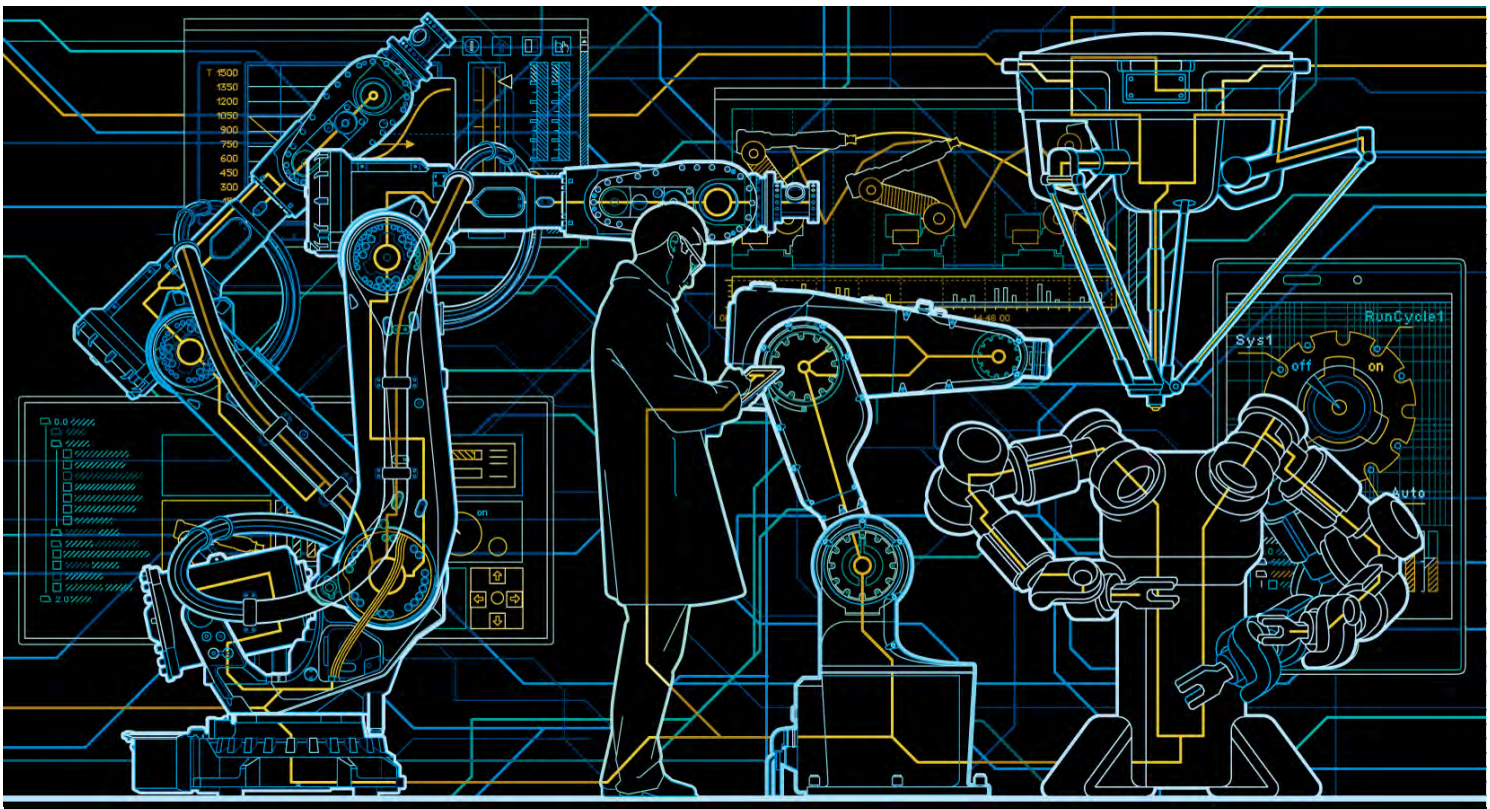


技术参考手册
RAPID语言内核

Power and productivity
for a better world™

ABB

Trace back information:
Workspace R16-1 version a6
Checked in 2016-03-01
Skribenta version 4.6.209

技术参考手册

RAPID语言内核

RobotWare 6.03

文档编号: 3HAC050946-010

修订: B

ABB AB
Robotics Products
Se-721 68 Västerås
瑞典

目表

手册概述

关于本手册

本手册对ABB Robotics机械臂编程语言RAPID给出了正式说明。

本手册的阅读对象

本手册适用于有一些编程经验的人员，例如，机械臂程序员。

操作前提

读者应具备一定的编程经验，且学过操作员手册 - *Introduction to RAPID*。

参考信息

参考文档	文档编号
操作员手册 - *Introduction to RAPID*	*3HAC029364-010*
操作员手册 - 带 *FlexPendant* 的 *IRC5*	*3HAC050941-010*
技术参考手册 - *RAPID*指令、函数和数据类型	*3HAC050917-010*
技术参考手册 - *RAPID*语言概览	*3HAC050947-010*
技术参考手册 - 系统参数	*3HAC050948-010*

修订版

版本号	描述
-	随 RobotWare 6.0 发布。
A	随 RobotWare 6.01 发布。 • 为第*119*页的*Nostepin*模块补充了信息。
B	随 RobotWare 6.03 发布。 • 细微纠正。

如何查阅本手册

书面约定

程序示例通常以输出文件或打印机输出形式呈现，区别于以如下形式呈现在FlexPendant示教器上的程序：

- FlexPendant示教器中隐藏的特定控制字码，如表明程序开始和结束的字码；
- 以标准格式打印出来的数据声明和程序声明，如*VAR num reg1*。

在本手册说明中，所有指令、函数和数据类型的名称都要用等宽字体表示，如TPWrite。变量、系统参数和功能的名称用斜体表示。所列事件号示例中的注释不译（即使翻译本手册时也一样不译）。

语法规则

用简化语法和形式语法对指令和函数进行说明。若您是用FlexPendant示教器编程，则由于机械臂自身能保证所用语法的正确性，因此通常只需了解简化语法。

简化语法示例

如下为一种含指令TPWrite的简化语法示例。

```
TPWrite String [\Num] | [\Bool] | [\Pos] | [\Orient] [\Dnum]
```

- 强制性参数未括上括号。
- 用方括号[]将可选参数括起来，但可忽略这些参数。
- 互相排斥的参数不能同时存在于同一指令中，在同一指令中就要用竖线|隔开。
- 用波形括号{ }将可重复任意次的参数括起来。

上述示例采用了如下参数：

- String为强制性参数。
- Num、Bool、Pos、Orient和Dnum为可选参数。
- Num、Bool、Pos、Orient和Dnum互相排斥。

形式语法示例

```
TPWrite
  [String ':='] <expression (IN) of string>
  ['\'Num':=' <expression (IN) of num> ] |
  ['\'Bool':=' <expression (IN) of bool> ] |
  ['\'Pos':=' <expression (IN) of pos> ] |
  ['\'Orient ':=' <expression (IN) of orient> ]
  ['\' Dnum':=' <expression (IN) of dnum]';'
```

- 方括号[ ]中的文字可忽略。
- 互相排斥的参数不能同时存在于同一指令中，在同一指令中就要用竖线|隔开。
- 用波形括号{ }将可重复任意次的参数括起来。
- 用单引号（两个撇号''）将为获得正确语法而写出的符号括起来。
- 参数（斜体）数据类型和其他字符括到尖括号中< >。欲知更多详细信息，请参见程序参数说明。

用特殊语法EBNF编写语言和特定指令的基本元素。规则不变，而且还有所增加。

- 符号::=等同于被定义为。
- 至于尖角括号<>中的文字，将另起一行单独说明。

下一页继续

示例

```
GOTO <identifier> ';'
  <identifier> ::= <ident> | <ID>
  <ident> ::= <letter> {<letter> | <digit> | '_'}
```

此页刻意留白

1 简介

1.1 设计目标

RAPID语言概念

RAPID语言支持分层编程方案。在分层编程方案中，可为特定机器人系统安装新程序、数据对象和数据类型。该方案能对编程环境进行自定义（扩展编程环境的功能），并获得RAPID编程语言的充分支持。

此外，RAPID语言还带有若干强大功能：

- 对任务和模块进行模块化编程
- 无返回值程序和有返回值程序
- 类型定义
- 变量、永久数据对象、常量
- 算术
- 控制结构
- 步退执行支持
- 错误恢复
- 撤销执行支持
- 中断处理
- 占位符

1.2 语言摘要

任务与模块

RAPID应用被称作一项任务。一项任务包括一组模块。一个模块包含一组数据和程序声明。任务缓冲区用于存放系统当前在用（在执行、在开发）的模块。

RAPID语言区分了任务模块和系统程序模块。一个任务模块被视作任务/应用的一部分，而一个系统程序模块被视作系统的一部分。系统程序模块在系统启动期间自动加载到任务缓冲区，旨在（预）定义常用的系统特定数据对象（工具、焊接数据、移动数据等）、接口（打印机、日志文件..）等。

虽然（除了系统程序模块外）单个任务模块通常包含小应用，但较大应用可能包含主任务模块，主任务模块反过来又引用一项或多项其他任务模块所含的程序和/或数据。

一项任务模块包含任务的入口无返回值程序。运行任务实际上表示执行该入口程序。入口程序无法具备参数。

程序

有三类程序：有返回值程序、无返回值程序和软中断程序。

- 有返回值程序将返回特定类型的值，用于表达式中。
- 无返回值程序不返回任何值，用于语句中。
- 软中断程序提供了中断响应手段。软中断程序可与特定中断关联起来，随后，在发生该中断的情况下，被自动执行。

用户程序

用户（定义）程序利用RAPID声明来进行定义。

RAPID程序声明指定了程序名称、程序参数、数据声明、语句，可能也指定了回退处理器和/或错误处理器和/或撤销处理器。

预定义程序

预定义程序由系统提供，一直可供使用。

有两类预定义程序：内置程序和安装程序。

- 内置程序（如有返回值运算程序一样）属于RAPID语言的一部分。
- /安装程序是用于控制机械臂、夹具、传感器等的、与应用或设备有关的程序。

> **注意**
>
> 从用户角度讲，内置程序和安装程序并无区别。

数据对象

有四种数据对象：常量、变量、永久数据对象和参数。

- 永久（数据对象）可描述为“永久”变量。在两次会话之间，永久变量将保持值。
- 在每次新会话开始时，即，当加载模块时（模块变量）或调用程序时（程序变量），将失去（重新初始化）变量值。

数据对象可呈结构化（记录），也可呈维度化（数组、矩阵等）。

下一页继续

语句

语句可为简单语句或复合语句。反过来，复合语句又可能包含其他语句。标签是“空操作”语句，可用于定义在程序中指定的（goto）位置。语句将被接连执行，除非goto、return、raise、exit、retry或trynext语句、或发生中断或错误，造成从另一点继续执行。

赋值语句将改变变量、永久数据对象或参数的值。

在将任何调用参数与无返回值程序的相应参数关联起来后，无返回值程序调用将引起无返回值程序被执行。RAPID语言支持对无返回值程序名称进行后期绑定。

goto语句会导致程序在标签指定位置继续执行。

return语句将终止程序的求值。

raise语句用于发出和传递错误。

exit语句将终止任务的求值。

connect语句用于指定中断编号，并将中断编号与软中断（中断服务）程序关联起来。

retry和trynext语句用于在错误发生后，重新开始求值。

if和test语句用于选择。if语句能够允许基于条件值，选择语句表。test语句将选择一组（或不选择）语句表，具体取决于表达式的值。

for和while语句用于迭代。只要循环变量处于指定值域内，for语句就会重复执行语句表的求值。在每一迭代结束时，将（以可选增量）更新循环变量。只要满足条件，while语句便会重复执行语句表的求值。在每一迭代开始时，会对条件进行求值和核实。

步退执行

RAPID语言支持对语句进行逐步步退执行。在RAPID程序开发期间，步退执行对调试、测试和调节十分有用。RAPID无返回值程序可能包含回退处理器（语句表），回退处理器定义了无返回值程序的步退执行“行为”。

错误恢复

发生运行中的错误会造成正常程序执行被中止。反过来，可转由用户提供的错误处理器来进行控制。任何程序声明均可能包括错误处理器。错误处理器能取得有关错误的信息，并可能采取一些行动来对错误做出响应。如若合适，错误处理器可将错误返回给引起出错的语句来进行控制（retry）或返回给引起出错的语句的后一个语句来进行控制（trynext）或返回给程序调用点来进行控制。如果无法进一步执行程序，那么，错误处理器至少能确保平稳地中止了任务。

撤销执行

通过将程序指针移到程序外，可以在任意点中止程序。在一些情况下，当程序正在执行特定敏感程序时，不适合终止程序。采用撤销处理器，或许可以保护这些敏感程序，使其不会出现意外的程序复位。如果程序被中止，那么，将自动执行撤销处理器。该代码一般应执行清理动作，比如，关闭文件。

下一页继续

中断

中断发生原因在于用户定义（中断）条件变为真。中断不像错误，中断与特定代码段的执行无直接关联（不同步）。发生中断会引起正常程序执行被中止，可转由软中断程序来进行控制。为响应中断而采取必要动作后，软中断程序可从中断点重新开始执行程序。

数据类型

任何RAPID对象（值、表达式、变量、有返回值程序等）都具备一个数据类型。数据类型可为内置型，或为安装型（对照安装程序），还可为用户定义型（在RAPID语言中定义）。内置型数据为RAPID语言的一部分，而各站点之间的安装型数据集或用户定义型数据集可能不同。

> **注意**
>
> 从用户角度讲，内置型数据、安装型数据和用户定义型数据并无区别。

数据还可分为三种类型：原子型、记录型和别名型。原子型的定义必须为内置型或安装型，而记录型或别名型也可为用户定义型。

- 原子数据类型被命名为“原子型”是因为这种数据类型未根据任何其他类型进行定义，无法分成各部分或各分量。
- 记录型由一组命名的有序分量累积而来。
- 别名型按定义来讲等同于另一类型。别名型能够对数据对象进行分类。

除了原子型、记录型和别名型分类外，每一类型还具备一个值类型。存在三种值类型：值型、非值型和半值型。

- 值型对象代表一些形式的值，比如，*3.55*或*John Smith*）。
- 非值（型）对象代表一些物理对象或逻辑对象的隐藏描述或密封描述，比如一个文件。
- 半值对象有两类，一种为基本非值型，另一种为关联值型，关联值型可用于表示非值型的一些性质。

内置数据类型

内置原子型数据有`bool`、`num`、`dnum`和`string`。

- `bool`为枚举类型，其值为真或假，提供了一种开展逻辑计算和关联计算的方式。
- `num`型支持精确算术计算和近似算术计算。
- `string`型表示字符序列。

内置记录型数据有`pos`、`orient`和`pose`。

- `pos`型表示空间位置（矢量）。
- `orient`型表示在空间中的方位。
- `pose`型表示坐标系（位置/方位组合）。

内置别名型数据有`errnum`和`intnum`。`Errnum`和`intnum`均为`num`的别名，用于表示错误和中断编号。

将利用算术运算符、关系运算符和逻辑运算符以及预定义程序来定义内置型数据的对象的运算。

下一页继续

安装数据类型

安装型数据的概念是，能够使用适当参数类型来使用安装程序。安装型数据可为原子型、记录型、或为别名型。

用户定义数据类型

用户定义型数据能更轻松地自定义应用程序，也能够编写更易读取的RAPID程序。

占位符

占位符概念支持对RAPID程序进行结构化生成和修改。离线编程工具和在线编程工具可利用占位符来临时表示RAPID程序的“未定义”部分。含占位符的程序在语法上是正确的，可加载到任务缓冲区（也可从任务缓冲区保存）。如果RAPID程序中的占位符未引起语义错误（参见第17页的错误分类），那么，该程序甚至可被执行，但遇到的占位符会引起执行错误（参见第17页的错误分类）。

1.3 语法表示法

上下文无关语法

RAPID语言的上下文无关语法利用巴科斯范式的变体EBNF来表示。

- 粗体大写字表示保留字和占位符，比如，**WHILE**。
- 引用字符串表示其他终止符，比如，'+'。
- 括入尖角括号的字符串表示语法类、非终止符，比如<constant expression>。
- 符号::=系指被定义为，比如，`<dim> ::= <constant expression>`
- 终止符和/或非终止符列表表示一个序列，比如，`GOTO<identifier> ';'`
- 方括号中括的是可选项。这些可选项可能不会出现，也可能出现一次，比如，`<return statement> ::= RETURN [ <expression> ] ';'`
- 竖线将替代项分隔开，比如，`OR | XOR`
- 大括号中括的是重复项。这些重复项可能不会出现，也可能出现多次，比如，`<statement list> ::= { <statement> }`
- 圆括号用于将各概念分层次地组到一起，比如，`(OR|XOR)<logical term>`

1.4 错误分类

错误类型

按照检测时间，错误可分为静态错误或执行错误。

静态错误

静态错误的检出时间为模块加载到任务缓冲区时（参见第115页的任务模块）或程序修改后、执行前。

错误类型	示例	示例说明
词汇错误，非法词法单元	`b := 2E52786;`	指数超出范围
语法错误，违反语法规则	`FOR i 5 TO 10 DO`	缺FROM关键字
语义错误，违反语义规则，典型的类型错误	`VAR num a;` `a := "John";`	数据类型不匹配
致命（系统资源）错误	-	程序太复杂（嵌套）

执行错误

在任务执行期间出现（并检出）执行错误。

- 算法错误，比如，除零
- 输入/输出错误，比如，无此文件或器件
- 致命（系统资源）错误，比如，执行栈溢出

RAPID语言的错误处理器能够恢复非致命执行错误，参见第97页的错误恢复。

此页刻意留白

2 词汇元素

2.1 字符集

定义

RAPID语言的语句采用标准ISO 8859-1（Latin-1）字符集来构建。此外，还将识别换行符、tab及换页控制字符。

描述

```
<character> ::= -- ISO 8859-1 (Latin-1)--
<newline> ::= -- newline control character --
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d |
     e | f
<letter> ::= <upper case letter> | <lower case letter>
<upper case letter> ::=
  A | B | C | D | E | F | G | H | I | J
  | K | L | M | N | O | P | Q | R | S | T
  | U | V | W | X | Y | Z | À | Á | Â | Ã
  | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í
  | Î | Ï | 1) | Ñ | Ò | Ó | Ô | Õ | Ö | Ø
  | Ù | Ú | Û | Ü | 2) | 3) | ß
<lower case letter> ::=
a | b | c | d | e | f | g | h | i | j
| k | l | m | n | o | p | q | r | s | t
| u | v | w | x | y | z | ß | à | á | â | ã
| ä | å | æ | ç | è | é | ê | ë | ì | í
| î | ï | 1) | ñ | ò | ó | ô | õ | ö | ø
| ù | ú | û | ü | 2) | 3) | ÿ
```

2.2 词法单元

定义

RAPID语句是一个序列的词法单元，也被称作标记。RAPID标记包括：

- 标识符中；
- 保留字中；
- 文字
- 分隔符
- 占位符中。
- comments

限制

标记是不可分的。除了字符串文字和注释外，标记中不得出现空格。

必须用一个或多个空格、tab、换页符或换行符将标识符、保留字或数字文字与末尾、相邻标识符、保留字或数字文字隔开。可用一个或多个空格、tab、换页符或换行符来分隔其他标记组合。

2.3 标识符

定义

标识符用于为对象命名。

```
<identifier> ::= <ident> | <ID>
<ident> ::= <letter> {<letter> | <digit> | '_'}
```

限制

标识符最大长度为32字符。

标识符的所有字符均很重要。只在相应大小写字母使用上有区别的各标识符将被视作是相同的标识符。

占位符<ID> （参见第15页的占位符以及第27页的占位符）可用于表示标识符。

2.4 保留字

定义

下列字为保留字。它们在 RAPID语言中都有特殊意义，因此不能用作标识符。

在语法未特别指定的情况下，不得使用保留字。

此外，还有许多预定义数据类型名称、系统数据、指令和有返回值程序也不能用作标识符。

ALIAS	AND	BACKWARD	CASE
CONNECT	CONST	DEFAULT	DIV
DO	ELSE	ELSEIF	ENDFOR
ENDFUNC	ENDIF	ENDMODULE	ENDPROC
ENDRECORD	ENDTEST	ENDTRAP	ENDWHILE
ERROR	EXIT	FALSE	FOR
FROM	FUNC	GOTO	IF
INOUT	LOCAL	MOD	MODULE
NOSTEPIN	NOT	NOVIEW	OR
PERS	PROC	RAISE	READONLY
RECORD	RETRY	RETURN	STEP
SYSMODULE	TEST	THEN	TO
TRAP	TRUE	TRYNEXT	UNDO
VAR	VIEWONLY	WHILE	WITH
XOR			

2.5 数字文字

定义

数字文字表示数值。

```
<num literal> ::=
  <integer> [ <exponent> ]
  | <decimal integer>) [<exponent>]
  | <hex integer>
  | <octal integer>
  | <binary integer>
  | <integer> '.' [ <integer> ] [ <exponent> ]
  | [ <integer> ] '.' <integer> [ <exponent> ]
<integer> ::= <digit> {<digit>}
<decimal integer> ::= '0' ('D' | 'd') <integer>
<hex integer> ::= '0' ('X' | 'x') <hex digit> {<hex digit>}
<octal integer> ::= '0' ('O' | 'o') <octal digit> {<octal digit>}
<binary integer> ::= '0' ('B' | 'b') <binary digit> {<binary digit>}

<exponent> ::= ('E' | 'e') ['+' | '-'] <integer>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d |
     e | f
<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<binary digit> ::= 0 | 1
```

限制

数字文字必须处于ANSI IEEE 754《浮点数算术标准》规定的值域内。

示例

例如：7990 23.67 2E6 .27 2.5E-3 38.

