

VAL 3 REFERENCE MANUAL

Version 7

Documentation addenda and errata can be found in the "readme.pdf" document delivered with the controller's CdRom.

TABLE OF CONTENTS

1 - INTRODUCTION.....	13
2 - VAL 3 LANGUAGE ELEMENTS.....	17
2.1 APPLICATIONS.....	19
2.1.1 Definition	19
2.1.2 Default content	19
2.1.3 Start/stop	19
2.1.4 Application parameters.....	19
2.1.4.1 Unit of length.....	20
2.1.4.2 Amount of stack memory	20
2.2 PROGRAMS.....	21
2.2.1 Definition	21
2.2.2 Re-entry.....	21
2.2.3 Start() program	21
2.2.4 Stop() program	21
2.2.5 Program control instructions.....	22
Comment //	22
call program	22
return	22
if control instruction	23
while control instruction	24
do ... until control instruction	24
for control instruction	25
switch control instruction	26
2.3 DATA.....	28
2.3.1 Definition	28
2.3.2 Simple types.....	28
2.3.3 Structured types	28
2.3.4 Containers	29
2.4 DATA INITIALIZATION.....	29
2.4.1 Simple type data.....	29
2.4.2 Structured type data	29
2.5 VARIABLES.....	30
2.5.1 Definition	30
2.5.2 Variable scope.....	30
2.5.3 Accessing a variable value.....	30
2.5.4 Instructions applying to all variables.....	31
num size (*)	31
bool isDefined (*)	31
bool insert (*)	32
bool delete (*)	33
num getData (string sDataName, *)	33

2.5.5	Instructions specific to array variables	35
	void append (*)	35
	num size (*, num nDimension)	35
	void resize (*, num nDimension, num nSize)	35
2.5.6	Instructions specific to collection variables	36
	string first (*)	36
	string next (*)	36
	string last (*)	36
	string prev (*)	36
2.6	PROGRAM PARAMETERS	37
2.6.1	Parameter by element value	38
2.6.2	Parameter by element reference.....	38
2.6.3	Parameter by array or collection reference	39
3	- SIMPLE TYPES	41
3.1	BOOL TYPE	43
3.1.1	Definition	43
3.1.2	Operators	43
3.2	NUM TYPE	44
3.2.1	Definition	44
3.2.2	Operators	45
3.2.3	Instructions.....	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	BIT FIELD TYPE	50
3.3.1	Definition	50
3.3.2	Operators	50
3.3.3	Instructions.....	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 TYPE	54
3.4.1	Definition	54
3.4.2	Operators	54
3.4.3	Instructions	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 TYPE	59
3.5.1	Definition	59
3.5.2	Operators	59
3.5.3	Instructions	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 TYPE	63
3.6.1	Definition	63
3.6.2	Instructions	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 TYPE	65
3.7.1	Definition	65
3.7.2	Operators	65
3.7.3	Instructions	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	- USER INTERFACE	69
4.1	USER PAGE	71
4.2	SCREEN TYPE	71
4.2.1	Selecting the user screen	71
4.2.2	Writing to a user screen.....	71
4.2.3	Reading from a user screen	71

4.3	INSTRUCTIONS	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	- TASKS	81
5.1	DEFINITION	83
5.2	RESUMING AFTER A RUNTIME ERROR	83
5.3	VISIBILITY	83
5.4	SEQUENCING	84
5.5	SYNCHRONOUS TASKS	85
5.6	OVERRUN	85
5.7	INPUTS / OUTPUTS REFRESH	85
5.8	SYNCHRONIZATION	86
5.9	SHARING RESOURCES	87
5.10	INSTRUCTIONS	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 - LIBRARIES	95
6.1 DEFINITION	97
6.2 INTERFACE	97
6.3 INTERFACE IDENTIFIER	97
6.4 CONTENT	97
6.5 ENCRYPTION	98
6.6 LOADING AND UNLOADING.....	99
6.7 INSTRUCTIONS.....	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 - USER TYPE	103
7.1 DEFINITION	105
7.2 CREATION.....	105
7.3 USE	105
8 - ROBOT CONTROL	107
8.1 INSTRUCTIONS.....	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 - ARM POSITIONS	115
9.1 INTRODUCTION	117
9.2 JOINT TYPE.....	117
9.2.1 Definition	117
9.2.2 Operators	118
9.2.3 Instructions	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 TYPE	121
9.3.1	Definition	121
9.3.2	Orientation.....	122
9.3.3	Operators	124
9.3.4	Instructions	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 TYPE	127
9.4.1	Definition	127
9.4.2	Use.....	127
9.4.3	Operators	128
9.4.4	Instructions	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 TYPE	129
9.5.1	Definition	129
9.5.2	Use.....	129
9.5.3	Operators	130
9.5.4	Instructions	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 TYPE	132
9.6.1	Definition	132
9.6.2	Operators	132
9.6.3	Instructions	133
	num distance (point pPosition1, point pPosition2)	133
	point compose (point pPosition, frame fReference, trsf trTransformation)	133
	point apro (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 TYPE	136
9.7.1	Introduction	136
9.7.2	Definition	136
9.7.3	Operators	137
9.7.4	Configuration (RX/TX arm).....	138
	9.7.4.1 Shoulder configuration.....	138
	9.7.4.2 Elbow configuration.....	139
	9.7.4.3 Wrist configuration	139
9.7.5	Configuration (RS/TS arm).....	140
9.7.6	Instructions	140
	config config (joint jPosition)	140

10 - MOVEMENT CONTROL	141
10.1 TRAJECTORY CONTROL.....	143
10.1.1 Types of movement: point-to-point, straight line, circle	143
10.1.2 Movement sequencing: Blending	145
10.1.2.1 Blending	145
10.1.2.2 Cancel blending	146
10.1.2.3 Joint blending, Cartesian blending	146
10.1.3 Movement resumption	147
10.1.4 Particularities of Cartesian movements (straight line, circle).....	148
10.1.4.1 Interpolation of the orientation.....	148
10.1.4.2 Configuration change (Arm RX/TX)	150
10.1.4.3 Singularities (Arm RX/TX)	152
10.2 MOVEMENT ANTICIPATION	152
10.2.1 Principle.....	152
10.2.2 Anticipation and blending	153
10.2.3 Synchronization	153
10.3 SPEED MONITORING	154
10.3.1 Principle.....	154
10.3.2 Simple settings	154
10.3.3 Advanced settings	155
10.3.4 Enveloppe error	155
10.4 REAL-TIME MOVEMENT CONTROL.....	155
10.5 MDESC TYPE	156
10.5.1 Definition	156
10.5.2 Operators	156
10.6 MOVEMENT INSTRUCTIONS.....	157
num movej (joint jPosition, tool tTool, mdesc mDesc)	157
num movej (point pPosition, tool tTool, mdesc mDesc)	157
num movei (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 getMoveId ()	162
num setMoveId (num nMoveId)	163

11 - OPTIONS	165
11.1 COMPLIANT MOVEMENTS WITH FORCE CONTROL	167
11.1.1 Principle	167
11.1.2 Programming.....	167
11.1.3 Force control	167
11.1.4 Limitations	168
11.1.5 Instructions.....	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: REAL TIME CONTROL ON A PATH	170
11.2.1 Principle	170
11.2.2 Programming.....	170
11.2.3 Constraints	170
11.2.4 Safety	171
11.2.5 Limitations	171
11.2.6 Instructions.....	171
num alterMovej (joint jPosition, tool tTool, mdesc mDesc)	171
num alterMovej (point pPosition, tool tTool, mdesc mDesc)	171
num alterMoveI (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 (trsf trAlteration)	174
num alterStopTime ()	175
11.3 OEM LICENCE CONTROL	176
11.3.1 Principles.....	176
11.3.2 Instructions	176
string getLicence (string sOemLicenceName, string sOemPassword)	176
11.4 ABSOLUTE ROBOT	177
11.4.1 Principle	177
11.4.2 Operation	177
11.4.3 Limitations	177
11.4.4 Instructions.....	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 CONTINUOUS AXIS	179
11.5.1 Principle	179
11.5.2 Instructions.....	179
joint resetTurn (joint jReference)	179

12 - APPENDIX	181
12.1 RUNTIME ERROR CODES.....	183
12.2 CONTROL PANEL KEYBOARD KEY CODES.....	184
13 - ILLUSTRATION	185
14 - INDEX.....	187

CHAPTER 1

INTRODUCTION

VAL 3 is a high-level programming language designed to control **Stäubli** robots in all kinds of applications.

VAL 3 language combines the basic features of a standard real-time high-level computer language with functionalities that are specific to the control of an industrial robot cell:

- robot control tools
- geometrical modelling tools
- input/output control tools

This reference manual explains the essential concepts of robot programming and describes the **VAL 3** instructions which fall into the following categories:

- Language elements
- Simple types
- User interface
- Tasks
- Libraries
- User types
- Robot control
- Arm position
- Movement control
- Screen type: for MCP screen display

Each instruction, together with its syntax, is listed in the table of contents for quick reference purposes.

CHAPTER 2

VAL 3 LANGUAGE ELEMENTS

The **VAL 3** programming language consists of applications. A **VAL 3** application contains both programs and data. A **VAL 3** application can also refer to other applications, used either as libraries, or as user type definitions.

2.1. APPLICATIONS

2.1.1. DEFINITION

A **VAL 3** application is a self-contained software package designed for controlling robots and inputs/outputs associated with a controller.

A **VAL 3** application comprises the following elements:

- a set of **programs**: the **VAL 3** instructions to be executed
- a set of **global data**: the data shared by all programs in the application
- a set of **libraries**: external applications used to share programs and/or data
- a set of **user types**: external applications used as templates to define structured data in the application

When an application is running, it also contains:

- a set of **tasks**: the programs being executed simultaneously

2.1.2. DEFAULT CONTENT

A new **VAL 3** application is created by copying the content of a predefined, template application. New user specific templates can be created. They simply consist in a standard **VAL 3** application placed in a dedicated folder on the controller.

A **VAL 3** application can be started only if it contains both a **start()** and a **stop()** program. Without **start()** and **stop()** programs, a **VAL 3** application can only be used as a library or a user type definition. It is possible to define applications that contain only data, or only programs.

2.1.3. START/STOP

The starting of a **VAL 3** application is managed by the controller. It can be either a user request from the **MCP** user interface, or automatic as part of the boot process.

Only one **VAL 3** application can be started at one time. This application can however use simultaneously many other applications (as libraries), and start many different execution tasks.

When a **VAL 3** application is ran, its **start()** program is executed.

A **VAL 3** application stops automatically when its last task is completed: the **stop()** program is then executed. All the tasks created by libraries, if any remain, are deleted in the reverse order to that in which they were created.

If a **VAL 3** application is stopped via the **MCP** user interface, the start task, if it still exists, is immediately destroyed. The **stop()** program is run next, and then any remaining application tasks are deleted in the reverse order to that in which they were created.

2.1.4. APPLICATION PARAMETERS

The following parameters can be used to configure a **VAL 3** application:

- unit of length
- amount of stack memory

These parameters cannot be accessed via a **VAL 3** instruction and can only be changed via the **MCP** user interface or using **VAL 3 Studio** in **Stäubli Robotics Suite**.

2.1.4.1. UNIT OF LENGTH

In **VAL 3** applications, the unit of length is either millimeter or inch. It is used by the **VAL 3** geometrical data types: frame, point, joint (for linear axes), transformation, tool, and trajectory blending.

The unit of length of an application is defined when an application is created, and it cannot be subsequently changed.

2.1.4.2. AMOUNT OF STACK MEMORY

Some memory is needed for each **VAL 3** task to store:

- The call stack (the list of program calls being executed in this task)
- The parameters for each program of the call stack
- The local variables for each program of the call stack

By default, each task has **5000** bytes for stack memory. There is usually no need to change that parameter.

This level may not be sufficient for applications containing large arrays of local variables or recursive algorithms: in this case, it must be increased via the **MCP** user interface or using **VAL 3 Studio** in **Stäubli Robotics Suite**, or the application must be optimized, by reducing the number of programs in the call stack, or by using global variables in place of local variables.

2.2. PROGRAMS

2.2.1. DEFINITION

A program is a sequence of **VAL 3** instructions to be executed.

A program consists of the following elements:

- The sequence of **instructions**: the **VAL 3** instructions to be executed
- A set of **local variables**: the internal program data
- A set of **parameters**: the data supplied to the program when it is called

Programs are used to group sequences of instructions that can be executed at various points in an application. In addition to saving programming time, they also simplify the structure of the applications, facilitate programming and maintenance and improve readability.

The number of instructions in a program is limited only by the amount of memory available in the system.

The number of local variables and parameters is limited only by the size of the stack memory for the application (see chapter 2.1.4.2).

2.2.2. RE-ENTRY

The programs are re-entrant; this means that a program can call itself recursively (**call** instruction), or it can be called simultaneously by several tasks. Each call of a program uses its own unique local variables and parameters. No interaction is possible between two different calls of the same program.

2.2.3. START() PROGRAM

The **start()** program is the program called when the **VAL 3** application is ran. It cannot have any parameters.

Typically, this program includes all the operations required to execute the application: initialization of the global variables and the outputs, creating the application tasks, etc.

The application does not terminate at the end of the **start()** program, as long as other application tasks are still running.

The **start()** program can be called from within a program (**call** instruction) in the same way as any other program.

2.2.4. STOP() PROGRAM

The **stop()** program is the program called when the **VAL 3** application stops. It cannot have any parameters.

Typically, this program includes all the operations required to stop the application correctly: resetting the outputs and stopping the application tasks according to an appropriate sequence, etc.

The **stop()** program can be called from within a program (**call** instruction) in the same way as any other program, but calling the **stop()** program does not stop the application.

2.2.5. PROGRAM CONTROL INSTRUCTIONS

Comment //

Syntax

```
// <String>
```

Function

A line starting with « // » is not executed and the execution resumes on the next line. You cannot use « // » in the middle of a line, they must be the first characters on the line.

Example

```
// This is an example of a comment
```

call program

Syntax

```
call program([parameter1][,parameter2])
```

Function

The call instruction executes a user-defined program. The number and the type of the expressions after the program name must match the interface of the program. The expressions specified as parameters are first executed in the order they are specified. The local variables are then initialized, and the execution of the program starts with its begin instruction.

The execution of a call is completed when the program executes a return or an end instruction.

Example

```
// Calls the pick() and place() programs for i, j between 1 and 10
for i = 1 to 10
  for j = 1 to 10
    call pick(pPallet1[i, j])
    call place(pPallet2[i, j])
  endFor
endFor
```

return

Syntax

```
return
```

Function

The return instruction terminates the execution of the current program immediately. If this program was called by a **call**, execution resumes after the **call** in the calling program. Otherwise (if the program is the **start()** program or the starting point of a task), the current task is completed. The return instruction has exactly the same effect as the end instruction at the end of the program.

A program is often easier to understand and maintain when its execution always ends with the end instruction. The use of a return instruction in the middle of a program is then not desirable.

if control instruction

Syntax

```

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

```

Function

The **if...elseif...else...endif** sequence evaluates successively the Boolean expressions marked with the **if** or **elseif** keywords, until one expression is true. The instructions following the Boolean expression are then executed, up to the next **elseif**, **else** or **endif** keyword. The program finally resumes after the **endif** keyword. If all Boolean expressions marked with **if** or **elseif** are false, the instructions between the **else** and **endif** keywords are executed (if the **els** keyword is present). The program then resumes after the **endif** keyword.

There is no restriction on the number of **elseif** expressions within an **if...endif** sequence.

The **if...elseif...else...endif** sequence can be replaced with the **switch...case...default...endSwitch** sequence when the different possible values of a single expression are tested.

Example

This program converts a **day** written in a **string** (**sDay**) into a **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
  // Weekend !
  nDay=0
endif

```

See also

switch control instruction

while control instruction

Syntax

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

Function

The instructions between **while** and **endWhile** are executed when the Boolean **bCondition** expression is (**true**). If the Boolean **bCondition** expression is not true at the first evaluation, the instructions between **while** and **endWhile** are not executed.

Parameter

bool bCondition Boolean expression to be evaluated

Example

```
// This simple program makes a signal flash as long as the robot is moving  
diLamp = false  
while (isSettled()==false)  
//Inverses the value of the diLamp: true false  
  diLamp = !diLamp  
//Waits ½ s  
  delay(0.5)  
endWhile  
diLamp = false
```

do ... until control instruction

Syntax

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

Function

The instructions between **do** and **until** are executed until the Boolean **bCondition** expression is (**true**).

The instructions between **do** and **until** are executed once if the Boolean **bCondition** expression is true during its first evaluation.

Parameter

bool bCondition Boolean expression to be evaluated

Example

```
// This program loops until the Enter key is pressed  
do  
// Waits for a key to be pressed  
  nKey = get()  
// Tests the Enter key code  
until (nKey == 270)
```

for control instruction

Syntax

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

Function

The instructions between **for** and **endFor** are executed until the **nCounter** exceeds the specified **nEnd** value.

The **nCounter** is initialized by the **nBeginning** value. If **nBeginning** exceeds **nEnd**, the instructions between **for** and **endFor** are not executed. At each iteration, the **nCounter** is incremented by the **nStep** value, and the instructions between **for** and **endFor** are repeated if the **nCounter** does not exceed **nEnd**.

If **nStep** is positive, the **for** loop stops when **nCounter** is greater than **nEnd**. If **nStep** is negative, the **for** loop stops when **nCounter** is less than **nEnd**.

Parameter

num nCounter	num type variable used as a counter
num nBeginning	numerical expression used to initialize the counter
num nEnd	numerical expression used for the loop end test
[num nStep]	numerical expression used to increment the counter

Example

```
// This program rotates axis 1 from -90° to +90° in -10° steps
for nPos = 90 to -90 step -10
  jDest.j1 = nPos
  movej(jDest, flange, mNomSpeed)
  waitEndMove()
endFor
```

switch control instruction

Syntax

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

Function

The **switch...case...default...endSwitch** sequence evaluates successively the expressions marked with the **case** keyword until one expression is equal to the initial expression after the **switch** keyword.

The instructions following the expression are then executed, up to the **break** keyword. The program finally resumes after the **endSwitch** keyword.

If no **case** expression is equal to the initial **switch** expression, the instructions between the **default** and **endSwitch** keywords are executed (if the **default** keyword is present).

There is no restriction on the number of **case** expressions within an **switch...endSwitch** sequence. The expressions after the **case** keyword must have the same type as the expression after the **switch** keyword.

The **switch...case...default...endSwitch** sequence is very similar to the **if...elseif...else...endif** sequence. It accepts not only Boolean expressions, but any type that supports the standard "is equal to" "==" operator.

Example

This program reads a **num** (**nMenu**) corresponding to a keystroke and modifies a **string** **s** in consequence.

```
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
```

This program converts a **day** written in a **string** (**sDay**) into a **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
  // Not a week day !
  nDay=0
  break
endSwitch
```

2.3. DATA

2.3.1. DEFINITION

A data is a set of values to be used as parameter or result of **VAL 3** instructions.

A data consists of the following elements:

- a set of values
- a type, that defines the possible values and allowed operations on the data. Boolean, Numeric and String are the most simple data types
- a container, that defines the way the values are stored in the data. Element, Array, Collection are the possible data containers in **VAL 3**

2.3.2. SIMPLE TYPES

The **VAL 3** language supports the following simple types:

- **bool** type: for Boolean values (true/false)
- **num** type: for numeric values (integer or floating point numbers)
- **string** type: for character strings (Unicode characters)
- **dio** type: for digital inputs/outputs
- **aio** type: for numeric inputs/outputs (analog or digital)
- **sio** type: for serial ports inputs/outputs and ethernet sockets
- **screen** type: for MCP screen display and keyboard access

In the documentation, the type of the variable is indicated with the initial lower case letters in its name:

- **bVariable** is a variable of type **bool**
- **nVariable** is a variable of type **num**
- **sVariable** is a variable of type **string**
- **diVariable** is a variable of type **dio**
- **aiVariable** is a variable of type **aio**
- **siVariable** is a variable of type **sio**
- **scVariable** is a variable of type **screen**

2.3.3. STRUCTURED TYPES

A structured type combines several simpler types into a new, higher level type. Each sub-type is given a name, and can be accessed individually as a field of the structure. Adequate types in an application organize the data in a way that makes computations and program evolutions easier.

The **VAL 3** language supports the following structured types made of simple types:

- **trsf** type: for Cartesian geometrical transformations
- **frame** type: for Cartesian geometrical frames
- **tool** type: for robot mounted tools
- **point** type: for the Cartesian positions of a tool
- **joint** type: for robot axis positions
- **config** type: for robot configurations
- **mdesc** type: for robot movement parameters

The **VAL 3** language supports also user types, that combines **VAL 3** simple, structured, or even other user types into a new type. A user type can be used in an application exactly as a standard type.

In the documentation, the type of the variable is indicated with the initial lower case letters in its name:

- **trVariable** is a variable of type **trsf**
- **fVariable** is a variable of type **frame**
- **tVariable** is a variable of type **tool**
- **pVariable** is a variable of type **point**
- **jVariable** is a variable of type **joint**
- **cVariable** is a variable of type **config**
- **mVariable** is a variable of type **mdesc**

2.3.4. CONTAINERS

The data container defines the way the values are stored in the data:

- An 'element' container simply consists in a single value. True (Boolean), 0 (numeric), 'text' (string), have an element container.
- An 'array' container consists in a set of values identified by 1, 2 or 3 integer indices. The starting index in arrays is always 0.
- A 'collection' container consists in a set of values identified by a string key. Any non-empty string can be used as value identifier.

A one dimension array container with a single value (index 0) is considered as an element container.

In the documentation, when needed, the container of the variable is identified with the name of the variable:

- **s1dArray** is a one dimension array of type **string**
- **s2dArray** is a two dimension array of type **string**
- **s3dArray** is a three dimension array of type **string**
- **sColl** is a collection of type **string**

Some instructions (to handle arrays or collections) do not care of the type of the data. In the documentation, the type is then replaced with a star: '*'.

2.4. DATA INITIALIZATION

2.4.1. SIMPLE TYPE DATA

The precise syntax for the initialisation of a simple type data is specified in the chapter describing each simple type. An array or a collection must be initialized element by element. The initialization value is **false** for a bool, **0** for a num and "" (empty string) for a string.

Example

In this example, bBool is a Boolean, nPi a numeric and sString a string variable.

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

2.4.2. STRUCTURED TYPE DATA

The value of a structured type data is defined by the sequence of its fields values between brackets, separated with commas. Empty fields values are replaced with 0. The sequence order is specified in the chapter describing each structured type. The value of a structure may include values of other nested sub-structures. An array or a collection of a structured type must be initialized element by element.

Example

The point type is made of a trsf and a config type. A point variable may be initialized as shown:

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

The transformation of the point could also be initialized with:

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

2.5. VARIABLES

2.5.1. DEFINITION

A variable is a data referenced by its name in an application or a program.

A variable is identified by:

- a name: a character string
- a scope: where the variable can be accessed (within a single program, shared by programs within an application, or shared between applications)
- a set of values
- a data type (simple or structured type)
- a data container (element, array or collection)

A variable name is a string of **1** to **15** characters selected from "**a..zA..Z0..9_**", and starting with a letter.

2.5.2. VARIABLE SCOPE

The variable scope can be:

- global: all programs in the application can use the variable, or
- local: the variable can only be accessed by the program in which it is declared

When a global variable and a local variable have the same name, the program in which the local variable is declared will use the local variable and will be unable to access the global variable.

When an application is used as a library, each global variable can be declared either as public or private. A public variable can be accessed by the applications that use the library, private variables can only be accessed within the library.

2.5.3. ACCESSING A VARIABLE VALUE

The access to the value of a variable depends on its container:

- the value of an element container is accessed with the name of the variable (without square brackets): nVariable.
- a value in an array is accessed with its numerical indices between square brackets after the name of the variable: n1dArray[nIndex], n2dArray[nIndex1, nIndex2], n3dArray[nIndex1, nIndex2, nIndex3].
- a value of a collection is accessed with its string key between square brackets after the name of the variable: nCollection[sKey]

A one dimension array container with a single value (index 0) is considered as an element container. Its value can be accessed without brackets: n1dArray is equivalent to n1dArray[0].

The numerical indices used to access a value in an array are rounded to the nearest integer value: n2dArray[5.01, 6.99] is equivalent to n2dArray[5, 7]

The index used to access a value in an array ranges between 0 and the size of the dimension minus one.

The fields of a structured type variable can be accessed using a '.' followed by the field name: pPoint.trsf.x refers to the value of the 'x' field of the 'trsf' field of the point data pPoint.

Example

Initialisation of simple type variables with different containers:

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



```
// Test if this library defines it
if (nLoadCode==0) or (nLoadCode==11)
  if (isDefined(part:sVersion)==false)
    // initial version
    sLibVersion = "v1.0"
  else
    sLibVersion = part:sVersion
  endif
endif
```

bool insert(*)

Function

This instruction creates a new value of the variable's type, and stores it in the variable container. The new value is initialized with the default value of the type. The size of the variable is increased by one.

For a one dimension array, the new value is inserted at the specified index position. The index position may be equal to the size of the array: the insertion is then made at the end of the array. "`insert(s1dArray[size(s1dArray)])`" is equivalent to "`append(s1dArray)`".

For collections, insert is accepted only if the key is not already used in the collection. The new value is associated with the specified key, and the function returns `true`. The instruction has no effect and returns `false` if the key was already in use. The instruction `isDefined()` can be used to check whether a key is used in a collection or not.

This instruction is not supported for two and three dimensions arrays (use the `resize()` instruction instead) and for local array variables. The size of a variable is limited to 9999 values. A runtime error is generated when the size of the variable exceeds this limit.

The insert instruction allocates system memory. The performance of memory allocation is not guaranteed. Therefore an extensive use of `insert()` should be avoided in VAL 3 applications where performance is an issue.

Example

This example adds a new article in a list.

```
// Ask for a new article name
put ("New article ?")
get (sArticleName)
println("")
// Ask for the position in the list
put ("Position ? ")
get (nIndex)
if (nIndex<0) or (nIndex>size(sArticleList))
  println("Error: invalid position")
else
  // Add new article in the list
  insert(sArticleList[nIndex])
  sArticleList[nIndex] = sArticleName
endif
```

This example adds a new article key in a collection.

```
// Ask for a new article name
put ("New article ?")
get (sArticleName)
if isDefined(sArticleColl[sArticleName])==true
  println("Error: reference already defined")
else
  // Add new article in the collection
  insert(sArticleColl[sArticleName])
endif
```

bool delete(*)

Function

This instruction deletes the specified value from the container of the variable. The size of the variable is decreased by one.

A runtime error is generated if the specified index or key is out of range. The instruction `isDefined()` can be used to check whether a key is used in a collection or not.

The size of a collection can be null, but an array variable must always have at least one element. A runtime error is generated when trying to delete the last element of an array.

This instruction is not supported for two and three dimensions arrays (use the `resize()` instruction instead) and for local array variables.

The `delete()` instruction frees system memory. The performance of memory garbage collection is not guaranteed. Therefore an extensive use of `delete()` should be avoided in VAL 3 applications where performance is an issue.

Example

This example deletes an article in a collection.

```
// Ask for the article to delete
put ("Article to remove ?")
get (sArticleName)
if isDefined(sArticleColl[sArticleName]) == true
  // remove the article from the collection
  delete(sArticleColl[sArticleName])
else
  println("Error: article not defined")
endIf
```

num getData(string sDataName, *)

Function

This instruction copies the value of a data, specified by its name `sDataName`, into the specified variable. If both the data and the variable are one dimension arrays, the `getData()` instruction copies all array's entries until the end of one of the array is reached. The instruction returns the number of copied entries in the variable.

The data name must have the following format: "library:name[index]", where "library:" and "[index]" are optional:

- "name" is the name of the data
- "library" is the name of the library identifier in which the data is defined
- "index" is the numerical value of the index to access when the data is a one dimension array

The instruction returns an error code when the data copy could not be performed:

Returned value	Description
<code>n > 0</code>	The variable was successfully updated with n entries copied
<code>-1</code>	The data does not exist
<code>-2</code>	The library identifier does not exist
<code>-3</code>	The index is out of range
<code>-4</code>	The data's type does not match the variable's type

Example

This program merges 2 arrays of points pApproach[] and pTrajectory[] from a library into a single local array pPath[].

```
// Copy approach points in path
i = getData("Part:pApproach", pPath)
if(i > 0)
  nPoints = i
  // Append trajectory points in path
  i = getData("Part:pTrajectory", pPath[nPoints])
  if(i >0)
    nPoints=nPoints+i
  endIf
endIf
```

2.5.5. INSTRUCTIONS SPECIFIC TO ARRAY VARIABLES

void append(*)

Function

This instruction creates a new value of the variable's type, and stores it at the end of the one dimension array variable. The new value is initialized with the default value of the type. The size of the variable is increased by one.

This instruction is not supported for two and three dimensions arrays, and for local array variables. The size of a variable is limited to 9999 values. A runtime error is generated when the size of the variable exceeds this limit.

The append instruction allocates system memory. The performance of memory allocation is not guaranteed. Therefore an extensive use of `append()` should be avoided in VAL 3 applications where performance is an issue.

Example

This example appends a new article in a list.

```
// Ask for a new article name
put ("New article ?")
get (sArticle)
println("")
append(sArticleList)
sArticleList[size(sArticleList)-1] = sArticle
```

num size(*, num nDimension)

Function

This instruction returns the size of the specified dimension in the array. If nDimension exceeds the array's dimensions, the function returns 0. For a one dimension array, `size(s1dArray, 1)` is equivalent to `size(s1dArray)`.

