ROBOTICS

MAN AND MACHINE

VAL3 REFERENCE MANUAL

Version 6

D28065204B - 01/2008

STÄUBLI

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

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

VAL3 is a high-level programming language designed to control **Stäubli** robots in all kinds of applications.

VAL3 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 **VAL3** instructions which fall into the following categories:

- Language elements
- Simple types
- User interface
- Tasks
- Libraries
- Robot control
- Arm position
- Movement control

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

CHAPTER 2

VAL3 LANGUAGE ELEMENTS

VAL3 consists of the following elements:

- applications
- programs
- libraries
- data types
- constants
- variables (global data, local data and parameters)
- tasks

2.1. APPLICATIONS

2.1.1. DEFINITION

A **VAL3** application is a self-contained software package designed for controlling robots and inputs/outputs associated with a controller.
A **VAL3** application comprises the following elements:

- a set of **programs**: the **VAL3** instructions to be executed
- a set of **global data**: the variables used by all programs
- a set of **libraries**: the outside instructions and variables used by the application

When an application is running, it also contains:

- a set of **tasks**: the programs being executed in parallel

2.1.2. DEFAULT CONTENT

A **VAL3** application always contains the **start()** and **stop()** programs, a **world** frame (**frame** type) and a **flange** tool (**tool** type).
When a **VAL3** application is created, it also contains the instructions and data types that are specific to the arm model.
Further details of these elements can be found in the chapters describing each element type.

2.1.3. START/STOP

VAL3 instructions are not used to control applications: applications can only be loaded, unloaded, started and stopped via the **MCP** user interface of the controller.
When a **VAL3** application is ran, its **start()** program is executed.
A **VAL3** 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 **VAL3** 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 **VAL3** application:

- unit of length
- amount of stack memory

These parameters cannot be accessed via a **VAL3** instruction and can only be changed via the **MCP** user interface or using the **SRS** editor.

2.1.4.1. UNIT OF LENGTH

In **VAL3** applications, the unit of length is either millimeter or inch. It is used by the **VAL3** geometrical data types: frame, point, 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 **VAL3** 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.
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 the **SRS** editor, 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 **VAL3** instructions to be executed.
A program consists of the following elements:

- The sequence of **instructions**: the **VAL3** 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.

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 concurrently by several tasks. Each instance of a program has its own unique local variables and parameters.

2.2.3. START() PROGRAM

The **start()** program is the program called when the **VAL3** 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, if 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 **VAL3** 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.

2.3. DATA TYPES

2.3.1. DEFINITION

A **VAL3** variable or constant type is a characteristic that allows the system to control the applications and programs that can use it.
All the **VAL3** constants and variables have a type. This enables the system to run an initial check when editing a program and hence detect certain programming errors immediately.

2.3.2. SIMPLE TYPES

The **VAL3** language supports the following simple types:

- **bool** type: for Boolean values (true/false)
- **num** type: for numeric values
- **string** type: for character strings
- **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

2.3.3. STRUCTURED TYPES

Structured types combine typed data, the fields of the structured type. Fields of the structured type can be accessed individually by their name.
The **VAL3** language supports the following structured 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 revolute positions
- **config** type: for robot configurations
- **mdesc** type: for robot movement parameters

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. All variables must be declared in the global data section or the local data section before they can be used.

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 values in its fields. The sequence order is specified in the chapter describing each structured type. The '{ }' symbols can be used to indicate a structure.

Example

In this example, p is a point variable and dummy a program taking a trsf and a dio data as parameters.

```
p = {{100, -50, 200, 0, 0, 0}, {sfree, efree, wfree}}
call dummy({a+b, 2* c, 120, limit(c, 0, 90), 0, 0}, io:valve1)
```

2.4.3. ARRAY OF CONSTANTS

An array must be initialized entry by entry.

Example

In this example, j is a 5-element array of joints.

```
// For 6 axis arms
j[0] = {0, 0, 0, 0, 0, 0}
j[1] = {90, 0, 90, 0, 0, 0}
j[2] = {-90, 0, 90, 0, 0, 0}
j[3] = {90, 0, 0, -90, 0, 0}
j[4] = {-90, 0, 0, -90, 0, 0}
```

2.5. VARIABLES

2.5.1. DEFINITION

A variable is a data item referenced by its name in a program.
A variable is identified by:

- its name: a character string
- its type: one of the **VAL3** types described previously
- its size: for an array, the number of elements it contains
- its scope: the program or programs that can use the variable

A variable name is a string of **1** to **15** characters selected from **"a..zA..Z0..9_"**, and starting with a letter.
All variables can be used as arrays. Simple variables are size **1**. The **size()** instruction returns the size of a variable.

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.
Program parameters are local variables that can only be accessed in the program in which they are declared.

2.5.3. ACCESSING A VARIABLE VALUE

The elements of an array can be accessed by using an index between square brackets **'['** and **']'**. The index must be between **0** and (size-**1**), otherwise a runtime error is generated.
If no index is specified, the index **0** is used: **var[0]** is equivalent to **var**.
The fields of structured type variables can be accessed using a **'.'** followed by the field name.

Example

```
num a                  // a is a size 1 num type variable
num b[10]              // b is a size 10 num type variable
trsf t
point p

a = 0                  // Initialization of a simple type variable
a[0] = 0               // Correct: equivalent to a = 0
b[0] = 0               // Initialization of the first element in array b
b = 0                  // Correct: equivalent to b[0] = 0
b[5] = 5               // Initialization of the sixth element in array b
b[5.13] = 7            // Correct: equivalent to b[5] = 7 (only the integer part is used)

b[-1] = 0              //  error: index less than 0
b[10] = 0              //  error: index too high (an array of size 10 has indices in 0..9)

t = p.trsf             // Initialization of t
p.trsf.x = 100         // Initialization of the x field of the trsf field of the p variable
```

2.5.4. PARAMETER PASSED "BY VALUE"

When a parameter is passed "by value", the system creates a local variable and initializes it with the value of the variable or expression supplied by the calling program.
The variables of the calling program used as "by value" parameters do not change, even if the called program changes the value of the parameter.
A data array cannot be passed by value.

