROFIdrive电报）写的输出

一个只读的输出变量不能被指定给循环任务的过程图像

—变量的数据类型符合 I/O 变量可能的数据类型（ 220 页

4.或者你可以选择性地输入或选择以下：（不指 BOOL数据类型）

—阵列长度（阵列大小）

—过程图像或直接访问

只有清除了只读检查框才能被指定
对于过程图像， 选择你要指定的 I/O 变量的循环任务。 要选择任务， 必须在执行系统中已经激活此任务。

对于直接访问，选择空白输入
—错误状态的行为策略（见 SIMOTION基础功能手册）
—代替数值（如果阵列，每个元素）
—显示格式（如果阵列，每个元素），当你在符号浏览器监视变量时

现在你可以使用符号浏览器或 SIMOTION设备的任意程序来访问变量。

注意：
循环任务的过程图像必须注意一以下：
●一个变量只能被指定给一个任务
●一个输出或输出的每个字节只能被指定给一个 I/O 变量

在 BOOL数据类型中，请注意：
●不能定义循环任务的过程图像和错误的策略。通过一个 I/O 变量定义的全部字节的动作是适用的。（默认：直接访问或 CPU 停止）
●一个 I/O 变量独自的比特可以通过比特访问功能来访问

当改变 I/O 变量（插入和删除 I/O 变量，改变名称和地址）时，注意：
●其他 I/O 变量的内部地址可能改变，使得所有 I/O 变量不一致
●如果发生这种情况，所有包含访问 I/O 变量的所有程序源必须重新编辑

注意：
只能在离线模式创建 I/O 变量。在 SIMOTION SCOUT中创建 I/O 变量，然后在程序源 （如：ST源， MCC 源， LAD/FBD源）中使用。

可以读写输出，但是输入为只读。

在你可以监视和修改新的或更新的 I/O 变量前，你必须下载项目到目标系统。

你可以像使用其他变量一样使用 I/O 变量，见 226 页访问 I/O 变量。

3.3.3.3 输入 I/O 地址的语法图

对于直接访问或循环任务的过程图像 I/O 变量定义的 I/O 地址，使用下列语法图。这不仅说明了地址，也说明了访问的数据类型和访问模式（输入 /输出）。

Data type	Syntax for		Permissible address range					
	Input	Output	Direct access		Process image		e.g. direct access D435 V4.1	
BOOL	PIn.x	PQn.x	n: x:	0 .. *MaxAddr* 0 .. 7		-[1]	n: x:	0 .. 16383 0 .. 7
BYTE	PIBn	PQBn	n:	0 .. *MaxAddr*	n:	0 .. *MaxAddr*	n:	0 .. 16383
WORD	PIWn	PQWn	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. 16382
DWORD	PIDn	PQDn	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64 .. 16380

n = logical address
x = bit number

MaxAddr =	Maximum I/O address of the SIMOTION device depending on the version of the SIMOTION kernel, see address range of the SIMOTION devices in "direct access and process image of the cyclical tasks (Page 214)".

[1] For data type BOOL, it is not possible to define the process image for cyclic tasks. The behavior defined via an I/O variable for the entire byte is applicable (default: direct access).

例如：

逻辑地址 1022 输入， WORD数据类型：PIW1022

逻辑地址 63 输出， bit 3,BOOL数据类型：PQ63.3

注意：查看 216 页直接访问和循环任务的过程图像的 I/O 地址的规则

3.3.3.4 可能的 I/O 变量的数据类型

下列数据类型可以指定给 I/O 变量，以直接访问或循环任务的过程图像（ 214 页）。数据类型的宽度必须与 I/O 地址的数据类型宽度相符合。

如果你给 I/O 变量指定一个数字型数据类型，你可以以整数访问这些变量

I/O 地址数据类型	I/O 变量可能的数据类型
BOOL (PIn.x, PQn.x)	BOOL
BYTE (PIBn, PQBn)	BYTE, SINT, USINT
WORD (PIWn, PQWn)	WORD, INT, UINT
DWORD (PIDn, PQDn)	DWORD, DINT, UDINT

3.3.4 背景任务的固定过程图像的访问

背景任务的过程图像是 SIMOTION 设备 RAM 的存储区域， SIMOTION 设备的子集 I/O 地址空间反射到这个区域。每个镜像会随着背景任务一起刷新，并在整个周期中任务保持一致。当编程背景任务（循环编程）时会优先使用过程图像。

背景任务的固定过程图像的大小为　64byte （地址范围　0...63 ）

最重要的属性与直接访问和循环任务过程图像的的比较包含在　212 页直接访问和过程图像的属性

注意：不能使用访问循环任务的过程图像的　I/O 地址。这些地址不能读写给背景任务的固定过程图像。

规则：
不适用 216 页直接访问和循环任务的过程图像的　I/O 地址。项目（如：下载时）一致性检查时不需要考虑这些到背景任务的固定过程图像的访问。

不显示在 I/O 中或不在 HW config 配置的的址视为普通的存储地址。

可以通过下列方法访问背景任务的固定过程图像：
●使用一个绝对　PI 访问（ 221 页）：绝对　PI 地址访问包含输入 / 输出地址和数量类型的标识符
●使用一个符号　PI 访问（ 223 页）：声明一个变量，以供相关的绝对　PI 访问参考
—一个单元变量
—在程序中的一个静态本地变量
●使用一个 I/O 变量 (225 页)：在符号浏览器中，你为整个设备定义一个有效的　I/O 变量，以供相关的绝对　PI 访问参考

注意：
观察如果输入和输出以　Little Endian 字节顺序工作 （如：SIMOTION设备 C230-2 or C240集成数字输入），又满足下列条件：

1.输入和输出在地址　0...62 间配置
2.创建直接访问（数据类型　WORD, INT 或 UINT）的 I/O 变量以输入和输出
3.也可以通过背景任务的固定过程图像访问这些输入和输出

下列内容是有效的：
●数据类型 WORD 访问通过 I/O 变量和背景任务的固定过程图像提供相同的结果
●带 _getInOutByte function 单独字节访问在　Little Endian 顺序中提供这些
●通过背景任务的固定过程图像访问独自的字节或位在　Big EndiaN顺序中提供这些

3.3.4.1 背景任务的固定过程图像的绝对访问（绝对　PI 访问）

通过直接使用地址（带隐式数据类型）标识符来绝对访问背景任务的固定过程图像。标识符（ 222 页）的语法图描述在下面章节中。

你可以如同正常变量一样使用绝对　PI 访问的标识符。

注意：输出可以读写，但是输入为只读。

3.3.4.2 一个绝对过程图像访问的标识符语法

绝对访问背景任务的固定过程图像，使用下语法图。不仅说明地址，还要说明访问的数据类型和访问模式（输入 /输出）。

可以使用下列标识符：

●对于背景任务的固定过程图像的符号访问的声明（见 223 页）

●对于创建 I/O 变量以访问背景任务的固定过程图像（见 225 页）

Data type	Syntax for		Permissible address range	
	Input	Output		
BOOL	%In.x or %IXn.x[1]	%Qn.x or %QXn.x[1]	n: x:	0 .. 63 [2] 0 .. 7
BYTE	%IBn	%QBn	n:	0 .. 63 [2]
WORD	%IWn	%QWn	n:	0 .. 63 [2]
DWORD	%IDn	%QDn	n:	0 .. 63 [2]

n = logical address

x = bit number

[1] The syntax %IXn.x or %QXn.x is not permitted when defining I/O variables.

[2] Except for the addresses used in the process image of the cyclic tasks.

逻辑地址 62 输入， WORD数据类型：%IW62

逻辑地址 63 输出， bit 3,BOOL数据类型：%Q63.3

注意：不能使用访问循环任务的过程图像的访问地址。这些地址不能读写给背景任务的固定过程图像。

注意：

不适用 216 页直接访问和循环任务的过程图像的 I/O 地址。项目（如：下载时）一致性检查时不需要考虑这些到背景任务的固定过程图像的访问。

不显示在 I/O 中或不在 HW config 配置的的址视为普通的存储地址。

Table 5-33 Examples of absolute CPU memory access

```
status1 := %I1.1; // BOOL data type
status2 := %IB10; // BYTE data type
status3 := %IW20; // WORD data type
status4 := %ID20; // DWORD data type

%Q1.1 := status1; // BOOL data type
%QB20 := status2; // BYTE data type
%QW20 := status3; // WORD data type
%QD20 := status4; // DWORD data type
```

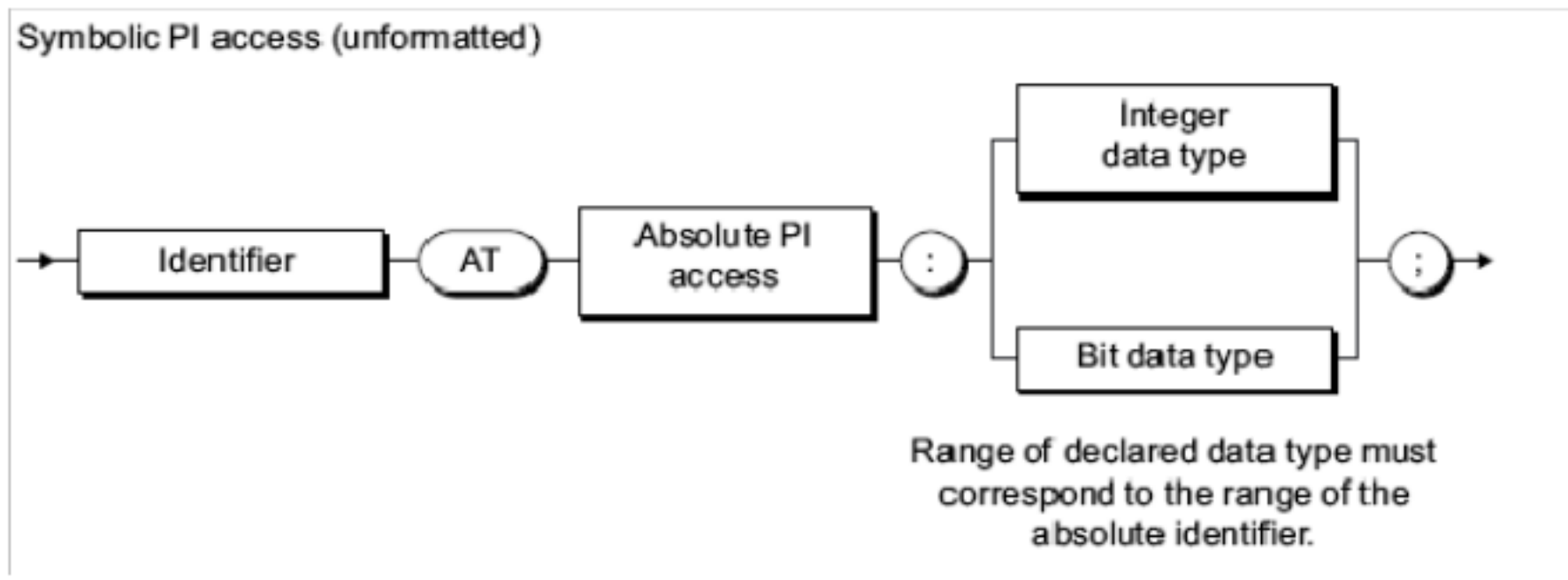
3.3.4.3 背景任务的固定过程图像的符号访问（符号 PI 访问）

你可以符号访问背景任务的固定过程图像，不需要经常指定绝对过程图像访问。

你可以声明符号访问：

- 作为一个程序的一个静态变量（在声明部分 VAR/END_VAR 的结构中）
- 作为一个单元变量（在 ST 源文件的 interface 或 implementation 部分的 VAR_GLOBAL/END_VAR 结构中）

PI 访问的符号名称声明语法图如下：



关于绝对 PI 访问，见 222 页绝对 PI 访问的标识符语法图。

声明的整数或 bit 数据类型的范围必须与绝对 PI 访问的范围相符合。 在声明一个数字型数据类型之后，你可以把过程图像的内容作为一个整数写地址。

见 224 页声明的例子

3.3.4.4 可能的符号 PI 访问的数据类型

与绝对 PI 访问不同的数据类型可以被指定为背景任务的固定过程图像。数据类型的宽度必须与绝对 PI访问的数据类型的宽度相对应。

- 符号 PI访问的声明（ 223 页）
- 创建一个 I/O 变量（ 225 页）

如果你给 PI 访问或 I/O 变量指定一个数字型数据类型，你可以作为整数访问这些变量。

Data type of the absolute PI access	Possible data types of the symbolic PI access
BOOL (%In.x, %IXn.x, %Qn.x. %QXn.x)	BOOL
BYTE (%IBn, %QBn)	BYTE, SINT, USINT
WORD (%IWn, %QWn)	WORD, INT, UINT
DWORD (%IDn, %PQDn)	DWORD, DINT, UDINT