Example

The array variable must be specified without square brackets:

```
//Ok
nNbElements=size(s3dArray,3)
// Compilation error: unexpected indices
nNbElements=size(s3dArray[1,2,3],3)
```

void resize(*, num nDimension, num nSize)

Function

This instruction creates or deletes values in an array so that the size of the specified dimension matches the nSize value. The creation or deletion of values is done at the end of the array. The new values, if any, are initialized with the default value of the type.

This instruction is not supported for local array variables. The size of a variable is limited to 9999 values. A runtime error is generated when the size of the variable exceeds this limit.

The `resize()` instruction allocates or frees system memory. The performance of memory handling is not guaranteed. Therefore an extensive use of `resize()` should be avoided in VAL 3 applications where performance is an issue.

Example

The variable must be specified without index. The next instruction modifies s2dArray so that its second dimension is 5.

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

2.5.6. INSTRUCTIONS SPECIFIC TO COLLECTION VARIABLES

string first(*)

Function

This instruction returns the first key of a collection, sorted by key alphabetic order. If the collection is empty, the instruction returns an empty string "".

string next(*)

Function

This instruction returns the next key of a collection, sorted by key alphabetic order. If the specified key is the last key of the collection, the instruction returns an empty string "".

Example

This example prints on the user page all elements of a collection by key alphabetic order:

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

string last(*)

Function

This instruction returns the last key of a collection, sorted by key alphabetic order. If the collection is empty, the instruction returns an empty string "".

string prev(*)

Function

This instruction returns the previous key of a collection, sorted by key alphabetic order. If the specified key is the first key of the collection, the instruction returns an empty string "".

Example

This example prints on the user page all elements of a collection by key reverse alphabetic order:

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

2.6. PROGRAM PARAMETERS

Sub-program parameters are data that are transmitted from a calling program to a sub-program, with the call instruction. In the sub-program, parameters are like local variables that are initialized automatically when the sub-program is started.

There are different ways to pass a variable to a sub-program:

- you may want to pass only one value (one element) of the variable, or the container (array or collection) of the variable as a whole.
- you may allow the sub-program to modify the value of the variable (passing the variable "by reference"), or just pass a copy of the value to the sub-program, making sure the variable remains unchanged (passing a variable "by value").

A variable can be passed as parameter:

- by element value.
- by element reference.
- by array or collection reference.

Passing a container (array or collection) by value is not allowed.

A reference is marked with the '&' symbol after the data type in the program interface definition:

`num& nData` is a num parameter passed by element reference.

`num& n1dArray[]` is a num parameter passed by array reference (one dimension array).

`num& n2dArray[,]` is a num parameter passed by array reference (two dimensions array).

`num& n3dArray[,,,]` is a num parameter passed by array reference (three dimensions array).

`num& nCollection[""]` is a num parameter passed by collection reference.

In the documentation, the same notation is used for instruction description:

`bool pointToJoint(tool tTool, joint jInitial, point pPosition, joint& jResult)` is an instruction returning a boolean value, taking a tool, a joint, and a point data passed by element value, and a joint passed by element reference.

`num fromBinary(num& nDataByte[], num nDataSize, string sDataFormat, num& nValue[])` is an instruction returning a numerical value, taking a one dimension array as first parameter (passed by reference), a numerical and a string data passed by element value, and a one dimension array as last parameter (passed by reference).

2.6.1. PARAMETER BY ELEMENT VALUE

When a parameter is defined by element value, the system creates a local variable and initializes it with the value of the **VAL 3** instruction supplied by the calling program. If the supplied instruction is a variable, the parameter is initialized with a copy of the value of the variable. All changes made to the value of the parameter in the sub-program have no impact on the value of the variable in the calling program.

Example

Let `sendMessage(string sMessage)` be a program with a single parameter passed by element value.

`sendMessage()` can be used with a constant data or the result of computation:

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

`sendMessage()` can be used with values from elements, arrays or collections:

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

After these calls, the value of `sMessage`, `sMessageArray[23]`, `s2dArray[12,3]`, `s3dArray[5,7,9]`, `sMessageColl[sMessageName]` will not have been changed by the instructions in `sendMessage()`.

2.6.2. PARAMETER BY ELEMENT REFERENCE

When a parameter is defined by element reference, the system creates a local variable and initializes it with a link to the data supplied by the calling program. The variable passed by reference may have an element, array or collection container, but only the value specified in the call is passed to the sub-program. The container of the parameter is always an element. All changes made to the value of the parameter in the sub-program directly affect the corresponding value of the calling program's data. It is not possible to pass a **VAL 3** constant data or the result of a **VAL 3** expression by element reference.

Example

Let `sendMessage(string& sMessage)` be a program with a single parameter passed by element reference.

`sendMessage()` cannot be used with a constant data or the result of computation:

```
// compilation errors: variable expected as parameter
call sendMessage ("Waiting for signal StartCycle")
call sendMessage ("Waiting for signal"+sSignalName)
```

`sendMessage()` can be used with values from elements, arrays or collections:

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

After these calls, the value of `sMessage`, `sMessageArray[23]`, `s2dArray[12,3]`, `s3dArray[5,7,9]`, `sMessageColl[sMessageName]` may have been changed by the instructions in `sendMessage()`.

2.6.3. PARAMETER BY ARRAY OR COLLECTION REFERENCE

When a parameter is defined by array or collection reference, the system creates a local variable and initializes it with a link to the data supplied by the calling program. The container of the parameter in the sub program is the same as the container of the supplied variable: a one, two or three dimensions array, or a collection. All changes made to any value of the parameter in the sub-program directly affect the corresponding value of the calling program's data.

It is possible for one dimension arrays to pass only part of the array to the sub-program, by specifying the first accessible element in the array. For two dimensions arrays, three dimensions arrays, and collections, it is not possible to pass only part of the array or collection to the sub-program. The variable must then be passed without square brackets [] specifying an index or a key. It is not possible to pass a **VAL 3** constant data or the result of a **VAL 3** expression by array or collection reference.

Example

Let `send1dMessage(string& s1dArray[])` be a program with a single parameter passed by array reference (one dimension).

Let `send2dMessage(string& s2dArray[])` be a program with a single parameter passed by array reference (two dimensions).

Let `send3dMessage(string& s3dArray[])` be a program with a single parameter passed by array reference (three dimensions).

Let `sendCollMessage(string& sMessageColl[""])` be a program with a single parameter passed by collection reference.

None of these programs can be used with a constant data or the result of computation:

```
// compilation errors: array variable expected as parameter
call send1dMessage ("Waiting for signal StartCycle")
call send1dMessage ("Waiting for signal"+l_sSignalName)
```

The container of the passed variable must match the declared container of the parameter:

```
// compilation errors: 1d array variable expected
call send1dMessage (sMessageColl)
call send1dMessage (s2dArray[12,3])
// compilation error: collection variable expected
call sendCollMessage (sMessage)
```

It is not possible to pass part of an array or collection, except for 1d arrays. Arrays and collections must be specified without index or key.

```
// correct parameter
call send2dMessage (s2dArray)
call send3dMessage (s3dArray)
call sendCollMessage (sMessageColl)
call send1dMessage (sMessageArray)
call send1dMessage (sMessageArray[23])
// compilation errors: unexpected indices for the array
call send2dMessage (s2dArray[12,3])
call send3dMessage (s3dArray[5,7,9])
// compilation error: unexpected indices for the collection
call sendCollMessage (sMessageColl[l_sMessageName])
```

In the latter case, only the part of the array `sMessageArray` starting with index 23 is passed to the `send1dMessage()` program. Values of `sMessageArray` with index lower than 23 cannot be accessed by `send1dMessage()`.

CHAPTER 3

SIMPLE TYPES

3.1. BOOL TYPE

3.1.1. DEFINITION

bool type values or constants can be:

- **true**: true value
- **false**: false value

When a **bool** type variable is initialized, its default value is **false**.

3.1.2. OPERATORS

In ascending order of priority:

bool < bool & bVariable > = < bool bCondition >	Assigns the value of bCondition to the variable bVariable and returns the value of bCondition
bool < bool bCondition1 > or < bool bCondition2 >	Returns the value of the logical OR between bCondition1 and bCondition2 . bCondition2 is only assessed if bCondition1 is false .
bool < bool bCondition1 > and < bool bCondition2 >	Returns the value of the logical AND between bCondition1 and bCondition2 . bCondition2 is only assessed if bCondition1 is true .
bool < bool bCondition1 > xor bool < bCondition2 >	Returns the value of the exclusive OR between bCondition1 and bCondition2
bool < bool bCondition1 > != < bool bCondition2 >	Tests the equality of the values of bCondition1 and bCondition2 . Returns true if the values are different, and otherwise returns false .
bool < bool bCondition1 > == < bool bCondition2 >	Tests the equality of the values of bCondition1 and bCondition2 . Returns true if the values are identical, and otherwise returns false .
bool ! < bool bCondition >	Returns the negation of the value of the bCondition

To avoid confusions between = and == operators, the = operator is not allowed within VAL 3 expressions used as instruction parameter. **if**(**bCondition1**=**bCondition2**) would be interpreted as **bCondition1**=**bCondition2**; **if**(**bCondition1**==**true**). But often the intention was to write: **if**(**bCondition1**==**bCondition2**), which is really different !

3.2. NUM TYPE

3.2.1. DEFINITION

The **num** type represents a numerical value with about **14** significant digits.

The accuracy of each numerical computation is therefore limited by these **14** significant digits.

This must be taken into account when testing the equality of two numerical values: this must normally be done within a specific level.

The format of numerical type constants is as follows:

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

The 'e' is a marker for numerical scientific notation, in replacement of '10^': 1e3 is equal to 1×10^3 (or 1000), 1e-2 is equal to 1×10^{-2} (or 0.01).

The default initialization value of **num** type variables is **0**.

Example

A test on the result of numerical computation must take into account the numerical unaccuracy of computations.

```
if cos(nAngle) == 0, if abs(cos(nAngle)) < 1e-10 should better be replaced with if abs(cos(nAngle)) < 1e-10.
```

Here are some constant numbers:

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

3.2.2. OPERATORS

In ascending order of priority:

num <num& nVariable> = <num nValue>	Assigns nValue to the variable nVariable and returns nValue .
bool <num nValue1> != <num nValue2>	Returns true if nValue1 is not equal to nValue2 , otherwise returns false .
bool <num nValue1> == <num nValue2>	Returns true if nValue1 is equal to nValue2 , otherwise returns false .
bool <num nValue1> >= <num nValue2>	Returns true if nValue1 is greater than or equal to nValue2 , otherwise returns false .
bool <num nValue1> > <num nValue2>	Returns true if nValue1 is definitely greater than nValue2 , otherwise returns false .
bool <num nValue1> <= <num nValue2>	Returns true if nValue1 is less than or equal to nValue2 , otherwise returns false .
bool <num nValue1> < <num nValue2>	Returns true if nValue1 is definitely less than nValue2 , otherwise returns false .
num <num nValue1> - <num nValue2>	Returns the difference between nValue1 and nValue2 .
num <num nValue1> + <num nValue2>	Returns the sum of nValue1 and nValue2 .
num <num nValue1> % <num nValue2>	Modulo operation: It returns the remainder of the integer division of nValue1 by nValue2 . A runtime error is generated if nValue2 is 0 . The sign of the remainder is the same as that of nValue1 .
num <num nValue1> / <num nValue2>	Returns the quotient of nValue1 by nValue2 . A runtime error is generated if nValue2 is 0 .
num <num nValue1> * <num nValue2>	Returns the product of nValue1 and nValue2 .
num - <num nValue>	Returns the inverse of nValue .

To avoid confusions between = and == operators, the = operator is not allowed within VAL 3 expressions used as instruction parameter. nCos=cos(nAngle=30) must be replaced with nAngle=30; nCos=cos(nAngle).

num atan(num nValue)

Function

Returns the inverse tangent of **nValue**, in degrees. The resulting angle is between **-90** and **+90** degrees.

Example

`atan(1)` returns 45

num abs(num nValue)

Function

Returns the absolute value of **nValue**.

Example

A test on the result of numerical computation must take into account the numerical unaccuracy of computations:

`if cos(nAngle)==0` should better be replaced with `if abs(cos(nAngle))<1e-10`.

`abs(3.1415)` returns 3.1415

`abs(-3.1415)` returns 3.1415

num sqrt(num nValue)

Function

Returns the square root of **nValue**.

A runtime error is generated if **nValue** is negative.

Example

`sqrt(9)` returns 3

num exp(num nValue)

Function

Returns the exponential function of **nValue**.

A runtime error is generated if **nValue** is too large.

Example

`exp(1)` returns 2.718281828459

num power(num nX, num nY)

Function

Returns **nX** to the power **nY**: nX^{nY}

A runtime error is generated if **nX** is negative or null, or if the result is too large.

Example

This program computes in 2 different ways 5 to the power 7.

```
// First way: power instruction
nResult = power(5,7)
// Second way: power(x,y)=exp(y*ln(x)) (with numerical inaccuracy)
nResult = exp(7*ln(5))
```

num ln(num nValue)

Function

Returns the natural logarithm of **nValue**.

A runtime error is generated if **nValue** is negative or zero.

Example

`ln(2.718281828)` returns 0.99999999983113

num log(num nValue)

Function

Returns the common logarithm of **nValue**.

A runtime error is generated if **nValue** is negative or zero.

Example

`log(1000)` returns 3

num roundUp(num nValue)

Function

Returns **nValue** rounded up to the nearest integer.

Example

`roundUp(7.8)` returns 8

`roundUp(-7.8)` returns -7

num roundDown(num nValue)

Function

Returns **nValue** rounded down to the nearest integer.

Example

`roundDown(7.8)` returns 7

`roundDown(-7.8)` returns -8

num round(num nValue)

Function

Returns **nValue** rounded up or down to the nearest integer.

Example

`round(7.8)` returns 8

`round(-7.8)` returns -8

`round(0.5)` returns 1

`round(-0.5)` returns 0

num min(num nX, num nY)

Function

Returns the minimum values of **nX** and **nY**.

Example

`min(-1, 10)` returns -1

num max(num nX, num nY)

Function

Returns the maximum values of **nX** and **nY**.

Example

`max(-1, 10)` returns 10

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

Function

Returns **nValue** limited by **nMin** and **nMax**.

Example

`limit(30, -90, 90)` returns 30

`limit(100, -90, 90)` returns 90

`limit(-100, -90, 90)` returns -90

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

Function

Returns **nValue1** if **bCondition** is **true**, otherwise returns **nValue2**.

Example

`sel(true, -90, 90)` returns -90

`sel(false, -90, 90)` returns 90

3.3. BIT FIELD TYPE

3.3.1. DEFINITION

A bit field is a mean to store and exchange in a compact way a series of bits (Boolean values or digital Inputs/Outputs). **VAL 3** does not provide a specific data type to handle bit fields, but reuses the num type to store a 32-bits bit field as a positive integer value in the range $[0, 2^{32}]$.

Any **VAL 3** numerical value can be seen as 32-bits bit field; the bit field handling instructions automatically round a numerical value into a 32-bits positive integer that is then treated as a 32-bits bit field.

3.3.2. OPERATORS

The standard operators of the num type apply on a bit field: '=', '==', '!='.

3.3.3. INSTRUCTIONS

num bNot(num nBitField)

Function

This instruction returns the bitwise logical 'not' (negation) operation on a 32-bits bit field. (The *i* th bit of the result is set to 1 if the *i* th bit of the input is 0). This result is therefore a positive integer in the range $[0, 2^{32}]$.

The numerical input is first rounded to a positive integer in the range $[0, 2^{32}]$ before the bitwise operation is applied.

Example

This program resets bits *i* to *j* of a bit field nBitField using a mask nMask.

```
// Compute a bit mask with bits i to j set to 1 (see bOr for explanations)
nMask=(power(2,j-i+1)-1)*power(2,i)
// Invert the mask to have all bits to 1 except bit i to j
nMask=bNot(nMask)
// Reset bits i to j using the bitwise 'and'
nBitField=bAnd(nBitField, nMask)
```

num bAnd(num nBitField1, num nBitField2)

Function

This instruction returns the bitwise logical 'and' operation on two 32-bits bit fields. (The *i* th bit of the result is set to 1 if the *i* th bits of both inputs are set to 1). This result is therefore a positive integer in the range $[0, 2^{32}]$.

The numerical inputs are first rounded to a positive integer in the range $[0, 2^{32}]$ before the bitwise operation is applied.

Example

This program displays a 32 bits bit field nBitField on screen by testing each bit one after the other:

```
for i=31 to 0 step -1
// Compute the mask for the i th bit
nMask=power(2,i)
if bAnd(nBitField, nMask)==nMask
put("1")
else
put("0")
endif
endFor
putln("")
```

num bOr(num nBitField1, num nBitField2)

Function

This instruction returns the bitwise logical 'or' operation on two 32-bits bit fields. (The *i* th bit of the result is set to 1 if the *i* th bit of at least one input is set to 1). This result is therefore a positive integer in the range $[0, 2^{32}]$.

The numerical inputs are first rounded to a positive integer in the range $[0, 2^{32}]$ before the bitwise operation is applied.

Example

This program computes in two different ways a bit field mask where the *i* th to the *j* th bits are set.

```
// First way: logical 'or' on bits i to j
nBitField=0
for k=i to j
  nBitField=bOr(nBitField, power(2,k))
endFor
// Second way: compute a bit mask of (j-i) bits
nBitField=(power(2,j-i+1)-1)
// Then shift the bit mask by i bits
nBitField=nBitField*power(2,i)
```

num bXor(num nBitField1, num nBitField2)

Function

This instruction returns the bitwise logical 'xor' (exclusive or) operation on two 32-bits bit fields. (The *i* th bit of the result is set to 1 if the *i* th bits of the two inputs are different). This result is therefore a positive integer in the range $[0, 2^{32}]$.

The numerical inputs are first rounded to a positive integer in the range $[0, 2^{32}]$ before the bitwise operation is applied.

Example

This program inverts bits *i* to *j* of the bit field nBitField:

```
// Compute mask for bits i to j (see bOr example)
nMask=(power(2,j-i+1)-1)*power(2,i)
// Invert bits i to j using the mask
nBitField=bXor(nBitField,nMask)
```

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

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

Function

The purpose of the **toBinary/fromBinary** instructions is to enable the exchange of numerical values between two devices, using a serial line or a network connection. The numerical values are first encoded into a stream of bytes. The bytes are then sent to the peer device. Finally the peer device decodes the bytes to recover the initial numerical values. Different binary encodings of numerical values are possible.

The **toBinary** instruction encodes numerical values into an array of bytes (8-bits bit field, positive integer in the range [0, 255]) as specified by the data format **sDataFormat**. The number of numerical values **nValue** to encode is given by the **nValueSize** parameter. The result is stored in the **nDataByte** array, and the instruction returns the number of encoded bytes in this array.

A runtime error is generated if the number of values to encode **nValueSize** is greater than the size of **nValue**, if the specified format is not supported or if the result array **nDataByte** is not large enough to encode all input data.

The **fromBinary** instruction decodes an array of bytes into numerical values **nValue**, as specified by the data format **sDataFormat**. The number of bytes to decode is given by the **nDataSize** parameter. The result is stored in the **nValue** array and the instruction returns the number of values in this array. If some binary data are corrupted (bytes out of the range [0, 255] or invalid floating point encoding), the instruction returns the opposite of the number of correctly decoded values (negative value).

A runtime error is generated if the number of bytes to decode **nDataSize** is greater than the size of **nDataByte**, if the specified format is not supported or if the result array **nValue** is not large enough to decode all input data.

The supported binary encodings are given by the table below:

- The sign "-" indicates the encoding of a signed integer (the last bit of the bit field encodes the sign of the value).
- The digit gives the number of bytes for the encoding of each numerical value.
- The ".0" extension marks floating point values encoding (both IEEE 754 single and double precision encodings are supported).
- The final letter specifies the order of the bytes: "l" for 'little endian' (the less significant byte is encoded first), "b" for 'big endian' (the most significant byte is encoded first). The 'big endian' encoding is the standard for networking applications (TCP/IP).

"-1"	Signed byte
"1"	Unsigned byte
"-2l"	Signed word, little endian
"-2b"	Signed word, big endian
"2l"	Unsigned word, little endian
"2b"	Unsigned word, big endian
"-4l"	Signed double word, little endian
"-4b"	Signed double word, big endian
"4l"	Unsigned double word, little endian
"4b"	Unsigned double word, big endian
"4.0l"	Single precision floating point value, little endian
"4.0b"	Single precision floating point value, big endian
"8.0l"	Double precision floating point value, little endian
"8.0b"	Double precision floating point value, big endian

The native **VAL 3** format for numerical data is the double precision encoding. This format must be used to exchange numerical values without loss of accuracy.

Example

The first program encodes a **trsf** data **trShiftOut** into a byte array **nByteOut** and sends it with the **siTcpClient** serial connection. The second program reads the bytes from the **siTcpServer** serial connection and convert them back into a **trsf trShiftIn**.

```
// ---- Program to send a trsf ----
// Copy the trsf coordinates into a numerical buffer
nTrsfOut[0]=trShiftOut.x
nTrsfOut[1]=trShiftOut.y
nTrsfOut[2]=trShiftOut.z
nTrsfOut[3]=trShiftOut.rx
nTrsfOut[4]=trShiftOut.ry
nTrsfOut[5]=trShiftOut.rz
// Encode 6 numerical values (double precision floating point, therefore 8 bytes) into 6*8=48 bytes in nByteOut[48]
array
toBinary(nTrsfOut, 6, "8.0b", nByteOut)
// Send nByte array (48 bytes) through tcpClient
sioSet(siTcpClient, nByteOut)

// ---- Program to read a trsf ----
nb=0
i=0
while (nb<48)
  nb=sioGet(siTcpServer, nByteIn[i])
  if(nb>0)
    i=i+nb
  else
// Communication error
    return
  endif
endwhile
if (fromBinary(nByteIn, 48, "8.0b", nTrsfIn) != 6)
  // Corrupted data
  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 TYPE

3.4.1. DEFINITION

String type variables are used to store texts. The string type supports the standard Unicode character set. Note that the correct display of a Unicode character depends on the character fonts installed on the display device.

A string is stored on 128 bytes; the maximum number of characters in a string depends on the characters used, because the internal character encoding (Unicode UTF8) uses from 1 byte (for ASCII characters) to 4 bytes (3 for Chinese characters).

The maximum length of a ASCII string is therefore 128 characters; the maximum length of a Chinese string is 42 characters.

The default initialization value of string type variables is "" (empty string).

3.4.2. OPERATORS

In ascending order of priority:

string <string& sVariable> = <string sString>	Assigns sString to the variable sVariable and returns sString .
bool <string sString1> != <string sString2>	Returns true if sString1 and sString2 are not identical, otherwise returns false .
bool <string sString1> == <string sString2>	Returns true if sString1 and sString2 are identical, otherwise returns false .
string <string sString1> + <string sString2>	Returns the first characters (limited to 128 bytes) of sString1 concatenated with sString2 .

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter. `nLen=len(sString="hello wild world")` must be replaced with `sString="hello wild world"; nLen=len(sString)`.

3.4.3. INSTRUCTIONS

string toString(string sFormat, num nValue)

Function

This instruction returns a character string representing **nValue** according to the **sFormat** display format.

The format is "**size.precision**", where **size** is the minimum size of the result (spaces are added at the beginning of the string if necessary), and **precision** is the number of significant digits after the decimal point (the **0** at the end of the string are replaced by spaces). By default, **size** and **precision** equal **0**. The value's integer portion is never shortened, even if its display length exceeds **size**.

Example

returns

```
nPi = 3.141592654
toString(".4", nPi) returns "3.1416"
toString("8", nPi) returns "      3" (7 spaces before the '3')
toString("8.4", nPi) returns "  3.1416" (2 spaces before the '3')
toString("8.4", 2.70001) returns "  2.7   " (2 spaces before the '2', 3 spaces after the '7')
toString("", nPi) returns "3"
toString("1.2", 1234.1234) returns "1234.12"
```

See also

string chr(num nCodePoint)

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

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

Function

This instructions finds the numerical **nValue** represented at the beginning of the **sString** specified, and returns **sString** in which all the characters have been deleted until the next representation of a numerical value.

If the beginning of the **sString** does not represent a numerical value, **bReport** is set to **false** and **nValue** is not modified, otherwise **bReport** is set to **true**.

Example

```
toNum("10 20 30", nVal, bOk) returns "20 30", nVal equals 10, bOk equals true
toNum("a10 20 30", nVal, bOk) returns "a10 20 30", nVal is unchanged, bOk equals false
toNum("10 end", nVal, bOk) returns "", nVal equals 10, bOk equals true
```

This program displays successively 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)
```

See also

string toString(string sFormat, num nValue)

string chr(num nCodePoint)

Function

This instructions returns the string made up of the specified Unicode code point character, if it is a valid Unicode code point. Otherwise it returns an empty string.

The following table gives the Unicode code points below **128** (it matches the **ASCII** character table). The characters in grey boxes are control codes that may be replaced with a question mark when the string is displayed.

All valid Unicode code points are supported by the **VAL 3** string type. However, the display of the character depends on the installed character fonts on the display device. The complete list of Unicode characters can be found at <http://www.unicode.org> (search '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

Example

`chr(65)` returns "A"

See also

`num asc(string sText, num nPosition)`

num asc(string sText, num nPosition)

Function

This instruction returns the Unicode code point of the **nPosition** index character.

It returns -1 if **nPosition** is negative or greater than the specified text length.

Example

```
asc("A", 0) returns 65
```

See also

string chr(num nCodePoint)

string left(string sText, num nSize)

Function

This instructions returns the first **nSize** characters of **sText**. If **nSize** is greater than the length of **sText**, the instruction returns **sText**.

A runtime error is generated if **nSize** is negative.

Example

```
left("hello world", 5) returns "hello"
```

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

string right(string sText, num nSize)

Function

This instruction returns the last **nSize** characters of **sText**. If the number specified is greater than the length of **sText**, the instruction returns **sText**.

A runtime error is generated if **nSize** is negative.

Example

```
right("hello world", 5) returns "world"
```

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

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

Function

Returns **nSize** characters of **sText** from the **nPosition** index character, stopping at the end of **sText**.

A runtime error is generated if **nSize** or **nPosition** are negative.

Example

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

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

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

Function

This instruction returns **sText** in which **sInsertion** is inserted after the **nPosition** index character. If **nPosition** is greater than the size of **sText**, **sInsertion** is inserted at the end of **sText**. The result is truncated if it exceeds 128 bytes.

A runtime error is generated if **nPosition** is negative.

Example

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

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

Function

This instruction returns **sText** in which **nSize** have been deleted from the **nPosition** index character. If **nPosition** is greater than the length of **sText**, the instruction returns **sText**.

A runtime error is generated if **nSize** or **nPosition** are negative.

Example

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

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

Function

This instruction returns **sText** in which **nSize** characters have been replaced from the **nPosition** index character by **sReplacement**. If **nPosition** is greater than the length of **sText**, the instruction returns **sText**.

A runtime error is generated if **nSize** or **nPosition** are negative.

Example

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

num find(string sText1, string sText2)

Function

This instruction returns the index (between **0** and **127**) of the first character in the first occurrence of **sText2** in **sText1**. If **sText2** does not appear in **sText1**, the instruction returns **-1**.

Example

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

num len(string sText)

Function

This instruction returns the number of characters in **sText**.

Example

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

See also

`num getDisplayLen(string sText)`

3.5. DIO TYPE

3.5.1. DEFINITION

dio type variables are used to interface a **VAL 3** application with system digital inputs and outputs. A **dio** variable stores a link to a system digital input or output, the "physical address".

All instructions using a **dio** type variable not linked to an input/output declared in the system generate a runtime error. The default initialization value of **dio** type variables is an undefined link. The link of a **dio** variable can be initialized from another **dio** variable, from the robot **MCP**, or using **VAL 3 Studio** in **Stäubli Robotics Suite**.

3.5.2. OPERATORS

In ascending order of priority:

bool < dio diOutput> = < bool bCondition>	Assigns bCondition to the diOutput status and returns bCondition . A runtime error is generated if diOutput is not linked to a system output.
bool < dio diInput1> != < bool bInput2>	Returns true if diInput1 and bInput2 do not have the same status, otherwise returns false .
bool < dio diInput> != < bool bCondition>	Returns true if the diInput status is not equal to bCondition , otherwise returns false .
bool < dio diInput> == < bool bCondition>	Returns true if the diInput status is equal to bCondition , otherwise returns false .
bool < dio diInput1> == < dio diInput2>	Returns true if diInput1 and diInput2 have the same status, otherwise returns false .

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter. `if(diOutput=diInput)` would be interpreted as: `diOutput=diInput`; `if(diOutput==true)`. But often the intention was to write: `if(diOutput==diInput)`, which is really different !

CAUTION:

The '=' operator between two **dio** variables does not exist any more with **VAL 3 s7** for consistency with other '=' operators (refer to the definition of the '=' operator for user types). It can be easily replaced with the '=' operator between a **dio** and a **bool**:
`diOut = diIn` of earlier **VAL 3** versions can be replaced with `diOut = (diIn==true)`.

3.5.3. INSTRUCTIONS

void dioLink(dio& diVariable, dio diSource)

Function

This instruction links **diVariable** to the input/output to which **diSource** is linked.

Example

This application uses a signal that can be configured with different hardware devices. The program below tests which device is installed to initialize the diSignal variable that is then used in the rest of the application.

```

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("Error: no io device installed")
endif

```

num dioGet(dio diArray[])

Function

This instruction returns the numerical value from **diArray** read as an integer written in binary code, i.e.: **diArray[0] + 2 * diArray[1] + 4 * diArray[2] + ... + 2^k * diArray[k]**, where **diArray[i] = 1** if **diArray[i]** is **true**, otherwise **0**.