Example:

```
program dummy(num x)      // x is passed by value
begin
  x=0
  putln(x)                // displays 0
end

num a
a=10
putln(a)                  // displays 10
call dummy(a)             // displays 0
putln(a)                  // displays 10: a is not modified by dummy()
```

2.5.5. PARAMETER PASSED "BY REFERENCE"

When a parameter is passed "by reference", the program no longer works on a copy of the data item passed by the caller, but on the data item itself, which is simply renamed locally.

The values of the variables of the calling program used as "by reference" parameters change when the called program changes the value of the parameter.

All the components of an array passed by reference can be used or modified. If an array component is passed by reference, that component and all following components can be used and modified. In this case, the parameter is seen as an array that starts with the component passed by the call. The **size()** instruction can be used to determine the effective parameter size.

When a constant or an expression are passed "by reference", the corresponding assigned parameter has no effect: the parameter retains the value of the constant or expression.

Example:

```
program dummy(num& x)           // x is passed by reference
begin
  x=0
  putln(x)                      // displays 0
end

program element(num& x)
begin
  x[3] = 0
  putln(size(x))
end

num a
num b[10]
a=10
putln(a)                        // displays 10
call dummy(a)                   // displays 0
putln(a)                        // displays 0: a is modified by dummy()
b[2] = 2
b[5] = 5
call element(b[2])              // displays 8, elements 0 and 1 in b are not passed
putln(b[5])                     // displays 0: b[5] was linked to x[3] and modified in program
                                element()
```

2.6. SEQUENCE CONTROL INSTRUCTIONS

Comment //

Syntax

// <String>

Function

A line starting with « // » is not evaluated and the evaluation 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

Runs the specified program with the specified parameters.

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 (i, j)
    call place (i, j)
  endFor
endFor
```

return program

Syntax

return

Function

Exits 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. A return happens automatically at the end of a program**.**

if control instruction

Syntax

if <bool bCondition>
<instructions>
S6.4 **[elseIf <bool bAlternateCondition1>**
<instructions>]
../..
S6.4 **[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 evaluated, 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 evaluated (if the **else** 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 better 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 **Condition** expression is **(true)**.
If the Boolean **Condition** 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

```
dio diLamp
// Causes a signal to flash while the robot is working
diLamp = false
while (isSettled()==false)
  diLamp = !diLamp              //Inverses the value of the diLamp: true false
  delay(0.5)                    // Waits ½ s
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

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

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 **nCounter** exceeds **nEnd** if it is greater than **nEnd**. If **nStep** is negative, the **nCounter** exceeds **nEnd** if it 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

```
num i
joint jDest
jDest = {0,0,0,0,0,0}
// Rotates axis 1 from 90° to -90° in -10-degree steps
for i = 90 to -90 step -10
  jDest.j1 = i
  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 evaluated, 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 evaluated (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...elself...else...endlf** sequence.
S6.4 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 **key strike** 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
  // Weekend !
  nDay=0
  break
endIf
```

CHAPTER 3

SIMPLE TYPES

3.1. INSTRUCTIONS

num **size**(variable)

Syntax

num size(<variable>)

Function

Returns the size of the **variable**.
If the **variable** is a program parameter passed by reference, the size depends on the index specified when calling up the program. If the **variable** is a single ^{2} and not an array, the size will be 1.

Parameter

variable	variable of any type

Example

```
num nArray[10]
program printSize(num& nparameter)
begin
  putln(size(nparameter))
end
call printSize(nArray)              // displays 10
call printSize(nArray[6])           // displays 4
```

S6.4 num **getData**(string sDataName, & variable)

Syntax

num getData(<string sDataName>, <& variable>)

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 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 an array

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

Returned value	Description
n > 0	The variable was successfully updated with n entries copied
-1	The data does not exists
-2	The library identifier does not exist
-3	The index is out of range
-4	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 specified by a string sPart, into a single local array pPath.

```
i = getData(sPart+":pApproach", pPath)
if(i > 0)
  nPoints = i
  i = getData(sPart+":pTrajectory", pPath[nPoints])
  if(i >0)
    nPoints=nPoints+i
  endIf
endIf
if (i<0)
  putln("Missing data in part  "+sPart)
endIf
```

3.2. BOOL TYPE

3.2.1. 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**.

3.2.2. 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 VAL3 expressions used as instruction parameter.

3.3. NUM TYPE

3.3.1. 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.

Example

```
putln(sel(cos(90)==0,1,-1))                    // displays -1
putln(sel(abs(cos(90))<0.000000000000001,1,-1))  // displays 1
```

The format of numerical type constants is as follows:

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

Example

```
1
0.2
-3.141592653
```

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

3.3.2. 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>	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 VAL3 expressions used as instruction parameter.

3.3.3. INSTRUCTIONS

num **sin**(num nAngle)

Syntax

num sin(<num nAngle>)

Function

Returns the sine of **nAngle**.

Parameter

num nAngle	angle in degrees

Example

```
putln(sin(30))                    // displays 0.5
```

num **asin**(num nValue)

Syntax

num asin(<num nValue>)

Function

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

A runtime error is generated if **nValue** is greater than **1** or less than **-1**.

Parameter

num nValue	Numerical expression

Example

```
putln(asin(0.5))            // displays 30
```

num **cos**(num nAngle)

Syntax

num cos(<num nAngle>)

Function

Returns the cosine of **Angle**.

Parameter

num nAngle	angle in degrees

Example

```
putln(cos(60))              // displays 0.5
```

num **acos**(num nValue)

Syntax

num acos(<num nValue>)

Function

Returns the inverse cosine of **nValue**, in degrees. The resulting angle is between **0** and **180** degrees.

