

VAL3 参考手册

第 7 版

您可在与控制箱一起交货的 CDROM 的 "readme.pdf" 文档中找到附录和勘误表

目录

1 - 引言	13
2 - VAL 3 语言基础知识.....	17
 2.1 软件应用.....	19
2.1.1 定义.....	19
2.1.2 默认内容.....	19
2.1.3 程序启动和终止.....	19
2.1.4 软件应用参数.....	19
2.1.4.1 长度单位.....	20
2.1.4.2 栈内存的数量.....	20
 2.2 程序.....	21
2.2.1 定义.....	21
2.2.2 重入程序.....	21
2.2.3 Start() 程序.....	21
2.2.4 Stop() 程序.....	21
2.2.5 程序控制指令.....	22
注释 //	22
call 程序	22
return	22
if 控制指令	23
while 控制指令	24
do ... until 控制指令	24
for 控制指令	25
switch 控制指令	26
 2.3 数据	28
2.3.1 定义.....	28
2.3.2 简单类型.....	28
2.3.3 结构类型.....	28
2.3.4 数据容器.....	29
 2.4 数据初始化	29
2.4.1 简单类型数据.....	29
2.4.2 结构类数据.....	29
 2.5 变量	30
2.5.1 定义.....	30
2.5.2 变量的作用范围	30
2.5.3 访问一个变量值	30
2.5.4 适用于所有变量的指令	31
num size (*)	31
bool isDefined (*)	31
bool insert (*)	32
bool delete (*)	33
num getData (string sDataName, *)	34

2.5.5 数组变量的特殊指令	35
void append (*)	35
num size (*, num nDimension)	35
void resize (*, num nDimension, num nSize)	35
2.5.6 集合变量的特殊指令	36
string first (*)	36
string next (*)	36
string last (*)	36
string prev (*)	36
2.6 程序参数	37
2.6.1 按元素值的参数	38
2.6.2 按元素引用的参数	38
2.6.3 按数组或集合引用的参数	39
3 - 简单类型	41
3.1 BOOL 类型	43
3.1.1 定义	43
3.1.2 运算符	43
3.2 NUM 类型	44
3.2.1 定义	44
3.2.2 运算符	45
3.2.3 指令	46
num sin (num nAngle)	46
num asin (num nValue)	46
num cos (num nAngle)	46
num acos (num nValue)	46
num tan (num nAngle)	46
num atan (num nValue)	47
num abs (num nValue)	47
num sqrt (num nValue)	47
num exp (num nValue)	47
num power (num nX, num nY)	47
num ln (num nValue)	48
num log (num nValue)	48
num roundUp (num nValue)	48
num roundDown (num nValue)	48
num round (num nValue)	48
num min (num nX, num nY)	49
num max (num nX, num nY)	49
num limit (num nValue, num nMin, num nMax)	49
num sel (bool bCondition, num nValue1, num nValue2)	49
3.3 位字段类型	50
3.3.1 定义	50
3.3.2 运算符	50
3.3.3 指令	50
num bNot (num nBitField)	50
num bAnd (num nBitField1, num nBitField2)	50
num bOr (num nBitField1, num nBitField2)	51
num bXor (num nBitField1, num nBitField2)	51
num toBinary (num nValue[], num nValueSize, string sDataFormat, num& nDataByte[])	52
num fromBinary (num nDataByte[], num nDataSize, string sDataFormat, num& nValue[])	52

3.4 STRING 类型	54
3.4.1 定义	54
3.4.2 运算符	54
3.4.3 指令	54
string toString (string sFormat, num nValue)	54
string toNum (string sString, num& nValue, bool& bReport)	55
string chr (num nCodePoint)	56
num asc (string sText, num nPosition)	57
string left (string sText, num nSize)	57
string right (string sText, num nSize)	57
string mid (string sText, num nSize, num nPosition)	57
string insert (string sText, string sInsertion, num nPosition)	58
string delete (string sText, num nSize, num nPosition)	58
string replace (string sText, string sReplacement, num nSize, num nPosition)	58
num find (string sText1, string sText2)	58
num len (string sText)	58
3.5 DIO 类型	59
3.5.1 定义	59
3.5.2 运算符	59
3.5.3 指令	60
void dioLink (dio& diVariable, dio diSource)	60
num dioGet (dio diArray[])	60
num dioSet (dio diArray[], num nValue)	61
num ioStatus (dio diInputOutput)	61
num ioStatus (dio diInputOutput, string& sDescription, string& sPhysicalPath)	62
3.6 AIO 类型	63
3.6.1 定义	63
3.6.2 指令	63
void aioLink (aio& aiVariable, aio aiSource)	63
num aioGet (aio aiInput)	63
num aioSet (aio aiOutput, num nValue)	63
num ioStatus (aio aiInputOutput)	64
num ioStatus (aio diInputOutput, string& sDescription, string& sPhysicalPath)	64
3.7 SIO 类型	65
3.7.1 定义	65
3.7.2 运算符	65
3.7.3 指令	66
void sioLink (sio& siVariable, sio siSource)	66
num clearBuffer (sio siVariable)	66
num sioGet (sio siInput, num& nData[])	66
num sioSet (sio siOutput, num& nData[])	66
num sioCtrl (sio siChannel, string nParameter, *value)	67
4 - 用户界面	69
4.1 用户页面	71
4.2 屏幕类型	71
4.2.1 用户屏幕的选择	71
4.2.2 用户屏幕上的写入	71
4.2.3 从用户屏幕读入	71

4.3 指令	72
void userPage() , void userPage(screen scPage) , void userPage(bool bFixed)	72
void gotoxy(num nX, num nY) , void gotoxy(screen scPage, num nX, num nY)	72
void cls() , void cls(screen scPage)	72
void setTextMode(num nMode) ,	
void setTextMode(screen scPage, num nMode)	72
num getDisplayLen(string sText)	73
void put(string sText) , void put(screen scPage, string sText)	
void put(num nValue) , void put(screen scPage, num nValue) ,	
void putln(string sText) , void putln(screen scPage, string sText) ,	
void putln(num nValue) , void putln(screen scPage, num nValue) ,	74
void title(string sText) , void title(screen scPage, string sText)	74
num get(string& sText) , num get(screen scPage, string& sText) ,	
num get(num& nValue) , num get(screen scPage, num& nValue) ,	
num get() , num get(screen scPage)	74
num getKey() , num getKey(screen scPage)	76
bool isKeyPressed(num nCode) ,	
bool isKeyPressed(screen scPage, num nCode)	76
void popUpMsg(string sText)	76
bool logMsg(string sText)	77
string getProfile()	77
num setProfile(string sUserLogin, string sUserPassword)	77
string getLanguage()	78
bool setLanguage(string sLanguage)	79
string getDate(string sFormat)	79
5 - 任务	81
5.1 定义	83
5.2 运行时错误后的重启	83
5.3 可见性	83
5.4 时序	84
5.5 同步任务	85
5.6 超时运行	85
5.7 输入 / 输出刷新	85
5.8 同步	86
5.9 资源分配	87
5.10 指令	88
void taskSuspend(string sName)	88
void taskResume(string sName, num nSkip)	88
void taskKill(string sName)	89
void setMutex(bool& bMutex)	89
string help(num nErrorCode)	89
num taskStatus(string sName)	90
void taskCreate string sName, num nPriority, program(...)	91
void taskCreateSync string sName, num nPeriod, bool& bOverrun, program(...)	92
void wait(bool bCondition)	93
void delay(num nSeconds)	93
num clock()	94
bool watch(bool bCondition, num nSeconds)	94

6 - 库	95
6.1 定义	97
6.2 接口	97
6.3 接口标识符	97
6.4 内容	97
6.5 加密	98
6.6 加载和卸载	99
6.7 指令	101
num identifier: libLoad (string sPath)	101
num identifier: libLoad (string sPath, string sPassword)	101
num identifier: libSave() , num libSave()	101
num libDelete (string sPath)	101
string identifier: libPath() , string libPath()	102
bool libList (string sPath, string& sContents[])	102
bool identifier: libExist (string sSymbolName)	102
7 - 用户类型	103
7.1 定义	105
7.2 创建	105
7.3 使用	105
8 - 机器人控制	107
8.1 指令	109
void disablePower()	109
void enablePower()	109
bool isPowered()	109
bool isCalibrated()	110
num workingMode() , num workingMode (num& nStatus)	110
num esStatus()	111
bool safetyFault (string& sSignalName)	111
num ioBusStatus (string& sErrorDescription[])	111
num getMonitorSpeed()	112
num setMonitorSpeed (num nSpeed)	112
string getVersion (string sComponent)	113
9 - 机器人手臂位置	115
9.1 引言	117
9.2 JOINT 类型	117
9.2.1 定义	117
9.2.2 运算符	118
9.2.3 指令	118
joint abs (joint jPosition)	118
joint herej()	119
bool isInRange (joint jPosition)	119
void setLatch (dio diInput) (CS8C only)	120
bool getLatch (joint& jPosition) (CS8C only)	120

9.3 TRSF 类型	121
9.3.1 定义	121
9.3.2 方向	122
9.3.3 运算符	124
9.3.4 指令	124
num distance (trsf trPosition1, trsf trPosition2)	124
trsf interpolateL (trsf trStart, trsf trEnd, num nPosition)	125
trsf interpolateC (trsf trStart, trsf trIntermediate, trsf trEnd, num nPosition)	126
trsf align (trsf trPosition, trsf Reference)	126
9.4 FRAME 类型	127
9.4.1 定义	127
9.4.2 使用	127
9.4.3 运算符	128
9.4.4 指令	128
num setFrame (point pOrigin, point pAxisOx, point pPlaneOxy, frame& fResult)	128
trsf position (frame fFrame, frame fReference)	128
void link (frame fFrame, frame fReference)	128
9.5 TOOL 类型	129
9.5.1 定义	129
9.5.2 使用	129
9.5.3 运算符	130
9.5.4 指令	130
void open (tool tTool)	130
void close (tool tTool)	131
trsf position (tool tTool, tool tReference)	131
void link (tool tTool, tool tReference)	131
9.6 POINT 类型	132
9.6.1 定义	132
9.6.2 运算符	132
9.6.3 指令	133
num distance (point pPosition1, point pPosition2)	133
point compose (point pPosition, frame fReference, trsf trTransformation)	133
point appro (point pPosition, trsf trTransformation)	134
point here (tool tTool, frame fReference)	134
point jointToPoint (tool tTool, frame fReference, joint jPosition)	134
bool pointToJoint (tool tTool, joint jInitial, point pPosition, joint& jResult)	135
trsf position (point pPosition, frame fReference)	135
void link (point pPoint, frame fReference)	135
9.7 CONFIG 类型	136
9.7.1 引言	136
9.7.2 定义	136
9.7.3 运算符	137
9.7.4 姿态 (RX/TX 机器人手臂)	138
9.7.4.1 肩部姿态	138
9.7.4.2 肘部姿态	139
9.7.4.3 腕部姿态	139
9.7.5 姿态 (RS/TS 机器人手臂)	140
9.7.6 指令	140
config config (joint jPosition)	140

10 - 运动控制	141
 10.1 轨迹控制.....	143
10.1.1 运动类型: 点到点, 直线, 圆周	143
10.1.2 运动连接: 轨迹混合	145
10.1.2.1 轨迹混合	145
10.1.2.2 取消混合	146
10.1.2.3 关节轨迹混合, 笛卡尔轨迹混合	146
10.1.3 运动恢复	147
10.1.4 笛卡尔运动的特性 (直线, 圆)	148
10.1.4.1 运动方向插值.....	148
10.1.4.2 姿态变化 (RX/TX 机器人手臂).....	150
10.1.4.3 奇点 (RX/TX 机器人手臂)	152
 10.2 运动预处理	152
10.2.1 原理.....	152
10.2.2 预测和轨迹混合	152
10.2.3 同步.....	153
 10.3 速度监控.....	154
10.3.1 原理.....	154
10.3.2 简单调节	154
10.3.3 高级调节	155
10.3.4 工作范围错误.....	155
 10.4 实时运动控制	155
 10.5 MDESC 类型	156
10.5.1 定义.....	156
10.5.2 运算符	156
 10.6 运动指令.....	157
num movej (joint jPosition, tool tTool, mdesc mDesc)	157
num movej (point pPosition, tool tTool, mdesc mDesc)	157
num movel (point pPosition, tool tTool, mdesc mDesc)	157
num movec (point pIntermediate, point pTarget, tool tTool, mdesc mDesc)	158
void stopMove()	159
void resetMotion() , void resetMotion (joint jStartingPoint)	159
void restartMove()	160
void waitEndMove()	160
bool isEmpty()	161
bool isSettled()	161
void autoConnectMove (bool bActive), bool autoConnectMove()	161
num getSpeed (tool tTool)	162
joint getPositionErr()	162
void getJointForce (num& nForce)	162
num getMoveld()	162
num setMoveld (num nMoveld)	163

11 - 可选项	165
 11.1 力控制的柔性运动.....	167
11.1.1 原理	167
11.1.2 编程	167
11.1.3 力的控制	167
11.1.4 限制	168
11.1.5 指令	168
num movejf (joint jPosition, tool tTool, mdesc mDesc, num nForce)	168
num movelf (point pPosition, tool tTool, mdesc mDesc, num nForce)	169
bool isCompliant()	169
 11.2 ALTER: 路径实时控制	170
11.2.1 原理	170
11.2.2 编程	170
11.2.3 限制	170
11.2.4 安全须知	171
11.2.5 限制	171
11.2.6 指令	171
num alterMovej (joint jPosition, tool tTool, mdesc mDesc)	171
num alterMovej (point pPosition, tool tTool, mdesc mDesc)	171
num alterMovel (point pPosition, tool tTool, mdesc mDesc)	172
num alterMovec (point pIntermediate, point pTarget, tool tTool, mdesc mDesc)	172
num alterBegin (frame fAlterReference, mdesc mMaxVelocity)	173
num alterBegin (tool tAlterReference, mdesc mMaxVelocity)	173
num alterEnd()	174
num alter (trsfl trAlteration)	174
num alterStopTime()	175
 11.3 OEM 许可证管理	176
11.3.1 原理	176
11.3.2 指令	176
string getLicence (string sOemLicenceName, string sOemPassword)	176
 11.4 绝对机器人	177
11.4.1 原理	177
11.4.2 操作	177
11.4.3 限制	177
11.4.4 指令	178
void getDH (num& theta[], num& d[], num& a[], num& alpha[], num& beta[])	178
void getDefaultDH (num& theta[], num& d[], num& a[], num& alpha[], num& beta[])	178
bool setDH (num& theta[], num& d[], num& a[], num& b[], num& alpha[], num& beta[])	178
 11.5 连续轴	179
11.5.1 原理	179
11.5.2 指令	179
joint resetTurn (joint jReference)	179

12 - 附录	181
12.1 运行时错误代码	183
12.2 控制面板键盘按键代码	184
13 - 插图	185
14 - 索引	187

章节 1

引言

VAL 3 是一种设计用于在各种应用中控制史陶比尔 **Stäubli** 机器人的高级程序语言。

VAL 3 语言包含了标准实时控制高级计算机程序语言，和机器人工作单元专用的功能：

- 机器人控制工具
- 几何建模工具
- 输入 / 输出控制工具

此参考指南旨在解释机器人编程时必须知道的概念，并根据下列类型分类详细解释 **VAL 3** 语言指令：

- **VAL3 基础知识**
- 简单类型
- 用户界面
- 任务
- 库
- 用户类型
- 机器人控制
- 手臂位置
- 运动控制
- 屏幕类型：用于手动示教盒 (MCP) 的屏幕显示

每条指令和它的语句格式显示在目录中用于快速查阅。

章节 2

VAL 3 语言基础知识

VAL 3 编程语言由软件应用组成。一个**VAL 3** 软件应用包括程序和数据。一个**VAL 3** 软件应用也可以引用其他被用作库或作为用户类型定义的软件应用。

2.1. 软件应用

2.1.1. 定义

一个**VAL 3** 软件应用是一个自包含的软件包，设计用于控制机器人和与一个控制器相关联的输入输出。一个**VAL 3** 软件应用包括以下要素：

- 一组**程序**: 要执行的**VAL 3** 指令
- 一组全局变量: 在软件应用中所有程序共享的数据
- 一组**库**: 用来共享程序和 / 或数据的外部软件应用
- 一组用户类型: 在软件应用中用作定义结构数据的模板的外部软件应用

当一个软件应用正在执行时，它还包括：

- 一组**任务**: 同时执行的程序

2.1.2. 默认内容

通过复制预定义内容，软件应用模板来建立一个新的**VAL 3** 软件应用。可以创建新用户特定模板。它们简单地由一个在控制器的一个专门文件夹中的标准**VAL 3** 软件应用组成。

只有当一个**VAL 3** 软件应用同时包含一个 **start()** 和一个 **stop()** 程序时才能启动。若无 **start()** 和 **stop()** 程序，一个**VAL 3** 软件应用只能被用作一个库或一个用户类型定义。可以定义仅包含数据，或仅包含程序的软件应用。

2.1.3. 程序启动和终止

控制器管理一个**VAL 3** 软件应用的启动。它可以是一个来自**MCP** 用户界面的用户请求，或是作为启动过程的自动部分。

每次只能启动一个**VAL 3** 软件应用。不过，一个软件应用可以同时使用多个其他的软件应用（例如库），和启动多个不同的执行任务。

当一个**VAL 3** 软件应用运行时，它的 **start()** 程序被执行。

最后一项任务完成时，**VAL 3** 软件应用自动停止：然后就执行 **stop()** 程序。如果仍有任务未执行，由程序库创建的所有任务便都按照其被创建时的相反顺序被一一删除。

如果**VAL 3** 软件应用的停止是通过用户界面 (**MCP**) 进行的，启动任务（如果还存在的话）就立即被摧毁。接下来执行 **stop()** 程序，然后所有尚未执行的软件应用的任务按照其被创建时的反向顺序被删除。

2.1.4. 软件应用参数

以下参数可用来设置一个**VAL 3** 软件应用：

- 长度单位
- 内存堆栈的容量

这些参数不能通过**VAL 3** 指令来访问，但是可以通过**MCP** 用户界面或使用**Stäubli Robotics Suite** 中的**VAL 3 Studio** 来修改。

2.1.4.1. 长度单位

在 **VAL 3** 软件应用中，长度单位为毫米或英寸。**VAL 3** 的几何数据类型使用此单位：坐标系、点、关节（对于线性轴）、变换、工具、轨迹混合。

一个软件应用的长度单位在其创建时被定义，之后不可再更改。

2.1.4.2. 栈内存的数量

对于每个要储存的 **VAL 3** 任务都需要一定的内存：

- 调用栈（在这个任务中将要执行的程序列表）
- 调用栈的每个程序的参数
- 调用栈的每个程序的局部变量

默认情况下，每个任务有栈内存的 **5000** 个字节。通常不需要改变此参数。

这个内存数量对于包含大量局部变量或递归算法的软件应用可能会不足：

在此情况下，必须使用 **MCP** 用户界面或 **Stäubli Robotics Suite** 中的 **VAL 3 Studio** 来增加栈内存的数量，或者通过减少在调用栈中的程序数目或使用全局变量而不是局部变量来优化软件应用。

2.2. 程序

2.2.1. 定义

一个程序是要被执行的一系列 **VAL 3** 指令。

一个程序由下列部分构成:

- 指令顺序: 要执行的 **VAL 3** 指令
- 一组**局部变量**: 程序内部数据
- 一组参数: 程序调用时给程序提供的数据

程序用来将在一个软件应用中多点处被执行的指令顺序组成组。为了节省编程时间, 它们也简化软件应用的结构, 方便编程和维护, 并提高可读性。

一个程序的指令数仅受系统中的可用存储空间的限制。

局部变量和参数的数量仅受此软件应用的栈内存的大小的限制(见第 2.1.4.2 章节)。

2.2.2. 重入程序

程序可重入, 也就是说一个程序可递归调用其本身 (**call** 指令), 或是同时被多个任务所调用。一个程序的每个调用都使用自己专有的局部变量和参数。不可能在这同一程序的两个不同调用之间交互。

2.2.3. START() 程序

start() 程序是当 **VAL 3** 软件应用运行时调用的程序。它不能含有任何参数。

这个程序通常包含执行此软件应用所需要的所有运算: 全局变量和输出的初始化, 创建一个软件应用任务, 等等。

只要其他软件应用任务仍然运行, 此软件应用就不在 **start()** 程序结束处停止。

start() 程序可以在一个程序 (**call** 指令) 中被调用, 就像任意一个其它程序一样。

2.2.4. STOP() 程序

stop() 程序是在 **VAL 3** 软件应用停止时被调用的程序。它不能含有任何参数。

此程序一般包括正确结束此软件应用所需的所有运算: 按照一个合适的顺序, 重置输出和停止软件应用任务, 等等。

就像其他程序一样, **stop()** 程序可以在一个程序中被调用 (**call** 指令), 但是调用 **stop()** 程序不能停止此软件应用。

2.2.5. 程序控制指令

注释 //

句法

// <String>

功能

« // » 开始的一行指令不被执行，在下一行执行重新开始。« // » 不能在一行的中间使用，它们必须是这一行的首字符。

例如

// 这是一个注释例子

call 程序

句法

call program([parameter1][,parameter2])

功能

调用指令执行一个用户定义的程序。在这个程序名后的表达式的数量和类型必须与该程序的接口匹配。指定作为参数的表达式按它们被指定的顺序首先被执行。然后初始化局部变量，并且启动该程序的执行命令和它的开始指令。

当程序执行一个返回或一个结束指令时，就完成了一个调用执行。

例如

```
// 调用 pick() 和 place() 程序用于在 1 到 10 之间的 i,j
for i = 1 to 10
  for j = 1 to 10
    call pick(pPallet1[i,j]),
    call place(pPallet2[i,j])
  endFor
endFor
```

return

句法

return

功能

返回指令立即结束当前程序的执行。如果此程序已经被一个 **call** 调用，在这个调用程序中的 **call** 之后执行重新开始。否则（如果该程序是 **start()** 程序或者是一个任务的开始点），当前任务就结束。返回指令与在这个程序结尾处的结束指令效果完全相同。

当一个程序的执行总是随结束指令结束时，往往更容易理解和维护。那么在一个程序中使用一个返回指令是不适当的。

if 控制指令

句法

```
if <bool bCondition>
  <instructions>
[elseif <bool bAlternateCondition1>
  <instructions>]
...
[elseif <bool bAlternateConditionN>
  <instructions>]
[else
  <instructions>]
endif
```

功能

if...elseif...else...endif 流程连续判断由关键词 **if** 或 **elseif** 标记的布尔表达式，直至一个表达式为真。然后，在布尔表达式之后的指令被执行，直至下一个 **elseif**、**else** 或 **endif** 关键词。在 **endif** 关键词后，程序继续执行。如果由 **if** 或 **elseif** 表示的所有布尔表达式为假，在关键词 **else** 和 **endif** 之间的指令就被执行（如果关键词 **else** 存在的话）。在 **endif** 关键词后，该程序重新开始执行。

在一个 **if...endif** 流程中，**elseif** 表达式的数目无限制。

当测试到一个单一表达式可能有不同的数值时，**if...elseif...else...endif** 流程可以用 **switch...case...default...endSwitch** 流程来替代。

例如

此程序将写在一个 **string(sDay)** 中的一天转变成一个 **num(nDay)**。

```
put("Enter a day: ")
get(sDay)
if sDay=="Monday"
  nDay=1
elseif sDay=="Tuesday"
  nDay=2
elseif sDay=="Wednesday"
  nDay=3
elseif sDay=="Thursday"
  nDay=4
elseif sDay=="Friday"
  nDay=5
else
  // 周末!
  nDay=0
endif
```

另见

switch 控制指令

while 控制指令

句法

```
while <bool bCondition>
  <instructions>
endWhile
```

功能

在 **while** 和 **endWhile** 之间的指令都是在布尔表达式 **bCondition** 是 (**true**) 时执行的。
如果布尔表达式 **bCondition** 在第一次求值时不是真的，那么在 **while** 和 **endWhile** 之间的指令将不被执行。

参数

bool bCondition	需要求值的布尔表达式
------------------------	------------

例如

```
// 只要机器人运动，此简单程序就使得一个信号闪烁
diLamp = false
while (isSettled() ==false)
// 倒置 diLamp 的值: true false
diLamp = !diLamp
// 等待 1/2 s
delay(0.5)
endWhile
diLamp = false
```

do ... until 控制指令

句法

```
do
  <instructions>
until <bool bCondition>
```

功能

执行在 **do** 和 **until** 之间的指令，直至布尔表达式 **bCondition** 为 (**true**)。

如果布尔表达式 **bCondition** 在第一次求值期间是真的，那么在 **do** 和 **until** 之间的指令只执行一次。

参数

bool bCondition	需要求值的布尔表达式
------------------------	------------

例如

```
// 该程序循环，直到按下回车键
do
// 等待点击一个按键
nKey = get()
// 测试 Enter 按键的代码
until (nKey == 270)
```

for 控制指令

句法

```
for <num nCounter> = <num nBeginning> to <num nEnd> [step <num nStep>]
  <instructions>
endFor
```

功能

在 **for** 和 **endFor** 之间的指令被执行直到 **nCounter** 超出指定 **nEnd** 的数值。

nCounter 被数值 **nBeginning** 初始化。如果 **nBeginning** 超过 **nEnd**, 那么在 **for** 和 **endFor** 之间的指令就不被执行。每次循环, **nCounter** 增加数值 **nStep**, 如果 **nCounter** 不超过 **nEnd**, 重复在 **for** 和 **endFor** 之间的指令。假如 **nStep** 是正的, 当 **nCounter** 大于 **nEnd** 时, **for** 循环停止。假如 **nStep** 是负的, 当 **nCounter** 小于 **nEnd** 时, **for** 循环停止。

参数

num nCounter	num 类型的变量用作计数器
num nBeginning	数值表达式用来初始化计数器
num nEnd	数值表达式用于循环结束测试
[num nStep]	数值表达式用于计数器的递增

例如

```
// 此程序使 1 轴以每 -10° 从 -90° 旋转到 +90°
for nPos = 90 to -90 step -10
  jDest.j1 = nPos
  movej(jDest, flange, mNomSpeed)
  waitEndMove()
endFor
```

switch 控制指令

句法

```
switch <expression>
case <value1> [, <value2>]
<instructions1-2>
break
[case <value3> [, <value4>]
<instructions3-4>
break ]
[default
<Default Instructions>
break ]
endSwitch
```

功能

switch...case...default...endSwitch 流程连续判断由 **case** 关键词表示的表达式，直至一个表达式等于一个 **switch** 关键词后的初始表达式。

然后表达式后的指令被执行，直至关键词 **break**。在 **endSwitch** 关键词后，程序继续执行。

如果没有任何表达式 **case** 等于初始表达式 **switch**，在关键词 **default** 和 **endSwitch** 之间的指令被执行（如果关键词 **default** 存在）。

在一个 **switch...endSwitch** 流程中，**case** 表达式的数目无限制。**case** 关键词后的表达式必须与关键词 **switch** 后的表达式为相同类型。

switch...case...default...endSwitch 流程与 **if...elseif...else...endif** 流程非常类似。

它不仅接受布尔表达式，也接受任何支持标准运算符 "is equal to" "==" 的表达式。

例如

此程序读取对应于一个按键的 **num(nMenu)**，并因此修改一个 **string s**。

```
nMenu = get()
switch nMenu
case 271
s = "Menu 1"
break
case 272
s = "Menu 2"
break
case 273, 274, 275, 276, 277, 278
s = "Menu 3 to 8"
break
default
s = "this key is not a menu key"
break
endSwitch
```

此程序将写在一个 **string(sDay)** 中的一天转变成一个 **num(nDay)**。

```
put("Enter a day: ")
get(sDay)
switch sDay
case "Monday"
nDay=1
break
case "Tuesday"
nDay=2
break
case "Wednesday"
```

```
nDay=3
break
case "Thursday"
    nDay=4
break
case "Friday"
    nDay=5
break
default
    // 这不是工作日 !
    nDay=0
break
endSwitch
```

2.3. 数据

2.3.1. 定义

一个数据是作为参数或 **VAL 3** 指令结果的一组数值。

一个数据由下列部分组成：

- 一组值
- 一个类型，定义可能的值并允许对数据进行运算。布尔值，数字和字符串是最简单的数据类型
- 一个容器，定义数值被存储在数据中的方式。元素、数组、集合在 **VAL 3** 中都是可能的数据容器

2.3.2. 简单类型

VAL 3 语言支持以下简单类型：

- **bool** 类型：用于所有布尔值（真 / 假）
- **num** 类型：用于所有数值（整数或浮点数）
- **string** 类型：用于字符串（Unicode 编码）
- **dio** 类型：用于数字输入 / 输出
- **aio** 类型：用于所有数字的输入 / 输出值（模拟的或数字的）
- **sio** 类型：用于串口上的输入 / 输出和以太网套接字
- **screen** 类型：用于 MCP 的屏幕显示和键盘存取

在本资料中，变量类型由其名称中的小写首字母来表示的：

- **bVariable** 是 **bool** 型变量
- **nVariable** 是 **num** 型变量
- **sVariable** 是 **string** 型变量
- **diVariable** 是 **dio** 型变量
- **aiVariable** 是 **aio** 型变量
- **siVariable** 是 **sio** 型变量
- **scVariable** 是 **screen** 型变量

2.3.3. 结构类型

一个结构类型集合多个更简单类型而形成一个新的，更高级别的类型。每个子类型都有一个名称，可以作为一个结构字段单独对它们进行访问。软件应用中的适当类型以某种方式组织数据，使得计算和程序演变更容易。

VAL 3 语言支持下述由简单类型组成的结构类型：

- **trsf** 类型：用于笛卡尔几何变换
- **frame** 类型：用于笛卡尔几何坐标系
- **tool** 类型：用于安装在机器人上的工具
- **point** 类型：用于一个工具的笛卡尔位置
- **joint** 类型：用于机器人轴的位置
- **config** 类型：用于机器人的形态
- **mdesc** 类型：用于机器人的移动参数

VAL 3 语言还支持用户类型，此类型将 **VAL 3** 简单、结构、或者其他用户类型组合成一个新类型。用户类型可完全作为一个标准类型用在一个应用中。

在本资料中，变量类型由其名称中的小写首字母来表示的：

- **trVariable** 是 **trsf** 型变量
- **fVariable** 是 **frame** 型变量
- **tVariable** 是 **tool** 型变量
- **pVariable** 是 **point** 型变量
- **jVariable** 是 **joint** 型变量
- **cVariable** 是 **config** 型变量
- **mVariable** 是 **mdesc** 型变量

2.3.4. 数据容器

数据容器定义数值被存储在数据中的方式:

- 一个简单的元素容器由一个单值组成。真(布尔值), 0(数字), '文本'(字符串)均有一个元素容器。
- 一组由 1, 2 或 3 整数指数识别的数值组成一个'数组'容器。在数组中的初始指数总为 0。
- 一组由字符串键识别的数值组成一个'集合'容器。任何非空字符串可以被用作数值标识符。

一个单一值的一维数组(指数 0)被认作一个元素容器。

在本资料中, 当需要时候, 变量的容器用该变量的名字来识别:

- **s1dArray** 是一个 **string** 类型的一维数组
- **s2dArray** 是一个 **string** 类型的二维数组
- **s3dArray** 是一个 **string** 类型的三维数组
- **sColl** 是一个 **string** 类型的集合

有些指令(来处理数组或集合)与数据类型无关。那么在本资料中, 该类型就由一个星号代替: '*'.