2.6 布尔文字

定义

布尔文字表示逻辑值。

```
<bool literal> ::= TRUE | FALSE
```

2.7 字符串文字

定义

字符串文字是以双引号（"）字符包围的零个以上字符组成的一个序列。

```
<string literal> ::= '"' { <character> | <character code> } '"'
     <character code> ::= '\' <hex digit> <hex digit>
```

可以使用字符代码，则能让我们在字符串文字中纳入不可见字符（二进制数据）。如果应在字符串文字中纳入反斜杠字符或双引号字符，则必须将反斜杠字符或双引号字符书写两次。

示例

```
"A string literal"
"Contains a "" character"
"Ends with BEL control character\07"
"Contains a \\ character"
```

2.8 分隔符

定义

分隔符包含任一下列字符：

{ } ( ) [ ] , . = < > + -* / : ; ! \ ?

分隔符也可包含任一下列复合符号：

:= <> >= <=

2.9 占位符

定义

离线编程工具和在线编程工具可利用占位符来临时表示RAPID程序的“未定义”部分。含占位符的程序在语法上是正确的，可加载到任务缓冲区（也可从任务缓冲区保存）。如果RAPID程序中的占位符未引起语义错误（参见第17页的错误分类），那么，该程序甚至可被执行，但遇到的占位符会引起执行错误（参见第17页的错误分类）。

RAPID语言将识别下列占位符：

占位符	描述
<TDN>	（表示一个）数据类型定义
<DDN>	（表示一个）数据声明
<RDN>	程序声明
<PAR>	参数声明
<ALT>	替代参数声明
<DIM>	数组维度
<SMT>	语句
<VAR>	数据对象引用（变量、永久数据对象或参数）
<EIT>	if语句中的else if子句
<CSE>	test语句的case子句
<EXP>	表达式
<ARG>	过程调用参数
<ID>	标识符

2.10 备注

定义

注释以感叹号（!）开头，以换行符结束。注释决不能包含换行符。

<comment> ::= '!' { <character> | <tab> } <newline>

注释对RAPID代码序列的意义无影响，注释的唯一目的在于向读者说明代码。

每一RAPID注释占一整条源行，可呈现为：

- 类型定义表的一个元素（参见数据声明表）
- 记录分量表的一个元素
- 数据声明表的一个元素（参见第89页的无返回值程序声明）
- 程序声明表的一个元素（参见第116页的模块声明）
- 语句表的一个元素（参见第67页的语句表）

在一个模块中处于最后一个数据声明（参见第39页的数据声明）和第一个程序声明（参见第85页的程序声明）之间的注释将被视作程序声明表的一部分。而处于最后一个数据声明和程序第一个语句之间的注释将被视作语句表（参见第67页的语句表）的一部分。

示例

```
! Increase length
length := length + 5;
IF length < 1000 OR length > 14000 THEN
  ! Out of bounds
  EXIT;
ENDIF
...
```

2.11 数据类型

定义

RAPID数据类型以其名称进行标识，可为内置数据、安装数据或用户定义数据（按RAPID语言定义）。

```
<data type> ::= <identifier>
```

内置数据类型为RAPID语言的一部分，而各站点之间的安装型数据集或用户定义型数据集可能不同。若能够使用适当的参数类型，则安装型数据支持安装程序的使用。用户定义型数据能够为应用程序工程师准备能读懂的易编程应用包。从用户角度讲，内置型数据、安装型数据和用户定义型数据没有区别。

有三种不同类型：

- 第*31*页的*atomic*数据类型
- 第*33*页的*record*数据类型
- 第*35*页的*alias*数据类型

类型定义通过让标识符与数据类型描述关联起来，引入了一个别名或记录。类型定义可用占位符<TDN>表示。

```
<type definition> ::=
  [LOCAL] ( <record definition> | <alias definition> )
  | <comment>
  | <TDN>
```

类型定义可出现在模块的开头部分（参见第*115*页的任务模块）。

可选局部命令将数据对象分为局部数据对象或者全局数据对象（参见第*42*页的数据对象的范围规则）。

示例

记录定义

```
LOCAL RECORD object
  num usecount;
  string name;
ENDRECORD
```

别名定义

```
ALIAS num another_num;
```

占位符定义

```
<TDN>
```

2.12 数据类型的范围规则

定义

类型定义范围表示类型的显示范围，取决于声明的位置和上下文。

预定义类型的范围包括任何RAPID模块。

用户定义类型常常定义在模块内部。下列范围规则对模块类型定义有效：

- 局部模块类型定义的范围包括其所处模块。
- 全局模块类型定义的范围还包括任务缓冲区的其他模块。
- 在范围之内，模块类型定义隐藏了同名的预定义类型。
- 在范围之内，局部模块类型定义隐藏了同名的全局模块类型。
- 同一模块中声明的两个模块对象不可同名。
- 任务缓冲区中，两个不同模块中声明的两个全局对象不可同名。

2.13 atomic数据类型

定义

atomic数据类型被命名为原子型是因为它们未按其他类型来定义，该数据类型不可分成各个部分或各个分量。原子型的内部结构（实现）是隐藏的。

内置原子型数据有数字型num和dnum、逻辑型bool以及文本型string。

num型

num对象表示一个数值。num型表示ANSI IEEE 754《浮点数算术标准》指定的域。

在子域-8388607至(+)8388608中，num对象可用于表示整数（精确）值。只要运算元和结果保持在num的整数子域范围内，则算术运算符+、-和*（参见第62页的操作员）将保持整数表示。

num的示例

示例	描述
`VAR num counter;`	变量的声明
`counter := 250;`	num文字使用

dnum型

dnum对象表示一个数值。dnum型表示ANSI IEEE 754《浮点数算术标准》指定的域。

在子域-4503599627370496至(+)4503599627370496中，dnum对象可用于表示整数（精确）值。只要运算元和结果保持在dnum的整数子域范围内，则算术运算符+、-和*（参见第62页的操作员）将保持整数表示。

dnum的示例

示例	描述
`VAR dnum value;`	变量的声明
`value := 2E+43;`	dnum文字使用

bool型

bool对象表示一个逻辑值。

bool型表示二值逻辑的域，即，真或假。

bool的示例

示例	描述
`VAR bool active;`	变量的声明
`active := TRUE;`	bool文字使用

string型

string对象表示一个字符串。

下一页继续

string型表示所有序列的图形字符（ISO 8859-1）和控制字符（数字代码范围0 .. 255中的非ISO 8859-1字符）的域。字符串可包括0至80个字符（固定的80字符存储格式）。

string的示例

示例	描述
`VAR string name;`	变量的声明
`name := "John Smith";`	dnum文字使用

2.14 record数据类型

定义

record数据类型为一种带命名有序分量的复合类型。record类型的值为由各分量的值组成的复合值。一个分量可具备atomic型或record型。

内置记录型有pos、orient和pose。可用的安装记录型和用户定义记录型数据集明显不受RAPID规范约束。

记录定义

record型将靠记录定义引入。

```
<record definition> ::=
  RECORD <identifier> <record component list>
  ENDRECORD
<record component list> ::=
  <record component definition> | <record component definition>
       <record component list>
<record component definition> ::=
  <data type> <record component name> ';'
```

例如,

```
RECORD newtype
  num x;
ENDRECORD
```

记录值

record值可用聚合表示法来表示。

下列示例给出了pos记录的聚合值。

```
[ 300, 500, depth ]
```

为各分量赋值

record数据对象的特定分量可用该分量的名称来进行访问。

下列示例为pos变量p1的X分量赋值。

```
p1.x := 300;
```

默认域

除另行指示外，记录型的域为各分量的域的笛卡儿积。

pos型

pos对象表示在3D空间中的矢量（位置）。pos型有三个分量，即，[x, y, z]。

组件	数据类型	描述
x	num	位置的x轴分量
y	num	位置的y轴分量
z	num	位置的z轴分量

下一页继续

pos的示例

示例	描述
VAR pos p1;	变量的声明
p1 := [10, 10, 55.7];	聚合使用
p1.z := p1.z + 250;	分量使用
p1 := p1 + p2;	运算符使用

orient型

orient对象表示在3D空间中的方位（旋转）。orient型有四个分量，即，[q1, q2, q3, q4]。

组件	数据类型	描述
q1	num	第一个四元数分量
q2	num	第二个四元数分量
q3	num	第三个四元数分量
q4	num	第四个四元数分量

四元数表示法是表示空间中的方位的最简洁表示法。可利用这方面的可用预定义有返回值程序来指定替代方位格式（比如，欧拉角）。

orient的示例

示例	描述
VAR orient o1;	变量的声明
o1 := [1, 0, 0, 0];	聚合使用
o1.q1 := -1;	分量使用
o1 := Euler(a1,b1,g1);	有返回值程序使用

pose型

pose对象表示在3D空间中的3D坐标系。pose型有两个分量，即，[trans, rot]。

组件	数据类型	描述
trans	pos	平移原点
rot	orient	旋转

pose的示例

示例	描述
VAR pose p1;	变量的声明
p1 := [[100, 100, 0], o1];	聚合使用
p1.trans := homepos;	分量使用

2.15 alias数据类型

定义

alias数据类型被定义为等同于另一种类型。alias类型提供一种对象分类手段。系统可采用alias分类来查找和显示与类型相关的对象。

alias类型由alias定义引入。

```
<alias definition> ::=
  ALIAS <type name> <identifier> ';'
```

注意

一个alias类型不可在另一个alias类型基础上进行定义。内置alias类型有errnum和intnum - 两者均是num的alias。

alias的示例

示例	描述
`ALIAS num newtype;`	newtype类型为num的别名。
`CONST level low := 2.5;` `CONST level high := 4.0;`	alias类型level的使用（num的别名）

errnum类型

errnum类型为num的alias，用于表示错误编号。

intnum类型

intnum类型为num的alias，用于表示中断编号。

2.16 数据类型的值类型

定义

针对对象数据类型和对象值之间的关系，可将数据类型归为下列中的任一种：

- 值数据类型
- 非值（专用）数据类型
- 半值数据类型

基本值型数据为内置原子型num、dnum、bool和string。所有分量均为值型的record类型，本身也是个值型，比如，内置型pos、orient和pose。在值型基础上定义的alias型本身也是个值型，比如，内置型errnum和intnum。

至少带一个半值分量并且所有其他分量为值型的的记录型数据本身是一个半值型。在半值型基础上定义的alias型数据本身是一个半值型。

所有其他型数据为非值型，比如，至少带一个非值分量的record型以及在非值型基础上定义的alias型。

数组的值类型与元素值类型相同。

值数据类型

值型对象仅被视作表示一些形式的“值”（比如，5、[10, 7, 3.25]、"John Smith"、TRUE）。相反，非值（型）对象表示一些物理对象或逻辑对象的隐藏描述/密封描述（描述符），比如，iodev（文件）类型。

非值数据类型

仅可采用安装程序（“方法”）来修改非值对象的内容（“值”）。非值对象在RAPID程序中仅可用作参数var或ref的调用参数。

比如，非值对象logfile的使用

```
VAR iodev logfile;
...
! Open logfile
Open "flp1:LOGDIR" \File := "LOGFILE1.DOC ", logfile;
...
! Write timestamp to logfile
Write logfile, "timestamp = " + GetTime();
```

半值数据类型

半值对象很特殊。半值对象有两种类型，一种为“基本”非值型，一种为关联（值）型，关联（值）型可用于表示非值型的一些性质。在值上下文中应用时（见下表），RAPID语言将半值对象视作值对象，在其他情况下视作非值对象。在半值型基础上进行的读取或更新（值）操作的语义（意义/结果）由类型本身决定。

比如，在值上下文中半值对象sig1（signaldi的关联型为num）的使用。

```
VAR signaldi sig1;
...
! use digital input sig1 as value object
IF sig1 = 1 THEN
  ...
...
! use digital input sig1 as non-value object
```

下一页继续

```
IF DInput(sig1) = 1 THEN
  ...
```

请注意，半值对象（类型）可拒绝“按值”读取或更新。

比如，由于sig1表示一项输入器件，因此，sig1拒绝下列赋值。

```
VAR signaldi sig1;
...
sig1 := 1;
```

对象使用与数据类型的值类型的可能组合和不可能组合

下表给出了对象使用与数据类型的值类型的哪些组合可能合法（表中的“X”），哪些组合是不可能的或非法的（表中的“-”）。

对象声明	值	非值	半值
常量	X	-	不适用
永久数据对象	X	-	不适用
进行初始化的变量	X	-	不适用
不进行初始化的变量	X	X	X
程序参数：in	X	-	-
程序参数：var	X	X	X
程序参数：pers	X	-	-
程序参数：ref（仅适用于安装程序）	X	X	X
程序参数：inout var	X	-	-
程序参数：inout pers	X	-	-
有返回值程序的返回值	X	-	-

对象引用	值	非值	半值
赋值[i]	X	-	X [ii]
赋值	X	X [iii]	X iii
赋值[iv]	X	-	X ii

i 有关目标的更多信息，请参见第69页的赋值语句和第78页的Connect语句。
ii 采用关联型（值）。
iii 参数var或ref的调用参数
iv 表达式中采用的对象

2.17 Equal型

定义

如果两个对象的结构（阶数、维度和分量数）相同并且满足下列任一条件，那么，这两个对象的类型等同

- 两个对象具备同样的类型名（所包含的任意别名类型名称首先被定义类型所替代）。
- 其中一个对象为一个聚合（数组或记录），（所有）相应元素/分量的类型都等同。
- 其中一个对象为值型，另一个对象为半值型，第一个对象的类型和半值对象的关联型等同。请注意，仅在值上下文中，这点方有效。

2.18 数据声明

定义

有四种数据对象：

- 常量，CONST
- 变量，VAR
- 永久数据对象，PERS
- 参数

除了预定义数据对象（参见第41页的预定义数据对象）和循环变量（参见第81页的For语句）外，必须对所有数据对象进行声明。数据声明通过将标识符与数据类型关联起来，引入了一个常量、一个变量或一个永久数据对象。有关参数声明的信息，请参见第86页的参数声明。

可用占位符<DDN>来表示数据声明。

```
<data declaration> ::=
[LOCAL] ( <variable declaration> | <persistent declaration> |
     <constant declaration> )
| TASK ( <variable declaration> | <persistent declaration>
| <comment>
| <DDN>
```

关于永久数据对象

永久（数据对象）可描述为一个“永久”变量。在每一新会话开始时，即，在模块加载时（模块变量）或程序调用（程序变量）时，将失去（重新初始化）变量值，而在各会话之间，永久数据对象将保持值。实现这点的方式在于，让永久数据对象的值更新，会自动让永久数据对象声明的初始化值更新。当模块（或任务）被保存时，任何永久数据对象声明的初始化值将反映出该永久数据对象的当前值。此外，永久数据对象保存在系统公共“数据库”，可供控制系统其他部分访问（更新、引用）。

声明及可访问性

数据声明可出现在模块（参见第115页的任务模块）和程序（参见第85页的程序声明）的开头部分。

可选局部命令将数据对象分为局部数据对象或全局数据对象（参见第42页的数据对象的范围规则）。请注意，仅可在模块等级（不可在程序内部）采用局部命令。

可选任务命令将永久数据对象和变量数据对象归入与系统全局对象相反的任务全局对象。在范围规则中，两种全局类型之间无区别。

但任务全局永久数据对象的当前值通常是任务的特有值，不被其他任务共享。如果不同任务中的系统全局永久数据对象以同样的名称和类型进行声明，那么这些系统全局永久数据对象将共享当前值。

仅在安装共享模块中，将变量声明为任务全局对象才会有效。加载模块或安装模块中的系统全局变量是任务特有的，不供其他任务共享。

> **注意**
>
> 仅在模块等级（不可在程序内部）才可采用任务命令。

下一页继续

示例

示例	描述
`LOCAL VAR num counter;`	变量的声明
`CONST num maxtemp := 39.5;`	常量的声明
`PERS pos refpnt := [100.23, 778.55, 1183.98];`	永久数据对象的声明
`TASK PERS num lasttemp := 19.2;`	永久数据对象的声明
`<DDN>`	声明占位符

2.19 预定义数据对象

定义

预定义数据对象由系统提供，一直可供使用。预定义数据对象自动进行声明，可从任何模块对其进行引用。请参见第133页的内置数据对象。

2.20 数据对象的范围规则

定义

数据对象的范围表示对象的显示范围，取决于其声明的位置和上下文。

预定义数据对象的范围包括任何RAPID模块。

模块数据对象

在程序外声明的数据对象被称作模块数据对象（模块变量、模块常量或模块永久数据对象）。下列范围规则对模块数据对象有效：

- 局部模块数据对象的范围包括其所处模块。
- 全局模块数据对象的范围还包括任务缓冲区的其他模块。
- 在范围之内，模块数据对象隐藏了同名的预定义对象。
- 在范围之内，局部模块数据对象隐藏了同名的全局模块对象。
- 同一模块中声明的两个模块对象不可同名。
- 任务缓冲区中，两个不同模块中声明的两个全局对象不可同名。
- 全局数据对象和模块不可共享同一名称。