A runtime error is generated if **nValue** is greater than **1** or less than **-1**.

Parameter

num nValue	Numerical expression

Example

```
putln(acos(0.5))            // displays 60
```

num **tan**(num nAngle)

Syntax

num tan(<num nAngle>)

Function

Returns the tangent of **Angle**.

Parameter

num nAngle	angle in degrees

Example

```
putln(tan(45))                    // displays 1.0
```

num **atan**(num nValue)

Syntax

num atan(<num nValue>)

Function

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

Parameter

num nValue	Numerical expression

Example

```
putln(atan(1))                    // displays 45
```

num **abs**(num nValue)

Syntax

num abs(<num nValue>)

Function

Returns the absolute value of **nValue**.

Parameter

num nValue	Numerical expression

Example

```
putln(sel(abs(45)==abs(-45),1,-1))   // displays 1
```

num **sqrt**(num nValue)

Syntax

num sqrt(<num nValue>)

Function

Returns the square root of **nValue**.

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

Parameter

num nValue	Numerical expression

Example

```
putln(sqrt(9))                    // displays 3
```

num **exp**(num nValue)

Syntax

num exp(<num nValue>)

Function

Returns the exponential function of **nValue**.

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

Parameter

num nValue	Numerical expression

Example

```
putln(exp(1))                     // displays 2.718282
```

num **power**(num nX, num nY)

Syntax

num power(<num nX>, <num nY>)

Function

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

An 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)

Syntax

num ln(<num nValue>)

Function

Returns the natural logarithm of **nValue**.

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

Parameter

num nValue	Numerical expression

Example

```
putln(ln(2.718281828))          // displays 1
```

num **log**(num nValue)

Syntax

num log(<num nValue>)

Function

Returns the common logarithm of **nValue**.

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

Parameter

num nValue	Numerical expression

Example

```
putln(log(10))                  // displays 1
```

num **roundUp**(num nValue)

Syntax

num roundUp(<num nValue>)

Function

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

Parameter

num nValue	Numerical expression

Example

```
putln(roundUp(7.8))          // Displays 8
putln(roundUp(-7.8))         // Displays -7
```

num **roundDown**(num nValue)

Syntax

num roundDown(<num nValue>)

Function

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

Parameter

num nValue	Numerical expression

Example

```
putln(roundDown(7.8))        // Displays 7
putln(roundDown(-7.8))       // Displays -8
```

num **round**(num nValue)

Syntax

num round(<num nValue>)

Function

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

Parameter

num nValue	Numerical expression

Example

```
putln(round(7.8))            // Displays 8
putln(round(-7.8))           // Displays -8
```

num **min**(num nX, num nY)

Syntax

num min(<num nX>, <num nY>)

Function

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

Parameter

num nX	Numerical expression
num nY	Numerical expression

Example

```
putln(min(-1,10))              // Displays -1
```

num **max**(num nX, num nY)

Syntax

num max(<num nX>, <num nY>)

Function

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

Parameter

num nX	Numerical expression
num nY	Numerical expression

Example

```
putln(max(-1,10))              // Displays 10
```

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

Syntax

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

Function

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

Parameter

num nValue	Numerical expression
num nMin	Numerical expression
num nMax	Numerical expression

Example

```
putln(limit(30,-90,90))        // displays 30
putln(limit(100,90,-90))       // displays 90
putln(limit(-100,-90,90))      // displays -90
```

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

Syntax

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

Function

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

Parameter

bool bCondition	Boolean expression
num nValue1	Numerical expression
num nValue2	Numerical expression

Example

```
putln(sel(bFlag,a,b))
// is equivalent to
if bFlag==true
  putln(a)
else
  putln(b)
endIf
```

S6.4 3.4. BIT FIELD TYPE

3.4.1. 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). **VAL3** 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 **VAL3** 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.

3.4.2. OPERATORS

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

3.4.3. INSTRUCTIONS

num **bAnd**(num nBitField1, num nBitField2)

Syntax

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 (nBitField and nMask)==nMask
    put("1")
  else
    put("0")
  endIf
endFor
putln("")
```

num **bOr**(num nBitField1, num nBitField2)

Syntax

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)

Syntax

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 **bNot**(num nBitField)

Syntax

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

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

Syntax

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 **VAL3** 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 **tShiftOut** into a byte array **nByteOut** and sends it with the **tcpClient** serial connection. The second program reads the bytes from the **tcpServer** serial connection and convert them back into a **trsf tShiftIn**.

```
// ---- Program to send a trsf ----
// Copy the trsf coordinates into a numerical buffer
nTrsfOut[0]=tShiftOut.x
nTrsfOut[1]=tShiftOut.y
nTrsfOut[2]=tShiftOut.z
nTrsfOut[3]=tShiftOut.rx
nTrsfOut[4]=tShiftOut.ry
nTrsfOut[5]=tShiftOut.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(io:tcpClient, nByteOut)


// ---- Program to read a trsf ----
nb=i=0
while (nb<48)
  nb=sioGet(io:tcpServer, 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
  tShiftIn.x=nTrsfIn[0]
  tShiftIn.y=nTrsfIn[1]
  tShiftIn.z=nTrsfIn[2]
  tShiftIn.rx=nTrsfIn[3]
  tShiftIn.ry=nTrsfIn[4]
  tShiftIn.rz=nTrsfIn[5]
endIf
```

3.5. STRING TYPE

3.5.1. 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.

3.5.2. 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 **VAL3** expressions used as instruction parameter.

3.5.3. INSTRUCTIONS

string **toString**(string sFormat, num nValue)

Syntax

string toString(<string sFormat>, <num nValue>)

Function

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**.

Parameter

string sFormat	Character string type expression
num nValue	Numerical expression

Example