2.4. 数据初始化

2.4.1. 简单类型数据

一个简单类型数据初始化的精确句法在描述每个简单类型的章节中有特定描述。一个数组或一个集合必须按元素逐个初始化。对于一个 **bool**, 初始值为 **false**, 对于一个 **num** 为 0, 对于一个字符串为 ""(空串)。

例如

在这个示例中, **bBool** 是布尔变量, **nPi** 为数值变量, **sString** 为字符串变量。

```
bBool = true
nPi = 3.141592653
sString = "this is a string"
```

2.4.2. 结构类数据

一个结构类数据的值由在括号之间, 以逗号分隔的字段值的顺序来确定。空字段值用 0 替代。流程的顺序将在介绍各种结构类型的章节中详细说明。一个结构的值可能包括其他嵌套子结构的数值。一个结构类型的一个数组或一个集合必须按元素逐个初始化。

例如

点类型由一个 **trsF** 和一个姿态类型组成。一个点变量可以如下所示被初始化:

```
pPosition = {{100, -50, 200, 0, 0, 0}, {sfree, efree, wfree}}
```

该点的变换也可以被初始化:

```
pPosition.trsF = {100, -50, 200, , ,}
```

2.5. 变量

2.5.1. 定义

一个变量是在一个应用或一个程序中根据其名称编号的数据。

变量具有特点:

- 一个名称: 字符串
- 一个作用域: 变量可以被访问的地方 (在单个程序中, 在一个软件应用中由程序共享的, 或在软件应用之间共享的)
- 一组值
- 一个数据类型 (简单类型或结构类型)
- 一个数据容器 (元素、数组或集合)

一个变量名是一串包含在 "**a..zA..Z0..9_**" 中的从 **1** 到 **15** 个字符, 且由字母开头。

2.5.2. 变量的作用范围

变量作用范围为:

- 全局变量: 这个软件应用的所有程序都可以使用这个变量, 或者
- 局部变量: 此变量只能通过程序被访问, 在该程序中变量已被声明

当全局变量和局部变量名称相同时, 申明局部变量的程序将使用局部变量, 无法访问全局变量。

当一个软件应用被用作一个库时, 每个全局变量都可以被申明为公共的或私有的。一个公共变量可以通过使用库的软件应用进行访问, 私有变量只能在库中被访问。

2.5.3. 访问一个变量值

对一个变量值的访问视其容器而定:

- 一个元素容器的数值可用该变量名称进行访问 (无方括号): **nVariable**。
- 一个数组中的数值可用其在变量名之后方括号之间的数值索引来访问: **n1dArray[nIndex]**, **n2dArray[nIndex1, nIndex2]**, **n3dArray[nIndex1, nIndex2, nIndex3]**。
- 一个集合的数值可用其在变量名之后方括号之间的键来访问: **nCollection[sKey]**

一个单一值的一维数组(指数 0)被认作一个元素容器。其数值可以不用括号被访问: **n1dArray** 等同于 **n1dArray[0]**。用来访问一个数组中的数值的数字指数被四舍五入至最接近的整数值: **n2dArray[5.01, 6.99]** 等同于 **n2dArray[5, 7]** 用于访问一个表中的数值的指数介于 0 和维数减 1 之间。

结构类型变量的字段可用一个在'.'后加字段名来进行访问: **pPoint.trsf.x** 指的是点数据 **pPointx** 字段的'x'字段值。

例如

具不同容器的简单变量的初始化:

```
nPi = 3.141592653
sMonth[0] = "January"
sProductName["D 243 064 40 A"] = "VAL 3 CdRom"
```

2.5.4. 适用于所有变量的指令

num **size(*)**

功能

该指令返回用该变量可访问的值数:

- 一个元素容器变量的大小是 1。
- 一个一维数组的大小就是该数组中的元素的数目。
- 一个集合的大小就是该集合中的元素的数目。

对于二维和三维数组，大小指令需要一个第二参数来指定维数。对于一个一维数组，**size(s1dArray)** 等于 **size(s1dArray, 1)**。

被传递为数组引用的一个一维数组参数的大小取决于调用该程序时指定的索引。只有从指定索引开始的数组部分才可以在子程序中被访问: **size(s1dArray[nIndex]) = size(s1dArray) - nIndex.**

参数

variable

任意类型的变量

例如

确定大小的变量必须不用括号被指定:

```
// OK
nNbElements=size(sCollection)
// 编译错误: 不应有的键
nNbElements=size(sCollection[sKey])
```

对于一个一维数组，一个索引指明子数组的起始: **size(s1dArray[nIndex])** 是以 **nIndex** 索引开始的子数组的大小。

bool **isDefined(*)**

功能

如果指定的元素是在一个数组或一个集合中定义的，该指令返回 **true**，如果该元素没有被定义则返回 **false**。

它可以用来测试一个元素是否在一个集合中被定义；它还可以用来测试是否一个库执行它接口的一个变量。它还根据接口是新版本还是老版本，来处理一个库接口的更新，适配它的应用。

例如

此示例将一个新的 **article** key 加在一个集合中。

```
// 请求一个新引用名
put("New reference ?")
get(sReference)
if isDefined(sReferenceColl[sReference])==true
  putln(" 错误: reference already defined")
else
// 将 new article 添加在该集合中
  insert(sReferenceColl[sReference])
endif
```

该示例测试一个库的接口。

```
// 载入 part 库
nLoadCode = part:libLoad(sPartPath)
// 在该库的版本 1 中 part:sVersion 没有被定义
// 测试如果该库定义了它
if (nLoadCode==0) or (nLoadCode==11)
  if (isDefined(part:sVersion)==false)
```

```
// 初始版
sLibVersion = "v1.0"
else
    sLibVersion = part:sVersion
endiff
endiff
```

bool insert(*)

功能

这条指令创建变量类型的一个新值，并将它存储在变量容器中。新的值被该类型的默认值初始化。变量的大小就增加了 1。

对于一个一维数组，新值被插入在指定的索引位置。索引位置可能等于该数组的大小：那么插入就在该数组结尾处完成。`"insert(s1dArray[size(s1dArray)])"` 等同于 `"append(s1dArray)"`。

对于集合，只有当键还没有在该集合中被使用时，插入才会被接受。新的值与该指定的键相关，函数返回 `true`。如果键已被使用，指令不产生效果，返回 `false`。指令 `isDefined()` 可以用来检查是否一个键已被用在一个集合中。

该指令不支持二维和三维的数组（使用 `resize()` 指令代替）和局部数组变量。一个变量的大小不能超过 9999 值。当该变量的大小超过此限制，就产生一个运行时错误。

插入指令分配系统内存。无法保证内存分配的性能。鉴于此性能问题，应该避免在 VAL 3 软件应用中大量使用 `insert()`。

例如

此示例将一个新的 article 加在列表中。

```
// 请求一个新 article 名
put("New article ?")
get(sArticleName)
putln("")
// 请求列表中的位置
put("位置 ?")
get(n索引)
if (n索引<0) or (n索引>size(sArticleList))
    putln("错误 : invalid position")
else
    // 将 new article 添加在该列表中
    insert(sArticleList[nIndex])
    sArticleList[nIndex] = sArticleName
endiff
```

此示例将一个新的 article key 加在一个集合中。

```
// 请求一个新 article 名
put("New article ?")
get(sArticleName)
if isDefined(sArticleColl[sArticleName]) == true
    putln("错误 : reference already defined")
else
    // 将 new article 添加在该集合中
    insert(sArticleColl[sArticleName])
endiff
```

bool **delete**(*)

功能

该指令将从指定的值从该变量的容器中删除。变量的大小就减 1。

如果指定的索引或键超出了范围，就产生运行时错误。指令 **isDefined()** 可以用来检查是否一个键已被用在一个集合中。

一个集合的大小可以为零，但是一个数组变量必须始终至少有一个元素。当试图删除数组的最后一个元素时，产生一个运行时错误。

该指令不支持二维和三维的数组（使用 **resize()** 指令代替）和局部数组变量。

delete() 指令释放系统内存。无法保证内存垃圾回收的性能。鉴于此性能问题，应该避免在 VAL 3 软件应用中大量使用 **delete()**。

例如

此示例在一个集合中删除 article。

```
// 请求要删除的 article
put("Article to remove ?")
get(sArticleName)
if isDefined(sArticleColl[sArticleName]) ==true
    // 从该集合中删除 article
    delete(sArticleColl[sArticleName])
else
    putln(" 错误: article not defined")
endif
```

num **getData(string sDataName, *)**

功能

这个指令将由它的名称 **sDataName** 指明的数据值复制在指定的变量中。如果数据和变量都是一维数组, **getData()** 指令复制此数组的所有输入值, 直至到达该数组的最后输入值。这个指令返回复制在变量中的输入数。

数据必须按照如下的格式来命名: "library:name[index]", 此处 "library:" 和 "[index]" 可选择:

- "name" 是这个数据的名称
- "library" 是定义这个数据的库的识别字符名
- 当数据是一维数组时, "index" 是要访问的索引的数值

当数据复制无法执行时, 指令会返回一个错误代码:

返回值	说明
n > 0	变量已成功用 n 复制输入得以更新
-1	数据不存在
-2	库标识符不存在
-3	索引值超出范围
-4	数据类型与变量类别不匹配

例如

这个程序将一个库的点 pApproach[] 和 pTrajectory[] 的 2 个数组合并成一个单一局部数组 pPath[]。

```
// 复制路径中的接近点
i = getData("Part:pApproach", pPath)
if(i > 0)
nPoints = i
// 将轨迹点添加在路径中
i = getData("Part:pTrajectory", pPath[nPoints])
if(i >0)
nPoints=nPoints+i
endif
endif
```

2.5.5. 数组变量的特殊指令

void append(*)

功能

这条指令创建变量类型的一个新值，并将它存储在一维数组变量的末尾。新的值被该类型的默认值初始化。变量的大小就增加了 1。

该指令不支持二维和三维的数组，以及局部数组变量。一个变量的大小不能超过 9999 值。当该变量的大小超过此限制，就产生一个运行时错误。

该 **append** 指令分配系统内存。无法保证内存分配的性能。鉴于此性能问题，应该避免在 VAL 3 软件应用中大量使用 **append()**。

例如

该示例将一个新的 **article** 加在列表中。

```
// 请求一个新 article 名
put("New article ?")
get(sArticle)
println("")
append(sArticleList)
sArticleList[size(sArticleList)-1] = sArticle
```

num size(*, num nDimension)

功能

该指令返回数组中指定维数的大小。如果 **nDimension** 超过数组的维数，函数返回 0。对于一个一维数组，**size(s1dArray, 1)** 等于 **size(s1dArray)**。

例如

数组变量必须指定没有方括号：

```
//OK
nNbElements=size(s3dArray,3)
// 编译错误：不应有的索引
nNbElements=size(s3dArray[1,2,3],3)
```

void resize(*, num nDimension, num nSize)

功能

该指令创建或删除数组中的值，以便指定维数的大小与 **nSize** 值适配。值的创建或删除在数组末尾进行。新值（如果有的话）被该类型的默认值初始化。

这个指令不支持局部数组变量。一个变量的大小不能超过 9999 值。当该变量的大小超过此限制，就产生一个运行时错误。

resize() 指令分配或释放系统内存。无法保证内存处理的性能。鉴于此性能问题，应该避免在 VAL 3 软件应用中大量使用 **resize()**。

例如

必须指定变量没有索引。下一个指令修改 **s2dArray**，以便它的第二个维数为 5。

```
resize(s2dArray, 2, 5)
```

2.5.6. 集合变量的特殊指令

string first(*)

功能

该指令返回一个集合的第一个键 (按照字母表顺序)。如果集合为空，则返回一个空字符串 ""。

string next(*)

功能

该指令返回一个集合的下一个键 (按照字母顺序排序)。如果指定的键是集合的最后一个键，指令返回一个空字符串 ""。

例如

这个示例将一个集合按键字母顺序排列的所有元素打印在用户页面上：

```
sKey = first(sCollection)
while sKey != ""
    sKey = next(sCollection[sKey])
    println(sKey)
endWhile
```

string last(*)

功能

该指令返回一个集合的最后一个键 (按照键的字母顺序排序)。如果集合为空，则返回一个空字符串 ""。

string prev(*)

功能

该指令返回一个集合的先前的键 (按照键的字母顺序排序)。如果指定的键是集合的第一个键，指令返回一个空字符串 ""。

例如

这个示例将一个集合按键字母倒序排列的所有元素打印在用户页面上：

```
sKey = last(sCollection)
while sKey != ""
    sKey = prev(sCollection[sKey])
    println(sKey)
endWhile
```

2.6. 程序参数

子程序参数是从一个调用程序传送到一个子程序的带调用指令的数据。在子程序中，当子程序启动时，参数就像局部变量那样被自动初始化。

可有不同的方法将一个变量传递到一个子程序：

- 您可能想仅传递变量的一个值（一个元素），或者作为一个整体的变量的容器（数组或集合）。
- 您可能会允许子程序修改变量值（“通过引用”传递变量），或者仅传递子程序的一个拷贝，同时确保变量保持不变（“通过数值”传递变量）。

一个变量可作为参数传递：

- 通过元素值。
- 通过元素引用。
- 通过数组或集合引用。

通过值传递一个容器（数组或集合）是不允许的。

一个在程序接口定义中在数据类型后带有‘&’符号的一个引用：

`num& nData` 是一个通过元素引用传递的 `num` 参数。

`num& n1dArray[]` 是一个通过数组引用传递的 `num` 参数。

`num& n2dArray[,]` 是一个通过数组引用传递的 `num` 参数（二维数组）。

`num& n3dArray[,,]` 是一个通过数组引用传递的 `num` 参数（三维数组）。

`num& nCollection[""]` 是一个通过集合引用传递的 `num` 参数。

在本资料中，相同的概念用于指令描述：

`bool pointToJoint(tool tTool, joint jInitial, point pPosition, joint& jResult)` 是一条指令：返回一个布尔值，取得一个工具，一个关节，通过元素值传递的点数据，以及通过元素引用传递的关节。

`num fromBinary(num& nDataByte[], num nDataSize, string sDateFormat, num& nValue[])` 是一条指令：返回一个数值，取得一个作为第一参数的一维数组（通过引用传递），一个通过元素值传递的数值和字符串数据，以及一个作为最后参数的一维数值（通过引用传递）。

2.6.1. 按元素值的参数

当参数由“按元素值”定义时，系统创建一个局部变量，并用由调用程序提供的**VAL 3**指令的数值将其初始化。如果提供的指令是一个变量，参数被该变量值的一个拷贝初始化。子程序中的参数值的所有改变对调用程序中的变量值不产生影响。

例如

使 `sendMessage(string sMessage)` 成为一个具元素值传递的单一参数的程序。

`sendMessage()` 可以与一个常数数据或计算结果一起使用：

```
call sendMessage ("Waiting for signal StartCycle")
call sendMessage ("Waiting for signal"+sSignalName)
```

`sendMessage()` 可以与来自元素、数组或集合的值一起使用：

```
call sendMessage (sMessage)
call sendMessage (sMessageArray[23])
call sendMessage (s2dArray[12,3])
call sendMessage (s3dArray[5,7,9])
call sendMessage (sMessageColl [sMessageName])
```

在这些调用后，`sendMessage()` 中的指令将不会改变 `sMessage`, `sMessageArray[23]`, `s2dArray[12,3]`, `s3dArray[5,7,9]`, `sMessageColl[sMessageName]` 的值。

2.6.2. 按元素引用的参数

当一个参数是按元素引用定义的，系统将创建一个局部变量并用一个由调用程序提供的数据的链接将它初始化。通过引用传递的变量可以有一个元素，数组或集合容器，但只有在调用中被指定的值被传递到子程序。参数的容器始终是一个元素。子程序中的参数值的所有改变影响调用程序的数据的对应值。不可能通过元素引用传递一个**VAL 3**常数数据或**VAL 3**表达式的结果。

例如

使 `sendMessage(string& sMessage)` 成为一个带元素引用传递的单一参数的程序。

`sendMessage()` 不能与一个常数数据或计算结果一起使用：

```
// 编译错误：作为参数的期待变量
call sendMessage ("Waiting for signal StartCycle")
call sendMessage ("Waiting for signal"+sSignalName)
```

`sendMessage()` 可以与来自元素、数组或集合的值一起使用：

```
call sendMessage (sMessage)
call sendMessage (sMessageArray[23])
call sendMessage (s2dArray[12,3])
call sendMessage (s3dArray[5,7,9])
call sendMessage (sMessageColl [sMessageName])
```

在这些调用后，`sMessage`, `sMessageArray[23]`, `s2dArray[12,3]`, `s3dArray[5,7,9]`, `sMessageColl[sMessageName]` 值可能会被 `sendMessage()` 中的指令改变。

2.6.3. 按数组或集合引用的参数

当一个参数是按数组或集合引用定义的，系统创建一个局部变量并用一个由调用程序提供的数据的链接将它初始化。在子程序中的参数容器与所提供的变量的容器相同：一个一维、二维或三维数组，或一个集合。子程序中的参数值的所有改变直接影响调用程序的数据的对应值。

通过指明在该数组中的第一个可访问的元素，一维数组可以仅传递部分数组到子程序。对于二维和三维数组，以及集合，不可能仅将部分数组或集合传递给子程序。那么，变量必须通过无[]括号，指明一个索引或一个键来传递。不可能通过数组或集合引用传递一个 **VAL 3** 常数数据或 **VAL 3** 表达式的结果。

例如

使 `send1dMessage(string& s1dArray[])` 成为一个带数组引用（一维）传递的单一参数的程序。

使 `send2dMessage(string& s2dArray[])` 成为一个带数组引用（二维）传递的单一参数的程序。

使 `send3dMessage(string& s3dArray[])` 成为一个带数组引用（三维）传递的单一参数的程序。

使 `sendCollMessage(string& sMessageColl[""])` 成为一个带集合引用传递的单一参数的程序。

这些程序中的任何一个都不能与一个常数数据或计算结果一起使用：

```
// 编译错误：作为参数的期待数组变量
call send1dMessage ("Waiting for signal StartCycle")
call send1dMessage ("Waiting for signal"+1_sSignalName)
```

传递变量的容器必须与该参数申明容器相适配：

```
// 编译错误：1d 期待数组变量
call send1dMessage (sMessageColl)
call send1dMessage (s2dArray[12, 3])
// 编译错误：期待集合变量
call sendCollMessage (sMessage)
```

不可能传递一个数组或集合的一部分，除一维数组之外。必须指定数组和集合无索引或键。

```
// 正确参数
call send2dMessage (s2dArray)
call send3dMessage (s3dArray)
call sendCollMessage (sMessageColl)
call send1dMessage (sMessageArray)
call send1dMessage (sMessageArray[23])
// 编译错误：数组不应有的索引
call send2dMessage (s2dArray[12, 3])
call send3dMessage (s3dArray[5, 7, 9])
// 编译错误：集合不应有的索引
call sendCollMessage (sMessageColl[1_sMessageName])
```

在后面给出的例子中，只有从指定索引 23 开始的数组部分 `sMessageArray` 才可以传递给 `send1dMessage()` 程序。带小于索引 23 的 `sMessageArray` 的值不能被 `send1dMessage()` 访问。

章节 3

简单类型

3.1. BOOL 类型

3.1.1. 定义

bool 类型的值或常量可以是:

- **true**: 真值
- **false**: 假值

当一个 **bool** 类型变量被初始化时, 其默认值为 **false**。

3.1.2. 运算符

按递增优先顺序:

bool <bool& bVariable> = <bool bCondition>	将 bCondition 的值赋予变量 bVariable 并返回 bCondition 的值
bool <bool bCondition1> or <bool bCondition2>	返回 bCondition1 和 bCondition2 之间的或逻辑值。 只有 bCondition1 是 true 时, bCondition2 才被估值。
bool <bool bCondition1> and <bool bCondition2>	返回 bCondition1 和 bCondition2 之间的与逻辑值。 只有 bCondition1 是 true 时, bCondition2 才被估值。
bool <bool bCondition1> xor <bool bCondition2>	返回 bCondition1 和 bCondition2 之间的异或值
bool <bool bCondition1> != <bool bCondition2>	测试 bCondition1 和 bCondition2 值的相等性。假如数值不同, 返回 true , 否则, 返回 false 。
bool <bool bCondition1> == <bool bCondition2>	测试 bCondition1 和 bCondition2 值的相等性。假如数值相同, 返回 true , 否则, 返回 false 。
bool ! <bool bCondition>	返回 bCondition 的反值

为了避免 = 和 == 操作符的混淆, = 操作符不允许在 VAL 3 表达式中作为一个指令参数使用。

if(bCondition1=bCondition2) 可以被解释为 **bCondition1=bCondition2; if(bCondition1==true)**。但是经常要写为: **if(bCondition1==bCondition2)**, 这实在是不同的!

3.2. NUM 类型

3.2.1. 定义

num 类代表一个约有 **14** 位有效数字的数值。

因此，每次数字计算最多精确到 **14** 个有效数字。

这在测试两个数值的相等性时必须予以考虑：这通常都在一个指定的有效级内进行测试。

数值型常数的格式如下：

```
[ - ] <digits> [ .<digits> ] [ e [ - ] <digits> ]
```

'e' 是科学记数法用的标记，用来替代 '10^'：1e3 等于 1×10^3 (或 1000), 1e-2 等于 1×10^{-2} (或 0.01)。

num 型变量的默认初始值是 **0**。

例如

数值计算结果的测试必须考虑数字计算的不精确度。

`if cos(nAngle)==0, if abs(cos(nAngle))<1e-10` 最好用 `if abs(cos(n))<1e-10` 代替 `if abs(cos(nAngle))<1e-10`。

下面给出一些常数示例：

```
1
0.2
-3.141592653
6.02214179e23
1.054571628e-34
```

3.2.2. 运算符

按递增优先顺序：

num <num nVariable> = <num nValue>	给变量 nVariable 赋值 nValue 并返回 nValue 。
bool <num nValue1> != <num nValue2>	如果 nValue1 不等于 nValue2 就返回 true ，否则返回 false 。
bool <num nValue1> == <num nValue2>	如果 nValue1 等于 nValue2 就返回 true ，否则返回 false 。
bool <num nValue1> >= <num nValue2>	如果 nValue1 大于或等于 nValue2 就返回 true ，否则返回 false 。
bool <num nValue1> > <num nValue2>	如果 nValue1 大于 nValue2 就返回 true ，否则返回 false 。
bool <num nValue1> <= <num nValue2>	如果 nValue1 小于或等于 nValue2 就返回 true ，否则返回 false 。
bool <num nValue1> < <num nValue2>	如果 nValue1 小于 nValue2 就返回 true ，否则返回 false 。
num <num nValue1> - <num nValue2>	返回 nValue1 和 nValue2 的差。
num <num nValue1> + <num nValue2>	返回 nValue1 和 nValue2 的和。
num <num nValue1> % <num nValue2>	模数运算：返回被 nValue2 整除 nValue1 的余数。如果 nValue2 为 0 ，就产生一个运行时错误。余数记号与 nValue1 的相同。
num <num nValue1> / <num nValue2>	返回 nValue1 被 nValue2 除的商。如果 nValue2 为 0 ，就产生一个运行时错误。
num <num nValue1> * <num nValue2>	返回 nValue1 和 nValue2 的积。
num - <num nValue>	返回 nValue 的相反数。

为了避免=和==操作符的混淆，=操作符不允许在VAL 3表达式中作为一个指令参数使用。**nCos=cos(nAngle=30)**必须用**nAngle=30; nCos=cos(nAngle)**。

3.2.3. 指令

num sin(num nAngle)

功能

返回 **nAngle** 的正弦值。

参数

num nAngle 角度以度表示

例如

`sin(30)` 返回 0.5

num asin(num nValue)

功能

返回 **nValue** 反正弦值，以度表示。其值在 **-90** 度和 **+90** 度之间。

如果 **nValue** 的值大于 **1** 或小于 **-1**，就产生一个运行时错误。

例如

`asin(0.5)` 返回 30

num cos(num nAngle)

功能

返回 **Angle** 的余弦值。

参数

num nAngle 角度以度表示

例如

`cos(60)` 返回 0.5

num acos(num nValue)

功能

返回 **nValue** 的反余弦值，以度表示。其值在 **0** 度和 **180** 度之间。

如果 **nValue** 的值大于 **1** 或小于 **-1**，就产生一个运行时错误。

例如

`acos(0.5)` 返回 60

num tan(num nAngle)

功能

返回 **Angle** 正切值。

参数

num nAngle 角度以度表示

例如

`tan(45)` 返回 1.0

num atan(num nValue)

功能

返回 **nValue** 反正切值，以度表示。其值在 **-90** 度和 **+90** 度之间。

例如

`atan(1)` 返回 45

num abs(num nValue)

功能

返回 **nValue** 绝对值。

例如

数值计算结果的测试必须考虑数字计算的不精确度：

`if cos(nAngle)==0` 最好用 `if abs(cos(nAngle))<1e-10` 代替。

`abs(3.1415)` 返回 3.1415

`abs(-3.1415)` 返回 3.1415

num sqrt(num nValue)

功能

返回 **nValue** 平方根。

如果 **nValue** 为负值，就产生一个运行时错误。

例如

`sqrt(9)` 返回 3

num exp(num nValue)

功能

返回 **nValue** 指数函数。

如果 **nValue** 值太大，就产生一个运行时错误。

例如

`exp(1)` 返回 2.718281828459

num power(num nX, num nY)

功能

返回 **nX** 的 **nY** 次方：**nX^{nY}**

如果 **nX** 的值为负数或为空，或者如果计算结果太大，就产生一个运行时错误。

例如

此程序使用 2 种方法计算 5 的 7 次幂。

```
// 第一种方法：power 指令
nResult = power(5, 7)
// 第二种方法：power(x,y)=exp(y*ln(x)) ( 使用不精确算法 )
nResult = exp(7*ln(5))
```

num **ln**(num nValue)

功能

返回 **nValue** 自然对数。

如果 **nValue** 值为负或为 0，就产生一个运行时错误。

例如

`ln(2.718281828)` 返回 0.9999999983113

num **log**(num nValue)

功能

返回 **nValue** 常用对数。

如果 **nValue** 值为负或为 0，就产生一个运行时错误。

例如

`log(1000)` 返回 3

num **roundUp**(num nValue)

功能

返回向上取整 **nValue** 的值。

例如

`roundUp(7.8)` 返回 8

`roundUp(-7.8)` 返回 -7

num **roundDown**(num nValue)

功能

返回向下取整的 **nValue** 值。

例如

`roundDown(7.8)` 返回 7

`roundDown(-7.8)` 返回 -8

num **round**(num nValue)

功能

返回 **nValue** 向上或向下取整的最接近整数。

例如

`round(7.8)` 返回 8

`round(-7.8)` 返回 -8

`round(0.5)` 返回 1

`round(-0.5)` 返回 0

num min(num nX, num nY)

功能

返回 **nX** 和 **nY** 的最小值。

例如

`min(-1, 10)` 返回 -1

num max(num nX, num nY)

功能

返回 **nX** 和 **nY** 的最大值。

例如

`max(-1, 10)` 返回 10

num limit(num nValue, num nMin, num nMax)

功能

返回由 **nMin** 和 **nMax** 限定的 **nValue** 值。

例如

`limit(30, -90, 90)` 返回 30

`limit(100, -90, 90)` 返回 90

`limit(-100, -90, 90)` 返回 -90

num sel(bool bCondition, num nValue1, num nValue2)

功能

如果 **bCondition** 是 **true**, 返回 **nValue1**, 否则返回 **nValue2**。

例如

`sel(true, -90, 90)` 返回 -90

`sel(false, -90, 90)` 返回 90

3.3. 位字段类型

3.3.1. 定义

一个位字段是以紧凑形式存储和交换一系列位字节(布尔值或数字输入/输出)的方法。**VAL 3** 不提供一种特殊的数据类型来管理位字段，但是重新使用数值类型以一个在 $[0, 2^{32}]$ 范围中的正整数形式来存储一个 32-位的位字段。

任何 **VAL 3** 数值都可以被视作一个 32 位的位字段；处理指令的位字段自动将一个数值四舍五入至一个 32 位正整数，然后该正整数被当作 32 位位字段来处理。

3.3.2. 运算符

数字类型的标准运算符可以用于一个位字段：'='，'=='，'!='。

3.3.3. 指令

num bNot(num nBitField)

功能

这条指令返回位逻辑 "NOT"(否定)运算在一个 32 位字段上。(如果输入的第 i 位为 0，结果的第 i 位将置为 1)。因此得到在 $[0, 2^{32}]$ 范围内的一个正整数。

在二进制运算被应用之前，数字输入值首先会被四舍五入成一个在 $[0, 2^{32}]$ 范围内的正整数。

例如

这个程序使用一个掩码 nMask 来重置一个 nBitField 位字段的 i 位到 j 位。

```
// 计算一个位掩码，其 i 到 j 位已置为 1(见有关 bOr 的解释)
nMask=(power(2,j-i+1)-1)*power(2,i)
// 倒置掩码，使所有的位都为 1，除了 i 位到 j 位之外
nMask=bNot(nMask)
// 使用二进制运算 'and' 来重置 i 到 j 位
nBitField=bAnd(nBitField, nMask)
```

num bAnd(num nBitField1, num nBitField2)

功能

这个指令将二进制逻辑运算 "AND" 传递到 32 位的两个位字段上。如果两个第 i 个输入同时设置为 1 的话，第 i 个位将会设置成 1。因此得到在 $[0, 2^{32}]$ 范围内的一个正整数。

在二进制运算被应用之前，数字输入值首先会被四舍五入成一个在 $[0, 2^{32}]$ 范围内的正整数。

例如

此程序通过逐个测试每个位字段来显示一个 32 位的位字段 nBitField:

```
for i=31 to 0 step -1
// 计算第 i 位的掩码
nMask=power(2,i)
if bAnd(nBitField, nMask)==nMask
  put("1")
else
  put("0")
endif
endFor
putln("")
```

num bOr(num nBitField1, num nBitField2)

功能

这个指令将二进制逻辑运算 "OR" 传送到 32 位的两个位字段上。(如果至少一个输入的第 i 位被置为 1，则计算结果的第 i 位被置为 1)。因此得到在 [0, 2^32] 范围内的一个正整数。

在二进制运算被应用之前，数字输入值首先会被四舍五入成一个在 [0, 2^32] 范围内的正整数。

例如

此程序用两种不同的方法计算一个位字段的掩码，在这个位字段中，第 i 到第 j 的位已激活。

```
// 第一种方法：第 i 位到第 j 位上的逻辑 'or'
nBitField=0
for k=i to j
    nBitField=bOr(nBitField, power(2,k))
endFor
// 第二种方法：计算 (j-i) 位的位掩码
nBitField=(power(2,j-i+1)-1)
// 然后将位掩码移动 i 位
nBitField=nBitField*power(2,i)
```

num bXor(num nBitField1, num nBitField2)

功能

这条指令将二进制逻辑运算 "XOR"(唯一的 OR) 传送到 32 位的两个位字段上。(如果两个输入的第 i 位不同，结果的第 i 位将置为 1)。因此得到在 [0, 2^32] 范围内的一个正整数。

在二进制运算被应用之前，数字输入值首先会被四舍五入成一个在 [0, 2^32] 范围内的正整数。

例如

此程序将 nBitField 位字段的 i 位倒置为 j 位：

```
// 计算 i 位到 j 位的掩码 (见 bOr 个示例)
nMask=(power(2,j-i+1)-1)*power(2,i)
// 使用掩码倒置位字段 i 到 j
nBitField=bXor(nBitField,nMask)
```

**num toBinary(num nValue[], num nValueSize, string sDataFormat,
num& nDataByte[])**

**num fromBinary(num nDataByte[], num nDataSize,
string sDataFormat, num& nValue[])**

功能

toBinary/fromBinary 指令的目的是允许在两个设备之间通过串行或网络连接来进行数值的交换。数值首先被编码成一个字节流。然后字节被发送给同级设备。最后，此同级设备将字节解码以恢复初始数值。有几种数值的二进制编码方式。

根据在数据格式 **sDataFormat** 的规定，指令 **toBinary** 将数值编码成一个字节数组 (8 位字节字段，在区间 [0, 255] 内的正整数)。要编码的数值量 **nValue** 的数目由 **nValueSize** 参数给出。结果被保存在 **nDataByte** 数组中，并且该指令将已编码的字节数返回到此数组中。

如果要编码的数值 **nValueSize** 的数目大于 **nValue**，如果指定的格式不被支持，或者如果结果的数组 **nDataByte** 不够大来给所有的输入数据编码，就产生一个运行时错误。

根据数据格式 **sDataFormat** 的规定，指令 **fromBinary** 将一个字节数组解码为数值 **nValue**。要解码的字节的数目由参数 **nDataSize** 给出。结果保存在 **nValue** 数组内，指令将这个数值数目返回在此数组中。如果某些二进制数据损坏了 (字节出区间 [0, 255] 之外或无效浮点编码)，指令就返回与正确编码值的数的相反值 (负值)。

如果要解码的字节数 **nDataSize** 大于 **nDataByte**，如果规定格式不被支持，或如果结果数组 **nValue** 不够大来解码所有的输入数据，就产生一个运行时错误。

支持二进制编码由下表给出：

- 符号 "-" 表示一个带符号整数的编码 (字节字段的最后一个字节给数值的符号编码)。
- 数字给出用于每个数值编码的字节数。
- ".0" 后缀表示浮点值编码 (支持 IEEE 754 单精度和双精度编码)。
- 最后一个字母表示字节的顺序: "l" 用于 "小端" (低地址存放最低有效位), "b" 用于 "大端" (低地址存放最高有效位)。"大端" 编码方式是网络应用 (TCP/IP) 的标准编码。

"-1"	带符号字节
"1"	无符号字节
"-2l"	带符号字, 小端
"-2b"	带符号字, 大端
"2l"	无符号字, 小端
"2b"	无符号字, 大端
"-4l"	带符号双字, 小端
"-4b"	带符号双字, 大端
"4l"	无符号双字, 小端
"4b"	无符号双字, 大端
"4.0l"	单精度浮点值, 小端
"4.0b"	单精度浮点值, 大端
"8.0l"	双精度浮点值, 小端
"8.0b"	双精度浮点值, 大端

用于数字数据的 **VAL 3** 的原生格式为双精度编码。这个格式必须用来交换数值量而不失去精度。

例如

第一个程序将 **trsf** 数据 **trShiftOut** 编码为字节数组 **nByteOut**, 并通过串行连接 **siTcpClient** 发送。第二个程序读取来自串行连接 **siTcpServer** 的字节, 并将其变换为 **trsf trShiftIn**。

```
// ---- 发送一个 trsf 的程序 ----
// 将一个 trsf 的坐标复制在一个数字变量缓存区中
nTrsfOut[0]=trShiftOut.x
nTrsfOut[1]=trShiftOut.y
nTrsfOut[2]=trShiftOut.z
nTrsfOut[3]=trShiftOut.rx
nTrsfOut[4]=trShiftOut.ry
nTrsfOut[5]=trShiftOut.rz
// 将 6 个数值(双精度浮点, 因此 8 字节)编码为在数组 nByteOut[48] 中的 6*8=48 位字节
toBinary(nTrsfOut, 6, "8.0b", nByteOut)
// 通过 tcpClient 发送 nByte 数组(48 字节)
sioSet(siTcpClient, nByteOut)

// ---- 用于读取一个 trsf 的程序 ----
nb=0
i=0
while (nb<48)
    nb=sioGet(siTcpServer, nByteIn[i])
    if (nb>0)
        i=i+nb
    else
        // 通讯错误
        return
    endIf
endWhile
if (fromBinary(nByteIn, 48, "8.0b", nTrsfIn) != 6)
    // 损坏的数据
    return
else
    trShiftIn.x=nTrsfIn[0]
    trShiftIn.y=nTrsfIn[1]
    trShiftIn.z=nTrsfIn[2]
    trShiftIn.rx=nTrsfIn[3]
    trShiftIn.ry=nTrsfIn[4]
    trShiftIn.rz=nTrsfIn[5]
endif
```

3.4. STRING 类型

3.4.1. 定义

字符串类型变量用于储存文本。字符串类支持标准 **Unicode** 字符集。注意 **Unicode** 字符的正确显示依赖于安装在显示设备上的字符字体。

一个字符串以 **128** 字节保存，一个字符串的字符的最大数量由所用的字符决定，因为内部字符编码 (**UnicodeUTF8**) 从 **1** 字节 (用于 **ASCII** 字符) 到 **4** 字节 (**3** 用于中文字符)。

因此，一个 **ASCII** 字符串的最大长度为 **128** 个字符，中文字字符串的最大长度为 **42** 个字符。

字符串类型变量的默认初始值是 **""** (空字符串)。

3.4.2. 运算符

按递增优先顺序：

string <string& sVariable> = <string sString>	给变量 sVariable 赋值 sString 并返回 sString 。
bool <string sString1> != <string sString2>	假如 sString1 和 sString2 不相同，则返回 true ，否则返回 false 。
bool <string sString1> == <string sString2>	假如 sString1 和 sString2 相同，则返回 true ，否则返回 false 。
string <string sString1> + <string <sString2>	返回 sString1 加 sString2 的前面部分字符 (最多 128 位)。

为了避免 **=** 和 **==** 操作符的混淆，**=** 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。**nLen=len(sString="hello wild world")** 必须用 **sString="hello wild world"; nLen=len(sString)**。

3.4.3. 指令

string toString(string sFormat, num nValue)

功能

根据 **sFormat** 显示格式，指令返回代表 **nValue** 的一串字符串。

格式是 **"size.precision"**，**size** 就是结果的最小尺寸 (如必要，在字符串前加空格)，**precision** 是小数点后的有效位数 (在字符串后 **0** 由空格代替)。在默认情况下，**size precision** 等于 **0**。但数值的整数部分是决不能被缩短，即便其显示长度超出 **size**。

例如

返回

```

nPi = 3.141592654
toString(".4", nPi) 返回 "3.1416"
toString("8", nPi) 返回 "      3" ('3' 前面的 7 个空格)
toString("8.4", nPi) 返回 " 3.1416" ('3' 前面的 2 个空格)
toString("8.4", 2.70001) 返回 " 2.7    " ('2' 前面的 2 个空格, '7' 后面的 3 个空格)
toString("", nPi) 返回 "3"
toString("1.2", 1234.1234) 返回 "1234.12"

```

另见

string chr(num nCodePoint)
string toNum(string sString, num& nValue, bool& bReport)

string toNum(string sString, num& nValue, bool& bReport)

功能

该指令查找在指定 **sString** 的开头代表的数值 **nValue**, 并返回 **sString**, 它的所有字符都被删除, 直至一个数值 **value** 的下一个表达。

如果 **sString** 不是以数字值开头, **bReport** 设置为 **false**, **nValue** 不改变, 否则 **bReport** 设置为 **true**。

例如

```
toNum("10 20 30", nVal, bOk) 返回 "20 30", nVal 等于 10, bOk 等于 true
toNum("a10 20 30", nVal, bOk) 返回 "a10 20 30", nVal 不变, bOk 等于 false
toNum("10 end", nVal, bOk) 返回 "", nVal 等于 10, bOk 等于 true
```

该程序相继显示 90, 0, -7.6, 17.3

```
sBuffer = "+90 0.0 -7.6 17.3"
do
    sBuffer = toNum(sBuffer, nVal, bOk)
    putln(nVal)
until (sBuffer=="") or (bOk != true)
```