程序数据对象

在程序内声明的数据对象被称作程序数据对象（程序变量、程序常量）。请注意，本上下文中的程序数据对象概念也包括程序参数（参见第*86*页的参数声明）。

下列范围规则对程序数据对象有效：

- 程序数据对象的范围包括其所处程序。
- 在范围之内，程序数据对象隐藏了同名的预定义对象或用户定义对象。
- 同一程序中声明的两个程序数据对象不可同名。
- 程序数据对象不可与同一程序中声明的标签同名。
- 有关程序和任务模块的信息，请参见第*85*页的程序声明和第*115*页的任务模块。

2.21 存储类别

定义

数据对象的存储类别决定了系统为数据对象分配内存和解除内存分配的时间。数据对象的存储类别本身取决于数据对象的种类及其声明的上下文，既可为静态存储，也可为易失存储。

常量、永久数据对象和模块变量为静态存储。当声明对象的模块（参见第115页的任务模块）被加载后，将分配储存静态数据对象的值所需的内存。这意味着，为永久数据对象或模块变量分配的值将一直保持不变，直至下一次赋值。

程序变量（以及参数，请参见第86页的参数声明）属于易失存储类。在调用含变量声明的程序后，将首次分配储存易失对象的值所需的内存。随后，在返回程序调用器时，将解除内存分配。这也就是说，在程序调用前，程序变量的值一直都不明确，且在程序执行结束时，常常会遗失该值（即，该值变为不明确）。

在递归程序调用（程序直接或间接调用自身）链中，针对“同一程序变量”，各个程序实例均收到了自己的内存位置 - 即，生成了含相同变量的若干实例。

2.22 变量声明

定义

变量由变量声明引入。

```
<variable declaration> ::=
  VAR <data type> <variable definition> ';'
<variable definition> ::=
  <identifier> [ '{' <dim> { ',' <dim> } '}' ] [ ':=' <constant
       expression> ]
<dim> ::= <constant expression>
```

例如,

```
VAR num x;
VAR pos curpos := [b+1, cy, 0];
```

如第39页的数据声明所述，变量可声明为局部变量、任务变量或系统全局变量。

声明数组变量

通过在声明中添加维度信息，可为任一类变量（包括安装型）赋予一个数组（1阶、2阶或3阶）形式。维度表示法必须表示一个大于0的整数值（参见第31页的num型）。

例如,

```
! pos (14 x 18) matrix
VAR pos pallet{14, 18};
```

声明值型变量

值型变量（参见第36页的数据类型的值类型）可进行初始化（被赋予初始值）。用于对变量进行初始化的常量表达式的数据类型必须等同于变量类型。

例如,

```
VAR string author_name := "John Smith";
VAR pos start := [100, 100, 50];
VAR num maxno{10} := [1, 2, 3, 9, 8, 7, 6, 5, 4, 3];
```

非初始化变量的初始值

非初始化变量（或变量分量/元素）收到下列初始值

数据类型	初始值
num（或num的别名）	0
dnum（或dnum的别名）	0
bool（或bool的别名）	FALSE
string（或string的别名）	""
安装atomic型	所有数据位0'ed

2.23 永久数据对象声明

定义

永久数据对象由永久数据对象声明来引入。请注意，仅可在模块等级（不可在程序内）声明永久数据对象。可为永久数据对象赋予任何值数据类型。

```
<persistent declaration> ::=
PERS <data type> <persistent definition> ';'
<persistent definition> ::=
<identifier> [ '{' <dim> { ',' <dim> } '}' ] [':=' <literal
     expression>]
```

> **注意**
>
> 只有系统全局永久数据对象的文字表达式可忽略。

例如，

```
PERS num pcounter := 0;
```

声明数组永久数据对象

通过在声明中添加维度信息，可为任一类（包括安装型）永久数据对象赋予一个数组（1阶、2阶或3阶）形式。维度表示法必须表示一个大于0的整数值（参见第31页的 *num*型）。

例如，

```
! 2 x 2 matrix
PERS num grid{2, 2} := [[0, 0], [0, 0]];
```

永久数据对象的初始值

如第39页的数据声明所述，永久数据对象可声明为局部对象、任务全局对象或系统全局对象。局部永久数据对象和任务全局永久数据对象必须进行初始化（被赋予一个初始值）。对系统全局永久数据对象而言，可省去初始值。用于对永久数据对象进行初始化的文字表达式的数据类型必须等同于永久数据对象类型。请注意，永久数据对象的值更新，会自动引起永久数据对象声明（若未被省去）的初始化表达式更新。

例如，

```
MODULE ...
  PERS pos refpnt := [0, 0, 0];
  ...
  refpnt := [x, y, z];
  ...
ENDMODULE
```

如果执行时变量x、y和z的值分别为100.23、778.55和1183.98并且模块被保存，那么，被保存的模块将如下所示：

```
MODULE ...
  PERS pos refpnt := [100.23, 778.55, 1183.98];
  ...
  refpnt := [x, y, z];
  ...
ENDMODULE
```

下一页继续

非初始化永久数据对象的初始值

无初始值的永久数据对象（或永久数据对象分量/元素）收到下列初始值

数据类型	初始值
num（或num的别名）	0
dnum（或dnum的别名）	0
bool（或bool的别名）	FALSE
string（或string的别名）	""
安装atomic型	所有数据位0'ed

2.24 常量声明

定义

常量代表一个静态值，将由常量声明引入。常量的值无法修改。常量可被赋予任何值数据类型。

```
<constant declaration> ::=
  CONST <data type> <constant definition> ';'
<constant definition> ::=
  <identifier> [ '{' <dim> { ',' <dim> } '}' ] ':=' <constant
       expression>
<dim> ::= <constant expression>
```

例如，

```
CONST num pi := 3.141592654;
CONST num siteno := 9;
```

声明数组常量

通过在声明中添加维度信息，可为任一类（包括安装型）常量赋予一个数组（1阶、2阶或3阶）形式。维度表示法必须表示一个大于0的整数值（参见第*31*页的*num*型）。常量值的数据类型必须等同于常量类型。

例如，

```
CONST pos seq{3} := [[614, 778, 1020], [914, 998, 1021], [814, 998,
     1022]];
```

此页刻意留白

3 表达式

3.1 表达式介绍

定义

表达式指定了对一个值的求值。表达式可表示为占位符<EXP>。

```
<expression> ::=
  <expr>
  | <EXP>
<expr> ::= [ NOT ] <logical term> { ( OR | XOR ) <logical term> }
<logical term> ::= <relation> { AND <relation> }
<relation> ::= <simple expr> [ <relop> <simple expr> ]
<simple expr> ::= [ <addop> ] <term> { <addop> <term> }
<term> ::= <primary> { <mulop> <primary> }
<primary> ::=
  <literal>
  | <variable>
  | <persistent>
  | <constant>
  | <parameter>
  | <function call>
  | <aggregate>
  | '(' <expr> ')'
<relop> ::= '<' | '<=' | '=' | '>' | '>=' | '<>'
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/' | DIV | MOD
```

求值顺序

相关运算符的相对优先级决定了求值的顺序。圆括号能够覆写运算符的优先级。上述规则暗示了如下运算符优先级：

优先级	操作员
最高	* / DIV MOD
	+ -
	< > <> <= >= =
	AND
最低	XOR OR NOT

先求解优先级较高的运算符的值，然后再求解优先级较低的运算符的值。优先级相同的运算符则按从左到右的顺序挨个求值。

示例表达式	求值顺序	备注
a + b + c	(a + b) + c	从左到右的规则
a + b * c	a + (b * c)	*高于+
a OR b OR c	(a OR b) OR c	从左到右的规则
a AND b OR c AND d	(a AND b) OR (c AND d)	AND高于OR
a < b AND c < d	(a < b) AND (c < d)	<高于AND

下一页继续

二元运算符是取两个运算元的运算符（即，+、-、*等）。二元运算符的左运算元的求值优先于右运算元。请注意，涉及AND和OR运算符的表达式的求值将进行优化，从而在左运算元求值后，能确定运算结果的情况下，不对表达式的右运算元求值。

3.2 常量表达式

定义

在数据声明中，用常量表达式来表示值。

```
<constant expression> ::= <expression>
```

注意

常量表达式是一般表达式的特殊形式。在任何等级均不可包含变量、永久数据对象或有返回值程序调用！

示例

```
CONST num radius := 25;
CONST num pi := 3.141592654;
! constant expression
CONST num area := pi * radius * radius;
```

3.3 文字表达式

定义

文字表达式用于表示永久数据对象声明的初始化值。

```
<literal expression> ::= <expression>
```

文字表达式是一般表达式的特殊形式，仅可包含各成分依次为文字表达式的单个聚合或单个文字值（数值文字前带有+或-）。

示例

```
PERS pos refpnt := [100, 778, 1183];
PERS num diameter := 24.43;
```

3.4 条件表达式

定义

条件表达式用于表示逻辑值。

```
<conditional expression> ::= <expression>
```

条件表达式是一般表达式的特殊形式。结果类型必须为bool型（真或假）。

示例

```
counter > 5 OR level < 0
```

3.5 文字

定义

文字是用于表示特定数据类型的一个恒定值的词法单元（不显示）。

```
<literal> ::=
  <num literal>
  | <string literal>
```

示例

示例	描述
0.5, 1E2	数字文字
"limit"	string文字
TRUE	bool文字

3.6 变量

定义

根据变量的类型和维度，最多可以三种不同方式来引用变量。变量引用可能意味着整个变量、变量的一个元素（数组）、或变量的一个分量（记录）。

```
<variable> ::=
  <entire variable>
  | <variable element>
  | <variable component>
```

按照上下文，变量引用表示变量的位置或值。

整个变量

将以变量标识符来引用整个变量。

```
<entire variable> ::= <ident>
```

如果变量为一个数组，那么引用表示引用了所有元素。如果变量为一个记录，那么引用表示引用了所有分量。

> **注意**
>
> 占位符<ID>（参见第21页的标识符）无法用于表示整个变量。

```
VAR num row{3};
VAR num column{3};
...
! array assignment
row := column;
```

变量元素

将以元素的索引号来引用数组变量元素。

```
<variable element> ::= <entire variable> '{' <index list> '}'
<index list> ::= <expr> { ',' <expr> }
```

索引表达式必须表示为一个大于0的整数值（参见第31页的num型）。索引值1是指选择数组的第一个元素。索引值不可有悖于声明的维度。索引表中的元素数必须与数组的声明阶数（1、2或3）拟合。

示例	描述
`column{10}`	引用相关栏第十项元素
`mat{i * 10, j}`	引用矩阵元素

变量分量

将以分量名称引用记录变量分量。

```
<variable component> ::= <variable> '.' <component name>
<component name> ::= <ident>
```

> **注意**
>
> 占位符<ID>（参见第21页的标识符）无法用于表示分量名称。

3.7 永久数据对象

定义

永久数据对象引用可意味着引用整个永久数据对象、引用永久数据对象的一个元素（数组）、或引用永久数据对象的一个分量（记录）。

```
<persistent> ::=
  <entire persistent>
  | <persistent element>
  | <persistent component>
```

永久数据对象引用方面的规则与变量引用方面的规则相符，参见第*55*页的变量。

3.8 常量

定义

常量引用可意味着引用整个常量、引用常量的一个元素（数组）、或引用常量的一个分量（记录）。

```
<constant> ::=
  <entire constant>
  | <constant element>
  | <constant component>
```

常量引用方面的规则与变量引用方面的规则相符，参见第55页的变量。

3.9 参数

定义

参数引用可意味着引用整个参数、引用参数的一个元素（数组）、或引用参数的一个分量（记录）。

```
<parameter> ::=
  <entire parameter>
  | <parameter element>
  | <parameter component>
```

参数引用方面的规则与变量引用方面的规则相符，参见第55页的变量。

3.10 聚合体

定义

一个聚合表示一个复合值，该复合值为一个数组值或一个记录值。将由表达式来指定每一聚合项。

```
<aggregate> ::= '[' <expr> { ',' <expr> } ']'
```

示例	描述
`[x, y, 2*x]`	pos聚合
`["john", "eric", "lisa"]`	string数组聚合
`[[ 100, 100, 0 ], [ 0, 0, z ]]`	pos数组聚合
`[[1, 2, 3], [a, b, c]]`	num矩阵（2*3）聚合

聚合的数据类型

将由（必须能由）上下文来确定聚合的数据类型。各聚合项的数据类型必须等同于确定类型的相应聚合项的类型。

在下列示例中，IF子句是非法的，原因在于任何聚合的数据类型都无法由上下文确定。

```
VAR pos p1;
! Aggregate type pos - determined by p1
p1 := [1, -100, 12];
IF [1,-100,12] = [a,b,b] THEN
  ...
```

3.11 函数调用

定义

通过有返回值程序调用，将对特定的有返回值程序进行求值，并且调用中会收到有返回值程序返回的值。有返回值程序可为预定义的或用户定义的。

```
<function call> ::= <function> '(' [ <function argument list> ]
     ')'
<function> ::= <ident>
```

变元

有返回值程序调用的参数将数据传递至所调用的有返回值程序（并且也可从调用的有返回值程序传递数据）。调用参数的数据类型必须等同于有返回值程序的相应参数（参见第86页的参数声明）的类型。参数可为必要参数、可选参数，也可为条件参数，可选参数可省去，但是（当前）调用参数的顺序必须与参数的顺序相同。两个或两个以上的参数可声明为相互排斥，在此情况下，同一参数表最多只能存在其中一个参数。条件参数用于通过程序调用链，支持可选参数的平滑传递。

```
<function argument list> ::=
  <first function argument> { <function argument> }
<first function argument> ::=
  <required function argument>
  | <optional function argument>
  | <conditional function argument>
<function argument> ::=
  ',' <required function argument>
  | <optional function argument>
  | ',' <optional function argument>
  | <conditional function argument>
  | ',' <conditional function argument>
<required function argument> ::=
  [ <ident> ':=' ] <expr>
<optional function argument> ::=
  '\' <ident> [ ':=' <expr>]
<conditional function argument> ::=
  '\' <ident> '?' <parameter>
```

必要参数

将用","来将必要参数与进程（如有）参数分开。可包含参数名，也可省去参数名。

示例	描述
`polar(3.937, 0.785398)`	两项必要参数
`polar(dist := 3.937, angle := 0.785398)`	使用名称

可选参数或条件参数

可选参数或条件参数前面跟有'\'和参数名。必须指定参数名的规范。开关（参见第86页的参数声明）型参数有些特殊，仅用于表示（参数的）存在。因此，开关型参数不包含参数表达式。利用条件语法可以传递开关型参数。

示例	描述
`cosine(45)`	一项必要参数

下一页继续

示例	描述
`cosine(0.785398\rad)`	一项必要参数和一项开关型（可选）参数
`dist(pnt:=p2)`	一项必要参数
`dist(\base:=p1, pnt:=p2)`	一项必要参数和一项可选参数

如果存在（调用有返回值程序的）指定可选参数，那么认为条件参数是'存在'的（参见第86页的参数声明），否则我们只会认为条件参数被"省去"。请注意，指定参数必须为可选参数。

比如，如果存在可选参数b，那么`distance := dist(\base ? b, p);`被解读为`distance := dist(\base := b, p);`，否则会解读为`distance := dist(p);`

因此，有了条件参数这一概念，当处理可选参数的传递时，我们便无需多个程序调用"版本"。

例如，

```
IF Present(b) THEN
  distance := dist(\base:=b, p);
ELSE
  distance := dist(p);
ENDIF
```

可利用一项以上的条件参数来与相互排斥的参数的一项以上替代参数相匹配（参见第86页的参数声明）。在此情况下，它们中最多可存在一项（可引用存在的可选参数）。

举例而言，如果可选参数high和low中最多存在一项，那么有返回值程序`FUNC bool check (\switch on | switch off`可以`check(\on ? high \ off ? low`形式进行调用。

参数表

有返回值程序的参数表（参见第86页的参数声明）为每一参数指定一个访问模式。参数的访问模式对相应调用参数施加了限制，指定了RAPID语言如何传递该调用参数。有关程序参数、访问模式和调用参数限制的完整说明，请参见第85页的程序声明。

3.12 操作员

定义

可用运算符可分为四类

- 乘运算符
- 加运算符
- 关系运算符
- 逻辑运算符

下表指定了各运算符的结果类型和合法运算元类型。请注意，关系运算符=和<>是对数组有效的唯一运算符，将运算符与类型不等同于下述指定类型的运算元结合使用（参见第38页的*Equal*型），将引起类型错误（参见第17页的错误分类）。

乘运算符

运算符	操作	运算元类型	结果类型
*	乘法	num * num	num [i]
*	乘法	dnum * dnum	dnum [i]
*	矢量数乘	num * pos或pos * num	pos
*	矢积	pos * pos	pos
*	旋转连接	orient * orient	orient
/	除法	num / num	num
/	除法	dnum / dnum	dnum
DIV	整数除法	numi DIV num [i]	num
DIV	整数除法	dnumi DIV dnum [ii]	dnum
MOD	整数模运算；余数	numi MOD numi	num
MOD	整数模运算；余数	dnumi MOD dnumii	dnum

i 必须表示一个整数值。
ii dnum必须表示一个正整数值（≥0）。

加运算符

运算符	操作	运算元类型	结果类型
+	加法	num + num	num [i]
+	加法	dnum + num	dnum [i]
+	一目减；保留符号	+num或dnum或+pos	同左 [ii]，[i]
+	矢量加法	pos + pos	pos
+	串连接	string + string	string
-	减法	num - num	num [i]
-	减法	dnum - dnum	dnum [i]
-	一目减；更改符号	-num或-dnum或-pos	同左 [ii], [i]

下一页继续

运算符	操作	运算元类型	结果类型
-	矢量减法	pos - pos	pos

i 只要运算元和结果仍在数值类型的整数子域内，那么就可保留整数（精确）表示法。
ii 收到的结果类型与运算元类型相同。若运算元有一个alias数据类型，则可收到alias“基准”类型（num、dnum或pos）的结果。

关系运算符

运算符	操作	运算元类型	结果类型
<	小于	num < num	bool
<	小于	dnum < dnum	bool
<=	小于等于	num <= num	bool
<=	小于等于	dnum <= dnum	bool
=	等于	任意类型[i]=任意类型	bool
>=	大于等于	num >= num	bool
>=	大于等于	dnum >= dnum	bool
>	大于	num > num	bool
>	大于等于	dnum > dnum	bool
<>	不等于	任意类型 <>任意类型	bool

i 只有数值数据类型。运算元类型必须相等。

逻辑运算符

运算符	操作	运算元类型	结果类型
AND	和	bool AND bool	bool
XOR	异或	bool XOR bool	bool
OR	或	bool OR bool	bool
NOT	否；非	NOT bool	bool

此页刻意留白

4 语句

4.1 语句介绍

定义

利用安装程序（和类型）来支持机械臂应用编程者的特定需求，能够将RAPID语句数限制到最少水平。RAPID语句支持一般的编程需求，但实际当中，没有机械臂模型的特定RAPID语句。语句仅可出现在程序定义之中。

```
<statement> ::=
  <simple statement>
  | <compound statement>
  | <label>
  | <comment>
  | <SMT>
```