3.3.4.5 符号 PI 访问的例子

如果你要访问 CPU存储区域（ 222 页绝对 PI 访问）%IB10，但是你可以灵活的在程序中改动，然后如下在 CPU存储区域中声明一个 myInput 变量：

```
VAR
    myInput AT %IB10 : BYTE;
END_VAR
```

如果你想使用存储区域的整数数值，如下声明 myInput 变量：

```
VAR
    myInput AT %IB10 : BYTE;
END_VAR
```

如果你想在程序中使用 CPU 存储区域，而不是 %IB10，你只需要在变量声明中更改绝对 PI访问。

3.3.4.6 为访问背景任务固定过程图像而创建一个 I/O 变量

在符号浏览器的细节窗口创建 I/O 变量，为此，必须在离线模式下进行。

下列是这个规程的简要描述

1.在 SIMOTION SCOUT的项目导航中，在 SIMOTION设备子目录中选择 I/O 元素

2.在细节浏览中，选择符号浏览器标签，然后滚动到最末的变量表（空行）
3.在表格的最后一行，输入或选择以下：
—变量名称
—在 I/O 地址下，根据“ 一个绝对 PI 访问标识符的语法图（ 222 页）”的绝对 PI 访问
（例外 :不允许 BOOL数据类型的语法 %IXn.x or %QXn.x）
—根据“ 224 页可能的符号 PI 访问的数据类型 ”的 I/O 变量的数据类型
4.选择使用的显示格式来监视符号浏览器中的变量

现在你可以使用符号浏览器或 SIMOTION设备的其他程序来访问变量

注意：
只能在离线模式下创建 I/O 变量。在 SIMOTION SCOUT中创建 I/O 变量，然后在程序源中使用。

注意你可以读写输出，但是输入是只读。

在你可以监视和修改新的或更新的 I/O 变量之前，你必须下载项目到目标系统。

可以如其他变量一样使用 I/O 变量，见 226 页访问 I/O 变量。

3.3.5 访问 I/O 变量

你创建一个 I/O 变量以：
●直接访问或循环任务的过程图像（ 214 页）
●背景任务的固定过程图像的访问（ 220 页）。可以如其他变量一样使用 I/O 变量。

注意：
如果你把单元变量或本地变量声明为同样的名称 (如： var-name)，使用 _device.var-name（预先定义的命名空间，见命名空间中的预先定义的命名空间表格）指定 I/O 变量。

访问作为循环任务的过程图像所创建的一个 I/O 变量是可能的。用 _direct.var-name 或 _device._direct.var-name 指定直接访问。

在变量访问中发生错误时， 如果你想从默认的动作中 deviate，你可以使用 _getSafeValue 和 _setSafeValue功能 (见 SIMOTION功能手册）。

对于访问 I/O 变量的相关错误，见 SIMOTION基础功能手册。

3.4 使用库

库为你提供了用户定义的数据类型、 功能和功能块，这些都可以在 SIMOTION设备中使用。

库可以用所有编程语言编写，也可以在所有程序源（ ST源文件， MCC单元）中使用。

在线帮助中可以通过插入和管理库获得更多细节信息。

注意：
关于程序源文件的名称的规则同样适用于库的名称，见 21 页插入 ST源文件。特别是，名称允许的长度基于 SIMOTION kernel版本
●SIMOTION kernelV4.1 ： 最大 128 个字符
●SIMOTION kernelV4.0 ： 最大 8 个字符

高于 V4.0 的 SIMOTION kernel版本，直到一致性检查或执行项目的下载时才能检测超出库的允许长度。

3.4.1 编辑一个库

在库中，你可以使用所有的 ST命令，除了在下表中列出的几点。下表中列出的有些变量也不允许访问。

禁止的命令：
●_getTaskId 功能（见 SIMOTION 基础功能手册）
●_getAlarmId 功能（见 SIMOTION 基础功能手册）
●_checkEqualTask 功能（见 SIMOTION 基础功能手册）
●下列功能时针对高于 SIMOTION kernelV3.0 的版本
—任务控制命令
—任务运行时间测量命令
—信息编程的命令
有了这些功能，配置的信息的任务名称被转移
●如果库不是基于设备的 （如：不参考 SIMOTION 设备或 SIMOTION kernel 版本的编辑）
—SIMOTION 设备的系统功能（ SIMOTION 设备的参数手册）
—基于版本的系统功能

禁止的变量访问：
●单元变量（保留和不保留）
●全局设备变量（保留和不保留）
●I/O 变量
●技术目标的实例和系统变量
●任务名称和配置信息的变量 (_task 和 _alarm namespaces ）
●如果库不是基于设备的 （如：不参考 SIMOTION 设备或 SIMOTION kernel 版本的编辑）
—SIMOTION 设备的系统变量（ SIMOTION 设备的参数手册）
—技术目标的配置数据（见 SIMOTION 技术包配置数据的参数手册）

注意：在库中程序状态 debug 功能是不可用的。

编辑一个独自的库

编辑一个独自的库的规程如下：

1.在项目导航中选择库

2.选择编辑 >目标属性菜单命令

3.选择 TPs>TOs 标签

4.选择能作为编辑库的基础使用的 SIMOTION设备（在 SIMOTION kernel版本）和技术包，见 SIMOTION基础功能手册

5.从文本菜单中选择接受和编辑

参考所有选择的 SIMOTION设备、版本和技术包（和独立设备）编辑库

注意：

如果编辑库导入另一个库，注意以下：

1.对于导入的库，至少相同设备和 SIMOTION kernel版本必须作为导入库选择。如果满足预先条件（参考 SIMOTION基础功能手册），可以编辑独立设备为导入库

2.导入库必须参考所有的组态设备， kernel 版本和技术包各自编译。作为项目 -范围编辑的部分的库的编辑通常不是很有效。

作为项目 -wide 编辑的部分的库的编辑

当你编辑整个 SIMOTION项目时（例如：通过选择项目 >保存和重新编辑所有或通过一个 XML 导入），使用的库页同样被编辑。

注意：

当执行项目—范围编辑时，注意以下：

1.系统自动识别库之间的相关性，选择合适的编辑顺序

2.编辑一个库只能参考在项目中配置的和使用库的 SIMOTION设备（包括 SIMOTION kernel 版本）

3.忽略为库设置的其他 SIMOTION设备和 kernel 版本

3.4.2 库的 know-how 保护

你可以保护库和源文件免受第三方未授权的访问。保护库或源只能通过输入密码以 纯文本打开或导出。

你可以：

●给一个库的源提供 KNOW-HOW保护

只有源有未授权访问保护

SIMOTION设备的设置包括 SIMOTION kernel 和所有技术包，库在这些地方进行编辑。用户也可以更改和接受。请参见 SIMOTION基础功能手册。

用户可以为 SIMOTION其他设备和 kernel 版本使用库

●给库提供 KNOW-HOW 保护

下列有未授权访问保护：
—库的所有源
—SIMOTION设备的设置包括 SIMOTION kernel和所有技术包，库在这些地方进行编辑

这样就阻止了用户为 SIMOTION其他设备和 kernel 版本使用库。

如果要想达到此目标，使用此设置。

注意：如果以 XML 格式导出，库或源以加密方式导出。当导入编码的 XML 文件时，包括登陆和密码的 KNOW-HOW保护原地保持可用。

3.4.3 从库中使用数据类型，功能和功能块

在从库中使用数据类型， 功能和功能块时， 你必须使这些对 ST源文件已知。 为此 ,, 在 ST 源文件 interface 部分使用下列结构

USELIB library-name [AS namespace];

库名为显示在项目导航中的名称。 当需要说明多个库时， 用逗号隔开作为一个列表输入，如：

```
USELIB library-name_1 [AS namespace_1],
       library-name_2  [AS namespace_2],
       library-name_3  [AS namespace_1],  ...
```

你可以使用选项 AS 命名空间 add-on 来定义一个命名空间（见 233 页）
●在有和 SIMTION设备的 ST源文件相同名称的库中你可以访问数据类型、功能和功能块
●你可以在库中使用命名空间来改变数据类型、功能和功能块的名称，以便他们名称不同

你也可以给不同的库指定相同的命名空间。

Table 5-36 Example of use of namespaces with libraries

```
INTERFACE
    USELIB Bib_1 AS NS_1, Bib_2 AS NS_2;
    PROGRAM Main_Program;
END_INTERFACE

IMPLEMENTATION
    FUNCTION Function1 : VOID
        VAR
            ComID : CommandIdType;
        END_VAR
        ComId := _getCommandId();
    END_FUNCTION

    PROGRAM Main_program
        function1();          // Function from this source
        NS_1.Var1:=1;
        NS_2.Var1:=2;
        NS_1.function1();     // Function from the Bib1 library
        NS_2.function1();     // Function from the Bib2 library
    END_PROGRAM
END_IMPLEMENTATION
```

3.5 相同的标识符和命名空间的使用

3.5.1 相同的标识符的使用

使用相同名称的单元变量和本地变量（程序变量、 FB变量、 FC变量）是可以的。当编辑一个程序源时，编辑器搜寻带现行 POU的标识符。小的有效范围优先于大的有效范围。

因此你可以在不同的源文件部分使用相同的标识符，只要遵循下列规则。如果一个高级别的标识符在一个单元或 POE中被隐藏，编辑器产生一个报警。

注意：
在确定的情况下，如果相关的技术包没有被导入，编辑器不会产生报警。

在一个程序组织单元（ POU）中的标识符
在一个 POU 中下列标识符必须是唯一的：

- POU的本地变量
- POU的本地数据类型

下列标识符可能不一样

- 预留标识符

当隐藏下列标识符时编辑器产生报警

●单元变量，数据类型和 POU 或相同的或导入的单元

●标准系统功能，标准系统功能块，和相关的数据类型

●SIMOTION 设备系统功能和系统数据类型

●从导入的库中的 POU 和数据类型

—可以通过输入一个用户定义的命名空间解决

●从导入的技术包中的系统功能和系统数据类型

—可以通过输入一个用户定义的命名空间解决

●SIMOTION 设备变量（系统变量， I/O 变量，全局设备变量）

—可以通过输入一个预先定义的命名空间 -device 解决

●在 SIMOTION 设备中配置的技术目标

—可以通过输入一个预先定义的命名空间 _to 解决

在一个单元中的标识符

在一个单元中下列标识符必须是唯一的：

●单元变量（在 interface 或 implementation 部分声明的）

●数据类型（在 interface 或 implementation 部分声明的）

●POU

下列标识符不能一致

●预留标识符

●单元变量，数据类型和 POU 导入单元

●标准系统功能，标准系统功能块和相关的数据类型

●SIMOTION 设备的系统功能，系统数据类型

●从导入的库中的 POU 和数据类型

—可以通过输入一个用户定义的命名空间解决

●从导入的技术包中的系统功能和系统数据类型

—可以通过输入一个用户定义的命名空间解决

当隐藏下列标识符时编辑器产生报警

●SIMOTION 设备变量（系统变量， I/O 变量，全局设备变量）

—可以通过输入一个预先定义的命名空间 -device 解决

●在 SIMOTION 设备中配置的技术目标

—可以通过输入一个预先定义的命名空间 _to 解决

在 SIMOTION设备的标识符（如： I/O 变量，全局设备变量）

在 SIMOTION设备中下列标识符必须是唯一的：

●全局设备变量

●SIMOTION 设备的系统变量

●SIMOTION 设备的系统功能和系统数据类型

下列标识符不能相同：

●预留标识符

●标准系统功能，标准系统功能块和相关的数据类型

下面的例子解释了这种现象。对于单元变量（大有效范围）和 FC 变量（小有效范围）的相同名称的使用，只有在功能中声明的变量在源文件部分是有效的。单元变量仅在没有声明的本地变量的 POU 中有效。见例子：

Table 5-37 Example of identifier validity

```
TYPE
    type_a   :   (enum1, enum2, enum3);
END_TYPE

VAR_GLOBAL
    var_a, var_b  : DINT;      // Unit variables
END_VAR

FUNCTION fc_1  : VOID
    VAR
        var_a  : type_a;       // Declaration hides UNIT variable //
        var_c  : DINT;         Local variable
    END_VAR
    // Permitted statements
    var_a  := enum2;           // Access to local variable
    var_b  := 100;             // Access to unit variable
    var_c  := -1;              // Access to local variable
    // Invalid statement
    // var_a  := 200;
END_FUNCTION

FUNCTION fc_2  : VOID
    VAR
        var_b  : type_a;       // Declaration hides UNIT variable //
        var_c  : type_a;       Local variable
    END_VAR
    // Permitted statements
    var_a  := -100;            // Access to unit variable
    var_b  := enum3;           // Access to local variable
    var_c  := enum1;           // Access to local variable
    // Invalid statement
    // var_b  := 200;
END_FUNCTION
```

3.5.2 命名空间

可以使用名称访问定义在程序源（在库中，技术包，和 SIMOTION设备上）之外的数据类型，单元变量，功能和功能块。

当编辑一个程序源时，编辑器搜寻带现行 POU 的标识符。在程序源中声明的数据类型、变量、功能或功能块隐藏在源之外定义的相同名称的标识符，见 231 页相同标识符的使用。为了访问这些隐藏的标识符，你在确定的例子中可以使用命名空间。

用户定义的命名空间
在库和技术包的导入结构中，你可以定义名称来库和技术包的搜寻数据类型、功能、功能块。

```
USELIB library-name 1   [AS lib namespace 1],
    library-name 2   [AS lib namespace 2],
    library-name 3   [AS lib namespace 1],   ...

USEPACKAGE tp-name 1   [AS tp namespace 1],
    tp-name 2   [AS tp namespace 2],
    tp-name 3   [AS tp namespace 1],   ...
```

也可以使用命名空间来在不同的库中统一名称。

如果你要从一个库中或一个技术包使用一个数据类型、功能或功能块，把命名空间标识符置于名称之前，用一个句号隔开，如：namespace.fc-name，namespace.fb-name，namespace.type-name

例子
下面的例子显示了如何选择 CAM 技术包，指定命名空间 CAM1，然后使用命名空间：

Table 5-38 Example of selecting a technology package and using a namespace

```
INTERFACE
    USEPACKAGE Cam AS Cam1;
    USES ST 2;
    FUNCTION function1;
END INTERFACE

IMPLEMENTATION
    FUNCTION function1   : VOID
    VAR INPUT
        p_Axis   : posAxis;
    END VAR
    VAR
        retVal   : DINT;
    END VAR

    retVal:= Cam1. enableAxis   (
            axis   := p_Axis,
            nextCommand   := Cam1.WHEN_COMMAND_DONE,
            commandId   :=   getCommandId()   );
    END FUNCTION
END IMPLEMENTATION
```

注意：如果为导入的库或技术包来定义一个命名空间或，必须被定义，如果从库或技术包使用功能、功能块或数据类型。见下例： Cam1._enableAxis, Cam1.WHEN_COMMAND_DONE.