A runtime error is generated if a member of **diArray** is not linked to a system input/output.

Example

```

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

```

See also

num dioSet(dio diArray[], num nValue)

num dioSet(dio diArray[], num nValue)

Function

This instruction converts the integer part of **nValue** in binary code, assigns it to the outputs of **diArray**, and returns the corresponding value, i.e.:

$diArray[0] + 2 * diArray[1] + 4 * diArray[2] + \dots + 2^k * diArray[k]$, where $diArray[i] = 1$ if $diArray[i]$ is **true**, otherwise **0**.

A runtime error is generated if a member of **diArray** is not linked to a system output.

Example

Using di4bitsArray, array of size 4:

`dioSet(di4bitsArray, 10)` returns 10

`dioSet(di4bitsArray, 26)` returns 10, because 26 requires 5 bits in a binary encoding: $10 = 26 - 2^4$

See also

`num dioGet(dio diArray[])`

num ioStatus(dio diInputOutput)

Function

This instruction returns a positive number if the specified input output variable is working, and a negative number if it is in error. The returned value details the status of the input output:

0	The input output is working.
1	The input output is working, but is locked by the operator. Inputs have then a fixed value (controlled by the operator) that may differ from the hardware value. Outputs have then a fixed value, controlled by the operator: writing to the output has no effect at all. The lock mode is a debugging mean.
2	The input output is simulated (software input output, no impact to hardware).
-1	The input output is not working because the link (physical address) is not defined.
-2	The input output is not working because the link (physical address) does not match any system input output. The hardware device corresponding to the physical address is either not installed or could not be initialized.
-3	The input output is not working because the input output device is in error.

Example

This application uses a signal that can be configured with different hardware devices. The program below tests which device is installed to initialize the diSignal variable that is then used in the rest of the application.

```

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("Error: no io device installed")
endif

```

See also

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

`num ioStatus(aio aiInputOutput)`

`num ioBusStatus(string& sErrorDescription[])`

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

Function

This instruction performs exactly as the ioStatus instruction described above, but returns in addition the description text and the link (physical address) for the specified input output.

The description is a free text defined with the input output control tools. The format of the physical link depends on the input output device. It has usually the form: 'deviceName\moduleName\ioAddress'.

Example

This program tests a signal and displays error information if it is not working.

```
if ioStatus(diSignal, sDecription, sPath)<0
    putln("Signal "+sPath+ "in error")
    putln("Description:"+sDecription)
endif
```

See also

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

3.6. AIO TYPE

3.6.1. DEFINITION

aiO type variables are used to interface a **VAL 3** application with system analog inputs and outputs. A **aiO** variable stores a link to a system analog input or output, the "physical address".

All instructions using a **aiO** type variable not linked to an input/output declared in the system generate a runtime error. The default initialization value of **aiO** type variables is an undefined link. The link of a **aiO** variable can be initialized from another **aiO** variable, from the robot **MCP**, or using **VAL 3 Studio** in **Stäubli Robotics Suite**.

3.6.2. INSTRUCTIONS

void **aiOLink**(**aiO**& aiVariable, **aiO** aiSource)

Function

This instructions links **aiVariable** to the input/output to which **aiSource** is linked.

Example

This application uses a signal that can be configured with different hardware devices. The program below tests which device is installed to initialize the aiSignal variable that is then used in the rest of the application.

```

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("Error: no io device installed")
endif

```

num **aiOGet**(**aiO** aiInput)

Function

This instruction returns the numerical value of **aiInput**.
A runtime error is generated if **aiInput** is not linked to a system input/output.

Example

aiOGet(aiSensor) returns the current sensor value

See also

num **aiOSet**(**aiO** aiOutput, **num** nValue)

num **aiOSet**(**aiO** aiOutput, **num** nValue)

Function

This instruction assigns **nValue** to **aiOutput** and returns **nValue**. If the value being set is out of the range of the **aiO**, the returned number will be the actual value of the **aiO** output.

A runtime error is generated if **aiOutput** is not linked to a system output.

Example

aiOSet(aiCommand, -12.3) writes -12.3 to the output command and returns -12.3 if aiCommand is a floating point output.

aiOSet(aiCommand, 12.3) writes 12 to the output command and returns 12 if aiCommand is an integer output.

See also

`num aioGet(aio ailInput)`

`num ioStatus(aio ailInputOutput)`

Function

This instruction returns a positive number if the specified input output variable is working, and a negative number if it is in error. The returned value details the status of the input output:

0: The input output is working.

1: The input output is working, but is locked by the operator. Inputs have then a fixed value (controlled by the operator) that may differ from the hardware value. Outputs have then a fixed value, controlled by the operator: writing to the output has no effect at all. The lock mode is a debugging mean.

2: The input output is simulated (software input output, no impact to hardware).

-1: The input output is not working because the link (physical address) is not defined.

-2: The input output is not working because the link (physical address) does not match any system input output. The hardware device corresponding to the physical address is either not installed or could not be initialized.

-3: The input output is not working because the input output device is in error.

Example

This application uses a signal that can be configured with different hardware devices. The program below tests which device is installed to initialize the `aiSignal` variable that is then used in the rest of the application.

```
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("Error: no io device installed")
endif
```

See also

`num ioStatus(dio dilInputOutput)`

`num ioStatus(aio dilInputOutput, string& sDescription, string& sPhysicalPath)`

Function

This instruction performs exactly as the `ioStatus` instruction described above, but returns in addition the description text and the link (physical address) for the specified input output.

The description is a free text defined with the input output control tools. The format of the physical link depends on the input output device. It has usually the form: 'deviceName\moduleName\ioAddress'.

Example

This program tests a signal and displays error information if it is not working.

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

See also

`num ioStatus(aio ailInputOutput)`

`num ioStatus(dio dilInputOutput)`

3.7. SIO TYPE

3.7.1. DEFINITION

The **sio** type is used to link a **VAL 3** variable to a serial port or an Ethernet socket connection. A **sio** input-output is characterized by:

- Parameters specific to the type of communication, defined in the system
- An end of string character, to allow the use of the **string** type
- A communication timeout delay

The serial system inputs-outputs are active at all times. The Ethernet socket connections are opened at the time of the initial reading or writing access by a **VAL 3** program. The Ethernet socket connections are closed automatically when the **VAL 3** application is closed.

All instructions using a **sio** type variable not linked to an input/output declared in the system generate a runtime error. The default initialization value of **sio** type variables is an undefined link. The link of a **sio** variable can be initialized from another **sio** variable, from the robot **MCP**, or using **VAL 3 Studio** in **Stäubli Robotics Suite**.

3.7.2. OPERATORS

When the communication time out delay is reached on reading or writing the serial input/output, a runtime error is generated.

string < sio siOutput > = < string sText >	Writes successively on siOutput the sText characters Unicode UTF8 codes, followed by the end of string character, and returns sText .
num < sio siOutput > = < num nData item >	Writes on siOutput the closest integer to nData item , modulo 256 , and returns the value actually sent.
num < num nData > = < sio silInput >	Reads a byte on silInput and assigns nData with the byte value.
string < string sText > = < sio silInput >	Reads on silInput a string of Unicode UFT8 characters and affects sText with this string. The characters that are not supported by the string type are ignored. The string is completed when the end of string character is read, or when sText reaches the maximum size of a string (128 bytes). The end of string character is not copied into sText .

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter. `nLen=len(sString=silInput)` must be replaced with `sString=silInput; nLen=len(sString)`.

3.7.3. INSTRUCTIONS

void sioLink(sio& siVariable, sio siSource)

Function

The instructions links **siVariable** to the serial system input/output to which **siSource** is linked.

See also

void dioLink(dio& diVariable, dio diSource)

void aioLink(aio& aiVariable, aio aiSource)

num clearBuffer(sio siVariable)

Function

This instruction clears the **siVariable** reading buffer and returns the number of characters deleted

For an Ethernet socket connection, **clearBuffer** deactivates (closes) the socket. **clearBuffer** returns **-1** if the socket has already been deactivated.

A runtime error is generated if **siVariable** is not connected to a system serial link or Ethernet socket.

num sioGet(sio siInput, num& nData[])

Function

This instruction reads a single character or an array of characters from **siInput** and returns the number of characters read.

The reading sequence stops when the **nData** array is full or when the input reading buffer is empty.

For an Ethernet socket connection, **sioGet** tries first to open a connection if there is no open connection. When the timeout for input communication has been reached, **sioGet** returns **-1**. If the connection is open, but there is no data in the input reader buffer, **sioGet** waits until data is received or until the end of the timeout period has been reached.

A runtime error is generated if **siInput** is not linked to a system serial port or Ethernet socket, or if **nData** is not a **VAL 3** variable.

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

Function

This instruction writes a character or an array of characters to **siOutput** and returns the number of characters written.

Numerical values are converted before transmission into integers between **0** and **255**, taking the nearest integer modulo **256**.

For an Ethernet socket connection, **sioSet** tries first to open a connection if there is no open connection. When the end of the output communication waiting time has been reached, **sioSet** returns **-1**. The number of characters written can be less than the size of **nData** if a communication error is detected.

A runtime error is generated if **siOutput** is not linked to a system serial port or Ethernet socket.

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

Function

This instruction modifies a communication parameter of the specified serial input/output siChannel.

For serial lines, some parameters or parameter values may not be supported by the hardware: refer to the controller's manual.

The instruction returns:

0	The parameter is successfully modified
-1	The parameter is not defined
-2	The parameter value has not the expected type
-3	The parameter value is not supported
-4	The serial channel is not ready to apply the parameter change (stop it first)
-5	The parameter is not defined for this type of channel

The supported parameters are given by the table below:

Parameter name	Parameter type	Description
"port"	num	(For TCP client or server) TCP port
"target"	string	(For TCP client) IP address of the TCP server to reach, such as "192.168.0.254"
"clients"	num	(For TCP server) Maximum number of simultaneous clients on the server
"endOfString"	num	(For serial line, TCP client and server) ASCII code for the end of string character to be used with sio '=' operators (in range [0, 255])
"timeout"	num	(For serial line, TCP client and server) Maximum response time for the communication channel. 0 means no time out.
"baudRate"	num	(For serial line) Communication speed
"parity"	string	(For serial line) Parity control: "none", "even" or "odd"
"bits"	num	(For serial line) Number of bits per byte (5, 6, 7 or 8)
"stopBits"	num	(For serial line) Number of stop bits per byte (1 or 2)
"mode"	string	(For serial line) Communication mode: "rs232" or "rs422"
"flowControl"	string	(For serial line) Flow control: "none" or "hardware"
"nagle"	bool	(For TCP client or server) enable (default) or disable the nagle optimization. Disabling nagle optimization improves response time but increases network load.

Example

This program configures the main parameters of a serial line.

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

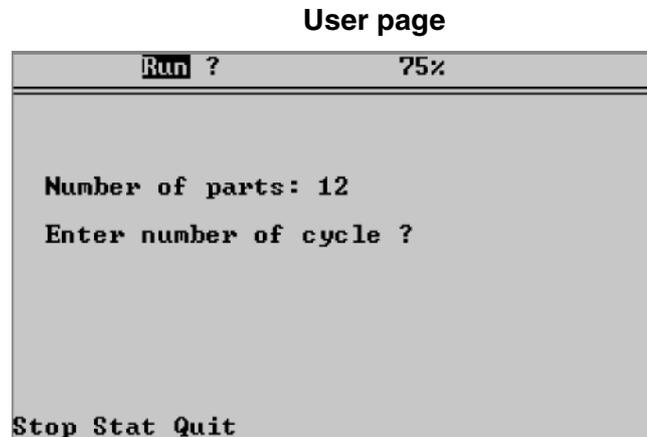

CHAPTER 4

USER INTERFACE

4.1. USER PAGE

In the **VAL 3** language, the user interface instructions are used to:

- display messages on a page of the manual control pendant (MCP) reserved for the application
- acquire keystrokes from the **MCP** keyboard



The user page has 14 lines of 40 columns. The last line is often used to create menus with the associated key. An additional line is available for a title display.

4.2. SCREEN TYPE

The user page context (displayed information and keystrokes) can be associated to a variable of the **screen** type. It is then easier to build and maintain several screens in an application, such as a production, a maintenance and a debugging screen.

It is possible to easily switch from one screen to another by pressing 'User-Down' or 'User-Up'. Pressing 'User-Shift' switches to the first screen, 'User-Shift-Up' to the last one.

The **screen** type takes a significant place in memory and can therefore not be used for local variables, unless the size of runtime memory for the application has been significantly increased.

4.2.1. SELECTING THE USER SCREEN

The **userPage()** instruction can be specified an optional **screen** variable as parameter. The current screen displayed on MCP switches then to the specified screen. If no **screen** parameter is specified, the current screen switches to the default user screen that is always defined.

4.2.2. WRITING TO A USER SCREEN

The instructions to write on the screen (**title()**, **gotoxy()**, **cls()**, **put()**, **putln()**) can be specified an optional **screen** variable as first parameter. The write operation is then done on the specified screen and does not affect the other screens. The modified screen may not be the screen being displayed. In that case the changes remain hidden until the current screen is changed with the **userPage()** instruction. If no **screen** parameter is specified, the modified screen is the default user screen that is always defined.

4.2.3. READING FROM A USER SCREEN

The instructions to get keystrokes (**get()**, **getKey()**, **isKeyPressed()**) can be specified an optional **screen** variable as first parameter. The input operation is then done on the specified screen: a keystroke can only be read by the screen being displayed on MCP. Other screens will not be affected. It is therefore possible to have several different screens waiting simultaneously for different keystrokes: only the current screen (being displayed) will be notified of effective keystrokes. If no **screen** parameter is specified, the impacted screen is the default user screen that is always defined.

4.3. INSTRUCTIONS

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

Function

This instruction displays on the **MCP** screen the specified user page, if any, or the default user page.

If the parameter **bFixed** is **true**, only the user page is accessible for the operator, except for the profile changing page that is accessible via the "Shift User" keyboard shortcut. When this page is displayed, it is possible to stop the application using the "Stop" key if the current user profile authorizes the action.

If the parameter is **false**, the other **MCP** pages become accessible again.

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

Function

This instruction places the cursor at the **(nX, nY)** coordinates on the specified user page, if any, or on the default user page. The coordinates of the top left-hand corner are **(0,0)** and those of the bottom right-hand corner are **(39, 13)**.

The **nX** column number is taken modulo **40**. The **nY** row number is taken modulo **14**.

See also

void cls(), void cls(screen scPage)

void cls(), void cls(screen scPage)

Function

This instruction clears the specified, or the default user page and sets the cursor to **(0,0)**.

See also

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

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

Function

This instruction modifies the display mode of the specified screen, if any, or the default screen. The new display mode does not affect the current display, but is applied to all new texts until a new text mode is defined. The supported modes are defined hereafter:

0	standard text mode (black on white background)
1	inverse video mode (white on black background)
2	flashing standard text mode
3	flashing inverse video mode

See also

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)`

Function

This instruction returns the length of **sText** on **MCP** display (number of columns needed to display **sText**).

For **ASCII** strings, the length on display is the number of characters in the string; `getDisplayLen()` is then identical to the `len()` instruction.

Some characters (Chinese) are displayed on two adjacent screen columns; `getDisplayLen()` is then greater than **sText** length, and can be used to control **sText** alignment on screen.

See also

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

Function

This instruction displays the specified **sText** or **nValue** (to **3** decimal places) at the cursor position on the specified user page, if any, or on the default user page. The cursor is then positioned on the character after the last character of the message displayed (**put** instruction), or on the first character of the next line (**putln** instruction).

At the end of a line, the display continues on the following line.

At the end of a page, the user page display moves up one line.

See also

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)

Function

This instruction changes the title of the specified user page, if any, or on the default user page.

The **title()** instruction does not change the current cursor position.

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)

Function

This instruction acquires a string, a number or a control panel key.

The parameter **sText** or **nValue** is displayed at the current cursor position and can be changed by the user. The entry is completed by pressing a menu key or the **Return** or **Esc** keys.

The instruction returns the code of the key used to end the entry.

Pressing **Return** or a menu key updates the **sText** or **nValue** variable. Pressing **Esc** does not change the variable.

If no parameter is passed, the **get()** instruction waits for the operator to press any key and returns the key code. The key that has been pressed is not displayed.

In all cases, the current position of the cursor is unaffected by the **get()** instruction.

Without Shift					With Shift						
3	Caps	Space			Move	3	Caps	Space			Move
283	-	32				283	-	32			
2	Shift	Esc	Help	Ret.	Run	2	Shift	Esc	Help	Ret.	Run
	282	-	255	-			270	282	-	255	
1	Menu	Tab	Up	Bksp	Stop	1	Menu	UnTab	PgUp	Bksp	Stop
	281	-	259	261			263	281	-	260	
	User	Left	Down	Right			User	Home	PgDn	End	
	-	264	266	268			-	265	267	269	

Menus (with or without **Shift**)

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

For standard keys, the code returned is the **ASCII** code of the corresponding character:

Without 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

With 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

With double 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

Example

This program reads a numeric value validated with the Return key:

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

See also

num getKey(), num getKey(screen scPage)

num getKey(), num getKey(screen scPage)

Function

This instruction acquires a keystroke from the control panel keyboard. It returns the code of the last key pressed since the last `getKey()` call, or -1 if no key has been pressed since then. A keystroke can only be detected when the specified user page, if any, or the default user page is displayed.

Unlike the `get()` instruction, `getKey()` returns immediately.

The key pressed is not displayed and the current cursor position remains unchanged.

Example

This program refreshes the display of the system clock until a key is pressed:

```
// First reset the code of the last key pressed
getKey()
while (getKey() == -1)
    gotoxy(0,0)
    put(toString("", clock()* 10))
    delay(0)
endWhile
```

See also

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)**

Function

This instruction returns the status of the key specified by its code (see `get()`), **true** if the key is pressed, otherwise **false**. A keystroke can only be detected when the specified user page, if any, or the default user page is displayed, except for the keys (1), (2) and (3) that are always detected.

See also

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)

Function

This instruction displays **sText** in a "popup" window above the current **MCP** window. This window remains displayed until it is confirmed by clicking on **Ok** in the menu or pressing the **Esc** key.

See also

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

bool logMsg(string sText)

Function

This instruction writes **sText** in the system history (error log). The message is saved with the current date and time. "USR" is added to the beginning of the string to label it as a user message. Some log messages may be lost if many messages are logged in the same second. The instruction returns then false so that the messages are logged later, if this is of importance.

Example

This program makes sure that the message is logged:

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

See also

void popUpMsg(string sText)

string getProfile()

Function

This instruction returns the name of the current user profile.

See also

num setProfile(string sUserLogin, string sUserPassword)

num setProfile(string sUserLogin, string sUserPassword)

Function

This instruction changes the current user profile (immediate effect).

The function returns:

- 0: The specified user profile is now effective
- 1: The specified user profile is not defined
- 2: The specified user password is not correct
- 3: 'staubli' is not allowed as user profile with this instruction
- 4: The current user profile is 'staubli' and cannot be changed with this instruction

See also

string getProfile()

string getLanguage()

Function

This instruction returns the current language of the robot controller.

Example

```
switch (getLanguage())  
  case "français"  
    sMessage="Attention!"  
  break  
  case "english"  
    sMessage="Warning!"  
  break  
  case "deutsch"  
    sMessage="Achtung!"  
  break  
  case "italiano"  
    sMessage="Avviso!"  
  break  
  case "español"  
    sMessage="¡Advertencia!"  
  break  
  default  
    sMessage="Warning!"  
  break  
endSwitch
```

See also

bool setLanguage(string sLanguage)

bool setLanguage(string sLanguage)

Function

This instruction modifies the current language of the robot controller: the specified language name sLanguage must match the name of a translation file on the controller. Refer to controller's manual to remove, or install additional languages on the robot controller.

Example

This program switches the robot language to Chinese:

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

See also

string getLanguage()

string getDate(string sFormat)

Function

This instruction returns the current date and/or time of the robot controller. The sFormat parameter specifies the format for the returned date. In this string, each occurrence of some keywords is replaced with the corresponding date or time value. The supported format keywords are listed in the table below:

Keyword	Description
%y	2-digits year (00-99), without century
%Y	4-digits year such as 2007
%m	Month (00-12)
%d	Day (00-31)
%H	Hour in 24-hour format (00-23)
%I	Hour in 12-hour format (01-12)
%p	A.M./P.M. indicator for 12-hour clock
%M	Minute (00-59)
%S	Seconds (00-59)

Example

This program displays date and hour in the format "January 01, 2007 13:45:23"

```
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  
  
// Display date and date in the form: "January 01, 2007 13:45:23"  
println(getDate(sMonth+" %d, %Y %H:%M:%S"))
```

CHAPTER 5

TASKS

5.1. DEFINITION

A task is a program that is running. An application can and usually will have several tasks running.

An application typically contains an arm movement task, an automation task, a user interface task, a safety signal monitoring task, communication tasks, etc.

A task is defined by the following elements:

- a name: a task identifier that is unique in the library or application
- a priority, or a period: a task sequencing parameter
- a program: the task entry (and exit) point
- a status: running or stopped
- the next instruction to be executed (and its context)

5.2. RESUMING AFTER A RUNTIME ERROR

When an instruction causes a runtime error, the task is stopped. The `taskStatus()` instruction is used to diagnose the runtime error. The task can then be resumed via the `taskResume()` instruction. If the runtime error can be corrected, the task can resume from the instruction line where it was stopped. Otherwise, it must be restarted from before or after that instruction line.

Starting and stopping the application

When an application starts, its `start()` program is executed in a task with the name of the application followed by '~', and with priority 10.

When an application stops, its `stop()` program is executed in a task with the name of the application preceded by '~', priority 10.

If a **VAL 3** application is stopped via the **MCP** user interface, the start task, if it still exists, is immediately destroyed. The `stop()` program is run next, then any remaining application tasks are deleted in the reverse order to that in which they were created, and finally libraries are unloaded from the memory.

5.3. VISIBILITY

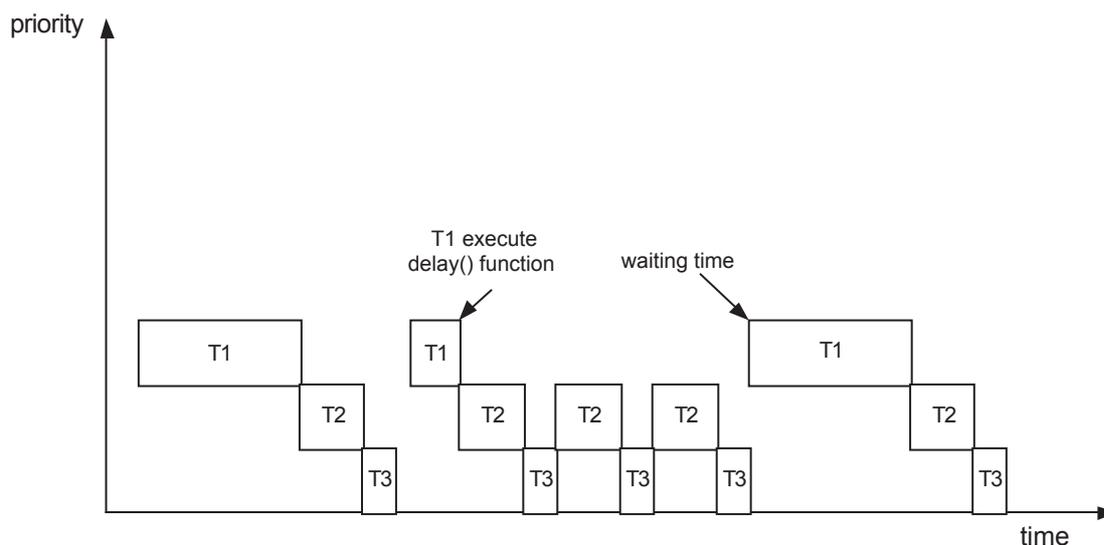
A task is visible only from within the program or library that created it. The instructions `taskSuspend()`, `taskResume()`, `taskKill()` and `taskStatus()` act on a task created by another library as if the task was not created. Two different libraries may therefore create tasks with the same name.

5.4. SEQUENCING

When several tasks of an application are running, they appear to run concurrently and independently. This is true if the whole application is observed over a sufficiently long period of time (about a second), but not true if its specific behaviour is examined over a short period of time.

In fact, as the system has only one processor, it can only execute one task at a time. Simultaneous execution is simulated by very fast sequencing of the tasks that execute a few instructions in turn before the system moves on to the next task.

Sequencing



VAL 3 task sequencing obeys the following rules:

1. The tasks are sequenced in the order in which they were created
2. During each sequence, the system attempts to execute a number of **VAL 3** instruction lines corresponding to the priority of the task.
3. When an instruction line cannot be terminated (runtime error, waiting for a signal, task stopped, etc.) the system moves on to the next **VAL 3** task.
4. When all **VAL 3** tasks have been completed, the system keeps some free time for lower priority system tasks (such as network communication, user screen refresh, file access), before a new cycle is started. The maximum delay between two sequential cycles is equal to the duration of the last sequencing cycle; but, most of the time, this delay is null because the system does not need it.

The **VAL 3** instructions that can cause a task to be sequenced immediately are as follows:

- **watch()** (condition wait timeout)
- **delay()** (timeout)
- **wait()** (condition waiting time)
- **waitEndMove()** (arm stop waiting time)
- **open()** and **close()** (arm stop waiting time followed by timeout)
- **get()** (keystroke waiting time)
- **taskResume()** (waits until the task is ready for restart)
- **taskKill()** (waits for the task to be actually killed)
- **disablePower()** (waits for power to be actually cut off)
- The instructions accessing the contents of the disk (**libLoad**, **libSave**, **libDelete**, **libList**, **setProfile**)
- The sio reading/writing instructions (operator =, **sioGet()**, **sioSet()**)
- **setMutex()** (waits for the Boolean mutex to be false)

5.5. SYNCHRONOUS TASKS

The sequence described above is the sequence of normal tasks, called asynchronous tasks, that are scheduled by the system so that they execute as fast as possible. It is sometimes necessary to schedule tasks at regular periods of time, for data acquisition or device control: such tasks are called synchronous tasks.

They are executed in the sequencing cycle by interrupting the current asynchronous task between two **VAL 3** lines. When the synchronous tasks have finished, the asynchronous task resumes.

The sequencing of the **VAL 3** synchronous tasks obeys the following rules:

1. Each synchronous task is sequenced exactly once per period of time specified at the task creation (for instance, once every 4 ms).
2. At each sequence, the system executes up to 3000 **VAL 3** instruction lines. It shifts to the next task when an instruction line cannot be completed immediately (runtime error, waiting for a signal, task stopped, ...). In practice, a synchronous task is often explicitly ended by using the "delay(0)" instruction to force the sequencing of the next task.
3. The synchronous tasks with same period are sequenced in the order in which they were created.

5.6. OVERRUN

If the execution of a **VAL 3** synchronous task takes longer than the specified period, the current cycle ends normally, but the next cycle is cancelled. This overrun error is signalled to the **VAL 3** application by setting the Boolean variable specified for this purpose at the task creation to "**true**". At the beginning of each cycle this Boolean variable thus shows whether the previous sequencing was carried out entirely or not.

5.7. INPUTS / OUTPUTS REFRESH

Inputs are refreshed before both the synchronous tasks and the asynchronous tasks are executed. In the same way, outputs are refreshed after both the synchronous tasks and the asynchronous tasks are executed.

WARNING:

It is not possible to specify which inputs / outputs are used by one task. As a consequence, each refresh is performed on all inputs / outputs. The refresh of inputs / outputs on Modbus, BIO board, MIO board, CIO board or AS-i bus are not controlled by the VAL 3 scheduler. They can be refreshed at any time during the sequencing of a VAL 3 task.

5.8. SYNCHRONIZATION

It is sometimes necessary to synchronize several tasks before they are executed.

If the amount of time required to execute each of the tasks is known beforehand, they can be synchronized by simply waiting for a signal generated by the slowest task. However, if it is not known which task is the slowest, it is necessary to use a more complex synchronizing mechanism for which an example of **VAL 3** programming is shown below.

Example

// Synchronization program for N tasks

The program `synchro(num& n, bool& bSynch, num nN)` hereafter must be called in each task to synchronize.

The variable `n` must be initialized to 0, `bSynch` to `false`, and `nN` to the number of tasks to synchronize.

```
begin
  n = n + 1
  // Task synchronization waiting instruction
  // makes sure all tasks are waiting here to resume operation
  wait((n==nN) or (bSynch==true))
  bSynch = true
  n = n - 1
  // Task release waiting instruction
  // makes sure all tasks have resumed operation to clear synch context
  wait((n==0) or (bSynch == false))
  bSynch = false
end
```

5.9. SHARING RESOURCES

When several tasks use the same system or cell resource (global datas, screen, keyboard, robot, etc.). it is important to ensure that there is no conflict between them.

A mutual exclusion (**'mutex'**) mechanism that protects a resource by allowing it to be accessed by only one task at a time can be used for this purpose. An example of mutex programming in **VAL 3** is shown below.

Example

This program display (num c) fills the screen with the same chars, making sure no other task is writing to the screen with the same program at the same time. bScreen must be initialized to **false**.

```
begin
  // make sure only one task accesses the screen at one time
  setMutex(bScreen)
  c=c%10
  // fill the screen with chars
  for y=0 to 13
    gotoxy(x,y)
    put(c)
  endFor
endFor
  // wait for screen refresh
  delay(0.2)
  // let other tasks access the screen now
  bScreen=false
end
```

5.10. INSTRUCTIONS

`void taskSuspend(string sName)`

Function

This instruction suspends the execution of the **sName** task.