```
num nPi
nPi = 3.141592654
putln(toString(".4", nPi))          // displays «3.1416»
putln(toString("8", nPi))           // displays «       3»
putln(toString("8.4", nPi))         // displays «  3.1416»
putln(toString("8.4", 2.70001))     // displays «  2.7   »
putln(toString("", nPi))            // displays «3»
putln(toString("1.2", 1234.1234))   // displays «1234.12»
```

See also

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

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

Syntax

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

Function

Computes 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**.

Parameter

string sString	Character string type expression
num& nValue	**num** type variable
bool& bReport	**bool** type variable

Example

```
num nVal
bool bOk
putln(toNum("10 20 30", nVal, bOk))    //  displays «20 30», nVal equals 10, bOk equals true
putln(toNum("a10 20 30", nVal, bOk))   //  displays «a10 20 30», nVal is unchanged, bOk equals
                                       false
putln(toNum("10 end", nVal, bOk))      //  displays «», nVal equals 10, bOk equals true
buffer = "+90 0.0 -7.6 17.3"
do
  buffer = toNum(buffer, nVal, bOk)
  putln(nVal)                          //  displays successively 90, 0, -7.6, 17.3
until (bOk != true)
```

See also

string toString(string sFormat, num nValue)

string **chr**(num nCodePoint)

Syntax

string chr(<num nCodePoint>)

Function

Returns the string made up of the specified Unicode code point character, if it is a valid Unicode code point. Otherwise 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 **VAL3** 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

Parameter

num nCodePoint	Expression of **num** type

Example

```
putln(chr(65))                  // displays «A»
```

See also

num asc(string sText, num nPosition)

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

Syntax

num asc(<string sText>, <num nPosition>)

Function

Returns the Unicode code point of the **nPosition** index character.
Returns -1 if **nPosition** is negative or greater than the specified text length.

Parameter

string sText	Character string type expression
num nPosition	Numerical expression

Example

```
putln(asc("A",0))            // displays 65
```

See also

string chr(num nCodePoint)

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

Syntax

string left(<string sText>, <num nSize>)

Function

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.

Parameter

string sText	Character string type expression
num nSize	Numerical expression

Example

```
putln(left("hello world",5))    // displays «hello»
```

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

Syntax

string right(<string sText>, <num nSize>)

Function

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.

Parameter

string sText	Character string type expression
num nSize	Numerical expression

Example

```
putln(right("hello world",5))   // displays «world»
```

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

Syntax

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.

Parameter

string sText	Character string type expression
num nSize	Numerical expression
num nPosition	Index in the string (from **0** to **127**)

Example

```
putln(mid(«hello wild world»,4,6))        // displays «wild»
```

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

Syntax

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

Function

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.

Parameter

string sText	Character string type expression
string sInsertion	Character string type expression
num nPosition	Index in the string (from 0 to 127)

Example

```
putln(insert("hello world","wild ",6))    // displays «hello wild world»
```

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

Syntax

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

Function

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.

Parameter

string sText	Character string type expression
num nSize	Numerical expression
num nPosition	Index in the string (from **0** to **127**)

Example

```
string sSource
sSource = "hello wild world"
putln(delete(sSource,5,6))                // displays «hello world»
putln(sSource)                            // displays «hello wild world»
```

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

Syntax

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

Function

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.

Parameter

string sText	Character string type expression
string sReplacement	Character string type expression
num nSize	Numerical expression
num nPosition	Index in the string (from **0** to **127**)

Example

```
putln(replace("hello ? world","wild",1,6))    // displays «hello wild world»
```

num **find**(string sText1, string sText2)

Syntax

num find(<string sText1>, <string sText2>)

Function

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**.

Parameter

string sText1	Character string type expression
string sText2	Character string type expression

Example

```
putln(find("hello wild world","wild"))        // displays 6
```

num **len**(string sText)

Syntax

num len(<string sText>)

Function

Returns the number of characters in **sText**.

Parameter

string sText	Character string type expression

Example

```
putln(len("hello wild world"))          // displays 16
```

See also

num getDisplayLen(string sText)

3.6. DIO TYPE

3.6.1. DEFINITION

The **dio** type is used to link a **VAL3** variable to a system digital input/output.

The inputs/outputs declared in the system can be used in a **VAL3** application from the io library, without having to be declared in the application as a global or local variable. The **dio** type is therefore used as an alias for a system input/output or as a parameter when calling a program.

All instructions using a **dio** type variable not linked to an input/output declared in the system generate a runtime error.

By default, a **dio** type variable is not linked to a system input/output and therefore generates a runtime error if used as such in a program before being linked.

3.6.2. OPERATORS

In ascending order of priority:

bool <dio diOutput> = <dio diInput>	Assigns the **diInput** status to **diOutput**, and returns the status. A runtime error is generated if **diOutput** is not linked to a system output.
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 **VAL3** expressions used as instruction parameter.

3.6.3. INSTRUCTIONS

void **dioLink**(dio& diVariable, dio diSource)

Syntax

void dioLink(<dio& diVariable>, <dio diSource>)

Function

Links **diVariable** to the input/output to which **diSource** is linked.

A runtime error is generated if **diVariable** is an input/output of the io library.

Parameter

dio& diVariable	Digital input/output type variable
dio diSource	Expression of **dio** type

Example

```
dio dGripper1
dio dGripper2
dioLink(diGripper1,          //  links diGripper1 to valve1 system input/output
io:valve1)
dioLink(diGripper2,          // links diGripper2 to the diGripper1 input/output, and therefore to
diGripper1)                  valve1
dioLink(diGripper1,          // diGripper2 is now linked to valve2, and diGripper1 is still
io:valve2)                   linked to valve1
```

num **dioGet**(dio diArray)

Syntax

num dioGet(<dio diArray>)

Function

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.

Parameter

dio diArray	Expression of **dio** type

Example