预先定义的命名空间

device- 、 project-specific 变量、TaskID 和 AlarmID 变量的命名空间被预先定义。 如有必要，在变量名称前写指定， 用一个句号隔开， 例如：_device.var-name 或 _task.task-name

命名空间	描述
_alarm	对于 AlarmId：_alarm.name 变量包含带名称标识符的信息的 AlarmId（见 SIMOTION基础功能手册）
_device	对于 device-specific变量（SIMOTION设备全局设备变量， I/O 变量和系统变量）
_direct	对于直接访问 I/O 变量—见 214 页直接访问和循环任务的过程图像 本地名称为 _device。允许作为 in _device._direct.name 嵌套
_project	对于项目中 SIMOTION 设备的名称；只能和其他设备上的技术目标使用。 有唯一的技术目标的项目范围名称，使用为名称和系统变量
_task	对于 TaskID：_task.name 变量包含带名称标识符的任务的 TaskId
_to	对于在 SIMOTION设备上配置的技术目标，和系统变量和组态数据。 不对于系统功能和技术目标的数据类型。这时如有必要，为导入的技术包使用用户定义的命名空间

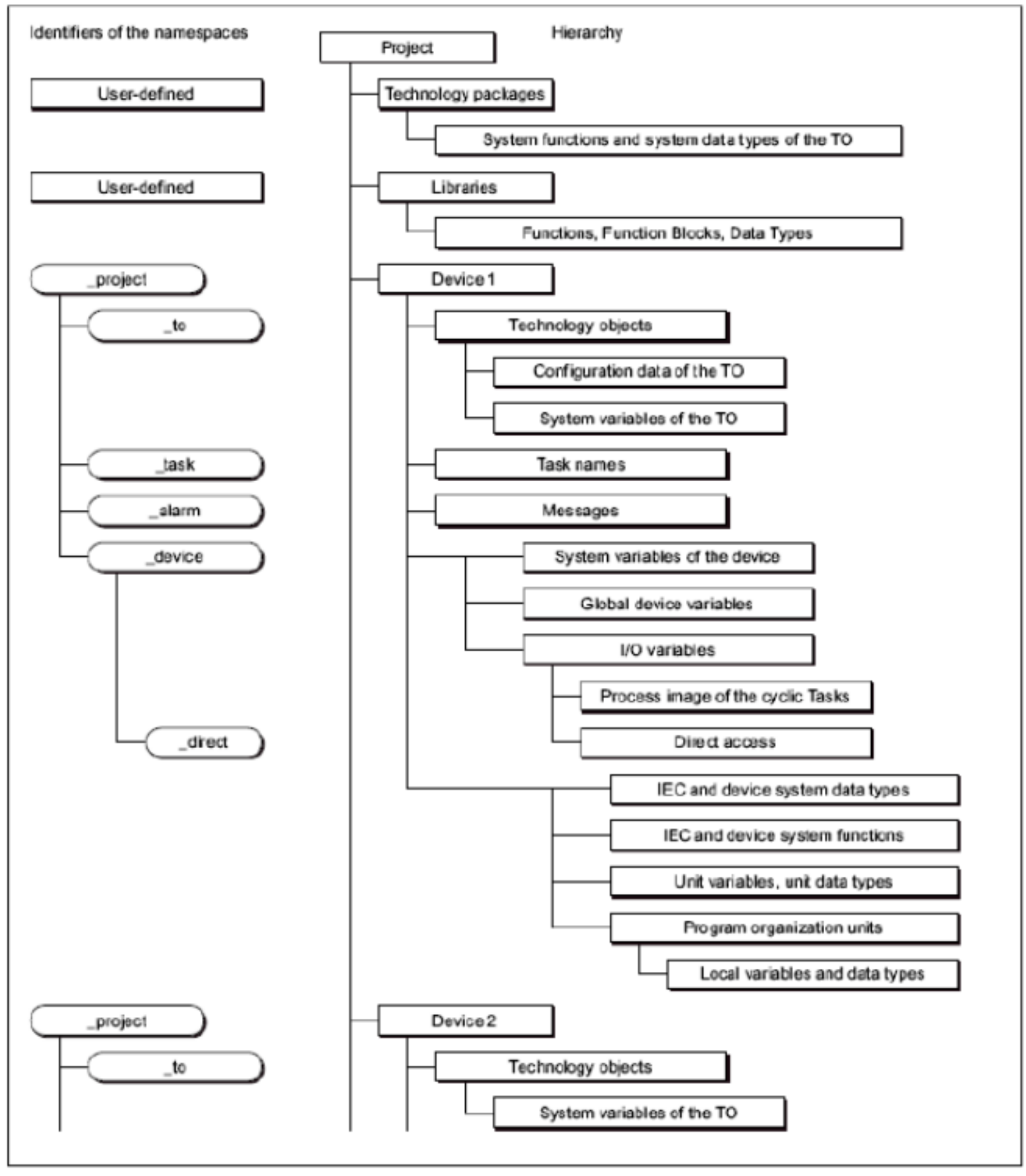


Figure 5-7 Namespaces and identifier hierarchy

3.6 参考数据

参考数据为你提供了一个概述：

- 使用声明中的信息的标识符和使用（交叉对照表）
- 功能调用和嵌套（程序结构）
- 程序源中不同数据区域的存储要求（代码属性）

见：

3.6.1 交叉对照表

交叉对照表列出了程序源（如： ST源文件， MCC源文件）中所有标识符：
●作为变量、数据类型或程序组织单元（程序，功能，功能块）声明
●作为之前在声明中定义的数据标识符使用
●作为在一个程序组织单元中的语句部分中的变量使用

按照要求生成一个交叉对照表作为：
●一个独立的程序源（如： ST源文件， MCC源文件， LAD/FBD源）
●SIMOTION设备的所有程序源
●项目所有的程序源和库
●库（所有的库，单个的库）

3.6.1.1 创建一个交叉对照表单

为创建一个交叉对照表单：

1.在项目导航中，选择要创建交叉对照表的元素
2.选择菜单编辑 >参考数据 >创建
在细节窗口的标签页显示交叉对照表

3.6.1.2 交叉对照表的内容

每个标识符的交叉对照表包含：
●标识符名称（对于结构和枚举，独自的部件和元素）
●类型（如数据类型， POU类型）
●声明位置（如：程序源的名称，技术包的名称）
●标识符现行使用的信息：
—使用的类型（如： R=读访问， W=写访问，变量类型 =声明）
—程序源的路径细节（ SIMOTION设备，程序源的名称）
—在程序源中的区域（如： implementation 部分， POU名称）
—程序源的程序语言
—在 ST源文件的行号（或 MCC 图表中的块号，或在 LAD/FBD源中的参考号）

注意：
创建的交叉对照表自动保存，在项目导航选择合适的元素之后可以显示交叉对照表。为了显示交叉对照表，选择 编辑>参考数据 >显示> 交叉对照表菜单命令 .

当再次创建一个交叉对照表时，选择性地更新（对应在项目导航中的选择元素）。其他存在的交叉对照数据被保留和显示，如适用。

注意：
在 MCC编程中激活单步式监视

每个任务指定 2 个变量： TSI#dwuser_1 和 TSI#dwuser_2 ，可以读写。

当激活单步式监视时，如果至少一个 MCC 图表被指定给相关任务，编辑器使用这些变量来控制单步式监视。用户不能使用这些变量，因为这些内容被单步式监视覆写，可能会引起不良副作用。

3.6.1.3 交叉对照表的使用

在一个交叉对照表中，你可以：
●按照字母顺序分类列的内容
●设置过滤功能（通过快捷菜单，通过点击鼠标右键）
●按照顺序复制剪贴板的内容，粘贴到一个 EXCEL表格程序中
●打印内容
●打开参开的程序源和相关 ST命令（或 MCC或 LAD/FBD元素）行光标位置
—在交叉对照表中的对应行双击
—将光标移至交叉对照表中的对应行，然后点击进入应用按钮

更多信息见在线帮助

3.6.2 程序结构

在程序中，结构是所有的功能调用和在选择的元素内的嵌套。 当成功创建交叉对照表时，你可以选择性地显示程序结构为：
●一个独立的程序源（如： ST源文件， MCC源文件， LAD/FBD源）
●SIMOTION设备的所有程序源
●项目所有的程序源和库
●库（所有的库，单个的库，在一个库中的独自的程序源）

遵循以下步骤：
1.在项目导航中，选择你想显示的程序结构的元素
2.选择菜单编辑 >参考数据 >显示 > 程序结构
在细节窗口中，交叉对照表的标签被程序结构标签替换

3.6.2.1 程序结构的内容

树形结构出现，显示：

●作为各自的 base

—在程序源中声明的程序组织单元（程序，功能，功能块）

—使用的执行系统任务

关于输入的结构，见下表：

元素	描述
Base（声明的 POU或使用的任务）	用逗号隔开的列表 ●POU或任务的标识符 ●用 add-on 声明的 POU或任务所在的从程序源的标识符 ●最小和最大栈需求（在本地数据栈上的 POU 或任务），用 byte（最小和最大） ●最小和最大全局栈需求（在本地数据栈，包括调用的 POU 上的 POU或任务），用 byte（最小和最大）
参考 POU	用逗号隔开的列表 ●POU或任务的标识符 ●可选： POU声明的程序源 /技术包的标识符： Add-on (UNIT): 用户定义程序源 Add-on (LIB): 库 Add-on (TP): 从技术包来的系统功能 ●仅对于功能块：实例的标识符 ●仅对于功能块：实例声明的程序源的标识符 Add-on (UNIT): 用户定义程序源 Add-on (LIB): 库 ●POU调用的源的行（编译）；用 "|" 隔开行