另见

string toString(string sFormat, num nValue)

string **chr(num nCodePoint)**

功能

如果是一个有效Unicode码点，指令返回由规定的Unicode码点字符组成的字符串。否则，指令返回一个空字符串。

下面列表给出了小于 **128** 的 Unicode 码点 (它与 **ASCII** 字符表相一致)。在灰格子里的字符是控制码，当显示字符串时，它们可能被一个问号替代。

VAL 3 字符串类型支持所有有效 Unicode 码点。但是，字符的显示取决于在显示设备上安装的字符字体。详细的 Unicode 字符列表可以在 <http://www.unicode.org> 上找到 (搜索 'Code Charts')。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NUL	SOH	STX	ETX	EOT	ENQ	ACQ	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
" "	!	"	#"	\$	%	&	'	()	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	()	~		DEL

例如

`chr(65)` 返回 "A"

另见

`num asc(string sText, num nPosition)`

num **asc(string sText, num nPosition)**

功能

该指令返回 **nPosition** 索引字符的 Unicode 码点。

如果 **nPosition** 是负值或大于指定的文本长度，指令返回 -1。

例如

`asc("A", 0)` 返回 65

另见

`string chr(num nCodePoint)`

string **left(string sText, num nSize)**

功能

该指令返回 **sText** 的前 **nSize** 个字符。假如 **nSize** 大于 **sText** 的长度，指令返回 **sText**。

如果 **nSize** 为负值，就产生一个运行时错误。

例如

`left("hello world", 5)` 返回 "hello"

`left("hello world", 20)` 返回 "hello world"

string **right(string sText, num nSize)**

功能

该指令返回 **sText** 的最后 **nSize** 个字符。假如指定数大于 **sText** 的长度，指令返回 **sText**。

如果 **nSize** 为负值，就产生一个运行时错误。

例如

`right("hello world", 5)` 返回 "world"

`right("hello world", 20)` 返回 "hello world"

string **mid(string sText, num nSize, num nPosition)**

功能

返回从索引字符 **nPosition** 开始的，在 **sText** 后结束的 **sText** 的 **nSize** 个字符。

如果 **nSize** 或者 **nPosition** 为负值，产生一个运行时错误。

例如

`mid("hello wild world", 4, 6)` 返回 "wild"

`mid("hello wild world", 20, 6)` 返回 "wild world"

string insert(string sText, string sInsertion, num nPosition)

功能

该指令返回 **sText**, 在此字符串中, **sInsertion** 在索引字符 **nPosition** 后被插入。假如 **nPosition** 比 **sText** 大, **sInsertion** 就被插在 **sText** 的末尾。如果它超出了 128 字节, 结果的尾数就被截去。

如果 **nPosition** 为负值, 就产生一个运行时错误。

例如

```
insert ("hello world", "wild", 6) 返回 "hello wild world"
```

string delete(string sText, num nSize, num nPosition)

功能

该指令返回 **sText**, 在此字符串中, **nSize** 已经从 **nPosition** 索引字符中被删除。假如 **nPosition** 大于 **sText** 的长度, 指令返回 **sText**。

如果 **nSize** 或者 **nPosition** 为负值, 产生一个运行时错误。

例如

```
delete ("hello wild world", 5, 6) 返回 "hello world"
```

string replace(string sText, string sReplacement, num nSize, num nPosition)

功能

该指令返回 **sText**, 在此字符串中, 来自 **nPosition** 索引字符的 **nSize** 个字符被 **sReplacement** 代替. 假如 **nPosition** 大于 **sText** 的长度, 指令返回 **sText**。

如果 **nSize** 或者 **nPosition** 为负值, 产生一个运行时错误。

例如

```
replace ("hello ? world", "wild", 1, 6) 返回 "hello wild world"
```

num find(string sText1, string sText2)

功能

该指令返回 **sText1** 中的 **sText2** 的第一次发生的第一字符的索引 (在 0 和 127 之间)。如果 **sText2** 不出现在 **sText1** 中, 指令返回 -1。

例如

```
find ("hello wild world", "wild") 返回 6
```

num len(string sText)

功能

该指令返回 **sText** 中的字符数。

例如

```
len ("hello wild world") 返回 16
```

另见

num getDisplayLen(string sText)

3.5. DIO 类型

3.5.1. 定义

dio 类型变量用于将一个 **VAL 3** 应用与系统的数字输入/输出联系起来。一个 **dio** 变量将一个链路存储到一个系统数字输入或输出，即“物理地址”。

所有使用 **dio** 类型的变量，没有与一个在系统中申明的输入 / 输出链接的指令均会产生一个运行时错误。**dio** 类型变量的默认初始值是一个不确定的链路。一个 **dio** 变量的链路可以通过另一个 **dio** 变量，通过机器人 **MCP** 初始化，或使用 **Stäubli Robotics Suite** 中的 **VAL 3 Studio** 来初始化。

3.5.2. 运算符

按递增优先顺序：

bool <dio diOutput> = <bool bCondition>	指定 bCondition 为 diOutput 的状态，并返回 bCondition 。如果 diOutput 没有与系统输出链接，产生一个运行时错误。
bool <dio diInput1> != <bool bInput2>	如果 diInput1 和 bInput2 状态不同，返回 true ，否则返回 false 。
bool <dio diInput> != <bool bCondition>	如果 diInput 状态不等于 bCondition 就返回 true ，否则返回 false 。
bool <dio diInput> == <bool bCondition>	如果 diInput 的状态等于 bCondition ，就返回 true ，否则返回 false 。
bool <dio diInput1> == <dio diInput2>	如果 diInput1 和 diInput2 状态相同就返回 true ，否则返回 false 。

为了避免 = 和 == 操作符的混淆，= 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。**if(diOutput==diInput)** 可以被解释为：**diOutput=diInput; if(diOutput==true)**。但是经常要写为：**if(diOutput==diInput)**，这实在是不同的！

注意：

鉴于与其他 '=' 运算符相一致的原因，对于 **VAL 3 s7**，在两个变量 **dio** 之间的 '=' 运算符不再存在（参考用户类型的 '=' 运算符的定义）。在一个 **dio** 和一个 **bool** 之间的 '=' 运算符可以方便地替代它：以前的 **VAL 3** 版本的 **diOut = diIn** 可以用 **diOut = (diIn==true)** 替换。

3.5.3. 指令

void dioLink(dio& diVariable, dio diSource)

功能

该指令将 **diVariable** 与 **diSource** 链接在其上的输入输出链接起来。

例如

这个应用使用一个可以用不同硬件设备设置的信号。以下给出的程序测试那个被安装用来初始化 **diSignal** 变量的设备，然后该变量被用于其余的应用中。

```
if (ioStatus(diDevice1Signal) >= 0)
// device 1 is installed: use it
    dioLink(diSignal, diDevice1Signal)
elseIf (ioStatus(diDevice2Signal) >= 0)
// device 2 is installed: use it
    dioLink(diSignal, diDevice2Signal)
else
    println(" 错误: no io device installed")
endif
```

num dioGet(dio diArray[])

功能

该指令返回 **diArray** 的数值，它以二进制编码写的整数读取，例如：

diArray[0] + 2 * diArray[1] + 4 * diArray[2] + ... + 2^k * diArray[k], 如果 **diArray[i]** 是 **true**，那么 **diArray[i] = 1**，否则 **0**。

如果一个 **diArray** 的元素没有与系统的输入 / 输出链接，会产生一个运行时错误。

例如

```
diArray[0] = false
diArray[1] = true
diArray[2] = false
diArray[3] = true
dioGet(diArray) 返回 10 = 0+2*1+4*0+8*1
```

另见

num dioSet(dio diArray[], num nValue)

num dioSet(dio diArray[], num nValue)

功能

该指令转换以二进制表达的 **nValue** 的整数部分，将它赋值给 **diArray** 的输出，并返回对应的值，例如：

diArray[0] + 2 * diArray[1] + 4 * diArray[2] + ... + 2^k * diArray[k], 如果 **diArray[i]** 是 **true**，那么 **diArray[i] = 1**，否则 **0**。

如果 **diArray** 的一个元素没有与系统输出链接，产生一个运行时错误。

例如

使用 **di4bitsArray**，大小为 4 的数组：

dioSet(di4bitsArray, 10) 返回 10

dioSet(di4bitsArray, 26) 返回 10, 由于 26 要求一个二进制编码的 5 字节： $10 = 26 - 2^4$

另见

num dioGet(dio diArray[])

num ioStatus(dio diInputOutput)

功能

如果指定的输入输出变量工作，该指令返回一个正数，如果变量错误，则返回一个负数。返回的值详细给出输入输出的状态：

0	输入 / 输出工作。
1	输入输出工作，但被运算符锁住。那么输入就有一个定值（由运算符控制），它可以不同于硬件值。那么输出就有一个由运算符控制的定值：该输出的写入没有任何作用。锁模式就是调试方法。
2	模拟输入输出（软件输入输出，对硬件无影响）。
-1	由于没有定义链路，输入输出不工作（物理地址）。
-2	由于与任何系统输入输出不匹配，输入输出不工作（物理地址）。与物理地址对应的硬件设备既无法安装，也无法初始化。
-3	由于输入输出设备出错，输入输出不工作。

例如

这个应用使用一个可以用不同硬件设备设置的信号。以下给出的程序测试那个被安装用来初始化 **diSignal** 变量的设备，然后该变量被用于其余的应用中。

```

if (ioStatus(diDevice1Signal) >= 0)
    // device 1 is installed: use it
    dioLink(diSignal, diDevice1Signal)
elseIf (ioStatus(diDevice2Signal) >= 0)
    // device 2 is installed: use it
    dioLink(diSignal, diDevice2Signal)
else
    putln(" 错误: no io device installed")
endif

```

另见

num ioStatus(dio diInputOutput, string& sDescription, string& sPhysicalPath)
num ioStatus(aio aiInputOutput)
num ioBusStatus(string& sErrorDescription[])

**num ioStatus(dio diInputOutput, string& sDescription,
 string& sPhysicalPath)**

功能

此指令完全按照以上描述的 **ioStatus** 指令执行，此外返回指定输入输出的描述文本和链路（物理地址）。

描述是一个用输入输出控制工具定义的自由文本。物理链路的格式取决于输入输出设备。通常它的形式为：：
‘deviceName\moduleName\ioAddress’。

例如

如果它不工作，该程序测试一个信号并显示错误信息。

```
if ioStatus(diSignal, sDescription, sPath)<0  
    println("Signal "+sPath+ "in error")  
    println("说明: "+sDescription)  
endif
```

另见

num ioStatus(aio aiInputOutput)
num ioStatus(dio diInputOutput)
num ioBusStatus(string& sErrorDescription[])

3.6. AIO 类型

3.6.1. 定义

aio 类型变量用于连接一个 **VAL 3** 应用和系统模拟输入和输出。一个 **aio** 变量将一个链路存储到一个系统模拟数字输入或输出，即“物理地址”。

所有使用 **aio** 类型的变量，没有与一个在系统中申明的输入 / 输出链接的指令均会产生一个运行时错误。**aio** 类型变量的默认初始值是一个不确定的链路。一个 **aio** 变量的链路可以被通过另一个 **aio** 变量，通过机器人 **MCP** 初始化，或使用 **Stäubli Robotics Suite** 中的 **VAL 3 Studio** 来初始化。

3.6.2. 指令

void aioLink(aio& aiVariable, aio aiSource)

功能

该指令将 **aiVariable** 与 **aiSource** 链接在其上的输入输出链接起来。

例如

这个应用使用一个可以用不同硬件设备设置的信号。以下给出的程序测试那个被安装用来初始化 **aiSignal** 变量的设备，然后该变量被用于其余的应用中。

```
if(ioStatus(aiDevice1Signal)>=0)
// device 1 is installed: use it
    aioLink(aiSignal, aiDevice1Signal)
elseIf (ioStatus(aiDevice2Signal)>=0)
// device 2 is installed: use it
    aioLink(aiSignal, aiDevice2Signal)
else
    putln(" 错误: no io device installed")
endif
```

num aioGet(aio ailnput)

功能

该指令返回 **ailnput** 的数值。

如果 **ailnput** 没有与系统的输入 / 输出链接的话，产生一个执行错误。

例如

aioGet(aiSensor) 返回当前传感器的值

另见

num aioSet(aio aiOutput, num nValue)

num aioSet(aio aiOutput, num nValue)

功能

该指令将 **nValue** 赋值于 **aiOutput** 并返回 **nValue**。如果设置的值超过了 **aio** 的范围，返回的数字将是 **aio** 输出的真实值。

如果 **aiOutput** 没有与系统输出链接，产生一个运行时错误。

例如

aioSet(aiCommand, -12.3) 将 -12.3 写入到输出指令并返回 -12.3 如果 **aiCommand** 是一个浮点输出值。

aioSet(aiCommand, 12.3) 将 12 写入到输出指令并返回 12 如果 **aiCommand** 是一个整数输出。

另见

num aioGet(aio ailnput)

num ioStatus(aio ailnputOutput)

功能

如果指定的输入输出变量工作，该指令返回一个正数，如果变量错误，则返回一个负数。返回的值详细给出输入输出的状态：

0: 输入 / 输出工作。

1: 输入输出工作，但被运算符锁住。那么输入就有一个定值(由运算符控制)，它可以不同于硬件值。那么输出就有一个由运算符控制的定值：该输出的写入没有任何作用。锁模式就是调试方法。

2: 模拟输入输出 (软件输入输出，对硬件无影响)。

-1: 由于没有定义链路，输入输出不工作 (物理地址)。

-2: 由于与任何系统输入输出不匹配，输入输出不工作 (物理地址)。与物理地址对应的硬件设备既无法安装，也无法初始化。

-3: 由于输入输出设备出错，输入输出不工作。

例如

这个应用使用一个可以用不同硬件设备设置的信号。以下给出的程序测试那个被安装用来初始化aiSignal变量的设备，然后该变量被用于其余的应用中。

```
if(ioStatus(aiDevice1Signal)>=0)
// device 1 is installed: use it
    aioLink(aiSignal, aiDevice1Signal)
elseIf (ioStatus(aiDevice2Signal)>=0)
// device 2 is installed: use it
    aioLink(aiSignal, aiDevice2Signal)
else
    println(" 错误: no io device installed")
endif
```

另见

num ioStatus(dio dilnputOutput)

**num ioStatus(aio dilnputOutput, string& sDescription,
string& sPhysicalPath)**

功能

该指令完全按照以上描述的 ioStatus 指令执行，但除此之外还返回指定输入输出的描述文本和链路（物理地址）。

描述是一个用输入输出控制工具定义的自由文本。物理链路的格式取决于输入输出设备。通常它的形式为：：
‘deviceName\moduleName\ioAddress’。

例如

如果它不工作，该程序测试一个信号并显示错误信息。

```
if ioStatus(aiSignal, sDescription, sPath)<0
    println("Signal "+sPath+ "in error")
    println(" 说明: "+sDescription)
endif
```

另见

num ioStatus(aio ailnputOutput)

num ioStatus(dio dilnputOutput)

3.7. SIO 类型

3.7.1. 定义

sio 类型用来将一个 **VAL 3** 变量与一个串行口或一个 Ethernet socket connection 链接起来。一个 **sio** 输入输出的特性如下：

- 在系统中被定义的专用于通信类型的参数
- 字符串的最后一个字符，用于 **string** 类型的使用
- 一个通信的超时等待

系统输入输出串口一直是激活的。在 **VAL 3** 程序进行初始读或写访问时，Ethernet socket connection 接口打开。当 **VAL 3** 软件应用停止时，Ethernet socket connection 接口就自动关闭。

所有使用 **sio** 类型的变量，没有与一个在系统中申明的输入 / 输出链接的指令均会产生一个运行时错误。**sio** 类型变量的默认初始值是一个不确定的链路。一个 **sio** 变量的链路可以被通过另一个 **sio** 变量，通过机器人 **MCP** 初始化，或使用 **Stäubli Robotics Suite** 中的 **VAL 3 Studio** 来初始化。

3.7.2. 运算符

当读或写输入 / 输出时，如通信时间超时，产生一个执行错误。

string <sio siOutput> = <string sText>	在 siOutput 上连续写入 UnicodeUTF8 编码的 sText 个字符，由字符串字符结尾后，返回 sText 。
num <sio siOutput> = <num nData item>	在 siOutput 上写入最接近 nData item 的整数，模数 256 ，返回已传送的数值。
num <num nData> = <sio siInput>	读出 siInput 上的一个字节，将 nData 赋给字节值。
string <string sText> = <sio siInput>	在 siInput 上读取一个 Unicode UFT8 字符的字符串，并将这个字符串赋值给 sText 。不被 string 类型支持的字符被忽略。当读取字符串的最后字符或 sText 达到一个 string(128 bytes) 的最大大小时，字符串完成。字符串的最后字符就不复制在 sText 中。

为了避免 = 和 == 操作符的混淆，= 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。

nLen=len(sString=silInput) 必须用 **sString=silInput; nLen=len(sString)**。

3.7.3. 指令

void **sioLink**(**sio& siVariable, sio siSource**)

功能

指令将 **siVariable** 与 **siSource** 链接在其上的串行系统输入输出链接起来。

另见

void dioLink(dio& diVariable, dio diSource)
void aioLink(aio& aiVariable, aio aiSource)

num **clearBuffer**(**sio siVariable**)

功能

指令清除 **siVariable** 阅读缓冲储存器并返回由被删除的字符数

对于一个 Ethernet socket connection 接口, **clearBuffer** 关闭此接口。**clearBuffer** 返回 -1, 如果套接字已经关闭。如果 **siVariable** 没有与系统串行接口或以太网套接字链接的话, 会产生一个运行时错误。

num **sioGet**(**sio silnput, num& nData[]**)

功能

该指令读取一个单字符或一个来自 **silnput** 的数组, 并返回读取的字符数目。

当 **nData** 数组已满或输入读取缓冲已空, 读取过程停止。

对于一个 Ethernet socket connection 接口, 如果没有连接的话, **sioGet** 首先尝试打开连接。当达到输入通信时间时, **sioGet** 就返回 -1。如果连接打开, 但输入读取缓冲储存器内没有数据, **sioGet** 就等待直到接收到数据或者直达到超时时间。

如果 **silnput** 没有与系统的串行接口或 Ethernet 套接字链接, 或者 **nData** 不是一个 **VAL 3** 变量, 产生一个运行时错误。

num **sioSet**(**sio siOutput, num& nData[]**)

功能

该指令将一个字符或一个字符数组写到 **siOutput**, 并返回写入的字符数。

在传送前, 数值被转换成 0 到 255 之间的整数, 取最接近模数 256 的整数。

对于一个 Ethernet socket connection 接口, 如果没有连接的话, **sioSet** 首先尝试打开连接。当输出通信等待时间到达时, **sioSet** 返回 -1。如果发现通信错误, 那么写入的字符数可以小于 **nData** 的大小。

如果 **siOutput** 没有与系统的串行接口或 Ethernet 套接字链接, 产生一个运行时错误。

num **sioCtrl**(**sio siChannel**, **string nParameter**, ***value**)

功能

这条指令修改一个指定的串行输入和输出 **siChannel** 的通讯参数。

对于串行线，硬件可能不支持某些参数或某些数值：请参考控制器手册。

指令返回：

0	参数已成功修改
-1	参数没有被定义
-2	参数值没有希望的类型
-3	不支持这参数值
-4	串行通道没有准备好来应用参数改变(首先停止串行通道)
-5	参数没有被定义用于此通道类型

所支持的参数在下表中列出：

参数名	参数类型	说明
"port"	num	(用于 TCP 客户端或服务器) TCP 端口
"target"	string	(用于 TCP 客户端) 要连接的 TCP 服务器的 IP 地址, 如 "192.168.0.254"
"clients"	num	(用于 TCP 服务器) 同时连接到服务器上的最大客户端数目
"endOfString"	num	(用于串行通讯, TCP 客户端和服务器) 字符串字符末端的 ASCII 编码, 与 '=' 运算符一起使用 (在区间 [0, 255] 中)
"timeout"	num	(用于串行通讯, TCP 客户端和服务器) 通讯通道的最大响应时间。0 表示无连接超时限制。
"baudRate"	num	(用于串行线路) 通讯速度
"parity"	string	(用于串行线路) 奇偶性控制: "none", "even" 或 "odd"
"bits"	num	(用于串行线路) 用字节表示的位数 (5, 6, 7 或 8)
"stopBits"	num	(用于串行线路) 用字节表示的结束位的数目 (1 或 2)
"mode"	string	(用于串行线路) 通讯模式: "rs232" 或 "rs422"
"flowControl"	string	(用于串行线路) 流量控制: "none" 或 "hardware"
"nagle"	bool	(对于 TCP 客户端或服务器) 启动 (默认) 或中断 nagle 优化。中断 nagle 优化可以改善响应时间, 但增加网络负载。

例如

此程序设置串行线路的主要参数。

```
sioCtrl(siPortSerial1, "baudRate", 115200)
sioCtrl(siPortSerial1, "bits", 8)
sioCtrl(siPortSerial1, "parity", "none")
sioCtrl(siPortSerial1, "stopBits", 1)
sioCtrl(siPortSerial1, "timeout", 0)
sioCtrl(siPortSerial1, "endOfString", 13)
```

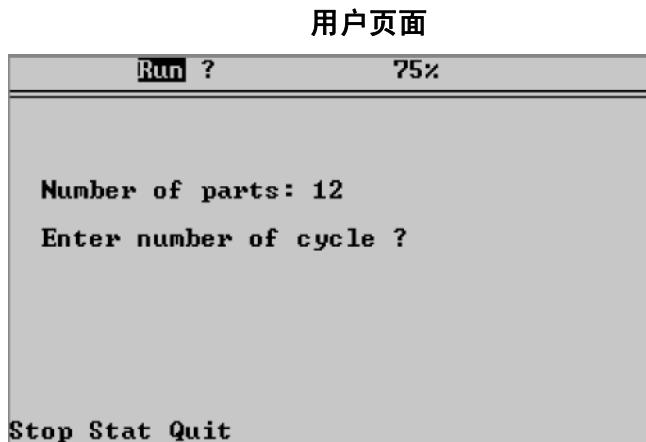

章节 4

用户界面

4.1. 用户页面

在 **VAL 3** 语言中，用户界面的指令用于：

- 在手动示教盒 (**MCP**) 的一页上显示用于软件应用的信息
- 获取 **MCP** 键盘的按键数据



用户页面有 14 行 40 列。最后一行常用来创建带关联键的菜单。补充行可以用来显示一个标题。

4.2. 屏幕类型

用户页面上下文（显示的信息和按键）可以与一个 **screen** 类型的变量相关联。那么，它就更容易在一个程序中建立和维护多个屏幕，如生产、维护和调试屏幕。

通过按 'User-Down' 或 'User-Up' 键，可以从一个屏幕轻松切换到另一个。按 'User-Shift' 切换到第一个屏幕，按 'User-Shift'-Up' 到最后一个屏幕。

screen 类型在内存中取得一个有效位置，因此不能用于局部变量，除非软件应用的运行时内存的大小已大大地增加。

4.2.1. 用户屏幕的选择

userPage() 指令可以指定一个可选的 **screen** 变量作为参数。于是在 **MCP** 上显示的当前屏幕切换到指定屏幕。如果 **screen** 参数没有被指定，当前屏幕切换到默认用户屏幕，此屏幕总是被定义的。

4.2.2. 用户屏幕上的写入

写在屏幕 (**title()**, **gotoxy()**, **cls()**, **put()**, **putln()**) 上的指令可以用一个可选变量 **screen** 作为第一个参数来指定。写操作然后在指定的屏幕上执行，而不影响其他屏幕。修改的屏幕不能是显示的屏幕。在这种情况下，修改屏幕隐藏，直到用 **userPage()** 指令改变当前的屏幕。如果 **screen** 参数没有被指定，修改屏幕就是默认用户屏幕，此屏幕总是被定义的。

4.2.3. 从用户屏幕读入

获得按键 (**get()**, **getKey()**, **isKeyPressed()**) 的指令可以用一个可选 **screen** 变量作为第一个参数来指定。那么输入操作就在该指定屏幕完成：只有显示在 **MCP** 上的屏幕才能读出一个键击。其他屏幕将不受影响。因此可能有多个不同屏幕正同时等待不同的键击：只有当前屏幕（正在显示的）才被告知有效的键击。如果 **screen** 参数没有被指定，有效屏幕就是默认用户屏幕，此屏幕总是被定义的。

4.3. 指令

**void userPage(), void userPage(screen scPage),
void userPage(bool bFixed)**

功能

指令在 **MCP** 屏幕上显示指定的用户页面，如果有的话，或者默认用户页面。

如果参数 **bFixed** 是 **true**，除了键盘上的快捷键 " 用户切换 (Shift User)" 用于进行账号更改的页面外，对操作者来说只有用户页面是可用的。当用户页面被显示时，如果当前用户权限允许的话，可以使用 "Stop" 按钮来停止此软件应用。

如果参数是 **false**，控制器 **MCP** 的其他页面重新可被访问。

**void gotoxy(num nX, num nY),
void gotoxy(screen scPage, num nX, num nY)**

功能

该指令将光标放置指定用户页面的 (**nX, nY**) 坐标上，如果有的话，或者默认用户页面。左上角的坐标为 **(0,0)**，右下角的为 **(39, 13)**。

nX 列数取模数值 **40**。 **nY** 行数取模数值 **14**。

另见

void cls(), void cls(screen scPage)

void cls(), void cls(screen scPage)

功能

该指令清除指定的，或默认的用户页面，并将光标置为 **(0,0)**。

另见

void gotoxy(num nX, num nY), void gotoxy(screen scPage, num nX, num nY)

**void setTextMode(num nMode),
void setTextMode(screen scPage, num nMode)**

功能

该指令修改指定屏幕的显示模式，如果有的话，或者默认屏幕。新的显示模式不影响当前显示，但被用于所有新文本，直至定义一个新文本模式。支持模式定义如下：

- | | |
|---|-----------------|
| 0 | 标准文本模式 (黑字白底) |
| 1 | 反转视频方式 (黑地白字) |
| 2 | 频闪标准文本模式 |
| 3 | 频闪反转视频模式 |

另见

void put(string sText), void put(screen scPage, string sText) void put(num nValue), void put(screen scPage, num nValue), void putln(string sText), void putln(screen scPage, string sText), void putln(num nValue), void putln(screen scPage, num nValue),

num **getDisplayLen(string sText)**

功能

该指令返回在 **MCP** 显示屏上的 **sText** 长度 (显示 **sText** 所需的列数)。

对于 **ASCII** 字符串，显示的长度就是字符串中的字符数，那么 **getDisplayLen()** 和 **len()** 指令相同。

有一些字符 (中文) 的显示要使用两个相连的列，那么 **getDisplayLen()** 比 **sText** 的长度大，并且可以用来控制 **sText** 在屏幕上的排列。

另见

num len(string sText)

```
void put(string sText), void put(screen scPage, string sText)
void put(num nValue), void put(screen scPage, num nValue),
void putIn(string sText), void putIn(screen scPage, string sText),
void putIn(num nValue), void putIn(screen scPage, num nValue),
```

功能

该指令在指定用户页面上在光标位置显示指定的 **sText** 或 **nValue**(到 3 位小数位), 如果有的话, 或者在默认用户页面上。然后光标被定位在所显示信息的最后一个字符后的字符上 (**put** 指令), 或者被定位在下一行的第一个字符上 (**putIn** 指令)。

在每行的最后, 显示在下一行继续。

在页面的最后, 用户页面的显示向上移一行。

另见

```
void popUpMsg(string sText)
bool logMsg(string sText)
void title(string sText), void title(screen scPage, string sText)
```

```
void title(string sText), void title(screen scPage, string sText)
```

功能

该指令改变指定用户页面的标题, 如果有的话, 或者在默认用户页面上的标题。

title() 指令不改变当前光标的位置。

```
num get(string& sText), num get(screen scPage, string& sText),
num get(num& nValue), num get(screen scPage, num& nValue),
num get(), num get(screen scPage)
```

功能

该指令获取一个字符串, 一个数字或一个控制面板键。

参数 **sText** 或 **nValue** 在当前光标位置显示, 可以由用户更改。按菜单键或 **Return** 或 **Esc** 键来完成输入。

指令返回结束输入所用键的代码。

按 **Return** 或菜单键, 更新 **sText** 或 **nValue** 变量。按 **Esc** 不改变变量。

如果没有任何参数通过, **get()** 指令等待操作者按任意键, 然后返回它的编码。被按的键不显示。

在任何情况下, 光标的当前位置都不受 **get()** 指令的影响。

不按 Shift				按 Shift			
3 283	Caps	Space		3 283	Caps	Space	
	-	32			-	32	
2 282	Shift	Esc	Help	Ret.	2 282	Shift	Esc
	-	255	-	270		-	-
	Menu	Tab	Up	Bksp		Menu	UnTab
	-	259	261	263		-	PgUp
1 281	User	Left	Down	Right	1 281	User	Home
	-	264	266	268		-	PgDn
							End
							-
							269

菜单 (带或不带 Shift)

F1 271	F2 272	F3 273	F4 274	F5 275	F6 276	F7 277	F8 278

对于标准键，返回的编码是相应字符的 **ASCII 码**:

不按 Shift										
q	w	e	r	t	y	u	i	o	p	
113	119	101	114	116	121	117	105	111	112	
a	s	d	f	g	h	j	k	l	<	
97	115	100	102	103	104	106	107	108	60	
z	x	c	v	b	n	m	.	,	=	
122	120	99	118	98	110	109	46	44	61	

按 Shift										
7	8	9	+	*	;	()	[]	
55	56	57	43	42	59	40	41	91	93	
4	5	6	-	/	?	:	!	{	}	
52	53	54	45	47	63	58	33	123	125	
1	2	3	0	"	%	-	.	,	>	
49	50	51	48	34	37	95	46	44	62	

双击 Shift										
Q	W	E	R	T	Y	U	I	O	P	
81	87	69	82	84	89	85	73	79	80	
A	S	D	F	G	H	J	K	L	}	
65	83	68	70	71	72	74	75	76	125	
Z	X	C	V	B	N	M	\$	\	=	
90	88	67	86	66	78	77	36	92	61	

例如

该程序读出用 Return 键确认的数值:

```
do
    nKey = get (nValue)
until (nKey == 270)
```

另见

num getKey(), **num getKey(screen scPage)**

num getKey(), num getKey(screen scPage)

功能

该指令获得控制面板键盘的一个键击。它返回自从最后调用 `getKey()` 后所按的最后按键的编码，如果之后没有按任何键的话，返回 `-1`。只有在指定用户页面时，才能探测到一个键击，如果说有的话，或者显示默认用户页面。

与 `get()` 指令不同，`getKey()` 立即返回。

所按的键不显示，并且当前光标位置保持不变。

例如

该程序刷新系统时钟的显示，直至按下一个按键：

```
// 首次重置最后所按按键的代码
getKey()
while (getKey () == -1)
    gotoxy (0, 0)
    put (toString ("", clock () * 10))
    delay (0)
endWhile
```

另见

`num get(string& sText), num get(screen scPage, string& sText), num get(num& nValue), num get(screen scPage, num& nValue), num get(), num get(screen scPage), void userPage(), void userPage(screen scPage), void userPage(bool bFixed)`
`bool isKeyPressed(num nCode), bool isKeyPressed(screen scPage, num nCode)`

**bool isKeyPressed(num nCode),
bool isKeyPressed(screen scPage, num nCode)**

功能

该指令返回以代码指定的键的状态（参见 `get()`），如果按了键返回 `true`，否则返回 `false`。只有在指定用户页面时，才能检测到一个键击，如果说有的话，或者显示默认用户页面，除了键 (1), (2) 和 (3)，它们是始终被检测的。

另见

`num get(string& sText), num get(screen scPage, string& sText), num get(num& nValue), num get(screen scPage, num& nValue), num get(), num get(screen scPage), void userPage(), void userPage(screen scPage), void userPage(bool bFixed)`

void popUpMsg(string sText)

功能

该指令在当前 **MCP** 窗口之上，在一个窗口 "popup" 中显示 `sText`。此窗口保持显示直至点击 **Ok** 菜单或按 **Esc** 键确认。

另见

`void userPage(), void userPage(screen scPage), void userPage(bool bFixed)`
`void put(string sText), void put(screen scPage, string sText) void put(num nValue), void put(screen scPage, num nValue), void putIn(string sText), void putIn(screen scPage, string sText), void putIn(num nValue), void putIn(screen scPage, num nValue),`

bool logMsg(string sText)

功能

该指令在系统历史记录中写入 **sText** (错误记录)。信息用当前日期和时间被保存。“USR”被加在字符串的起始，以便将其标记为用户信息。如果在同一时间内记录许多信息的话，某些长信息可能会丢失。那么，该指令返回 **false**，以便信息稍后被记录，如果此信息很重要。

例如

该程序确保信息被记录：

```
while logMsg (sMessage) ==false
delay(0)
endWhile
```

另见

void popUpMsg(string sText)

string getProfile()

功能

该指令返回当前用户权限名。

另见

num setProfile(string sUserLogin, string sUserPassword)

num setProfile(string sUserLogin, string sUserPassword)

功能

该指令改变当前用户权限 (立即生效)。

函数返回：

- 0: 指定的用户权限现被激活
- 1: 指定的用户权限未定义
- 2: 指定用户密码不正确
- 3: 不允许用该指令将 ‘staubli’ 设置为用户权限名
- 4: 当前用户权限名为 ‘staubli’，不能通过这个指令进行改变

另见

string getProfile()

string getLanguage()

功能

这条指令返回机器人控制器的当前语言。

例如

```
switch(getLanguage())
  case "francais"
    sMessage="Attention!"
  break
  case "english"
    sMessage="Warning!"
  break
  case "deutsch"
    sMessage="Achtung!"
  break
  case "italiano"
    sMessage="Avviso!"
  break
  case "espanol"
    sMessage=";Advertencia!"
  break
  default
    sMessage="Warning!"
  break
endSwitch
```

另见

bool setLanguage(string sLanguage)

bool **setLanguage(string sLanguage)**

功能

该指令修改机器人控制器的当前语言：指定语言名 **sLanguage** 必须与控制器中的翻译文件的名称相符合。请参考控制器手册来在机器人控制器中删除或安装新的语言。

例如

该程序将控制器语言切换到中文：

```
if (setLanguage ("chinese") ==false)
  putln ("The Chinese language is not available on the robot controller")
endif
```

另见

string getLanguage()

string **getDate(string sFormat)**

功能

该指令返回机器人控制器的当前日期和 / 或时间。**sFormat** 参数指明要返回日期的格式。在这个字符串中，一些关键字的每次出现都被对应的日期或时间值所代替。被支持的格式关键字在下表中列出：

关键字	说明
%y	年份用 2 个数字 (00-99) 表示，没有世纪
%Y	年份用 4 个数字表示，如 2007
%m	月份 (00-12)
%d	日期 (00-31)
%H	24 小时制 (00-23)
%I	12 小时制 (01-12)
%p	A.M./P.M. 指示的 12 小时制
%M	分 (00-59)
%S	秒 (00-59)

例如

这个程序以 "January 01, 2007 13:45:23" 格式显示日期和时间

```
switch (getDate ("%m"))
  case "01"
    sMonth="January"
  break
  case "02"
    sMonth="February"
  break
  case "03"
    sMonth="March"
  break
  case "04"
    sMonth="April"
  break
  case "05"
    sMonth="May"
  break
  case "06"
    sMonth="June"
  break
  case "07"
```

```
sMonth="July"
break
case "08"
    sMonth="August"
break
case "09"
    sMonth="September"
break
case "10"
    sMonth="October"
break
case "11"
    sMonth="November"
break
case "12"
    sMonth="December"
break
default
    sMonth="???"
break
endSwitch
// 日期和时间的显示形式: "January 01, 2007 13:45:23"
putln(getDate(sMonth+" %d, %Y %H:%M:%S"))
```