```
dio diCode[4]
diCode[0] = false
diCode[1] = true
diCode[2] = false
diCode[3] = true
putln(dioGet(diCode))        // displays 10 = 0 + 2 * 1 + 4 * 0 + 8 * 1
```

See also

num dioSet(dio diArray, num nValue)

num **dioSet**(dio diArray, num nValue)

Syntax

num dioSet(<dio diArray>, <num nValue>)

Function

Assigns the whole part of **nValue** in binary code to the outputs in the **diArray**, and returns the value actually assigned, 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 **dTable** is not linked to a system output.

Parameter

dio diArray	Expression of **dio** type
num nValue	Expression of **num** type

Example

```
dio dCode[4]
putln(dioSet(diCode, 10)        // displays 10 = 0 + 2 * 1 + 4 * 0 + 8 * 1
putln(dioSet(diCode, 26)        // displays 10, code is not big enough to encode 26 completely
```

See also

num dioGet(dio diArray)

3.7. AIO TYPE

3.7.1. DEFINITION

The **aio** type is used to link a **VAL3** variable to a system numerical input/output (integer or floating point value).
The inputs/outputs declared in the system can be used in a **VAL3** application from the io application, without having to be declared in the library as a global or local variable. The **aio** type is therefore used as an alias for a system analog input/output or as a parameter when calling a program.
All instructions using a **aio** type variable not linked to an input/output declared in the system generate a runtime error.
By default, a **aio** type variable is not linked to a system input/output and therefore generates a runtime error if used as such in a program before being linked

3.7.2. INSTRUCTIONS

void **aioLink**(aio& aiVariable, aio aiSource)

Syntax

void aioLink(<aio& aiVariable>, <aio aiSource>)

Function

Links **aiVariable** to the input/output to which **aiSource** is linked.
A runtime error is generated if **aiVariable** is an input/output of the io library.

Parameter

aio& aiVariable	**aio** type variable
aio aiSource	Expression of **aio** type

Example

```
aio aiSensor1
aio aiSensor2
aioLink(aiSensor1, io:system1)    //  links aiSensor1 to system1 system input/output
aioLink(aiSensor2, aiSensor1)     // links aiSensor2 to the aiSensor1 input/output, and therefore
                                  to system1
aioLink(aiSensor1, io:system2)    // aiSensor2 is now linked to system2, and aiSensor1 is still
                                  linked to system1
```

num **aioGet**(aio aiInput)

Syntax

num aioGet(<aio aiInput>)

Function

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

Parameter

aio& aiInput	Expression of **aio** type

Example

```
aio aiSensor
putln(aioGet(aiSensor))           // displays the current sensor value
```

See also

num aioSet(aio aiOutput, num nValue)

num **aioSet**(aio aiOutput, num nValue)

Syntax

num aioSet(<aio aiOutput>, <num nValue>)

Function

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.

Parameter

aio& aiOutput	Expression of **aio** type
num nValue	Expression of **num** type

Example

```
aio aiCommand
putln(aioSet(aiCommand, -12.3))      // displays -12.3
```

See also

num aioGet(aio aiInput)

3.8. SIO TYPE

3.8.1. DEFINITION

The **sio** type is used to link a **VAL3** 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 activated at the time of the initial reading or writing access by a **VAL3** program. The Ethernet socket connections are deactivated automatically when the **VAL3** application is closed.

The inputs/outputs declared in the system are usable in a **VAL3** application from the io library, without having to be declared in the application as a global or local variable. The **sio** type is therefore used as an alias for a system sio input/output or as a parameter when calling a program.

All instructions using a **sio** type variable not linked to an input/output declared in the system generate a runtime error.

By default, a **sio** type variable is not linked to a system input/output and therefore generates a runtime error if used as such in a program before being linked.

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 siInput>	Reads a byte on **siInput** and assigns **nData** with the byte value.
string <string sText> = <sio siInput>	Reads on **siInput** 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 **VAL3** expressions used as instruction parameter.

3.8.2. INSTRUCTIONS

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

Syntax

void sioLink(<sio& siVariable>, <sio siSource>)

Function

Links **siVariable** to the serial system input/output to which **siSource** is linked.

A runtime error is generated if **siVariable** is an input/output of the io library.

Parameter

sio& siVariable	**sio** type variable
sio siSource	Expression of **sio** type

Example

```
sio siSensor1
sio siSensor2
sioLink(siSensor1, io:portSerial1)    //  links siSensor1 to portSerial1 system input/output
sioLink(siSensor2, siSensor1)         // links siSensor2 to the siSensor1 input/output, and
                                      therefore to portSerial1
sioLink(siSensor2, io:portSerial1)    // links siSensor2 to portSerial1, siSensor1 is still linked
                                      to portSerial1
```

num **clearBuffer**(sio siInput)

Syntax

num clearBuffer(<sio siInput>)

Function

Empties the **siInput** reading buffer and returns the number of characters thus 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 **siInput** is not connected to a system serial link or Ethernet socket.

num **sioGet**(sio siInput, num& nData)

Syntax

num sioGet(<sio siInput>,<num& nData>)

Function

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 of all to make a connection if there is no active connection. When the timeout for input communication has been reached, **sioGet** returns **-1**. If the connection is active, but there are no data in the input reader buffer, **sioGet** waits until data are received or until the end of the waiting 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 **VAL3** variable.

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

Syntax

num sioSet(<sio siOutput>,<num& nData>)

Function

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 of all to make a connection if there is no active 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.

S6.4 num **sioCtrl**(sio siChannel, string nParameter, value)

Syntax

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"

Example

This program sets the parameters for a serial line.