3.6.3 代码属性

你可以找到更过关于程序源的不同数据区域存储要求的信息。

成功创建交叉对照表时，你可以选择性显示代码属性为：

●一个独立的程序源（如： ST源文件， MCC源文件， LAD/FBD源）

●项目所有的程序源和库

●库（所有的库，单个的库，在一个库中的独自的程序源）

遵循以下步骤：

1.在项目导航中，选择你想显示的代码属性

2.选择菜单编辑 >参考数据 >显示 > 代码属性菜单

在细节窗口中，交叉对照表的标签被代码属性标签替换

3.6.3.1 代码属性内容

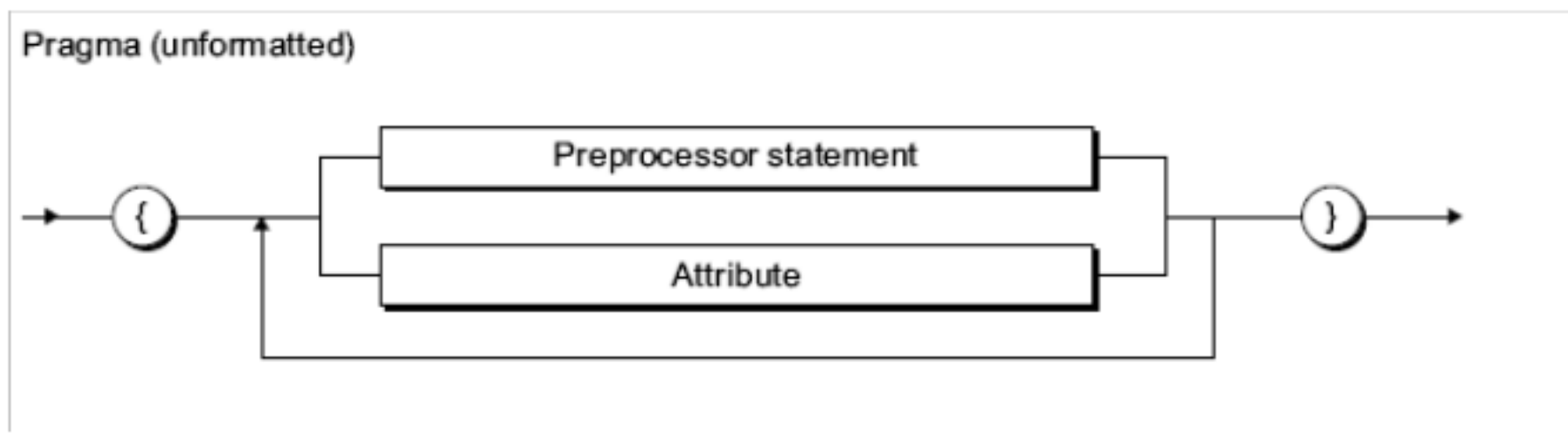
下列内容在一个表格中显示，作为所有选择的程序源文件

●程序源文件的标识符
—动态数据：所有单元变量（保留和不保留，在 interface 和 implementation 部分）
—保留数据：在 interface 和 implementation 部分的保留变量
—interface 数据： interface 部分的单元变量（保留和不保留）
●参考源的编号

3.7 控制预处理器和 pragma 编辑

使用一个 pragma 来插入 ST源文件文本（如，语句），将影响 ST源文件的编译。

pragma 用在{ }括号中，包含：
●用于控制预处理器的预处理器语句，见 243 页控制预处理器
带包含在 ST 源文件中预处理器语句的 pragma 通过预处理器计算，作为控制语句。
●控制编辑器的选项属性，见 247 页用属性控制编辑器
带包含在 ST 源文件中预处理器语句的 pragma 通过预处理器计算，作为控制语句。



注意：确认使用正确的 pragma 语法图（如：属性的大小写符号）。
忽略未识别的 pragma，不产生报警信息。

3.7.1 控制一个预处理器

预处理器为编辑准备一个 ST源。例如：字符串可以作为标识符的替代文本被定义，或者为编辑隐藏 / 显示源程序的部分。

默认预处理器是未启用的。可以如下激活：
●在项目中的程序源文件和编程语言为全局，见 45 页编辑器的全局设置

●程序源文件为本地，见 46 页本地编辑器设置

在一个程序源文件的编辑中，你将得知预处理器的动作。但是，这要求激活第 7 级的报警显示，见 49 页报警级别的意义。你为发生的报警和信息说明细节：

●在编辑器的全局或本地设置

●he _U7_PoeBld_编辑选项 :在 ST源文件中 = warning:n:off or warning:n:on 属性，见 247 页控制编辑器的属性。

如编辑器信息，预处理器的信息在编辑 / 检查输出标签的细节窗口显示。

注意

你可以查看由预处理器修改的 ST源文件的文本

1.打开 ST源文件

2.选择 ST源文件 >执行预处理器菜单命令

修改的源文本在编辑 / 检查输出标签的细节窗口显示。

3.7.1.1 预处理器语句

你可以通过 pragma 中的语句来控制预处理器。可以使用下图中的语句。这些语句作用于 ST源文件的后续行。

可以在 SIMOTION设备或一个库中的 ST源文件使用。

在 ST 源文件的属性对话框中给预处理器定义（见 51 页定义预处理器）。这使得你用 KNOW-HOW保护在 ST源文件中控制预处理器。（见 51 页 ST源 KNOW-HOE保护）

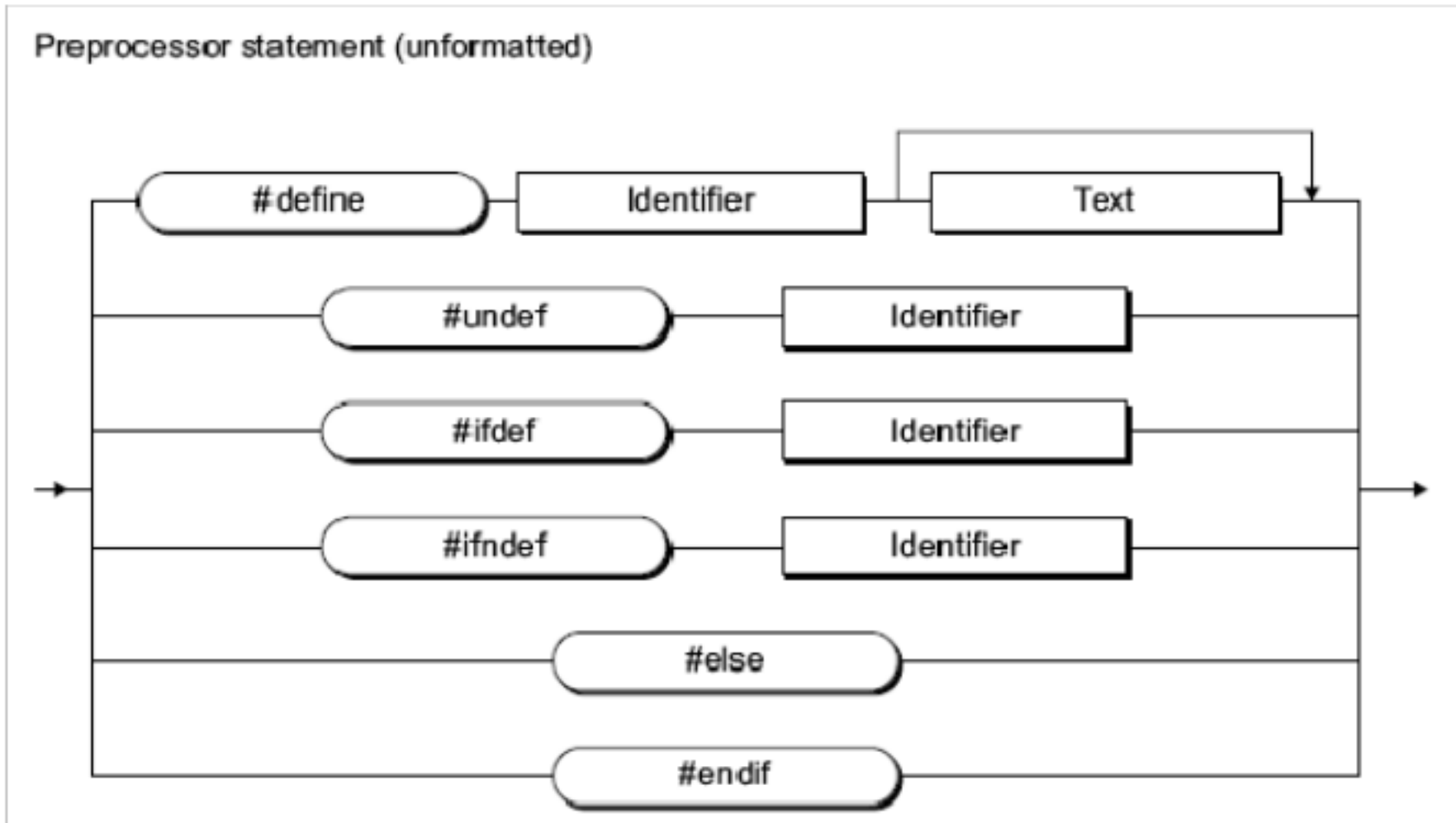


Figure 5-9 Syntax of a preprocessor statement

语句	意义
#define	指定的标识符将被下面指定的文本所代替。 允许的字符：见脚注表
#undef	删除了标识符的替换规则
#ifdef	对于不同的结构（条件编辑）： 如果定义了指定的标识符，编辑器编辑程序行（直到下个 pragma 包含 #else 或 #endif）
#ifndef	对于不同的结构（条件编辑）： 如果没有定义指定的标识符，编辑器编辑程序行（直到下个 pragma 包含#else 或 #endif）
#else	对于不同的结构（条件编辑）： 替代分支到 #ifdef 或 #ifndef 如果没有满足先前 #ifdef 或 #ifndef 查询，编辑器编辑程序行（直到下个 pragma 包含 #endif）
#endif	包含 #ifdef 或 #ifndef 不同的结构
允许的字符： ●标识符：与标识符的规则一致 ●文本：任意字符的顺序，而不是 \ (反斜杠)， 单引号) and ”(双引号). 不允许关键词 USES,	

USELIB和 USEPACKAGE

注意：
每个预处理器语句必须用一个新行开始，一个换行符结束。因此包括 pragma 的波形括号{}必须放在 ST源文件的单独行中！

在#定义语句中的 pragma，请注意：
●导出在 ST源文件的 interface 部分带 #定义语句的 pragma。用 USES语句导入定义标识符到 ST相同 SIMOTION设备或相同库的其他源文件
●在库的 pragma 中定义的标识符不恩能够被导入到 SIMOTION设备的 ST源文件
●预留标识符的重新定义是不可能的

你也可以在 ST 源文件的属性对话框中定义预处理器。相同标识符的不同定义，在 ST 源文件中的 #定义语句有优先权。

3.7.1.2 预处理器语句的例子

Table 5-42 Example of preprocessor statements

ST source file With preprocessor statements	Preprocessor output

INTERFACE
FUNCTION_BLOCK fb1;
VAR_GLOBAL
g_var : INT;
END_VAR
// Preprocessor definitions
{
#define my_define g_var
#define my_call f(my_define)
}
// my_define -> g_var
// my_call -> f(g_var)
END_INTERFACE
	INTERFACE
	FUNCTION_BLOCK fb1;
	VAR_GLOBAL
	g_var : INT;
	END_VAR
	{
	}
	END_INTERFACE

IMPLEMENTATION FUNCTION f : INT VAR_INPUT i : INT; END_VAR f := i; END_FUNCTION	IMPLEMENTATION FUNCTION f : INT VAR_INPUT i : INT; END_VAR f := i; END_FUNCTION

FUNCTION_BLOCK fb1
VAR_INPUT
i_var : INT;
END_VAR
VAR_OUTPUT
o_var : INT;
END_VAR
my_define := i_var;
// Delete the preprocessor definition
// For my_define
{
#undef my_define
}
	FUNCTION_BLOCK fb1
	VAR_INPUT
	i_var : INT;
	END_VAR
	VAR_OUTPUT
	o_var : INT;
	END_VAR
	g_var := i_var;
	{
	}

o_var := my_call + 1;
{
#ifdef my_define
}
	o_var := f(g_var) + 1;
	{

my_define := i_var;
{
#endif
}
	}

END_FUNCTION_BLOCK END_IMPLEMENTATION	END_FUNCTION_BLOCK END_IMPLEMENTATION

3.7.2 属性控制编辑器

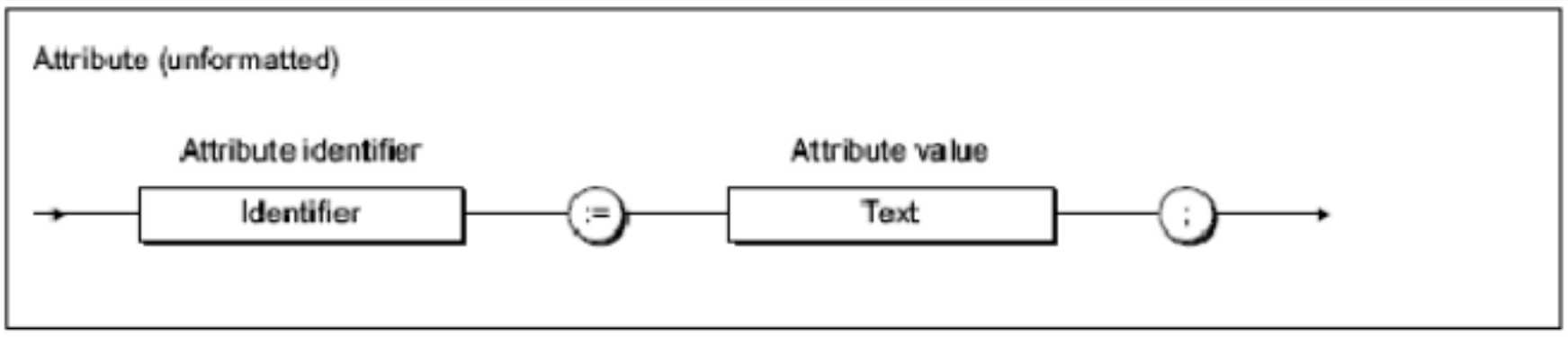


Figure 5-10 Syntax of an attribute for compiler options

属性标识符	属性值	意义
_U7_PoeBld_编辑选项	属性影响一个 ST源文件中编辑器报警的输出。也影响 ST源文件所有的续行。	
	warning:n:off	不显示数字 N 指定的报警
	warning:n:on	显示数字 N 指定的报警
	允许值为 n n = 0 to 7: 报警级别 ,见 49 页报警级别的意义 n = 16000 及更高 : 报警的编号	
HMI 导出	属性默认改变在 HMI 设备上可用的单元变量。 直接用以下声明块放在相关的关键词之后： ● VAR_GLOBAL ● VAR_GLOBAL RETAIN 仅影响相关声明块中声明的单元变量。 HMI导出的详细描述，特别是基于 SIMOTION kernel版本的属性的影响，见 208页变量和 HMI设备。	
	FALSE	在 ST源文件的 interface 部分。在相关的声明块中声明的单元变量在 HMI 设备上不可用。
	TRUE	在 ST源文件的 implementation 部分。在相关的声明块中声明的单元变量在 HMI 设备上可用。
BlockInit_OnChange	只关于 SIMOTION kernel V3.2 无论当相关的声明块中的版本标识符改变时下载是否可行, 属性改变标准定义。只能直接放在下列声明块相关的关键词之后： ●VAR_GLOBAL(在 interface 或 implementation 部分) ●VAR_GLOBAL RETAIN(在 interface 或 implementation 部分) ●VAR（在激活仅此一次创建程序实例时，仅指单元中的程序） 仅影响相关声明块中声明的变量。 见 207 页全局变量的版本 ID 和下载时的初始化	
	FALSE	当声明块的版本标识符改变时 （默认），在 RUN 模式下的下载是不可能的
	TRUE	即使声明块的版本标识符改变 （默认），在 RUN 模式下的下载是可能的。初始化声明块的变量
BlockInit_OnDeviceRun	只关于 SIMOTION kernel V4.1 无论当相关的声明块中的版本标识符改变时下载是否可行, 属性改	