简单语句或复合语句

语句可为简单语句，也可为复合语句。复合语句可能又包含其他语句。标签是可用于定义程序中指定（Goto）位置的“空操作”语句。占位符<SMT>可用于表示语句。

```
<simple statement> ::=
  <assignment statement>
  | <procedure call>
  | <goto statement>
  | <return statement>
  | <raise statement>
  | <exit statement>
  | <retry statement>
  | <trynext statement>
  | <connect statement>
<compound statement> ::=
  <if statement>
  | <compact if statement>
  | <for statement>
  | <while statement>
  | <test statement>
```

4.2 语句的终止

定义

复合语句（但简洁if语句除外）以语句的特定关键字结尾。简单语句以分号（;）结结尾。标签以冒号（:）结尾。注释以换行符结尾（参见第28页的备注）。语句终止符被视作语句的一部分。

示例	描述
`WHILE index < 100 DO`	
`! Loop start`	换行符将终止一个注释
`next:`	“:”将终止一个标签。
`index := index + 1;`	“;”将终止赋值语句。
`ENDWHILE`	“endwhile”将终止while语句。

4.3 语句表

定义

零个或零个以上语句组成的一个序列被称作语句表。语句表的各语句将接连被执行，除非goto、return、raise、exit、retry或trynext语句或发生错误/中断造成从另一点继续执行。

```
<statement list> ::= { <statement> }
```

程序和复合语句均包含语句表。没有特定的语句表分隔符。语句表的开头和末尾均按上下文来决定。

示例	描述
`IF a > b THEN`	
`pos1 := a * pos2;`	语句表的开头
`! this is a comment`	
`pos2 := home;`	语句表的末尾
`ENDIF`	

4.4 标签语句

定义

标签是用于定义指定程序位置的“空操作”语句。goto语句（参见第72页的*Goto*语句）会致使从标签位置继续执行。

```
<label> ::= <identifier> ':'
```

例如

```
next:
...
GOTO next;
```

标签的范围规则

下列范围规则对标签有效。

- 标签的范围包括其所处程序。
- 在范围之内，标签隐藏了同名的预定义对象或用户定义对象。
- 同一程序中声明的两个标签不可同名。
- 标签不可与同一程序中声明的程序数据对象同名。

4.5 赋值语句

定义

赋值语句用表达式定义的值去替代变量、永久数据对象或参数（赋值目标）的当前值。赋值目标和表达式必须为同等类型。请注意，赋值目标必须为值数据类型或半值数据类型（参见第36页的数据类型的值类型）。赋值目标可用占位符<VAR>来表示。

```
<assignment statement> ::= <assignment target> ':=' <expression>
     ';'
<assignment target> ::=
  <variable>
  | <persistent>
  | <parameter>
  | <VAR>
```

示例

示例	描述
`count := count +1;`	整个变量的赋值
`home.x := x * sin(30);`	分量赋值
`matrix{i, j} := temp;`	数组元素赋值
`posarr{i}.y := x;`	数组元素/分量
`assignment <VAR> := temp + 5;`	占位符使用

4.6 过程调用

定义

无返回值程序调用将启动一个无返回值程序的求值。在无返回值程序终结后，将继续后续语句的求值。无返回值程序可为预定义的，也可为用户定义的。占位符<ARG>可用于表示未定义的参数。

```
<procedure call> ::= <procedure> [ <procedure argument list> ] ';'
<procedure> ::=
  <identifier>
  | '%' <expression> '%'
<procedure argument list> ::=
  <first procedure argument> { <procedure argument> }
<first procedure argument> ::=
  <required procedure argument>
  | <optional procedure argument>
  | <conditional procedure argument>
  | <ARG>
<procedure argument> ::=
  ',' <required procedure argument>
  | <optional procedure argument>
  | ',' <optional procedure argument>
  | <conditional procedure argument>
  | ',' <conditional procedure argument>
  | ',' <ARG>
<required procedure argument> ::=
  [ <identifier> ':=' ] <expression>
<optional procedure argument> ::=
  '\' <identifier> [ ':=' <expression> ]
<conditional procedure argument> ::=
  '\' <identifier> '?' ( <parameter> | <VAR> )
```

无返回值程序名称

可用标识符（前期绑定）来对无返回值程序（名称）进行静态指定，或者也可在（字符串类型）表达式运行期间（后期绑定）对无返回值程序（名称）进行求值。即便前期绑定应被视作“正常”的无返回值程序调用形式，但有时后期绑定能提供极有效的简洁代码。

下列示例展示了前期绑定与后期绑定的比较。

前期绑定	后期绑定

TEST product_id CASE 1:
  proc1 x, y, z;
CASE 2:
  proc2 x, y, z;
CASE 3:
  ...
	例1：
	% "proc" + NumToStr(product_id, 0) % x, y, z;
	...
	例2：
	VAR string procname {3} := ["proc1", "proc2", "proc3"];
	...
	% procname{product_id} % x, y, z;
	...

下一页继续

在正常情况下，语句中的字符串表达式%<expression>%是无返回值程序名称符合范围规则要求的字符串，但字符串也可具备指定程序位置的封闭表达前缀。

“name1:name2”指定了模块“name1”中的无返回值程序“name2"（请注意，可在该程序中局部声明无返回值程序“name2"）。“name2"指定了其中一个任务模块的全局无返回值程序“name2"，在必须从系统等级（安装内置对象）完成向下调用时，这将非常有用。

后期绑定

请注意，后期绑定仅可用于无返回值程序调用，不可用于有返回值程序调用。

无返回值程序调用的参数表方面的一般性规则恰好与有返回值程序调用的相同。欲知更多详情，请参见第60页的函数调用和第85页的程序声明。

示例	描述
`move t1, pos2, mv;`	无返回值程序调用
`move tool := t1, dest := pos2, movedata := mv;`	以名称
`move \reltool, t1, dest, mv;`	以开关型reltool
`move \reltool, t1, dest, mv \speed := 100;`	以可选型speed
`move \reltool, t1, dest, mv \time := 20;`	以可选型time

通常，将按常规范围规则来对无返回值程序引用求解（绑定），但后期绑定可无需按常规范围规则。

4.7 Goto语句

定义

goto语句会导致程序在标签位置继续执行。

```
<goto statement> ::= GOTO <identifier> ';'
```

注意

goto语句不可造成受语句表控制的情况。

例如,

```
next:
i := i + 1;
...
GOTO next;
```

4.8 Return语句

定义

return语句将终止一项程序的执行，并在合适时，指定一个返回值。一个程序可包含任意数量的return语句。return语句可出现在语句表或程序错误处理器/回退处理器的任意地方以及复合语句的任何层级。在任务的入口（参见第115页的任务模块）程序中执行return语句，将终止任务的求值。在软中断（参见第114页的软中断程序）程序中执行return语句，将从中断点重新开始执行。

```
<return statement> ::= RETURN [ <expression> ] ';'
```

限制

表达式类型必须等同于有返回值程序的类型。无返回值程序和软中断程序中的return语句不得包含return表达式。

例如，

```
FUNC num abs_value (num value)
  IF value < 0 THEN
    RETURN -value;
    ELSE
    RETURN value;
  ENDIF
ENDFUNC

PROC message ( string mess )
  write printer, mess;
  RETURN; ! could have been left out
ENDPROC
```

4.9 Raise语句

定义

raise语句用于明确发出或传递错误。

```
<raise statement> ::= RAISE [ <error number> ] ';'
<error number> ::= <expression>
```

错误编号

包含明确错误编号的raise语句，将发出带该编号的错误。错误编号（参见第97页的错误恢复）表达式必须表示为处于1至90范围内的一个整数值（参见第31页的num型）。包含错误编号的raise语句不得出现在程序的错误处理器中。

不带错误编号的raise语句仅可出现在程序的错误处理器中，也将在传递该错误的程序被调用时，再次发出（重新发出）同一（当前）错误。由于软中断程序仅可被系统调用（作为对中断的响应），所以，错误将从软中断程序传递到系统错误处理器（参见第97页的错误恢复）。

例如，

```
CONST errnum escape := 10;
...
RAISE escape; ! recover from this position
...
ERROR
  IF ERRNO = escape THEN
    RETURN val2;
  ENDIF
ENDFUNC
```

4.10 Exit语句

定义

exit语句用于立即终止任务的执行。

```
<exit statement> ::= EXIT ';'
```

不像从任务的入口程序返回，利用exit语句进行任务终止，还将禁止系统自动重启任务的尝试。

例如，

```
TEST state
  CASE ready:
  ...
  DEFAULT :
  ! illegal/unknown state - abort
  write console, "Fatal error: illegal state";
  EXIT;
ENDTEST
```

4.11 Retry语句

定义

retry语句用于在错误发生后，重新开始执行程序，将最先执行（重新执行）引起错误的语句。

```
<retry statement> ::= RETRY ';'
```

retry语句仅可出现在程序的错误处理器。

例如，

```
...
! open logfile
open \append, logfile, "temp.log";
...
ERROR
  IF ERRNO = ERR_FILEACC THEN
    ! create missing file
    create "temp.log";
    ! resume execution
    RETRY;
  ENDIF
  ! propagate "unexpected" error RAISE; ENDFUNC
  RAISE;
ENDFUNC
```

4.12 Trynext语句

定义

trynext语句用于在错误发生后，重新开始执行程序，将最先执行引起错误的语句的后一个语句。

```
<trynext statement> ::= TRYNEXT ';'
```

trynext语句仅可出现在程序的错误处理器。

例如，

```
...
! Remove the logfile
delete logfile;
...
ERROR
  IF ERRNO = ERR_FILEACC THEN
    ! Logfile already removed - Ignore
    TRYNEXT;
  ENDIF
  ! propagate "unexpected" error
  RAISE;
ENDFUNC
```

4.13 Connect语句

定义

connect语句将分配中断编号，将中断编号指定给一个变量或参数（关联目标），再将其与中断程序关联起来。当（如果）带该特定中断编号的中断随后发生时，系统将调用被关联的中断程序来对中断做出响应。关联目标可表示为占位符<VAR>。

```
<connect statement> ::= CONNECT <connect target> WITH <trap> ';'
<connect target> ::=
  <variable>
  | <parameter>
  | <VAR>
<trap> ::= <identifier>
```

操作前提

connect目标必须为num数据类型（或具备num的别名），必须作为（或表示）一个模块变量（非程序变量）。如果将一个参数用作connect目标，则该参数必须为VAR参数或INOUT/VAR参数，请参见第86页的参数声明。分配的中断编号无法“断开”，也无法与另一个中断程序关联起来。同一关联目标不可与同一软中断程序进行不止一次关联。这意味着，同一connect语句不可被执行一次以上，两个相同的connect语句（关联目标相同且软中断程序相同）中，只有一个connect语句可在一次会话期间被执行。但请注意，同一软中断程序可关联一个以上的中断编号。

例如，

```
VAR intnum hp;
PROC main()
...
CONNECT hp WITH high_pressure;
...
ENDPROC

TRAP high_pressure
  close_valve\fast;
  RETURN;
ENDTRAP
```

4.14 IF语句

定义

IF语句将按一个或多个条件表达式的值，对若干语句表中的一个语句表求值，或不对任何语句表求值。

```
<if statement> ::=
  IF <conditional expression> THEN <statement list>
  {ELSEIF <conditional expression> THEN <statement list> | <EIT>
       }
  [ ELSE <statement list> ]
  ENDIF
```

条件表达式将连续进行求值，直至其中一个求值为真。然后，将执行相应的语句表。如果没有任何条件表达式求值为真，那么将执行（可选）else子句。可用占位符<EIT>来表示未定义的elseif子句。

例如，

```
IF counter > 100 THEN
  counter := 100;
ELSEIF counter < 0 THEN
  counter := 0;
ELSE
  counter := counter + 1;
ENDIF
```

4.15 简洁IF语句

定义

除了一般的结构化if语句外（参见第79页的*IF*语句），RAPID语言还提供了一种替代的简洁if语句。如果条件表达式求值为真，那么，简洁if语句将对简单的单个语句进行求值。

```
<compact if statement> ::=
  IF <conditional expression> ( <simple statement> | <SMT> )
```

可用占位符<SMT>来表示未定义的简单语句。

例如，

```
IF ERRNO = escape1 GOTO next;
```

4.16 For语句

定义

for语句将重复对语句表进行求值，而循环变量将在指定范围内递增（或递减）。一个可选步骤子句能够选择增量（或减量）值。

循环变量：

- 以外观来进行声明（num型）。
- 具备语句表范围（do .. endfor）。
- 隐藏同名的其他对象。
- 是只读的，即，无法被for循环的语句更新。

```
<for statement> ::=
  FOR <loop variable> FROM <expression>
  TO <expression> [ STEP <expression> ]
  DO <statement list> ENDFOR
<loop variable> ::= <identifier>
```

最初，from表达式、to表达式和step表达式将进行求值，将保持其值。它们只求值一次。循环变量以from值开头。如果未指定step值，则默认step值为1（值域在递减的情况下，为-1）。

在每一（非首个）新循环前，将更新循环变量，并对照值域核实新值。只要循环变量的值违背（超出）值域，那么将继续执行后续语句。

from表达式、to表达式和step表达式均必须为num（数字）型。

例如，

```
FOR i FROM 10 TO 1 STEP -1 DO
  a{i} := b{i};
ENDFOR
```

4.17 While语句

定义

只要指定的条件表达式求值为真，那么，while语句将重复对语句表进行求值。

```
<while statement> ::=
  WHILE <conditional expression> DO
  <statement list> ENDWHILE
```

将在每一新循环前，对条件表达式进行求值和核实。只要条件表达式求值为假，那么将继续执行后续语句。

例如，

```
WHILE a < b DO
  ...
  a := a + 1;
ENDWHILE
```

4.18 Test语句

定义

test语句将按表达式的值，对若干语句表中的一个语句表求值，或不对任何语句表求值。

```
<test statement> ::=
  TEST <expression>
  { CASE <test value> { ',' <test value> } ':' <statement list> )
       | <CSE> }
  [ DEFAULT ':'<statement list> ]
  ENDTEST
<test value> ::= <expression>
```

每一语句表前都跟有一个测试值表，指定选择特定替代项的值。test表达式可以为任何值数据类型或半值数据类型（参见第36页的数据类型的值类型）。测试值的类型必须等同于测试表达式的类型。执行test语句，将选择一项替代项或不选择任何替代项。在不止一个测试值与测试表达式拟合的情况下，仅第一个测试值才会被识别。可用占位符<CSE>来表示未定义的case子句。

如果没有case子句与表达式拟合，那么将对可选默认子句求值。

例如，

```
TEST choice
CASE 1, 2, 3 :
  pick number := choice;
CASE 4 :
  stand_by;
DEFAULT:
  write console, "Illegal choice";
ENDTEST
```

此页刻意留白

5 程序声明

5.1 程序声明介绍

定义

一项程序是可执行代码的指定载体。用户程序将以RAPID程序声明来进行定义。预定义程序将由系统提供，一直可供使用。

程序分为三类：无返回值程序、有返回值程序和软中断程序。

有返回值程序将返回特定类型的值，用于表达式上下文中（参见第*60*页的函数调用）。

无返回值程序不返回任何值，用于语句上下文中（参见第*70*页的过程调用）。

软中断程序能对中断进行响应（参见第*113*页的中断）。软中断程序可与特定中断关联起来（使用connect语句，参见第*78*页的*Connect*语句），在后续发生该特定中断的情况下，被自动执行。决不可从RAPID代码明确调用软中断程序。

可用占位符<RDN>表示程序声明。

```
<routine declaration> ::=
  [LOCAL] ( <procedure declaration>
    | <function declaration>
    | <trap declaration> )
  | <comment>
  | <RDN>
```

程序声明指定了程序的下列内容：

- 名称
- 数据类型（仅对有返回值程序有效）
- 参数（不适用于软中断程序）
- 数据声明和语句（体）
- 回退处理器（仅对无返回值程序有效）
- 错误处理器
- 撤销处理器

限制

程序声明仅可出现在模块的最后一部分（参见第*115*页的任务模块），程序声明无法进行嵌套，即，无法在程序声明中对程序进行声明。

程序声明的可选局部命令将程序归为局部程序或全局程序（参见第*88*页的程序的范围规则）。

5.2 参数声明

定义

程序声明的参数表指定了调用程序时必须/可提供的参数（实参）。参数将为必要参数或可选参数。可选参数可从程序调用的参数表省去（参见第42页的数据对象的范围规则）。两项或两项以上可选参数可声明为彼此排斥，在此情况下，程序调用中最多只可有其中一项可选参数。如果程序调用提供了相应参数，那么将称存在可选参数，否则将称不存在可选参数。不可设置或使用不存在的可选参数的值。可利用预定义有返回值程序Present来测试可选参数是否存在。可用占位符<PAR>、<ALT>、<DIM>来表示参数表的未定义部分。

```
<parameter list> ::=
  <first parameter declaration> { <next parameter declaration> }
<first parameter declaration> ::=
  <parameter declaration>
  | <optional parameter declaration>
  | <PAR>
<next parameter declaration> ::=
  ',' <parameter declaration>
  | <optional parameter declaration>
  | ',' <optional parameter declaration>
  | ',' <PAR>
<optional parameter declaration> ::=
  '\' ( <parameter declaration> | <ALT> ) { '|' ( <parameter
       declaration> | <ALT> ) }
<parameter declaration> ::=
  [ VAR | PERS | INOUT ] <data type> <identifier> [ '{' ( '*' {
       ',' '*' } ) | <DIM> '}' ]
  | 'switch' <identifier>
```

操作前提

调用参数的数据类型必须等同于相应参数的数据类型。

访问模式

每一参数均具备访问模式。可用访问模式有in（默认）、var、pers、inout和ref。访问模式指定了RAPID语言如何将相应调用参数转移到参数。

- in参数以调用参数（表达式）的值为初始值。可将参数用作（比如，赋予一个新值）一般程序变量。
- var、pers、inout或ref参数将用作调用参数（数据对象）的别名。这意味着，参数的更新也将是调用参数的更新。

> **注意**
>
> RAPID程序无法拥有ref参数，仅预定义程序才可拥有。

参数的指定访问模式将相应调用参数限为合法（下表中的“X”）或非法（下表中的“-”）。

变元	in	var	pers	inout	ref
常量	X				X

下一页继续

变元	in	var	pers	inout	ref
只读变量[i]	X				X
变量	X	X		X	X
参数in	X	X	-	X	X
参数var	X	X	-	X	X
参数pers	X	-	X	X	X
参数inout var	X	X	- [ii]	X	X
参数inout pers	X	- [ii]	X	X	X
任何其他表达式	X	-	-	-	-

[i] 比如，FOR循环变量（参见第*81*页的*For*语句）、errno、intno。
[ii] 执行错误（参见第*17*页的错误分类）。

内置程序

可利用内置程序IsPers和IsVar来测试inout参数是否是变量或永久调用参数的别名。

开关型

（仅）可将特殊类型switch分配给可选参数，这能提供采用“开关型参数”的手段，即，仅以名称（非值）赋予的参数。开关型的域为空，没有值能转移给开关型参数。采用开关型参数的唯一途径是利用预定义有返回值程序Present来核实其存在性，或在程序调用中，将其作为一个参数进行传递。