If the task is already **STOPPED**, the instruction has no effect.

A runtime error is generated if **sName** does not correspond to any **VAL 3** task, or corresponds to a **VAL 3** task created by another library.

See also

`void taskResume(string sName, num nSkip)`

`void taskKill(string sName)`

`void taskResume(string sName, num nSkip)`

Function

This instruction resumes the execution of the **sName** task on the line located **nSkip** instruction lines before or after the current line.

If **nSkip** is negative, the program resumes before the current line. If the task status is not **STOPPED**, the instruction has no effect.

A runtime error is generated if **sName** does not correspond to a **VAL 3** task, corresponds to a **VAL 3** task created by another library, or if there is no instruction line at the specified **nSkip**.

See also

`void taskSuspend(string sName)`

`void taskKill(string sName)`

void taskKill(string sName)

Function

This instruction suspends and then deletes the **sName** task. When the instruction has been executed, the **sName** task is no longer present in the system.

If there is no **sName** task, or if the **sName** task was created by another library, the instruction has no effect.

See also

void taskSuspend(string sName)

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

void setMutex(bool& bMutex)

Function

This instruction waits for the **bMutex** variable to be false, then set it to true.

This instruction is required to use a Boolean variable as a mutual exclusion mechanism for protecting shared resources (see chapter 5.9).

string help(num nErrorCode)

Function

This instruction returns the description of the task runtime error code specified with the **nErrorCode** parameter. The description is given in the current controller's language.

Example

This program checks if the "robot" task is in error, and displays the error code to the operator if any.

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

num taskStatus(string sName)

Function

This instruction returns the current status of the **sName** task, or the task runtime error code if the latter is in error condition:

Code	Description
-1	There is no task sName created by the current library or application
0	The task sName is suspended without runtime error (taskSuspend() instruction or debug mode)
1	The task sName created by the current library or application is running
10	Invalid numerical calculation (division by zero).
11	Invalid numerical calculation (e.g. ln(-1))
20	Access to an array with an index that is larger than the array size.
21	Access to an array with a negative index.
29	Invalid task name. See taskCreate() instruction.
30	The specified name does not correspond to any VAL 3 task.
31	A task with the same name already exists. See taskCreate instruction.
32	Only 2 different periods for synchronous tasks are supported. Change scheduling period.
40	Not enough memory space available.
41	Not enough memory space to run the task. See the run memory size.
60	Maximum instruction run time exceeded.
61	Internal VAL 3 interpreter error
70	Invalid instruction parameter. See the corresponding instruction.
80	Uses data or a program from a library not loaded in the memory.
81	Incompatible kinematic: Use of a point/joint/config that is not compatible with the arm kinematic.
82	The reference frame or tool of a variable belongs to a library and is not accessible from the variable's scope (library not declared in the variable's project, or reference variable is private).
90	The task cannot resume from the location specified. See taskResume() instruction.
100	The speed specified in the motion descriptor is invalid (negative or too great).
101	The acceleration specified in the motion descriptor is invalid (negative or too great).
102	The deceleration specified in the motion descriptor is invalid (negative or too great).
103	The translation velocity specified in the motion descriptor is invalid (negative or too great).
104	The rotation velocity specified in the motion descriptor is invalid (negative or too great).
105	The reach parameter specified in the movement descriptor is invalid (negative).
106	The leave parameter specified in the movement descriptor is invalid (negative).
122	Attempt to write in a system input.
123	Use of a dio, aio or sio input/output not connected to a system input/output.
124	Attempt to access a protected system input/output
125	Read or write error on a dio , aio or sio (field bus error)
150	Cannot run this movement instruction: a previous movement request could not be completed (point out of reach, singularity, configuration problem, etc.)
153	Movement command not supported
154	Invalid movement instruction: target out of reach, or check the movement descriptor.
160	Invalid flange tool coordinates
161	Invalid world tool coordinates
162	Use of a point without a reference frame. See Definition.
163	Use of a frame without a reference frame. See Definition.
164	Use of a tool without reference tool. See Definition.
165	Invalid frame or reference tool (global variable linked to a local variable)
250	No runtime licence for this instruction, or demo licence is over.

See also

void taskResume(string sName, num nSkip)

void taskKill(string sName)

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

Function

This instruction creates and starts up the **sName** task.

sName must contain **1** to **15** characters selected from "**a..zA..Z0..9_**". There must not be another task with the same name created by the same library.

Execution of **sName** begins with a call to **program** using the parameters specified. It is not possible to use a local variable for a parameter passed by reference, to make sure that the variable is not deleted before the task is completed.

The task ends by default with the last instruction line of **program**, or earlier, if it is deleted explicitly.

nPriority must be between **1** and **100**. When the task is sequenced, the system executes a number of instruction lines corresponding to the **nPriority**, or fewer if a blocking instruction is encountered (see the chapter entitled Sequencing).

A runtime error is generated if the system does not have enough memory to create the task, if **sName** is not valid or already in use in the same library, or if **nPriority** is not valid.

Example

```
// start a new task to read a message
taskCreate "t1", 10, read(sMessage)
// waits for the end of task t1
wait(taskStatus("t1") == -1)
// Use the message
println(sMessage)
```

See also

void taskSuspend(string sName)

void taskKill(string sName)

num taskStatus(string sName)

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

Function

This instruction creates and starts a synchronous task.

The execution of the task starts with the call of the specified program with the specified parameters. It is not possible to use a local variable for a parameter passed by reference, to make sure that the variable is not deleted before the task is completed.

A runtime error is generated if the system doesn't have enough memory to create the task, or if one or more parameters are invalid.

For a detailed description of synchronous tasks (see chapter 5.5).

Parameters

string sName	Name of the task to create. It must contain 1 to 15 characters selected from "_a-zA..Z0..9". There cannot be another task with the same name belonging to the same application or library.
num nPeriod	Period of the task to create (s). The specified value is rounded down to a multiple of 4 ms (0.004 seconds). Any positive period is supported, but the system supports only two different periods of synchronous tasks at the same time.
bool& bOverrun	Boolean variable to signal overrun errors. Only global variables are supported, to make sure that the variable is not deleted before the task.
program	Name of the VAL 3 program to call when the task is started, with its parameters between parenthesis.

Example

```
// Create a supervisor task scheduled every 20 ms  
taskCreateSync "supervisor", 0.02, bSupervisor, supervisor()
```

void wait(bool bCondition)

Function

This instruction puts the current task on hold until **bCondition** is **true**.

The task remains **RUNNING** during the waiting time. If **bCondition** is **true** at the first evaluation, the task in question is executed immediately (the next task is not sequenced).

See also

void delay(num nSeconds)

bool watch(bool bCondition, num nSeconds)

void delay(num nSeconds)

Function

This instruction puts the current task on hold for **nSeconds**.

The task remains **RUNNING** during the waiting time. If **nSeconds** is negative or null, the system sequences the next **VAL 3** task immediately.

Example

This program loops to get a key, taking care not to use unnecessary CPU resource:

```
// First reset the code of the last key pressed
getKey()
while (getKey() == -1)
  gotoxy(0,0)
  put(toString("", clock()* 10))
  // let another task perform its operation immediately
  delay(0)
endWhile
```

See also

num clock()

bool watch(bool bCondition, num nSeconds)

num clock()

Function

This instruction returns the current value of the internal system clock expressed in seconds.

The internal system clock is accurate to within one millisecond. It is initialized at **0** when the controller is started up and is thus unrelated to calendar time.

Example

To compute the execution delay between two instructions, store the clock value before the first instruction:

```
nStart=clock()
```

After the last instruction, compute the execution delay with:

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

See also

void delay(num nSeconds)

bool watch(bool bCondition, num nSeconds)

bool watch(bool bCondition, num nSeconds)

Function

This instruction puts the current task on hold until **bCondition** is **true** or **nSeconds** seconds have elapsed.

Returns **true** if the waiting time ends when **bCondition** is **true**, otherwise returns **false** when the waiting time ends because the time has expired.

The task remains **RUNNING** during the waiting time. If **bCondition** is **true** at the first evaluation, the same task is executed immediately, otherwise the system sequences the other **VAL 3** tasks (even if **nSeconds** is up to and including **0**).

Example

This program waits for a signal and displays an error message after 20 s.

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

See also

void delay(num nSeconds)

void wait(bool bCondition)

num clock()

CHAPTER 6

LIBRARIES

6.1. DEFINITION

A **VAL 3** library is a **VAL 3** application that has variables or programs that can be reused by another application or by other **VAL 3** libraries.

Being a **VAL 3** application, a **VAL 3** library comprises the following components:

- a set of **programs**: the **VAL 3** instructions to be executed
- a set of **global variables**: the library data
- a set of **libraries**: the external instructions and variables used by the library

When a library is being run, it can also contain:

- a set of **tasks**: The programs that are specific to the library being run

All applications can be used as a library and all libraries can be used as an application, if the **start()** and **stop()** programs are defined in them.

6.2. INTERFACE

A library's global programs and variables are either public or private. Only global programs and variables that are public are accessible outside the library. Private programs and global variables can only be used by the library programs.

All the public global programs and variables from a library form its interface: a number of different libraries can have the same interface, as long as their public programs and variables use the same names.

The tasks created by a library program are always private, i.e. they can only be accessed by that library.

6.3. INTERFACE IDENTIFIER

To use a library, an application needs to first declare an identifier assigned to it, and then request, in a program, that the library be loaded into the memory under that identifier.

The identifier is assigned to the library interface and not to the library itself. Any library presenting the same interface can then be loaded under that identifier. This mechanism can be used, for example, to define a library for every possible part of an application, and then load only the part currently being processed by each cycle.

6.4. CONTENT

A library does not have any required content: it can contain only programs, or only variables, or both.

Library content is accessed by writing the identifier's name followed by ':' in front of the name of the library program or data, for example:

```
// Load the "article_7" library under the "article" identifier
article:libLoad("article_7")
// Display as title the content of the 'sName' variable of the article_7 library
title(article:sName)
// Call the init() program for the current article
call article:init()
```

Accessing the content of a library that has not yet been loaded into the memory causes a runtime error.

6.5. ENCRYPTION

VAL 3 supports encrypted libraries, based on the widely used ZIP compression & encryption tools.

An encrypted library is a standard ZIP file of the content of the library directory (caution: advanced 128-bit and 256-bit AES encryption is not supported). The name of the zip file must have the '.zip' extension and have less than 15 characters (including extension). To have a robust encryption, the ZIP password should have more than 10 characters and should not be found in a dictionary.

Secret password, public password

The **VAL 3** interpreter must have access to the secret ZIP password to load an encrypted library; for this, a PC tool is provided with **Stäubli Robotics Suite** to encode the secret ZIP password into a public **VAL 3** password. The public **VAL 3** password makes it possible to use the encrypted library in a **VAL 3** program. But the library content remains secret because the ZIP password cannot be computed from the **VAL 3** password.

Project encryption

It is not possible to directly encrypt the start application on the controller. A complete application can be encrypted by:

- Declaring its start() program as public.
- Creating another application that simply loads the encrypted application as a library and calls its start() program.

6.6. LOADING AND UNLOADING

When a **VAL 3** application is opened, all the libraries declared are analysed to build the corresponding interfaces. This step does not load the libraries into the memory.

CAUTION:

Circular references between libraries are not supported. If library A uses library B, library B cannot use library A.

When a library is loaded, its global datas are initialized and its programs checked to detect any syntax errors. When several different library identifiers load the same library on disk, they share the same library in memory. The library is then loaded only once and reused by all identifiers. In the example below, lib1 and lib2 use the same data in memory.

```
lib1:libLoad("appData")
lib1:sText = "lib1"
lib2:libLoad("appData")
// The change on lib2:sText applies here also to lib1:sText
lib2:sText = "lib2"
```

It is not necessary to unload a library, this is done automatically when the application ends, or when a new library is loaded to replace another one.

When a **VAL 3** application is stopped via the **MCP** user interface, the **stop()** program is run first, then all the application tasks, and its libraries, if any are left, are destroyed.

Access path

The **libLoad()**, **libSave()** and **libDelete()** instructions use a library access path, specified as a character string. An access path comprises an (optional) root, an (optional) path and a library name, in the following format:

```
root://Path/Name
```

The root specifies the file medium: **"Floppy"** for a diskette, **"USB0"** for a device on a **USB** port (stick, floppy disk), **"Disk"** for the controller's flash disk, or the name of an **Ftp** connection defined on the controller for a network access.

By default, the root is **"Disk"** and the path is blank.

Example

```
// load library "article_1" on Disk Equivalent to "Disk://article_1"
article:libLoad("article_1")
// Save library on USB device
article:libSave("USB0://articles/article_1")
// Load the default article defined within the current application
article:libLoad("./defaultArticle")
```

Error codes

The **VAL 3** library handling functions never generate runtime errors but they send back an error code used to check the instruction result and troubleshoot any problems that may arise.

Code	Description
0	No error
10	The library identifier has not been initialized by <code>libLoad()</code> .
11	Library loaded, but public interface does not match. A runtime error 80 will result if the VAL 3 program tries to access missing items. See libExist instruction or libExist instructions.
12	Cannot load the library: the library contains invalid data or programs, or, for an encrypted library, the specified password is not correct.
13	Cannot unload the library: The library is being used by another task.
14	Cannot unload the library: The library owns a running VAL 3 task. All tasks created by VAL 3 programs from the library must be completed before the library is unloaded.
20	File access error: invalid path root.
21	File access error: invalid path.
22	File access error: invalid name.
23	Encrypted library expected. Library is not or badly encrypted.
>=30	File reading/writing error.
31	Cannot save the library: the path specified already contains a library. To replace a library on disk, first delete it with <code>libdelete()</code> .
32	Driver reports "Device not found"
33	Driver reports "Device error"
34	Driver reports "Device timeout"
35	Driver reports "Device write protected"
36	Driver reports "Disk not present"
37	Driver reports "Disk not formatted"
38	Driver reports "Disk full"
39	Driver reports "File not found"
40	Driver reports "Read only file"
41	Driver reports "Connection refused"
42	Driver reports "Ftp server does not answer"
43	Driver reports "Ftp kernel error"
44	Driver reports "Ftp parameters error"
45	Driver reports "Ftp access error"
46	Driver reports "Ftp disk full"
47	Driver reports "Invalid Ftp user login"
48	Driver reports "Ftp connection not defined"

6.7. INSTRUCTIONS

num identifier:libLoad(string sPath)

num identifier:libLoad(string sPath, string sPassword)

Function

This instruction initializes the library identifier by loading the library program and variables into the memory following the specified **sPath**. The specified (optional) **sPassword** parameter is used as decryption key for encrypted libraries. The specified **sPassword** must be the public **VAL 3** password computed from the secret ZIP password of the encrypted library (see chapter 6.5, page 98).

The instruction returns **0** after successful loading, a library loading error code if there are still tasks running that were created by the library, if the library access path is invalid, if the library contains syntax errors or if the library specified does not correspond to the interface declared for the identifier.

See also

num identifier:libSave(), num libSave()

num identifier:libSave(), num libSave()

Function

This instruction saves the variables and programs assigned to the library's identifier. If **libSave()** is called without an identifier, the application containing the **libSave()** instruction is saved. If a parameter is specified, the content is saved via the specified **sPath**. Otherwise, the content is saved via the path specified on loading.

The instruction returns **0** if the content has been saved, an error code if the identifier has not been initialized, if the path is invalid, if a writing error occurs or if the path specified already contains a library.

CAUTION:

Some devices such as the controller's flash disk support only a limited number of write access. If a frequent use of **libSave()** is made in a program (once or more every minute), it must be made on a device supporting it.

See also

num libDelete(string sPath)

num libDelete(string sPath)

Function

This instruction deletes the library located in the specified **sPath**.

The instruction returns **0** if the specified library does not exist or has been deleted, and an error code if the identifier has not been initialized, if the path is invalid or if a writing error occurs.

See also

num identifier:libSave(), num libSave()

string identifier:libPath(), string libPath()

string identifier:libPath(), string libPath()

Function

This instruction returns the access path of the library associated with the identifier, or that of the calling application if no identifier is specified.

See also

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

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

Function

This instruction lists the contents of the specified **sPath** path in the **sContents** array. Returns **true** if the **sContents** array lists the full result, and **false** if the array is too small to hold the full list.

All elements of the **sContents** array are first initialized to "" (empty string). After **libList()** is executed, the end of the list is therefore found by searching the first empty string in the **sContents** array.

If **sContents** is a global variable, the size of the array is automatically enlarged as required to enable storage of the full result.

See also

string identifier:libPath(), string libPath()

bool identifier:libExist(string sSymbolName)

Function

The **libExist** instruction tests whether a symbol (a global data or program) is defined in a library. It returns true if the symbol exists and is accessible (public), else false.

The symbol name for a program must be appended with "()": "mySymbol" denotes a data name, whereas "mySymbol()" denotes a program name.

The **libExist** instruction is useful to test if an input/output is defined on a controller; it is also helpful to handle the evolution of a library's interface, and adapt its use depending if it is a newer or older version of the interface.

Example

This example tests the interface of a library.

```
// Load part library
nLoadCode = part:libLoad(sPartPath)
// part:sVersion was not defined in the first version of the library
// Test if this library defines it
if (nLoadCode==0) or (nLoadCode==11)
  if (part:libExist("sVersion")==false)
    // initial version
    sLibVersion = "v1.0"
  else
    sLibVersion = part:sVersion
  endif
endif
```

This program calls the 'init' program of the 'protocol' library, if any:

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

See also

bool isDefined(*)

CHAPTER 7

USER TYPE

7.1. DEFINITION

A user type is a structured type defined within a VAL 3 application, where it can be used as a standard type. A user type combines simple, structured or even other user types into a new data type. User types increase the abstraction level of programs and make them easier to understand, develop and maintain. They require however a higher initial design effort to identify the adequate types that best fit the application constraints.

A user type is a set of fields, where each field consists in:

- a name: a character string
- a data type (simple, structured, or user type)
- a data container (element, array or collection)
- a default set of values

The fields of the VAL 3 standard types always use an element container (with a single value). The fields of user types may use array or collection container and therefore contain several values. The default value for the field defines the default number of elements in the field container, and for each of these elements, its default value. In a variable defined with a user type, you may change at any time not only the values of its fields, but also the number of elements of the field containers.

7.2. CREATION

The fields of a user type have the same characteristics than the global data of a VAL 3 application. This is why the creation of a new user type just consists in selecting a VAL 3 application and associating it with a name.

- The set of the public global data in the selected application defines the set of the user type's fields, with their default value.
- The name defines the name to be used for the new user type in the application where it is defined.

The private data and the programs of the application used as type definition are ignored in the user type.

Once a new user type is defined in an application, it is possible to create data of this type. The resulting application can then also be used as type definition.

7.3. USE

The fields of a user type variable can be accessed using a '.' followed by the field name: `userVariable.field1.field2` refers to the value of the 'field2' field of the 'field1' field of the data `userVariable`. The creation or deletion of elements in a field container are supported like for a variable, with the [insert\(\)](#), [delete\(\)](#), [append\(\)](#) or [resize\(\)](#) instructions. When a new element is created, each field is assigned a default value consisting in the elements and their values defined in the application used as type definition.

CAUTION:
The fields of type point, tool or frame are not linked by default.

The '=' operator is always defined between two variables of the same user type. After the '=' operator is executed, the left variable is a copy of the right variable: the fields have the same number of elements in their container, and the elements the same value.

CHAPTER 8

ROBOT CONTROL

This chapter lists the instructions that allow access to the status of the various parts of the robot.

8.1. INSTRUCTIONS

void disablePower()

Function

This instruction cuts off the power supply to the arm and waits until the power supply has actually been cut off. If the arm is moving, it stops abruptly on its trajectory before the power is switched off.

See also

void enablePower()
bool isPowered()

void enablePower()

Function

In remote mode, this instruction switches the arm power on.

This instruction does not have any effects in local, manual or test modes, or when the power supply is being switched off. It does generate a message in the log, so avoid repeated undelayed attempts to enable power.

Example

```
// Switches on the power and waits for the arm power to be switched on
enablePower()
  if (watch(isPowered(), 5) == false)
    putln("Arm power supply cannot be switched on")
  endif
```

See also

void disablePower()
bool isPowered()

bool isPowered()

Function

This instruction returns the power status of the arm:

true: the arm is under power

false: the arm power is switched off, or is being switched on

bool isCalibrated()

Function

This instruction returns the calibration status of the robot:

true: all the robot axes are calibrated

false: at least one robot axis is not calibrated

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

Function

This instruction returns the current working mode of the robot:

Mode	Status	Working mode	Status
0	0	Invalid or transitional	-
1	0	Manual	Programmed movement
	1		Connection movement
	2		Joint jogging
	3		Cartesian (Frame jogging)
	4		Tool jogging
	5		To point (Point jogging)
	6		Hold
2	0	Test	Programmed movement (< 250 mm/s)
	1		Connection movement (< 250 mm/s)
	2		Fast programmed movement (> 250 mm/s)
	3		Hold
3	0	Local	Move (programmed movement)
	1		Move (connection movement)
	2		Hold
4	0	Remote	Move (programmed movement)
	1		Move (connection movement)
	2		Hold

num esStatus()

Function

This instruction returns the status of the E-Stop circuit:

Code	Status
0	No E-Stop (E-Stop chain is closed).
1	No E-Stop any more, waiting validation. In manual mode, the MCP must be on its support to enable power.
2	E-Stop open, or waiting for the correction of a hardware fault.

See also

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

bool safetyFault(string& sSignalName)

bool safetyFault(string& sSignalName)

Function

This instruction returns true if a hardware fault on the safety chain must be fixed and acknowledge. In that case, sSignalName is updated with the name of the faulty signal.

See also

num esStatus()

num ioBusStatus(string& sErrorDescription[])

Function

This instruction checks the status of the field bus devices and returns the number of devices in error, zero if no device is in error. For each device in error, a text description is added in the sErrorDescription string array. The format of the error description is: 'statusValue:deviceName\moduleName'.

If sErrorDescription is a global variable, the size of the array is automatically enlarged as required to enable storage of the full result.

The status value is an error number that depends on the field bus device and protocol.

See also

num ioStatus(dio dilInputOutput, string& sDescription, string& sPhysicalPath)

num ioStatus(aio dilInputOutput, string& sDescription, string& sPhysicalPath)

num getMonitorSpeed()

Function

This instruction returns the current monitor speed of the robot (in the range [0, 100]).

Example

This program, to be called in a specific task, checks that the first robot cycle is done at low speed:

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

See also

num setMonitorSpeed(num nSpeed)

num setMonitorSpeed(num nSpeed)

Function

This instruction modifies the current monitor speed of the robot. [setMonitorSpeed\(\)](#) is always able to reduce the monitor speed. To increase it, [setMonitorSpeed\(\)](#) is effective only if the robot is in remote working mode and if the operator does not have access to the monitor speed (when the current user profile does not allow use of the speed buttons, or the MCP has been disconnected).

It returns 0 if the monitor speed has been successfully modified, else a negative error code:

Code	Description
-1	The robot is not in remote working mode
-2	The monitor speed is under the control of the operator: change the current user profile to remove operator access to monitor speed
-3	The specified speed is not supported: it must be in the range [0, 100]

See also

num getMonitorSpeed()

string getVersion(string sComponent)

Function

This instruction returns the version of different hardware and software components of the robot controller. The table below lists the supported components and, for each, the format of the returned value:

Component	Description
"VAL 3"	Controller VAL 3 version, such as "s7.0 - Jun 18 2010 - 16:01:17"
"ArmType"	Type of the arm attached to the controller, such as "tx90-S1" or "rs60-S1-D20-L200"
"Tuning"	Version of the arm tuning, such as "R3"
"Mounting"	Arm mounting, such as "floor", "wall" or "ceiling"
"ControllerSN"	Serial number of the controller, such as "F07_12R3A1_C_01"
"ArmSN"	Serial number of the arm, such as "F07_12R3A1_A_01"
"Starc"	Version of the Starc firmware package (CS8C), such as "1.16.3 - Sep 27 2007 - 16:01:17"
"Name of the licence"	Status of the controller software license: "" (not installed or demo delay expired), "demo" or "enabled" The names of the installed controller licenses (such as "alter", "compliance", "remoteMcp", "oemLicence", "plc", "testMode", "mcpMode"...) are listed in the Control Panel of the robot pendant

Example

```

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

```

See also

string getLicence(string sOemLicenceName, string sOemPassword)

CHAPTER 9

ARM POSITIONS

9.1. INTRODUCTION

This chapter describes the **VAL 3** data types used to program the arm positions used in a **VAL 3** application. Two position types are defined in **VAL 3**: joint positions (**joint** type) that give the angular position of each revoluted axis and the linear position of each linear axis, and Cartesian points (**point** type) that give the Cartesian position of the tool center point at the end of the arm relative to a reference frame.

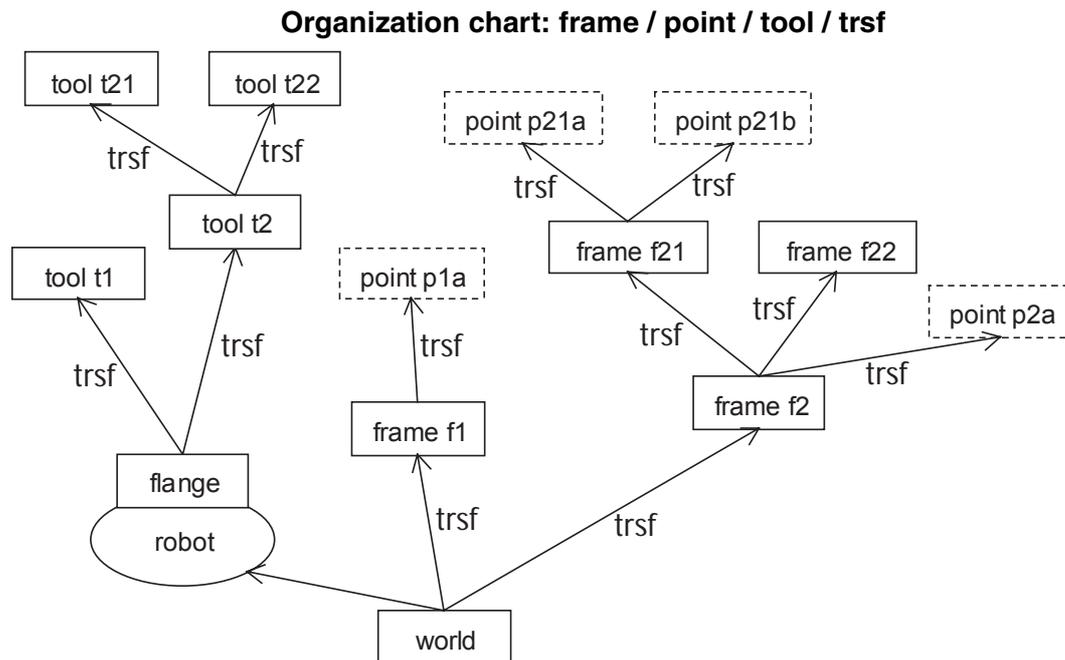
The **tool** type describes a tool and its geometry used to position and control the speed of the arm; it describes also how to activate the tool (digital output, delay).

The **frame** type describes a geometric reference frame. The use of frames makes geometrical point manipulation much simpler and more intuitive.

The **trsf** type describes a geometric transformation. It is used by the **tool**, **point** and **frame** types.

Finally, the **config** type describes the more advanced concept of arm configuration.

The relationships between these various types can be summarized as follows:



9.2. JOINT TYPE

9.2.1. DEFINITION

A joint location (**joint** type) defines the angular position of each revoluted axis and the linear position of each linear axis.

The **joint** type is a structured type, with the following fields, in this order:

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 ... (one field for each axis)

These fields are expressed in degrees for the rotary axes, and in millimeters or inches for the linear axes. The origin of each axis is defined by the type of arm used.

By default, each field of a **joint** type variable is initialized at the value **0**.

9.2.2. OPERATORS

In ascending order of priority:

<code>joint <joint& jPosition1> = <joint jPosition2></code>	Assigns jPosition2 to the jPosition1 variable field by field and returns jPosition2 .
<code>bool <joint jPosition1> != <joint jPosition2></code>	Returns true if a jPosition1 field is not equal to the corresponding jPosition2 field, to within the accuracy of the robot, otherwise it returns false .
<code>bool <joint jPosition1> == <joint jPosition2></code>	Returns true if each jPosition1 field is equal to the corresponding jPosition2 field, to within the accuracy of the robot, otherwise it returns false .
<code>bool <joint jPosition1> > <joint jPosition2></code>	Returns true if each jPosition1 field is greater than the corresponding jPosition2 field, otherwise it returns false .
<code>bool <joint jPosition1> < <joint jPosition2></code>	Returns true if each jPosition1 field is less than the corresponding jPosition2 field, otherwise it returns false . Caution: jPosition1 > jPosition2 is not the same as !(jPosition1 < jPosition2)
<code>joint <joint jPosition1> - <joint jPosition2></code>	Returns the difference, field by field, between jPosition1 and jPosition2 .
<code>joint <joint jPosition1> + <joint jPosition2></code>	Returns the sum, field by field, of jPosition1 and jPosition2 .

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter.

9.2.3. INSTRUCTIONS

`joint abs(joint jPosition)`

Function

This instruction returns the absolute value of a joint **jPosition**, field by field.