	变标准定义。只能直接放在下列声明块相关的关键词之后： ●VAR_GLOBAL(在 interface 或 implementation 部分) ●VAR (在激活仅此一次创建程序实例时，仅指单元中的程序) 仅影响相关声明块中声明的变量。见 194 页变量类型的存储范围	
	DISABLE	在从停止转到运行模式时 （默认），在相关的声明块中声明的变量没有初始化
	ALWAYS	在从停止转到运行模式时 （默认），在相关的声明块中声明的变量初始化
注意：确保使用正确的大小写符号的属性！		

注意：在一个声明块中的 HMI_Export, BlockInit_OnChange 或 BlockInit_OnDeviceRun属性的插入，删除和更改不改变它的版本标识符！

Table 5-44 Example of attributes for compiler options

```
INTERFACE
    VAR_GLOBAL
        { HMI_Export := FALSE;
          BlockInit_OnChange := TRUE; }
        // No HMI export, download in RUN possible
        x : DINT;
    END_VAR
    FUNCTION_BLOCK fb1;
END_INTERFACE

IMPLEMENTATION
    VAR_GLOBAL
        { HMI_Export := TRUE;
          BlockInit_OnDeviceRun := ALWAYS; }
        // HMI export, initialization for the STOP -> RUN transition
        y : DINT;
END_VAR
FUNCTION_BLOCK fb1
    VAR_INPUT
        i_var    : INT;
    END_VAR
    VAR_OUTPUT
        o_var    : INT;
    END_VAR

    { _U7_PoeBld_CompilerOption := warning:2:on; }
    o_var := REAL_TO_INT(1.0);    // Warning 16004
    { _U7_PoeBld_CompilerOption := warning:2:off; }
    o_var := REAL_TO_INT(1.0);    // No warning 16004
    { _U7_PoeBld_CompilerOption := warning:16004:on; }
    o_var := REAL_TO_INT(1.0);    // Warning 16004
    { _U7_PoeBld_CompilerOption := warning:16004:off; }
    o_var := REAL_TO_INT(1.0);    // No warning 16004
    { _U7_PoeBld_CompilerOption := warning:2:off;
      _U7_PoeBld_CompilerOption := warning:16004:on; }
    o_var := REAL_TO_INT(1.0);    // Warning 16004
  END_FUNCTION_BLOCK
END_IMPLEMENTATION
```

3.8 跳转语句和标签

除了控制语句（见 130 页控制语句），一个跳转语句也是可用的。

用 GOTO语句编写跳转语句，指定跳转标签到你想要跳转的地方。跳转只允许在 POU中发生。

在你想要重新编写的语句之前输入跳转标签（用冒号隔开）。

或者，你可以在 POU中声明跳转标签（在 POU中用 LABEL/END_LABEL结构）。在语句部分仅可以使用声明的跳转标签。

Table 5-45 Example of syntax for jump statements

```
FUNCTION func : VOID
    VAR
        x, y, z BOOL;
    END_VAR
    LABEL
        lab_1, lab_2;      // Declaration of the jump labels
    END_LABEL
    x := y;
    lab_1 : y := z;         // Jump label with statement
    IF x = y THEN
        GOTO lab_2;         // Jump statement
    END_IF;
    GOTO lab_1;             // Jump statement
    lab_2 : ;               // Jump label with blank statement
END_FUNCTION
```

注意：
仅在特定的条件（例如： troubleshooting ）中使用 GOTO语句。根据结构编程的规则不能全部使用。

只在 POU中允许跳转。

下列跳转是非法的：

- 跳至从属的控制结构（ WHILE, FOR等）
- 从一个 WAITFORCONDITION结构跳转来的
- 在 CASE语句中的跳转

仅在使用的 POU中声明跳转标签。如果声明跳转标签，只能使用声明的跳转标签。

4. 错误源和程序调试

本章描述了编程错误的不同的来源和图和有效编程。你也可以学习到程序测试可用的选项。所有可能的编辑器错误信息，　即编辑器错误，　见 350 页编辑器错误信息和解决方法。每个错误的可能解决方法都有描述。

4.1 避免错误和有效编程的注释

SIMOTION 基础功能手册列出了一些常见的错误源，提示编程者或阻止程序适当的执行。注释如下：

- 指定算术表达式的数据类型
- 在循环任务开启功能
- 在循环任务中等待时间
- 下载错误
- CPU没有转到运行模式
- CPU停止
- 本地数据栈的大小
- 等

除此之外，你还能找到关于有效编程的注释，尤其是：

- 以运行时间为导向的编程
- 更改优化式编程

4.2 程序调试

在编译中 ST编译器检查并显示的语法错误。在程序执行期间的运行时间错误由系统报警显示，或者直接导致停止运行模式。可以使用　　ST测试功能找到逻辑编辑错误。如：符号浏览器，状态程序，追踪等。

4.2.1　程序测试的模式

4.2.1.1SIMOTION 设备模式

程序测试时有多种不同的模式可用。

如何选择 SIMOTION设备的模式：

1.在项目导航中突出　SIMOTION设备

2.在快捷菜单中选择“ 测试模式 ”
3.选择需要的模式（见下表）
如果你选择“ debug 模式 ”
—接受安全信息
—配置 life-sign 监视
观察下列部分： 254 页 life-sign 监视
4.用 OK 确认
SIMOTION设备转到选择模式
当 SIMOTION设备转到“ debug 模式 ”时
—如果 SIMOTION SCOUT现在处于离线模式，自动建立（在线模式）与目标系统的通讯
—在状态栏显示激活的 debug 模式
显示断点工具栏

设置	意义
Process mode	优化 SIMOTION设备上的程序执行以达到最佳系统性能。 下面的诊断功能是可用的， 虽然因为优化的最大系统性能而功能受限： ●在符号浏览器或可看的表格中的监视变量 ●程序状态（仅限制） —变量的限制监视（如：在循环中的变量，系统功能的回送值） —关于 SIMOTION kernel版本 V4.0 每个任务仅监视一种程序源（如： ST 源，MCC源， LAD/FBD源） —高于 SIMOTION kernel版本 V3.2 只是监视一种程序源（如： ST源， MCC 源，LAD/FBD源） ●对于驱动和功能生成器带测量功能的追踪工具（仅限制）， 见在线帮助 —每个 SIMOTION设备仅一次追踪
Test mode	Process mode的诊断功能在全范围内可用： ●在符号浏览器或可看的表格中的监视变量 ●程序状态 —监视所有变量 —关于 SIMOTION kernel版本 V4.0 每个任务仅监视多种程序源（如： ST 源，MCC源， LAD/FBD源） —高于 SIMOTION kernel版本 V3.2 —每个任务仅监视一种程序源（如： ST源，MCC源， LAD/FBD源） ●对于驱动和功能生成器带测量功能的追踪工具（仅限制）， 见在线帮助 —每个 SIMOTION设备仅四次追踪

	注意：运行时间和存储使用随着诊断功能使用的增加而增加
Debug mode	在 SIMOTIONv3.2版本可用。 除了测试模式的诊断功能， 你还可以使用以下功能： ●断点 在一个程序源文件中， 你可以设置断点（271 页）。当达到一个激活的断点时，停止选择任务 ●控制 MOTION 任务 在设备诊断的任务管理器中，你可以使用 MOTION TASK 的任务控制命令， 见 SIMOTION基础功能手册 项目仅有一种 SIMOTION 设备可以转到 DEBUG模式。 SIMOTION SCOU处于在线模式，如：与目标系统通讯。观察下列部分： life-sign 监视的重要信息，见 254 页

4.2.1.2life-sign 监视的重要信息

警告：
你必须注意合适的安全规定。

使用 debug 模式或仅带通过激活恰当的短期监视时间的 life-sign 监视功能的控制面板！ 否则在 PC和 SIMOTION设备通讯连接时会产生问题，轴会以不可控的方式运动。

仅仅为调试、诊断和服务目的开放功能。只有授权的技术人员才能使用功能。高级别控制的安全关闭没有影响。

因此，在硬件中必须有一个急停电路。用户应当采取适当措施。

接受安全注释：
在选择 debug 模式或控制面板之后， 你必须接受安全注释。 你可以给 life-sign 监视设置参数：
按照以下操作
1.点击设置按钮
“ debug 设置 ”窗口打开
2.如下节描述，读取安全注释，设置 life-sign 监视参数

life-sign 监视参数设置
在 life-sign 监视参数设置窗口，如下列描述进行：
1.读警告

2.点击安全注释按钮来打开窗口
3.不要给默认的 life-sign 监视做任何的改动
仅在特别的环境中和查看所有安全警告时更改
4.点击接受以确认你已经阅读了安全注释和正确设置 life-sign 监视的参数

注意：
按下空格键或转到一个不同的 Windows 应用会导致：
●Debug 模式中激活的断点
—SIMOTION设备转到停止模式
输出失效 （ODIS）
●使用控制面板（控制 PC的优先级）控制一个轴或一个驱动
—停止轴或驱动
—重新设置开启

警告：在所有的运行模式不能保证此功能。因此，在硬件中必须有急停电路。用户必须采取适当措施。

4.2.1.3life-sign 监视参数

Field	描述
life-sign 监视	SIMOTION设备和 SIMOTION SCOUT定期交换 life-sign 信号来确保一个正确的功能连接。如果 life-sign 的交换被干扰的时间长于设置的监视时间，将会产生下列： ●Debug 模式中激活的断点 —SIMOTION设备转到停止模式 输出失效 （ODIS） ●使用控制面板 （控制 PC的优先级） 控制一个轴或一个驱动 —停止轴或驱动 —重新设置开启 下列参数是可能的： ●激活的检查框 如果选择检查框， life-sign 监视被激活。 life-sign 监视的失效不总是可能的。 ●监视时间 输入超时 注意：不要给默认的 life-sign 监视做任何的改动 仅在特别的环境中和查看所有安全警告时更改

安全信息	请查看警告！ 点击按钮来获得更多的安全信息。 见 254 页 life-sign 监视的重要信息

4.2.2 符号浏览器

4.2.2.1 符号浏览器的属性

在符号浏览器中，你可以查看，如有必要，更改名称、数据类型和变量数值。特别是，
见下列变量：

- 一个程序或功能块的单元变量和静态变量
- SIMOTION设备或技术目标的系统变量
- I/O 变量或全局设备变量

对于这些变量，你可以

- 查看变量值的快照
- 查看更改的变量值
- 更改变量值

但是，符号浏览器仅可以显示 /修改变量值，如果已经在目标系统中装载项目，并建立了到目标系统的通讯。

4.2.2.2 使用符号浏览器

要求：

- 确保已经建立与目标系统的通讯， 已经下载一个项目到目标系统。 装载示范程序的项目，见 66 页执行示范程序
- 你可以允许用户程序，但是却没有必要。如果程序没有运行，仅见变量的初始值。

基于监视的变量所在的存储区域的程序被保存。

在单元的用户内存或保留内存中的变量

- 在程序源文件（单元） interface 部分的保留和不保留单元变量
- 在程序源文件（单元） implementation 部分的保留和不保留单元变量
- 实例作为单元变量声明的功能块的静态变量
- 如果程序源文件用仅此一次创建程序实例的编辑选项来编辑（ 44 页）
  —程序的静态变量
  —实例作为静态变量声明的功能块的静态变量