例如，

```
PROC glue ( \switch on | switch off, ... ! switch parameters
  ...
  IF Present(off ) THEN
    ! check presence of optional parameter 'off'
    ...
ENDPROC
glue\off, pos2; ! argument use
```

数组

数组可能会以调用参数的形式进行传递。数组调用参数的阶数必须与相应参数的阶数相符。数组参数的维度一致（带‘*’标记）。实际维度与程序调用中的相应调用参数的维度进行后期绑定。程序可以预定义有返回值程序Dim来确定参数的实际维度。

例如，

```
... , VAR num pallet{*,*}, ...
! num-matrix parameter
```

5.3 程序的范围规则

定义

对象的范围表示名称显示范围。预定义程序的范围包括RAPID模块。下列范围规则对用户程序有效：

- 局部用户程序的范围包含其所处模块。
- 全局用户程序的范围还包括任务缓冲区的其他模块。
- 在范围之内，用户程序隐藏了同名的预定义对象。
- 在范围之内，局部用户程序隐藏了同名的全局模块对象。
- 同一模块中声明的两个模块对象不可同名。
- 任务缓冲区中，两个不同模块中声明的两个全局对象不可同名。
- 全局用户程序和模块不可共享同一名称。

其他范围规则

参数方面的范围规则与程序变量方面的范围规则相符。有关程序变量范围的信息，请参见第*42*页的数据对象的范围规则。

有关任务模块的信息，请参见第*115*页的任务模块。

5.4 无返回值程序声明

定义

无返回值程序声明为程序定义绑定一个标识符。

```
<procedure declaration> ::=
  PROC <procedure name>
  '(' [ <parameter list> ] ')'
  <data declaration list>
  <statement list>
  [ BACKWARD <statement list> ]
  [ ERROR [ <error number list> ] <statement list> ]
  [ UNDO <statement list> ]
  ENDPROC
<procedure name> ::= <identifier>
<data declaration list> ::= { <data declaration> }
```

数据声明表可包含注释，请参见第28页的备注。

求值和终止

将以return语句（参见第73页的Return语句）来明确终止对无返回值程序的求值，或在到达无返回值程序末尾（ENDPROC、BACKWARD、ERROR或UNDO）时，即暗示着其执行终止。

比如，让num数组中的所有元素乘以：

```
PROC arrmul( VAR num array{*}, num factor)
  FOR index FROM 1 TO Dim( array, 1 ) DO
    array{index} := array{index} * factor;
  ENDFOR
ENDPROC ! implicit return
```

预定义Dim有返回值程序将返回数组的维度。

后期绑定

后期绑定调用中将用到的无返回值程序将被视作一个特例。即，（从同一后期绑定语句调用的）无返回值程序的参数在可选/必要参数和模式方面应匹配，并且无返回值程序的参数也应为同一基本类型。举例而言，如果需要一个无返回值程序的第二个参数，并将该参数声明为VAR num，那么，被同一后期绑定语句调用的其他无返回值程序的第二个参数，应具备第二个基本num型的必要VAR参数。这些无返回值程序也应具备同样的参数号。如果存在相互排斥的可选参数，那么，这些可选参数也必须在同等意义上匹配。

5.5 有返回值程序声明

定义

有返回值程序声明为有返回值程序定义绑定一个标识符。

```
<function declaration> ::=
  FUNC <data type>
  <function name>
  '(' [ <parameter list> ] ')'
  <data declaration list>
  <statement list>
  [ ERROR [ <error number list> ] <statement list> ]
  [ UNDO <statement list> ]
  ENDFUNC
<function name> ::= <identifier>
```

有返回值程序可具备（返回）任意值数据类型（包括任何可用的安装数据类型）。有返回值程序无法具备维度，即，有返回值程序无法返回数组值。

求值和终止

必须以return语句来终止有返回值程序的求值，参见第73页的*Return*语句。

如，可返回矢量长度。

```
FUNC num veclen(pos vector)
  RETURN sqrt(quad(vector.x) + quad(vector.y) + quad(vector.z));
  ERROR
    IF ERRNO = ERR_OVERFLOW THEN
      RETURN maxnum;
    ENDIF
    ! propagate "unexpected" error
    RAISE;
ENDFUNC
```

5.6 软中断程序声明

定义

软中断程序声明为软中断程序定义绑定一个标识符。软中断程序可用connect语句与中断（编号）关联起来，请参见第78页的*Connect*语句。请注意，一个软中断程序可与许多（或不与任何）中断关联起来。

```
<trap declaration> ::=
  TRAP <trap name>
  <data declaration list>
  <statement list>
  [ ERROR [ <error number list> ] <statement list> ]
  [ UNDO <statement list> ]
  ENDTRAP
<trap name> ::= <identifier>
```

求值和终止

将利用return语句（参见第73页的*Return*语句）来明确终止软中断程序的求值，或者，在到软中断程序的末尾（endtrap、error或undo）时，即暗示着终止软中断程序的求值。将从中断点继续执行程序。

比如，响应低压中断。

```
TRAP low_pressure
  open_valve\slow;
  ! return to point of interrupt
  RETURN;
ENDTRAP
```

比如，响应高压中断。

```
TRAP high_pressure
  close_valve\fast;
  ! return to point of interrupt
  RETURN;
ENDTRAP
```

此页刻意留白

6 步退执行

6.1 步退执行介绍

定义

RAPID语言支持对语句进行逐步步退执行。在RAPID程序开发期间，步退执行对调试、测试和调节十分有用。RAPID无返回值程序可能包含回退处理器（语句表），回退处理器定义了无返回值程序（调用）的步退执行“行为”。

限制

下列常规限制对步退执行有效：

- 仅简单（非复合）语句才可被步退执行。
- 不可在语句表顶部从程序进行回退（并且无法实现程序调用）。
- 简单语句具备下列回退行为。
- 无返回值程序调用（预定义或用户定义）可进行任意回退行为 - 采取一些行动、不采取行动或否决[I]回退调用。无返回值程序定义中对该行为进行了定义。
- 进行步退执行的无返回值程序调用的调用参数通常将（甚至在被否决的情况下也将）被执行并转移到无返回值程序的参数处，方式正如步进执行的方式一样。调用参数表达式（可能包括有返回值程序调用）通常被“步进”执行。
- 注释、标签、赋值语句和connect语句均作为“空操作”语句来执行，而所有其他简单语句否决I步退执行。

I 不支持步退执行，未开展任何步骤。

6.2 回退处理器

定义

无返回值程序可包含回退处理器，该回退处理器定义了无返回值程序调用的步退执行。

反向处理器实际上是过程的一部分，同时任何程序数据的范围都由过程的反向处理器构成。

例如，

```
PROC MoveTo ()
  MoveL p1,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p4,v500,z10,tool1;
BACKWARD
  MoveL p4,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p1,v500,z10,tool1;
ENDPROC
```

在步进执行期间调用无返回值程序MoveTo时，前3个指令将按下列代码中的编号被执行。回退指令（后3个）未被执行。

```
PROC MoveTo ()
  1. MoveL p1,v500,z10,tool1;
  2. MoveC p2,p3,v500,z10,tool1;
  3. MoveL p4,v500,z10,tool1;
BACKWARD
  MoveL p4,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p1,v500,z10,tool1;
ENDPROC
```

在步退执行期间调用无返回值程序MoveTo时，后3个指令将按下列代码中的编号被执行。步进指令（前3个）未被执行。

```
PROC MoveTo ()
  MoveL p1,v500,z10,tool1;
  MoveC p2,p3,v500,z10,tool1;
  MoveL p4,v500,z10,tool1;
BACKWARD
  1. MoveL p4,v500,z10,tool1;
  2. MoveC p2,p3,v500,z10,tool1;
  3. MoveL p1,v500,z10,tool1;
ENDPROC
```

限制

程序的反向处理器或错误处理器中的指令可能不能步退执行。不能嵌入步退执行，也就是说，不能同时步退执行同一调用链中的两个指令。

另请参阅第95页的回退处理器中对*Move*指令的限制

不带回退处理器的无返回值程序

不带回退处理器的无返回值程序不能进行步退执行。而含空回退处理器的无返回值程序可作为空操作被执行。

6.3 回退处理器中对Move指令的限制

限制

回退处理器中的Move指令类型和序列必须为同一程序中步进执行的移动指令类型和序列的镜像。在下列示例中，各指令的编号展示了其执行顺序。

```
PROC MoveTo ()
  1. MoveL p1,v500,z10,tool1;
  2. MoveC p2,p3,v500,z10,tool1;
  3. MoveL p4,v500,z10,tool1;
BACKWARD
  3. MoveL p4,v500,z10,tool1;
  2. MoveC p2,p3,v500,z10,tool1;
  1. MoveL p1,v500,z10,tool1;
ENDPROC
```

注意：MoveC中CirPoint p2和ToPoint p3的顺序应该保持一致。

采用Move指令，即指，所有指令将让机械臂或附加轴进行一定的移动，如MoveL、SearchC、TriggJ、ArcC、PaintL ...

> **小心**
>
> 脱离反向处理器中的该编程限制会导致错误的后向运动。在部分后向路径上，直线运动会导致圆周运动，或出现相反的情况。

此页刻意留白

7 错误恢复

7.1 错误处理器

定义

执行错误（参见第17页的错误分类）系指RAPID程序代码特定段的执行出现异常情况。错误会造成无法进一步执行（或者说至少执行是危险的）。“溢出”和“除零”均为错误的例子。错误以特有的错误编号为标识，从而能总是被系统识别。发生错误会引起正常程序执行被中止，将改为由错误处理器控制。错误处理器能响应并且可能恢复程序执行期间出现的错误。如果无法进一步执行，那么错误处理器至少能确保，任务被平稳地中止。

任意程序都可包含一个错误处理器。错误处理器实际上是程序的组成部分，任意程序数据对象（变量、常量或参数）的范围中也包括程序的错误处理器。若程序求值期间出现错误，则将改由错误处理器进行控制。

例如，

```
FUNC num safediv(num x, num y)
  RETURN x / y;
ERROR
  IF ERRNO = ERR_DIVZERO THEN
    ! return max numeric value
    RETURN max_num;
  ENDIF
ENDFUNC
```

ERRNO

预定义（只读）变量ERRNO包含（最新）错误的错误编号，可供错误处理器用于标识相应错误。在采取必要行动后，错误处理器可：

- 用RETRY语句，从出错语句重新开始执行程序，参见第76页的*Retry*语句。
- 用TRYNEXT语句，从出错语句的后一个语句重新开始执行程序，参见第77页的*Trynext*语句。
- 用RETURN语句，改为重新由程序调用器控制，参见第73页的*Return*语句。若该程序为有返回值程序，则RETURN语句必须指定相应的返回值；
- 用RAISE语句，将错误传递到程序调用器，参见第74页的*Raise*语句。“由于不熟悉此错误，需由调用器来处理此错误”。

系统错误处理器

若不含错误处理器的程序中出现错误或到达错误处理器末尾（ENDFUNC、ENDPROC或ENDTRAP），则将调用系统错误处理器。系统错误处理器只是报告错误和停止执行。

> **注意**
>
> 错误处理器或回退处理器中出现的错误既不能恢复也无法进行响应。通常会将此类错误传递至系统错误处理器。

在程序调用链中，每个程序可能都有自己的错误处理器。若带错误处理器的程序中出现错误，并用RAISE语句直接明确传送错误，则在传送错误的程序被调用时，会再次

下一页继续

发出同一错误。在没有错误处理器的情况下到达调用链顶端（任务的入口程序）时，或到达调用链中任一错误处理器的末尾时，将调用系统错误处理器。只有系统错误处理器才会报告错误并停止执行。由于系统只能调用软中断程序（以对中断做出响应），因此，软中断程序中的错误将被传送至系统错误处理器。

程序导致的错误

除了系统检测和发出的错误外，程序还可用RAISE语句明确发出错误，请参见第74页的*Raise*语句。这可从复杂情形中恢复。举例而言，这可用于脱离深嵌套代码位置。可采用1至90范围内的错误编号。

例如，

```
CONST errnum escape1 := 10;
  ...
  RAISE escape1;
  ...
ERROR
  IF ERRNO = escape1 THEN
    RETURN val2;
  ENDIF
ENDFUNC
```

7.2 关于长跳转的错误恢复

定义

可采用带长跳转的错误恢复，以绕过正常程序调用和返回机制，从而处理异常或特殊情况。为此，必须指定特定错误恢复点。用RAISE指令，将进行长跳转，转由该错误恢复点进行执行控制。

一般以带长跳转的错误恢复来从深嵌套代码位置（无论执行等级为何），尽快尽量简单地改由更高等级进行执行控制。

执行等级

执行等级系指RAPID程序运行所处的背景。系统有三种执行等级，即，常规等级、软中断等级和用户等级。

- 常规等级：所有程序都从此等级开始，它为最低等级。
- 软中断等级：软中断程序在这一等级被执行，这一等级覆写常规等级，但可被用户等级覆写。
- 用户等级：事件程序和服务程序在这一等级被执行，这一等级覆写常规等级和软中断等级，它是最高等级，不可被其他等级所覆写。

错误恢复点

带长跳转的错误恢复的关键在于特征错误恢复点。

错误恢复点是一个常规ERROR子句，但带有扩展语法以及以一对圆括号括住的错误编号表，参见下列示例。

```
MODULE example1
  PROC main()
    ! Do something important
    myRoutine;
    ERROR (56, ERR_DIVZERO)
      RETRY;
  ENDPROC
ENDMODULE
```

语法

错误恢复点含下列语法：(EBNF)

```
[ ERROR [ <error number list> ] <statement list> ]
<error number list> ::= '(' <error number> { ',' <error number>}
     ')'
<error number> ::=
  <num literal>
  | <entire constant>
  | <entire variable>
  | <entire persistent>
```

采用带长跳转的错误恢复

```
MODULE example2
  PROC main()
    routine1;
    ! Error recovery point
```

下一页继续

```
    ERROR (56)
      RETRY;
  ENDPROC

  PROC routine1()
    routine2;
  ENDPROC

  PROC routine2()
    RAISE 56;
  ERROR
    ! This will propagate the error 56 to main
    RAISE;
  ENDPROC
ENDMODULE
```

系统按下列顺序处理长跳转：

- 发出的错误编号将从调用程序的错误处理器开始搜索，搜索到当前调用链的顶部。如果在调用链中的任何程序处，存在带有发出错误编号的错误恢复点，那么，将在该程序的错误处理器继续执行程序。
- 如果在当前执行等级未发现错误恢复点，则将在前一执行等级继续搜索，直至到达常规等级。
- 如果在任何执行等级都未发现错误恢复点，则错误将被发出并在调用程序的错误处理器（如有）中被处理。

通过执行等级边界的错误恢复

如果能以长跳转来通过执行等级边界进行执行控制，即，程序执行可从软中断程序、用户程序跳转至Main程序，不论调用链处于软中断等级、用户等级和常规等级的何处。对于需从程序中的适当安全位置来继续执行或开始执行程序的异常情况而言，这对这类异常情况的处理很有帮助。

当在一个执行等级长跳转到另一等级时，该等级可有一个活动指令。由于从一个错误处理器长跳转到另一个错误处理器，因此，系统撤销了该活动指令（比如，活动的MoveX指令将清除此路径的一部分）。

附加信息

使用预定义常量LONG_JMP_ALL_ERR，或许可以捕捉处于错误恢复点的所有种类错误。当采用带长跳转的错误恢复时，请遵守下列限制的要求：

- 勿假设错误恢复点的执行模式（连续、循环或步进）与错误发生点的相同。长跳转时未继承执行模式。
- 使用StorePath时请谨慎。在进行长跳转前，通常调用RestoPath，否则结果是不可预测的。
- 在长跳转后，错误恢复点的重试次数不设为零。
- 在错误恢复点使用TRYNEXT时请谨慎，如果在有返回值程序调用中发生错误，那么结果可能是不可预测的，如下列示例所述。

下一页继续

例如，

```
MODULE Example3
  PROC main
    WHILE myFunction() = TRUE DO
      myRoutine;
    ENDWHILE
    EXIT;
  ERROR (LONG_JMP_ALL_ERR)
    TRYNEXT;
  ENDPROC
ENDMODULE
```

如果在有返回值程序myFunction中发生错误，并且在主程序捕获该错误，那么，TRYNEXT指令将让下一指令（在此情况下，系指EXIT）来进行执行控制。这是因为WHILE指令被视为出错的指令。

UNDO处理器

使用长跳转时，在不执行错误处理器或程序的末尾的情况下，一个或若干无返回值程序可能会被放弃。如果未使用撤销处理器，那么这些程序可能留下未了结的末尾。在下列示例中，如果采用长跳转并且routine1中无撤销处理器，那么，routine1将让文件日志处于打开状态。

为了确保每一程序在自身之后均进行清理，请在因长跳转而无法完成执行的任何程序中，使用撤销处理器。

例如，

```
MODULE example4
  PROC main()
    routine1;
    ! Error recovery point
  ERROR (56)
    RETRY;
  ENDPROC

  PROC routine1()
    VAR iodev log;
    Open "HOME:" \File:= "FILE1.DOC", log;
    routine2;
    Write log, "routine1 ends with normal execution";
    Close log;
  ERROR
    ! Another error handler
  UNDO
    Close log;
  ENDPROC

  PROC routine2()
    RAISE 56;
  ERROR
    ! This will propagate the error 56 to main
    RAISE;
  ENDPROC
```

下一页继续

```
ENDMODULE
```

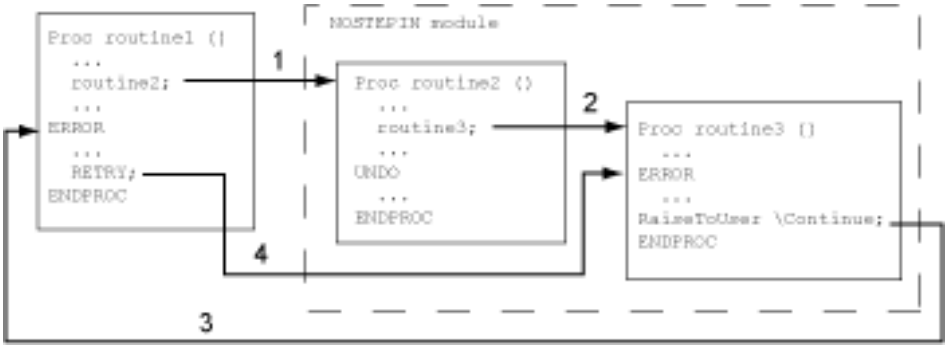
7.3 Nostepin程序

定义

nostepin模块中的nostepin程序可在调用链中调用彼此。在调用链的其中一个程序的错误处理器中采用RAISE指令，将让错误在调用链中步进一步。为了用RAISE指令向用户等级（nostepin模块外）发出错误，调用链中的每一程序均必须具备发出该错误的错误处理器。

使用RaiseToUser指令，可在调用链中将错误步进数步。然后，由调用链的最后一个程序的错误处理器来处理错误。最后一个程序并非nostepin程序。

如果用参数\Continue来调用RaiseToUser，那么，引起错误的指令（nostepin程序中）将被记住。如果处理错误的错误处理器以RETRY或TRYNEXT结尾，那么将从错误发生点继续执行。



xx1300000275

1	routine2被调用
2	routine3被调用
3	向用户等级发出错误
4	执行被恢复到在引起错误的routine3中执行。

注意

一个或若干程序可被放弃，不执行错误处理器或程序的末尾。在本示例中，如果RaiseToUser使用了参数\BreakOff，而非\Continue，那么，routine2就会发生上述情况。为了确保此程序不留下任何未了结的末尾（比如，被打开的文件），请确保具备会进行清理（比如，关闭文件）的撤销处理器。