Details

The absolute value of a joint, with the ">" or "<" joint operators, is useful to compute easily a distance between a joint position and a reference position.

Example

```
jReference = {90, 45, 45, 0, 30, 0}
jMaxDistance = {5, 5, 5, 5, 5, 5}
j = herej()
// Checks that all the axis are less than 5 degrees from the reference
if(!(abs(j - jReference) < jMaxDistance))
  popUpMsg("Move closer to the marks")
endif
```

See also

Operator < (joint)

Operator > (joint)

joint herej()

Function

This instruction returns the current arm joint position.

When arm power is on, the returned value is the position sent to the amplifiers by the controller, and not the position read from the axis encoders.

When arm power is off, the returned value is the position read from the axis encoders ; because of noise in the encoder measurements, the position may vary a bit even when the arm is stopped.

The controller joint position is refreshed every 4 ms.

Example

```
//Wait until the arm is near the reference position, with a 60 s time out
bStart = watch(abs(herej() - jReference) < jMaxDistance, 60)
if bStart==false
  popUpMsg("Move closer to the start position")
endIf
```

See also

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

bool isInRange(joint jPosition)

Function

This instruction tests if a joint position is within the software joint limits of the arm.

When the arm is out of the software joint limits (after a maintenance operation), it is not possible to move the arm with a **VAL 3** application, only manual moves are possible (with the move direction restricted to moving it toward the limits).

Example

```
// Check if the current position is within the joint limits
if isInRange(herej())==false
  putLn("Please place the arm within its workspace")
endIf
```

See also

joint herej()

void setLatch(dio dilnInput) (CS8C only)

Function

This instruction enables the robot position latch on the next rising edge of the input signal.

Details

The robot position latching is a hardware feature that is supported only by the fast inputs of the CS8C controller (fln0, fln1).

The detection on the rising edge of the input signal is guaranteed only if the signal remains low during at least 0.2 ms before the rising edge, and high during at least 0.2 ms after the rising edge.

CAUTION:

The latch is enabled only after some time (between 0 and 0.2 ms) after the setLatch instruction is executed. You may need to add a delay(0) instruction after setLatch to make sure the latch is effective before the next VAL 3 instruction is executed.

Runtime error 70 (invalid parameter value) is generated if the specified digital input does not support robot position latching.

See also

bool getLatch(joint& jPosition) (CS8C only)

bool getLatch(joint& jPosition) (CS8C only)

Function

This instruction reads the last latched robot position.

The function returns true if there is a valid latched position to read. If a latch is pending, or if latching has never been enabled, the function returns false and the position is not updated.

getLatch returns the same latched position until a new latch is enabled with the setLatch instruction.

The arm position is refreshed in the CS8C controller every 0.2 ms; the latched position is the position of the arm between 0 and 0.2 ms after the rising edge of the fast input.

Example

```
setLatch(diLatch)
// Wait for setLatch to be effective before using getLatch
delay(0)
// Wait for a latched position during 5 seconds.
bLatch = watch(getLatch(jPosition)==true, 5)
if bLatch==true
    println("Successful position latch")
else
    println("No latch signal was detected")
endif
```

See also

void setLatch(dio dilnInput) (CS8C only)

joint herej()

9.3. TRSF TYPE

9.3.1. DEFINITION

A transformation (**trsf** type) defines a position and / or orientation change. It is the mathematical composition of a translation and a rotation.

A transformation itself doesn't represent a position in space, but can be interpreted as the position and orientation of a Cartesian point or frame relative to another frame.

The **trsf** type is a structured type whose fields are, in this order:

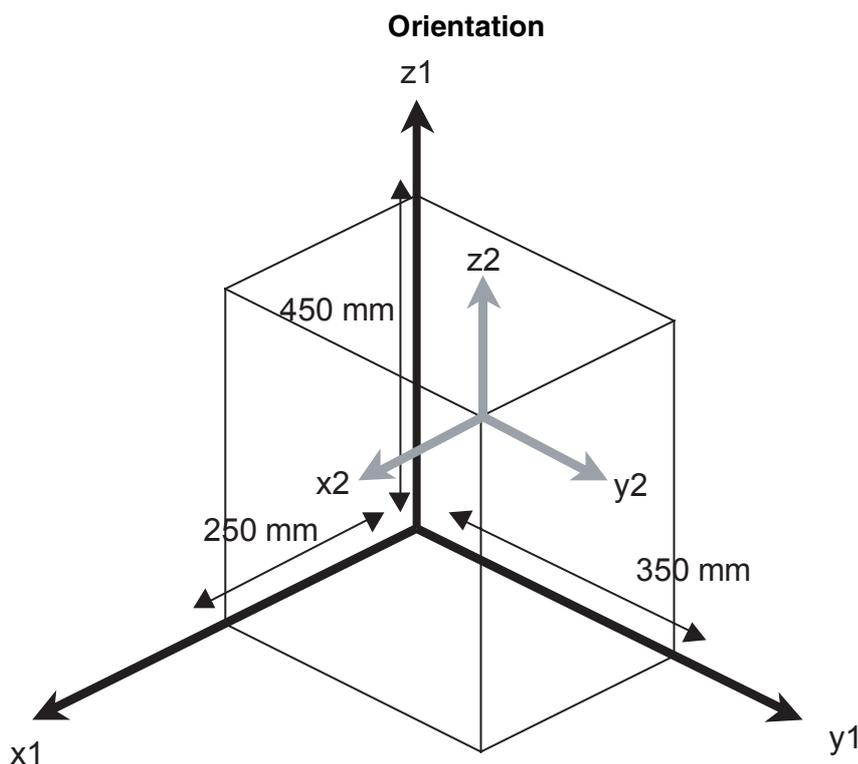
num x	Translation along the x axis
num y	Translation along the y axis
num z	Translation along the z axis
num rx	Rotation around the x axis
num ry	Rotation around the y axis
num rz	Rotation around the z axis

The **x**, **y** and **z** fields are expressed in the unit of length of the application (millimeter or inch, see the chapter entitled Unit of length). The **rx**, **ry** and **rz** fields are expressed in degrees.

The **x**, **y** and **z** coordinates are the Cartesian coordinates of the translation (or the position of a point or frame in the reference frame). When **rx**, **ry** and **rz** are zero, the transformation is a translation without change of orientation.

When a **trsf** type variable is initialized, its default value is **{0,0,0,0,0,0}**.

9.3.2. ORIENTATION

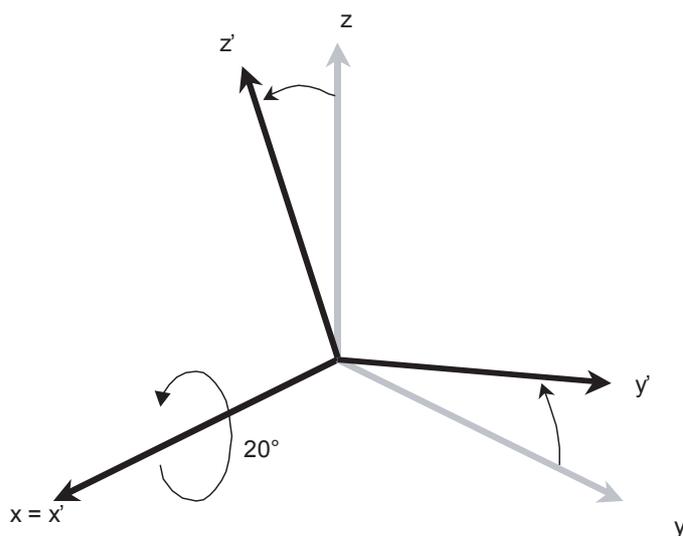


The position of frame **R2** (grey) relative to **R1** (black) is:
 $x = 250\text{mm}$, $y = 350\text{mm}$, $z = 450\text{mm}$, $r_x = 0^\circ$, $r_y = 0^\circ$, $r_z = 0^\circ$

Coordinates **rx**, **ry** and **rz** correspond to the angles of rotation that must be applied successively around the **x**, **y** and **z** axis to obtain the orientation of the frame.

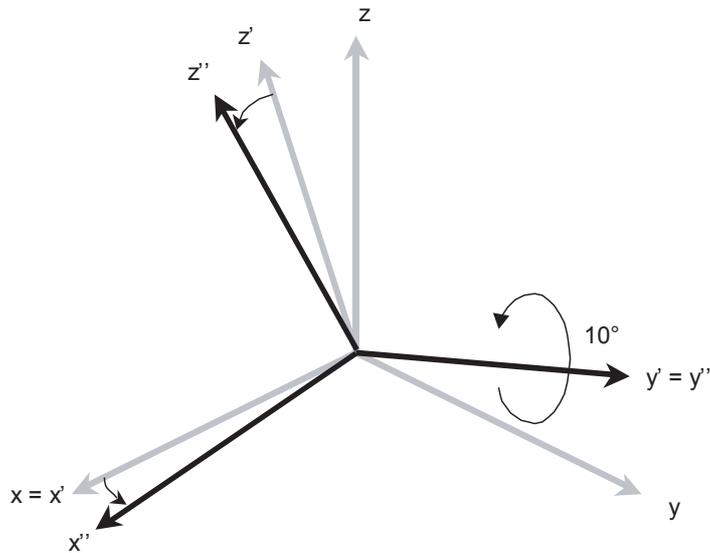
For example, orientation **rx = 20°**, **ry = 10°**, **rz = 30°** is obtained as follows. First, the frame (**x,y,z**) is rotated through **20°** around the **x** axis. This gives a new frame (**x',y',z'**). The **x** and **x'** axis coincide.

Frame rotation about the axis: X



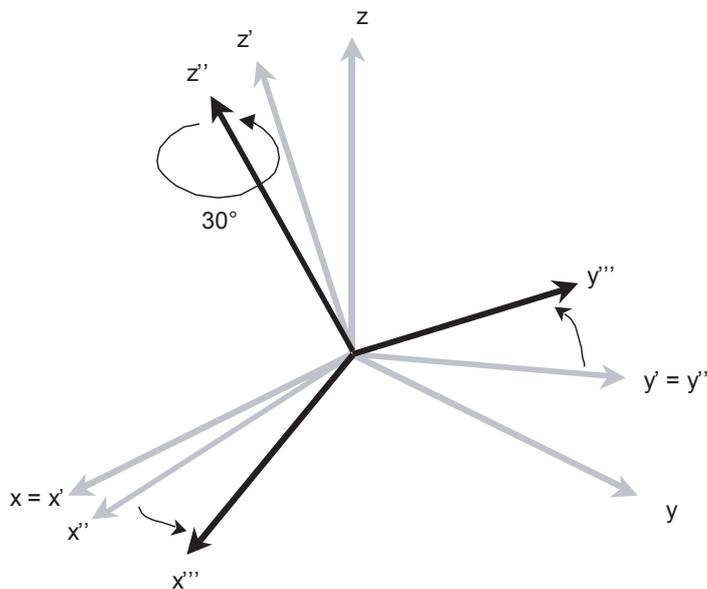
Then the frame is rotated through **20°** around the **y'** axis of the frame obtained at the previous step. This gives a new frame (**x'',y'',z''**). The **y'** and **y''** axis coincide.

Frame rotation about the axis: Y'



Lastly, the frame is rotated through **20°** about the **z''** axis of the frame obtained at the previous step. The orientation of the new frame obtained (**x''',y''',z'''**) is defined by **rx, ry, rz**. The **z''** and **z'''** axis coincide.

Frame rotation about the axis: Z''



The position of frame **R2** (grey) relative to **R1** (black) is:
 $x = 250\text{mm}, y = 350\text{ mm}, z = 450\text{mm}, rx = 20^\circ, ry = 10^\circ, rz = 30^\circ$

The values of **rx, ry** and **rz** are defined modulo **360** degrees. When the system calculates **rx, ry** and **rz**, their values are always between **-180** and **+180** degrees. Several possible values of **rx, ry**, and **rz** still remain: The system ensures that at least two coordinates are between **-90** and **90** degrees (unless **rx** is **+180** and **ry 0**). When **ry** is **90** degrees (**modulo 180**), the selected value of **rx** is zero.

9.3.3. OPERATORS

In ascending order of priority:

<code>trsf <trsf& trPosition1> = <trsf trPosition2></code>	Assigns trPosition2 to the trPosition1 variable field by field and returns trPosition2 .
<code>bool <trsf trPosition1> != <trsf trPosition2></code>	Returns true if a trPosition1 field is not equal to the corresponding trPosition2 field, otherwise it returns false .
<code>bool <trsf trPosition1> == <trsf trPosition2></code>	Returns true if each trPosition1 field is equal to the corresponding trPosition2 field, otherwise it returns false .
<code>trsf <trsf trPosition1> * <trsf trPosition2></code>	Returns the geometrical composition of the trPosition1 and trPosition2 transformations. Caution! Usually, trPosition1 * trPosition2 != trPosition2 * trPosition1!
<code>trsf ! <trsf trPosition></code>	Returns the inverse transformation of trPosition .

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter.

9.3.4. INSTRUCTIONS

num distance(trsf trPosition1, trsf trPosition2)

Function

Returns the distance between **trPosition1** and **trPosition2**.

CAUTION:

To ensure that the distance is valid, position 1 and position 2 must be defined relative to the same reference frame.

Example

This line computes the distance between two tools:

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

See also

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)

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

Function

This instruction returns an intermediate position aligned with a start position **trStart** and a target position **trEnd**. The **nPosition** parameter specifies the linear interpolation to apply according to the equation, for the x coordinate: $\text{trsf.x0} = \text{trStart.x} + (\text{trEnd.x} - \text{trStart.x}) * \text{nPosition}$. The same equation holds for Y and Z coordinates.

The orientation rx, ry, rz is computed with a similar, but more complex equation. The algorithm used by interpolateL is the same as the algorithm used by the motion generator to compute intermediate positions on a moveL instruction.

interpolateL(trStart, trEnd, 0) returns **trStart**; interpolateL(trStart, trEnd, 1) returns **trEnd**;
interpolateL(trStart, trEnd, 0.5) returns the middle position in between **trStart** and **trEnd**. A negative value of the **nPosition** parameter results in a position 'before' **trStart**. A value greater than 1 results in a position 'after' **trEnd**.

A runtime error is generated if the parameter nPosition is not in the range]-1, 2[.

See also

trsf position(point pPosition, frame fReference)

trsf position(frame fFrame, frame fReference)

trsf position(tool tTool, tool tReference)

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

trsf align(trsf trPosition, trsf Reference)

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

Function

This instruction returns an intermediate position on the arc of a circle defined by positions **trStart**, **trIntermediate** and **trEnd**. The **nPosition** parameter specifies the circular interpolation to apply. The algorithm used by **interpolateC** is the same as the algorithm used by the motion generator to compute intermediate positions on a **movec** instruction.

interpolateC(trStart, trIntermediate, trEnd, 0) returns **trStart**; **interpolateC**(trStart, trIntermediate, trEnd, 1) returns **trEnd**; **interpolateC**(trStart, trIntermediate, trEnd, 0.5) returns the middle position on the arc in between **trStart** and **trEnd**. A negative value of the **nPosition** parameter results in a position 'before' **trStart**. A value greater than 1 results in a position 'after' **trEnd**.

A runtime error is generated if the arc is not correctly defined (positions too close), or if the rotation interpolation remains undetermined (see the "Movement control - interpolation of orientation" chapter).

See also

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)

Function

This instruction returns the input **trPosition** with modified orientation so that the Z axis of the returned orientation is aligned with the nearest axis X, Y or Z of the reference orientation of **trReference**. The X, Y, Z coordinates of **trPosition** and **trReference** are not used: the x, y, z coordinates of the returned value are the same as the x, y, z coordinates of **trPosition**.

See also

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 TYPE

9.4.1. DEFINITION

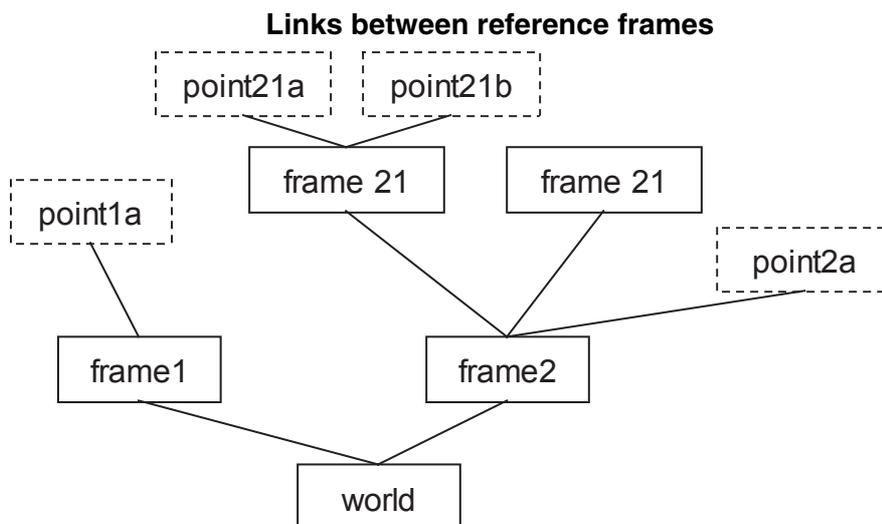
The frame type is used to define the position of reference frames in the cell.

The frame type is a structured type with only one accessible field:

trsf trsf position of the frame in its reference frame

The **reference frame** of a **frame** type variable is defined when it is initialized (via the user interface, via the = operator, or the **link()** instruction). The **world** reference frame is always defined in a **VAL 3** application: a reference frame is linked to the **world** frame, either directly or via other frames.

A runtime error is generated during a geometrical calculation if the **world** frame coordinates have been modified.



By default, local frame variables, and frames in user type variables, have no reference frame. Before they can be used, they must be initialized from another frame with the '=' operator, or via one of the **link()**, and **setFrame()** instructions.

9.4.2. USE

The use of reference frames in a robotic application is highly recommended for the following purposes:

- **To give a more intuitive view of the application points**
The display of the cell's points is structured according to the hierarchical structure of the frames.
- **To update the position of a set of points quickly**
When an application point is linked to an object, it is advisable to define a frame for that object and link the **VAL 3** points to the frame. If the object is moved, simply reattach the frame to allow all linked points to be corrected at the same time.
- **To reproduce a trajectory in several places in the cell**
Define the trajectory points relative to a working frame and teach a frame for each position in which the trajectory is to be reproduced. By assigning the value of a taught frame to the working frame, the entire trajectory "moves" to the taught frame.
- **To make it easier to calculate geometrical movements**
The **compose()** instruction allows geometrical movements expressed in any reference frame to be performed on any point. The **position()** instruction is used to calculate the position of a point in any reference frame.

9.4.3. OPERATORS

In ascending order of priority:

frame <frame& fReference1> = <frame fReference2>	Assigns the position and the reference frame of fReference2 to the fReference1 variable.
bool <frame fReference1> != <frame fReference2>	Returns true if fReference1 and fReference2 do not have the same reference frame or the same position in their reference frame.
bool <frame fReference1> == <frame fReference2>	Returns true if fReference1 and fReference2 have the same position in the same reference frame.

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter.

9.4.4. INSTRUCTIONS

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

Function

This instruction computes the coordinates of **fResult** from its origin point **pOrigin**, from a **pAxisOx** point on the axis (**Ox**), and a **pPlaneOxy** point on the plane (**Oxy**).

The **pAxisOx** point must be on the side of the positive **x** values. The **pPlaneOxy** point must be on the side of the positive **y** values.

The function returns:

- 0** No error.
- 1** The **pAxisOx** point is too close to the **pOrigin**.
- 2** The **pPlaneOxy** point is too close to the axis (**Ox**).

A runtime error is generated if one of the points has no reference frame.

trsf **position**(frame fFrame, frame fReference)

Function

This instruction returns the coordinates of the frame **fFrame** in the reference frame **fReference**.

A runtime error is generated if **fFrame** or **fReference** have no reference frame.

See also

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

void **link**(frame fFrame, frame fReference)

Function

This instruction changes the reference frame of **fFrame** and set it to **fReference**. The position of the frame in the reference frame remains unchanged.

See also

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

9.5. TOOL TYPE

9.5.1. DEFINITION

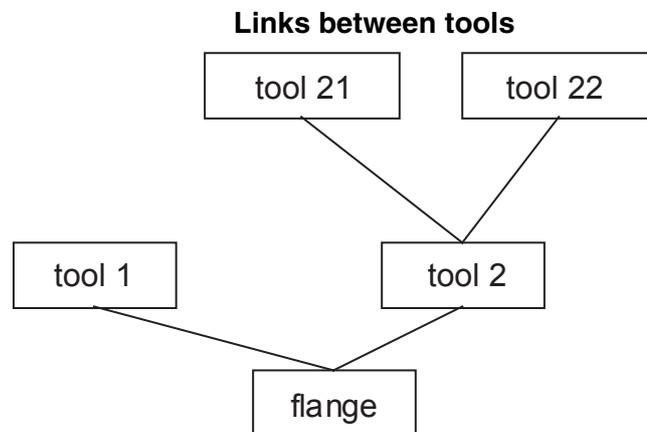
The **tool** type is used to define the geometry and action of a tool.

The **tool** type is a structured type with the following fields, in this order:

trsf trsf	position of the tool center point (TCP) in its basic tool
dio gripper	Output used to activate the tool
num otime	Time required to open the tool (seconds)
num ctime	Time required to close the tool (seconds)

The **reference tool** of a **tool** type variable is defined when it is initialized (via the user interface, the = operator or the [link\(\)](#) instruction). The **flange** tool is always defined in a **VAL 3** application: all tools are linked to the **flange** tool, either directly or via other tools.

A runtime error is generated during a geometrical computation if the **flange** tool coordinates have been modified.



By default, the output of a tool is the system **valve1** output, the opening and closing times are **0** and the basic tool is **flange**. Local tool variables, and tools in user type variables, have no reference tool. Before they can be used, they must be initialized from another tool with the '=' operator, or the [link\(\)](#) instruction.

9.5.2. USE

The use of tools in a robotic application is highly recommended for the following purposes:

- **To control the speed of movement**
During manual or programmed movements, the system controls the Cartesian speed at the end of the tool.
- **To reach the same points with different tools**
Simply select the **VAL 3** tool corresponding to the physical tool at the end of the arm.
- **To control the tool wear or a tool change**
The tool wear can simply be compensated by the update of the geometrical coordinates of the tool.

9.5.3. OPERATORS

In ascending order of priority:

tool <tool& tTool1> = <tool tTool2>	Assigns the position and the basic tool of tTool2 to the tTool1 variable.
bool <tool tTool1> != <tool tTool2>	Returns true if tTool1 and tTool2 do not have the same basic tool, the same position in their basic tool, the same digital output or the same opening and closing times.
bool <tool tTool1> == <tool tTool2>	Returns true if tTool1 and tTool2 have the same position in the same basic tool, and use the same digital output with the same opening and closing times.

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter.

9.5.4. INSTRUCTIONS

void open(tool tTool)

Function

This instruction activates the tool (opening) by setting its digital output to **true**.

Before activating the tool, **open()** waits for the robot to reach the required point by carrying out the equivalent of a **waitEndMove()**. After activation, the system waits for **otime** seconds before executing the next instruction.

This instruction does not make sure that the robot is stabilized in its final position before the tool is activated.

When it is necessary to wait for complete stabilization of the movement, the **isSettled()** instruction must be used.

A runtime error is generated if the **tTool dio** is not defined or is not an output, or if a previously recorded motion command cannot be run.

Example

```
// the open() instruction is equivalent to:
waitEndMove()
tTool.gripper=true
delay(tTool.otime)
```

See also

void close(tool tTool)

void waitEndMove()

void close(tool tTool)

Function

This instruction activates the tool (closing) by setting its digital output to **false**.

Before activating the tool, **close()** waits for the robot to stop at the point by carrying out the equivalent of a **waitEndMove()**. After activation, the system waits for `ctime` seconds before executing the next instruction.

This instruction does not make sure that the robot is stabilized in its final position before the tool is activated.

When it is necessary to wait for complete stabilization of the movement, the **isSettled()** instruction must be used.

A runtime error is generated if the **tTool dio** is not defined or is not an output, or if a previously recorded motion command cannot be run.

Example

// the close instruction is equivalent to:

```
waitEndMove()
tTool.gripper = false
delay(tTool.ctime)
```

See also

Type tool

void open(tool tTool)

void waitEndMove()

trsf position(tool tTool, tool tReference)

Function

This instruction returns the coordinates of the tool **tTool** in the **tReference** tool.

A runtime error is generated if **tTool** or **tReference** have no reference tool.

See also

trsf position(point pPosition, frame fReference)

trsf position(frame fFrame, frame fReference)

void link(tool tTool, tool tReference)

Function

This instruction changes the reference tool of **tTool** and set it to **tReference**. The position of the tool in the reference tool remains unchanged.

See also

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

9.6. POINT TYPE

9.6.1. DEFINITION

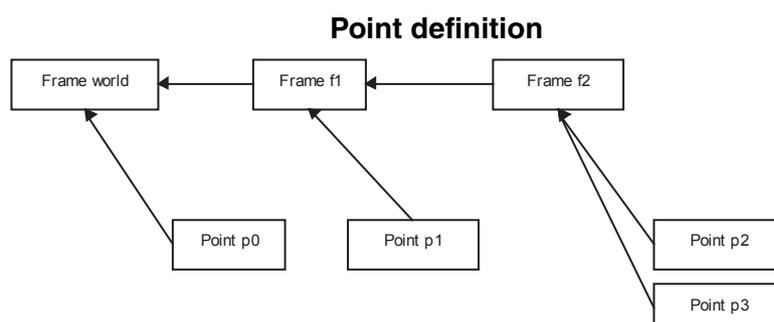
The **point** type is used to define the position and orientation of the robot tool in the cell.

The **point** type is a structured type with the following fields, in this order:

trsf trTrsf position of the point in its reference frame

config config arm configuration used to reach the position

The reference frame of a **point** is a **frame** type variable defined when it is initialized (via the user interface, using the = operator and the **link()**, **here()**, **apro()** and **compose()** instructions).



A runtime error is generated if a **point** type variable with no defined reference frame is used.

CAUTION:

By default, local point variables, and points in user type variables, have no reference frame. Before they can be used, they must be initialized from another point with the '=' operator, or via one of the **link(), **here()**, **apro()** and **compose()** instructions.**

9.6.2. OPERATORS

In ascending order of priority:

point <point& pPoint1> = <point pPoint2>	Assigns the position, the configuration and the reference frame of pPoint2 to the pPoint1 variable.
bool <point pPoint1> != <point pPoint2>	Returns true if pPoint1 and pPoint2 do not have the same reference frame or the same position in their reference frame.
bool <point pPoint1> == <point pPoint2>	Returns true if pPoint1 and pPoint2 have the same position in the same reference frame.

To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter.

9.6.3. INSTRUCTIONS

`num distance(point pPosition1, point pPosition2)`

Function

This instruction returns the distance between **pPosition1** and **pPosition2**.

A runtime error is generated if **pPosition1** or **pPosition2** does not have a defined reference frame.

Example

This program waits for the arm to be closer than 10 mm to the position **pTarget**

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

See also

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)`

Function

This instruction returns the **pPosition** to which the geometrical transformation **trTransformation** is applied relative to **fReference** frame.

CAUTION:

The rotation component of **tTransformation** usually modifies not only the orientation of **pPosition**, but also its Cartesian coordinates (unless **pPosition** is located at the origin of **fReference** frame).

If we only want **tTransformation** to modify the orientation of **pPosition**, it is necessary to update the result using the Cartesian coordinates of **pPosition** (see example).

The reference frame and the configuration of the point returned are those of **pPosition**.

A runtime error is generated if **pPosition** has no defined reference frame.

Example

```
// modification of the orientation without modification of Position
pResult = compose(pPosition, fReference, trTransformation)
pResult.trsf.x = pPosition.trsf.x
pResult.trsf.y = pPosition.trsf.y
pResult.trsf.z = pPosition.trsf.z
// modification of Position without modification of the orientation
trTransformation.rx = trTransformation.ry = trTransformation.rz = 0
pResult = compose(pResult, fReference, trTransformation)
```

See also

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

point appro(point pPosition, trsf trTransformation)

Function

This instruction returns a point modified by a geometric transformation. The transformation is defined relatively to the same reference frame as the input point.

The reference frame and the configuration of the returned point are those of the input point.

A runtime error is generated if **pPosition** has no defined reference frame.

Example

```
// Approach:move to 100 mm mm above the point (z axis)
movej(appro(pDestination,{0,0,-100,0,0,0}), flange, mNomDesc)
// Go to point
move1(pDestination, flange, mNomDesc)
```

See also

Operator **trsf** <trsf trPosition1> * <trsf trPosition2>

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

point here(tool tTool, frame fReference)

Function

This instruction returns the current position of the **tTool** tool in **fReference** frame (the position commanded and not the position measured). The reference frame of the point returned is **fReference**. The configuration of the point returned is the current configuration of the arm.

See also

joint **herej**()

config **config**(joint jPosition)

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

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

Function

This instruction returns the position of the **tTool** in the **fReference** frame when the arm is in the joint position **jPosition**.

The reference frame of the point returned is **fReference**. The configuration of the point returned is the configuration of the arm in the joint **jPosition** position.

See also

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)

Function

This instruction computes the joint position **jResult** corresponding to the specified point **pPosition**. It returns **true** if **jResult** is updated, **false** if no solution has been found.

The joint position to be located corresponds to the configuration of the **pPosition**. Fields with the value **free** do not determine the configuration. Fields with the value **same** specify the same configuration as **jInitial**.