章节 5

任务

5.1. 定义

一个任务就是一个正在运行的程序。一个软件应用可以并通常有多个运行的任务。

一个软件应用通常包括机械臂运动任务，自动化任务，用户接口任务，安全信号监控任务，通信任务，等等。

一个任务由下述元素来定义：

- 一个名称：一个在库或软件应用中的唯一的任务标识
- 一个优先级，或一段时间：任务序列参数
- 一个程序：任务的进入（或退出）点
- 一个状态：运行或停止
- 要执行的下一条指令（和它的上下文）

5.2. 运行时错误后的重启

当一条指令引起一个运行时错误时，任务就停止执行。**taskStatus()** 指令用来诊断运行时错误。而 **taskResume()** 指令可以重新启动任务。如果运行时错误可以被修正，任务可以在其停止的指令行处继续执行。否则，必须从指令行的前面或后面一行重新开始。

启动和中止软件应用

当一个软件应用启动时，它的 **start()** 程序就在软件应用名加'~'，优先级为 **10** 的任务中执行。

当一个软件应用结束时，它的 **stop()** 程序就在'~'加软件应用名和优先级为 **10** 的一个任务中执行。

如果 **VAL 3** 软件应用的停止是通过用户界面 (**MCP**) 进行的，启动任务（如果还存在的话）就立即被摧毁。接下来执行 **stop()** 程序，然后所有尚未执行的软件应用的任务按照其被创建时的反向顺序被删除，最后库从内存中被卸载。

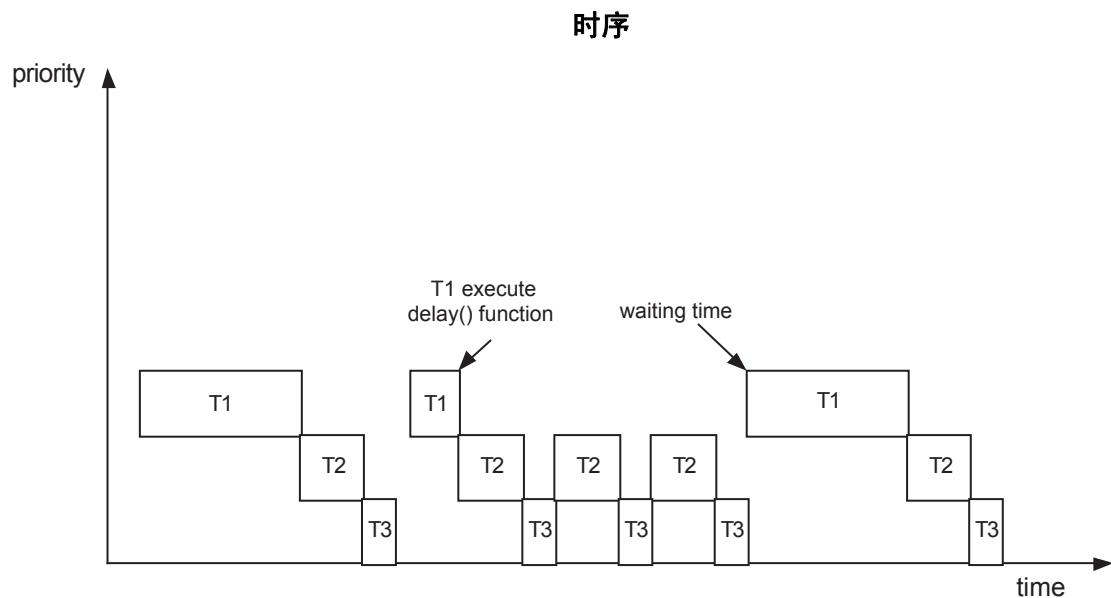
5.3. 可见性

一个任务只在创建它的程序或库中可见。指令 **taskSuspend()**, **taskResume()**, **taskKill()** 和 **taskStatus()** 对于由另外一个库所创建的任务的作用好象这个任务没有被创建一样。因此两个不同的库可以创建相同名称的任务。

5.4. 时序

当一个软件应用的多个任务正在运行时，他们似乎会同时且独立地进行。如果观察整个软件应用足够长的时间（大约一秒种）确实是如此的，但如果在一个短时间内观察它的特殊行为的话，就不是这样的。

事实上，由于系统中只有一个处理器，每次只能执行一个任务。同时执行是通过任务快速顺序模拟进行的，在系统进入下一个任务之前依次执行多个指令。



VAL 3 任务的时序遵循以下规则：

1. 根据任务建立的先后顺序来排序
2. 在每个顺序中，系统会根据任务的优先级尝试执行许多 **VAL 3** 指令。
3. 当一条指令行无法被终止时（运行时错误，等待信号，任务停止，等等），系统执行下一个 **VAL 3** 任务。
4. 当所有的 **VAL 3** 任务完成时，系统在一个新的周期开始前为低优先级的系统任务（如网络通讯，用户屏幕刷新，文件访问等）保留可用的时间。

在两个顺序周期之间的最大等待时间等于最后一个顺序周期的持续时间，但是，此等待时间在很多情况下是空的，因为系统不需要它。

可以使一个任务被立刻顺序执行的 **VAL 3** 指令如下：

- **watch()** (条件的超时等待)
- **delay()** (超时)
- **wait()** (条件等待时间)
- **waitEndMove()** (手臂停止等待时间)
- **open()** 和 **close()** (手臂停止等待时间，接着超时)
- **get()** (按键等待时间)
- **taskResume()** (等待任务准备好重新开始)
- **taskKill()** (等待这个任务真正结束)
- **disablePower()** (等待电源被真正切断)
- 这些指令可以访问硬盘内容 (**libLoad**, **libSave**, **libDelete**, **libList**, **setProfile**)
- sio 读 / 写指令 (操作符 =, **sioGet()**, **sioSet()**)
- **setMutex()** (等待布尔量 mutex 变为 false)

5.5. 同步任务

以上描述的顺序是普通任务的顺序，称为异步任务，它们由系统排定顺序，以最快速度执行。有时需要定期安排任务，如获得数据或外围设备控制，这些任务叫做同步任务。

它们在周期中的运行通过打断两个 **VAL 3** 行之间的异步任务来执行。当同步任务完成时，异步任务恢复。

VAL 3 同步任务的顺序遵守下列规则：

1. 每个同步任务精确地根据任务创建时的指定，每周期时间排入一次（例如，每 4 ms 一次）。
2. 在每个顺序中，系统执行多达 3000 行的 **VAL 3** 指令。当一个指令行不能完成时（执行错误，等待信号，任务停止等），立刻转到下一个任务。
在实际应用中，经常通过强制系统转到下一个任务的顺序的 "delay(0)" 指令来结束一个同步任务。
3. 相同周期的同步任务的时序按照它们创建的顺序来确定。

5.6. 超时运行

如果一个 **VAL 3** 同步任务的执行超过了指定的时间，当前周期正常结束，但是下一个周期被取消。这个超时运行错误通过设置布尔变量来通知 **VAL 3** 软件应用，该布尔变量在任务创建为 "true" 时专为此而指定的。因此，在每个周期的开始，这个布尔变量指明前一个顺序字段是否被完整执行。

5.7. 输入 / 输出刷新

输入在同步和异步任务执行之前被刷新。以同样的方法，输出在同步和异步任务执行之后被刷新。

警告：

不可能指定一个任务使用哪些输入和输出。因此，每一次刷新都在所有的输入 / 输出上执行。

VAL 3 调度程序不控制在 Modbus, BIO board, MIO board, CIO board 或 AS-i bus 上的输入 / 输出的刷新。它们可以在一个 **VAL 3** 任务的顺序字段期间的任何时间被刷新。

5.8. 同步

有时需要在多个任务执行之前来使它们同步。

如果预先知道执行每项任务所需要的时间，它们就可以通过简单地等待由最慢的任务产生的信号来实现同步。然而，如果不知道那个任务是最慢的，就需要使用一个更复杂的同步化机制，如下所示 **VAL 3** 程序就是一例。

例如

// N 个任务的同步程序

此后该程序 `synchro(num& n, bool& bSynch, num nN)` 必须在每个要同步的任务中被调用。

`n` 变量必须初始化为 0, `bSynch`, `false`, 和要同步的任务数 `nN`。

```
begin
  n = n + 1
  // 任务同步等待指令
  // 确保所有的任务都在这里等待以继续运行
  wait((n==nN) or (bSynch==true))
  bSynch = true
  n = n - 1
  // 任务释放等待指令
  // 确保所有的任务都已经继续运行，以清除同步上下文
  wait((n==0) or (bSynch == false))
  bSynch = false
end
```

5.9. 资源分配

当多个任务使用相同的系统或机器人工作单元资源（全局数据，屏幕，键盘，机器人等）时。确保它们之间无冲突是很重要的。

互斥机制 (**'mutex'**) 能保护资源，为此目的，采用每次只允许访问一个任务的方法来实现。**VAL 3** 中的互斥程序示例如下。

例如

该程序显示 (num c) 使用相同的字符填充屏幕，确保没有其他的任务在同一时间使用相同的程序写到屏幕。**bScreen** 必须初始化为 **false**。

```
begin
    // 确保一次只有一个任务访问屏幕
    setMutex (bScreen)
    c=c%10
    // 用字符填充屏幕
    for y=0 to 13
        gotoxy (x,y)
        put (c)
    endFor
    endFor
    // 等待屏幕刷新
    delay (0.2)
    // 现在让其他任务访问屏幕
    bScreen=false
end
```

5.10. 指令

void taskSuspend(string sName)

功能

该指令暂停 **sName** 任务的执行。

如果任务已经是 **STOPPED**, 指令无效。

如果 **sName** 不对应于任何一个 **VAL 3** 任务, 或者对应于由另一个库建立的一个任务 **VAL 3**, 就产生一个运行时错误。

另见

void taskResume(string sName, num nSkip)

void taskKill(string sName)

void taskResume(string sName, num nSkip)

功能

在位于当前行的前面或后面的 **nSkip** 指令行的这一行上该指令继续执行任务 **sName**。

如果 **nSkip** 为负数, 程序在当前行前继续执行。如果任务没有处在 **STOPPED** 状态, 指令无效。

如果 **sName** 与一个任务 **VAL 3** 不对应, 但对应于由另一个库建立的一个任务 **VAL 3**, 或者在指定的 **nSkip** 的地方没有指令行, 产生一个执行错误。

另见

void taskSuspend(string sName)

void taskKill(string sName)

void taskKill(string sName)

功能

该指令暂停 **sName** 任务的执行，然后将它删除。当指令已经被执行，任务 **sName** 就不再出现在系统中。

如果没有 **sName** 任务，或者 **sName** 任务由其他的库所创建，则指令无效。

另见

void taskSuspend(string sName)

void taskCreate string sName, num nPriority, program(...)

void setMutex(bool& bMutex)

功能

等待 **bMutex** 变量为 **false**，然后将它设置为 **true**。

该指令要求使用一个布尔变量作为一个互斥机制，以保护共享资源（见第 5.9 章）。

string help(num nErrorCode)

功能

此指令返回用 **nErrorCode** 参数指明的运行时错误编码的说明。说明用当前控制器语言给出。

例如

此程序检查任务 "robot" 是否出错，如果有错则为操作者显示出错代码。

```

nErrorCode=taskStatus ("robot")
if (nErrorCode > 1)
  gotoxy(0,12)
  put(help(nErrorCode))
endif

```

num taskStatus(string sName)

功能

该指令返回 **sName** 任务的当前状态，或者运行时错误代码的任务，如果它是在错误条件下：

代码	说明
-1	当前库或程序没有创建 sName 任务
0	任务 sName 暂停，无运行时错误 (taskSuspend() 指令或调试模式)
1	由当前库或软件应用创建的任务 sName 正在运行之中
10	数字计算无效（被零除）。
11	无效数字计算（如 ln(-1) ）
20	访问一个索引超过此数组大小的数组。
21	访问一个负索引的数组。
29	任务名无效。见 taskCreate() 指令。
30	指定名没有相应的任何 VAL 3 任务。
31	同名的任务已经存在。见 taskCreate 指令。
32	仅支持同步任务的 2 个不同周期。更改调度周期。
40	无足够可用的内存空间。
41	无足够的运行该任务的内存空间。查看运行内存的大小。
60	超过最大指令运行时间。
61	VAL 3 内部解释错误
70	指令参数无效。见相应指令。
80	使用了没有装载到内存中的一个程序库的数据或程序。
81	运动不兼容：一个 point/joint/config 的使用与手臂运动不兼容。
82	一个变量的参考坐标系或工具属于一个库，但是这个库不能从变量的作用域进行访问（库未在此变量的项目中声明，或者参考变量是专用的）。
90	此任务无法在指定处继续执行。见 taskResume() 指令。
100	运动描述符中指明的速度无效（负值或太大）。
101	运动描述符中指明的加速度无效（负值或太大）。
102	运动描述符中指明的减速值无效（负值或太大）。
103	运动描述符中指明的平移速度无效（负数或太大）。
104	运动描述符指明的旋转速度无效（负数或太大）。
105	运动描述符中指明的 reach 参数无效（负值）。
106	运动描述符中指明的 leave 参数无效（负值）。
122	试图写入系统输入。
123	使用了一个未与系统输入 / 输出相连的的 dio 、 aio 或 sio 输入 / 输出。
124	试图访问一个受保护的系统输入 / 输出
125	dio 、 aio 或 sio 上的读取或写入错误（现场总线错误）
150	无法运行该运动指令：前一个运动要求未完成（无法到达该点，奇点，姿态问题，等）
153	不支持的运动命令
154	无效的动作指令：目标无法达到，或检查运动描述符。
160	flange 工具的坐标无效
161	world 工具坐标无效
162	使用了一个无参考坐标系的 point 。见定义。
163	使用无参考坐标系的一个坐标系。见定义。
164	使用无参考工具的一个工具。见定义。
165	坐标系或参考工具无效（全局变量和一个局部变量链接）
250	此指令无运行时许可，或者试用许可已经过期。

另见

[void taskResume\(string sName, num nSkip\)](#)

[void taskKill\(string sName\)](#)

void taskCreate string sName, num nPriority, program(...)

功能

该指令创建并启动 **sName** 任务。

sName 必须包含从 "a..ZA..Z0..9_" 中选择的 **1 到 15** 个字符。由相同的库所创建的任何任务都不能同名。

sName 的执行用一个使用指定参数的调用 **program** 来开始。不能将一个局部变量用作一个随引用传递的参数，以便确保变量在任务完成之前没有被删除。

任务由 **program** 的最后指令行给予默认终止，或者更早一点终止，如果任务已明显被删除。

nPriority 必须在 **1 到 100** 之间。当任务被定序时，系统执行大量对应于 **nPriority** 的指令行，或者如果遇上一个封锁指令就会执行较少的指令行（参见定序章节）。

如果系统没有足够的内存来建立这个任务，如果 **sName** 无效或已经在同一个库中被使用，或者 **nPriority** 无效，产生一个运行时错误。

例如

```
// 开始一个读出信息的新任务
taskCreate "t1", 10, read(sMessage)
// 等待任务 t1 结束
wait(taskStatus("t1") == -1)
// 使用这条信息
putln(sMessage)
```

另见

```
void taskSuspend(string sName)
void taskKill(string sName)
num taskStatus(string sName)
```

void taskCreateSync string sName, num nPeriod, bool& bOverrun, program(...)

功能

该指令创建并启动一个同步任务。

这个任务通过调用指定程序和指定参数来开始执行。不能将一个局部变量用作一个随参考传递的参数，以便确保变量在任务完成之前没有被删除。

当系统没有足够的内存来创建一个任务，或者一个或更多的参数无效时，将产生一个运行时错误。

同步任务的详细描述见相关章节（见第 5.5 章）。

参数

string sName	要创建的任务名。它必须包含 1 到 15 个从 "a..zA..Z0..9" 中选择的字符。在同一 个软件应用或者库中，不能有同名的其他任务。
num nPeriod	要创建任务的周期 (s)。这个指定的值向下取整到 4 ms 的倍数 (0.004 秒)。支持任 何正周期，但是系统同时只能支持同步任务的两个不同周期。
bool& bOverrun	信号超时错误的布尔变量。只支持全局变量，以确保在任务之前这个变量没有被删 除。
program	当任务开始时调用的 VAL 3 程序名，在括号中是它的参数。

例如

```
// 创建一个每 20 ms 执行的管理任务
taskCreateSync "supervisor", 0.02, bSupervisor, supervisor()
```

void wait(bool bCondition)

功能

该指令暂停当前任务，直至 **bCondition** 为 **true**。

在等待期间，任务保持 **RUNNING** 状态。如果第一次赋值 **bCondition** 为 **true**，有关任务立即被执行（下个任务没有定序）。

另见

void delay(num nSeconds)
bool watch(bool bCondition, num nSeconds)

void delay(num nSeconds)

功能

该指令暂停 **nSeconds** 的当前任务。

在等待期间，任务保持 **RUNNING** 状态。如果 **nSeconds** 是负的或空的，系统立即给下一个 **VAL 3** 任务定序。

例如

这个程序循环来得到一个键，注意不要使用不必要的 CPU 资源：

```
// 首次重置最后所按按键的代码
getKey()
while(getKey() == -1)
    gotoxy(0,0)
    put(toString("", clock() * 10))
// 让另一个任务立即执行它的运算
delay(0)
endWhile
```

另见

num clock()
bool watch(bool bCondition, num nSeconds)

num **clock()**

功能

该指令返回内部系统时钟的当前值，用秒表示。

系统内部时钟的精确度为毫秒。当控制器启动时，时钟初始值为 **0**，因此它与日历时间没有关联。

例如

要计算两个指令之间的执行延迟，在第一个指令前存储时钟值：

```
nStart=clock()
```

在最后的指令之后，计算执行延迟：

```
nDelay = clock() -nStart
```

另见

void delay(num nSeconds)

bool watch(bool bCondition, num nSeconds)

bool watch(bool bCondition, num nSeconds)

功能

该指令暂停当前任务，直到 **bCondition** 为 **true** 或已经过了 **nSeconds** 秒。

如果当 **bCondition** 为 **true** 时，等待时间结束，返回 **true**，否则当等待时间结束时返回 **false**，因为已超时。

在等待期间，任务保持 **RUNNING** 状态。如果第一次赋值时 **bCondition** 为 **true**，立即执行同一任务，否则系统给另一个 **VAL 3** 任务定序（即使 **nSeconds** 直至和包括 **0**）。

例如

该程序等待一个信号并在 **20 s** 之后显示一个错误信息。

```
if (watch (diSignal==true, 20)) == false
  popUpMsg (" 错误: waiting for Signal")
  wait(diSignal==true)
endif
```

另见

void delay(num nSeconds)

void wait(bool bCondition)

num clock()

章节 6

库

6.1. 定义

一个 **VAL 3** 库是一个 **VAL 3** 软件应用，其变量或程序可以被其他软件应用或其他 **VAL 3** 库所重新使用。作为一个 **VAL 3** 软件应用，一个 **VAL 3** 库包含以下部分：

- 一组**程序**: 要执行的 **VAL 3** 指令
- 一组**全局变量**: 库数据
- 一组**库**: 库所使用的外部指令和变量

一个库正在运行时，它还可以包括：

- 一组**任务**: 正在运行的这个库专用的程序

所有的软件应用可以用作库，所有的库也可以用作一个软件应用，只要 **start()** 和 **stop()** 程序在其中被定义。

6.2. 接口

一个库的全局程序和变量可以是共用的或专有的。只有全局程序和变量是公用的才可以在库外被访问。库程序只能使用私有程序和全局变量。

所有公用全局程序和变量来自于它接口的一个库：许多不同的库可能有相同的接口，只要它们的公用程序和变量使用相同的名字。

由一个库程序创建的任务通常是专用的，也就是说它们只能被该库所访问。

6.3. 接口标识符

要使用一个库，一个软件应用需要首先声明一个赋值给它的标识符，然后在一个程序中提出请求，请求在这个标识符下将此库装载到内存中。

标识符被赋值给库的接口，并不是给库本身。那么表示同个接口的任何库都可以在这个标识符下被下载。这个机制可以使用，例如，为一个软件应用的每一个可能的部分定义一个库，然后只装载被每一个周期处理的当前部分。

6.4. 内容

一个库没有必须的内容：它可以只包含程序，或只包含变量，或两者。

例如通过在库程序名或数据前写下标识符名后跟 ‘:’ 来访问库的内容：

```
// 在 "article" 标识符下下载 "article_7" 库
article:libLoad("article_7")
// 显示作为标题的 article_7 库的 'sName' 变量的内容
title(article:sName)
// 调用当前部分的 init() 程序
call article:init()
```

访问一个还没有被加载到内存的库的内容，会产生一个运行时错误。

6.5. 加密

VAL 3 支持加密库，基于广泛使用的 ZIP 压缩和加密工具。

一个加密库是一个库的目录内容的标准加密文件（注意：不支持高级 128-bit 和 256-bit AES 加密）。Zip 文件名必须有 '.zip' 扩展名，少于 15 个字符（包括扩展名）。为提高加密效果，ZIP 密码必须在 10 字符以上，而且不能在字典中找到。

保密密码，公用密码

VAL 3 编译器必须能够访问保密 ZIP 密码来加载一个加密库，为此，给 **Stäubli Robotics Suite** 提供一个 PC 工具来将保密 ZIP 密码编码为一个公共 **VAL 3** 密码。公共 **VAL 3** 密码使得可以在一个 **VAL 3** 程序中使用加密库。但是库的内容仍然是加密的因为 **VAL 3** 密码不能计算 ZIP 密码。

项目加密

不可能直接加密控制器上的启动软件应用。一个完整的软件应用可以通过以下方法加密：

- 将其 `start()` 程序声明为公用的。
- 创建另一个软件应用，将加密的软件应用简单地加载为一个库，并调用它的 `start ()` 程序。

6.6. 加载和卸载

当打开一个 **VAL 3** 软件应用时，所有申明的库都被分析以建立对应的接口。这个步骤不将库加载在内存中。

注意:

不支持库之间的循环引用。如果库 A 使用库 B，库 B 不能使用库 A。

当载入一个库时，它的全局数据被初始化，它的程序被检查来探测任何语法错误。当几个不同库的标识符将同一个库加载在磁盘上时，它们共享内存中的同一个库。然后库仅加载一次，并被所有的标识符所重新使用。在下述示例中，lib1 和 lib2 使用内存中的相同数据。

```
lib1:libLoad("appData")
lib1:sText = "lib1"
lib2:libLoad("appData")
// lib2:sText 的改变此处也适用于 lib1:sText
lib2:sText = "lib2"
```

一个库的卸载并非是必须的，当软件应用结束或一个新库被加载来替代另一个库时，卸载自动进行。

当一个 **VAL 3** 软件应用通过 **MCP** 用户界面被停止时，**stop()** 程序最先运行，接下来所有软件应用的任务和它的库（假如它们还存在的话）都被摧毁。

访问路径

libLoad(), **libSave()** 和 **libDelete()** 指令使用一个库的路径，以字符串的形式给出。访问路径包括根符（可选），路径（可选）和库名，格式如下：

root://Path/Name

根符规定文件介质："**Floppy**" 对应软盘，"**USB0**" 对应一个在 **USB** 接口上的设备（记忆棒、软盘），"**Disk**" 对应控制器的闪存，或者在控制器上定义的用于网络访问的 **Ftp** 连接的名称。

默认情况下，根符为 "**Disk**"，路径为空。

例如

```
// 将 "article_1" 库加载在磁盘上 相当于 "Disk://article_1"
article:libLoad("article_1")
// 将库保存在 USB 上
article:libSave("USB0://articles/article_1")
// 将默认 article 加载在当前软件应用中
article:libLoad("./defaultArticle")
```

错误代码

VAL 3 库处理功能决不会产生运行时错误，但是它们发回一个用来检查指令结果和检修可能产生的问题的错误编码。

代码	说明
0	无错误
10	库的标识符没有被 libLoad() 初始化。
11	库已加载，但是公用接口不匹配。如果 VAL 3 程序试图访问丢失的项目，将产生一个运行时错误 80。见 libExist 指令或 isDefined() 指令。
12	无法加载该库：该库含有无效的数据或程序，或者，对应一个加密的库，指定的密码无效。
13	无法卸载该库：此库已被其他任务使用。
14	无法卸载该库：这个库包含一个运行的 VAL 3 任务。在这个库卸载之前，必须完成这个库的由 VAL 3 程序创建的所有任务。
20	文件存取错误：路径的根符无效。
21	文件存取错误：路径无效。
22	文件存取错误：文件名无效。
23	需要加密库。库未加密或错误加密。
>=30	读 / 写文件错误。
31	无法保存该库：指定的路径已经包含了一个库。要替换磁盘上的一个库，首先用 libdelete() 删除它。
32	驱动程序报告 "设备未找到"
33	驱动程序报告 "设备错误"
34	驱动程序报告 "设备超时"
35	驱动程序报告 "设备写保护"
36	驱动程序报告 "磁盘不存在"
37	驱动程序报告 "磁盘未格式化"
38	驱动程序报告 "磁盘已满"
39	驱动程序报告 "文件未找到"
40	驱动程序报告 "只读文件"
41	驱动程序报告 "连接被拒绝"
42	驱动程序报告 "Ftp 服务器未响应"
43	驱动程序报告 "Ftp 内核错误"
44	驱动程序报告 "Ftp 参数错误"
45	驱动程序报告 "Ftp 访问错误"
46	驱动程序报告 "Ftp 磁盘已满"
47	驱动程序报告 "无效 Ftp 用户账号"
48	驱动程序报告 "Ftp 连接未定义"

6.7. 指令

num identifier:libLoad(string sPath)

num identifier:libLoad(string sPath, string sPassword)

功能

通过加载库程序和变量到内存，跟随指定的 **sPath**，这个指令初始化该库的标识符。指定的 (可选) **sPassword** 参数用作加密库的加密密钥。指定的 **sPassword** 必须是从加密库的专用 ZIP 密码计算得出的公用 **VAL 3** 密码 (见第 6.5 章，第 98 页)。

在成功加载后，指令返回 **0**，如果仍然有这个库创建的任务在运行，如果库访问路径无效，如果库包含语法错误或如果指定的库和标识符声明的接口不相符，返回一个库加载错误代码。

另见

num identifier:libSave(), **num libSave()**

num identifier:libSave(), **num libSave()**

功能

该指令保存赋值给库标识符的变量和程序。如果 **libSave()** 在没有标识符的情况下被调用，含有 **libSave()** 指令的软件应用被保存。如果参数被指定，通过指定 **sPath** 内容被保存。否则，内容通过在加载时指定的路径被保存。

如果内容已经被保存，指令返回 **0**；如果标识符没有被初始化，如果路径无效，如果产生一个写入错误，或者如果指明的路径已经包含一个库，则返回一个错误代码。

注意:

有些设备 (例如控制器的闪存) 仅支持有限数量的写访问。如果 **libSave()** 在一个程序中经常使用 (每分钟一次或更多)，它必须在支持它的设备上被使用。

另见

num libDelete(string sPath)

num libDelete(string sPath)

功能

该指令删除位于指定 **sPath** 中的库。

如果指定库不存在或者已经被删除，指令返回 **0**；如果标识符没有被初始化，如果路径无效，或如果产生一个写错误，则返回一个错误代码。

另见

num identifier:libSave(), **num libSave()**
string identifier:libPath(), **string libPath()**

string identifier:libPath(), string libPath()**功能**

该指令返回与标识符相关的这个库的访问路径，或者如果没有指定任何标识符，则返回调用程序的访问路径。

另见

bool libList(string sPath, string& sContents[])

bool libList(string sPath, string& sContents[])

功能

该指令列出在 **sContents** 数组中指定的 **sPath** 路径的内容。如果 **sContents** 数组包括完全列表，返回 **true**，如果数组太小无法保括完全列表，则返回 **false**。

sContents 数组中的所有元素首先初始化为 ""(空字符串)。在 **libList()** 执行之后，列表的末尾因此通过在 **sContents** 数组中搜索第一个空字符串而找到。

如果 **sContents** 是一个全局变量，数组的大小按照要求自动扩大，以便能够保存完整的结果。

另见

string identifier:libPath(), string libPath()

bool identifier:libExist(string sSymbolName)

功能

libExist 指令用来测试一个标识符(全局数据或程序)是否在一个库中被定义。如果符号存在或如果可以访问(公用的)，则返回 "true"，否则，则返回 "false"。

一个程序的符号名称必须添加 "()": "mySymbol" 表示一个数据名，而 "mySymbol()" 表示一个程序名。

如果一个输入 / 输出在一个控制器上定义的话，**libExist** 指令很有用；它还根据接口是新版本还是老版本，来管理一个库接口的更新，适配它的应用。

例如

该示例测试一个库的接口。

```
// 载入 part 库
nLoadCode = part:libLoad(sPartPath)
// 在该库的版本 1 中 part:sVersion 没有被定义
// 测试如果该库定义了它
if (nLoadCode==0) or (nLoadCode==11)
  if (part:libExist("sVersion")==false)
    // 初始版
    sLibVersion = "v1.0"
  else
    sLibVersion = part:sVersion
  endif
endif
```

这个程序在 "protocol" 库 (如果存在) 中调用程序 "init":

```
if(protocol:libExist("init()")==true)
  call protocol:init()
endif
```

另见

bool isDefined(*)

章节 7

用户类型

7.1. 定义

用户类型是在一个 **VAL 3** 应用中定义的结构类型，它可以作为一个标准类型用在应用中。一个用户类型将简单、结构、甚至其他的用户类型组合成一个新数据类型。用户类型增加程序的抽象层次，使它们更容易理解，开发和维护。不过，它们要求更高的初始设计来识别最适合该应用限制的适当类型。

一个用户类型是字段的集合，每个字段的组成如下：

- 一个名称：字符串
- 一个数据类型（简单、结构、或用户类型）
- 一个数据容器（元素、数组或集合）
- 一组默认数值

VAL 3 标准类型的字段总是使用一个元素容器（具有一个单值）。用户类型字段可以使用数组或集合容器，因此包含多个数值。该字段的默认值定义在该字段容器中的默认元素数量，以及这些元素的每个的默认值。在一个使用用户类型定义的变量中，您不仅可以随时改变其字段的数值，也可以改变字段容器的元素数目。

7.2. 创建

一个用户类型的字段具有与 **VAL 3** 软件应用的全局数据相同的特性。这就是为什么只须选择一个 **VAL 3** 软件应用并将其用一个名称联系起来就可以创建一个新的用户类型。

- 在所选择的应用中，该组公用全局数据定义用户类型的字段集和它们的默认值。
- 该名称定义在软件应用中要使用的新用户类型名。

用作类型定义的专用数据和该应用的程序在用户类型中被忽略。

一旦一个新用户类型在软件应用中被定义，就可以创建此类型的数据。然后生成的软件应用也可以被用作类型定义。

7.3. 使用

用户类型变量的字段可用一个在 ‘.’ 后加字段名来进行访问：`userVariable.field1.field2` 指的数据 `userVariable` 的 `'field1'` 字段的 `'field2'` 字段的值。对于诸如一个变量，支持使用 `insert()`, `delete()`, `append()` 或 `resize()` 指令来创建或删除一个字段容器的元素。当创建一个新元素，每个字段被赋予一个默认值，该值由元素和在软件应用中用作类型定义的它们的数值组成。

注意：
点类型、工具或坐标的字段默认情况下不链接。

`'='` 运算符不总是在同一个用户类型的两个变量之间被定义的。当 `'='` 运算符被执行后，余下的变量就是正确变量的拷贝：该字段具有在它们容器中的相同的元素数目，以及相同数值的元素。

章节 8

机器人控制

此章节列出了允许访问机器人各部件状态的指令。

8.1. 指令

void disablePower()

功能

该指令切断机器人手臂电源，并等待直到电源真正被切断。

如果机器人手臂正在运动，在切断电源前手臂在其运行轨迹上立即停止。

另见

void enablePower()
bool isPowered()

void enablePower()

功能

在远程模式下，该指令给机器人手臂上电。

此指令对当地模式、手动模式或测试模式，或者电源正在被切断时不会起任何作用。它在日志中生成信息，从而避免重复的来启用电源的无延期尝试。

例如

```
// 接通电源，并等待机器人手臂上电
enablePower()
if(watch(isPowered(), 5) == false)
  putln("Arm power supply cannot be switched on")
endif
```

另见

void disablePower()
bool isPowered()

bool isPowered()

功能

该功能返回机器人手臂的电源状态：

true: 手臂处于电源接通状态

false: 手臂电源关闭，或者正在上电中

bool isCalibrated()

功能

该指令返回机器人的校准状态:

true: 所有机器人轴被校准

false: 至少机器人的一根轴未被校准

num workingMode(), num workingMode(num& nStatus)

功能

该指令返回机器人的当前工作模式:

模式	状态	工作模式	状态
0	0	无效或转换中	-
1	0	手动	程控运动
	1		连接运动
	2		Joint jogging
	3		笛卡尔 (Frame jogging)
	4		Tool jogging
	5		到点 (Point jogging)
	6		Hold
2	0	测试	程控运动 (< 250 mm/s)
	1		连接运动 (< 250 mm/s)
	2		快速程控运动 (> 250 mm/s)
	3		Hold
3	0	本地	Move(程控运动)
	1		Move (连接运动)
	2		Hold
4	0	远程	Move(程控运动)
	1		Move (连接运动)
	2		Hold

num esStatus()

功能

该指令返回 E-Stop 回路的状态:

代码	状态
0	无 E-Stop(E-Stop 回路关闭)。
1	不再有 E-Stop, 等待确认。在手动模式下, MCP 必须位于它的支架上, 使得能够接通电源。
2	E-Stop 打开, 或等待改正硬件错误。

另见

num workingMode(), **num workingMode(num& nStatus)**
bool safetyFault(string& sSignalName)

bool safetyFault(string& sSignalName)

功能

如果在安全回路上的一个硬件错误被发现, 并被确认, 此指令返回 **true**。在这种情况下, **sSignalName** 被错误信号名所更新。

另见

num esStatus()

num ioBusStatus(string& sErrorDescription[])

功能

该指令检查现场总线设备的状况, 并返回出错的设备号, 如果没有设备出错, 返回零。对于每个出错的设备, 一个文本描述就添加在 **sErrorDescription** 字符串数组中。错误描述的格式为:

‘statusValue:deviceName\moduleName’。

如果 **sErrorDescription** 是一个全局变量, 数组的大小按照要求自动扩大, 以便能够保存完整的结果。

状态数值是一个取决于现场总线和协议的错误数。

另见

num ioStatus(dio diInputOutput, string& sDescription, string& sPhysicalPath)
num ioStatus(aio diInputOutput, string& sDescription, string& sPhysicalPath)

num **getMonitorSpeed()**

功能

此指令返回机器人的当前监视器速度 (在区间 [0, 100] 内)。

例如

此程序 (被调用在一个指定的任务中) 检查第一个机器人的循环是否以低速度执行:

```
while true
  if (nCycle < 2)
    if (getMonitorSpeed() > 10)
      stopMove()
      gotoxy(0, 0)
      println ("For the first cycle the monitor speed must remain at 10%")
      wait (getMonitorSpeed() <= 10)
    endif
    restartMove()
  endif
  delay(0)
endWhile
```