注意

如果将调用nostepin程序的程序（在此例中，为routine1）变为nostepin程序，那么，将不再由其错误处理器来处理错误。将一个程序变为nostepin程序，可能要求将错误处理器移动到用户层。

7.4 异步引起的错误

关于异步错误

如果一个移动指令以角区结束，那么在第一个移动指令完成其路径之前，必须执行下一个移动指令。否则，机械臂不知道如何在角区移动。如果每一个移动指令以较大角区，仅移动一段短距离，那么必须预先执行若干移动指令。

如果在机械臂移动期间有哪里错了，那么可能发生错误。然而，如果继续执行程序，那么，在出错时移动机械臂的是哪个移动指令这点上，并不明显。将通过处理异步错误来解决这项问题。

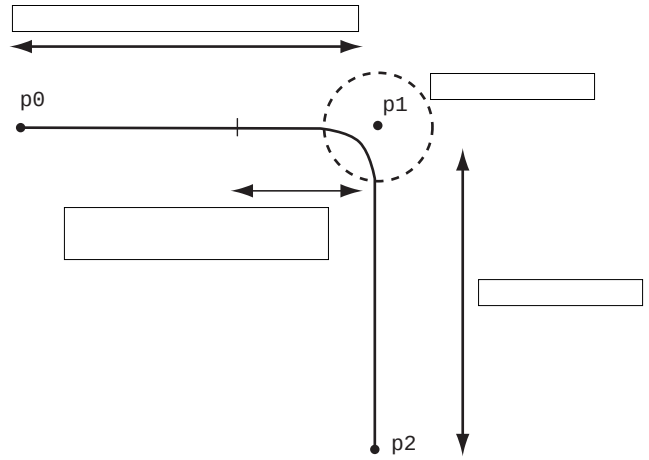
基本理念在于，异步错误与一个移动指令关联起来，将由调用该移动指令的程序中的错误处理器处理异步错误。

两类异步错误

有两种方式造成异步错误，引起的行为稍稍不同

- `ProcerrRecovery \SyncOrgMoveInst`造成的异步错误与产生当前机械臂路径的移动指令相关联。
- `ProcerrRecovery \SyncLastMoveInst`造成的异步错误与当前被执行的移动指令相关联。如果当前没有执行任何移动指令，则此错误与将要执行的下一移动指令相关联。

如果在第一条路径期间出现错误，但程序在计算第二条路径（参见下图），那么，行为取决于`ProcerrRecovery`的参数。如果以`\SyncOrgMoveInst`造成错误，那么该错误与第一个移动指令（产生第一条路径的指令）相关联。如果以`\SyncLastMoveInst`造成错误，那么，该错误与第二个移动指令（产生第二条路径的指令）相关联。



xx1300000276

在调用移动指令的程序中，尝试处理错误

如果生成一个能处理错误的程序来处理机械臂移动期间可能出现的过程错误，那么，你想要在此程序中处理此类过程错误。如果当程序指针处于一个子程序中时发出错误，那么你不想要该子程序的错误处理器来处理错误。

下一页继续

异步错误与机械臂当前执行的路径相关联。可由出错时其移动指令产生机械臂执行路径的那个程序中的错误处理器来处理异步错误。

在下述示例中，在机械臂到达p1前发生过程错误，但程序指针已经继续到子程序write_log。

程序示例

```
PROC main()
  ...
  my_process;
  ...
ERROR
  ...
ENDPROC
PROC my_process()
  ...
  MoveL p1, v300, z10, tool1;
  write_log;
  MoveL p2, v300, z10, tool1;
  ...
ERROR
  ...
ENDPROC
PROC write_log()
  ...
ERROR
  ...
ENDPROC
```

示例说明

如果未处理异步错误，那么程序指针处于write_log时发出的错误将由write_log中的错误处理器进行处理。异步错误的处理将确保由my_process中的错误处理器来处理错误。

以ProcerrRecovery \SyncOrgMoveInst产生的异步错误将由my_process.中的错误处理器来即刻进行处理。以ProcerrRecovery \SyncLastMoveInst产生的异步错误将等待程序指针到达my_process中的第二个移动指令，之后才在my_process的错误处理器中处理该错误。

> **注意**
>
> 如果子程序（本例中，系指write_log ）将具备移动指令，并且使用\SyncLastMoveInst，那么可能由该子程序中的错误处理器来处理该错误。

如果my_process 中的错误处理器以EXIT结尾，那么将停止所有程序执行。

如果my_process中的错误处理器以RAISE结尾，那么，将由main中的错误处理器处理该错误。程序调用my_process并放弃write_log 。如果主程序中的错误处理器以RETRY结尾，那么将重新开始执行my_process。

如果my_process中的错误处理器以RETRY或TRYNEXT结尾，那么将继续从程序指针所在位置（write_log中）继续执行程序。错误处理器应已经解决错误情况并调用

下一页继续

StartMove来让引起错误的指令重新开始移动。即便错误处理器以RETRY结尾，第一个MoveL指令也不会再次被执行。

> **注意**
>
> 在此情况下，TRYNEXT的执行方式与RETRY相同，这是因为可以从出错处重启系统。

放弃程序调用时会发生什么情况？

当执行到一个程序的末尾时，将放弃该程序调用。如果该程序调用已被放弃，那么，该程序调用的错误处理器无法调用。在下列示例中，在第一个my_process程序调用被放弃后，机械臂移动将继续（因为上一个移动指令有角区）。

程序示例

```
PROC main()
  ...
  my_process;
  my_process;
  ...
ERROR
  ...
ENDPROC
PROC my_process()
  ...
  MoveL p1, v300, z10, tool1;
  MoveL p2, v300, z10, tool1;
  ...
ERROR
  ...
ENDPROC
```

示例说明

如果在第一个my_process造成错误时，程序指针处于主程序中，那么不能以my_process程序调用中的错误处理器来处理错误。那么，由哪里处理此错误取决于异步错误如何发生。

- 如果以ProcerrRecovery \SyncOrgMoveInst发出错误，那么，将在调用链中步进一步来处理该错误。将由对被弃程序调用进行调用的程序的错误处理器来处理此错误。在上述示例中，如果my_process程序调用被放弃，那么将由主程序的错误处理器来处理此错误。
- 如果以ProcerrRecovery \SyncLastMoveInst发出错误，那么将由下一移动指令所在的错误处理器（即，my_process的第二个程序调用）来处理该错误。可延迟发出错误，直至程序指针到达下一移动指令。

> **提示**
>
> 为确保在一个程序中处理异步错误，请确保最后一个移动指令以一个停止点（非角区）结束，并且不采用\Conc。

下一页继续

示例

在本示例中，在程序my_process中可产生异步错误。将由my_process的错误处理器来处理这些错误。

将信号*do_myproc*设为1来启动一项进程流。信号*di_proc_sup*将监督此项进程，如果*di_proc_sup*变为1，那么将发出异步错误。在本简单示例中，在重新开始移动前，将*do_myproc*再次设为1来处理错误。

```
MODULE user_module
  VAR intnum proc_sup_int;
  VAR iodev logfile;

  PROC main()
    ...
    my_process;
    my_process;
    ...
  ERROR
    ...
  ENDPROC

  PROC my_process()
    my_proc_on;
    MoveL p1, v300, z10, tool1;
    write_log;
    MoveL p2, v300, z10, tool1;
    my_proc_off;
  ERROR
    IF ERRNO = ERR_PATH_STOP THEN
      my_proc_on;
      StartMove;
      RETRY;
    ENDIF
  ENDPROC

  PROC write_log()
    Open "HOME:" \File:= "log.txt", logfile \Append;
    Write logfile "my_process executing";
    Close logfile;
  ERROR
    IF ERRNO = ERR_FILEOPEN THEN
      TRYNEXT;
    ENDIF
  UNDO
    Close logfile;
  ENDPROC

  TRAP iprocfail
    my_proc_off;
    ProcerrRecovery \SyncLastMoveInst;
    RETURN;
  ENDTRAP
```

下一页继续

续前页

```
    PROC my_proc_on()
      SetDO do_myproc, 1;
      CONNECT proc_sup_int WITH iprocfail;
      ISignalDI di_proc_sup, 1, proc_sup_int;
    ENDPROC

    PROC my_proc_off()
      SetDO do_myproc, 0;
      IDelete proc_sup_int;
    ENDPROC
  ENDMODULE
```

程序指针处于write_log时的错误

如果机械臂正在朝p1移动的过程中发生一项过程错误，而程序指针已经在子程序write_log中时，会发生什么情况？

将在调用移动指令的程序（即my_process）中发出错误，并由其错误处理器来处理此错误。

由于在此示例中，ProcerrRecovery指令采用开关\SyncLastMoveInst，因此，在下一移动指令处于活动状态前，不会发出此错误。一旦my_process中的第二个MoveL指令处于活动状态，那么将发出错误，并在my_process的错误处理器中处理错误。

如果ProcerrRecovery使用了开关\SyncOrgMoveInst，那么将直接在my_process中发出错误。

执行完my_process时的错误

如果机械臂正在朝p2移动的过程中发生一项过程错误，而程序指针已经离开程序my_process，那么，会发生什么情况？

引起错误的程序调用（第一个my_process）已被放弃，并且其错误处理器无法处理错误。在哪里发出此错误取决于调用ProcerrRecovery时使用哪个开关。

由于在此示例中，ProcerrRecovery指令采用开关\SyncLastMoveInst，因此，在下一移动指令处于活动状态前，不会发出此错误。一旦第二个my_process程序调用中，一个移动指令处于活动状态，那么将发出错误，并在my_process的错误处理器中处理错误。

如果ProcerrRecovery采用开关\SyncOrgMoveInst，那么将在主程序中发出错误，\SyncOrgMoveInst的执行方式为，在引起错误的程序调用（my_process）被放弃的情况下，调用该程序的程序（main）将发出此错误。

注意

如果主程序中my_process调用之间存在一项移动指令，并且采用了\SyncLastMoveInst，那么将由主程序的错误处理器处理此错误。如果在my_process调用之间调用了带移动指令的另一项程序，那么将在该程序中处理此错误。这表明，采用\SyncLastMoveInst时，对于哪个是下一移动指令这点，必须施加一定的控制。

下一页继续

Nostepin移动指令和异步错误

用无返回值程序创建一个自定义nostepin移动指令时，建议采用ProcerrRecovery \SyncLastMoveInst。这样，可由nostepin指令来处理所有异步错误。

这要求，在整个移动序列期间，用户仅采用这类移动指令。移动序列必须开始并在停止点结束。只有两个指令具备同样的错误处理器时，这两个指令才能在同一移动序列中使用。这意味着，同一移动序列中可采用一个线性移动指令和一个圆周移动指令，这两个指令采用同样的无返回值程序和同样的错误处理器。

如果应向用户发出一个错误，那么请使用RaiseToUser \Continue。在错误解决后，那么，可从错误发生处继续执行。

撤销处理器

可以突然结束一个程序的执行，而不运行该程序的错误处理器。这意味着，该程序没有在自身之后进行清理。

在下列示例中，我们假设机械臂正在朝p1移动的过程中出现异步错误，但程序指针处于write_log中的Write指令。如果没有撤销处理器，那么文件的日志文件不会被关闭。

```
PROC main()
  ...
  my_process;
  ...
ERROR
  ...
ENDPROC
PROC my_process()
  MoveL p1, v300, z10, tool1;
  write_log;
  MoveL p2, v300, z10, tool1;
ERROR
  ...
ENDPROC
PROC write_log()
  Open .. logile ..;
  Write logfile;
  Close logfile;
ERROR
  ...
UNDO
  Close logfile;
ENDPROC
```

在可被异步错误中断的所有程序中，可用撤销处理器解决此问题。异步错误本质上难以知道发生错误时程序指针的所在位置。因此，采用异步错误时，在任何需要清理的情况下，使用撤销处理器。

7.5 指令SkipWarn

定义

在错误处理器中处理的错误仍将在事件日志中生成警告。如果因一些原因，不想要在事件日志中写入警告，那么可采用指令SkipWarn。

示例

在下列示例代码中，程序尝试写入其他机器人系统也能访问的文件。如果该文件繁忙，则程序将等待0.1秒后再重试。如果未采用SkipWarn，那么，日志文件也将为每次尝试写一个警告，即便这些警告完全不必要。通过添加SkipWarn指令，运算符无法注意到在首次尝试时文件繁忙。

需注意，最多重试次数取决于参数*No Of Retry*。为确保进行4次以上重试，必须配置此参数。

```
PROC routine1()
  VAR iodev report;
  Open "HOME:" \File:= "FILE1.DOC", report;
  Write report, "No parts from Rob1="\Num:=reg1;
  Close report;
ERROR
  IF ERRNO = ERR_FILEOPEN THEN
    WaitTime 0.1;
    SkipWarn;
    RETRY;
  ENDIF
ENDPROC
```

7.6 运动错误处理

关于运动错误处理

发生一项碰撞错误时（事件编号*50204 - Motion supervision*），RAPID语言执行不必停止。如果定义了系统参数*Collision Error Handler*，那么在声明后，将进入RAPID错误处理器来执行，如果满足进一步执行的所有条件，那么可继续执行。

这被称作运动错误处理。

为将碰撞错误与其他RAPID错误分开，请将errno变量设为ERR_COLL_STOP。

示例

为了在离开错误处理器后能开始运动，必须从错误处理器调用StartMove指令。对于MultiMove，所有任务必须在错误处理器中具备StartMove指令，即便未冲突的任务也如此。

```
PROC main()
  MoveJ p10, v200, fine, tool0;
  MoveJ p20, v200, fine, tool0;
ERROR
  TEST ERRNO
  CASE ERR_COLL_STOP:
    StorePath;
    MoveJ p30, v200, fine, tool0;
    RestoPath;
  ENDTEST
  StartMove;
  RETRY;
ENDPROC
```

限制

运动错误处理不同于常规RAPID错误处理，因为程序指针可处于运动指针之前。通常，当在错误处理器中执行RETRY/TRYNEXT时，程序指针会移动到发出错误时所处的位置。此行为与运动错误的一样。将由程序指针移动到发出错误时所处的位置，而非由运动指针移动到该位置。

这意味着在恢复运动错误时存在丢失指令的风险。

如果采用精点，则可用可预测方式来处理运动错误。但当采用区域、或带\Conc参数的移动指令、或无返回值程序调用等等时，那么，在处理运动错误时可能会错过若干位置。

同样，当采用无返回值程序调用时，在发出错误的情况下，程序指针和运动指针并非总处于同一无返回值程序中。将使用程序指针所在的无返回值程序的错误处理器，而非采用运动指针所在的错误处理器。因此，当采用无返回值程序调用时，请确保以精点结束运动序列。

> **小心**
>
> 采用运动错误处理器来处理碰撞时，建议采用精点。

此页刻意留白

8 中断

定义

中断系指由中断编号标识的程序定义事件。因中断条件变为真，会发生中断。中断不像错误，中断的发生与特定代码位置无直接关系（不同步）。发生中断会引起正常程序执行被中止，转由软中断程序进行控制。将分配中断编号，并用connect语句将之与软中断程序联系（关联）起来，参见第78页的*Connect*语句。用预定义程序来定义和操控中断条件。一项任务可定义任意数量的中断。

中断识别和响应

即使系统即刻识别中断的发生，但也只会在特定程序位置才会作出响应，即调用相应的软中断程序，其中特定位置如下所示：

- 下一（中断识别后）（任意类型）语句的入口处。
- 语句表的最后一个语句后。
- 等待程序执行期间的任意时候（比如，`WaitTime`）。

这意味着，在识别中断后，通常将继续正常程序执行，直至到达上述位置之一。这通常造成中断识别和响应之间延迟2-30 ms，具体取决于中断时正在进行哪种移动。

编辑中断

利用中断编号来标识中断/中断条件。中断编号并非只是“任意”编号。中断编号归系统“所有”，必须进行分配，用connect语句与软中断程序关联起来（参见第78页的*Connect*语句），然后才可用于标识中断。

例如，

```
VAR intnum full;
...
CONNECT full WITH ftrap;
```

将利用预定义程序来定义和编辑中断。中断的定义指定了中断条件，并将之与中断编号关联起来。

例如，

```
! define feeder interrupts
ISignalDI sig3, high, full;
```

中断条件必须是有效的，能被系统监视。通常，定义程序（比如，`ISignalDI`）将激活中断，但并非总是如此。处于活动状态的中断可以再次被变无效（反之亦然）。

例如，

```
! deactivate empty
ISleep empty;
! activate empty again
IWatch empty;
```

中断的删除将取消中断编号的分配，清除中断条件。无需明确删除中断。当一项任务的求值终止时，将自动删除中断。

例如，

```
! delete empty
IDelete empty;
```

可禁用和启用中断。若禁用中断，则可将发生的任意中断列入等待队列，到再次启用中断时再首先发出。请注意，中断队列可能包含不止一个处于等待状态的中断。排队

下一页继续

等待的中断按fifo顺序（先进先出）再次发出。在软中断程序求值期间通常禁用中断，参见第*114*页的软中断程序。例如：

```
! enable interrupts
IEnable;
! disable interrupts
IDisable;
```

软中断程序

软中断程序能对中断作响应。将利用connect语句将软中断程序与特定中断编号关联起来，参见第*78*页的*Connect*语句。如果发生中断，将即刻转由被关联的软中断程序进行控制（参见第*113*页的中断识别和响应）。

例如，

```
LOCAL VAR intnum empty;
LOCAL VAR intnum full;

PROC main()
...
  ! Connect feeder interrupts
  CONNECT empty WITH ftrap;
  CONNECT full WITH ftrap;
  ! define feeder interrupts
  ISignalDI sig1, high, empty;
  ISignalDI sig3, high, full;
  ...
ENDPROC

TRAP ftrap
  TEST INTNO
  CASE empty:
    open_valve;
  CASE full:
    close_valve;
  ENDTEST
  RETURN;
ENDTRAP
```

一项以上中断可与同一软中断程序关联起来。预定义（只读）变量INTNO包含中断编号，能被软中断程序用于标识中断。在采取必要行动后，可采用return语句来终止软中断程序（参见第*73*页的*Return*语句），或在到达软中断程序的末尾（endtrap或错误）时，将终止软中断程序。将在中断点继续执行。需注意，在软中断程序求值期间，通常禁用中断（参见第*113*页的编辑中断）。