For axis that can rotate through more than one full turn, there are several joint solutions with exactly the same configuration: the solution closest to **jInitial** is then taken.

No solution is possible if **pPosition** is out of reach (arm too short) or outside the software limits. If **pPosition** specifies a configuration, it may be outside the limits for that configuration, but within the limits for a different configuration.

A runtime error is generated if **pPosition** has no defined reference frame.

See also

joint herej()

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

trsf position(point pPosition, frame fReference)

Function

This instruction returns the coordinates of **pPosition** in **fReference** frame.

A runtime error is generated if **pPosition** has no reference frame.

Example

The distance between 2 points is the distance between their positions in world:

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

See also

num distance(point pPosition1, point pPosition2)

trsf position(tool tTool, tool tReference)

trsf position(frame fFrame, frame fReference)

void link(point pPoint, frame fReference)

Function

This instruction changes the reference frame of **pPoint** and set it to **fReference**. The position of the point in the reference frame remains unchanged.

See also

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

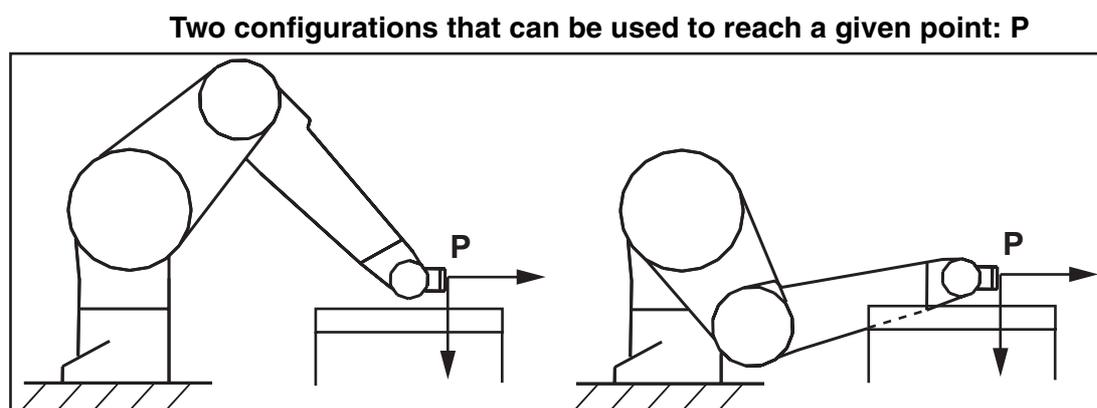
9.7. CONFIG TYPE

The configuration concept of a Cartesian point is an "advanced" concept; this chapter can be skipped the first time you read this document.

9.7.1. INTRODUCTION

There are generally several ways in which a robot can reach a given Cartesian point.

These possibilities are known as "configurations". The figure below illustrates two different configurations:



In some cases, among all the possible configurations, it is important to specify the ones that are valid and the ones that are to be prohibited. To deal with this problem, the **point** type is used to specify the configurations allowed for the robot, via its **config** type field as defined below.

9.7.2. DEFINITION

The **config** type is used to define the configurations authorized for a given Cartesian position.

It depends on the type of arm used.

For a **Stäubli RX/TX** arm, the **config** type is a structured type whose fields are, in that order:

shoulder	shoulder configuration
elbow	elbow configuration
wrist	wrist configuration

For a **Stäubli RS/TS** arm, the **config** type is limited to the **Shoulder** field:

shoulder	shoulder configuration
-----------------	------------------------

The **shoulder**, **elbow** and **wrist** fields can have the following values:

shoulder	righty	righty shoulder configuration imposed
	lefty	lefty shoulder configuration imposed
	ssame	Shoulder configuration change not allowed
	sfree	Free shoulder configuration

elbow	epositive	epositive elbow configuration imposed
	enegative	enegative elbow configuration imposed
	esame	Elbow configuration change not allowed
	efree	Free elbow configuration

wrist	wpositive	wpositive wrist configuration imposed
	wnegative	wnegative wrist configuration imposed
	wsame	Wrist configuration change not allowed
	wfree	Free wrist configuration

9.7.3. OPERATORS

In ascending order of priority:

config <config& configuration1> = <config configuration2>	Assigns the shoulder , elbow and wrist fields for configuration2 to the configuration1 variable.
bool <config configuration1> != <config configuration2>	Returns true if configuration1 and configuration2 do not have the same shoulder , elbow or wrist field values.
bool <config configuration1> == <config configuration2>	Returns true if configuration1 and configuration2 have the same shoulder , elbow or wrist field values.

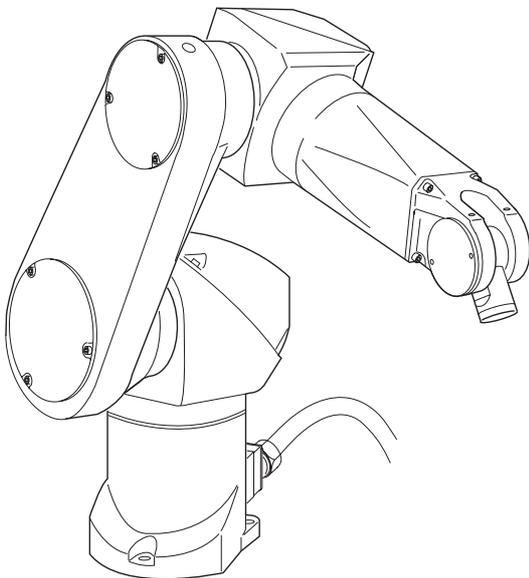
To avoid confusions between = and == operators, the = operator is not allowed within **VAL 3** expressions used as instruction parameter.

9.7.4. CONFIGURATION (RX/TX ARM)

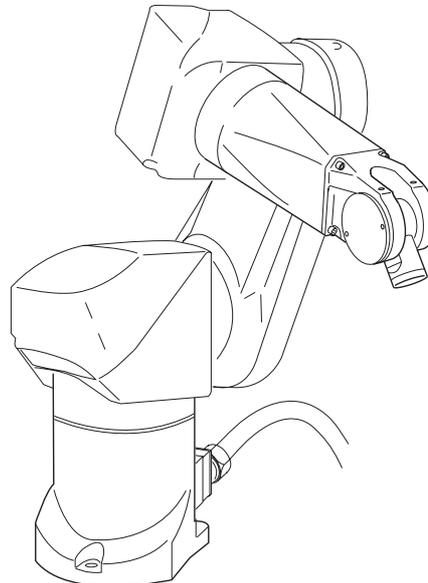
9.7.4.1. SHOULDER CONFIGURATION

To reach a given Cartesian point, the arm of the robot may be to the right or the left of the point: these two configurations are called **righty** and **lefty**.

Configuration: righty



Configuration: lefty

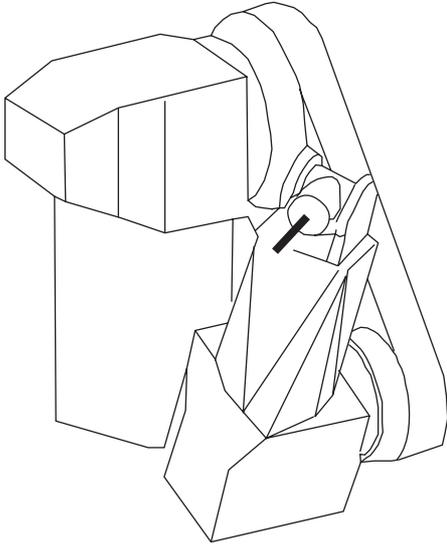


The righty configuration is defined by $(d1 * \sin(j2) + d2 * \sin(j2+j3) + \delta) < 0$, and the lefty configuration is defined by $(d1 * \sin(j2) + d2 * \sin(j2+j3) + \delta) \geq 0$, where $d1$ is the length of the robot arm, $d2$ the length of the forearm, and δ the distance between axis 1 and axis 2, in the x direction.

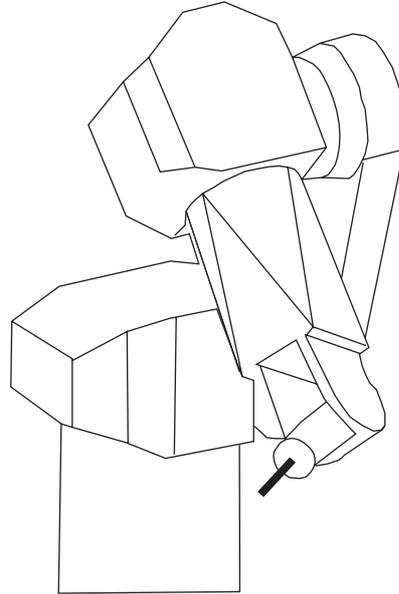
9.7.4.2. ELBOW CONFIGURATION

In addition to the shoulder configuration, there are two robot elbow configurations: the elbow configurations are called **epositive** and **enegative**.

Configuration: enegative



Configuration: epositive



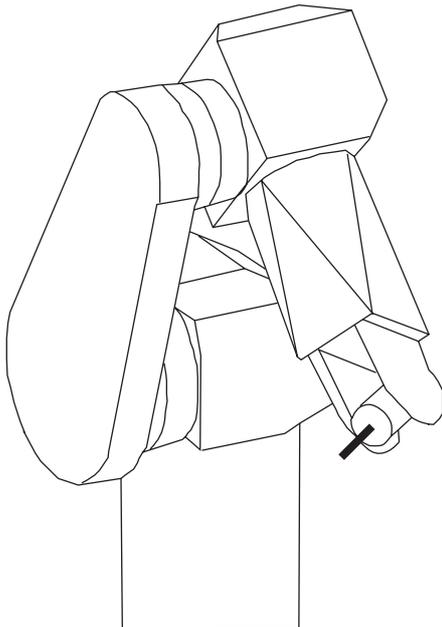
The **epositive** configuration is defined by $j3 \geq 0$.

The **enegative** configuration is defined by $j3 < 0$.

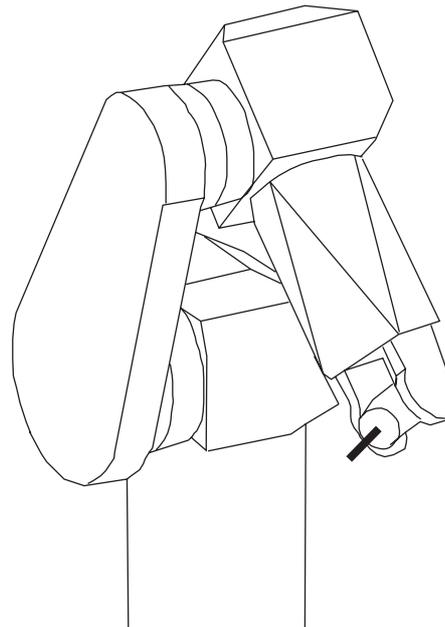
9.7.4.3. WRIST CONFIGURATION

In addition to the shoulder and elbow configurations, there are two robot wrist configurations. The two wrist configurations are called **wpositive** and **wnegative**.

Configuration: wnegative



Configuration: wpositive



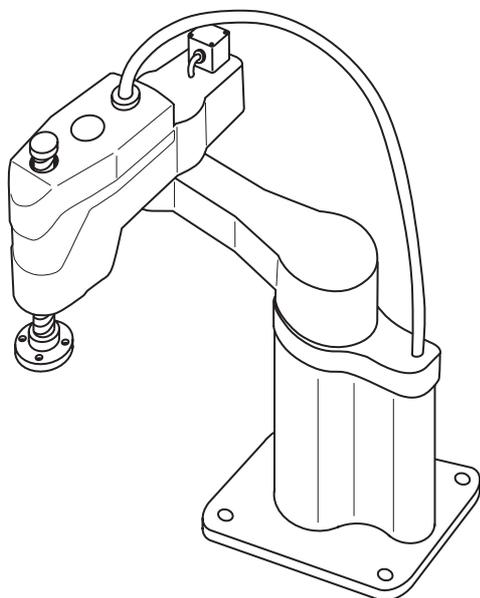
The **wpositive** configuration is defined by $j5 \geq 0$.

The **wnegative** configuration is defined by $j5 < 0$.

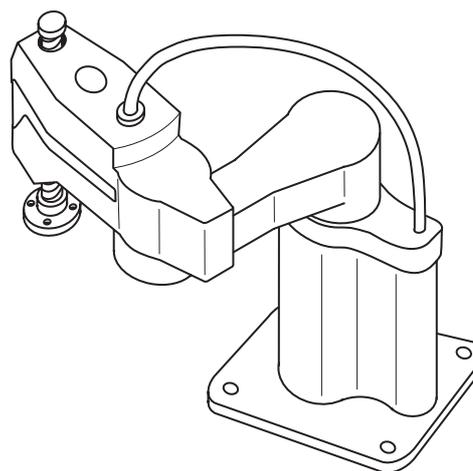
9.7.5. CONFIGURATION (RS/TS ARM)

To reach a given Cartesian point, the arm of the robot may be to the right or the left of the point: these two configurations are called **righty** and **lefty**.

Configuration: righty



Configuration: lefty



The **righty** configuration is defined by $\sin(j2) > 0$, and the **lefty** configuration is defined by $\sin(j2) \leq 0$.

9.7.6. INSTRUCTIONS

config config(joint jPosition)

Function

This instruction returns the configuration of the robot for the joint **jPosition** position.

See also

point here(tool tTool, frame fReference)

joint herej()

CHAPTER 10

MOVEMENT CONTROL

10.1. TRAJECTORY CONTROL

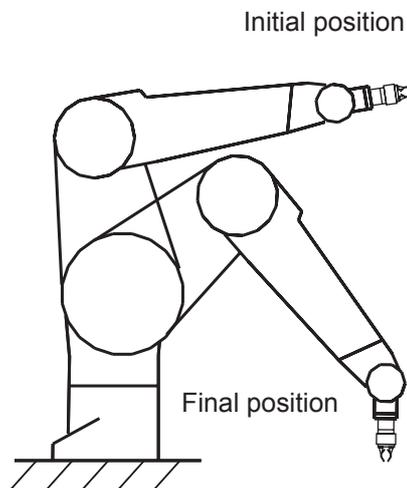
A succession of points is not sufficient to define the trajectory of a robot. It is also necessary to indicate the type of trajectory used between the points (curve or straight line), specify how the trajectories are linked together and define the movement speed parameters. This section therefore presents the different types of movements (**movej**, **movel** and **movec** instructions) and describes how to use the movement descriptor parameters (**mdesc** type).

10.1.1. TYPES OF MOVEMENT: POINT-TO-POINT, STRAIGHT LINE, CIRCLE

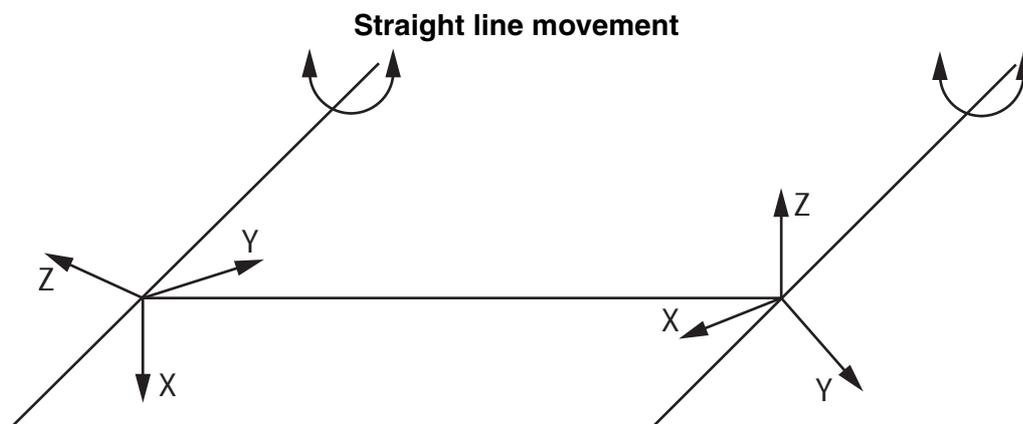
The robot's movements are mainly programmed using the **movej**, **movel** and **movec** instructions. The **movej** instruction can be used to make point-to-point movements, **movel** is used for straight line movements, and **movec** for circular movements.

A point-to-point movement is a movement in which only the final destination (Cartesian or joint position) is important. Between the start point and the end point, the tool center point follows a curve defined by the system to optimize the speed of the movement.

Initial and final positions

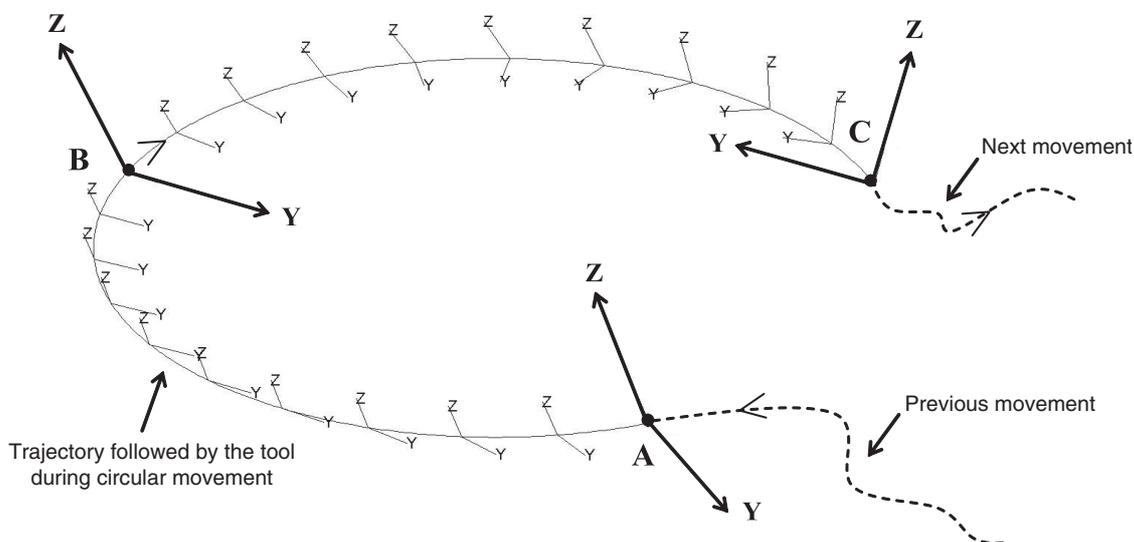


Conversely, in the case of a straight line movement, the tool center point moves along a straight line. The orientation is interpolated in a linear way between the initial and final orientation of the tool.



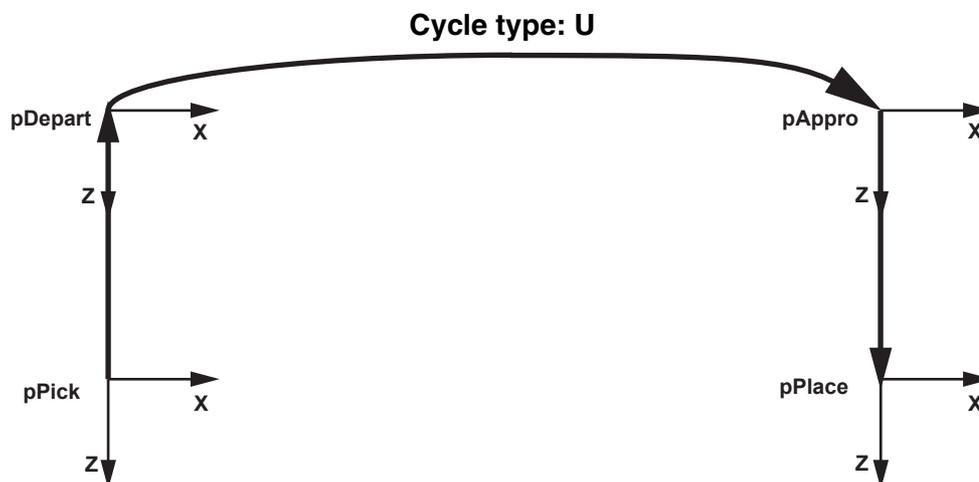
In a circular movement, the tool center point moves through an arc defined by **3** points, and the tool orientation is interpolated between the initial orientation, the intermediate orientation, and the final orientation.

Circular movement



Example:

A typical handling task involves picking up parts at one location and putting them down at another. Let us assume that the parts are to be picked up at the **pPick** point and put down at the **pPlace** point. To go from the **pPick** point to the **pPlace** point, the robot must pass through a disengagement point **pDepart** and an approach point **pAppro**.



Let us assume that the robot is initially at the **pPick** point. The program required to execute the movement can be written as follows:

```

move1(pDepart, tTool, mDesc)
movej(pAppro, tTool, mDesc)
move1(pPlace, tTool, mDesc)
  
```

Straight line movements are used for disengagement and approach. However, the main movement is a point-to-point movement, as the geometry of this part of the trajectory does not need to be accurately controlled, because the aim is to move as quickly as possible.

Note:

The geometry of the trajectory does not depend on the speed at which both these types of movement are executed. The robot always passes through the same position. This is particularly important when developing applications. It is possible to start with slow movements and then progressively increase the speed without distorting the trajectory of the robot.

10.1.2. MOVEMENT SEQUENCING: BLENDING

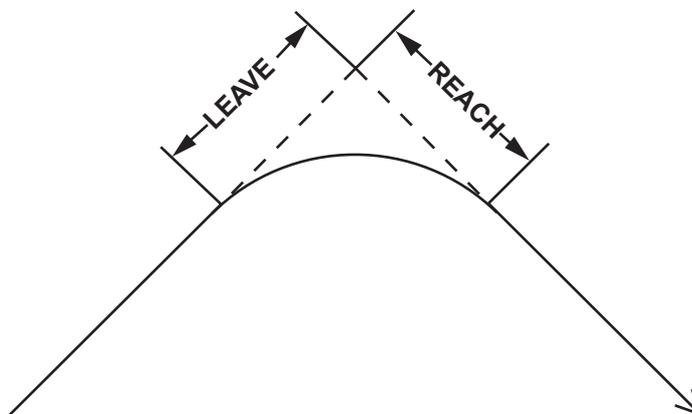
10.1.2.1. BLENDING

Let us now return to the example of the **U** cycle described in the previous chapter. In the absence of any specific movement sequencing control, the robot stops at the **pDepart** and **pAppro** points, as the trajectory is angled at these points. This unnecessarily increases the duration of the operation and there is no need to pass through these precise points.

The duration of the movement can be significantly reduced by "blending" the trajectory in the vicinity of the **pDepart** and **pAppro** points. To do so, we use the **blend** field of the movement descriptor. When this field is set to **off**, the robot stops at each point along the trajectory. However, when the parameter is set to **joint** or **Cartesian**, the trajectory is blended in the vicinity of each point and the robot no longer stops at the fly-by points.

When the **blend** field has the value **joint** or **Cartesian**, two other parameters must be specified: **leave** and **reach**. These parameters determine the distance from the arrival point at which the nominal trajectory is left (start of blending) and the distance from the arrival point at which it is rejoined (end of blending).

Definition of the distances: 'leave' / 'reach'

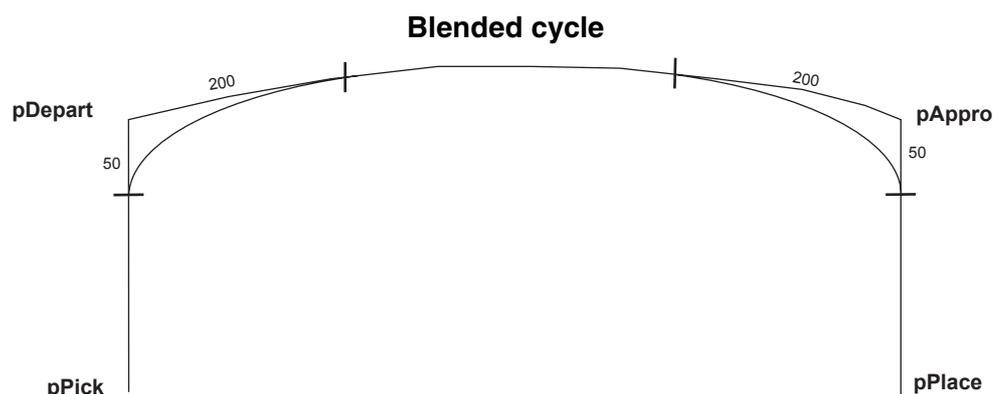


Example:

Let us return to the program described in the section entitled "Types of movement: point-to-point or straight line". The previous movement program can be modified as follows:

```
mDesc.blend = joint
mDesc.leave = 50
mDesc.reach = 200
move1(pDepart, tTool, mDesc)
mDesc.leave = 200
mDesc.reach = 50
movej(pAppro, tTool, mDesc)
mDesc.blend = off
move1(pPlace, tTool, mDesc)
```

The following trajectory is obtained:



The robot no longer stops at the **pDepart** and **pAppro** points. The movement is therefore faster. In fact, the larger the **leave** and **reach** distances, the faster the movement.

10.1.2.2. CANCEL BLENDING

The `waitEndMove()` instruction is used to cancel the effect of blending. The robot then completes the last programmed movement as far as its arrival point, as if the movement descriptor **blend** field were set to **off**.

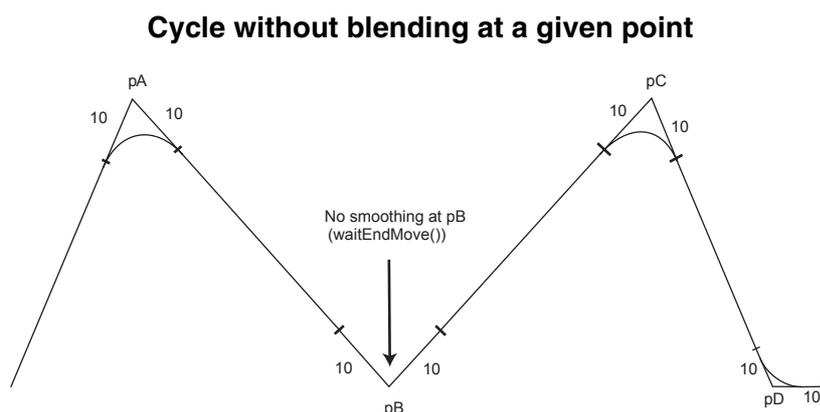
For example, let us examine the following program:

```

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)
etc.

```

The trajectory followed by the robot is then as follows:



10.1.2.3. JOINT BLENDING, CARTESIAN BLENDING

To make it very simple, a joint blending is like a point to point move between the leave and reach points, the Cartesian blending is like a circular move between these points.

- Joint blending is usually faster than Cartesian blending. But joint blending may lead to strange path when there is complex orientation change (typically a circle in a plane followed with a circle in an orthogonal plane), or for pure rotation moves.

- The speed and acceleration control is more accurate with the Cartesian blending. A Cartesian blending between two moves that are on the same plane is also guaranteed to be in this plane.

The most optimized shape for a blending depends on the application, but the VAL 3 interpreter has to choose the shape automatically. The result is usually not surprising, but sometimes it can be...

The choice may lead to an unexpected complex shape when the effective leave and reach distances are quite different. The computed shape reduces the curvature of the path for speed optimization, but the result may not be desirable for some process applications. When the leave and reach distances are equal, the Cartesian blending always result in a simple shape.

The Cartesian blending applies to both position and orientation. A complex orientation change may impact the shape of the blending, leading to unexpected results. There are also some restrictions with orientation change: like on a circle, big changes in orientation result in a motion error when several solutions are possible but the system has no criteria to choose one. When this happens, additional intermediate point(s) are needed to help the system find the correct orientation interpolation.

10.1.3. MOVEMENT RESUMPTION

When the arm power is cut off before the robot has finished its movement, following an emergency stop for example, movement resumption is required when power is restored to the system. If the arm has been moved manually during the stoppage, it may be in a position far from its normal trajectory. It is then necessary for movement resumption to take place without a collision occurring. The **VAL 3**'s trajectory control function provides the possibility of managing movement resumption using a "connection movement".

When movement resumes, the system ensures that the robot is indeed on its programmed trajectory: if there is any deviation, however slight, it automatically stores a point-to-point command to reach the exact position at which the robot left its trajectory: it is a "connection movement". This movement is made at low speed. It must be validated by the operator, except in automatic mode, in which it can be carried out without human intervention. The **autoConnectMove()** instruction is used to detail behaviour in automatic mode.

The **resetMotion()** instruction is used to cancel the current movement, and possibly to program a connection movement in order to resume a position at low speed and under the operator's control.

10.1.4. PARTICULARITIES OF CARTESIAN MOVEMENTS (STRAIGHT LINE, CIRCLE)

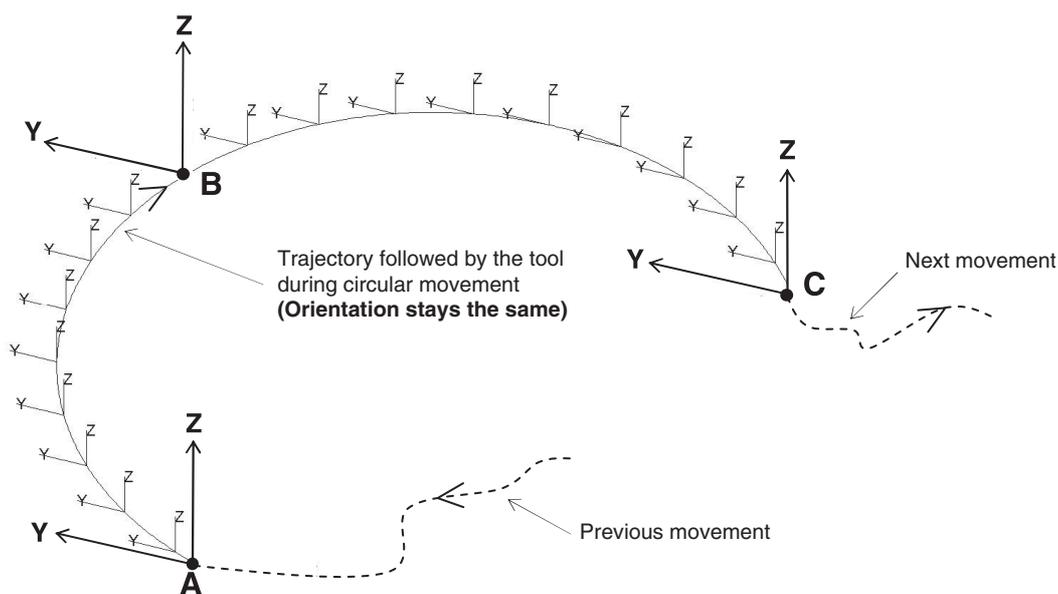
10.1.4.1. INTERPOLATION OF THE ORIENTATION

The **VAL 3** trajectory generator always minimizes the amplitude of tool rotations when moving from one orientation to another.