另见

num setMonitorSpeed(num nSpeed)

num **setMonitorSpeed(num nSpeed)**

功能

此指令修改机器人的当前监视器速度。**setMonitorSpeed()** 总是能降低监视器的速度。为了提高速度，只有当机器人在远程工作模式，操作员无法访问监视器的速度 (当当前用户权限不允许使用速度按钮，或 MCP 已断开) 的情况下，**setMonitorSpeed()** 才有效。

如果监视器速度已修改，指令返回 0，在相反的情况下，返回一个负错误代码:

代码	说明
-1	机器人不是远程工作模式
-2	监视器速度在操作者的控制之下: 修改当前用户权限，以取消操作者访问监视器速度的权限
-3	不支持指定速度: 速度必须在区间 [0, 100] 内

另见

num getMonitorSpeed()

string getVersion(string sComponent)

功能

该指令返回机器人控制器硬件和软件组件的不同版本。下表列出所支持的组件，以及每个组件的返回的值的格式：

项目	说明
"VAL 3"	控制器 VAL 3 版本，例如 "s7.0 - Jun 18 2010 - 16:01:17"
"ArmType"	与控制器相连的手臂的类型，例如 "tx90-S1" 或 "rs60-S1-D20-L200"
"Tuning"	手臂的调试版本号，例如 "R3"
"Mounting"	手臂安装方式，例如 "置地安装"，"壁挂安装" 或 "置顶安装"
"ControllerSN"	控制器序列号，例如 "F07_12R3A1_C_01"
"ArmSN"	手臂系列号，例如 "F07_12R3A1_A_01"
"Starc"	Starc 固件 (CS8C) 的版本号，例如 "1.16.3 - Sep 27 2007 - 16:01:17"
许可证名称	控制器的软件的使用许可证状态：""(没有安装或者演示版期限已过)，"demo" 或 "enabled" 已安装的控制器的许可证名称（例如 "alter", "compliance", "remoteMcp", "oemLicence", "plc", "testMode", "mcpMode"...) 在机器人的示教盒控制面板中列出

例如

```
if getVersion ("compliance") != "enabled"
  println ("The compliance license is missing on the controller")
endif
```

另见

string getLicence(string sOemLicenceName, string sOemPassword)

章节 9

机器人手臂位置

9.1. 引言

此章描述了在一个 **VAL 3** 软件应用中用于给手臂位置编程的 **VAL 3** 数据类型。

在 **VAL 3** 中定义了两种位置类型: 关节位置 (**joint** 类型) 给出每个旋转轴的角度位置和每个线性轴的线性位置, 笛卡尔坐标点 (**point** 类型) 给出相对于一个参考坐标系的机器人手臂端头的工具中心点的笛卡尔位置。

tool 类型描述一个工具和它的用于给手臂定位和控制手臂速度的几何位置, 它还描述了如何激活一个工具 (数字输出, 延迟)。

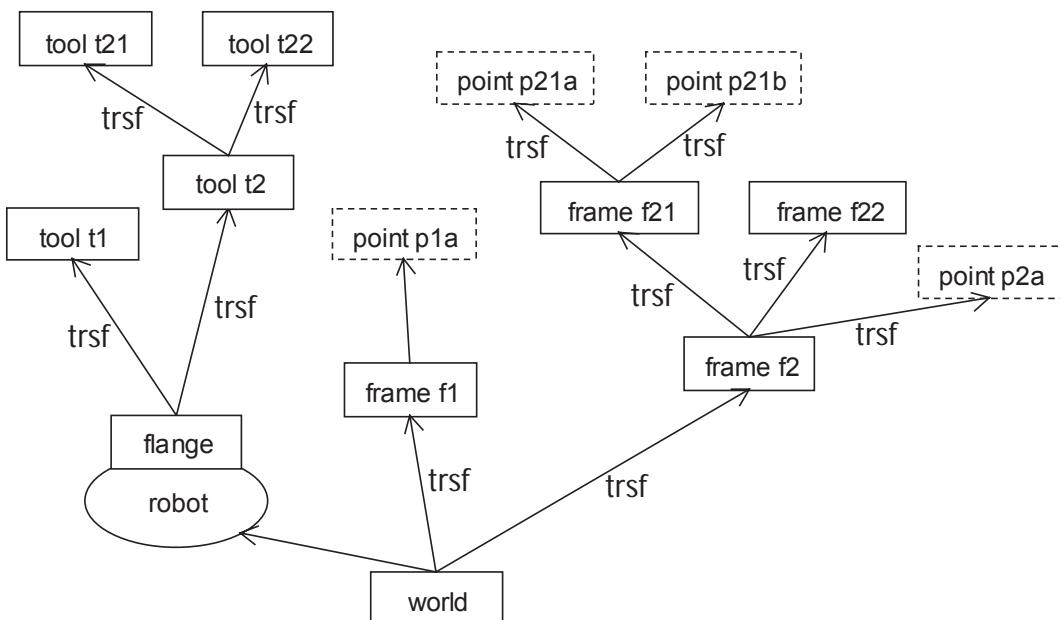
frame 类型描述一个几何参考坐标系。坐标系的使用使对几何点的处理更简单和更直观。

trsf 类型描述了一个几何变换。它被 **tool**, **point** 和 **frame** 类型所使用。

最后, **config** 类型描述机器人手臂设置的更高级的概念。

这些不同类型之间的联系概述如下:

组织图: **frame / point / tool / trsf**



9.2. JOINT 类型

9.2.1. 定义

一个关节位置 (**joint** 类型) 定义了每个旋转轴的角度位置和每个线性轴的线性位置。

joint 类型属于结构类型, 结构类型的字段按顺序为:

num j1	Joint position of axis 1
num j2	Joint position of axis 2
num j3	Joint position of axis 3
num j...	Joint position of axis ... (每轴一个字段)

对于旋转轴这些字段用度表示, 对于线性轴用毫米或英寸表示。每轴的原始点由所使用的机器人手臂的类型来确定。

一个 **joint** 类型变量的每个字段的默认初始值为 **0**。

9.2.2. 运算符

按递增优先顺序：

<code>joint <joint& jPosition1> = <joint jPosition2></code>	逐字段将 jPosition2 赋值给 jPosition1 变量，并返回 jPosition2 。
<code>bool <joint jPosition1> != <joint jPosition2></code>	如果一个 jPosition1 的字段不等于在机器人精度范围内的对应字段 jPosition2 ，返回 true ，否则返回 false 。
<code>bool <joint jPosition1> == <joint jPosition2></code>	如果每个 jPosition1 字段等于在机器人精度范围内的对应字段 jPosition2 ，就返回 true ，否则返回 false 。
<code>bool <joint jPosition1> > <joint jPosition2></code>	如果每个 jPosition1 字段大于对应的 jPosition2 字段，就返回 true ，否则返回 false 。
<code>bool <joint jPosition1> < <joint jPosition2></code>	如果每个 jPosition1 字段小于对应的 jPosition2 字段，就返回 true ，否则返回 false 。 注意： jPosition1 > jPosition2 不同于 !(jPosition1 < jPosition2)
<code>joint <joint jPosition1> - <joint jPosition2></code>	逐字段地返回 jPosition1 与 jPosition2 的差值。
<code>joint <joint jPosition1> + <joint jPosition2></code>	逐字段地返回 jPosition1 与 jPosition2 的和。

为了避免 = 和 == 操作符的混淆，= 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。

9.2.3. 指令

joint abs(joint jPosition)

功能

该指令逐字段地返回一个关节 **jPosition** 的绝对值。

详细

一个关节的绝对值，和 ">" 或 "<" 关节操作符，用来方便地计算一个关节位置和参考位置之间的距离。

例如

```
jReference = {90, 45, 45, 0, 30, 0}
jMaxDistance = {5, 5, 5, 5, 5, 5}
j = herej()
// 检查所有的轴距离参考点小于 5 度
if (!(abs(j - jReference) < jMaxDistance))
    popUpMsg("Move closer to the marks")
endif
```

另见

Operator < (joint)
Operator > (joint)

joint herej()

功能

该指令返回当前手臂关节位置。

当手臂上电时，返回值是控制器发送给放大器的位置，而不是从轴的编码器读出的位置。

当手臂下电时，返回值从轴的编码器读出；由于编码器测量的噪点，当手臂停止时位置可能会有一点变化。

控制器关节位置每 4 ms 刷新一次。

例如

```
// 等待机器人手臂接近参考位置，超时 60 s
bStart = watch(abs(herej() - jReference) < jMaxDistance, 60)
if bStart==false
    popUpMsg ("Move closer to the start position")
endif
```

另见

point here(tool tTool, frame fReference)
bool getLatch(joint& jPosition) (CS8C only)
bool isInRange(joint jPosition)

bool isInRange(joint jPosition)

功能

该指令测试一个关节位置是否在手臂关节软件的限位范围内。

如果手臂超出了软件关节限位（在一个维护操作之后），不能通过 **VAL 3** 软件应用运动手臂，只有手动运行（向限位方向的运动被限制）。

例如

```
// 检查当前的位置是否在关节限位范围内
if isInRange(herej())==false
    putln ("Please place the arm within its workspace")
endif
```

另见

joint herej()

void **setLatch(dio diInput)** (CS8C only)

功能

该指令在下一个输入信号的上升沿使机器人的位置闭锁有效。

详细

机器人的位置闭锁是一种硬件功能，仅被 CS8C 控制器的快速输入所支持 (fIn0, fIn1)。

只有在输入信号在上升沿前保持低电平至少 0.2 ms，和在上升沿后保持高电平至少 0.2 ms，在输入信号的上升沿上的检测才有效。

注意:

只有在 **setLatch** 指令执行之后的一定时间 (0 到 0.2 ms 之间) 后，位置的闭锁才有效。您或许需要在 **setLatch** 之后添加一个 **delay(0)** 指令来确保在下一个 **VAL 3** 指令被执行之前，闭锁是有效的。

如果指定的数字输入不支持机器人位置闭锁，产生一个运行时错误 70(无效的参数值)。

另见

bool getLatch(joint& jPosition) (CS8C only)

bool **getLatch(joint& jPosition)** (CS8C only)

功能

该指令读取最后的闭锁机器人位置。

如果有一个有效闭锁位置要读取，函数返回 **true**。如果是一个闭锁等待，或者闭锁从来没有被激活，函数返回 **false**，并且位置未更新。

getLatch 返回一个相同的闭锁位置，直到一个新的闭锁由 **setLatch** 指令激活为止。

在 CS8C 控制器中，手臂位置每 0.2 ms 刷新一次；闭锁位置对应于快速输入的上升沿后的 0 和 0.2 ms 之间的手臂位置。

例如

```
setLatch(diLatch)
// Wait for setLatch to be effective before using getLatch
delay(0)
// 等待闭锁位置 5 秒。
bLatch = watch(getLatch(jPosition)==true, 5)
if bLatch==true
    println("Successful position latch")
else
    println("No latch signal was detected")
endif
```

另见

void setLatch(dio diInput) (CS8C only)
joint herej()

9.3. TRSF 类型

9.3.1. 定义

一个变换 (**trs**f 类) 定义一个位置和 / 或方向改变。这是一个平移和旋转的数学合成。

一个变换本身不代表一个空间的位置，但是可以由一个点生成另一个点，或由一个坐标系联系到另一个坐标系。

ttrsf 类型是一种结构类型，其字段次序如下：

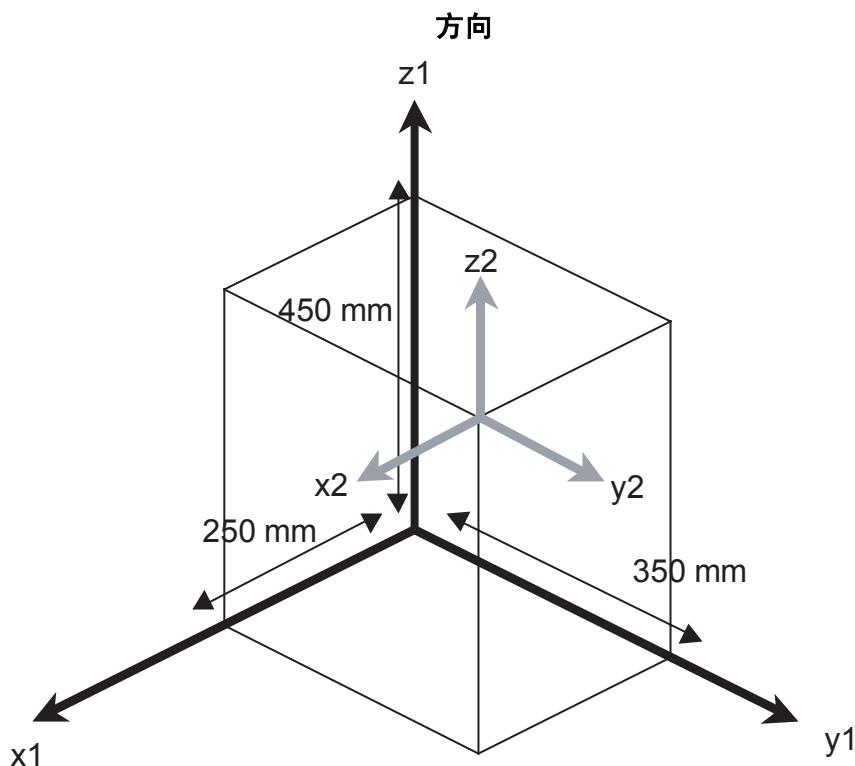
num x	x 轴的平移
num y	y 轴的平移
num z	z 轴的平移
num rx	绕 x 轴旋转
num ry	绕 y 轴旋转
num rz	绕 z 轴旋转

x, y 和 **z** 字段使用软件应用的长度单位来表示（毫米或英寸，参见长度单位章节）。**rx, ry** 和 **rz** 字段用度表示。

x, y 和 **z** 坐标是平移的笛卡尔坐标(或者是点的位置或者是参考坐标系内的坐标系)。当**rx, ry** 和 **rz** 是零时，是一个不改变方向的平移变换。

当一个 **trs**f 类型变量被初始化时，其默认值为 {0,0,0,0,0,0}。

9.3.2. 方向



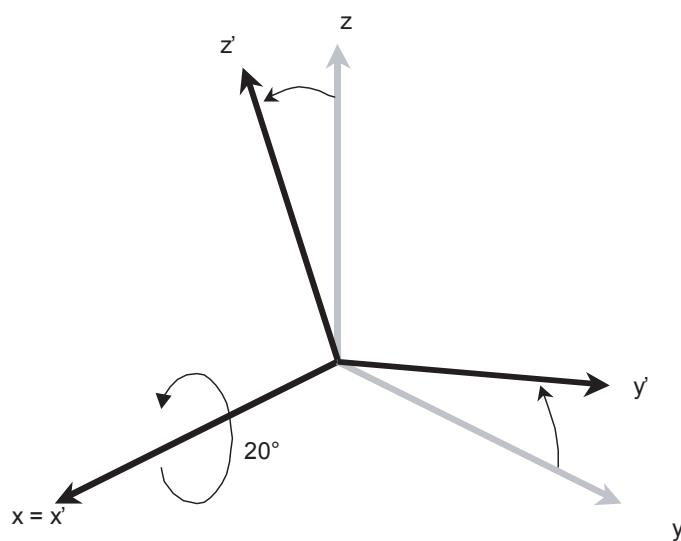
R2 坐标系（灰色）相对于 **R1**（黑色）的位置是：

$x = 250\text{mm}$, $y = 350\text{ mm}$, $z = 450\text{mm}$, $rx = 0^\circ$, $ry = 0^\circ$, $rz = 0^\circ$

rx, **ry** 和 **rz** 坐标对应于必须依次绕 **x**, **y** 和 **z** 轴作用的旋转角度，以获得坐标系的方向。

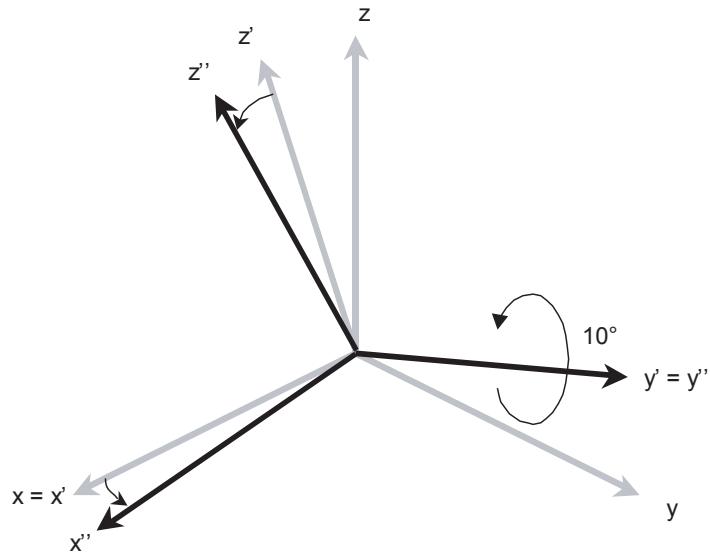
例如， $rx = 20^\circ$, $ry = 10^\circ$, $rz = 30^\circ$ 方向是由以下方式得到的。首先，坐标系 (x,y,z) 绕轴 **x** 旋转 **20°**。得到一个新的 (x',y',z') 坐标系。**x** 轴和 **x'** 轴重合。

绕轴的坐标系旋转：X 轴



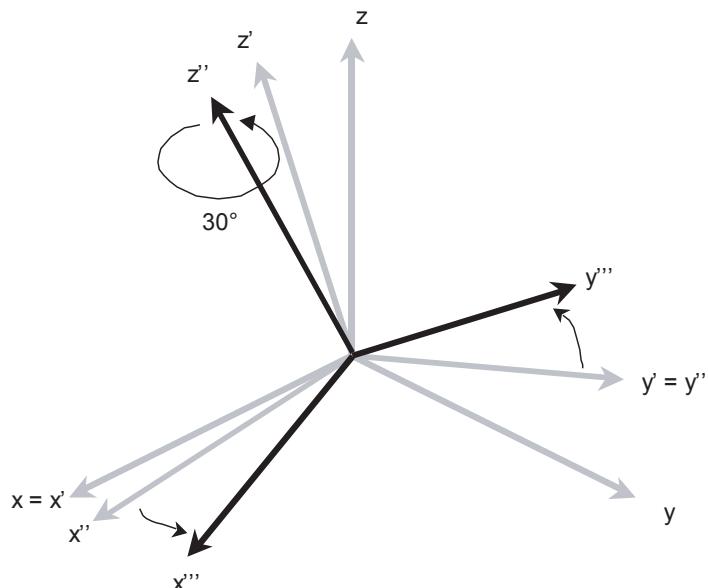
然后，坐标系统前一步骤得到坐标系的轴 **y'** 转动 **20°**。得到一个新的 (x'',y'',z'') 坐标系。**y'** 轴和 **y''** 轴重合。

绕轴的坐标系旋转: Y' 轴



最后, 坐标系绕前一步骤得到的坐标系的 **z''** 轴旋转 **20°**。得到新坐标系 (**x''',y''',z'''**), 其方向就是由 **rx, ry, rz** 定义的。**z''** 轴和 **z'''** 轴重合。

绕轴的坐标系旋转: Z'' 轴



R2 坐标系 (灰色) 相对于 **R1** (黑色) 的位置是:
 $x = 250\text{mm}$, $y = 350 \text{ mm}$, $z = 450\text{mm}$, $rx = 20^\circ$, $ry = 10^\circ$, $rz = 30^\circ$

rx, ry 和 **rz** 的值是以 **360** 度为模数的。当系统计算 **rx, ry** 和 **rz** 时, 它们的值总是在 **-180** 和 **+180** 度之间。**rx, ry**, 和 **rz** 还可能有另外几个值: 系统确保至少两个坐标在 **-90** 和 **90** 度之间 (除非 **rx** 是 **+180** 和 **ry** 是 **0**)。当 **ry** 等于 **90** 度 (modulo 180) 时, **rx** 的选值为零。

9.3.3. 运算符

按递增优先顺序：

trsf <trsf& trPosition1> = <trsf trPosition2>	逐字段将 trPosition2 赋值给 trPosition1 变量，并返回 trPosition2 。
bool <trsf trPosition1> != <trsf trPosition2>	如果一个 trPosition1 字段不等于对应的 trPosition2 字段，返回 true ，否则返回 false 。
bool <trsf trPosition1> == <trsf trPosition2>	如果每个 trPosition1 字段等于对应的 trPosition2 字段，返回 true ，否则返回 false 。
trsf <trsf trPosition1> * <trsf trPosition2>	返回 trPosition1 和 trPosition2 变换的几何组成。注意！在通常情况下， trPosition1 * trPosition2 != trPosition2 * trPosition1!
trsf ! <trsf trPosition>	返回 trPosition 的逆变换。

为了避免 = 和 == 操作符的混淆，= 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。

9.3.4. 指令

num **distance(trsf trPosition1, trsf trPosition2)**

功能

返回 **trPosition1** 和 **trPosition2** 之间的距离。

注意：
要确保距离有效，位置 1 和位置 2 必须相对于同一个坐标系来定义。

例如

该行计算两个工具之间的距离：

```
distance(position(tTool1, flange), position(tTool2, flange))
```

另见

point appro(point pPosition, trsf trTransformation)
point compose(point pPosition, frame fReference, trsf trTransformation)
trsf position(point pPosition, frame fReference)
num distance(point pPosition1, point pPosition2)

trsfcf interpolateL(trsfcf trStart, trsfcf trEnd, num nPosition)

功能

该指令返回与起始位置 **trStart** 和目标位置 **trEnd** 成一直线的一个中间位置。**nPosition** 参数规定根据方程式要应用于坐标系 x 的的线性插值: $\text{trsfcf.x0} = \text{trStart.x} + (\text{trEnd.x}-\text{trStart.x}) * \text{nPosition}$ 。同样的方程式也适用于 Y 和 Z 坐标系。

rx, ry, rz 方位是根据相同的方程式进行计算的, 但更复杂一些。**interpolateL**使用的算法与运动产生器在一个移动指令时来计算中间位置所使用的算法一样。

interpolateL(trStart, trEnd, 0)返回**trStart**; **interpolateL(trStart, trEnd, 1)**返回**trEnd**; **interpolateL(trStart, trEnd, 0.5)**返回 **trStart** 与 **trEnd** 之间的中间位置。**nPosition** 参数的一个负值给出一个位于 **trStart**"前面" 的位置。一个大于 1 的数值给出一个位于 **trEnd**"后面" 的位置。

如果参数 **nPosition** 不在区间 $] -1, 2 [$ 内, 则产生一个执行错误。

另见

trsfcf position(point pPosition, frame fReference)

trsfcf position(frame fFrame, frame fReference)

trsfcf position(tool tTool, tool tReference)

trsfcf interpolateC(trsfcf trStart, trsfcf trIntermediate, trsfcf trEnd, num nPosition)

trsfcf align(trsfcf trPosition, trsfcf Reference)

trsf interpolateC(trsf trStart, trsf trIntermediate, trsf trEnd, num nPosition)

功能

这条指令返回由位置 **trStart**, **trIntermediate** 和 **trEnd** 确定的一个圆的弧上的一个中间位置。**nPosition** 参数指明要应用的圆弧插补。**interpolateC** 使用的算法与运动产生器在 **movec** 指令中用来计算中间位置的算法是一样的。

interpolateC(trStart, trIntermediate, trEnd, 0) 返回 **trStart**; **interpolateC(trStart, trIntermediate, trEnd, 1)** 返回 **trEnd**; **interpolateC(trStart, trIntermediate, trEnd, 0.5)** 返回 **trStart** 和 **trEnd** 之间的弧上的一个中间位置。**nPosition** 参数的一个负值给出一个位于 **trStart** "前面" 的位置。一个大于 1 的数值给出一个位于 **trEnd** "后面" 的位置。

如果圆弧没有被正确定义 (位置太近), 或旋转插值仍未确定 (见 "运动控制 - 方向的插值" 章节), 则产生一个运行时错误。

另见

trsf position(point pPosition, frame fReference)
trsf position(frame fFrame, frame fReference)
trsf position(tool tTool, tool tReference)
trsf interpolateL(trsf trStart, trsf trEnd, num nPosition)
trsf align(trsf trPosition, trsf Reference)

trsf align(trsf trPosition, trsf Reference)

功能

此指令返回一个带经修改方向的输入值 **trPosition**, 使得所返回方向的 Z 轴与最接近 **trReference** 的参考方向的 X, Y 或 Z 轴对齐。**trPosition** 和 **trReference** 的坐标 X, Y, Z 没有被使用: 返回数值的 x, y, z 坐标与 **trPosition** 的 x, y, z 坐标相同。

另见

trsf position(point pPosition, frame fReference)
trsf position(frame fFrame, frame fReference)
trsf position(tool tTool, tool tReference)
trsf interpolateL(trsf trStart, trsf trEnd, num nPosition)
trsf interpolateC(trsf trStart, trsf trIntermediate, trsf trEnd, num nPosition)

9.4. FRAME 类型

9.4.1. 定义

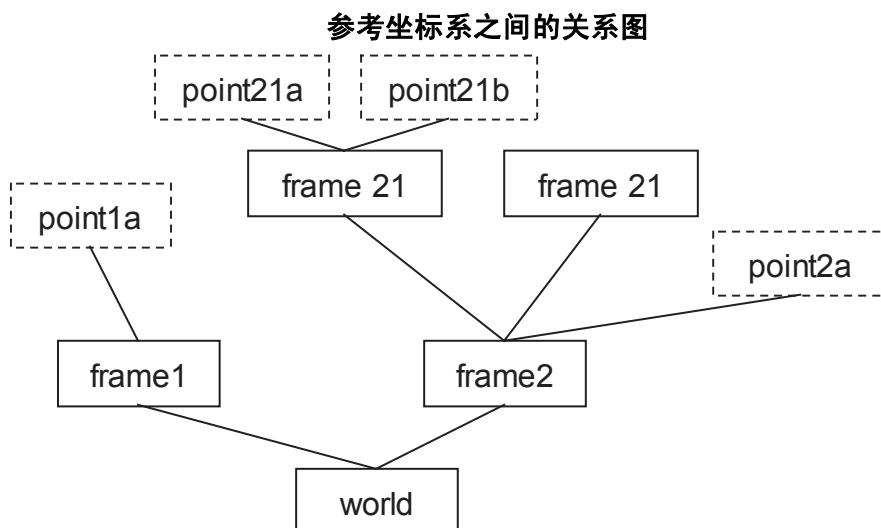
Frame 类型用来定义在自动化设备中的参考坐标系的位置。

Frame 类型属于结构类型，只有一个可访问的字段：

trsfrsf 在参考坐标系中的坐标系位置

一个 frame 类型变量的 **参照坐标系** 在它的初始化时被定义（通过用户界面或者运算符 =，或者 **link()** 指令）。**world** 参考坐标系总是在一个 **VAL 3** 软件应用中被定义：一个参考坐标系是直接或通过其他坐标系与 **world** 坐标系相连接。

如果 **world** 坐标系坐标已经被修改，则在几何计算过程中会产生一个运行时错误。



默认时，局部坐标系变量和用户类型变量中的坐标系没有参考坐标系。在它们可以被使用之前，必须用 '=' 运算符来进行初始化，或者通过 **link()** 和 **setFrame()** 指令中的一个来将它们初始化。

9.4.2. 使用

在机器人的应用中，对于下述用途，强烈建议使用参考坐标系：

- **给予应用点视看更直观**

工作单元的点的显示是按照坐标系的层次结构来组织的。

- **快速更新点集合的位置**

当一个软件应用点与一物体连接时，就应该给这物体定义一个坐标系，并将 **VAL 3** 点与这个坐标系相连。这样，如果物体移动，为使所有与其相连接的点同时随之改变，只需要重新定义坐标系就行了。

- **将轨迹复制到机器人工作单元的多个区域中**

可以定义此轨迹的点相对于一个工作坐标系的位置，并将轨迹要复制的每个区域告诉一个坐标系。把一个被示教坐标系的值赋予工作坐标系，从而整个轨迹就“转移”到了被示教的坐标系上。

- **用于几何位移简易计算**

compose() 指令用于执行在任何一个参考坐标系中表示的所有点的几何位移。**position()** 指令用于计算在任一个参考坐标系中的点的位置。

9.4.3. 运算符

按递增优先顺序：

frame <frame& fReference1> = <frame fReference2>	将 fReference2 的位置和参考坐标系赋值给变量 fReference1 。
bool <frame fReference1> != <frame fReference2>	如果 fReference1 和 fReference2 没有相同的参考坐标系或者在它们的参考坐标系中没有相同的位置，就返回 true 。
bool <frame fReference1> == <frame fReference2>	如果 fReference1 和 fReference2 在同一个参考坐标系中有相同的位置，就返回 true 。

为了避免 = 和 == 操作符的混淆，= 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。

9.4.4. 指令

num setFrame(point pOrigin, point pAxisOx, point pPlaneOxy, frame& fResult)

功能

该指令计算来自其原点 **pOrigin** 的，来自轴 (**Ox**) 上的 **pAxisOx** 点的，以及在平面 (**Oxy**) 上的一个 **pPlaneOxy** 点的 **fResult** 的坐标。

pAxisOx 点必须在 **x** 的正轴侧上。 **pPlaneOxy** 点必须在 **y** 的正轴侧上。

函数返回：

- 0** 无错误。
- 1** 点 **pAxisOx** 太接近 **pOrigin**。
- 2** 点 **pPlaneOxy** 太接近 (**Ox**) 轴。

如果其中一个点没有参考坐标系，则产生一个运行时错误。

trsf position(frame fFrame, frame fReference)

功能

此指令返回在参考坐标系 **fReference** 中的坐标系 **fFrame** 的坐标。

如果 **fFrame** 或 **fReference** 没有参考坐标系，则产生一个运行时错误。

另见

trsf position(point pPosition, frame fReference)
trsf position(tool tTool, tool tReference)

void link(frame fFrame, frame fReference)

功能

该指令改变 **fFrame** 的参考坐标系，并将它设置为 **fReference**。在参考坐标系中的坐标系位置保持不变。

另见

Operator frame <frame& fFrame1> = <frame fFrame2>

9.5. TOOL 类型

9.5.1. 定义

tool 类型可用来定义一个工具的几何形状和动作。

tool 类型是一种具有下列字段的结构类型，排序如下：

trs_f trs_f

在它的基本工具中的工具中间点 (TCP) 的位置

dio gripper

用于启动工具的输出

num otime

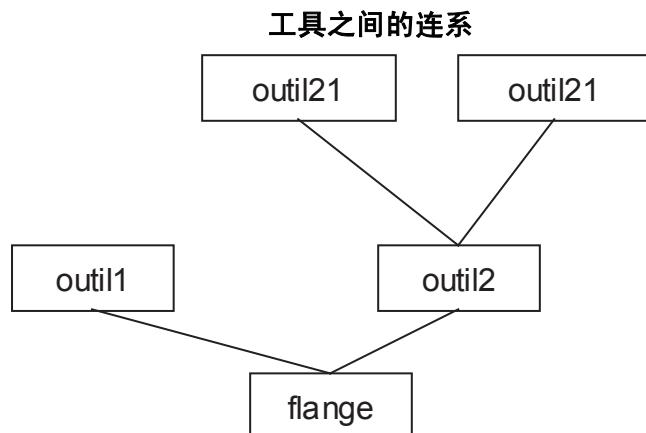
打开工具的时限 (秒)

num ctime

工具的关闭时限 (秒)

tool 类型变量的参考工具在它的初始化时被定义（通过用户界面，= 运算符或者 [link\(\)](#) 指令）。**flange** 工具总是在一个 **VAL 3** 应用中被定义的：所有的工具直接或通过别的工具与 **flange** 工具相联系。

如果 **flange** 工具坐标没有被修改，则在进行几何计算时会产生一个运行时错误。



在默认情况下，一个工具的输出就是 **valve1** 系统的输出，打开和关闭的时间为 **0**，基本工具为 **flange**。局部工具变量，和用户类型变量中的工具没有参考工具。在它们可以被使用之前，必须由另一个工具用 '=' 运算符或者 [link\(\)](#) 指令来将它们初始化。

9.5.2. 使用

强烈建议在机器人应用中使用工具：

- **用来控制移动速度**

当移动是手动控制时或是程序控制时，系统控制工具端头的笛卡尔速度。

- **使不同的工具到达同一点**

只须简单地选择一个与手臂末端的物理工具对应的 **VAL 3** 工具。

- **管理工具磨损或者工具更换**

通过更新工具的几何坐标来简单地补偿工具磨损。

9.5.3. 运算符

按递增优先顺序：

tool <tool& tTool1> = <tool tTool2>	将 tTool2 的位置和基本工具赋予变量 tTool1 。
bool <tool tTool1> != <tool tTool2>	如果 tTool1 和 tTool2 没有相同的基本工具，在基本工具中没有相同的位置，没有相同的数字输出或者没有相同的打开或关闭时间，则返回 true 。
bool <tool tTool1> == <tool tTool2>	如果 tTool1 和 tTool2 在基本工具中位置相同，使用相同的数字输出以及相同的打开或关闭时间，则返回 true 。

为了避免 = 和 == 操作符的混淆，= 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。

9.5.4. 指令

void open(tool tTool)

功能

该指令通过将工具的数字输出设置为 **true** 激活工具（打开）。

在激活工具之前，通过执行等同的一个 **waitEndMove()** 的方式，**open()** 等待机器人到达要求的点。在激活之后，系统在执行下一条指令之前，等待 **otime** 秒。

该指令不能保证在工具激活前机器人在最终位置的稳定性。在需要等待运动完全稳定的情况下，必须使用 **isSettled()** 指令。

如果 **tTool** 的 **dio** 没有被定义，或者不是一个输出，或者先前保存的一个运动指令无法运行，则会产生一个运行时错误。

例如

```
// 指令 open() 等同于:  
waitEndMove()  
tTool.gripper=true  
delay(tTool.otime)
```

另见

void close(tool tTool)
void waitEndMove()

void **close(tool tTool)**

功能

该指令通过将工具的数字输出设置为 **false** 来激活工具（关闭）。

在启动工具之前，通过等同于一个 **waitEndMove()** 的方式，**close()** 等待机器人完全停止。在激活之后，系统在执行下一条指令之前，等待 **ctime** 秒。

该指令不能保证在工具激活前机器人在最终位置的稳定性。在需要等待运动完全稳定的情况下，必须使用 **isSettled()** 指令。

如果 **tTool** 的 **dio** 没有被定义，或者不是一个输出，或者先前保存的一个运动指令无法运行，则会产生一个运行时错误。

例如

```
// 关闭指令等同于:  
waitEndMove()  
tTool.gripper = false  
delay(tTool.ctime)
```