对于下列步骤：

1.在项目导航中选择程序源文件（如： ST_1）
2.在细节窗口，点击符号浏览器标签

在符号浏览器中，程序源文件的所有变量包含在单元的用户内存和保留内存中。

- 程序源文件的所有单元变量
- 程序源文件用仅此一次创建程序实例的编辑选项来编辑： 程序源文件的程序和静态变量（包括功能块的实例）

任务的用户内存的变量

你可以使用符号浏览器来监视包含在相关任务的用户内存中的变量，见 194 页变量类型的内存范围。

如果没有默认的仅此一次创建程序实例的编辑选项来编辑程序源（单元），程序指定的用户内存任务包含下列变量：

- 程序的静态变量
- 实例作为静态变量声明的功能块的静态变量

根据下列步骤：

1.在 SIMOTION SCOUT的项目导航中，在 SIMOTION 设备的树形目录中选择 EXECUTION SYSTEM元素

2.在细节窗口，点击符号浏览器标签

符号浏览器显示了在执行系统中所有使用的任务。在用户内容的任务中包含的相关变量列在下方。

注意：

你可以用程序状态来监视临时变量（还有单元变量和静态变量），见 265 页程序状态的属性

系统变量和全局设备变量

在符号浏览器中你可以监视下列变量：

- SIMOTION设备的系统变量
- 技术目标的系统变量
- I/O 变量
- 全局设备变量

根据下列步骤：

1.在 SIMOTION SCOUT项目导航中选择适当元素

2.在细节窗口，点击符号浏览器标签

在符号浏览器中显示对应的变量

状态和控制变量

在状态值一栏中，显示现行变量值，并定期更新。

你可以更改一个或多个变量值。需要更改的变量如下操作：

1.在控制数值一栏中输入一个数值

2.在栏中激活检查框
3.点击立即控制按钮
输入的数值被写给选择的变量

注意：
当你更改多个变量的数值时注意：
按照顺序给变量写值。 花费几毫秒知道写下个数值。 在符号浏览器中从上到下更改变量。
所以一致性没有保证。

符号浏览器的显示更改
你可以固定主动对象符号浏览器的显示：
●为此，点击符号浏览器右上角  保留显示按钮。显示符号变为 
目标的变量任然显示，并在符号浏览器中更新，即使在项目导航中选择另外的目标。
●为了去除显示，再次点击图标  。 显示符号又变回 

显示无效的浮点数字
在符号浏览器中如下显示无效的浮点数：

LED	意义
1.#QNAN -1.#QNAN	根据 IEEE754 (NaN不是一个数字)的无效的位模式。在 signaling NaN (NaNs)和 quiet NaN (NaNq)之间没有差别
1.#INF -1.#INF	根据 IEEE754的+ infinity 的位模式 根据 IEEE754的- infinity 的位模式
-1.#IND	不确定的 Bit 模式

4.2.3 在 watch 表中监视变量

4.2.3.1 在 watch 表中的变量

在符号浏览器中，你可以看见在项目中的一个目标的变量。用项目状态，你仅可以看见在一个自由选择监视区域的 ST源文件的变量。

在 watch 表中，作为比对，你可以监视一个组别中不同源的选择的变量（如：程序源，技术目标， SINAMICS驱动—即使是不同的设备）

你可以在离线模式下查看变量的数据类型。你可以在离线模式下查看和修改变量的数值。

4.2.3.2 使用 watch 表格

你可以从不同的程序源、技术目标和 SIMOTION设备中（甚至在不同的设备）分组变量，

在 watch 表中可以查看，如果必要也可以修改。

创建一个 watch 表
创建一个 watch 表和指定变量的步骤
1.在项目导航中，选择监视文件夹
2.选择插入 >watch 表以创建一个 watch 表格，然后输入表格名称。 带名称的 watch 表会出现在监视文件夹中
3.在项目导航中，点击你想要把变量转移到 watch 表的目标
4.在符号浏览器，通过点击左侧栏的数字选择相应的变量行
5.从快捷菜单中选择 MOVE变量到 watch 表，如： Watch table_1
6.如果你点击 watch 表，你将在 watch 表标签的细节窗口看见选择的变量现在在 watch 表中
7.重复 3-6 步来监视不同目标的变量
如果与目标系统通讯，你可以监视变量内容。

4.2.4 程序运行

4.2.4.1 程序运行：显示代码位置和调用路径

你可以在一个 MOTION TASK执行和带调用路径的代码 （如：一个 ST源文件的行） 中显示位置。

遵循以下步骤：
1.在程序运行工具栏点击“ 显示程序运行 ”按钮
2.选择想要的 MOTION TASK
3.点击更新按钮

窗口显示：
●正在执行的代码中的位置说明程序源和 POU
●调用执行的代码位置的其他 POU 的代码中的递归位置

SIMOTION程序源文件的名称显示如下：

名称	意义
taskbind.hid	执行系统
stdfunc.pck	IEC库
device.pck	设备特有的库
tp-name.pck	tp-name 技术包的库 如： CAM 技术包的库为 cam.pck

4.2.4.2 参数调用栈程序运行

所有的配置的任务显示如下：

●在程序代码中的现行代码位置

●代码位置的调用路径

Field	描述
选择的 CPU	显示选择的 SIMOTION设备
刷新	点击按钮读取从 SIMOTION设备来的现行代码位置，在打开的窗口显示
调用任务	选择你想要执行代码位置的任务。 所有执行系统的配置任务
现行代码位置	显示在程序代码中执行的位置 （程序源文件的名称、行数、 POU名称）
调用者	递归显示（源文件名称，行数和 POU 名称，功能块实例的名称， 如适用） 代码位置调用在选择的任务中执行的代码位置

4.2.4.3 程序运行工具栏

你可以在通过工具栏的调用路径 MOTION TASK正在执行的代码（如： ST源文件的行）中显示位置。

符号	意义
	显示程序运行 点击按钮来打开程序运行调用栈窗口。 在这个窗口中， 你可以显示带调用路径的主动代码位置。 见： 263 页程序运行：显示代码位置和调用路径

4.2.5 程序状态

4.2.5.1 程序状态的属性

程序状态使得在执行程序时正确监视变量

你可以在 ST源文件中选择一个监视区域，然后监视。除了全局和静态本地数据，也可以

监视本地变量。

●简单数据类型变量

●一个结构的单独元素，，假设有一个指定

●一个阵列的单独元素，，假设有一个指定

●枚举数据类型变量

当在 ST源文件中运行选择的监视范围时，要监视的变量对应的缓冲充满 SIMOTION设备上的对应数值。一旦运行选择的监视范围，格式化缓冲以在 SIMOTION SCOUT中显示。SIMOTION SCOUT定期调用格式化的数值，并显示。

关于 SIMOTION kernelV3.2版本，在 ST源文件中你可以选择一个功能、 功能块或实例调用的位置。你可以查看这次调用的变量数值。

注意：因为限制的缓冲容量和对最小运行时间的篡改的要求，不能显示下列变量：

●完整的阵列

●完整的结构

显示独自的阵列元素或结构元素，但是假设在 ST源文件中发生了指定

	Process mode	Test mode
程序执行的优化	对于最大化的系统性能，仅限制诊断是可能的	全部的诊断选项
监视程序源（如：ST源文件、MCC源文件、 LAD/FBD源）的最大数	●关于 SIMOTION kernel版本 V4.0：每个任务最多一个程序源 ●高于 SIMOTION kernel版本 V3.2：最多一个程序源	●关于 SIMOTION kernel版本 V4.0：每个任务多个程序源 ●高于 SIMOTION kernel版本 V3.2：最多一个程序源
循环（如：WHILE,REPEAT,FOR	在 repeat 循环，记录被干扰。如果选择所有的循环，在第一次运行循环时显示数值	如果有 repeat，记录继续。如果选择所有的循环，在最后一次运行循环时显示数值
包含内部循环的系统功能（如：过程串的功能）	在某些情况下不显示数值	正确显示数值

注意：

程序状态要求额外的 CPU资源

如果要同时监视许多程序和状态程序请注意：

●必须激活测试模式（见 252 页 SIMOTION设备操作模式）

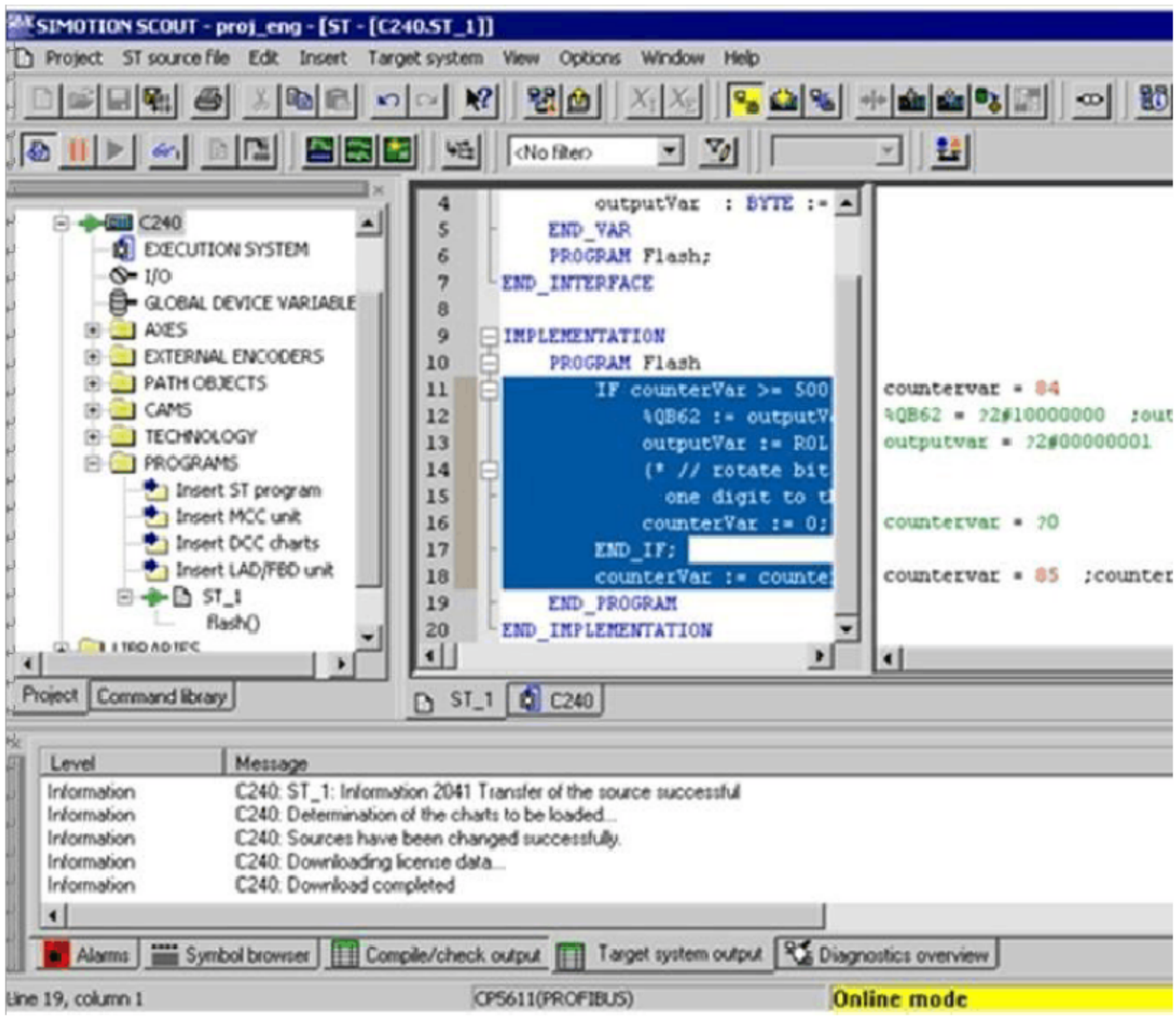
●高于 SIMOTION kernel版本 V4.0，必须指定程序到不同任务

4.2.5.2 使用状态程序

在你使用状态程序之前，必须命令系统运行特殊模式

1.确保在编辑时　ST源文件生成额外的　debug 代码
—项目导航中选择　ST源文件，选择编辑　>目标属性菜单命令
—选择编辑器标签来更改编辑器的本地设置（　　46 页）
—确保激活了启动状态程序检查框，并确认
也可以在编辑器的全局设置中更改编辑器选项
2.打开 ST源文件，用　ST源文件 >接受和编辑　来重新编辑
3.下载并以平常方式开启程序
4.点击 ST编辑菜单栏中程序状态按钮  ，开启测试模式