由于只有系统才能调用软中断程序（作为对中断的响应），因此，错误将从软中断程序传递到系统错误处理器，参见第*97*页的错误恢复。

9 任务模块

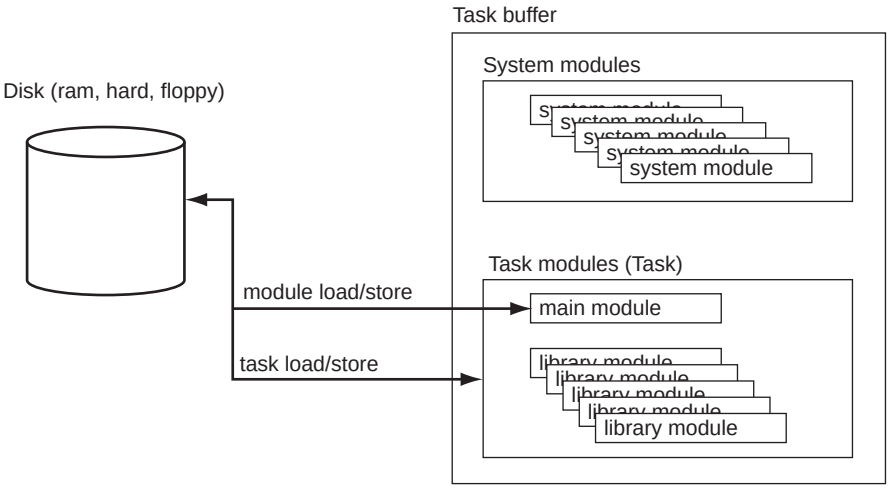
9.1 任务模块介绍

定义

一项RAPID应用被称作一项任务。一项任务包括一组模块。一个模块包括一组类型定义、数据和程序声明。将利用任务缓冲区来存放系统的当前在用（在执行、在开发）模块。任务缓冲区中的RAPID程序代码可作为单独模块或一组模块（即为一项任务）加载到/储存到面向文件的外部器件（通常为磁盘文件）/从面向文件的外部器件（通常为磁盘文件）加载/存储。

RAPID语言将区别系统程序模块和任务模块。任务模块被视为任务/应用的一部分，而系统程序模块被视为系统的一部分。系统程序模块在系统启动期间自动加载到任务缓冲区，旨在（预）定义常用的系统特定数据对象（工具、焊接数据、移动数据 ..）、接口（打印机、日志文件 ..）等。任务保存在文件上时，未含有系统程序模块。这意味着对系统程序模块所作的任何更新均将对任务缓冲区上当前拥有或随后加载的所有已有（原）任务造成影响。在其他意义上，任务模块和系统程序模块无区别；任务模块和系统程序模块可具备任意内容。

虽然（除了系统程序模块外）单个任务模块通常包含小应用，但较大应用可能包含主任务模块，主任务模块反过来又引用一项或多项其他库任务模块所含的程序和/或数据。



xx1300000277

举例而言，库模块可定义实际对象或逻辑对象（夹具、送料机、计数器等）的接口，或含有CAD系统生成的几何数据或在线数字化（示教）生成的几何数据。

一项任务模块包含任务的入口无返回值程序。运行任务实际上表示执行该入口程序。入口程序无法具备参数。

9.2 模块声明

定义

模块声明指定了一个模块的名称、属性和本体。模块名称将隐藏同名的任何预定义对象。两个不同模块不可共享同一名称。一个模块与一个全局模块对象（类型、数据对象或程序）不可共享同一名称。模块属性能让我们在模块加载到任务缓冲区后修改系统处理的一些方面。模块声明的本体包含一序列的数据声明，后接一序列的程序声明。

```
<module declaration> ::=
  MODULE <module name> [ <module attribute list> ]
  <type definition list>
  <data declaration list>
  <routine declaration list>
  ENDMODULE
<module name> ::= <identifier>
<module attribute list> ::= '(' <module attribute> { ',' <module
     attribute> } ')'
<module attribute> ::=
  SYSMODULE
  | NOVIEW
  | NOSTEPIN
  | VIEWONLY
  | READONLY
  <routine declaration list> ::= { <routine declaration> }
    <type definition list> ::= { <type definition> }
    <data declaration list> ::= { <data declaration> }
```

模块属性

模块属性具备下列意义：

属性	在被指定后，模块...
SYSMODULE	不是系统程序模块就是任务模块
NOVIEW	（是源代码）无法查看（仅能执行）
NOSTEPIN	在逐步执行时不能录入
VIEWONLY	不可修改
READONLY	不可修改，但该属性可以被取消。

一项属性不能指定一次以上。若存在属性，则必须按表顺序来指定属性（参见上文）。noview的说明不包括nostepin、viewonly和readonly（反之亦然）。viewonly的规范不包括readonly（反之亦然）。

示例

下列三个模块可表示一项（极简单）任务。

```
MODULE prog1(SYSMODULE, VIEWONLY)
  PROC main()
    ! init weldlib
    initweld;
    FOR i FROM 1 TO Dim(posearr,1) DO
      slow posearr{i};
    ENDFOR
```

下一页继续

```
  ENDPROC

  PROC slow(pose p)
    arcweld p \speed := 25;
  ENDPROC
ENDMODULE

MODULE weldlib
  LOCAL VAR welddata w1 := sysw1;
  ! weldlib init procedure
  PROC initweld()
    ! override speed
    w1.speed := 100;
  ENDPROC

  PROC arcweld(pose position \ num speed | num time)
    ...
  ENDPROC
ENDMODULE

MODULE weldpath ! (CAD) generated module
  CONST pose posearr{768} := [ [[234.7, 1136.7, 10.2], [1, 0, 0,
       0]], ... [[77.2, 68.1, 554.7], [1, 0, 0, 0]] ];
ENDMODULE
```

9.3 系统程序模块

定义

系统程序模块用于（预）定义系统特定数据对象（工具、焊接数据、移动数据..）、接口（打印机、日志文件..）等等。通常，系统程序模块在系统启动期间自动加载到任务缓冲区。

例如，

```
MODULE sysun1(SYSMODULE)
  ! Provide predefined variables
  VAR num n1 := 0;
  VAR num n2 := 0;
  VAR num n3 := 0;
  VAR pos p1 := [0, 0, 0];
  VAR pos p2 := [0, 0, 0];
  ...
  ! Define channels - open in init function
  VAR channel printer;
  VAR channel logfile;
  ...
  ! Define standard tools
  PERS pose bmtool := [...
  ! Define basic weld data records
  PERS wdrec wd1 := [ ...
  ! Define basic move data records
  PERS mvrec mv1 := [ ...
  ! Define home position - Sync. Pos. 3
  PERS robtarget home := [ ...
  ! Init procedure
  LOCAL PROC init()
    Open\write, printer, "/dev/lpr";
    Open\write, logfile, "/usr/pm2/log1"... ;
  ENDPROC
ENDMODULE
```

系统程序模块的选择将作为系统配置的一部分。

9.4 Nostepin模块

概述

通过在模块上设置参数NOSTEPIN，RAPID程序的逐步执行不会步进到程序中。程序的RAPID代码不会显示给用户。

修改在nostepin模块中的位置

程序步进时，将在所有移动指令前停止执行程序，但仅可能修改声明为程序参数的位置。将在程序编辑器中突出显示该参数。**ModPos**按钮变为活动状态。

变元	行为
作为程序参数的一个robtarget	整个程序都被执行后，程序将停止。然后，可能修改robtarget的位置。
作为程序参数的两个或更多robtarget	在第二个移动指令被执行前，程序将停止。然后，可能修改第一个robtarget的位置。 在第三个移动指令被执行前，程序将停止。然后，可能修改第二个robtarget的位置。 最终，整个程序都被执行后，程序将停止。然后，可能修改最后一个robtarget的位置。

注意

无法修改nostepin模块中声明的位置，在声明为程序参数的位置上采用Offs和RelTool有返回值程序时，也无法修改位置。

示例

```
MODULE My_Module(NOSTEPIN)
PROC MoveSquare(robtarget pos1, robtarget pos2, robtarget pos3)
  MoveJ pos1,v500,fine,tool0;
  !Before the next move instruction is run, the execution is stopped
       when stepping
  !Now you can modify the first position pos1
  MoveJ pos2,v500,fine,tool0;
  !Before the next move instruction is run, the execution is stopped
       when stepping
  !Now you can modify the second position pos2
  MoveJ pos3,v500,fine,tool0;
  !The third and last position pos3 can be modified when the whole
       procedure has been run
ENDPROC
ENDMODULE
```

此页刻意留白

10 语法概述

概述

每一规则或规则组前面都有引入该规则的程序段的引用。

第19页的字符集

```
<character> ::= --ISO 8859-1 (Latin-1)--
<newline> ::= -- newline control character --
<tab> ::= -- tab control character --
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<hex digit> ::= <digit> | A | B | C | D | E | F | a | b | c | d |
     e | f
<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<binary digit> ::= 0 | 1
<letter> ::=
  <upper case letter>
  | <lower case letters>
<upper case letter> ::=
  A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
  P | Q | R | S | T | U | V | W | X | Y | Z | À | Á | Â | Ã |
  Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í | Î | Ï | Ð | Ñ | Ò |
  Ó | Ô | Õ | Ö | Ø | Ù | Ú | Û | Ü | Ý | Þ | ß
<lower case letter> ::=
  a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
  p | q | r | s | t | u | v | w | x | y | z | ß | à | á | â |
  ã | ä | å | æ | ç | è | é | ê | ë | ì | í | î | ï | ð | ñ |
  ò | ó | ô | õ | ö | ø | ù | ú | û | ü | ý | þ | ÿ
```

第21页的标识符

```
<identifier> ::= <ident> | <ID>
<ident> ::= <letter> {<letter> | <digit> | '_'}
```

第23页的数字文字

```
<num literal> ::=
  <integer> [ <exponent> ]
    | <decimal integer>) [<exponent>]
    | <hex integer>
    | <octal integer>
    | <binary integer>
    | <integer> '.' [ <integer> ] [ <exponent> ]
    | [ <integer> ] '.' <integer> [ <exponent> ]
<integer> ::= <digit> {<digit>
<decimal integer> ::= '0' ('D' | 'd') <integer>
<hex integer> ::= '0' ('X' | 'x') <hex digit> {<hex digit>}
<octal integer> ::= '0' ('O' | 'o') <octal digit> {<octal digit>}
<binary integer> ::= '0' ('B' | 'b') <binary digit> {<binary digit>}
<exponent> ::= ('E' | 'e') ['+' | '-'] <integer>
```

第24页的布尔文字

```
<bool literal> ::= TRUE | FALSE
```

下一页继续

续前页

第25页的字符串文字

```
<string literal> ::= '"' { <character> | <character code> } '"'
<character code> ::= '\' <hex digit> <hex digit>
```

第28页的备注

```
<comment> ::= '!' { <character> | <tab> } <newline>
```

第29页的数据类型

```
<type definition> ::=
  [LOCAL] ( <record definition>
  | <alias definition> )
  | <comment>
  | <DN>
<record definition> ::=
  RECORD <identifier>
    <record component list>
  ENDRECORD
<record component list> ::=
  <record component definition>
  | <record component definition> <record component list>
<record component definition> ::=
  <data type> <record component name> ';'
<alias definition> ::=
  ALIAS <data type> <identifier> ';'
<data type> ::= <identifier>
```

第39页的数据声明

```
<data declaration> ::=
[LOCAL]
  (<variable declaration>
  | <persistent declaration>
  | <constant declaration>)
| TASK
  (<variable declaration>
  | <persistent declaration>)
| <comment>
| <DDN>
```

第44页的变量声明

```
<variable declaration> ::=
  VAR <data type> <variable definition> ';'
<variable definition> ::=
  <identifier> [ '{' <dim> { ',' <dim> } '}' ] [ ':=' <constant
       expression> ]
<dim> ::= <constant expression>
```

第45页的永久数据对象声明

```
<persistent declaration> ::=
  PERS <data type> <persistent definition> ';'
<persistent definition> ::=
  <identifier> [ '{' <dim> { ',' <dim> } '}' ] [ ':=' <literal
       expression> ]
```

下一页继续

注意

只有系统全局永久数据对象的文字表达式可忽略。

第47页的常量声明

```
<constant declaration> ::=
  CONST <data type> <constant definition> ';'
<constant definition> ::=
  <identifier> [ '{' <dim> { ',' <dim> } '}' ] ':=' <constant
       expression>
<dim> ::= <constant expression>
```

第49页的表达式

```
<expression> ::=
  <expr>
  | <EXP>
<expr> ::=
  [ NOT ] <logical term> { ( OR | XOR ) <logical term> }
<logical term> ::=
  <relation> { AND <relation> }
<relation> ::=
  <simple expr> [ <relop> <simple expr> ]
<simple expr> ::=
  [ <addop> ] <term> { <addop> <term> }
<term> ::=
  <primary> { <mulop> <primary> }
<primary> ::=
  <literal>
  | <variable>
  | <persistent>
  | <constant>
  | <parameter>
  | <function call>
  | <aggregate>
  | '(' <expr> ')'
<relop> ::= '<' | '<=' | '=' | '>' | '>=' | '<>'
<addop> ::= '+' | '-'
<mulop> ::= '*' | '/' | DIV | MOD
```

第51页的常量表达式

```
<constant expression> ::= <expression>
```

第52页的文字表达式

```
<literal expression> ::= <expression>
```

第53页的条件表达式

```
<conditional expression> ::= <expression>
```

第54页的文字

```
<literal> ::=
  <num literal>
  | <string literal>
  | <bool literal>
```

下一页继续

续前页

第55页的变量

```
<variable> ::=
  | <entire variable>
  | <variable element>
  | <variable component>
<entire variable> ::= <ident>
<variable element> ::= <entire variable> '{' <index list> '}'
<index list> ::= <expr> { ',' <expr> }
<variable component> ::= <variable> '.' <component name>
<component name> ::= <ident>
```

第56页的永久数据对象

```
<persistent> ::=
  <entire persistent>
  | <persistent element>
  | <persistent component>
```

第57页的常量

```
<constant> ::=
  <entire constant>
  | <constant element>
  | <constant component>
```

第58页的参数

```
<parameter> ::=
  <entire parameter>
  | <parameter element>
  | <parameter component>
```

第59页的聚合体

```
<aggregate> ::= '[' <expr> { ',' <expr> } ']'
```

第60页的函数调用

```
<function call> ::=
  <function> '(' [ <function argument list> ] ')'
<function> ::= <identifier>
<function argument list> ::=
  <first function argument> { <function argument>
<first function argument> ::=
  <required function argument>
  | <optional function argument>
  | <conditional function argument>
<function argument> ::=
  ',' <required function argument>
  | <optional function argument>
  | ',' <optional function argument>
  | <conditional function argument>
  | ',' <conditional function argument>
<required function argument> ::= [ <ident> ':=' ] <expr>
<optional function argument> ::= '\' <ident> [ ':=' <expr> ]
<conditional function argument> ::='\' <ident> '?' <parameter>
```

下一页继续

第65页的语句

```
<statement> ::=
  <simple statement>
  | <compound statement>
  | <label>
  | <comment>
  | <SMT>
<simple statement> ::=
  <assignment statement>
  | <procedure call>
  | <goto statement>
  | <return statement>
  | <raise statement>
  | <exit statement>
  | <retry statement>
  | <trynext statement>
  | <connect statement>
<compound statement> ::=
  <if statement>
  | <compact if statement>
  | <for statement>
  | <while statement>
  | <test statement>
```

第67页的语句表

```
<statement list> ::= { <statement> }
```

第68页的标签语句

```
<label> ::= <identifier> ':'
```

第69页的赋值语句

```
<assignment statement> ::=
  <assignment target> ':=' <expression> ';'
<assignment target> ::=
  <variable>
  | <persistent>
  | <parameter>
  | <VAR>
```

第70页的过程调用

```
<procedure call> ::=
  <procedure> [ <procedure argument list> ] ';'
<procedure> ::=
  <identifier>
  | '%' <expression> '%'
<procedure argument list> ::=
  <first procedure argument> { <procedure argu¬ment> }
<first procedure argument> ::=
  <required procedure argument>
  | <optional procedure argument>
  | <conditional procedure argument>
  | <ARG>
```

下一页继续

续前页

```
<procedure argument> ::=
  ',' <required procedure argument>
  | <optional procedure argument>
  | ',' <optional procedure argument>
  | <conditional procedure argument>
  | ',' <conditional procedure argument>
  | ',' <ARG>
<required procedure argument> ::=
  [ <identifier> ':=' ] <expression>
<optional procedure argument> ::=
  '\' <identifier> [ ':=' <expression> ]
<conditional procedure argument> ::=
  '\' <identifier> '?' ( <parameter> | <VAR> )
```

第72页的*Goto*语句

```
<goto statement> ::= GOTO <identifier> ';'
```

第73页的*Return*语句

```
<return statement> ::= RETURN [ <expression> ] ';'
```

第74页的*Raise*语句

```
<raise statement> ::= RAISE [ <error number> ] ';'
<error number> ::= <expression>
```

第75页的*Exit*语句

```
<exit statement> ::= EXIT ';'
```

第76页的*Retry*语句

```
<retry statement> ::= RETRY ';'
```

第77页的*Trynext*语句

```
<trynext statement> ::= TRYNEXT ';'
```

第78页的*Connect*语句

```
<connect statement> ::=
  CONNECT <connect target> WITH <trap> ';'
<connect target> ::=
  <variable>
  | <parameter>
  | <VAR>
<trap> ::= <identifier>
```

第79页的*IF*语句

```
<if statement> ::=
  IF <conditional expression> THEN
    <statement list>
  { ELSEIF <conditional expression> THEN
    <statement list>
    | <EIT> }
  [ ELSE
    <statement list> ]
  ENDIF
```

下一页继续

第80页的简洁IF语句

```
<compact if statement> ::=
  IF <conditional expression> ( <simple statement> | <SMT> )
```

第81页的For语句

```
<for statement> ::=
  FOR <loop variable> FROM <expression> TO <expression> [ STEP
       <expression> ] DO <statement list> ENDFOR
<loop variable> ::= <identifier>
```

第82页的While语句

```
<while statement> ::=
  WHILE <conditional expression> DO <statement list> ENDWHILE
```

第83页的Test语句

```
<test statement> ::=
  TEST <expression>
    { CASE <test value> { ',' <test value> } ':'
    <statement list> ) | <CSE> }
    [ DEFAULT ':'<statement list> ]
  ENDTEST
<test value> ::= <constant expression>
```

第85页的程序声明

```
<routine declaration> ::=
  [LOCAL] ( <procedure declaration> | <function declaration> |
       <trap declaration> )
  | <comment> | <RDN>
```

第86页的参数声明

```
<parameter list> ::=
  <first parameter declaration> { <next parameter declaration> }
<first parameter declaration> ::=
  <parameter declaration>
  | <optional parameter declaration>
  | <PAR>
<next parameter declaration> ::=
  ',' <parameter declaration>
  | <optional parameter declaration>
  | ',' <optional parameter declaration>
  | ',' <PAR>
<optional parameter declaration> ::=
  '\' ( <parameter declaration> | <ALT> ) { '|' ( <parameter
       declaration> |
  <ALT> ) }
<parameter declaration> ::=
  [ VAR | PERS | INOUT ] <data type> <identifier> [ '{' ( '*' {
       ',' '*' } ) |
  <DIM> '}' ]
  | 'switch' <identifier>
```

第89页的无返回值程序声明

```
<procedure declaration> ::=
  PROC <procedure name>
```

下一页继续

续前页

```
  '(' [ <parameter list> ] ')'
  <data declaration list>
  <statement list>
  [ BACKWARD <statement list> ]
  [ ERROR [ <error number list> ] <statement list> ]
  [ UNDO <statement list> ]
  ENDPROC
<procedure name> ::= <identifier>
<data declaration list> ::= { <data declaration> }
```

第*90*页的有返回值程序声明

```
<function declaration> ::=
  FUNC <data type>
  <function name>
  '(' [ <parameter list> ] ')'
  <data declaration list>
  <statement list>
  [ ERROR [ <error number list> ] <statement list> ]
  [ UNDO <statement list> ]
  ENDFUNC
<function name> ::= <identifier>
```