另见

Type **tool**

void open(tool tTool)

void waitEndMove()

trsf **position(tool tTool, tool tReference)**

功能

此指令返回在 **tReference** 工具中的工具 **tTool** 的坐标。

如果 **tTool** 或 **tReference** 没有参考工具，则会产生一个运行时错误。

另见

trsf position(point pPosition, frame fReference)

trsf position(frame fFrame, frame fReference)

void **link(tool tTool, tool tReference)**

功能

该指令改变 **tTool** 的参考工具，并将它设置为 **tReference**。在参考工具中的工具位置保持不变。

另见

Operator **tool <tool& tTool1> = <tool tTool2>**

9.6. POINT 类型

9.6.1. 定义

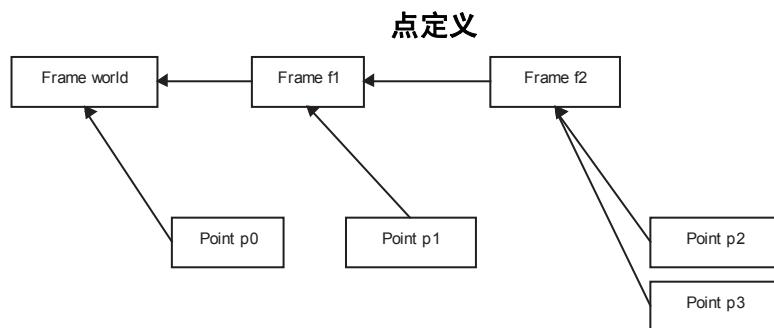
point 类型用于定义在自动化装置中机器人的工具的位置和方向。

point 类型是一种具有下列字段的结构类型，排序如下：

trsft **trTrsf** 参考坐标系中点的位置

config config 用于到达目标位置的手臂设置

一个 **point** 的参考坐标系是 **frame** 类型的变量，它是在它的初始化阶段被定义的（在用户界面中，用运算符 **=** 和指令 **link()**, **here()**, **appro()** 和 **compose()**）。



如果使用了一个没有定义参考坐标系的 **point** 类型变量，产生一个运行时错误。

注意：

默认时，局部点变量和用户类型变量中的点没有参考坐标系。在它们可以使用之前，必须由另一个点通过 '**=**' 运算符来初始化，或者通过 **link()**, **here()**, **appro()** 和 **compose()** 指令来初始化。

9.6.2. 运算符

按递增优先顺序：

point <point& pPoint1> = <point pPoint2>	将 pPoint2 的位置、设置和参考坐标系赋予变量 pPoint1 。
bool <point pPoint1> != <point pPoint2>	如果 pPoint1 和 pPoint2 没有相同的参考坐标系或者在它们的参考坐标系中没有相同的位置，就返回 true 。
bool <point pPoint1> == <point pPoint2>	如果 pPoint1 和 pPoint2 在同一个参考坐标系中有相同的位置，就返回 true 。

为了避免 **=** 和 **==** 操作符的混淆，**=** 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。

9.6.3. 指令

num distance(point pPosition1, point pPosition2)

功能

该功能返回 **pPosition1** 和 **pPosition2** 之间的距离。

如果 **pPosition1** 或 **pPosition2** 没有定义的参考坐标系，产生一个运行时错误。

例如

此程序等待手臂接近位置 **pTarget** 的距离小于 10 mm

```
wait (distance (here (tTool,world),pTarget)< 10)
```

另见

point appro(point pPosition, trsf trTransformation)
point compose(point pPosition, frame fReference, trsf trTransformation)
trsf position(point pPosition, frame fReference)
num distance(trsf trPosition1, trsf trPosition2)

point compose(point pPosition, frame fReference, trsf trTransformation)

功能

该指令返回 **pPosition**，相对于 **fReference** 坐标系而定义的几何变换 **trTransformation** 被应用在此位置上。

注意:

tTransformation 的转动分量通常不仅仅改变 **pPosition** 的方向，还改变它们的直角坐标（除非 **pPosition** 位于 **fReference** 的坐标原点上）。

如果希望 **tTransformation** 仅改变 **pPosition** 的方向，必须用 **pPosition** 的直角坐标来更新结果（参看举例）。

返回点的参考坐标系和设置就是 **pPosition** 的参考坐标系和设置。

如果 **pPosition** 没有任何定义的参考坐标系，产生一个运行时错误。

例如

```
// 改变方向, Position 不改变
pResult = compose (pPosition,fReference,trTransformation)
pResult.trsf.x = pPosition.trsf.x
pResult.trsf.y = pPosition.trsf.y
pResult.trsf.z = pPosition.trsf.z

// 改变 Position, 不改变方向
trTransformation.rx = trTransformation.ry =trTransformation.rz = 0
pResult = compose (pResult,fReference,trTransformation)
```

另见

Operator trsf <trsf pPosition1> * <trsf pPosition2>
point appro(point pPosition, trsf trTransformation)

point appro(point pPosition, trsf trTransformation)

功能

此指令返回一个经几何变换的修改点。几何变换按照与输入点相同的参考坐标系被定义。

参考坐标系和返回的点的设置就是输入点的坐标系和设置。

如果 **pPosition** 没有任何定义的参考坐标系，产生一个运行时错误。

例如

```
// 近似法：运动到点(z 轴)的上方 100 mm mm 处
movej(appro(pDestination, {0,0,-100,0,0,0}), flange, mNomDesc)
// 到点
movel(pDestination, flange, mNomDesc)
```

另见

Operator trsf <trsf trPosition1> * <trsf trPosition2>
point compose(point pPosition, frame fReference, trsf trTransformation)

point here(tool tTool, frame fReference)

功能

该指令返回 **tTool** 工具在 **fReference** 坐标系中的当前位置（命令的位置和非测量位置）。返回点的参考坐标系是 **fReference**。所返回的点的设置就是手臂的当前设置。

另见

joint herej()
config config(joint jPosition)
point jointToPoint(tool tTool, frame fReference, joint jPosition)

point jointToPoint(tool tTool, frame fReference, joint jPosition)

功能

当手臂在关节位置 **jPosition** 时，该指令返回 **fReference** 坐标系中的 **tTool** 的位置。

返回点的参考坐标系是 **fReference**。所返回的点的设置就是在关节 **jPosition** 位置的手臂的设置。

另见

point here(tool tTool, frame fReference)
bool pointToJoint(tool tTool, joint jInitial, point pPosition, joint& jResult)

**bool pointToJoint(tool tTool, joint jInitial, point pPosition,
joint& jResult)**

功能

此指令计算与指明点 **pPosition** 对应的关节位置 **jResult**。如果 **jResult** 被更新，指令返回 **true**，如果无解，则返回 **false**。

关节位置根据 **pPosition** 点的手臂姿态而确定。值为 **free** 的字段不决定设置。值为 **same** 的字段指明与 **jInitial** 相同的设置。

对于那些可以转动大于一周的轴，可有几个具有完全相同设置的关节方案：那么选择最接近 **jInitial** 的方案。

如果 **pPosition** 够不到（手臂太短）或超出软件限制，就不会有方案。如果 **pPosition** 指定一个设置，对于这个设置可能超出软件限制，但对于一个其他设置却在限制之内。

如果 **pPosition** 没有任何定义的参考坐标系，产生一个运行时错误。

另见

joint herej()

point jointToPoint(tool tTool, frame fReference, joint jPosition)

trsf position(point pPosition, frame fReference)

功能

该功能返回在 **fReference** 中的 **pPosition** 的坐标。

如果 **pPosition** 没有参考坐标系，则产生一个运行时错误。

例如

2 个点之间的距离就是它们在世界坐标中的位置的距离：

distance(position(pPoint1, world), position(pPoint2, world)) is **distance(pPoint1, pPoint2)**

另见

num distance(point pPosition1, point pPosition2)

trsf position(tool tTool, tool tReference)

trsf position(frame fFrame, frame fReference)

void link(point pPoint, frame fReference)

功能

该指令改变 **pPoint** 的参考坐标系，并将它设置为 **fReference**。在参考坐标系中的点的位置保持不变。

另见

Operator point <point& pPoint1> = <point pPoint2>

9.7. CONFIG 类型

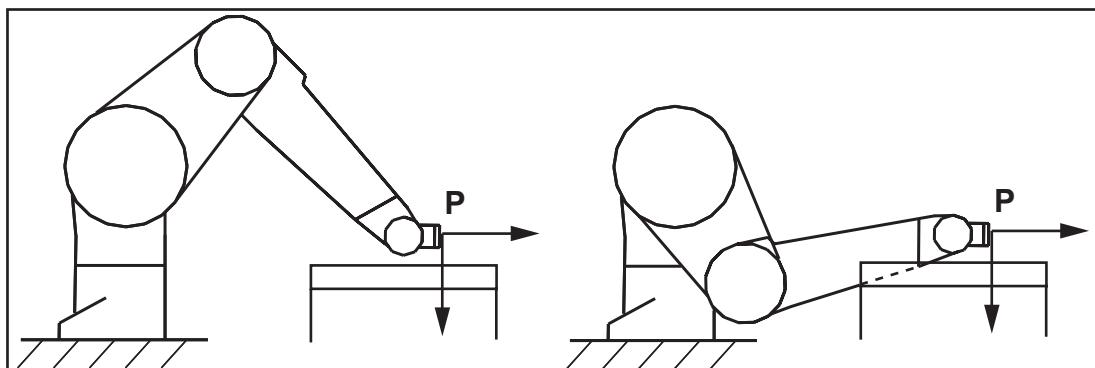
笛卡尔点的姿态概念是一个“高级”概念，如果您第一次阅读这个文档，可以先跳过。

9.7.1. 引言

通常机器人有多种可能的方式到达给定的笛卡尔点。

这些不同的可能性被称为“姿态”。下图解释了两种不同的姿态：

到达一个指定点的两种不同姿态：P



在某些情况下，在所有可能的姿态中，指明有效姿态和禁止姿态是很重要的。要处理这个问题，**point** 类型用来指明该机器人的允许姿态，通过下面定义的 **config** 类型来进行。

9.7.2. 定义

config 类型是用来给一个给定的笛卡尔位置定义允许的姿态。

它取决于使用的机器人手臂的类型。

对于一个史陶比尔 (Stäubli)RX/TX 机器人手臂，**config** 类型是一个结构类型，它的字段按序如下：

shoulder	肩部姿态
elbow	肘部姿态
wrist	腕部姿态

对于一个史陶比尔 (Stäubli)RS/TS 机器人手臂，**config** 类型仅限于 **Shoulder** 字段：

shoulder	肩部姿态
-----------------	------

shoulder,elbow 和 **wrist** 字段可以取下述值：

shoulder	righty	强制的 righty 肩部姿态
	lefty	强制的 lefty 肩部姿态
	ssame	禁止肩部姿态的改变
	sfree	自由肩部姿态

elbow	epositive	强制的 epositive 肘部姿态
	enegative	强制的 enegative 肘部姿态
	esame	禁止肘部姿态的改变
	efree	自由肘部姿态

wrist	wpositive	强制的 wpositive 手腕姿态
	wnegative	强制的 wnegative 手腕姿态
	wsame	禁止手腕姿态的改变
	wfree	自由手腕姿态

9.7.3. 运算符

按递增优先顺序：

config <config& configuration1> = <config configuration2>	将 configuration2 的 shoulder , elbow 和 wrist 字段赋予变量 configuration1 。
bool <config configuration1> != <config configuration2>	如果 configuration1 和 configuration2 与 shoulder , elbow 或 wrist 字段的值不同，则返回 true 。
bool <config configuration1> == <config configuration2>	如果 configuration1 和 configuration2 与 shoulder , elbow 或 wrist 的字段值相同，则返回 true 。

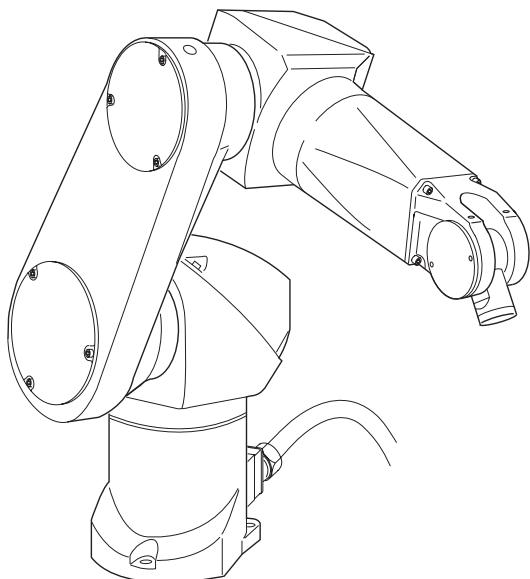
为了避免 = 和 == 操作符的混淆，= 操作符不允许在 **VAL 3** 表达式中作为一个指令参数使用。

9.7.4. 姿态 (RX/TX 机器人手臂)

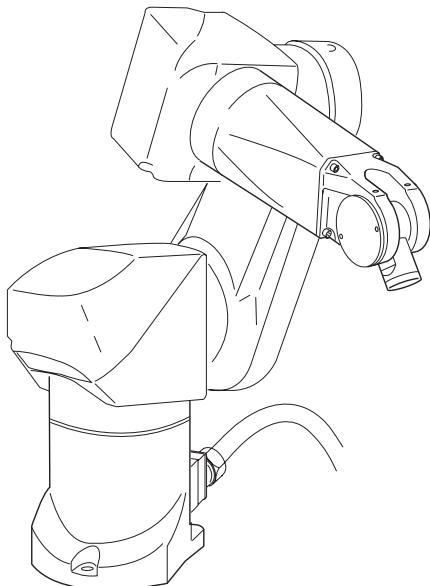
9.7.4.1. 肩部姿态

为了到达一个给定的笛卡尔点，机器人的手臂可在此点的右边或左边：这两种姿态分别被称为 **righty** 和 **lefty**。

姿态: righty



姿态: lefty

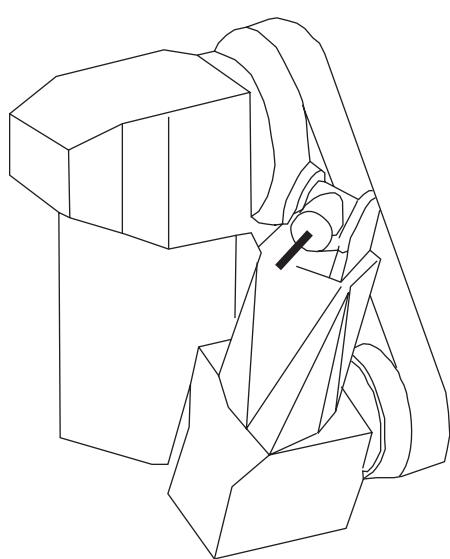


righty 姿态由 $(d1 * \sin(j2) + d2 * \sin(j2+j3) + \delta) < 0$ 定义, lefty 姿态由 $(d1 * \sin(j2) + d2 * \sin(j2+j3) + \delta) \geq 0$ 定义, 其中 $d1$ 是机器人手臂的长度, $d2$ 是前臂的长度, δ 是沿方向 x 上的轴 1 和轴 2 之间的距离。

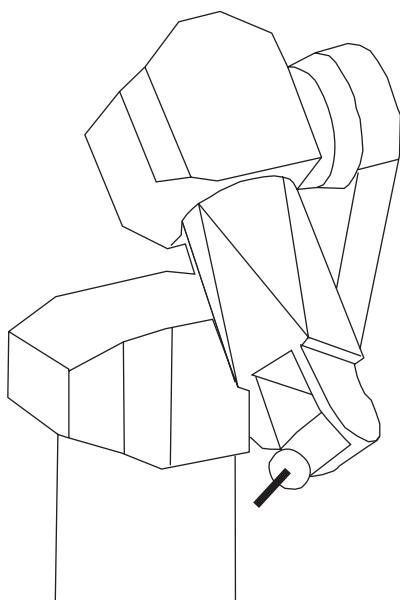
9.7.4.2. 肘部姿态

除了肩部的姿态外，机器人的肘部也有两种姿态：肘部姿态被称为 **epositive** 和 **enegative**。

姿态: enegative



姿态: epositive



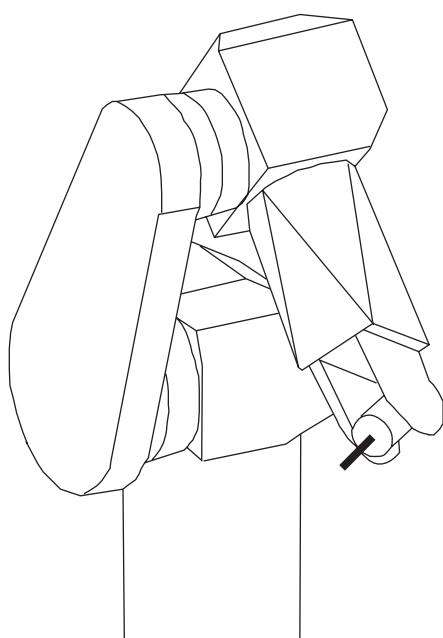
epositive 姿态由 $j3 \geq 0$ 定义。

enegative 姿态由 $j3 < 0$ 定义。

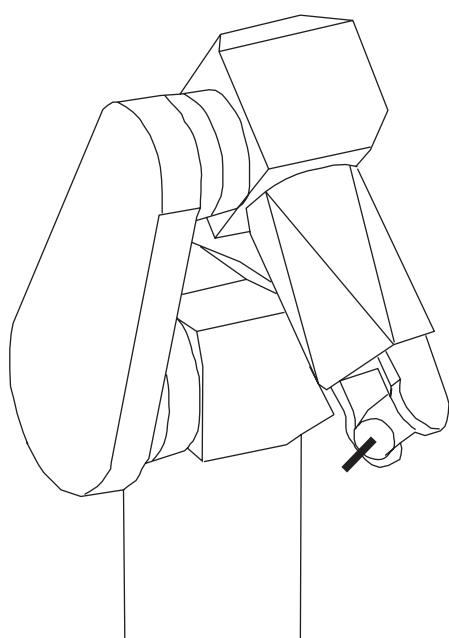
9.7.4.3. 腕部姿态

除了肩部姿态和肘部姿态之外，有两种机器人的腕部姿态。这两个腕部姿态被称为 **wpositive** 和 **wnegative**。

姿态: wnegative



姿态: wpositive



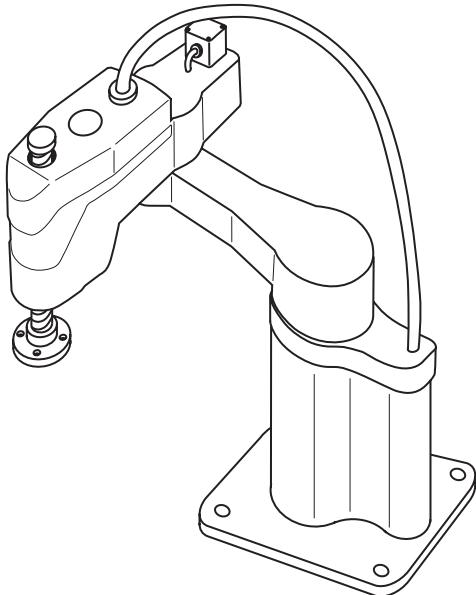
wpositive 姿态由 $j5 \geq 0$ 定义。

wnegative 姿态由 $j5 < 0$ 定义。

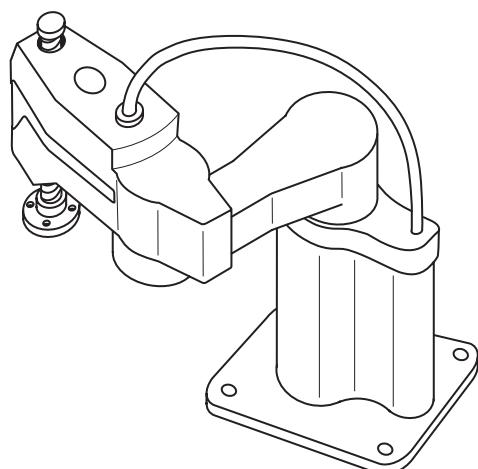
9.7.5. 姿态 (RS/TS 机器人手臂)

为了到达一个给定的笛卡尔点，机器人的手臂可在此点的右边或左边：这两种姿态分别被称为 **righty** 和 **lefty**。

姿态: righty



姿态: lefty



righty 的姿态由 $\sin(j2) > 0$ 来定义, **lefty** 的姿态由 $\sin(j2) \leq 0$ 来定义。

9.7.6. 指令

config config(joint jPosition)

功能

该指令返回关节 **jPosition** 位置的手臂姿态。

另见

point here(tool tTool, frame fReference)
joint herej()

章节 10

运动控制

10.1. 轨迹控制

连续的点不足以定义机器人的运动轨迹。还要指明这些点之间使用的轨迹类型（曲线或直线），指明轨迹之间的连接方式并定义运动速度的参数。这个章节介绍不同的运动形式 (**movej**, **movel** 和 **movec** 指令)，描述如何使用运动描述符参数 (**mdesc** 类型)。

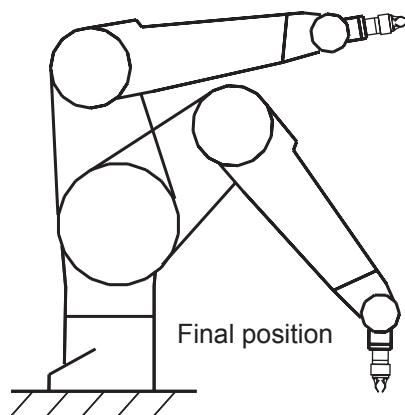
10.1.1. 运动类型：点到点，直线，圆周

机器人的运动主要使用 **movej**, **movel** 和 **movec** 指令进行编程。**movej** 指令用来编制点到点的运动，**movel** 用来编制直线运动，**movec** 用来编制圆周运动。

点对点运动是一个只有最终目的地（笛卡尔点或关节位置）是重要的运动。为了优化运动速度，在起点和终点之间，工具中心沿着一条由系统定义的曲线运动。

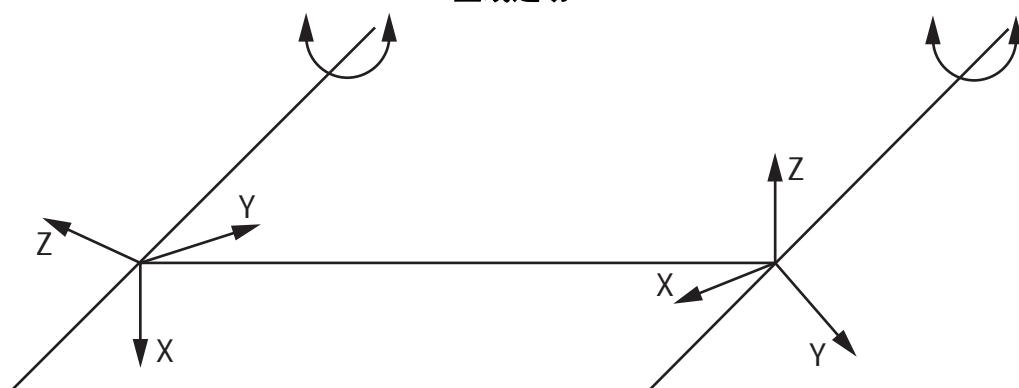
起始和终点位置

Initial position



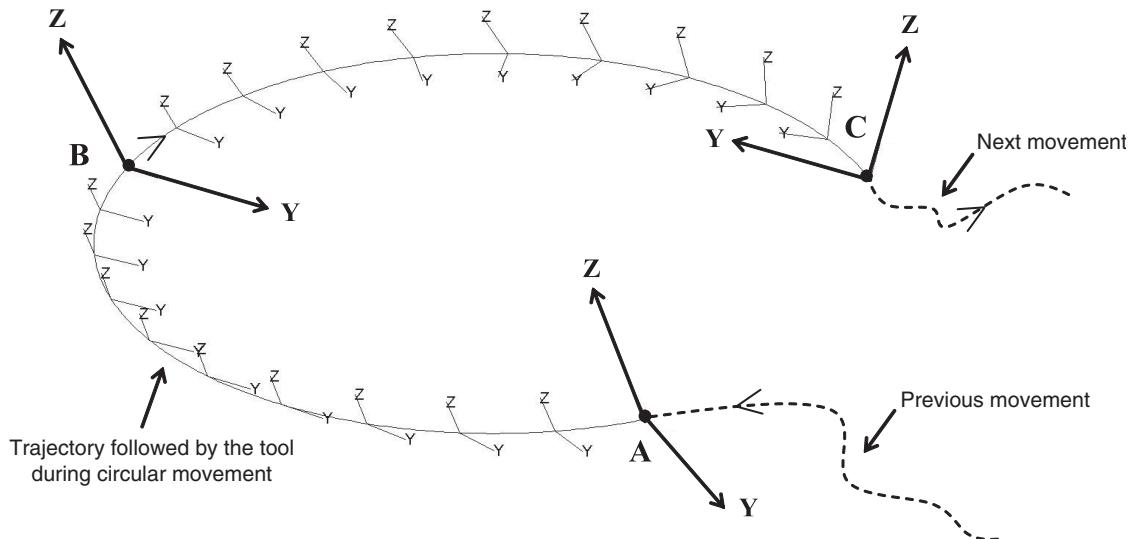
相反地，在直线运动的情况下，工具中心点沿直线运动。在工具起始点方向和终点方向之间以直线内插值法修正方向。

直线运动



在圆周运动中，工具中心点沿由 3 点定义的弧线运动，工具方向在起始点方向、中间点方向和终点方向之间以内插值法取向。

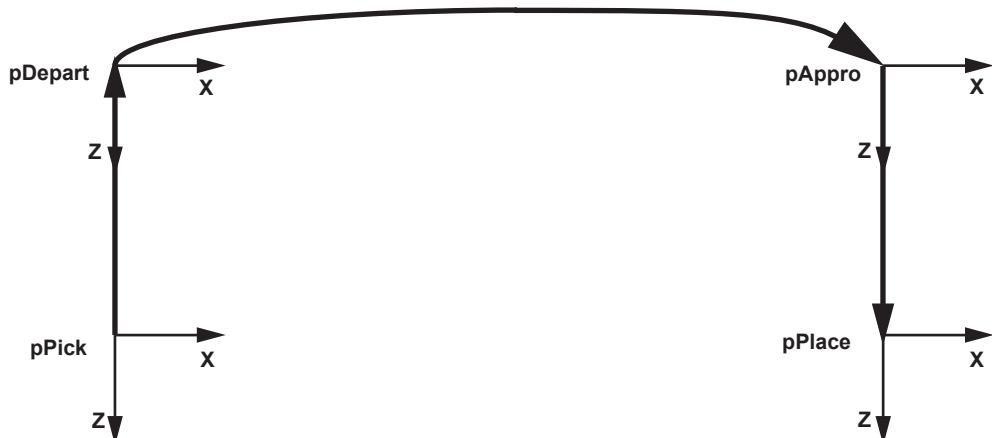
圆周运动



例如：

一个典型的抓取任务包括从一个地方抓取部件然后放到另一个地方。假定要从点 **pPick** 抓取部件，放到点 **pPlace**。从 **pPick** 点到 **pPlace** 点，机器人必须经过脱离点 **pDepart** 和接近点 **pAppro**。

循环型：U



假定机器人最初在点 **pPick**。执行运动的程序语言可以写为：

```
moveL (pDepart, tTool, mDesc)
moveJ (pAppro, tTool, mDesc)
moveL (pPlace, tTool, mDesc)
```

使用直线运动来实现离开或接近。然而，由于这部分轨迹的几何形状无须精确控制，而其目的是尽可能快的运动，所以点到点运动便成了主要的运动方式。

备注:

对于这两种类型的运动，轨迹的几何形状不取决于所执行的运动形式的速度。机器人总是通过同一位置。这一点对于开发应用软件特别重要。我们可以以慢速动作开始，然后逐步提高速度而不使机器人的轨迹变形。

10.1.2. 运动连接：轨迹混合

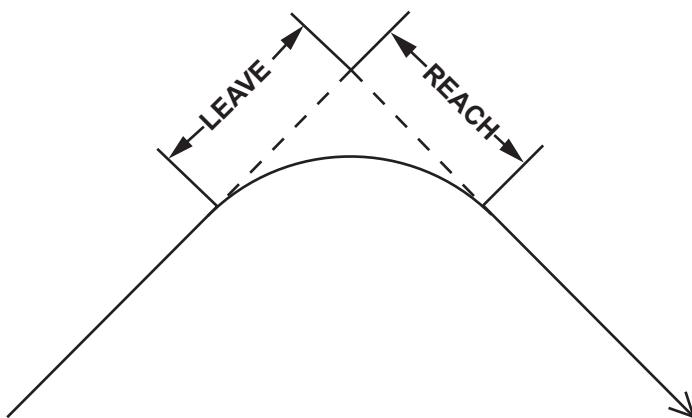
10.1.2.1. 轨迹混合

我们返回前一章节中描述的 **U** 循环的例子。如果没有特别的运动序列控制，机器人在 **pDepart** 和 **pAppro** 点停止，因为轨迹在这些点弯曲成角度。这样就毫无必要地增加了操作时间，同时也没必要精确地通过这些点。

通过将轨迹与邻近点 **pDepart** 和 **pAppro** 混合可大大地减少运动时间。为此，使用运动描述符 **blend** 字段。当此字段值为 **off** 时，机器人在轨迹的每一个点上都停止。不过，当参数设置为 **joint** 或 **Cartesian** 时，轨迹将在每一点附近混合，机器人不再在经过的点上停止。

当字段 **blend** 取值 **joint** 或 **Cartesian** 时，必须指定另外两个参数 **leave** 和 **reach**。这些参数决定了离到达点多少距离时离开标称轨迹（混合开始），离到达点多少距离时回到标称轨迹（混合结束）。

距离的定义: 'leave' / 'reach'

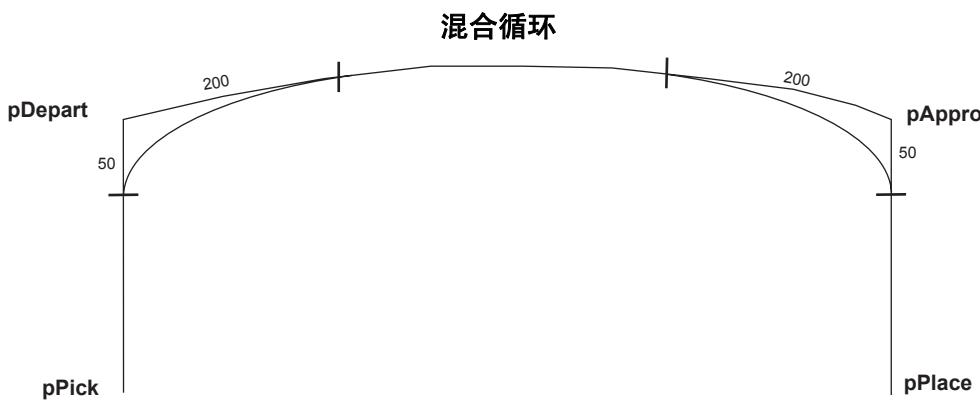


例如:

重新回到标题为“运动类型：点对点或直线”一章中描述的程序编制。前面的运动程序可修改如下：

```
mDesc.blend = joint
mDesc.leave = 50
mDesc.reach = 200
moveL(pDepart, tTool, mDesc)
mDesc.leave = 200
mDesc.reach = 50
moveJ(pAppro, tTool, mDesc)
mDesc.blend = off
moveL(pPlace, tTool, mDesc)
```

得到如下轨迹：



机器人不再在 **pDepart** 和 **pAppro** 点停留。运动因而更快。事实上，**leave** 和 **reach** 之间的距离越大，运动越快。

10.1.2.2. 取消混合作用

waitEndMove() 指令还用来取消混合作用。那么机器人完成最后的程序编制的运动，直至到达点，就像运动描述符的 **blend** 字段已经被设置为 **off**。

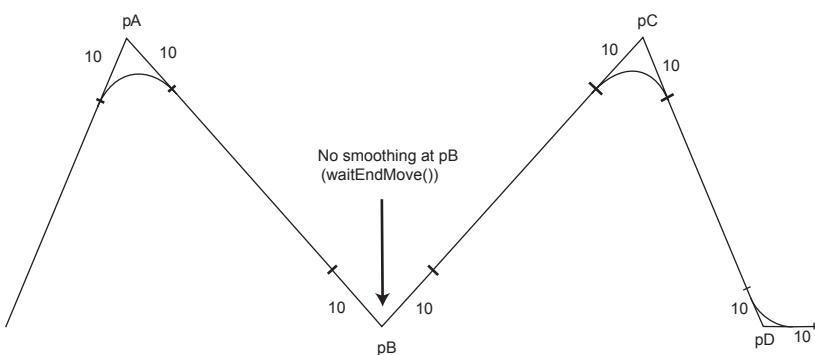
例如，观察下面的程序：

```
mDesc.blend = joint
mDesc.leave = 10
mDesc.reach = 10
movej(pA, tTool, mDesc)
movej(pB, tTool, mDesc)
waitEndMove()
movej(pC, tTool, mDesc)
movej(pD, tTool, mDesc)
```

等等

机器人遵循的轨迹如下：

在给定点无轨迹混合的循环



10.1.2.3. 关节轨迹混合，笛卡尔轨迹混合

为简单起见，一个关节轨迹混合就像在离开和到达点之间的一个点对点的运动，笛卡尔轨迹混合就像这些点之间的一个圆周运动。

- 关节轨迹混合通常比笛卡尔轨迹混合快。但是，当方向改变很复杂时(典型的例子为在一个平面中的一个圆，跟随一个在正交平面中的一个圆)，或者对于纯旋转运动，关节轨迹混合可能会得到奇怪的路径。
- 采用笛卡尔轨迹混合，速度和加速度控制更加准确。还能确保在相同平面上的两个运动之间的笛卡尔轨迹混合在同一平面上。

一个轨迹混合的最优化形状取决于这个软件应用，但 **VAL 3** 编译器必须自动选择该形状。结果不会出乎意料，但有时候它也可能会是 ...