ST编辑窗口被垂直分开：你可以在左侧窗格看见　　ST源文件，右侧窗格显示选择的变量和数值。



用程序状态测试的步骤如下：
1.在编辑器中，选择要测试的　ST源文件的部分
2. 关于 SIMOTION kernelV3.2版本：

如果你选择在一个程序源文件或多个任务中的多个位置调用的一个 POU 的部分：
输入程序状态的调用路径（见 268 页）

对于选择的部分，你可以在屏幕右侧窗口看见变量和数值，循环更新这些变量和数值：
●在当下路径中更改的数值用红色显示
●没有更改的数值用黑色显示
●没有值的变量，如：未使用的 IF 分支的变量用绿色显示，并用问号标明

●点击  按钮以停止监视 ST 编辑器工具栏（见 43 页）的程序变量，停止显示

●点击  按钮以继续监视 ST 编辑器工具栏（见 43 页）的程序变量，继续显示
你可以强制更新显示的数值

●点击  按钮以在 ST 编辑器工具栏上更新（见 43 页）

读取 SIMOTION设备的缓冲， 即使选择监视的范围没有被完全进行， 数值也不完整。 这是很有用的，例如：如果程序在等待一个 WAITFORCONDITION语句。

必须激活监视程序变量。

4.2.5.3 程序状态的调用路径

对于 SIMOTION kernelV3.2或更高版本，在监视功能和功能块的变量数值时你可以指定调用路径。这使你能查看调用的变量数值。

处于此目的，在下列情况中主动打开调用路径窗口：
●你已经选择一个功能的部分
在 SIMOTION设备程序源文件（如： ST源文件）中不同点调用功能
●你已经选择一个功能块的部分
在 SIMOTION设备程序源文件中不同点有多个功能块的实例或调用的实例
●你已经选择一个程序的部分
程序被指定给多余一个的任务

如何选择调用路径
在调用路径状态程序窗口，标明的 POU部分（编码位置）显示（ ST源文件名称，行数和 POU名称）
1.如果在多个任务中调用代码位置
—选择任务
2.选择要调用的代码位置（在调用的 POU 中）
你可以从下选择：
—在选择的任务中要调用的代码位置（ ST源文件名称，行数和 POU名称）
如果选择调用的代码位置被多个代码位置轮流调用，会显示更多的行
—所有

选择所有显示的代码位置。但是，所有的代码位置（到最顶级）是从调用的显示代码位置中选择的

高于 SIMOTION kernelV3.1版设备的程序状态
注意：
如果你使用高于 SIMOTION kernelV3.1版设备的程序状态：
●使用高于 V3.1 版的 SINOTION SCOU编辑程序，在描述格式中的调用路径不可用。你仅能使用在编辑时可用的诊断功能。
●如果使用高于 V3.2 版的 SINOTION SCOU编辑程序，你可以指定调用路径

当用编辑器的当前版本重新编辑时，注意以下：
●在其他影响中，在程序的数据存储区域会生成新的版本标识符
●删除和初始化 SIMOTION 设备上所有保留和不保留数据
●在某些情况中，要求少数程序源的更改
●当转回到原来的项目状态时，必须重新编辑程序

4.2.5.4 参数调用路径状态程序

Field	描述
调用任务	选择任务 被调用的代码位置中的所有任务对于选择是可用的
现行代码位置	POU（代码位置）的选择部分显示（程序源文件的名称、行数、 POU 名称）
调用者	选择调用代码位置 下列内容是可用的： ●在选择的任务（程序源文件的名称、行数、 POU名称）中调用的代码位置 如果选择的调用代码位置被多个代码位置轮流调用，显示更多的行 ●所有 选择所有显示的代码位置。但是，所有的代码位置（到最顶级）是从调用的显示代码位置中选择的

4.2.6 断点

4.2.6.1 设置断点的普通步骤

你在一个程序源（ ST源、 MCC 图表、 LAD/FBD源）的 POU中可以设置断点。一旦达到一个激活的断点，，带断点的 POU中的任务停止。如果启动位于程序或功能块中的任务停

止的断点，此 POU的静态变量数值在细节变量状态中显示。不显示临时变量（功能块的 IN/OUT 参数也如此）。你可以监视其他 POU的静态变量或符号浏览器中的单元变量。

要求：带 POU 的程序源（如： ST源文件， MCC图表， LAD/FBD程序）是开启的。

步骤如下：
1.在相关 SIMOTION设备上选择 debug 模式（见 271 页 debug 模式设置）
2.给 debug 任务分组，见 273 页 debug 任务分组
3.设置断点，见 276 页断点设置
4.定义调用路径，见 279 页定义一个单独的断点的调用路径
5.激活断点，见 285 页激活断点

4.2.6.2 设置 debug 模式

警告：
你必须注意合适的安全规定。

使用 debug 模式或仅带通过激活恰当的短期监视时间的 life-sign 监视功能的控制面板！ 否则在 PC和 SIMOTION设备通讯连接时会产生问题，轴会以不可控的方式运动。

仅仅为调试、诊断和服务目的开放功能。只有授权的技术人员才能使用功能。高级别控制的安全关闭没有影响。

因此，在硬件中必须有一个急停电路。用户应当采取适当措施。

设置 debug 模式的步骤如下：
1. 在项目导航中突出 SIMOTION设备
2.在快捷菜单中选择测试模式
3.选择 debug 模式（ 252 页）
4. 接受安全信息
5. 配置 life-sign 监视参数
6.确认 OK
如果在目标系统（离线模式）没有建立通讯，自动建立在线模式
在状态栏显示激活的 debug 模式
显示断点工具栏（ 278 页）

注意：不能在 debug 模式中更改程序源

注意：
按下空格键或转到一个不同的 Windows 应用将在 debug 模式中激活断点
- SIMOTION设备转到停止模式
- 输出失效 （ODIS）

警告：在所有的运行模式不能保证此功能。因此，在硬件中必须有急停电路。用户必须采取适当措施。

4.2.6.3 定义 debug 任务组

一旦达到一个激活的断点，停止指定给 debug 任务组别的所有任务。

要求：相关的 SIMOTION设备处于 debug 模式

步骤如下：

如何给 debug 任务组指定任务：

1. 在项目导航中突出相关 SIMOTION设备
2. 在快捷菜单中选择 debug 任务组
3. 在到达断点时选择要停止的任务

—如果你仅想停止独自的任务（在运行模式）：激活 debug 任务组选项在到达断点时指定要停止的所有任务给停止任务列表

—如果你仅想停止独自的任务（在运 HALT模式）：激活所有任务选项

在这种情况下，选择在恢复程序执行后是否再次发布输出和技术目标

注意：当达到激活断点时不同的动作，见下表

属性		要停止的任务	
		单独选定的任务（debug 任务组）	所有的任务
达到断点的动作			
	操作模式	RUN	STOP
	停止任务	仅在 debug 任务组的任务	所有任务
	输出	ACTIVE	DEACTIVATED(ODIS激活)
	技术	Closed-loop 控制	不闭环控制（ ODIS激活）
	任务的运行时间测量	所有任务激活	所有任务不能用
	任务的时间监视	在 debug 任务组中的任务无效	所有任务无效
	实时时钟	继续运行	继续运行
重启程序执行时的动作			
	操作模式	RUN	RUN
	启动任务	所有在 debug 任务组的任务	所有任务
	输出	ACTIVE	输出和技术目标的动作基于“继续”激活的输出（ ODIS无效）检查框。 ●激活：ODIS无效。发布所有
	技术	Closed-loop 控制	

			的输出和技术目标 ●失效：ODIS保持激活。 仅在项目的另一次下载之后发布所有的输出和技术目标

注意：如果没有激活的断点，你可以对 debug 任务组做改动。

步骤如下：

1.设置断点（见 276 页断点设置）
2.定义调用路径（见 279 页定义一个单独断点的调用路径）
3.激活断点（见 285 页激活断点）

4.2.6.4debug 任务组参数

使用窗口来定义 debug 任务组。一达到一个激活的断点，停止所有指定给 debug 任务组的任务。

这要求相关的 SIMOTION设备处于 debug 模式，见 252 页 SIMOTION设备的模式

FIELD	描述
Debug 任务组	如果你仅想停止独自的任务， 选择此选项。 在达到激活的断点时， SIMOTION设备保持运行模式。输出和技术目标保持激活。 在达到断点时指定要停止的所有任务给任务停止列表。
所有的任务	如果你仅想停止所有用户任务， 选择此选项。 在达到激活的断点之后， SIMOTION设备保持停止模式。输出和技术目标失效（ ODIS激活）。 在这种情况下，选择在恢复程序执行后是否再次发布输出和技术目标。
“ 重启 ”激活输出（ ODIS无效）	只要选择所有任务 激活检查框，选择在恢复程序执行后再次发布输出和技术目标。 在下载带失效激活框程序之后此案呢过发布所有的输出和技术目标

注意：激活断点的不同动作基于停止的任务，见 273 页定义 debug 任务组的表格

如果没有激活的断点，可以对 debug 任务组做改动

4.2.6.5debug 表格参数

debug 表显示了在 SIMOTION设备的程序源中所有的 debug 点

FIELD		描述
Debug 点（表）		
	Active	显示断点的激活状态 点击检查框来更改激活状态 见 285 页激活断点
	Source, line (POU)	代码位置以点集显示（程序源文件的名称、行数和 POU名称）
	Debug type	显示 Debug 点的类型（如：断点，追踪点）
	Call path	点击按钮来定义断点的调用路径 见 279 页定义一个单独断点的调用路径
所有的断点		
	Activate	点击按钮来激活 SIMOTION设备的所有断点 （在所有程序源中） 见 285 页激活断点
	Deactivate	点击按钮来失效 SIMOTION设备的所有断点 （在所有程序源中） 见 285 页激活断点
	Delete	点击按钮来清除 SIMOTION设备的所有断点 （在所有程序源中） 见 276 页设置断点

4.2.6.6 设置断点

要求：

1.打开带 POU（如： ST源文件、 MCC图表、 LAD/FBD程序）的程序源
2.相关 SIMOTION设备处于 debug 模式，见 271 页 debug 模式设置
3.定义 debug 任务组，见 273 页定义 debug 任务组

步骤如下：

如何设置一个端点：

1.选择没有设置断点的代码位置
—SIMOTION ST把光标放在包含一个语句的 ST源文件中的行
—SIMOTION MCC在 MCC图表中（除了注释块的模块）选择 MCC 命令
2.或者：
—选择 编辑 >设置断点菜单命令

在断点工具栏点击按钮 

删除一个断点的操作步骤如下：

1.选择断点的代码位置

2.或者：

—选择 编辑 >设置断点菜单命令

在断点工具栏点击按钮 

删除 SIMOTION设备上所有断点的步骤如下：

—选择 DEBUG>删除所有断点菜单命令

在断点工具栏点击按钮 

注意：

你不能设置断点：

- 对于 SIMOTION ST：包含 ONLY语句的行中
- 对于 SIMOTION MCC：在模块或注释块命令中
- 对于 DIMOTION LAD/FBD：在一个网络中
- 在其他 debug 点（如： trigger 点）设置的代码位置

你可以在位于 debug 表格中列出 SIMOTION设备的所有程序源

- 在断点工具栏点击 debug 表格按钮 

在 debug 表格中，你可以删除 SIMMOTION设备所有断点 (在程序源中)

- 点击删除所有断点按钮

在离开 debug 模式之后设置依然保存的断点，只在 debug 模式时显示。

你可以在程序源文件或 POU 中一起使用程序状态（ 266 页）诊断功能和断点。但是基于程序语言应用下列限制：

- SIMOTION ST对于 SIMOTION kernel V3.2 版，一个断点不能包含要测试的带程序状态的（标记的） ST源文件行
- SIMOTION MCC和 SIMOTION LAD/FBD：一个断点不能包含带程序状态的要测试的 MCC 表格（或 LAD/FBD程序的网络）命令

步骤如下：

1.定义调用路径，见 279 页定义一个单独断点的调用路径

2.激活断点，见激活断点（ 285 页）

4.2.6.7 断点工具栏

这个工具栏包含设置和激活断点的重要操作动作

符号	意义
	设置 /删除断点 点击图标以设置选择代码位置的断点或者删除一个已存在的断点 见 276 页设置断点
	激活 /失效断点 点击图标以激活或失效来设置选择代码位置的断点 见 285 页激活断点
	编辑调用路径 点击图标来定义断点的调用路径： ●如果选择带断点的代码位置：此断点的调用路径 ●如果选择不带断点的代码位置： POU 所有断点的调用路径 见:279 页定义一个单独断点的调用路径, 282 页所有断点的调用路径
	激活所有断点 点击图标来激活现行程序源或 POU（ST源文件、 MCC图表、 LAD/FBD 程序）中的所有断点 见 285 页激活断点
	失效所有断点 点击图标来失效现行程序源或 POU（ST源文件、 MCC图表、 LAD/FBD 程序）中的所有断点 见 285 页激活断点
	删除所有断点 点击图标来删除现行程序源或 POU（ST源文件、 MCC图表、 LAD/FBD 程序）中的所有断点 见 276 页设置断点
	Debug 表格 点击图标来显示 debug 表格 见 276 页 debug 表格参数
	显示调用栈 在达到激活的断点之后点击图标来： 查看当前断点的调用路径 查看要停止的 debug 任务组中其他任务的代码位置和调用路径 见 287 页显示调用栈
	恢复 点击图标以达到激活断点之后继续程序执行 见 285 页激活断点和 287 页显示调用栈