```
sioCtrl(io:portSerial1, "baudRate", 115200)
sioCtrl(io:portSerial1, "bits", 8)
sioCtrl(io:portSerial1, "parity", "none")
sioCtrl(io:portSerial1, "stopBits", 1)
sioCtrl(io:portSerial1, "timeout", 0)
sioCtrl(io:portSerial1, "endOfString", 13)
```

CHAPTER 4

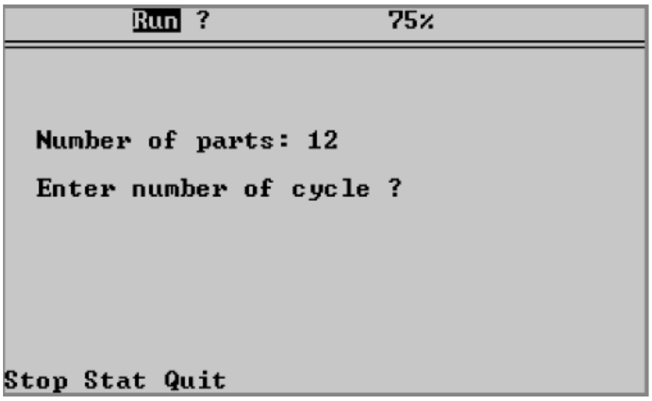
USER INTERFACE

4.1. USER PAGE

In the **VAL3** 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 on the **MCP** keyboard

User page



```
Run ?            75%

 Number of parts: 12
 Enter number of cycle ?

Stop Stat Quit
```

The user page has fourteen (**14**) **40**-column lines. The last line can be used to create menus with the associated key. An additional line is available for a title display.

4.2. INSTRUCTIONS

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

Syntax

void userPage ()
void userPage (<bool bFixed>)

Function

Displays the user page on the **MCP** screen.
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** user interface pages become accessible again.

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

Syntax

void gotoxy(<num nX>, <num nY>)

Function

Positions the cursor at the **(nX, nY)** coordinates on the 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**.

Parameter

num nX	Cursor column (**0** to **39**)
num nY	Cursor row (**0** to **13**)

See also

void cls()

void **cls**()

Syntax

void cls()

Function

Clears the user page and sets the cursor to **(0,0)**.

See also

void gotoxy(num nX, num nY)

num **getDisplayLen**(string sText)

Syntax

void getDisplayLen(string sText)

Function

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.

Parameter

string sText	Character string type expression

See also

num len(string sText)

void **put**() void **putln**()

Syntax

void put(<string sText>)
void put(<num nValue>)
void putln(<string sText>)
void putln(<num nValue>)

Function

Displays the specified **sText** or **nValue** (to **3** decimal places) at the cursor position on the 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.

Parameter

string sText	Character string type expression
num nValue	Numerical expression

See also

void popUpMsg(string sText)
void logMsg(string sText)
void title(string sText)

void **title**(string sText)

Syntax

void title(<string sText>)

Function

Changes the title of the user page.

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

Parameter

string sText	Character string type expression

num **get**()

Syntax

num get(<string& sString>)
num get(<num& nValue>)
num get()

Function

Acquires a string, a number or a control panel key.

The parameter **sString** 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 **sString** 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**

3	Caps	Space			
283	-	32			Move
				Ret.	-
2	Shift	Esc	Help		Run
282	-	255	-	270	-
	Menu	Tab	Up	Bksp	Stop
	-	259	261	263	-
1	User	Left	Down	Right	
281	-	264	266	268	

With **Shift**

3	Caps	Space			
283	-	32			Move
				Ret.	-
2	Shift	Esc	Help		Run
282	-	255	-	270	-
	Menu	UnTab	PgUp	Bksp	Stop
	-	260	262	263	-
1	User	Home	PgDn	End	
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

Parameter

string& sString	**string** type variable
num& nValue	**num** type variable

Example

```
num nValue
num nKey
// Waits for Return to be pressed to confirm the entry
do
 nKey = get (nValue)
until (nKey == 270)
```

See also

num getKey()

num **getKey**()

Syntax

num getKey()

Function

Acquires a key stroke from the control panel keyboard. Returns the code of the last key pressed since the last **getKey()** call, or **-1** if no key has been pressed since then. A key stroke can only be detected when the user page is displayed.

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

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

Example