当有效离开和到达距离很不同的时候，选择可能会得到意想不到的复杂形状。所计算的形状减少速度优化的路径弯曲度，但结果可能不适用于某些过程应用。当离开和到达距离相等时，笛卡尔轨迹混合得到的总是一个简单的形状。

笛卡尔轨迹混合对于位置和方向两者都适用。复杂的方向改变可能会影响轨迹混合的形状，从而导致意想不到的结果。方向改变也有一些限制：例如在一个圆周上，当可能有几个解决方案，而系统没有标准来进行选择时，大的方向改变将产生一个运动错误。当这样的情况发生时，需要额外的中间点来帮助系统找到正确的方向插值。

10.1.3. 运动恢复

以紧急停止为例，当手臂电源被切断时，机器人还没有完成它的运动，那么当系统重新接通电源后运动必须再继续。如果机器人在停止期间有手动运动，它可能位于远离它的正常轨迹的一个位置。因此必须确保重启运动而不发生碰撞。**VAL 3** 的轨迹控制功能利用一个“连接运动”使得运动恢复的管理成为可能。

当运动恢复时，系统确保机器人真正位于程序设定的轨迹上：在有偏差的情况下，即便很小，系统都会自动存储一条点到点的运动指令以到达机器人离开它的轨迹处的精确位置：这就是“连接运动”。此运动以慢速进行。这个运动必须由操作者确认，除了在自动模式下，此运动可以在没有人的干预下进行。**autoConnectMove()** 指令用来详细指明自动模式时的运动过程。

resetMotion() 指令用来取消当前运动，并可能编制一个连接动作的程序，使其能慢速并在操作者的控制之下返回一个位置。

10.1.4. 笛卡尔运动的特性（直线，圆）

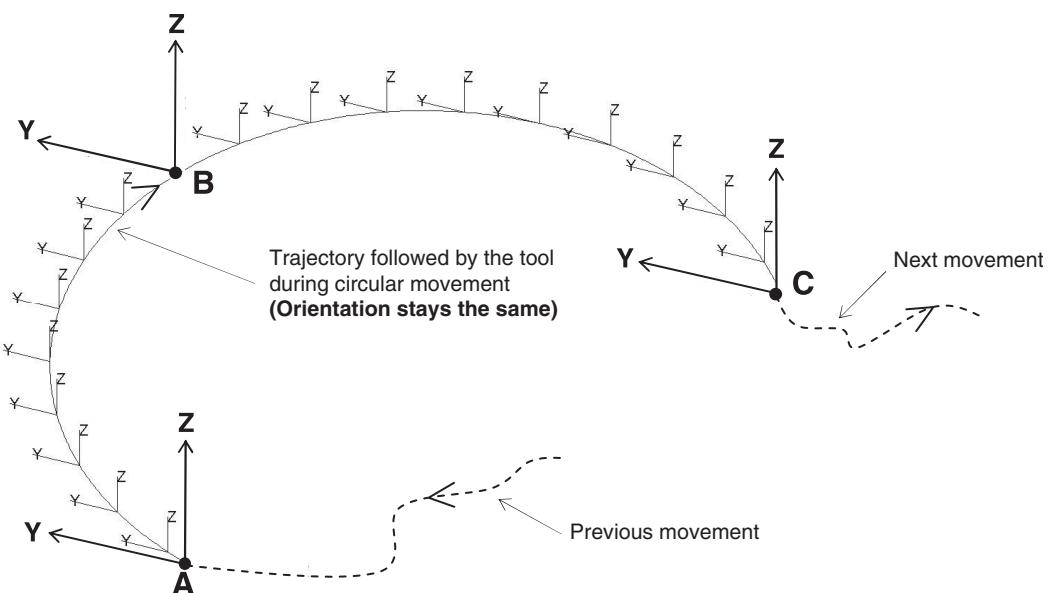
10.1.4.1. 运动方向插值

VAL 3 轨迹发生器从一个方向运动到另一个方向时常常会将工具的旋转幅度最小化。

这使得（作为特殊情况）可以对所有的直线或圆周运动编制一个绝对的，或与该轨迹比较方向不变的程序。

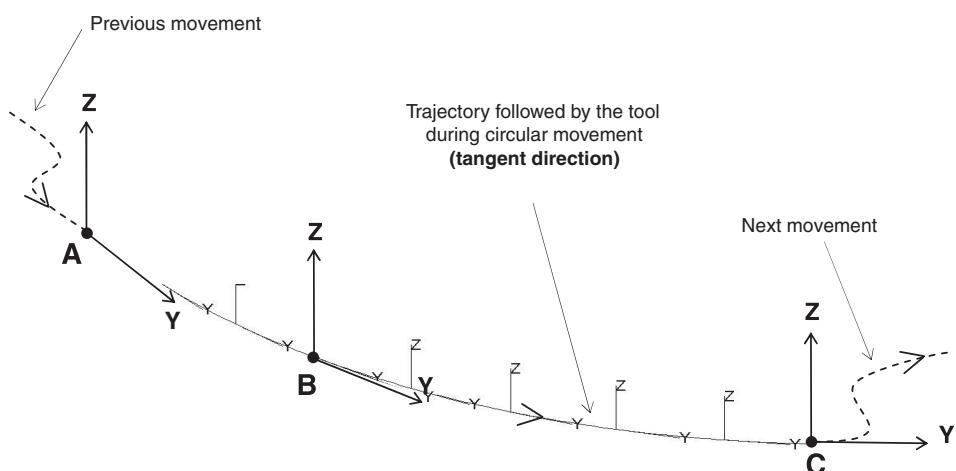
- 作为定向运动，一个圆的起点、终点和中间位置点必须取向相同。

绝对不变取向



- 对于相对于一条轨迹的定向（如工具坐标系的方向 **Y** 与轨迹相切），一个圆周的起点位置、终点和中间位置必须与轨迹的取向相同。

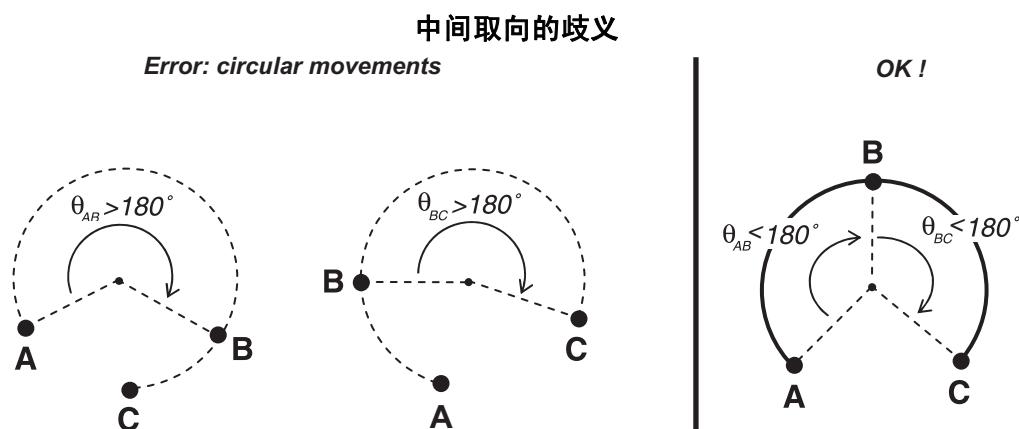
以轨迹为参照的定向



由此对圆周运动引起的限制:

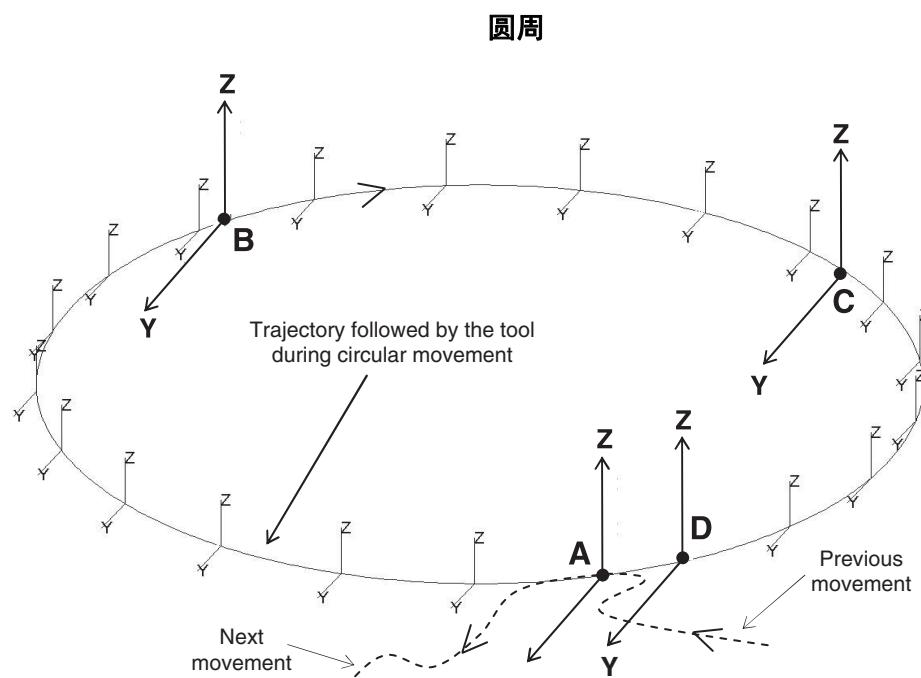
如果中间点和起始点或和终点形成 **180°** 或更大的角度, 将会有多个取向插值, 从而产生错误。

因此必须修改中间点的位置, 来删除中间取向的歧义。



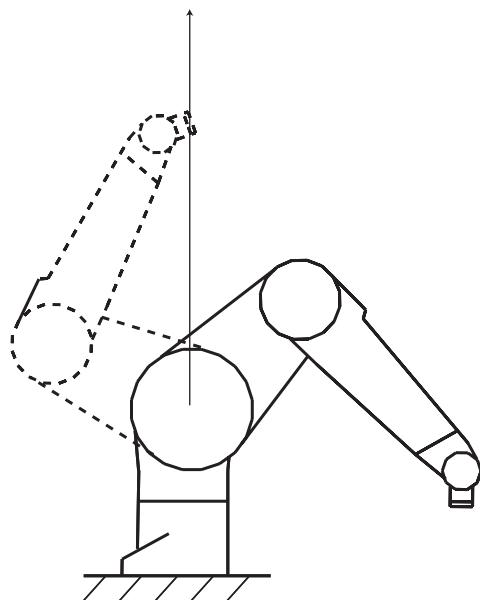
特别是, 在编制一个完整圆周的程序时要用到 **2** 个 movec 指令:

```
movec (B, C, tTool, mDesc)
movec (D, A, tTool, mDesc)
```



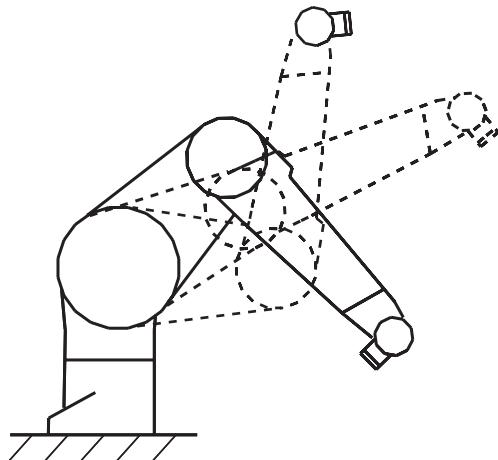
10.1.4.2. 姿态变化 (RX/TX 机器人手臂)

姿态变化: righty / lefty



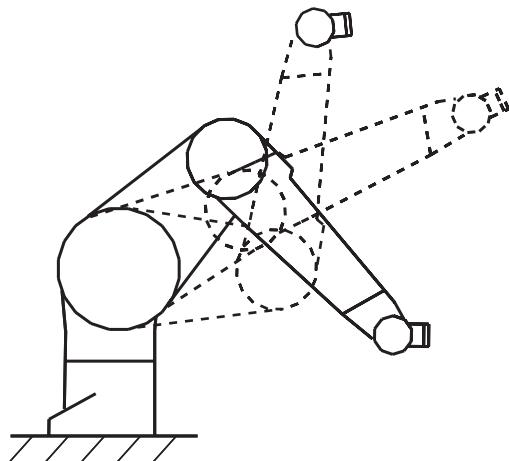
在肩部姿态改变时，机器人手腕中心必须垂直通过轴 1（但对于有偏心距的机器人并不确切是这样）。

肘关节姿态的正向 / 负向改变



肘关节姿态变化时，机器人手臂必须通过手臂直线的位置 ($j3 = 0^\circ$)。

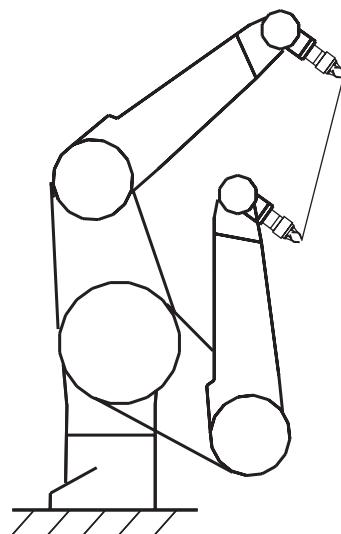
腕关节的正向 / 负向姿态改变



在腕关节姿态变化过程中，机器人手臂必须通过直线腕位置 ($j5 = 0^\circ$)。

因此，在姿态变化过程中，机器人必须通过特定的位置。但是，如果这些位置不在所希望的轨迹上，我们就不能命令一条直线或一个圆周运动通过这些位置。这就意味着**我们不能在直线或圆周运动中强加一个姿态变化**。

无法实现的肘部姿态改变

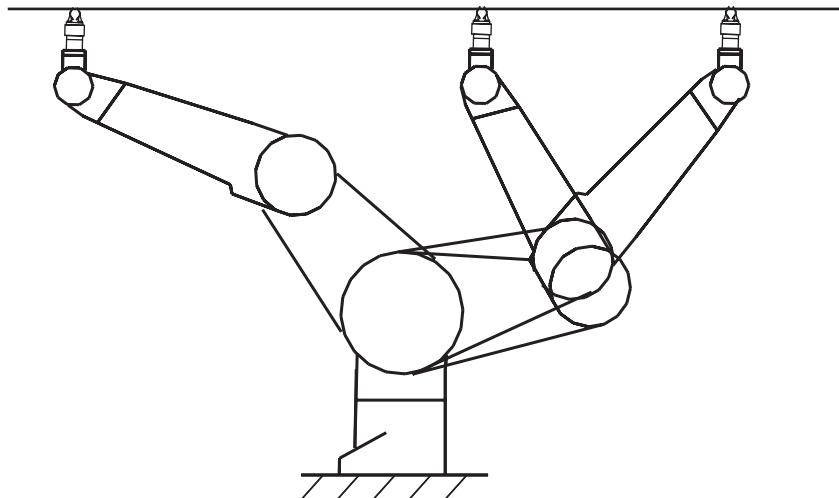


换句话说，在直线或圆周运动中，只有在与初始位置不冲突时才能强加一个姿态：因此，总是可以指定一个自由的或者一个与初始姿态完全一样的姿态。

在某些特定的情况下，直线或圆弧确实通过了一个可以发生姿态变化的位置。在这种情况下，如果姿态是自由状态，系统可以在直线和圆周运动过程中决定改变姿态。

对于圆周运动而言，不须考虑中间点的姿态。唯独需要考虑的是起始和终点位置的姿态。

可以实现的肩部姿态改变



10.1.4.3. 奇点 (RX/TX 机器人手臂)

奇点是所有 **6** 轴机器人的一种固有的特征。奇点可以被定义为机器人在该点改变姿态的点。因而一些轴相互对齐：两根对齐的轴线就如一根轴线那样运动，因此 **6** 轴机器人的局部运动象 **5** 轴机器人一样。因而末端执行器就无法执行某些运动。这在点到点运动中并不会成为一个难题：由系统产生运动总是可能的。另一方面，在直线或圆周运动中，我们强制给予运动的几何方式。如果运动不能实现，当机器人尝试运动时就产生错误。

10.2. 运动预处理

10.2.1. 原理

机器人运动的控制系统有点类似开汽车的驾驶员。此系统使机器人的速度适应轨迹的几何形状。因此越早知道轨迹，系统越能更好地优化运动速度。这就是为什么系统不会等到机器人当前运动完成后，再来计算接下来的行动指令。

考虑下面的几行程序：

```
movej (pA, tTool, mDesc)
movej (pB, tTool, mDesc)
movej (pC, tTool, mDesc)
movej (pD, tTool, mDesc)
```

假设程序执行到达这些程序行时，机器人停止运动。当第一项指令被执行时，机器人开始向 **pA** 点运动。远在机器人到达 **pA** 点之前，程序就立刻执行第二行。

当系统执行第二行时，机器人开始朝 **pA** 运动，系统进行记录，在 **pA** 点之后机器人必须到点 **pB**。程序继续执行下一行指令：当机器人正继续朝 **pA** 运动时，系统在 **pB** 后记录指令，机器人必须朝 **pC** 运动。由于程序的执行比机器人的实际运动要快很多，机器人可能在下一行被执行时仍然向 **pA** 运动。系统就是这样纪录下一个接续的点。

当机器人开始朝 **pA** 点运动时，它已经 " 知道 "，它在 **pA** 之后必须依次朝 **pB**、**pC** 和 **pD** 点运动。假如混合被激活，系统知道机器人在到达 **pD** 点之前不会停止。与假如它停止在 **pB** 或 **pC** 相比，它的速度因此可以加快许多。

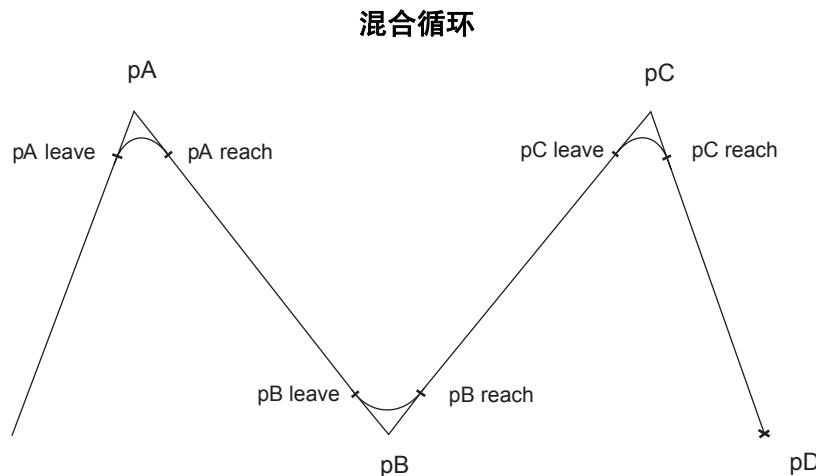
执行指令行仅记录连续运动的命令。机器人随后根据其能力执行命令。保存运动指令的内存要大，使得系统能优化运行轨迹。然而内存是有限的。当内存满的时候，程序在下一条运动指令处停止执行。当机器人完成当前运动后，程序继续执行，由此将系统中的内存空出来。

10.2.2. 预测和轨迹混合

此章节详细讨论运动被序列化时所发生的情况。再来看上述示例：

```
movej (pA, tTool, mDesc)
movej (pB, tTool, mDesc)
movej (pC, tTool, mDesc)
movej (pD, tTool, mDesc)
```

假设混合是在运动描述符 **mDesc** 中被激活的。当第一行被执行时，系统还不知道下续的运动是什么。只有在起始点和 **pA leave** 点之间的动作是确定的，**pA leave** 点是由系统通过运动描述符 **leave** 的数据来确定（见下图）。



只要第二行指令还未执行，那么 **pA** 点附近的混合轨迹段就不完全确定，因为系统还没有计算下一步的运动。在单步模式下，机器人就仅运动到 **pA leave** 点。当下一条指令被执行，**pA** 点（在 **pA leave** 和 **pA reach** 之间）附近的混合轨迹是可以确定的，以及直至 **pB leave** 点的运动也可以确定。这样机器人就可以一直前进到 **pB leave** 点。在单步模式下，只要用户还没有执行第三条指令，机器人就不会越过该点，依次类推。

该运行方式的好处是，在单步模式下和在程序的正常执行的情况下机器人经过的位置完全相同。

10.2.3. 同步

预测机理引起 **VAL 3** 指令行和相应的机器人运动之间的不同步：**VAL 3** 程序超前于机器人。

当需要在机器人的一个给定位置执行一个动作时，程序必须等到机器人完成它的动作之后才执行：**waitEndMove()** 指令用于这个同步。另一种方法是使用 **getMoveld()** 指令来检测手臂在轨迹上的运动（见 10.4，实时运动控制）。

因此，在下面的程序中：

```
movej (A, tTool, mDesc)
movej (B, tTool, mDesc)
waitEndMove ()
movej (C, tTool, mDesc)
movej (D, tTool, mDesc)
```

等等

当机器人就开始朝 **A** 点运动时，前两行指令已经执行完毕。程序于是被锁止在第三行指令，直到机器人稳定在 **B** 点。当机器人运动稳定在 **B** 点时，程序继续执行。

在激活工具前，**open()** 和 **close()** 指令同样要等待机器人完成它的运动。

10.3. 速度监控

10.3.1. 原理

沿一条轨迹的速度监控原理如下：

根据运动指令规定的速度和加速限制，机器人不断地运动和加速到它的最大能力。

运动命令含有两类在 **mdesc** 类型变量中定义的速度限制：

1. 速度 (关节速度)，加速和减速限制
2. 中心工具的笛卡尔速度限制

加速度决定在轨迹始端速度增加的速率。相反，减速度决定在轨迹末端的减速率。当使用高加速度和减速度时，虽然运动更快，但不稳定。当使用低值时，运动持续时间长一些，但更缓和。

10.3.2. 简单调节

当工具和机器人所搬运的物体不需要特别仔细操作时，对笛卡儿速度的限制就没必要。沿轨迹的速度通常调节如下：

1. 将笛卡尔速度限制设成很大的值，比如默认值，以确保这些值不影响其余的调节程序。
2. 用额定值 (**100%**) 初始化速度、加速度和减速度。
3. 用速度参数调节沿轨迹的速度。

为保持和谐的机器人手臂运动，加速和减速应该和速度一起被修改：加速度与减速度参数应该约等于速度参数的平方。例如， $120\% = 1.2$ 的速度更适合于 $1.2 \times 1.2 = 1.44 = 144\%$ 的加速度和减速度。高加减速度值可以提高速度值，但是同时也会引起手臂的振动。

10.3.3. 高级调节

为了控制工具的笛卡尔速度，比如以恒速执行完成一条轨迹，按如下方法进行：

1. 将笛卡尔速度限制设为所希望的值。
2. 用额定值 (**100%**) 初始化速度、加速度和减速度。
3. 然后仅使用笛卡尔速度参数来调节沿轨迹的运行速度。
4. 若速度不够，增大加速度和减速度参数。
若希望在明显弯曲部分自动减速，减小加速和减速参数。

10.3.4. 工作范围错误

关节速度和加速的额定值是机器人的额定负载值，与轨迹无关。

不过，机械人会经常运行得更快：机器人所能达到得最大速度取决于它的负载和运行轨迹。在适当的情况下（轻负载，正向重力作用）机器人能超过额定值而不会造成任何损害。

如果机器人携带一个比额定负载重的负载，或者如果关节速度和加速度值太高，机器人可能不总是按运动命令运动而产生一个工作范围错误。这些错误可以用减小速度和加速度参数的方法避免。

注意：

在一个奇点附近的直线运动情况下，一个小工具的运动需要大的关节运动。如果速度设得太高，机器人可能不服从命令并停止运动从而产生一个工作范围错误。

10.4. 实时运动控制

在该手册中前面章节所描述的运动指令没有即使生效：当执行每个命令时，一个运动命令就被保存在系统中。机器人接下去执行被记录的运动。

可瞬间控制机器人的运动，方法如下：

- 显示器的速度改变所有运动速度。可以用 **setMonitorSpeed()** 指令的瞬时效应来进行调节。不过，当操作者也可以调节 MCP 的速度时，该指令不可能增加速度。
- **stopMove()** 和 **restartMove()** 指令用于在轨迹上的运动的停止和继续。
- **resetMotion()** 指令用于停止正在进行中的动作和取消已经保存的动作命令。
- Alter 指令（选用）将一个立即生效的几何变换（平移，旋转，工具中心点的旋转）应用于路径。
- 它可以用 **getMoveld()** 指令精确地，并实时跟踪机器人在其路径上的位置。每个运动指令由该指令返回的一个数值来识别。**getMoveld()** 指令返回一个识别当前运动（整数部分）和该运动进展（小数部分）的数值。例如，一个 17.572 的运动表示当前运动就是返回 17 的运动指令，和机器人位于已经完成了该运动的 57.2 % 的位置。

10.5. MDESC 类型

10.5.1. 定义

mdesc 类型用于定义运动参数（速度、加速度、混合）。

mdesc 类型属于结构类型，结构类型的字段按顺序为：

num accel	最大允许关节加速度，以机器人的额定加速度的 % 表示。
num vel	最大允许关节速度，以机器人的额定速度的 % 表示。
num decel	最大允许关节减速度，以机器人的额定减速度的 % 表示。
num tvel	工具中心点的最大允许平移速度，根据软件应用单位长度用毫米 / 秒或英寸 / 秒表示。
num rvel	工具中心点的最大允许旋转速度，用度 / 秒表示。
blend blend	混合模式： off (无混合)， joint 或 Cartesian (混合)。
num leave	在混合模式 joint 和 Cartesian 中，在混合开始的目标点和下一个目标点之间的距离，根据软件应用的长度单位用毫米或英寸表示。
num reach	在混合模式 joint 和 Cartesian 中，在混合停止的目标点和下一个目标点之间的距离，根据软件应用的长度单位用毫米或英寸表示。

在“运动控制”章节的章首已经对这些参数作了详细阐述。

默认时，一个 **mdesc** 类型变量用 {100,100,100,9999,9999,joint,50,50} 来初始化。

10.5.2. 运算符

按递增优先顺序：

mdesc <mdesc& desc1> = <mdesc desc2>	将 desc2 的每个字段赋予 desc1 变量的对应字段。
bool <mdesc desc1> != <mdesc desc2>	如果 desc1 和 desc2 至少有一个域的值不同，就返回 true 。
bool <mdesc desc1> == <mdesc desc2>	如果 desc1 和 desc2 的字段值都相同，就返回 true 。

10.6. 运动指令

num movej(joint jPosition, tool tTool, mdesc mDesc)

num movej(point pPosition, tool tTool, mdesc mDesc)

功能

该指令使用 **tTool** 和 **mDesc** 运动参数来记录一个到 **pPosition** 或 **jPosition** 位置的关节运动的命令。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一。

注意:

系统不等待运动结束就接着执行下一个 VAL 3 指令：一些运动命令可以提前保存。当系统没有可用的内存来保存新的命令时，指令等待直至可以存入新命令。

在“运动控制”章节的章首已经对这些运动参数作了详细阐述。

如果 **mDesc** 包含无效值，或 **jPosition** 位置超出软件限位，或 **pPosition** 位置无法到达，或前一个保存的运动指令无法执行（目标位置超出工作范围），将会产生一个运行时错误。

另见

num movel(point pPosition, tool tTool, mdesc mDesc)
bool isInRange(joint jPosition)
void waitEndMove()
num movec(point plntermediate, point pTarget, tool tTool, mdesc mDesc)

num movel(point pPosition, tool tTool, mdesc mDesc)

功能

该指令使用 **tTool** 工具和 **mDesc** 运动参数来记录一个到 **pPosition** 点的直线运动的命令。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一。

注意:

系统不等待运动结束就接着执行下一个 VAL 3 指令：一些运动命令可以提前保存。当系统没有可用的内存来保存新的命令时，指令等待直至可以存入新命令。

在“运动控制”章节的章首已经对这些运动参数作了详细阐述。

如果 **mDesc** 包含无效参数，或 **pPosition** 无法到达，或一个朝 **pPosition** 的直线运动不可能实现，或如果先前保存的一个运动命令不能执行（目标位置超出限位），则会产生一个运行时错误。

另见

num movej(joint jPosition, tool tTool, mdesc mDesc)
void waitEndMove()
num movec(point plnterminate, point pTarget, tool tTool, mdesc mDesc)

```
num movec(point pIntermediate, point pTarget, tool tTool,
          mdesc mDesc)
```

功能

该指令纪录一个圆周运动的命令，该圆周运动从前一个运动的目的地位置开始，在 **point pTarget** 处结束，并经过 **point pIntermediate**。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一。

工具的方向以内插法进行，使得可以编制一个绝对不变方向或者相对于轨迹的不变方向。

注意：

系统不等待运动结束就接着执行下一个 **VAL 3** 指令：一些运动命令可以提前保存。当系统没有可用的内存来保存新的命令时，指令等待直至可以存入新命令。

关于各类运动参数和方向内插法在 "运动控制" 章的开头已给予详细的解释。

如果 **mDesc** 有无效值，如果 **point pIntermediate** (或 **point pTarget**) 位置无法到达，如果圆周运动无法执行 (见 "运动控制 - 方向插值" 章节)，或保存的前一个运动命令无法执行 (目标位置超出工作范围) 时，则会产生一个运行时错误。

另见

num movej(joint jPosition, tool tTool, mdesc mDesc)
num movel(point pPosition, tool tTool, mdesc mDesc)
void waitEndMove()

void stopMove()

功能

该指令使手臂在轨迹上停止运动，并暂停程序编制运动的执行。

注意:

该指令立即返回：VAL 3 任务不等待手臂运动完成就执行下一个指令。

用来执行停止运动的运动描述符就是那些用于当前运动的描述符。

运动只能在 **restartMove()** 或 **resetMotion()** 指令之后继续进行。

没有编制程序的运动（手动移动）仍是可能的。

例如

```
// 等待信号
wait(diSignal==true)
// 停止沿轨迹的运动
stopMove()
wait(diSignal==false)
// 重新开始沿轨迹的运动
restartMove()
```

另见

void restartMove()
void resetMotion(), void resetMotion(joint jStartingPoint)

void resetMotion(), void resetMotion(joint jStartingPoint)

功能

该指令停止机器人手臂在轨迹上的运动，并取消所有已经保存的运动命令。它将运动标识符重置为零。

注意:

该指令立即返回：VAL 3 任务不等待手臂运动完成就执行下一个指令。

如果编程的运动准许是被 **stopMove()** 指令中止的，此准许被恢复。

如果指定 **jStartingPoint** 关节位置，下一个运动命令只能从这个位置开始执行：必须事先执行一个连接运动，以便到达 **jStartingPoint** 位置。

如果没有指定任何关节点，无论机器人手臂的当前位置在什么地方，下个运动指令将在该位置开始执行。

另见

bool isEmpty()
void stopMove()
void autoConnectMove(bool bActive), bool autoConnectMove()
num setMovId(num nMovId)
joint resetTurn(joint jReference)

void restartMove()

功能

该指令恢复所编制的运动准许，并继续被 [stopMove\(\)](#) 指令中断的轨迹。

如果编制的运动许可没有被指令 [stopMove\(\)](#) 中断，此指令就没有任何作用。

另见

[void stopMove\(\)](#)

[void resetMotion\(\), void resetMotion\(joint jStartingPoint\)](#)

void waitEndMove()

功能

该指令取消保存的最后运动指令的混合，并等待该命令被执行。

该指令不等待机器人稳定在它的最终位置上，它只等待传送给变速器的位置指令与所希望的最终位置相符。在需要等待运动完全稳定的情况下，必须使用 [isSettled\(\)](#) 指令。

如果前面保存的一个运动无法执行（目标位置超出限位）时，则产生一个运行时错误。

例如

（见第 10.2 章）

另见

[bool isSettled\(\)](#)

[bool isEmpty\(\)](#)

[void stopMove\(\)](#)

[void resetMotion\(\), void resetMotion\(joint jStartingPoint\)](#)

bool isEmpty()

功能

如果所有的动作命令已经执行，该指令返回 **true**，如果至少还有一个命令要执行则返回 **false**。

例如

该程序取消保存的运动，如果有的话：

```
// 假如还有命令在运行当中
if isEmpty() ==false
    // 停止机器人的运动并取消命令
    resetMotion()
    putln("Movements have been cancelled")
endif
```

另见

void waitEndMove()
void resetMotion(), void resetMotion(joint jStartingPoint)

bool isSettled()

功能

如果机器人停止运动，该指令返回 **true**；如果它的位置还没有稳定，则返回 **false**。

注意:

机器人可能因不同原因停止运行，并因此在完成所有已保存的运动前得以稳定。为了知道机器人是否在其编程运动的结束处停止，使用 **isEmpty()**。

如果每个关节的位置误差在 50 ms 期间小于最大允许位置误差 1% 时，就认为位置已经稳定。

另见

bool isEmpty()
void waitEndMove()

void autoConnectMove(bool bActive), bool autoConnectMove()

功能

在远程模式下，如果手臂非常接近它的轨迹的话（距离小于最大允许漂移误差），连接运动是自动的。如果机器人手臂离它的轨迹太远的话，连接运动是自动的或者手动控制，取决于由 **autoConnectMove** 指令定义的模式：如果 **bActive** 等于 **true** 为自动模式，如果 **bActive** 等于 **false** 则为手动控制。在没有参数的情况下调用，**autoConnectMove** 返回当前连接运动的模式。

默认情况下，远程模式下的连接运动为手动控制。

注意:

在使用的正常条件下，当紧急停止时，手臂停在其轨迹上。因此，在远程模式下，机器人手臂能够自动重启，与 **autoConnectMove** 指令定义的连接运动模式无关。

另见

void resetMotion(), void resetMotion(joint jStartingPoint)

num getSpeed(tool tTool)

功能

此指令返回在指定工具 **tTool** 的 **TCPtTool** 处的当前笛卡尔平移速度。此速度是从关节的速度命令计算而得，而不是从关节速度反馈获得。

另见

point here(tool tTool, frame fReference)

joint getPositionErr()

功能

这个指令返回机器人手臂的当前关节位置错误。关节位置错误是发送给马达的关节位置命令与编码器测量的关节位置反馈之间的偏差。

另见

void getJointForce(num& nForce)

void getJointForce(num& nForce)

功能

这条指令返回从马达电流大小计算得出的当前扭矩 (旋转轴单位 N.m) 或力 (线性轴单位 N)。

关节力不仅仅是外部作用力的直接估算。此力还包括重力、摩擦力、黏度、惯性、噪音、电流传感器精度、马达电流与扭矩之间的关系。只有在参考条件下记录下这些力，并将这些力与在相似的条件下，加上另外的外力而测出的力相比较时，才能用它来估算外力

它仅仅返回力的一个数量级。它的精确度不能确保，对于每个应用，必须计算其精确度。

如果参数不是一个足够大小的数字数组，则产生一个运行时错误。

另见

joint getPositionErr()

num getMovId()

功能

此指令返回一个数值，该数值给出机器人手臂在路径上的当前位置。整数部分识别正在被执行的运动指令号。当运动指令被执行时，返回该整数。小数部分给出有关该运动执行的进展百分比 %。

一个运动标识符决不应该用 '==' 运算符来进行测试，但可以用 '>=' 运算符: **wait(getMovId() == 12)** 可能不再返回，因为运动标识符可以从 11.998 直接增加到 12.013，并且决不会确切地取所希望的值 (12)。因此您最好写入 **wait(getMovId() >= 12)**。

例如

该示例给出运动标识符在一条简单路径上改变的方法:

```
nIdA = movel(pA, tTool, mDesc)
nIdB = movel(pB, tTool, mDesc)
waitEndMove()
nId = getMovId()
```

在该程序执行期间:

假设返回值 **nIdA** 是 15。那么 **nIdB** 是 16：运动随每个运动指令自动增加 1。

- 当 **getMovId()** 为 15.8 时，机器人位置在运动到点 **pA** 的 80 %。
- 当 **getMovId()** 为 16.572 时，机器人位置在运动到点 **pB** 的 57.2 %。

- 当 `getMovId()` 为 17，机器人位置在运动 16 的 100 %，因此它就在点 pB。

nld 的值，在 `waitEndMove()` 之后，所以就是 $nldB+1=17$ 。

另见

`num movej(joint jPosition, tool tTool, mdesc mDesc)`
`num movel(point pPosition, tool tTool, mdesc mDesc)`
`num movec(point plIntermediate, point pTarget, tool tTool, mdesc mDesc)`

`num setMovId(num nMovId)`

功能

该指令改变接续运动指令的运动标识符。要使得相同路径总是使用同样的运动标识符值是很有用的。在一个 `resetMotion` 之后，运动标识符自动重置为 0。

在使用 `setMovId()` 或 `resetMotion()` 后，一个运动标识符和一个运动指令之间的关系可能会变得不确定：那么，一些记录的运动可能会有相同的运动标识符。因此不应该赋予 `setMovId()` 这样一个值，就是该值也是一个暂挂运动命令的运动标识符的值。

例如

```
resetMotion()
nId1 = getMovId()
setMovId(1000)
nId2 = getMovId()
nId3 = movel(pA, tTool, mDesc)
nId4 = movel(pB, tTool, mDesc)
waitEndMove()
nId5 = getMovId()
```

该程序被执行后，我们得到：

- nld1 为 0，因为运动标识符在 `resetMotion()` 后置为 0
- nld2 is 0: 运动标识符随 `setMovId()` 而改变
- nld3 is 1000: 运动指令返回先前确定的运动标识符，并为下一个运动增加标识符的值
- nld4 is 1001: 先前的运动指令增加了运动标识符
- nld5 is 1002: 在 `waitEndMove()` 之后，运动 1001 的 100 % 在这儿完成，那么运动标识符就是 $1001+1 = 1002$