4.2.6.8 定义一个单独断点的调用路径

要求：

1.打开带 POU的程序源（ ST源文件、 MCC 图表、 LAD/FBD程序）
2.相关的 SIMOTION设备处于 debug 模式，见 271 页设置 debug 模式
3.定义了 debug 任务组，见 273 页定义 debug 任务组
4.设置断点，见 276 页设置断点

步骤如下：
1.选择已经设置断点的代码位置：
●SIMOTION ST: 把光标放在 ST源的合适的行中
●SIMOTION MCC: 在 MCC图表中选择合适的命令
●SIMOTION LAD/FBD: 把光标放在 LAD/FBD程序合适的网络中

2.点击断点工具栏中编辑调用路径按钮 

在路径调用 / 任务选择断点窗口，显示标记的代码位置（程序源文件的名称、行数和 POU 名称）

3.当达到选择断点时， 在将要停止的用户程序 （在 debug 任务组中的所有任务） 中选择任务。
下列为可用：
—在这个调用级别开始的所有调用位置
当达到 debug 任务组的任意任务中激活断点时，总会开启用户程序
—可以达到的选择的断点中的独自的任务
当选择任务中的断点达到时，停止用户程序。任务必须在 debug 任务组中。
调用路径指定是可能的。

4.仅对功能和功能块：选择调用路径，如：要调用的代码位置（在调用的 POU 中）：
下列为可用：
—在这个调用级别的所有调用位置
没有指定调用路径。如果调用选择任务中的 POU，在激活断点停止用户程序
—选择一个单独的任务时：在选择的任务内调用代码位置（程序源文件的名称、行数和 POU名称）
指定调用路径。仅当从选择的代码位置调用 POU时，激活断点的用户程序停止。
如果选择调用代码位置的 POU也是从其他代码位置被调用，显示更多的行

5.如果仅在代码位置达到指定时间后激活断点，选择次数

注意：在 debug 表格中你可以定义到独自断点的调用路径：

1.点击断点工具栏中的 debug 表格按钮  ，打开 debug 表格窗口
2.在调用路径栏点击合适的按钮
3.如下所述进行步骤：
—指定任务
—定义调用路径（仅对功能和功能块）
—在断点激活后指定通过次数

如下所述进行：
激活断点，见 285 页激活断点

注意：
你可以使用 287 页“ 显示调用栈 ”功能来查看现行断点值的调用路径和 debug 任务组中其他停止任务的代码位置

4.2.6.9 断点调用路径 / 任务选择参数

Field	描述
选择的 CPU	显示选择的 SIMOTION设备
调用任务	当达到断点时，选择要停止的用户程序 （如：在 debug 任务组中的所有任务） 所在的任务 下列是可用的： ●在此调用级别开始的所有调用位置 当达到 debug 任务组的任意任务中激活断点时，总会开启用户程序 ●可以达到的选择的断点中的独自的任务 当选择任务中的断点达到时，停止用户程序。任务必须在 debug 任务组中。 调用路径指定是可能的。
现行代码位置	代码位置和设置断点（程序源文件的名称、行数和 POU 名称）一起显示
调用者	仅对功能和功能块： 见调用路径： 如调用的代码位置 （在调用的POU中） 下列为可用： ●在这个调用级别的所有调用位置 没有指定调用路径。如果达到任务中的 POU, 在激活断点停止用户程序 ●选择一个单独的任务时：在选择的任务内调用代码位置 （程序源文件的名称、 行数和POU名称）。 如果选择调用代码位置的 POU也是从其他代码位置被调用，显示更多的行
每次 nth 通过时激活断点	如果你不想激活断点， 直到达到特定次数的代码位置，设置这个数字。

注意：
如果没有激活的断点，你只能更改 debug 任务组

4.2.6.10 定义所有断点的调用路径

用这些步骤你可以：

●为所有 POU中(如：MCC表格， LAD/FBD程序或 ST源文件中的 POU)将来的断点选择一个默认设置

●接受和比较在此 POU中的先前设置的所有断点

要求：

1.打开带 POU（如： ST源文件、 MCC图表、 LAD/FBD程序）的程序源

2.相关 SIMOTION设备处于 debug 模式，见 271 页 debug 模式设置

3.定义 debug 任务组，见 273 页定义 debug 任务组

步骤如下：

定义 POU中所有将来的断点的调用路径，步骤如下：

1.选择没有设置断点的代码位置：

●SIMOTION ST: 把光标放在 ST源的合适的行中

●SIMOTION MCC: 在 MCC图表中选择合适的命令

●SIMOTION LAD/FBD: 把光标放在 LAD/FBD程序合适的网络中

2.点击断点工具栏中编辑调用路径按钮 

在路径调用 / 每个 POU 任务选择所有断点窗口， 显示标记的代码位置 （程序源文件的名称、行数和 POU 名称）

3.当达到选择断点时， 在将要停止的用户程序 （在 debug 任务组中的所有任务） 中选择任务。

下列为可用：

—在这个调用级别开始的所有调用位置

当达到 debug 任务组的任意任务中激活断点时，总会开启用户程序

—可以达到的选择的断点中的独自的任务

当选择任务中的断点达到时，停止用户程序。任务必须在 debug 任务组中。

调用路径指定是可能的。

4.仅对功能和功能块：选择调用路径，如：要调用的代码位置（在调用的 POU 中）：

下列为可用：

—在这个调用级别的所有调用位置

没有指定调用路径。如果调用选择任务中的 POU，在激活断点停止用户程序

—选择一个单独的任务时：在选择的任务内调用代码位置（程序源文件的名称、行数和 POU名称）

指定调用路径。仅当从选择的代码位置调用 POU时，激活断点的用户程序停止。

如果选择调用代码位置轮流被其他代码位置调用，显示更多的行

5.如果仅在代码位置达到指定时间后激活断点，选择次数

6.如果你想在 POU 中接受和比较所有先前设置的断点：
—点击接受

如下所述进行：
激活断点，见 285 页激活断点

注意：
你可以使用 287 页“ 显示调用栈 ”功能来查看现行断点值的调用路径和 debug 任务组中其他停止任务的代码位置

4.2.6.11 每个 POU 所有断点的调用路径 / 任务选择参数

在此你可以预先定义一个 POU 中设置的所有将来断点的调用路径。你也可以接受此 POU 中所有先前设置断点的设置。

Field	描述
选择的 CPU	显示选择的 SIMOTION 设备
调用任务	当达到 POU 中的断点时，选择要停止的用户程序（如：在 debug 任务组中的所有任务）所在的任务 下列是可用的： ●在此调用级别开始的所有调用位置 当达到 debug 任务组的任意任务 POU 中激活断点时，总会开启用户程序 ●可以达到的选择的断点中的独自的任务 当选择任务中的断点达到时，停止用户程序。任务必须在 debug 任务组中。 调用路径指定是可能的。
现行 POU	显示光标所在的 POU（程序源文件的名称、行数和 POU 名称）
调用者	仅对功能和功能块： 见调用路径： 如调用的代码位置 （在调用的 POU 中） 下列为可用： ●在这个调用级别的所有调用位置 没有指定调用路径。 如果在选择任务中调用 POU，在激活断点停止用户程序 ●选择一个单独的任务时：在选择的任务内调用代码位置 （程序源文件的名称、 行数和 POU 名称）。 指定调用路径。 仅当从选择的代码位置调用 POU 时，激活断点的用户程序停止。 如果选择调用代码位置的 POU 也是从其他

	代码位置被调用，显示更多的行
每次 nth 通过时激活断点	如果你不想激活断点， 直到达到特定次数的代码位置，设置这个数字。
应用此调用路径到此 POU的所有之前断点	点击应用按钮， 如果你想要应用此调用路径到此现行 POU的所有之前断点。任意存在的设置将被重写

4.2.6.12 激活断点

如果将对程序执行有影响，必须激活断点

要求：

1.打开带 POU的程序源（ ST源文件、 MCC 图表、 LAD/FBD程序）
2.相关的 SIMOTION设备处于 debug 模式，见 271 页设置 debug 模式
3.定义了 debug 任务组，见 273 页定义 debug 任务组
4.设置断点，见 276 页设置断点
5.定义调用路径，见 279 页定义一个单独断点的调用路径

如何激活一个单独的断点，步骤如下：

1.选择已经设置断点的代码位置：
●SIMOTION ST: 把光标放在 ST源的合适的行中
●SIMOTION MCC: 在 MCC图表中选择合适的命令
●SIMOTION LAD/FBD: 把光标放在 LAD/FBD程序合适的网络中

2.或：
—选择 debug>激活 / 失效断点菜单命令

在断点工具栏点击按钮 

激活 SIMOTION设备的所有断点（在所有程序源中），步骤如下：
—选择 debug>激活 / 失效断点菜单命

在断点工具栏点击按钮

注意：

在 debug 表格中也可以激活 /失效 SIMOTION设备的所有程序源的断点

1.点击断点工具栏中的 debug 表格按钮 ，打开 debug 表格窗口

2. 基于你想要激活 /失效的断点执行下列动作
—单独的断点：检查或清除相对应的检查框
—所有断点（在所有程序源中）：点击相对应的按钮

在激活断点的动作

在达到激活的断点时（使用选择的调用路径，见　279 页），停止所有指定给　debug 任务组的所有任务。动作基于　debug 任务组中的任务，并在　273 页的定义一个　debug 任务组中描述。断点部分被强调。

如果停止任务的断点位于一个程序或功能块中，在变量状态标签中显示此　POU 的静态变量。不显示临时变量（　in/out 参数也不显示）。你可以监视其他　POU 中的静态变量或在符号浏览器的单元变量（　257 页）。

你可以使用显示调用栈（　287 页）的功能来：
●查看现行断点的调用路径
●查看已经停止的　debug 任务组的其他任务所在的代码位置和调用路径

恢复程序执行
如何恢复程序执行：

点击断点工具栏中的按钮  来恢复（ ctrl+F8shortcut ）

失效断点：
1.选择已经激活断点的代码位置：
2.
—选择　debug>激活 / 失效断点菜单命令

在断点工具栏点击按钮 

失效 SIMOTION设备的所有断点（在所有程序源中），步骤如下：
—选择　debug>失效断点菜单命

在断点工具栏点击按钮 

4.2.6.13 显示调用栈

你可以使用调用栈功能来：
●查看现行断点的调用路径
●查看已经停止的　debug 任务组的其他任务所在的代码位置和调用路径

要求：
在一个激活的断点停止用户程序，如：停止　debug 任务组中的任务

步骤如下：
调用显示调用栈的步骤如下：

在断点工具栏点击显示调用栈按钮 

打开断点调用栈对话框。显示现行调用路径（包括调用任务和设置通过次数）

调用路径不可更改

使用显示调用栈功能的步骤如下：

1.保持打开断点调用栈对话框

2.显示停止的其他任务和代码位置，步骤如下：

—选择合适的任务。 debug 任务组中的所有任务都可以选择

显示代码位置，包括调用路径。如果代码位置包含在用户程序中，打开带 POU 的程序源（ST源文件、 MCC图表、 LAD/FBD程序），标记代码位置。

3.如何恢复程序执行

在断点工具栏点击恢复（ ctrl+F8shortcut）按钮 

当达到下一个激活的断点时， debug 任务组的任务会再次停止。 显示现行的调用路径， 包括调用任务。

4.点击“ OK”以关闭断点调用栈对话

关于 SIMOTION RT程序源的名称，见 263 页程序运行中的表格

4.2.6.14 断点调用栈参数

当达到一个激活的断点，在 debug 任务组中的每个任务都可以显示下列：

●在任务停止的程序代码（如： ST源文件的行）的位置

●此代码位置的调用路径

Field	描述
选择的 CPU	显示选择的 SIMOTION设备
调用任务	当达到断点时， 选择要停止的用户程序 （如：在 debug 任务组中的所有任务）所在的任务。 可以选择 Debug 任务组中的所有任务。
现行代码位置	显示停止选择的任务的程序代码的位置 （程序源文件的名称，行数和 POU名称）
调用者	递归显示在选择的任务内调用现行代码位置的代码位置 （程序源文件的名称， 行数和 POU名称，功能块实例的名称，如适用）

关于 SIMOTION RT程序源的名称，见 263 页程序运行中的表格。

4.2.7 追溯

使用追溯工具，你可以记录和保存变量值超时的过程（ z.B单元变量、本地变量、系统变量、 I/O 变量）。这使得你可以记录下优化，如：轴线。

你可以设置记录时间，显示多达四通道，选择 trigger 条件，定时调整参数设置，在不同的曲线显示中选择，等。

除了 isochronous 记录外，你可以在代码位置选择记录。 无论何时 ST源文件中的特定点运行程序，这让你可以选择变量数值。

追溯工具的内容在在线帮助中有详细描述。