第*91*页的软中断程序声明

```
<trap declaration> ::=
  TRAP <trap name>
  <data declaration list>
  <statement list>
  [ ERROR [ <error number list> ] <statement list> ]
  [ UNDO <statement list> ]
  ENDTRAP
<trap name> ::= <identifier>
<error number list> ::=
  '(' <error number> { ',' <error number>} ')'
<error number> ::=
  <num literal>
  | <entire constant>
  | <entire variable>
  | <entire persistent>
```

第*116*页的模块声明

```
<module declaration> ::=
  MODULE <module name> [ <module attriutelist>]
  <type definition list>
  <data declaration list>
  <routine declaration list>
  ENDMODULE
<module name> ::= <identifier>
<module attribute list> ::=
  '(' <module attribute> { ',' <module attribute> } ')'
<module attribute> ::=
  SYSMODULE
  | NOVIEW
```

下一页继续

```
  | NOSTEPIN
  | VIEWONLY
  | READONLY
<type definition list> ::= { <type definition> }
<routine declaration list> ::= { <routine declaration> }
```

此页刻意留白

11 内置程序

概述

有关ref访问模式的更多信息，请参见第60页的函数调用。

> **注意**
>
> 请注意，RAPID程序无法具备REF参数。

标记anytype表明参数可为任意数据类型。

> **注意**
>
> 请注意，anytype仅为此特性的一个标记，不得与“实际”数据类型相混淆。也请注意，RAPID程序无法被赋予anytype参数。

Dim

Dim有返回值程序用于取得数组（datobj的尺寸，将返回指定维度的数组元素数。

```
FUNC num Dim (REF anytype datobj, num dimno)
```

合法dimno值：

值	描述
1	选择第一个数组维度
2	选择第二个数组维度
3	选择第三个数组维度

Present

Present有返回值程序用于测试参数（datobj）是否是存在的，请参见第86页的参数声明。如果datobj是不存在的可选参数，那么有返回值程序将返回FALSE，否则，将返回TRUE。

```
FUNC bool Present (REF anytype datobj)
```

Break

Break（断点）无返回值程序引起程序的执行暂时被停止。Break将用于RAPID程序代码调试。

```
PROC Break ()
```

IWatch

IWatch无返回值程序将激活指定的中断（ino）。随后，可利用ISleep无返回值程序再次让中断无效。

```
PROC IWatch (VAR intnum ino)
```

ISleep

ISleep无返回值程序将让指定的中断（ino）无效。随后，可利用IWatch无返回值程序再次激活中断。

```
PROC ISleep (VAR intnum ino)
```

下一页继续

续前页

IsPers

IsPers有返回值程序用于测试数据对象（datobj）是否是永久对象（或永久对象的别名）

（请参见第86页的参数声明。在此情况下，有返回值程序将返回TRUE，否则将返回FALSE。

```
FUNC bool IsPers (INOUT anytype datobj)
```

IsVar

IsVar有返回值程序用于测试数据对象（datobj）是否是变量（或变量的别名）（请参见第86页的参数声明）。在此情况下，该有返回值程序将返回TRUE，否则将返回FALSE。

```
FUNC bool IsVar (INOUT anytype datobj)
```

12 内置数据对象

错误

下表描述了属于内核的错误。

欲知所有错误（内核错误和RAPID错误）的列表，请参见技术参考手册 - *RAPID*指令、函数和数据类型。

对象名称	对象类型	数据类型	描述
ERRNO	变量[i]	errnum	最近期的错误编号
INTNO	变量i	intnum	最近期的中断
ERR_ALRDYCNT	常量	errnum	已关联的软中断程序和变量
ERR_ARGDUPCND	常量	errnum	复制现有条件参数
ERR_ARGNOTPER	常量	errnum	参数并非为一个永久数据对象引用
ERR_ARGNOTVAR	常量	errnum	参数并非为一个变量引用
ERR_CALLPROC	常量	errnum	运行时（后期绑定）的无返回值程序调用错误（语法，非无返回值程序）
ERR_CNTNOTVAR	常量	errnum	CONNECT目标并非为一个变量引用
ERR_DIVZERO	常量	errnum	除零
ERR_EXECPHR	常量	errnum	无法执行占位符
ERR_FNCNORET	常量	errnum	缺少返回值
ERR_ILLDIM	常量	errnum	超出范围的数组维度
ERR_ILLQUAT	常量	errnum	非法方位值
ERR_ILLRAISE	常量	errnum	超出范围的RAISE中的错误编号
ERR_INOISSAFE	常量	errnum	如果尝试暂时停用安全中断，则采用ISleep。
ERR_INOMAX	常量	errnum	无法获得更多的中断编号
ERR_MAXINTVAL	常量	errnum	整数值过大
ERR_NOTARR	常量	errnum	数据对象并非一个数组
ERR_NOTEQDIM	常量	errnum	混合数组维度
ERR_NOTINTVAL	常量	errnum	非整数值
ERR_NOTPRES	常量	errnum	参数不存在
ERR_OUTOFBND	常量	errnum	数组下标越界
ERR_REFUNKDAT	常量	errnum	未知的整个数据对象的引用
ERR_REFUNKFUN	常量	errnum	未知的有返回值程序的引用
ERR_REFUNKPRC	常量	errnum	在链接时或运行时（后期绑定），未知的无返回值程序的引用
ERR_REFUNKTRP	常量	errnum	未知的软中断程序的引用
ERR_STRTOOLNG	常量	errnum	字符串过长
ERR_UNKINO	常量	errnum	未知中断编号

[i] 只读，仅可由系统更新，无法由RAPID程序更新。

此页刻意留白

13 内置对象

定义

有三组内置对象：

- 语言内核保留对象
- 安装对象
- 用户安装对象

语言内核保留对象是系统的一部分，无法取消（或在配置中省去）。这组中的对象为指令Present、变量intno、errno以及更多其他对象。这组中的对象集与所有任务（多任务）和安装中的相同。

大多数安装对象由内部系统配置在首次系统启动时（或采用重启模式**Reset RAPID**时）安装，无法被取消（比如，指令MoveL、MoveJ ...）。在每次系统启动时，将按用户配置来安装输入/输出信号及机械单元对应的数据对象。

最后的组用户安装对象是RAPID模块中定义的对象，在首次系统启动时或采用重启模式**Reset RAPID**时按用户配置安装。

对象可为任意RAPID对象，即，无返回值程序、有返回值程序、记录、记录分量、别名、常量、变量或永久数据对象。永久数据对象和变量的对象值可改变，但代码本身不可改变，因此，内置的恒定声明机器人位置的修改位置（modpos）是不被允许的。

内置RAPID代码决不可被查看。

对象范围

对象范围表示对象显示范围。只要内置等级和引用应用之间的一个等级上没有另一个对象使用同一对象名称，那么，内置对象在任务中的所有其他等级均可显示。

下表展示了不同地点引用的对象的范围等级查找顺序。

对象使用地点：	自身程序	自身模块（局部声明）	程序中全局（在一个模块中全局声明）	内置对象
用户模块或系统模块中声明的程序	1	2	3	4
用户模块或系统模块中声明的数据或程序		1	2	3
用户安装模块中声明的程序	1	2		3
用户安装模块中声明的数据或程序		1		2
安装对象（仅适用于系统开发者）				1

有一些方式可以将运行时间的引用与范围外的对象（非有返回值程序）绑定。有关数据对象，请参见技术参考手册 - *RAPID*指令、函数和数据类型中的SetDataSearch的说明。有关带查找的无返回值程序应用后期绑定，请参见第*70*页的过程调用的说明。

下一页继续

续前页

内置数据对象持久性值

将在内置PERS或VAR对象安装时，设置其初始值。可以从常规程序改变初始值。即便常规程序被复位、清除或更换，对象也将一直保持最新值。重新初始化对象的唯一方式是使用重启模式**Reset RAPID**来复位系统，或改变配置（那么，将执行自动的**Reset RAPID**）。

注意

每项任务配单独值的内置VAR对象的值将在执行“PP到Main”时被复位。ERRNO是每项任务配单独值的内置VAR对象的示例。

注意

内置PERS对象不会像常规PERS对象那样，用最新值替代初始值。

定义用户安装对象的方式

对用户安装对象进行安装的唯一方式是在RAPID模块中定义对象，在系统参数*Task modules*中，以该模块的文件路径，创建一个新实例。然后，必须将属性*Storage*设为*Built-in*（参见技术参考手册 - 系统参数中的系统参数、类型*Controller*。也有名为*TextResource*的*Task modules*属性，该属性只对内置对象有效，这或许能在RAPID代码中为标识符采用基于国家语言或站点的名称，而不改变代码本身。在正常情况下，属性不得变更，但高级用户除外，请参见第*139*页的文本文件。

注意

在该模块安装时，必须让系统知道内置模块中使用的所有引用。

14 任务间对象

定义

有两组任务间对象：

- 安装共享对象
- 系统全局永久数据对象

安装共享对象将配置为供所有任务共享。这能够通过为一项以上任务反复使用RAPID代码来节约内存。这也是共享非值数据和半值数据的唯一方式，请参见第133页的内置数据对象。对象可为任意RAPID语言对象，即，无返回值程序、有返回值程序、常量、变量或永久数据对象。

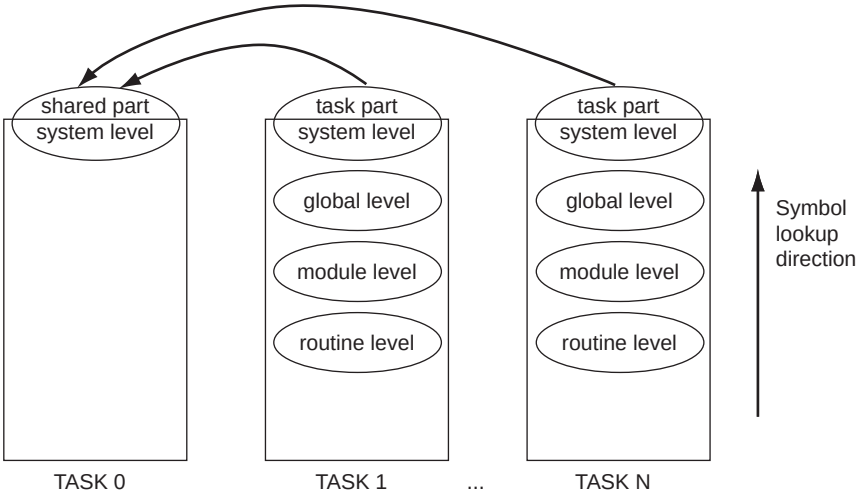
系统全局永久数据对象的当前值供那些声明为同名和同类型的所有任务共享。

符号等级

可在不同等级（一个程序、一个模块（局部）、一项任务的程序（一个模块中，被定义为全局）或系统等级），找到RAPID语言的符号。安装共享对象处于系统等级。

系统等级被分为两部分，即，共享部分和任务部分。任务部分的对象为该任务的局部对象，而共享部分的对象为所有任务的全局对象。

安装共享部分实际存在于任务0（共享任务）中，但也是每一任务的逻辑部分。



xx1300000278

符号查找将从引用对象的该位置（等级）开始，在未查找到的情况下，再从最近等级开始查找，如此类推。请参见前图的符号查找方向箭头。

数据对象处理

即便定义供数据对象共享，但其值可为任务中的局部值。对于安装系统变量errno、intno以及所有堆栈对象（程序中定义的对象）而言均如此。所有其他数据对象与其他任务共享值。这一情况要求仔细操控和读取这些值。

如果对象为原子型（数字型、布尔型...），那没什么问题。但如果对象并非原子型，那么请确保整个对象被读取/操控时，不受另一任务干扰。举例而言，如果对象为记录型，并且逐一指定了每一分量，那么从另一任务读取（两个记录分量的设置之间）将得到不一致的记录。

下一页继续

续前页

也请记住，可同时从一项以上任务调用一个程序，因此，程序应为重入型，即，采用局部堆栈对象（程序中声明的参数和数据对象）。

定义安装共享对象的方式

安装共享对象的唯一安装方式是在RAPID模块中定义对象，在系统参数中，用该模块的文件路径，创建一个新*Task*/*Automatic loading of Modules*实例。共享属性必须设为*YES*。请参见技术参考手册 - 系统参数中的系统参数域*Controller*。

系统全局永久数据对象

系统全局永久数据对象（比如，并非声明为任务或局部的对象）的当前值供那些声明同名和同类型的所有任务共享。即便声明所在的一个模块被取消，只要该模块不含该对象的最新声明，那么该对象仍存在。永久对象仅可为值型对象。

声明可以为永久对象指定初始值，但仅当模块首次安装或加载时，才会设置永久数据对象的初始值。

应用示例（若干初始值）：

任务1：
```
PERS tooldata tool1 := [...];
```
任务2：
```
PERS tooldata tool1 := [...];
```
请注意，不会以第二个加载模块中工具1的初始值来更新工具1的当前值。如果两个任务的初始值不同，这存在问题。该问题的解决方式为，仅在一个声明中指定初始值。

应用示例（一个初始值）：

任务1：
```
PERS tooldata tool1 := [...];
```
任务2：
```
PERS tooldata tool1;
```
加载两项任务后，将保证工具1的当前值等于任务1声明的初始值，不论模块加载顺序为何。建议为`tooldata`、`wobjdata`和`loaddata`等类型采用该方法。在运动任务中指定初始值和数据声明，省去其他任务的初始值。或许也可以完全不指定任何初始值。

应用示例（无初始值）：

任务1：
```
PERS num state;
```
任务2：
```
PERS num state;
```
在无初始值的情况下，状态的当前值将如变量一样进行初始化，在此情况下，状态将等于零。这一情况对任务间通信有用，在任务间通信中，当程序被保存或处于备用时，不得保存通信状态。

15 文本文件

定义

这是应用要求包含下列时应采用的最有效工具：

- 易换文本，比如，帮助和错误文本（客户也应能处理这类文件）。
- 内存节约，文本文件的文本字符串相较RAPID字符串，占用更少内存。

在文本文件中，在离线编辑器的帮助下，可使用ASCII字符串，并可从RAPID代码取得ASCII字符串。即便执行结果看上去完全不同，RAPID代码也不得以任何方式变更。

文本文件的语法

应用编程者必须构建一个（或数个）ASCII文件，即，包含文本资源字符串的文本文件。

文本文件组构如下：

```
<text_resource>::
# <comment>
<index1>: "<text_string>"
# <comment>
<index2>: "<text_string>"
...
```

参数	描述
<text_resource>	此名称标识了文本资源。此名称必须以“::”结束。如果系统中不存在此名称，那么系统将创建一个新文本资源；否则，文件中的索引将被添加到已存在的资源。应用开发者负责确保自己的资源不会与另一应用相冲突。一项很好的规则在于，将应用名作为资源名的前缀，比如，MYAPP_TXRES。资源名称不得超出80字符。请勿只使用字母数字来命名，以避免与系统资源相冲突。
<index>	此编号标识了文本资源中的<text_string>。此编号为应用定义编号，必须以“:”（冒号）结尾。
<text_string>	文本字符串必须起于一个新行，在该行末尾结束，或者如果文本字符串必须为数行，那么，必须插入换行字符串“\n”。
<comment>	注释通常起于一个新行，到达该行的末尾。注释前通常跟有“#”。注释不会加载到系统。

在程序执行期间检索文本

或许可以从RAPID代码检索文本字符串。为此采用有返回值程序TextGet和TextTabGet，请参见“RAPID支持函数”章节。

模块示例：write_from_file.mod

```
MODULE write_from_file
VAR num text_res_no;
VAR string text1;

PROC main()
IF TextTabFreeToUse("ACTION_TXRES") THEN
  TextTabInstall "HOME:/text_file.eng";
ENDIF

text_res_no := TextTabGet("ACTION_TXRES");
text1 := TextGet(text_res_no, 2);
```

下一页继续

续前页

```
    TPWrite text1; ! The word "Stop" will be printed.

    ENDPROC
    ENDMODULE
```

文本文件示例：text_file.eng

```
    #This is the text file .......
    ACTION_TXRES::
    # Start the activity
    1:
    Go
    # Stop the activity
    2:
    Stop
    3:
    Wait
    # Get Help
    10:
    Call_service_man
    #Restart the controller
    11:
    Restart
```

加载文本文件

可用RAPID指令TextTabInstall和有返回值程序TextTabFreeToUse来将文本文件加载到系统。

16 RAPID对象的存储分配

定义

电脑或控制器上存储的所有RAPID程序均为ASCII格式。当从计算机/控制器内存将RAPID程序加载到程序内存（内部格式）时，程序存储需要的内存空间比当前多四倍。

对于RAPID程序的内存优化，下文规定了一些常用指令、数据等的程序内存（内部格式（单位：字节））的存储分配。

对于其他指令或数据，在加载程序或程序模块后，可从操作消息10040，读取存储分配。

模块、程序、程序流和其他基本指令的存储分配

指令或数据	存储（单位：字节)
新空模块：MODULE module1 ... ENDMODULE	1732
无参数的新空无返回值程序：PROC proc1() ... ENDPROC	224
无参数的无返回值程序调用：proc1;	224
模块数值变量声明：VAR num reg1;	156
数值分配：reg1:=1;	44
简洁IF语句：IF reg1=1 proc1;	124
IF语句：IF reg1=1 THEN proc1; ELSE proc2; ENDIF	184
等待给定时间：WaitTime 1;	88
注释：! 0 - 7 chars (for every additional 4 chars)	36 (+4)
带0-80字符初始字符串值的模块字符串常量声明：CONST string string1 := "0-80 characters";	332
带0-80字符初始字符串值的模块字符串变量声明：VAR string string1 := "0-80 characters";	344
模块字符串变量声明：VAR string string1;	236
字符串分配：string1:= "0-80 characters";	52
在FlexPendant示教器上写入文本：TPWrite "0-80 characters";	176

Move指令的存储分配

指令或数据	存储（单位：字节)
模块机器人位置常量声明：`CONST robtarget p1 := [...];`	292
机械臂直线运动：`MoveL p1,v1000,z50,tool1;`	244
机械臂直线运动：`MoveL *,v1000,z50,tool1;`	312
机械臂圆周运动：`MoveC *,*,v1000,z50,tool1;`	432

输入/输出指令的存储分配

指令或数据	存储（单位：字节)
设置数字输出：`Set do1;`	88
设置数字输出：`SetDO do1,1;`	140
等到一个数字输入值较高时：`WaitDI di1,1;`	140

下一页继续

续前页

指令或数据	存储（单位：字节）
等到两个数字输入值较高时：`WaitUntil di1=1 AND di2=1;`	220

索引

Contact us

ABB AB
Discrete Automation and Motion
Robotics
S-721 68 VÄSTERÅS, Sweden
Telephone +46 (0) 21 344 400

ABB AS, Robotics
Discrete Automation and Motion
Nordlysvegen 7, N-4340 BRYNE, Norway
Box 265, N-4349 BRYNE, Norway
Telephone: +47 51489000

ABB Engineering (Shanghai) Ltd.
No. 4528 Kangxin Hingway
PuDong District
SHANGHAI 201319, China
Telephone: +86 21 6105 6666

www.abb.com/robotics

3HAC050946-010, Rev B, zh-cn