```
// Displays the system clock until any key is pressed
getKey()                          // Resets the code of the last key pressed
while (getKey()== -1)
  gotoxy(0,0)
  put(toString(«», clock()* 10))
endWhile
```

See also

num get(), void userPage(), void userPage(bool bFixed)

bool isKeyPressed(num nCode)

bool **isKeyPressed**(num nCode)

Syntax

bool isKeyPressed(<num nCode>)

Function

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

See also

num get(), void userPage(), void userPage(bool bFixed)

void **popUpMsg**(string sText)

Syntax

void popUpMsg(<string sText>)

Function

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(bool bFixed)
void put() void putln()

void **logMsg**(string sText)

Syntax

void logMsg(<string sText>)

Function

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.

See also

void popUpMsg(string sText)

string **getProfile**()

Syntax

string getProfile()

Function

Returns the name of the current user profile.

See also

num setProfile(string sUserLogin, string sUserPassword)

num **setProfile**(string sUserLogin, string sUserPassword)

Syntax

num setProfile(<string sUserLogin>, <string sUserPassword>)

Function

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()

S6.4 string **getLanguage**()

Syntax

string getLanguage()

Function

This instruction returns the current language of the robot controller.

Example

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

See also

bool setLanguage(string sLanguage)

S6.4

bool **setLanguage**(string sLanguage)

Syntax

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()

S6.4

string **getDate**(string sFormat)

Syntax

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"
putln(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: a 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 **VAL3** 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.

5.3. VISIBILITY

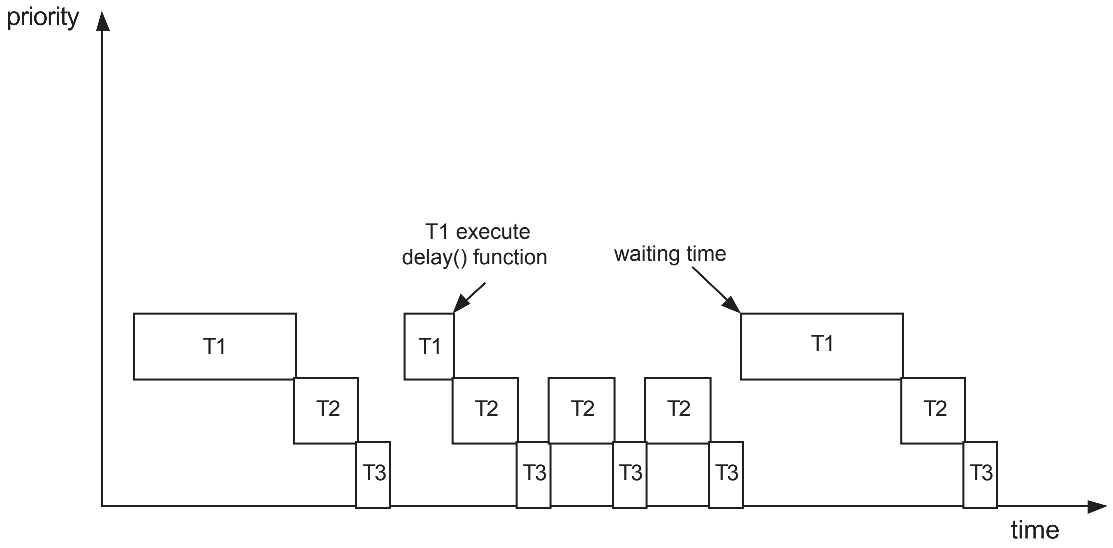
A task is visible only from within the program or library that created it. The instructions taskSuspend(), taskResume(), taskKill() and taskStatus() act on a task created by another library as if the task was not created. Two different libraries may therefore create tasks with the same name.

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



VAL3 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 **VAL3** 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 **VAL3** task.
4. When all **VAL3** 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 **VAL3** 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 **VAL3** lines. When the synchronous tasks have finished, the asynchronous task resumes.

The sequencing of the **VAL3** 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 **VAL3** 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 **VAL3** 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 **VAL3** 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 VAL3 scheduler. They can be refreshed at any time during the sequencing of a VAL3 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 **VAL3** programming is shown below.

Example

```
// synchronization of control for global variables
num n
bool bSynch
n=0                                            // Initialization of the global datas
bSynch=false
program Task1()
begin
  while(true)
    call synchro(n, bSynch, 2)                 // Synchronization with task 2
    <instructionsTask1>
  endWhile
end
program Task2()
begin
  while(true)
  call synchro(n, bSynch, 2)                   // Synchronization with task 1
  <instructionsTask2>
  endWhile
end
// Synchronization program for N tasks
program synchro(num& n, bool& bSynch, num N)
begin
  n = n + 1
  wait((n==N) or (bSynch==true))               //  Task synchronization waiting time
  bSynch = true
  n = n - 1
  wait((n==0) or (bSynch == false))            // Task release waiting time
  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 **VAL3** is shown below.

Example

```
bool bScreen
bScreen= false                                    // Initialization: screen resource is free

program Task1()
begin
  while(true)
    setMutex(bScreen)                             // Screen resource requested
    call fillScreen(1)
    bScreen = false                               // Screen resource released
    delay(0)                                      //  Proceeds to the next task
  endWhile
end

program Task2()
begin
  while(true)
    setMutex(bScreen)                             // Screen resource requested
    call fillScreen(2)
    bScreen = false                               // Screen resource released
    delay(0)                                      //  Proceeds to the next task
  endWhile
end

// program to fill the screen with the digit i
program fillScreen(num i)
num x
num y
begin
  i = i % 10
  for x = 0 to 39
    for y = 0 to 13
      gotoxy(x, y)
      put(i);
    endFor
  endFor
end
```

5.10. INSTRUCTIONS

void **taskSuspend**(string sName)

Syntax

void taskSuspend(<string sName>)

Function

Suspends 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 **VAL3** task, or corresponds to a **VAL3** task created by another library.

Parameter

string sName	Character string type expression

See also

void taskResume(string sName, num nSkip)
void taskKill(string sName)

void **taskResume**(string sName, num nSkip)

Syntax

void taskResume (<string sName>, <num nSkip>)

Function

Resumes 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 **VAL3** task, corresponds to a **VAL3** task created by another library, or if there is no instruction line at the specified **nSkip**.

Parameter

string sName	Character string type expression
num nSkip	Numerical expression

See also

void taskSuspend(string sName)
void taskKill(string sName)

void **taskKill**(string sName)

Syntax

void taskKill (<string sName>)

Function

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.

Parameter

string sName	Character string type expression

See also

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

void **setMutex**(bool& bMutex)

Syntax

void setMutex(<bool& bMutex>)

Function

Wait for the **bMutex** variable to be false, then set it to true.
This function is required to use a Boolean variable as a mutual exclusion mechanism for protecting shared resources (see chapter 5.9).

string **help**(num nErrorCode)

Syntax

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)

Syntax

num taskStatus (<string sName>)

Function

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	No runtime error
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 **VAL3** 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 **VAL3** 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.

Parameter

string sName	Character string type expression

See also

void taskResume(string sName, num nSkip)
void taskKill(string sName)

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

Syntax

void taskCreate <string sName>, <num nPriority>, program([p1] [,p2])

Function

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.
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.

Parameter

string sName	Character string type expression
num nPriority	Numerical expression
program	Name of an application program
p1	Type of expression specified by the program

Example

```
program display(string& sText)
begin
  putln(sText)
  sText = "stop"
end
string sMessage
program start()

begin
  sMessage = "start"
  taskCreate "t1", 10, display(sMessage)  // displays « start »
  wait(taskStatus("t1") == -1)            // waits for the end of t1
  putln(sMessage)                         // displays "stop"
end
```

See also

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

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

Syntax

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

Function

Creates and starts a synchronous task.
The execution of the task starts with the call of the specified program with the specified parameters.
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).