另见

`num getMovId()`
`void resetMotion(), void resetMotion(joint jStartingPoint)`

章节 11

可选项

11.1. 力控制的柔性运动

11.1.1. 原理

在一个标准运动命令中，机器人按程序设定的加速度和速度朝要求的位置运动。如果机器人手臂无法执行命令，为了到达所要求的位置，要求马达提供额外的力。当命令设定的位置与实际位置之间的相差太大时，就产生系统错误，从而切断机器人手臂的电源。

当机器人接受由命令设定的位置和实际位置之间的一定偏差时，它就被称为“柔性”。控制器可以编程为轨迹柔性，例如，通过控制应用于手臂上的力来与接受一个与编程轨迹相关的延迟或者提前。但是，不允许轨迹的任何偏差。

具体地说，**VAL 3** 的柔性运动可以允许在外力推或拉的作用下使手臂沿轨迹运动，或者通过控制手臂作用于此物体上的应力使得机器人手臂与一个物体接触。

11.1.2. 编程

柔性运动像标准运动一样可用 **movelf()** 和 **movejf()** 指令和一个用来控制机器人手臂施加的力的补充参数进行编程。在柔性运行期间，与标准运动一样，通过运动描述符来施加速度和加速度限制。运动可以沿轨迹朝一个方向或朝反向运动。

可以组合柔性运动或柔性运动和标准运动：一旦到达目标位置后，机器人执行下一个运动命令。**waitEndMove()** 指令用来等待柔性运动结束。

resetMotion() 指令取消所有已编程的运动，不管它是否柔性。在 **resetMotion()** 之后，机器人就不再执行柔性运动。

stopMove() 和 **restartMove()** 指令也应用于柔性运动：

stopMove() 将当前运动速度强置于零。如果是一个柔性运动，此运动因此停止，机器人就不再执行柔性运动直至 **restartMove()** 指令开始运行。

最后，**isCompliant()** 指令用来确保机器人处于柔性模式（例如，在允许一个外力作用到手臂上之前）。

11.1.3. 力的控制

当指定的力参数为零时，手臂处于被动状态，例如，只有在一个外力作用下，手臂才会运动。

当力参数为正时，就如一个外力将手臂朝指令位置推去：手臂自动移动，但是可以被一个加于指令力的外作用力拉回或加速。

当力参数为负时，就如一个外力将手臂朝它的初始位置推回去：因此要使手臂朝指令位置移动，必须提供一个比指令力更大的外力。

力的参数用手臂额定负载的百分比来表示。**100%** 表示手臂施加到命令位置的力，等于额定负载。旋转运动中，**100%** 对应于手臂上所允许的额定转矩。

当手臂达到运动描述符中规定的速度或加速度值时，机器人会尽一切可能抵抗任何速度或加速度的增加。

11.1.4. 限制

柔性运动要求机器人的一个特殊调整，并不是所有机器人都有此选项（咨询您的 **Stäubli** 联系人）。

柔性运动具有下述限制：

- 不可能在柔性运动的起始或结束时使用运动混合：手臂必然会在每个柔性运动的起始和结束处停止。
- 在一个柔性运动时，手臂可以重新退回到它的起始点，但不能超出该点：于是手臂会在起始点上突然停止。
- 手臂上的力参数不能超过 **1000%**。所得到的受力精确度受内部摩擦限制。它主要取决于手臂位置和命令轨迹的位置。
- 长时间的柔性运动需要很大的内存。如果系统没有足够的内存空间来执行整个运动进程，则产生一个运行时错误。

11.1.5. 指令

num movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)

功能

用 **tTool** 工具，**mDesc** 运动参数和 **nForce** 力命令记录一个到 **jPosition** 位置的柔性关节运动命令。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一

nForce 力命令是一个表示手臂受力大小的数值，不能超出 **±1000**。值 100 大约对应于手臂的标称重量。

注意：

系统不等待运动结束就接着执行下一个 **VAL 3** 指令：一些运动命令可以提前保存。当系统没有可用的内存来保存新的命令时，指令等待直至可以存入新命令。

在本章开头，已对不同的运动参数做了详细介绍。

如果 **mDesc** 或 **nForce** 有无效值，或 **jPosition** 超出软件限位，或前一个运动要求轨迹混合，或者前一个记录的运动指令无法执行（目标点超出工作范围），则产生一个运行时错误。

另见

num movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)
bool isCompliant()

num movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)

功能

该指令用 **tTool** 工具, **mDesc** 运动参数和 **nForce** 力命令记录一个到 **pPosition** 位置的直线运动命令。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一。

nForce 力命令是一个表示手臂受力大小的数值，不能超出 **±1000**。值 100 大约对应于手臂的标称重量。

注意:

系统不等待运动结束就接着执行下一个 VAL 3 指令：一些运动命令可以提前保存。当系统没有可用的内存来保存新的命令时，指令等待直至可以存入新命令。

在本章开头，已对不同的运动参数做了详细介绍。

如果 **mDesc** 或 **nForce** 有无效值，或 **pPosition** 无法到达，或到 **pPosition** 的运动无法成一直线，或前一个运动要求轨迹混合，或者前一个记录的运动命令无法执行（目标超出工作范围），则产生一个运行时错误。

另见

num movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)
bool isCompliant()

bool isCompliant()

功能

如果机器人处于柔性模式，该指令返回 **true**，否则返回 **false**。

例如

```
movelf(pPosition, tTool, mDesc, 0)
// 等待机器人真正处于柔性模式
wait(isCompliant())
// 注塑机的喷射命令
diEjection = true
// 等待柔性运动结束
waitEndMove()
// 重启一个标准运动
movej(jjDepart, tTool, mDesc)
```

另见

num movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)
num movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)

11.2. ALTER: 路径实时控制

Alter 笛卡尔

11.2.1. 原理

一个路径的笛卡尔变换允许将一个即刻生效的几何变换应用于该路径(平移、旋转、工具中心点的旋转)。这个特点使得使用一个外部传感器来修改一个标称路径成为可能，例如，为了准确地追踪工件的外形，或者在一个移动的工件上操作。

11.2.2. 编程

编程首先定义标称路径，然后，实时地确定一个与标称路径的偏差。

使用 `alterMoveL()`、`alterMoveJ()` 和 `alterMoveC()` 指令，标称路径被编程为标准运动。一些可变换运动可以相继执行，或者一些可变换运动可以和非可变换运动交替执行。我们可以在两个非可变换运动命令中将一个可变换路径定义为连续可变换运动命令。

变换本身是用 `alter()` 指令进行编程的。可以有不同的 `alter` 模式，取决于要应用的几何变换；模式用 `alterBegin()` 指令定义。最后需要 `alterEnd()` 指令指定如何中止这个变换，或者在标称运动完成之前，则下一个非可变换运动可以在不停止的情况下顺序执行；或者在标称运动完成之后，则仍然可能在当标称运动停止之后，用变换来移动手臂。

其他的运动控制指令在 `alter` 模式中仍然有效。

注意：

`waitEndMove`, `open` 和 `close` 指令等待标称运动结束，而不等待变换运动的结束。在一个 `waitEndMove` 之后，`VAL 3` 的执行可以恢复，即便由于一个变换的改变原因手臂仍正在运动。

11.2.3. 限制

同步，异步：因为 `alter` 命令被立即应用，变换中的改变必须被控制，使得合成的手臂路径保持无中断或噪音：

- 变换中的大的改变只能通过用特殊接近控制来逐步应用。
- 在变换的结束需要一个空变换速度，用一个特定的停止控制逐步获得。

同步命令：控制器每 4 ms 向放大器发送位置和速度命令。因此，变换命令必须与通讯周期同步，以便保持变换速度的控制。这通过使用同步 **VAL 3** 任务来实现(见“任务”章)。同样，如果数据有噪点或者取样周期和控制周期不同步，输入传感器可能首先要进行过滤。

平滑定序：只有在执行 `alterEnd` 时，跟随可变路径的第一不可变运动可以计算出来。因此，如果 `alterEnd` 的执行太接近于可变换运动的末尾，手臂可能在这个点附近降速甚至停止，直到下一个运动计算出来为止。

另外，在 `alterEnd` 执行之后，预先计算下一个运动的能力在可变换路径上添加了一些限制：必须保持相同的设置，确定所有的关节保持在相同的轴转向。如果 `alterEnd` 没有预先执行，那么在运动(可能不发生)期间会产生一个错误。

11.2.4. 安全须知

在任何时间，用户变换可能无效：目标超出工作范围，速度或加速度太高。当系统探测到这样的情况，会产生一个错误，并且手臂在最后有效位置立刻停止。需要重启来恢复运行。

当在运动过程中取消手臂动作 (**hold** 模式，停止请求或紧急制动) 时，一个在标称运动上的停止控制就类似于标准运动的控制。在某个延迟后，**alter** 模式也被自动取消，以保证手臂完全停止。当停止停止条件不存在时，运动可能恢复，并且 **alter** 模式再次自动激活。

11.2.5. 限制

一个空运动(当运动目标在起始位置上)被系统忽略。因此，需要一个非空运动来输入 **alter** 模式。为此，0.001 mm 的运动距离就足够了。

不可能为变换路径指定一个要求的设置；系统总是使用相同的设置。因此，不可能在变换路径中改变手臂的设置(即使使用 **alterMovej** 指令也不行)。

11.2.6. 指令

num alterMovej(joint jPosition, tool tTool, mdesc mDesc)

num alterMovej(point pPosition, tool tTool, mdesc mDesc)

功能

该指令保存一个可变换关节运动命令(关节空间中的一条线)。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一。

参数

jPosition/pPosition 点或者关节表达式，用于定义运动的结束位置。

tTool 工具表达式，定义在运动期间为笛卡尔速度控制而使用的工具中心点。

mDesc **mDesc** 表达式，定义运动的速度控制和轨迹混合参数。

详细

这个指令与 **movej** 指令完全一样，但后者允许运动的模式变换。更多详细信息见 **movej**。

num alterMoveL(point pPosition, tool tTool, mdesc mDesc)

功能

该指令保存一个可变换直线运动命令 (笛卡尔空间中的一条线)。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一。

参数

pPosition	点表达式，定义运动的结束位置。
tTool	工具表达式，定义在运动期间为笛卡尔速度控制而使用的工具中心点。在运动的末尾，工具中心点在指定的目标位置。
mDesc	mdesc 表达式，定义运动的速度控制和轨迹混合参数。

详细

这个指令与 moveL 指令完全一样，但后者允许运动的模式变换。更多详细信息见 moveL。

num alterMoveC(point pIntermediate, point pTarget, tool tTool, mdesc mDesc)

功能

该指令保存一个可修改的圆周运动命令。它返回赋予该运动的运动标识符，并给下一个运动命令的标识符增加一。

参数

pIntermediate	定义圆的中间点的点的表达式
pTarget	点表达式，定义运动的结束位置。
tTool	工具表达式，定义在运动期间为笛卡尔速度控制而使用的工具中心点。在运动的末尾，工具中心点在指定的目标位置。
mDesc	mdesc 表达式，定义运动的速度控制和轨迹混合参数。

详细

这个指令与 moveC 指令完全一样，但后者允许运动的模式变换。更多详细信息见 moveC。

num alterBegin(frame fAlterReference, mdesc mMaxVelocity)

num alterBegin(tool tAlterReference, mdesc mMaxVelocity)

功能

该指令初始化 **alter** 模式，用于正在执行的可变换路径。

参数

fAlterReference/tAlterReference 坐标系或工具表达式，定义用于变换偏差的参考。

mMaxVelocity **mdesc** 表达式，定义变换偏差的安全检查参数。

详细

alter 模式用 **alterBegin** 初始化，只能用 **alterEnd** 命令或 **resetMotion** 命令来终止。当到达可变换路径的末尾时，**alter** 模式保持激活，直到 **alterEnd** 被执行。

alter 命令的 **trsf** 表达式定义了 **alterReference** 附近的整个路径的变换：

- 路径用 **trsf** 的旋转部分绕坐标系或工具中心旋转。
- 然后这个路径用 **trsf** 的平移部分来进行平移。

alter 命令的 **trsf** 坐标系在 **alterReference** 中定义。

当一个坐标系用作参考坐标时，**alterReference** 在空间 (World) 是不变的。这个模式必须用在当路径的偏移已知或者在笛卡尔空间中被测出 (运动部件如传送带的运动)。

当一个工具用作参考时，相对于工具中心点 **alterReference** 不变。在当路径的偏移已知或者相对于工具中心点的偏移已测出时必须使用这个模式 (例如安装在工具上的工件形状传感器)。

动作描述符用于定义变换路径上的最大关节和笛卡尔速度 (使用运动描述符的 **vel**, **tvel** 和 **rvel** 字段)。如果变换速度超出了指定的限制，产生一个错误，并且手臂停止运动。

动作描述符的加速和减速字段在当停止状况发生时控制停止时间 (**eStop**, 保持模式, **VAL 3 stopMove()**)：路径的变换必须使用这些减速参数来停止 (见 **alterStopTime**)。

alterBegin 返回一个数值，用来指明指令的结果：

- | | |
|-----------|---|
| 1 | alterBegin 被成功执行 |
| 0 | alterBegin 正等待可变换运动启动 |
| -1 | alterBegin 被忽略，因为 alter 模式已经启动 |
| -2 | alterBegin 被拒绝 (alter 选项未激活) |
| -3 | alterBegin 被拒绝，因为运动出错。请求一个 resetMotion。 |

另见

num alterEnd()

num alter(trsf trAlteration)

num alterStopTime()

num alterEnd()

功能

该指令退出 `alter` 模式，并使得当前运动变为不再是可变换的。

详细

如果 `alterEnd` 在到达可变换路径的末尾时被执行，下一个非可变换运动（如果存在）将被自动启动。

如果 `alterEnd` 在可变换运动结束之前被执行，`alter` 偏移的当前值被应用到剩余的可变换路径中，直到第一个接续的非可变换运动为止。在同一个可变换路径上，不可能再次输入 `alter` 模式。

下一个非可变换运动，如果有的话，一旦 `alterEnd` 被执行就马上计算出来，所以在可变换路径和下一个非可变换路径之间的转换没有停顿。

`alterEnd` 返回一个数值，用来指明指令的结果：

- 1 `alterEnd` 被成功执行
- 1 `alterEnd` 被忽略，因为 `alter` 模式还未启动
- 3 `alterEnd` 被拒绝，因为运动出错。请求一个 `resetMotion`。

另见

`num alterBegin(frame fAlterReference, mdesc mMaxVelocity)`

`num alterBegin(tool tAlterReference, mdesc mMaxVelocity)`

num alter(`trsF` trAlteration)

功能

该指令指明可变换路径的一个新变换。

详细

`trsF` 变换引起的转换取决于由 `alterBegin` 指令选定的 `alter` 模式。变换坐标在由 `alterBegin` 指令指定的坐标系或工具中所确定。

这个变换由系统每 4 ms 应用一次：当几个 `alter` 指令在少于 4 ms 中被执行时，最后一个则被应用。通常 `alter` 指令需要在一个同步任务中执行，每 4 ms 强迫刷新一次变换。

这个变换必须仔细计算，使得得到的手臂位置和速度命令保持连续无噪音。或许需要适当地过滤一个传感器的输入，以在手臂路径上得到所希望的运动质量和方式。

当停止运动时（保持模式、紧急停止、`stopMove()` 指令），路径的变换被锁定，直到所有的停止条件被清除。

当路径变换无效时（无法到达的位置，超出速度限制），手臂将在最后的有效位置突然停止，变换模式在错误位置被锁定。请求一个 `resetMotion` 来恢复运行。`alter` 运动的速度限制用 `alterBegin` 指令来确定。

Alter 返回一个数值，用来指明指令的结果：

- 1 alter 被成功执行。
- 0 alter 正等待运动重启 (**alterStopTime** 为空)。
- 1 alter 被忽略，因为 alter 模式未启动或已经结束。
- 2 alter 被拒绝 (alter 选项未激活)。
- 3 alter 被拒绝，因为运动出错。请求一个 **resetMotion**。

另见

```
num alterBegin(frame fAlterReference, mdesc mMaxVelocity)
num alterBegin(tool tAlterReference, mdesc mMaxVelocity)
void taskCreateSync string sName, num nPeriod, bool& bOverrun, program(...)
```

num **alterStopTime()**

功能

当一个停止条件出现时，该指令在 **alter** 偏移被锁定前返回剩余时间。

详细

当停止条件出现时，如果用 **alterBegin** 指定的运动描述符的加速和减速参数被使用，系统判断停止手臂的时间。这个时间的最小值和系统强加给的时间（当一个 eStop 发生时，通常为 0.5s）通过 **alterStopTime** 返回。

当 **alterStopTime** 返回一个负值，无暂挂停止条件。当 **alterStopTime** 返回为空，则 **alter** 命令被锁定直到所有的停止状态被重设。

当 **alter** 模式未被激活时，**alterStopTime** 返回空值。

另见

```
num alterBegin(frame fAlterReference, mdesc mMaxVelocity)
num alterBegin(tool tAlterReference, mdesc mMaxVelocity)
num alter(trsf trAlteration)
```

11.3. OEM 许可证管理

11.3.1. 原理

一个 OEM 许可证是一个控制器特定的密钥，用于限制一个 **VAL 3** 项目在一些选定的控制器上的使用。

Stäubli Robotics Suite(*) 提供了一个工具，用于给一个 OEM 密码编码为公用的，控制器专有的 OEM 许可证，然后可以作为一个软件可选项安装在控制器上。使用 **getLicence()** 指令，一个项目或库可以测试是否安装有 OEM 许可证并确定是否只在选定的机器人控制器内使用。

为了在许可证的测试期间保护密码 OEM 和代码，**getLicence()** 指令必须在一个加密的库中使用。

支持 OEM 许可证的演示模式，在这种情况下，控制器设置有一个演示密码，**getLicence()** 指令将其通报给程序调用器。在 **VAL 3** 仿真器中，只需选择 "demo" 模式就可启用 OEM 许可证。

getLicence() 指令是个 **VAL 3** 选配件，需要在控制器安装一个运行许可证。如果没有定义此运行许可证，**getLicence()** 返回一个错误代码。

(*) 这个工具，一个可执行的 encrypttools.exe，要求使用特殊的 **Stäubli Robotics Suite** 许可证。

11.3.2. 指令

string getLicence(string sOemLicenceName, string sOemPassword)

功能

该指令返回规定的 OEM 许可证的状态：

"oemLicenceDisabled"	VAL 3 运行许可证 "oemLicence" 在控制器内不可用：OEM 许可证不能被测试。
""	OEM 运行许可证 "sOemLicenceName" 不可用(未定义，或密码无效)。
"demo"	OEM 运行许可证 "sOemLicenceName" 演示模式可用。
"enabled"	OEM 运行许可证 "sOemLicenceName" 可用。

另见

加密

11.4. 绝对机器人

11.4.1. 原理

一个‘绝对机器人’就是使用几何参数(通常称为‘DH parameters’)的手臂特殊标识的机器人。对于每个机器人这些参数都是特有的，并与各关节的真实方向和尺寸相符。绝对机器人以更高的精确度来到达由**CAD**工具指定的或在**VAL 3**中计算得到的笛卡尔位置(例如码垛中的位置)。笛卡尔曲线(长线条，圆)也更精确。绝对校准不改变手臂的可重复性。

DH 参数由一组平移(沿轴 X, Y, Z 的 a, b 和 d)和旋转(绕轴 X, Y 和 Z 的 alpha, beta, theta)组成

这些平移和旋转序列被定义，使得关节位置 {j1, j2, j3, j4, j5, j6} 与在法兰中心点的笛卡尔位置 pCart 相符：

```
pCart.trsf = {0, 0, 0, 0, 0, j1+theta[0]}
* {a[0], b[0], d[0], alpha[0], beta[0], j2+theta[1]}
* {a[1], b[1], d[1], alpha[1], beta[1], j3+theta[2]}
* {a[2], b[2], d[2], alpha[2], beta[2], j4+theta[3]}
* {a[3], b[3], d[3], alpha[3], beta[3], j5+theta[4]}
* {a[4], b[4], d[4], alpha[4], beta[4], j6+theta[5]}
* {a[5], b[5], d[5], alpha[5], beta[5], 0}
* {0, 0, d[6], 0, 0, 0}
```

仅仅对于特殊手腕要求 d[6] 参数。

11.4.2. 操作

必须使用一个单独的测量工具(如激光跟踪仪)来识别几何参数。**VAL 3** 语言不提供支持这个测量用的工具，但它提供了将测量的几何参数应用于机器人并立即生效的方法。这些参数也被保存在手臂的设置文件中，以便使它们在下一次重新启动时自动恢复。

几何参数可以在一个正在运行的**VAL 3**软件应用中予以改变。这使得可以在生产循环期间调节手臂的几何参数，如果机器人单元包括一个适当的测量工具的话。

11.4.3. 限制

只有当运动发生器为空时(没有挂起的运动)，才可以在一个正在运行的**VAL 3**软件应用中改变几何参数。

绝对机器人的修改的几何参数比标准几何参数具有更复杂的数学特性。因而手臂设置的概念(设置类型)不可能是很严密的了。一个笛卡尔位置变换为一个对应的关节位置可能会在关节限制附近或奇点位置附近失败。

11.4.4. 指令

```
void getDH (num& theta[], num& d[], num& a[], num& alpha[],  
           num& beta[])
```

```
void getDefaultDH(num& theta[], num& d[], num& a[], num& alpha[],  
                  num& beta[])
```

功能

这些指令返回指定数组中手臂的 DH 参数。参数 d 和 a 以 mm 为单位的平移；参数 θ ， α 和 β 是以度为单位的角度。[getDH\(\)](#) 返回与控制器连接的手臂的当前 DH 参数。[getDefaultDH\(\)](#) 返回手臂类型的标准 DH 参数。

另见

[bool setDH\(num& theta\[\], num& d\[\], num& a\[\], num& b\[\], num& alpha\[\], num& beta\[\]\)](#)

```
bool setDH(num& theta[], num& d[], num& a[], num& b[], num& alpha[],  
          num& beta[])
```

功能

只有在具有一个特殊运行时许可证时该指令才可用。它用 DH 参数来修改手臂几何参数。参数 d , a 和 b 是以 mm 为单位的平移；参数 θ , α 和 β 是以度为单位的角度。改变是立即的，也被应用于手臂设置文件，以便改变的几何参数在下一次重新启动时有效。

如果改变被成功应用，该指令将返回 **true**，如果由于新的几何参数与标准几何参数相差太大而没有被应用，则返回 **false**。[setDH\(\)](#) 等待运动为空来完成它的运行。

DH 数组的大小必须与机器人的轴数相符。为了修改法兰尺寸，可能在 d 数组中要有一个额外的输入：当缺少时，[setDH\(\)](#) 返回 **false**，并发送一条诊断信息到记录器。

另见

[void getDH \(num& theta\[\], num& d\[\], num& a\[\], num& alpha\[\], num& beta\[\]\)](#)

[void getDefaultDH\(num& theta\[\], num& d\[\], num& a\[\], num& alpha\[\], num& beta\[\]\)](#)

11.5. 连续轴

11.5.1. 原理

轴 6（对于 RX/TX 手臂）或 4 轴（对于 RS/TS 手臂）在一个机器人应用中可以是‘连续’的，条件是只有当它的位置在一圈中是重要的：在该循环中已经执行的圈数不重要。例如这应用于机器人正抓着一个由一个固定工具处理的工件的应用。

continuousAxis 选项使得可以在一个新循环开始前自动重置先前循环中执行的圈数。例如，如果该轴在位置 0° 和位置 720°（2 圈）末尾开始应用该循环，下个循环可能立即重置位置为 0°，并开始一个新循环，而轴无需从 720° 回到 0°。

11.5.2. 指令

joint resetTurn(joint jReference)

功能

该指令完全像标准 [resetMotion\(\)](#) 指令一样工作，等待手臂停止，并调节连续轴的位置，使得它尽可能接近指定的参考位置。在该指令执行后，它返回有效手臂位置。采用手臂上电或去电来进行调节操作，约需 50 ms。[resetTurn\(\)](#) 指令改变手臂的校准数据。随着下一次控制器重新启动，轴零位置自动重新初始化（更新的手臂，手臂中和控制器磁盘上的手臂特殊数据的更新）。

另见

[void resetMotion\(\)](#), [void resetMotion\(joint jStartingPoint\)](#)

章节 12

附录

12.1. 运行时错误代码

代码	说明
-1	没有由带特殊名的软件应用或库所建立的任务
0	任务暂停, 无运行时错误 (taskSuspend() 指令或调试模式)
1	指定的任务正在运行
10	数字计算无效 (被零除)。
11	无效数字计算 (如 ln(-1))
20	访问一个索引超过此数组大小的数组。
21	访问一个负索引的数组。
29	任务名无效。见 taskCreate() 指令。
30	指定名没有相应的任何 VAL 3 任务。
31	同名的任务已经存在。见 taskCreate 指令。
32	仅支持同步任务的 2 个不同周期。更改调度周期。
40	无足够可用的内存空间。
41	无足够的运行该任务的内存空间。查看运行内存的大小。
60	超过最大指令运行时间。
61	VAL 3 内部解释错误
70	指令参数无效。见相应指令。
80	使用了没有装载到内存中的一个程序库的数据或程序。
81	运动不兼容: 一个 <code>point/joint/config</code> 的使用与手臂运动不兼容。
82	一个变量的参考坐标系或工具属于一个库, 但是这个库不能从变量的作用域进行访问 (库未在此变量的项目中声明, 或者参考变量是专用的)。
90	此任务无法在指定处继续执行。见 taskResume() 指令。
100	运动描述符中指明的速度无效 (负值或太大)。
101	运动描述符中指明的加速值无效 (负值或太大)。
102	运动描述符中指明的减速值无效 (负值或太大)。
103	运动描述符中指明的平移速度无效 (负数或太大)。
104	运动描述符指明的旋转速度无效 (负数或太大)。
105	运动描述符中指明的 <code>reach</code> 参数无效 (负值)。
106	运动描述符中指明的 <code>leave</code> 参数无效 (负值)。
122	试图写入系统输入。
123	使用了一个未与系统输入 / 输出相连的 dio、aio 或 sio 输入 / 输出。
124	试图访问一个受保护的系统输入 / 输出
125	dio、aio 或 sio 上的读取或写入错误 (现场总线错误)
150	无法运行该运动指令: 前一个运动要求未完成 (无法到达该点, 奇点, 姿态问题, 等)
153	不支持的运动命令
154	无效的动作指令: 检查动作的描述符。
160	<code>flange</code> 工具的坐标无效
161	<code>world</code> 工具坐标无效
162	使用了一个无参考坐标系的 <code>point</code> 。见定义。
163	使用无参考坐标系的一个坐标系。见定义。
164	使用无参考工具的一个工具。见定义。
165	坐标系或参考工具无效 (全局变量和一个局部变量链接)
250	此指令无运行时许可, 或者试用许可已经过期。

12.2. 控制面板键盘按键代码

不按 Shift				按 Shift			
3 283	Caps -	Space 32		3 283	Caps -	Space 32	
2 282	Shift -	Esc 255	Help -	Ret. 270	Move -	Run -	Ret. 270
	Menu -	Tab 259	Up 261	Bksp 263	Stop -	Shift -	Run -
1 281	User -	Left 264	Down 266	Right 268		Menu -	UnTab 260
						PgUp 262	Bksp 263
						User -	Home 265
						PgDn 267	End 269

菜单 (带或不带 Shift) :

F1 271	F2 272	F3 273	F4 274	F5 275	F6 276	F7 277	F8 278

对于标准键, 返回的编码是相应字符的 ASCII 码:

不按 Shift									
q 113	w 119	e 101	r 114	t 116	y 121	u 117	i 105	o 111	p 112
a 97	s 115	d 100	f 102	g 103	h 104	j 106	k 107	l 108	< 60
z 122	x 120	c 99	v 118	b 98	n 110	m 109	.	,	= 61

按 Shift									
7 55	8 56	9 57	+	*	; 59	(40) 41	[91] 93
4 52	5 53	6 54	- 45	/ 47	? 63	:	! 58	{ 33	} 123
1 49	2 50	3 51	0 48	" 34	% 37	- 95	.	, 46	> 44

双击 Shift									
Q 81	W 87	E 69	R 82	T 84	Y 89	U 85	I 73	O 79	P 80
A 65	S 83	D 68	F 70	G 71	H 72	J 74	K 75	L 76	} 125
Z 90	X 88	C 67	V 86	B 66	N 78	M 77	\$ 36	\ 92	= 61

插图

中间取向的歧义	149
以轨迹为参照的定向	148
到达一个指定点的两种不同姿态: P	136
参考坐标系之间的关系图	127
可以实现的肩部姿态改变	152
圆周	149
圆周运动	144
在给定点无轨迹混合的循环	146
姿态: enegative	139
姿态: epositive	139
姿态: lefty	138
姿态: lefty	140
姿态: righty	138
姿态: righty	140
姿态: wnegative	139
姿态: wpositive	139
姿态变化: righty / lefty	150
工具之间的连系	129
循环型: U	144
方向	122
无法实现的肘部姿态改变	151
时序	84
混合循环	146
混合循环	153
点定义	132
用户页面	71
直线运动	143
组织图: frame / point / tool / trsf	117
绕轴的坐标系旋转: X 轴	122
绕轴的坐标系旋转: Y' 轴	123
绕轴的坐标系旋转: Z" 轴	123
绝对不变取向	148
肘关节姿态的正向 / 负向改变	150
腕关节的正向 / 负向姿态改变	151
起始和终点位置	143
距离的定义: 'leave' / 'reach'	145

索引

A

abs ((Instruction) 47, 118
accel 156
acos (Instruction) 46
aio 28, 63
aioGet (Instruction) 63
aioLink (Instruction) 63
aioSet (Instruction) 63, 64
alter (Instruction) 174
alterBegin (Instruction) 173
alterEnd (Instruction) 174
alterMovec (Instruction) 172
alterMovej (Instruction) 171
alterMovel (Instruction) 172
alterStopTime (Instruction) 175
append (Instruction) 35
appro (Instruction) 134
asc (Instruction) 57
asin (Instruction) 46
atan (Instruction) 47
autoConnectMove 147
autoConnectMove (Instruction) 161

B

bAnd (Instruction) 50
blend 145, 156
bNot (Instruction) 50
bool 28
bOr (Instruction) 51
bXor (Instruction) 51

C

call 22
call (Instruction) 22
chr (Instruction) 56
clearBuffer (Instruction) 66
clock (Instruction) 94
close 84
close (Instruction) 131
cls (Instruction) 72
codeAscii 56
compose (Instruction) 133
config 28, 117, 136
config (Instruction) 140
cos (Instruction) 46

D

decel 156
delay 84
delay (Instruction) 93
delete (Instruction) 33, 58
dio 28, 59
dioGet (Instruction) 60
dioLink (Instruction) 60
dioSet (Instruction) 61
disablePower (Instruction) 109
distance (Instruction) 124, 133
do ... until (Instruction) 24

E

elbow 136
enablePower (Instruction) 109
enegative 139
epositive 139
esStatus (Instruction) 111
exp (Instruction) 47

F

find (Instruction) 58
first (Instruction) 36
for (Instruction) 25
frame 28, 117
fromBinary (Instruction) 52

G

get 84
get (Instruction) 74
getData (Instruction) 34
getDate 79
getDefaultDH (Instruction) 178
getDH (Instruction) 178
getDisplayLen (Instruction) 73
getJointForce (Instruction) 162
getKey (Instruction) 76
getLanguage (Instruction) 78
getLatch (Instruction) 120
getLicence (Instruction) 176
getMonitorSpeed (Instruction) 112
getMoveld (Instruction) 162
getPosition (Instruction) 162
getProfile (Instruction) 77
getSpeed (Instruction) 162
getVersion (Instruction) 113
globale 30
gotoxy (Instruction) 72

H

help (Instruction) 89
here (Instruction) 134
herej (Instruction) 119

I

if (Instruction) 23
insert (Instruction) 32, 58
interpolateC (Instruction) 126
interpolateL (Instruction) 125
ioBusStatus (Instruction) 111
ioStatus (Instruction) 61, 62, 64
isCalibrated (Instruction) 110
isCompliant 167
isCompliant (Instruction) 169
isDefined (Instruction) 31
isEmpty (Instruction) 161
isInRange (Instruction) 119
isKeyPressed (Instruction) 76
isPowered (Instruction) 109
isSettled (Instruction) 161

J

joint 28
jointToPoint (Instruction) 134

L

last (Instruction) 36
leave 145, 156
left (Instruction) 57
lefty 138, 140
len (Instruction) 58
libDelete (Instruction) 101
libList (Instruction) 102
libLoad 100
libLoad (Instruction) 101
libPath (Instruction) 102
libSave (Instruction) 101
limit (Instruction) 49
link (Instruction) 128, 131
In (Instruction) 48
locale 30
log (Instruction) 48
logMsg (Instruction) 77

M

max (Instruction) 49
mdesc 28, 143, 156
mid (Instruction) 57
min (Instruction) 49
movec (Instruction) 158
movej 143
movej (Instruction) 157
movejf 167
movejf (Instruction) 168
movel 143
movel (Instruction) 157
movelf 167
movelf (Instruction) 169

N

next (Instruction) 36
num 28, 57

O

open 84
open (Instruction) 130

P

point 28
pointToJoint (Instruction) 135
popUpMsg (Instruction) 76
position (Instruction) 128, 131, 135
power (Instruction) 47
prev (Instruction) 36
put (Instruction) 74
putln (foncion) 74

R

reach 145, 156
replace (Instruction) 58
resetMotion 147, 159, 167
resetMotion (Instruction) 159
resetTurn (Instruction) 179
resize (Instruction) 35
restartMove 159, 167
restartMove (Instruction) 160
return (Instruction) 22
right (Instruction) 57
righty 138, 140
round (Instruction) 48
roundDown (Instruction) 48
roundUp (Instruction) 48
RUNNING 93
rvel 156

S

safetyFault (Instruction) 111
sel (Instruction) 49
setDH (Instruction) 178
setFrame (Instruction) 128
setLanguage (Instruction) 79
setLatch (Instruction) 120
setMonitorSpeed (Instruction) 112
setMoveld (Instruction) 163
setMutex (Instruction) 89
setProfile (Instruction) 77
setTextMode (Instruction) 72
shoulder 136
sin (Instruction) 46
sio 28, 65
SioCtrl (Instruction) 67
sioGet (Instruction) 66
sioLink (Instruction) 66
sioSet (Instruction) 66
size (Instruction) 31, 35

sqrt (Instruction) 47
stopMove 167
stopMove (Instruction) 159
STOPPED 88
string 28
switch (Instruction) 23, 26

T

tan (Instruction) 46
taskCreate (Instruction) 91
taskCreateSync (Instruction) 92
taskKill (Instruction) 89
taskResume 83
taskResume (Instruction) 88
taskStatus 83
taskStatus (Instruction) 90
taskSuspend (Instruction) 88
title (Instruction) 74
toBinary (Instruction) 52
toNum (Instruction) 55
tool 28, 117
toString (Instruction) 54
trsf 28, 117
trsf align (Instruction) 126
tvel 156

U

userPage (Instruction) 72

V

vel 156

W

wait 84
wait (Instruction) 93
waitForEndMove 84, 146, 167
waitForEndMove (Instruction) 160
watch 84
watch (Instruction) 94
while (Instruction) 24
wnegative 139
workingMode (Instruction) 110
wpositive 139
wrist 136