This makes it possible, as a particular case, to program a constant orientation, in absolute terms, or as compared with the trajectory, on all straight-line or circular movements.

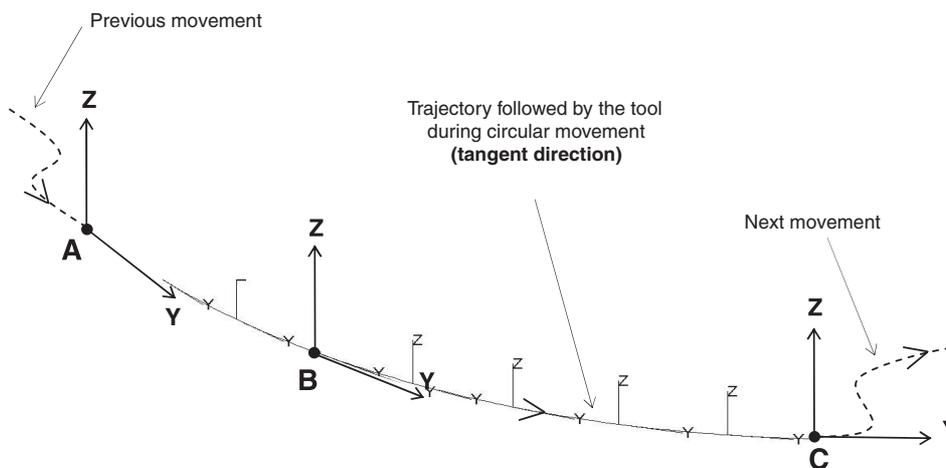
- For a constant orientation, the initial and final positions, and the intermediate position for a circle, must have the same orientation.

Constant orientation in absolute terms



- For a constant orientation as compared with the trajectory (e.g. direction Y for the tool marker tangent to the trajectory), the initial and final positions, and the intermediate position for a circle, must have the same orientation as compared with the trajectory.

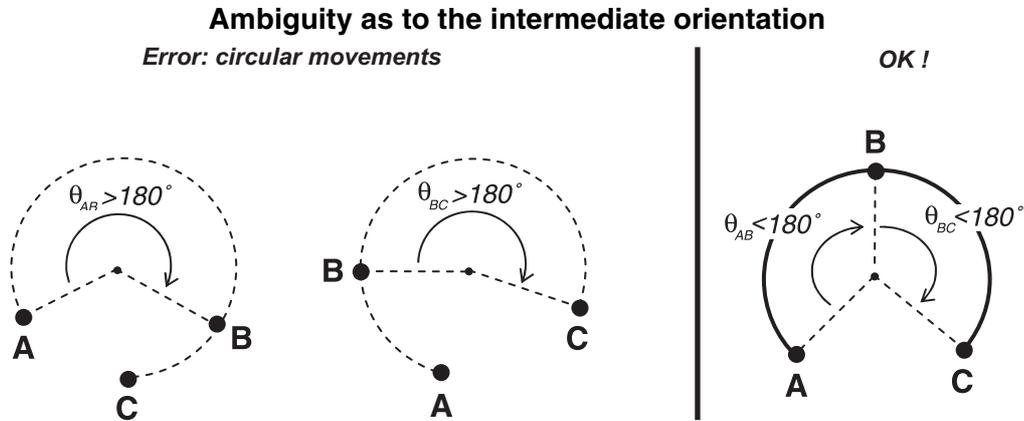
Constant orientation as compared with the trajectory



This results in a limitation for circular movements:

If the intermediate point forms an angle of **180°** or more with the initial point or the final point, there are several interpolation solutions for the orientation, and an error is generated.

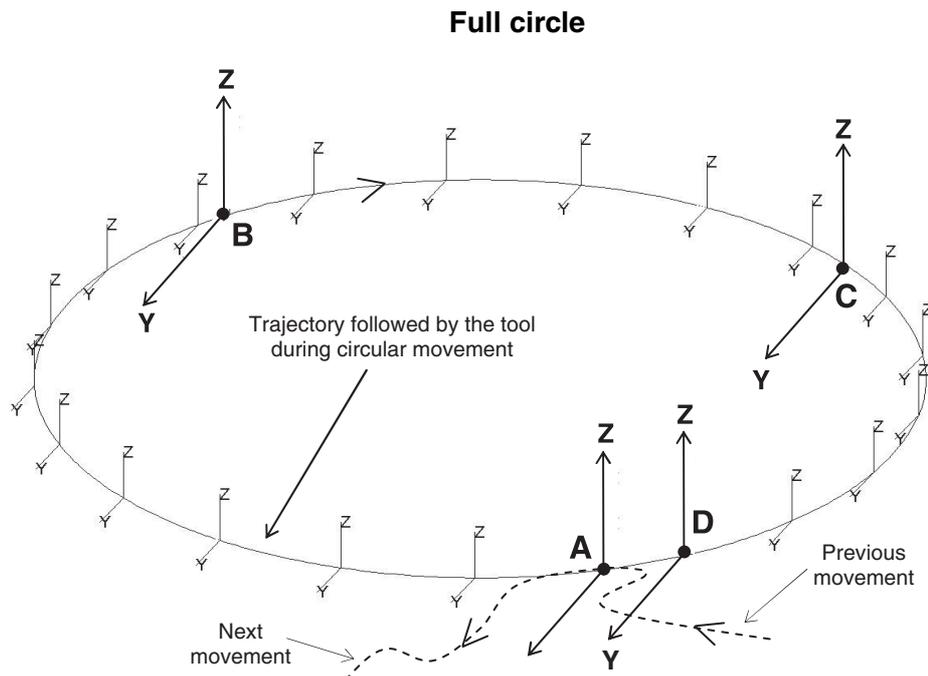
It is then necessary to modify the position of the intermediate point to remove the ambiguity from the intermediate orientations.



In particular, programming a full circle involves **2 movec** instructions:

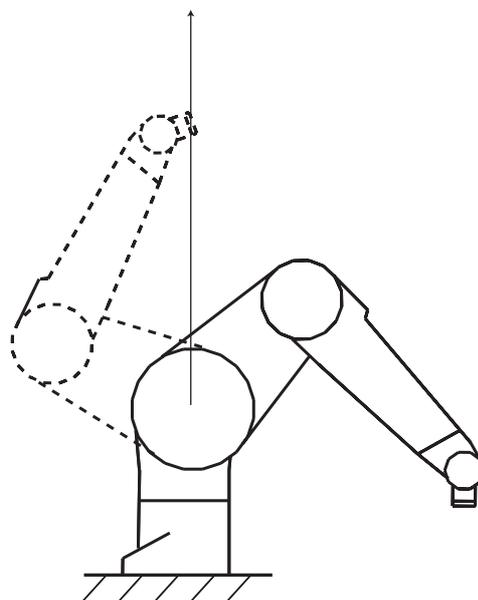
```

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



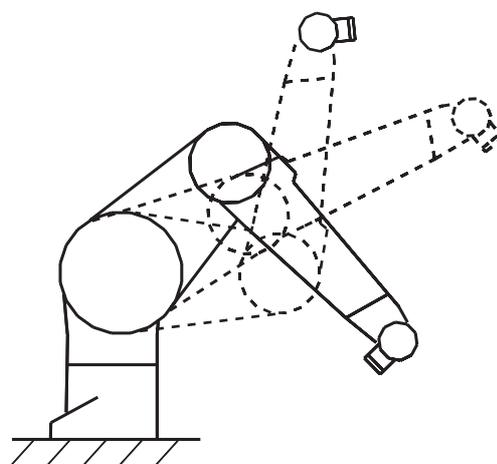
10.1.4.2. CONFIGURATION CHANGE (ARM RX/TX)

Configuration change: righty / lefty



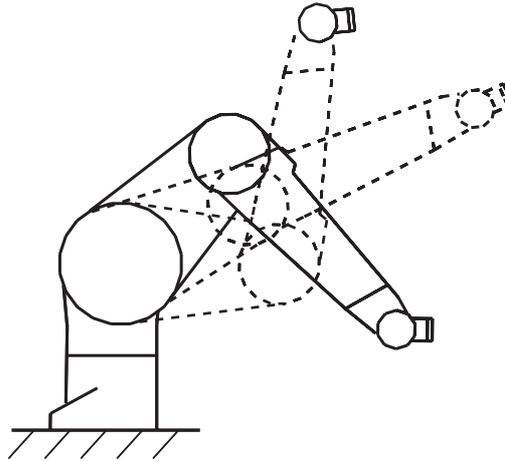
During a change of shoulder configuration, the centre of the robot's wrist has to pass vertically through axis **1** (but not exactly in the case of offset robots).

Positive/negative elbow configuration change



During a change of elbow configuration, the arm has to go through the straight arm position (**$j_3 = 0^\circ$**).

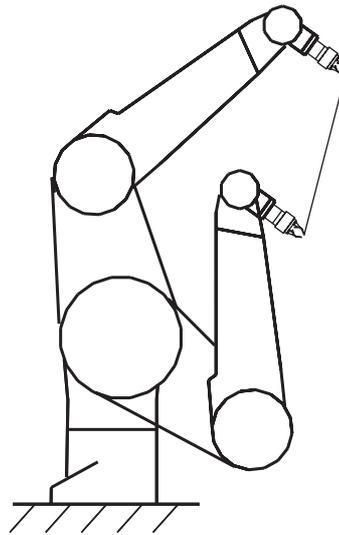
Positive/negative wrist configuration change



During a change of wrist configuration, the arm has to go through the straight wrist position ($j_5 = 0^\circ$).

The robot must therefore pass through specific positions during a configuration change. But we cannot require a straight-line or circular movement to pass through these positions if they are not on the desired trajectory! This means that **we cannot impose a change of configuration during a straight-line or circular movement**.

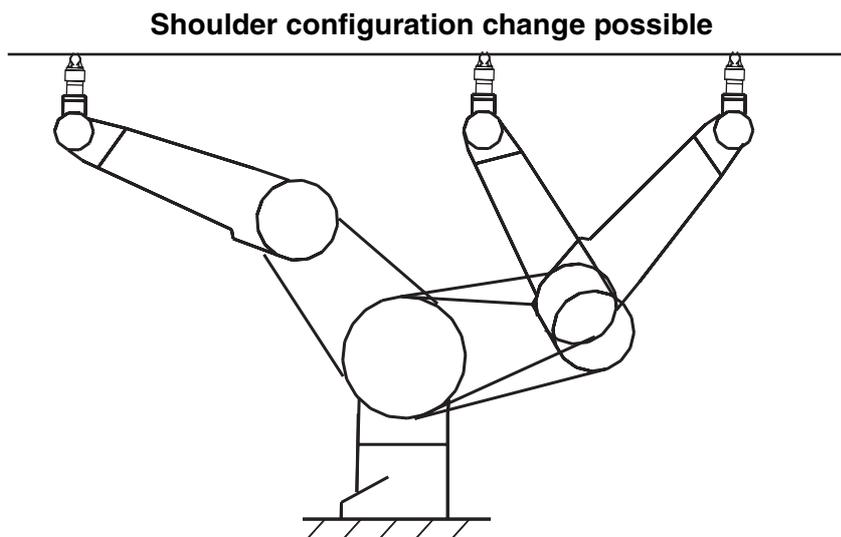
Elbow configuration change impossible



In other words, during a straight-line or circular movement, we can only impose a configuration if it is compatible with the initial position: it is therefore always possible to specify a free configuration, or one that is identical to the initial configuration.

In certain exceptional cases, the straight line or arc does indeed pass through a position in which a change of configuration is possible. In this case, if the configuration has been left free, the system can decide to change the configuration during a straight-line or circular movement.

For a circular movement, the configuration of the intermediate point is not taken into account. The only configurations that count are those of the initial and final positions.



10.1.4.3. SINGULARITIES (ARM RX/TX)

Singularities are an inherent characteristic of all **6**-axis robots. Singularities can be defined as the points at which the robot changes configuration. Certain axes are then aligned: two aligned axes behave as a single axis and the **6**-axis therefore behaves locally as a **5**-axis robot. The end effector is then unable to carry out certain movements. This is not a problem in the case of a point-to-point movement: system-generated movements are still possible. On the other hand, during a straight-line or circular movement, we impose a movement geometry. If the movement is impossible, an error is generated when the robot attempts to move.

10.2. MOVEMENT ANTICIPATION

10.2.1. PRINCIPLE

The system controls the movements of the robot in more or less the same way as a driver drives a car. It adapts the speed of the robot to the geometry of the trajectory. Thus the better the trajectory is known in advance, the better the system can optimize the speed of movement. This explains why the system does not wait for the current robot movement to be completed before taking the instructions for the next movement into account.

Let us consider the following program lines:

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

Let us suppose that the robot is stationary when the program reaches these lines. When the first instruction is executed, the robot starts to move towards point **pA**. The program then immediately proceeds to the second line, well before the robot reaches point **pA**.

When the system executes the second line, the robot starts to move towards **pA** and the system records the fact that after point **pA**, the robot must go to point **pB**. The program then continues with the next line: while the robot continues its movement towards **pA**, the system records the instruction that after **pB**, the robot must proceed to **pC**. As the program is executed much faster than the robot actually moves, the robot is probably still moving towards **pA** when the next line is executed. The system thus records the next successive points.

When the robot starts to move towards **pA**, it already "knows" that after **pA**, it must go successively to **pB**, **pC** and **pD**. If blending has been activated, the system knows that the robot will not stop before point **pD**. It can then accelerate faster than if it had to prepare to stop at **pB** or **pC**.

The fact of executing the instruction lines only records the successive movement commands. The robot then performs them according to its capabilities. The memory in which the movements are stored is large, to allow the system to optimize the trajectory. Nevertheless, it is limited. When it is full, the program stops at the next movement instruction. It resumes when the robot has completed the current movement, thus creating space in the system memory.

10.2.2. ANTICIPATION AND BLENDING

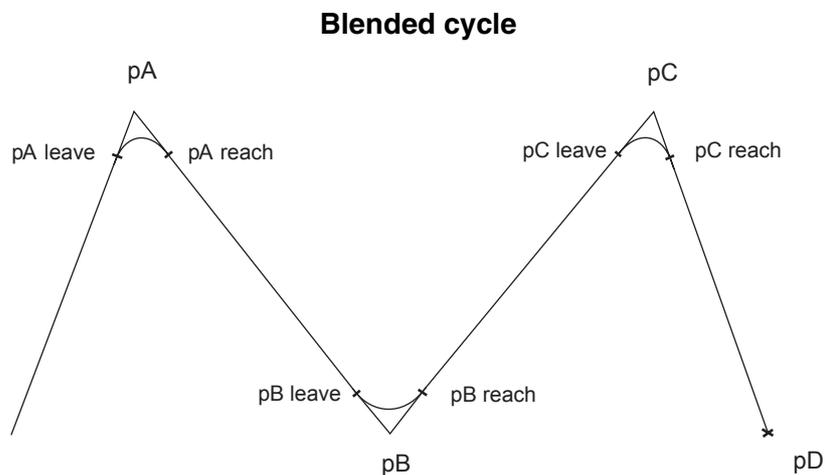
This section examines in detail what happens when the movements are sequenced. Let us look again at the previous example:

```

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

```

Let us assume that blending is activated in the movement descriptor, **mDesc**. When the first line is executed, the system does not yet know what the next movement will be. Only the movement between the start point and the **pA** leave point is fully determined, as the **pA** leave point is defined by the system from the movement descriptor **leave** data (see the figure below).



Until the second line is executed, the part of the blending trajectory in the vicinity of point **pA** has not been fully determined, as the system has not yet taken the next movement into account. In single-step mode, the robot does not go further than the **pA** leave point. When the next instruction is executed, the blending trajectory in the vicinity of point **pA** (between **pA** leave and **pA** reach) can be defined, together with the movement as far as point **pB** leave. The robot can then proceed to **pB** leave. In single-step mode, it will not go beyond this point until the user executes the third instruction, and so on.

The advantage of this operating mode is that the robot passes through exactly the same position in single-step mode as in normal program execution mode.

10.2.3. SYNCHRONIZATION

The anticipation mechanism causes desynchronization between the **VAL 3** instruction lines and the corresponding robot movements: the **VAL 3** program is ahead of the robot.

When it is necessary to carry out an action at a given robot position, the program has to wait for the robot to complete its movements: the `waitEndMove()` instruction is used for synchronization purposes. An alternative is to use the instruction `getMoveId()` to detect the arm's progress on the trajectory (see 10.4, real time movement control).

Thus in the following program:

```

movej (A, tTool, mDesc)
movej (B, tTool, mDesc)
waitEndMove ()
movej (C, tTool, mDesc)
movej (D, tTool, mDesc)
etc.

```

The first two lines are executed when the robot starts to move towards **A**. The program is then blocked at the third line until the robot is stabilized at point **B**. When the robot movement is stabilized at **B**, the program resumes.

The **open()** and **close()** instructions also wait for the robot to complete its movements before activating the tool.

10.3. SPEED MONITORING

10.3.1. PRINCIPLE

The principle of monitoring the speed along a trajectory is as follows:

The robot moves and accelerates at all times to its maximum capacity, in accordance with the speed and acceleration constraints imposed by the movement command.

The movement commands contain two types of speed constraints defined in a **mdesc** type variable:

1. The velocity (joint speeds), acceleration and deceleration constraints
2. The Cartesian speed constraints for the tool center point

Acceleration determines the rate at which the speed increases at the beginning of a trajectory. Conversely, deceleration determines the rate at which the speed decreases at the end of the trajectory. When high acceleration and deceleration values are used, the movements are faster, but jerkier. With low values, the movements take a little longer, but they are smoother.

10.3.2. SIMPLE SETTINGS

When the tool and the object carried by the robot do not need to be handled with special care, Cartesian speed constraints are not necessary. The speed along the trajectory is normally adjusted as follows:

1. Set the Cartesian speed constraints very high, for example to the default values, to ensure that they do not affect the rest of the setting procedure.
2. Initialize the velocity, acceleration and deceleration using the nominal values (**100%**).
3. Then adjust the speed along the trajectory using the velocity parameter.

To keep a harmonious arm behaviour, the acceleration and deceleration should be modified with the velocity: the acceleration and deceleration parameters should be roughly the square of the velocity parameter. For instance, a velocity of 120 % = 1.2 is best adapted with acceleration and deceleration of $1.2 \times 1.2 = 1.44 = 144$ %. Higher values for acceleration and deceleration give a more aggressive, but shakier arm behaviour.

10.3.3. ADVANCED SETTINGS

To control the Cartesian speed of the tool, for example to execute a trajectory at a constant speed, proceed as follows:

1. Set the Cartesian speed constraints to the values required.
2. Initialize the velocity, acceleration and deceleration using the nominal values (**100%**).
3. Then adjust the speed along the trajectory using the Cartesian speed parameters only.
4. If the speed is not sufficient, increase the acceleration and deceleration parameters.
If you want to brake automatically in sections with pronounced curves, reduce the acceleration and deceleration parameters.

10.3.4. ENVELOPPE ERROR

The nominal values for joint speed and acceleration are the nominal load values supported by the robot, irrespective of trajectory.

However, the robot can often operate faster: the maximum speeds that can be reached by the robot depend on its load and trajectory. In suitable cases (light load, positive gravitational effect) the robot can exceed its nominal values without any damage being caused.

If the robot is carrying a load that is heavier than its nominal load, or if the joint speed and acceleration values are too high, the robot cannot always obey its movement command and stops when an envelope error occurs. Such errors can be avoided by specifying lower velocities and acceleration parameters.

CAUTION:

In the case of straight line movements near a singularity, a small tool movement requires large joint movements. If the velocity is set too high, the robot cannot obey the command and stops when an envelope error occurs.

10.4. REAL-TIME MOVEMENT CONTROL

The movement commands previously described in this manual have no immediate effect: when each command is executed, a movement order is stored in the system. The robot then executes the stored movements.

The robot's movements can be controlled instantly, as follows:

- The monitor speed modifies the speed of all the movements. It can be adjusted with immediate effect with the `setMonitorSpeed()` instruction. However this instruction cannot increase the speed when the operator can also adjust it from the MCP.
- The `stopMove()` and `restartMove()` instructions are used to stop and restart movement along the trajectory.
- The `resetMotion()` instruction is used to stop the movement in progress and cancel the stored movement commands.
- The Alter instruction (option) applies to the path a geometrical transformation (translation, rotation, rotation at the tool center point) that is immediately effective.
- It is possible to track accurately, and in real time, the position of the robot on its path with the `getMoveId()` instruction. Each move instruction is identified with a numeric value returned by the instruction. The `getMoveId()` instruction returns a numeric value that identifies the current move (integer part), and the progress on this move (decimal part). For instance, a move id of 17.572 means that the current move is the move instruction that returned 17, and the robot position has performed 57.2 % of this move.

10.5. MDESC TYPE

10.5.1. DEFINITION

The **mdesc** type is used to define the movement parameters (speed, acceleration, blending).

The **mdesc** type is a structured type, with the following fields, in this order:

num accel	Maximum permitted joint acceleration as a % of the nominal acceleration of the robot.
num vel	Maximum permitted joint speed as a % of the nominal speed of the robot.
num decel	Maximum permitted joint deceleration as a % of the nominal deceleration of the robot.
num tvel	Maximum permitted translational speed of the tool center point, in mm/s or inches/s depending on the unit of length of the application.
num rvel	Maximum permitted rotational speed of the tool center point, in degrees per second.
blend blend	Blend mode: off (no blending), joint or Cartesian (blending).
num leave	In joint and Cartesian blend mode, distance between the target point at which blending starts and the next point, in mm or inches, depending on the unit of length of the application.
num reach	In joint and Cartesian blend mode, distance between the target point at which blending stops and the next point, in mm or inches, depending on the unit of length of the application.

A detailed explanation of these parameters is given at the beginning of the chapter entitled "Movement control".

By default, an **mdesc** type variable is initialized with **{100,100,100,9999,9999,joint,50,50}**.

10.5.2. OPERATORS

In ascending order of priority:

mdesc < mdesc & desc1 > = < mdesc desc2 >	Assigns each desc2 field to the field corresponding to the desc1 variable.
bool < mdesc desc1 > != < mdesc desc2 >	Returns true if the difference between desc1 and desc2 is at least one field.
bool < mdesc desc1 > == < mdesc desc2 >	Returns true if desc1 and desc2 have the same field values.

10.6. MOVEMENT INSTRUCTIONS

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

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

Function

This instruction records a command for a joint movement towards the **pPosition** or **jPosition** positions, using the **tTool** and the **mDesc** movement parameters. It returns the move id assigned to this movement, and increases by one the move id for the next move command.

CAUTION:

The system does not wait for the movement to be completed before proceeding to the next VAL 3 instruction: several movement commands can be stored in advance. When the system has used up all its available memory and has no room for another command, the instruction waits until the new command can be stored.

A detailed explanation of the movement parameters is given at the beginning of the chapter entitled "Movement control".

A runtime error is generated if **mDesc** contains invalid values, if **jPosition** is outside the software limits, if **pPosition** cannot be reached, or if a previously saved movement command cannot be run (destination out of reach).

See also

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

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

Function

This instruction records a command for a linear movement towards the **pPosition** point, using the **tTool** tool and the **mDesc** movement parameters. It returns the move id assigned to this movement, and increases by one the move id for the next move command.

CAUTION:

The system does not wait for the movement to be completed before proceeding to the next VAL 3 instruction: several movement commands can be stored in advance. When the system has used up all its available memory and has no room for another command, the instruction waits until the new command can be stored.

A detailed explanation of the movement parameters is given at the beginning of the chapter entitled "Movement control".

A runtime error is generated if **mDesc** contains invalid values, if **pPosition** cannot be reached, if a straight line movement towards **pPosition** is not possible or if a previously saved movement command cannot be run (destination out of reach).

See also

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

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

Function

This instruction records a command for a circular movement starting from the destination of the previous movement and finishing at point **pTarget** and passing through the point **pIntermediate**. It returns the move id assigned to this movement, and increases by one the move id for the next move command.

The tool orientation is interpolated in such a way that it is possible to program a constant orientation in absolute terms, or as compared with the trajectory.

CAUTION:

The system does not wait for the movement to be completed before proceeding to the next VAL 3 instruction: several movement commands can be stored in advance. When the system has used up all its available memory and has no room for another command, the instruction waits until the new command can be stored.

A detailed explanation of the various movement parameters and orientation interpolation can be found at the beginning of the "Movement Control" chapter.

A runtime error is generated if **mDesc** has invalid values, if point **pIntermediate** (or point **pTarget**) cannot be reached, if the circular movement is not possible (see the "Movement control - interpolation of orientation" chapter), or if a movement command recorded beforehand cannot be executed (destination out of reach).

See also

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

void stopMove()

Function

This instruction stops the arm on the trajectory and suspends authorization of the programmed movement.

CAUTION:
This instruction returns immediately: the VAL 3 task does not wait for the movement to be completed before proceeding to the next instruction.

The motion descriptor used to execute the stop are those used for the current movement.

The movements can only be resumed after a `restartMove()` or `resetMotion()` instruction.

Non-programmed movements (jog interface) are still possible.

Example

```
// waits for a signal
wait(diSignal==true)
// stops movements along the trajectory
stopMove()
wait(diSignal==false)
// restarts movements along the trajectory
restartMove()
```

See also

`void restartMove()`

`void resetMotion(), void resetMotion(joint jStartingPoint)`

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

Function

This instruction stops the arm on the trajectory and cancels all the stored movement commands. It resets the move id to zero.

CAUTION:
This instruction returns immediately: the VAL 3 task does not wait for the movement to be completed before proceeding to the next instruction.

The programmed movement authorization is restored if it was suspended by the `stopMove()` instruction.

If the `jStartingPoint` joint position is specified, the next movement command can only be run from this position: a connection movement must be performed beforehand to reach the `jStartingPoint` position.

If no joint position is specified, the next movement command is run from the arm's current position, wherever it is.

See also

`bool isEmpty()`

`void stopMove()`

`void autoConnectMove(bool bActive), bool autoConnectMove()`

`num setMoveId(num nMoveId)`

`joint resetTurn(joint jReference)`

void restartMove()

Function

This instruction restores the programmed movement authorization, and restarts the trajectory interrupted by the `stopMove()` instruction.

If the programmed movement authorization was not interrupted by the `stopMove()` instruction, this instruction has no effect.

See also

`void stopMove()`
`void resetMotion()`, `void resetMotion(joint jStartingPoint)`

void waitEndMove()

Function

This instruction cancels the blending of the last movement command recorded and waits for the command to be executed.

This instruction does not wait for the robot to be stabilized in its final position, it only waits until the position command sent to the drives corresponds to the desired final position. When it is necessary to wait for complete stabilization of the movement, the `isSettled()` instruction must be used.

A runtime error is generated if a previously stored movement cannot be run (destination out of reach).

Example

(see chapter 10.2)

See also

`bool isSettled()`
`bool isEmpty()`
`void stopMove()`
`void resetMotion()`, `void resetMotion(joint jStartingPoint)`

bool isEmpty()

Function

This instruction returns **true** if all the movement commands have been executed, returns **false** if at least one command is still being executed.

Example

This program cancels the recorded moves, if any:

```
// If commands are in progress
if isEmpty()==false
// Stop the robot and cancel the commands
resetMotion()
putln("Movements have been cancelled")
endif
```

See also

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

bool isSettled()

Function

This instruction returns **true** if the robot is stopped, and **false** if its position is not yet stabilized.

CAUTION:

The robot can be stopped for different reasons and may therefore be settled before all registered moves are completed. Use **isEmpty()** to know if the robot is stopped at the end of its programmed movement.

The position is considered as stabilized if the position error for each joint remains less than 1% of the maximum authorized position error, for 50 ms.

See also

bool isEmpty()
void waitEndMove()

void autoConnectMove(bool bActive), bool autoConnectMove()

Function

In the remote mode, the connection movement is automatic if the arm is very close to its trajectory (distance less than the maximum authorized drift error). If the arm is too far away from its trajectory, the connection movement is automatic or under manual control depending on the mode defined by the **autoConnectMove** instruction: automatically if **bActive** is **true**, in manual control mode if **bActive** is **false**. When called without parameters, **autoConnectMove** returns the current connection movement mode.

By default, the connection movement in remote mode is under manual control.

CAUTION:

Under normal conditions of use, the arm stops on its trajectory during an emergency stop. Hence in remote mode, the arm is able to restart automatically whatever the connection movement defined by the **autoConnectMove** instruction.

See also

void resetMotion(), void resetMotion(joint jStartingPoint)

num `getSpeed(tool tTool)`

Function

This instruction returns the current Cartesian translation speed at the TCPtTool of the specified tool tTool. The speed is computed from the joint velocity command and not from the joint velocity feedback.

See also

[point here\(tool tTool, frame fReference\)](#)

joint `getPositionErr()`

Function

This instruction returns the current joint position error of the arm. The joint position error is the difference between the joint position command sent to the drives and the joint position feedback measured by the encoders.

See also

`void getJointForce(num& nForce)`

void `getJointForce(num& nForce)`

Function

This instruction returns the current joint torque (N.m for revolute axis) or force (N for linear axis) computed from the motors currents.

The joint force is not a direct estimation of external efforts. It includes also gravity, friction, viscosity, inertia, noise and accuracy of current sensors, relation between motor current and torque. It can be used to estimate external efforts only by recording forces in reference conditions, and comparing them with forces measured in similar conditions with additional external efforts.

It returns only a order of magnitude for forces. There is no commitment on accuracy that must be evaluated with each application.

A runtime error is generated if the parameter is not an array of num with sufficient size.

See also

[joint getPositionErr\(\)](#)

num `getMoveId()`

Function

This instruction returns a numeric value that gives the current position of the robot on the path. The integer part identifies the number of the move instruction that is being executed. This integer is returned when the move instruction is executed. The decimal part gives the progress % on this move.

A move id should never be tested with the '==' operator, but with the '>=' operator: `wait(getMoveId()==12)` may never return, because the move id may increase in one step from 11.998 to 12.013 and never take exactly the expected value (12). You should write `wait(getMoveId())>=12` instead.

Example

This example shows how the move id changes on a simple path:

```
nIdA = move1(pA, tTool, mDesc)
nIdB = move1(pB, tTool, mDesc)
waitEndMove()
nId = getMoveId()
```

During execution of this program:

Suppose that the returned value nIdA is 15. Then nIdB is 16: the move id is automatically increased by one with each move instruction.

- when `getMoveId()` is 15.8, the robot position is at 80 % of the move to point pA.
- when `getMoveId()` is 16.572, the robot position is at 57.2 % of the move to point pB.
- when `getMoveId()` is 17, the robot position is at 100 % of the move 16, so it is at point pB.

The value of nId, after `waitEndMove()`, is therefore $nIdB+1=17$.

See also

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

`num move1(point pPosition, tool tTool, mdesc mDesc)`

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

`num setMoveId(num nMoveId)`

Function

This instruction changes the move id for the next move instruction. It is useful to arrange so that the same path always uses the same move id values. After a `resetMotion`, the move id is automatically reset to 0.

After using `setMoveId()` or `resetMotion()`, the relation between a move id and a move instruction may become uncertain: several recorded moves may then have the same move id. `setMoveId()` should therefore not be given a value that is also the move id of a pending move command.

Example

```
resetMotion()
nId1 = getMoveId()
setMoveId(1000)
nId2 = getMoveId()
nId3 = move1(pA, tTool, mDesc)
nId4 = move1(pB, tTool, mDesc)
waitEndMove()
nId5 = getMoveId()
```

After this program is executed, we have:

- nId1 is 0, because move id is set to 0 after `resetMotion()`
- nId2 is 0: move id was just changed with `setMoveId()`
- nId3 is 1000: a move instruction returns the move id previously defined, and increases it for the next move
- nId4 is 1001: the move id was increased by the previous move instruction
- nId5 is 1002: after `waitEndMove()`, 100 % of move 1001 is completed here, move id is then $1001+1 = 1002$

See also

`num getMoveId()`

`void resetMotion(), void resetMotion(joint jStartingPoint)`

CHAPTER 11

OPTIONS

11.1. COMPLIANT MOVEMENTS WITH FORCE CONTROL

11.1.1. PRINCIPLE

In a standard movement command, the robot moves to reach a requested position at a programmed rate of acceleration and speed. If the arm cannot follow the command, additional force will be requested from the motors in order to attempt to reach the desired position. When the deviation between the position set by the command and the true position is too great, a system error is generated that cuts off power to the robot arm.

The robot is said to be 'compliant' when it accepts certain deviation between the position set by command and the actual position. The controller can be programmed to be trajectory compliant, i.e. to accept a delay or advance in relation to the programmed trajectory, by controlling the force applied by the arm. On the other hand, no deviation in relation to the trajectory is allowed.

In practice, the **VAL 3**'s compliant movements can allow the arm to follow a trajectory while being pushed or pulled by an outside force, or come into contact with an object, with a check made on the force applied by the arm on the object.

11.1.2. PROGRAMMING

Compliant movements are programmed like standard movements, using the `movelf()` and `movejf()` instructions, with an additional parameter used to control the force applied by the arm. During the compliant movement, speed and acceleration limits are applied, in the same way as for standard movements, via the movement descriptor. The movement can take place along the trajectory, in either direction.

It is possible to combine compliant movements or combine compliant and standard movements: as soon as the destination position is reached, the robot moves on to the next movement. The `waitEndMove()` instruction is used to wait for the end of a compliant movement.

The `resetMotion()` instruction cancels all programmed movements, whether compliant or not. After `resetMotion()`, the robot is no longer compliant.

The `stopMove()` and `restartMove()` instructions also apply to compliant movements:

The `stopMove()` forces the current movement speed to zero. If it is a compliant movement, it is hence stopped and the robot is no longer compliant until the `restartMove()` instruction is run.

Lastly, the `isCompliant()` instruction is used to ensure that the robot is in compliant mode, for example before allowing any outside force to be applied to the arm.

11.1.3. FORCE CONTROL

When the specified force parameter is null, the arm is passive, i.e. it only moves when actuated by outside forces.

When the force parameter is positive, everything operates as though an outside force were pushing the arm to the position ordered: the arm moves on its own, but it can be held back or accelerated by outside action which is added to the force commanded.

When the force parameter is negative, everything operates as though an outside force were pushing the arm towards its initial position: to move the arm towards the position commanded, it is thus necessary to apply an outside force that is greater than the force commanded.

The force parameter is expressed as a percentage of the arm's nominal load. **100%** means that the arm applies a force towards the position commanded, that is equivalent to the nominal load. In rotation, **100%** corresponds to the nominal torque allowed on the arm.

When the arm's speed or acceleration reach the values specified in the movement descriptor, the robot opposes its full power to resist any attempt to increase its speed or rate of acceleration.

11.1.4. LIMITATIONS

Compliant movements require a specific robot tuning that is not available with all robots (consult your **Stäubli** contact).

Compliant movements present the following limitations:

- It is not possible to use blending at the start or the end of a compliant movement: the arm is bound to stop at the start and end of every compliant movement.
- When a compliant movement is made, the arm may move back to its starting point, but it cannot move back any further: the arm then stops suddenly at its starting point.
- The force parameter on the arm cannot exceed **1000%**. The precision obtained concerning the force applied is limited by internal friction. It depends mainly on the arm position and the trajectory commanded.
- Long compliant movements require a lot of internal memory capacity. A runtime error is generated if the system does not have enough memory to fully process the movement.

11.1.5. INSTRUCTIONS

num movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)

Function

This instruction records a compliant joint movement command towards the **jPosition** position using the **tTool** tool, the **mDesc** movement parameters, and a **nForce** force command. It returns the move id assigned to this movement, and increases by one the move id for the next move command

The **nForce** force command is a numerical value representing arm force and cannot exceed **±1000**. A value of 100 approximatively matches the weight of the nominal mass of the arm.

CAUTION:

The system does not wait for the movement to be completed before proceeding to the next VAL 3 instruction: several movement commands can be stored in advance. When the system has used up all its available memory and has no room for another command, the instruction waits until the new command can be stored.

A detailed explanation of the various movement parameters is given at the beginning of the section.

A runtime error is generated if **mDesc** or **nForce** have invalid values, if **jPosition** is outside the software limits, if the previous movement required blending or if a previously recorded movement command cannot be run (destination out of reach).

See also

num movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)
bool isCompliant()

num movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)

Function

This instruction records a compliant linear movement command towards the **pPosition** position using the **tTool** tool, the **mDesc** movement parameters and the **nForce** force command. It returns the move id assigned to this movement, and increases by one the move id for the next move command.

The **nForce** force command is a numerical value representing arm force and cannot exceed **±1000**. A value of 100 approximatively matches the weight of the nominal mass of the arm.

CAUTION:

The system does not wait for the movement to be completed before proceeding to the next VAL 3 instruction: several movement commands can be stored in advance. When the system has used up all its available memory and has no room for another command, the instruction waits until the new command can be stored.

A detailed explanation of the various movement parameters is given at the beginning of the section.

A runtime error is generated if **mDesc** or **nForce** have invalid values, if **pPosition** cannot be reached, if movement towards **pPosition** is impossible in a straight line, if the previous movement required blending or if a previously recorded movement command cannot be run (destination out of reach).

See also

num movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)

bool isCompliant()

bool isCompliant()

Function

This instruction returns **true** if the robot is in compliant mode, otherwise returns **false**.

Example

```
movelf(pPosition, tTool, mDesc, 0)
// Waits for the robot to actually be in compliant mode
wait(isCompliant())
// Commands press ejection
diEjection = true
// Waits for the end of compliant movement
waitEndMove()
// restart with a standard movement
movej(jjDepart, tTool, mDesc)
```

See also

num movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)

num movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)

11.2. ALTER: REAL TIME CONTROL ON A PATH

Cartesian Alter

11.2.1. PRINCIPLE

A Cartesian alteration of a path allows apply to the path a geometrical transformation (translation, rotation, rotation at the tool centre point) that is immediately effective.

This feature makes it possible to modify a nominal path using an external sensor, for, for example, track accurately the shape of a part, or operate on a moving part.

11.2.2. PROGRAMMING

The programming consists in defining first the nominal path, then, in real time, specifying a deviation to it.

The nominal path is programmed as for standard moves, with the `alterMoveI()`, `alterMoveJ()` and `alterMoveC()` instructions. Several alterable moves may succeed, or some alterable moves may alternate with not alterable moves. We will define the alterable path as the successive alterable move commands between two not alterable move commands.

The alteration itself is programmed with the `alter()` instruction. Different alter modes are possible depending on the geometrical transformation to apply; the mode is defined with the `alterBegin()` instruction. The `alterEnd()` instruction is finally needed to specify how to terminate the altering, either before the nominal move is completed, so that the next non alterable move can be sequenced without stop; either after, so that it remains possible to move the arm with alter while the nominal move is stopped.

The other motion control instructions remains effective in alter mode.

CAUTION:

The `waitEndMove`, `open` and `close` instructions wait for the end of the nominal move, not for the end of altered move. VAL 3 execution may therefore resume after a `waitEndMove` even if the arm is still moving because of a changing alteration.

11.2.3. CONSTRAINTS

Synchronisation, desynchronisation: Because the alter command is applied immediately, the change in the alteration must be controlled so that the resulting arm path remains without discontinuity or noise:

- A large change in the alteration can only be applied gradually with a specific approach control.
- The end of the altering requires a null alteration speed, obtained gradually with a specific stop control.

Synchronous command: The controller sends position and velocity commands every 4 ms to the amplifiers. As a consequence, the alter command must be synchronized with this communication period so that the alteration speed remains under control. This is done by using a synchronous **VAL 3** task (see Tasks chapter). In the same way, the sensor input may have to be filtered first if the data is noisy or if its sampling period is not synchronized with the controller period.

Smooth sequencing: The first non alterable move following an alterable path can be computed only when `alterEnd` is executed. As a consequence, if `alterEnd` is executed too near the end of the alterable move, the arm may slow down or even stop near this point, until the next move is computed.

Moreover, the ability to compute in advance the next move imposes some restrictions on the altered path after `alterEnd` is executed: It must then keep the same configuration, and make sure all joints remain in the same axis turn. It is then possible that an error is generated during the move that would not occur if `alterEnd` was not executed in advance.

11.2.4. SAFETY

At any time, the user alteration may be invalid: target out of reach, velocity or acceleration too high. When the system detects such situations, an error is generated and the arm is stopped suddenly at the last valid position. The motion needs to be reset to resume operation.

When the arm motion is disabled during a move (hold mode, stop request or emergency stop), a stop is controlled on the nominal move as for standard moves. After a certain delay, the alter mode is also automatically disabled to guaranty a complete stop of the arm. When the stop condition disappears, the move may resume and the alter mode is automatically enabled again.

11.2.5. LIMITATIONS

A null move (when the move target is on start position) is ignored by the system. As a consequence, you need a not null move to enter the alter mode. A move distance of 0.001 mm is enough for this.

It is not possible to specify the desired configuration for the altered path; the system always uses the same configuration. It is therefore not possible to change the configuration of the arm within an altered path (even with the alterMovej instruction).

11.2.6. INSTRUCTIONS

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

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

Function

This instruction records an alterable joint move command (a line in the joint space). It returns the move id assigned to this movement, and increases by one the move id for the next move command.

Parameter

jPosition/pPosition	Point or joint expression defining the end position of the move.
tTool	Tool expression defining the tool centre point used during the move for Cartesian speed control.
mDesc	mDesc expression defining the speed control and blending parameter for the move.

Details

This instruction behaves exactly as the movej instruction, except that it enables the alter mode for the move. See movej for more details.

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

Function

This instruction records an alterable linear move command (a line in the Cartesian space). It returns the move id assigned to this movement, and increases by one the move id for the next move command.

Parameter

pPosition	Point expression defining the end position of the move.
tTool	Tool expression defining the tool centre point used during the move for Cartesian speed control. At the end of the move, the tool centre point is at the specified target position.
mdesc	mdesc expression defining the speed control and blending parameter for the move.

Details

This instruction behaves exactly as the moveI instruction, except that it enables the alter mode for the move. See moveI for more details.

num alterMoveC(point pIntermediate, point pTarget, tool tTool, mdesc mDesc)

Function

This instruction records an alterable circular move command. It returns the move id assigned to this movement, and increases by one the move id for the next move command.

Parameter

pIntermediate	Point expression defining an intermediate point on the circle
pTarget	Point expression defining the end position of the move.
tTool	Tool expression defining the tool centre point used during the move for Cartesian speed control. At the end of the move, the tool centre point is at the specified target position.
mdesc	mdesc expression defining the speed control and blending parameter for the move.

Details

This instruction behaves exactly as the moveC instruction, except that it enables the alter mode for the move. See moveC for more details.

num alterBegin(frame fAlterReference, mdesc mMaxVelocity)

num alterBegin(tool tAlterReference, mdesc mMaxVelocity)

Function

This instruction initializes the alter mode for the alterable path being executed.

Parameter

fAlterReference/tAlterReference	Frame or tool expression defining the reference for the alter deviation.
mMaxVelocity	mdesc expression defining the safety check parameters for the alter deviation.

Details

The alter mode initiated with alterBegin terminates only with an alterEnd command, or a resetMotion. When the end of an alterable path is reached, the alter mode remains active until alterEnd is executed.

The trsf expression of the alter command defines a transformation of the whole path around alterReference:

- The path is rotated around the centre of the frame or tool using the rotation part of the trsf.
- Then the path is translated by the translation part of the trsf.

The trsf coordinates of the alter command are defined in alterReference base.

When a frame is used as reference, the alterReference is fixed in space (World). This mode must be used when the deviation of the path is known or measured in the Cartesian space (moving part such as conveyor tracking).

When a tool is used as reference, the alterReference is fixed relatively to the tool center point. This mode must be used when the deviation of the path is known or measured relatively to the tool center point (for example part shape sensor mounted on the tool).

The motion descriptor is used to define the maximum joint and Cartesian velocity on the altered path (using the fields vel, tvcl and rvel of the motion descriptor). An error is generated and the arm is stopped on path if the altered velocity exceeds the specified limits.

The accel and decel fields of the motion descriptor control the stop time when a stop condition occurs (eStop, hold mode, VAL 3 stopMove()): The path alteration must be stopped using these deceleration parameters (see alterStopTime).

alterBegin returns a numerical value to indicate the result of the instruction:

- 1 alterBegin was successfully executed
- 0 alterBegin is waiting for the start of the alterable move
- 1 alterBegin was ignored because the alter mode has already started
- 2 alterBegin is refused (alter option is not enabled)
- 3 alterBegin was refused because the motion is in error. A resetMotion is required.

See also

num alterEnd()
num alter(trsf trAlteration)
num alterStopTime()

num alterEnd()

Function

This instruction exits the alter mode and make the current move not alterable any more.

Details

If alterEnd is executed when the end of the alterable path is reached, the next not alterable move (if any) is started immediately.

If alterEnd is executed before the end of the alterable move, the current value of the alter deviation is applied to the rest of the alterable path, until the first next not alterable move. It is not possible to enter the alter mode again on the same alterable path.

The next not alterable move, if any, is computed as soon as alterEnd is executed so that the transition between the alterable path and the next not alterable move is made without stop.

alterEnd returns a numerical value to indicate the result of the instruction:

- 1 alterEnd was successfully executed
- 1 alterEnd was ignored because the alter mode has not yet started
- 3 alterEnd was refused because the motion is in error. A resetMotion is required.

See also

num alterBegin(frame fAlterReference, mdesc mMaxVelocity)

num alterBegin(tool tAlterReference, mdesc mMaxVelocity)

num alter(trsf trAlteration)

Function

This instruction specifies a new alteration of the alterable path.

Details

The transformation induced by the alteration trsf depends on the alter mode selected by the alterBegin instruction. The alteration coordinates are defined in the frame or tool specified with the alterBegin instruction.

The alteration is applied by the system every 4 ms: When several alter instructions are executed in less than 4 ms, the last one applies. Most often the alter instruction needs to be executed in a synchronous task to force an alteration refresh every 4 ms.

The alteration must be computed carefully so that the resulting arm position and speed commands remain continuous and without noise. A sensor input may need to be filtered adequately to reach the desired quality on the arm path and behaviour.

When the motion is stopped (hold mode, emergency stop, stopMove() instruction), the alteration of the path is locked until all stop conditions are cleared.

When the alteration of the path is invalid (unreachable position, out of speed limits), the arm will stop suddenly at the last valid position and the alter mode is locked in error. A resetMotion is required to resume operation. The velocity limits for the alter move are defined by the alterBegin instruction.

Alter returns a numerical value to indicate the result of the instruction:

- 1 alter was successfully executed.
- 0 alter is waiting for the motion to restart (alterStopTime is null).
- 1 alter was ignored because the alter mode is not started or already ended.
- 2 alter is refused (alter option is not enabled).
- 3 alter was refused because the motion is in error. A resetMotion is required.

See also

num alterBegin(frame fAlterReference, mdesc mMaxVelocity)

num alterBegin(tool tAlterReference, mdesc mMaxVelocity)

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

num alterStopTime()

Function

This instruction returns the remaining time before the alter deviation is locked, when a stop condition occurs.

Details

When a stop condition occurs, the system evaluates the time to stop the arm if the accel and decel parameters of the motion descriptor specified with alterBegin are used. The minimum of this time and the time imposed by the system (typically 0.5s when a eStop occurs) is returned by alterStopTime.

When alterStopTime returns a negative value, there is no pending stop condition. When alterStopTime returns null, the alter command is locked until all stop conditions are reset.

alterStopTime returns null when the alter mode is not enabled.

See also

num alterBegin(frame fAlterReference, mdesc mMaxVelocity)

num alterBegin(tool tAlterReference, mdesc mMaxVelocity)

num alter(trsf trAlteration)

11.3. OEM LICENCE CONTROL

11.3.1. PRINCIPLES

An OEM licence is a controller-specific key that makes it possible to restrict the use of a **VAL 3** project on some selected robot controllers.

A tool is provided with **Stäubli Robotics Suite**(*) to encode a secret OEM password into a public, controller specific, OEM licence, that can then be installed as a software option on the controller. By using the **getLicence()** instruction, a project or library can test if the OEM licence is installed and therefore make sure that it is used only by the selected robot controllers.

To keep the OEM password secret and protect the code where the licence is tested, the **getLicence()** instruction must be used in an encrypted library.

Demonstration mode of OEM licences is supported; in that case, the controller is simply configured with the "demo" key, and the **getLicence()** instruction notifies it to the caller. With the **VAL 3** emulator, the "demo" key is enough to fully enable the OEM licence.

The **getLicence()** instruction is a **VAL 3** option and requires the installation of a runtime licence on the controller. If this runtime licence is not defined, **getLicence()** returns an error code.

(*) This tool, a `encrypttools.exe` executable, requires a specific **Stäubli Robotics Suite** licence to be used.

11.3.2. INSTRUCTIONS

string getLicence(string sOemLicenceName, string sOemPassword)

Function

This instruction returns the status of the specified OEM licence:

"oemLicenceDisabled"	The VAL 3 runtime licence "oemLicence" is not enabled on the controller: the OEM licence cannot be tested.
""	The OEM licence sOemLicenceName is not enabled (undefined, or invalid password).
"demo"	The OEM licence sOemLicenceName is enabled in demonstration mode.
"enabled"	The OEM licence sOemLicenceName is enabled.

See also Encryption

11.4. ABSOLUTE ROBOT

11.4.1. PRINCIPLE

An 'absolute robot' is a robot using an arm-specific identification of the geometrical parameters (often known as 'DH parameters'). These parameters are specific for each robot and correspond to the real orientation and dimensions of each joint. An absolute robot has an increased accuracy to reach Cartesian positions specified by a CAD tool or computed in VAL 3 (such as positions in a pallet). Cartesian curves (long lines, circles) are also more accurate. Absolute calibration does not change the repeatability of the arm.

The DH parameters consists in a set of translations (a, b and d along axes X, Y, Z,) and rotations (alpha, beta, theta around axes X, Y and Z)

The sequence of these translations and rotations is defined so that the joint position {j1, j2, j3, j4, j5, j6} matches the Cartesian position pCart at flange center point with:

```
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}
```

The d[6] parameter is required only for specific arm wrists.

11.4.2. OPERATION

The geometrical parameters must be identified using a separate measurement tool (such as laser tracker). The VAL 3 language does not offer tools to support this measurement operation, but it gives a mean to apply the measured geometrical parameters to the robot with immediate effect. The parameters are also saved in the arm configuration file, so that they are automatically recovered with the next reboot.

The geometrical parameters can be changed in a running VAL 3 application. This makes it possible to adjust the geometry of the arm during the production cycle if the robot cell includes an adequate measurement tool.

11.4.3. LIMITATIONS

It is possible to change the geometrical parameters in a running VAL 3 application only when the motion generator is empty (no pending move).

The modified geometry of an absolute robot has more complex mathematical properties than a standard geometry. The notion of arm configuration (configuration type) cannot be made rigorous any more. The conversion of a Cartesian position into a corresponding joint position may fail near joint limits or singular positions.

11.4.4. INSTRUCTIONS

```
void getDH (num& theta[], num& d[], num& a[], num& alpha[],
              num& beta[])
```

```
void getDefaultDH(num& theta[], num& d[], num& a[], num& alpha[],
                   num& beta[])
```

Function

These instructions return in the specified arrays the DH parameters of the arm. Parameters d and a are translations in mm; parameters theta, alpha and beta are angles in degrees. `getDH()` return the current DH parameters of the arm attached to the controller. `getDefaultDH()` returns the standard DH parameters for the arm type.

See also

```
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[])
```

Function

This instruction is enabled only with the a specific runtime license. It modifies the arm geometry with DH parameters. Parameters d, a and b are translations in mm ; parameters theta, alpha and beta are angles in degree. The change is immediate, and is also applied to the arm configuration file so that the modified geometry is effective with the next reboot.

The instruction returns true if the change is successfully applied, false if the new geometry is not applied because it differs too much from the standard geometry parameters. `setDH()` waits for the motion to be empty to perform its operation.

The size of DH arrays must match the number of robot axes. An additional entry in the d array may be required to modify the flange dimension: when it is missing, `setDH()` returns false and a diagnostic message is sent to the logger.

See also

```
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. CONTINUOUS AXIS

11.5.1. PRINCIPLE

The axis 6 (for RX/TX arms) or axis 4 (for RS/TS arms) can be 'continuous' in a robotics application when only its position within one turn matters: the number of turns it has performed during the cycle has no importance. This applies for example to applications where the robot is holding a part that is processed by a fixed tool.

The continuousAxis option makes it possible to automatically reset the number of turns performed in the previous cycle before a new cycle is started. For example, if the axis starts the application cycle at position 0° and ends at position 720° (2 turns), the next cycle can reset instantaneously the position to 0° and start a new cycle, without having to move the axis back from 720° to 0°.

11.5.2. INSTRUCTIONS

`joint resetTurn(joint jReference)`

Function

This instruction performs like the standard `resetMotion()` instruction, waits for the arm to be stopped, and adjusts the position of the continuous axis so that it becomes as close as possible to the specified reference position. It returns the effective arm position after execution of the instruction. The adjustment operation is done with arm power enabled or not, and takes roughly 50 ms. The `resetTurn()` instruction changes the calibration data of the arm. With the next controller reboot, the axis zero position is automatically reinitialized (update of the arm specific data in the arm and on the controller disk).

See also

`void resetMotion()`, `void resetMotion(joint jStartingPoint)`

CHAPTER 12

APPENDIX

12.1. RUNTIME ERROR CODES

Code	Description
-1	There is no task created by this application or library with the specified name
0	The task is suspended without runtime error (taskSuspend() instruction or debug mode)
1	The specified task is running
10	Invalid numerical calculation (division by zero).
11	Invalid numerical calculation (e.g.ln(-1))
20	Access to an array with an index that is larger than the array size.
21	Access to an array with a negative index.
29	Invalid task name. See taskCreate() instruction.
30	The specified name does not correspond to any VAL 3 task.
31	A task with the same name already exists. See taskCreate instruction.
32	Only 2 different periods for synchronous tasks are supported. Change scheduling period.
40	Not enough memory space available.
41	Not enough memory space to run the task. See the run memory size.
60	Maximum instruction run time exceeded.
61	Internal VAL 3 interpreter error
70	Invalid instruction parameter. See the corresponding instruction.
80	Uses data or a program from a library not loaded in the memory.
81	Incompatible kinematic: Use of a point/joint/config that is not compatible with the arm kinematic.
82	The reference frame or tool of a variable belongs to a library and is not accessible from the variable's scope (library not declared in the variable's project, or reference variable is private).
90	The task cannot resume from the location specified. See taskResume() instruction.
100	The speed specified in the motion descriptor is invalid (negative or too great).
101	The acceleration specified in the motion descriptor is invalid (negative or too great).
102	The deceleration specified in the motion descriptor is invalid (negative or too great).
103	The translation velocity specified in the motion descriptor is invalid (negative or too great).
104	The rotation velocity specified in the motion descriptor is invalid (negative or too great).
105	The reach parameter specified in the movement descriptor is invalid (negative).
106	The leave parameter specified in the movement descriptor is invalid (negative).
122	Attempt to write in a system input.
123	Use of a dio, aio or sio input/output not connected to a system input/output.
124	Attempt to access a protected system input/output
125	Read or write error on a dio, aio or sio (field bus error)
150	Cannot run this movement instruction: a previous movement request could not be completed (point out of reach, singularity, configuration problem, etc.)
153	Movement command not supported
154	Invalid movement instruction: check the movement descriptor.
160	Invalid flange tool coordinates
161	Invalid world tool coordinates
162	Use of a point without a reference frame. See Definition.
163	Use of a frame without a reference frame. See Definition.
164	Use of a tool without reference tool. See Definition.
165	Invalid frame or reference tool (global variable linked to a local variable)
250	No runtime licence for this instruction, or demo licence is over.

12.2. CONTROL PANEL KEYBOARD KEY CODES

Without Shift					With Shift				
3	Caps	Space			3	Caps	Space		
283	-	32			283	-	32		
2	Shift	Esc	Help	Ret.	2	Shift	Esc	Help	Ret.
282	-	255	-	270	282	-	255	-	270
	Menu	Tab	Up	Bksp		Menu	UnTab	PgUp	Bksp
	-	259	261	263		-	260	262	263
1	User	Left	Down	Right	1	User	Home	PgDn	End
281	-	264	266	268	281	-	265	267	269

Menus (with or without Shift):

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

For standard keys, the code returned is the **ASCII** code of the corresponding character:

Without 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

With 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

With double 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

ILLUSTRATION

Ambiguity as to the intermediate orientation	149
Blended cycle	146
Blended cycle	153
Circular movement	144
Configuration change: righty / lefty	150
Configuration: enegative	139
Configuration: epositive	139
Configuration: lefty	138
Configuration: lefty	140
Configuration: righty	138
Configuration: righty	140
Configuration: wnegative	139
Configuration: wpositive	139
Constant orientation as compared with the trajectory	148
Constant orientation in absolute terms	148
Cycle type: U	144
Cycle without blending at a given point	146
Definition of the distances: 'leave' / 'reach'	145
Elbow configuration change impossible	151
Frame rotation about the axis: X	122
Frame rotation about the axis: Y'	123
Frame rotation about the axis: Z''	123
Full circle	149
Initial and final positions	143
Links between reference frames	127
Links between tools	129
Organization chart: frame / point / tool / trsf	117
Orientation	122
Point definition	132
Positive/negative elbow configuration change	150
Positive/negative wrist configuration change	151
Sequencing	84
Shoulder configuration change possible	152
Straight line movement	143
Two configurations that can be used to reach a given point: P	136
User page	71

INDEX

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) 33
 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
 ln (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 (funcion) 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
setMoveId (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
waitEndMove 84, 146, 167
waitEndMove (Instruction) 160
watch 84
watch (Instruction) 94
while (Instruction) 24
wnegative 139
workingMode (Instruction) 110
wpositive 139
wrist 136