Parameter

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 **VAL3** program to call when the task is started, with its parameters between parenthesis. It is not possible to use a local variable as parameter if it is passed by reference, to make sure that the variable is not deleted before the task.

Example

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

void **wait**(bool bCondition)

Syntax

void wait(<bool bCondition>)

Function

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).

Parameter

bool bCondition	Boolean expression

See also

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

void **delay**(num nSeconds)

Syntax

void delay(<num nSeconds>)

Function

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 **VAL3** task immediately.

Parameter

num nSeconds	Numerical expression

See also

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

num **clock**()

Syntax

num clock()

Function

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

```
num nStart
nStart=clock()
<instructions>
put("time required for the operation= " )
putln(clock()-nStart)
```

See also

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

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

Syntax

bool watch (<bool bCondition>, <num nSeconds>)

Function

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 evaluated immediately, otherwise the system sequences the other **VAL3** tasks (even if **nSeconds** is up to and including **0**).

Parameter

bool bCondition	Boolean expression
num nSeconds	Numerical expression

Example

```
while (watch (diSensor==true, 20)) == false
  popUpMsg("Waiting for part")
  wait(diSensor==true)
endWhile
```

See also

void delay(num nSeconds)
void wait(bool bCondition)
num clock()

CHAPTER 6

LIBRARIES

6.1. DEFINITION

A **VAL3** library is a **VAL3** application that has variables or programs that can be reused by another application or by other **VAL3** libraries.
Being a **VAL3** application, a **VAL3** library comprises the following components:

- a set of **programs**: the **VAL3** 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:

```
part:libLoad("part_7")     // Loads the "part_7" library identified as 'part'
title(part:Name)           // Displays as title the content of the name variable of the "part_7" library
call part:init()           // Calls up the init() program for the current part
```

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

S6.1 6.5. ENCRYPTION

VAL3 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 **VAL3** interpreter must have access to the secret ZIP password to load an encrypted library; for this, a PC tool is provided with Stäubli Robotics Studio(*) to encode the secret ZIP password into a public **VAL3** password. The public **VAL3** password makes it possible to use the encrypted library in a **VAL3** program. But the library content remains secret because the ZIP password cannot be computed from the **VAL3** password.

(*) This tool, a passwordZip.exe executable delivered with the **VAL3** emulator, requires a specific **SRS** licence to be used.

Project encryption

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

- Declaring its start() program as public.

- Creating another project that simply loads the encrypted project and call its start() program.

6.6. LOADING AND UNLOADING

When a **VAL3** 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.

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 **VAL3** 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

```
part:libLoad("part_1")
part:libSave("USB0://part")
part:libSave("Disk://part_x/part_1")
```

Error codes

The **VAL3** 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 **libLoad()**.
11	Library loaded, but public interface does not match. A runtime error 80 will result if the **VAL3** program tries to access missing items. See **libExist** instruction.
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 **VAL3** task. All tasks created by **VAL3** 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.

S6.1

File reading/writing error

Code	Description
31	Cannot save the library: the path specified already contains a library. To replace a library on disk, first delete it with **libdelete()**.
32	S6.2 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 "Connexion 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 connexion not defined"

6.7. INSTRUCTIONS

num identifier:**libLoad**(string sPath)

S6.1 num identifier:**libLoad**(string sPath, string sPassword)

Syntax

num identifier:libLoad(string sPath)

S6.1 **num identifier:libLoad(string sPath, string sPassword)**

Function

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 **VAL3** password computed from the secret ZIP password of the encrypted library (see chapter 6.5, page 94) .

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()**

Syntax

num identifier:libSave()
num identifier:libSave(string sPath)

Function

Saves the variables and programs assigned to the library's identifier. If **libSave()** is called without an identifier, the application of the library calling 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.

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.

See also

num libDelete(string sPath)

num **libDelete**(string sPath)

Syntax

num libDelete(string sPath)

Function

Deletes the library located in the specified **sPath**.

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**()

Syntax

string identifier: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)

Syntax

bool libList(string sPath, string& sContents)

Function

Lists the contents of the **sPath** path specified 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()

S6.4 bool identifier:**libExist**(string sSymbolName)

Syntax

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 program sends a log message **sLogMessage** to serial line COM1 if any, else logs it.

```
if(io:libExist("COM1")==true)
  io:COM1=sLogMessage
else
  logMsg(sLogMessage)
endIf
```

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

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

CHAPTER 7

ROBOT CONTROL

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

7.1. INSTRUCTIONS

void **disablePower**()

Syntax

void disablePower()

Function

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**()

Syntax

void enablePower()

Function

In remote mode, 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.

Example

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

See also

void disablePower()
bool isPowered()

bool **isPowered**()

Syntax

bool isPowered()

Function

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**()

Syntax

bool isCalibrated()

Function

Returns the calibration status of the robot:

true: all the robot axis are calibrated

false: at least one robot axis is not calibrated

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

Syntax

num workingMode (num& nStatus)
num workingMode()

Function

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		Revolute **(Joint)**
	3		Cartesian **(Frame)**
	4		**(Tool)**
	5		To point **(Point)**
	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

Parameter

num& nStatus	Numerical type variable.

num **esStatus**()

Syntax

num esStatus()

Function

Returns the status of the E-Stop circuit:

Code	Status
0	**All the E-Stops are inactive.**
1	**Waiting for validation after an emergency stop.**
2	**E-Stop open.**

See also

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

S6.4 num **getMonitorSpeed**()

Syntax

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
endWhile
```

See also

num setMonitorSpeed(num nSpeed)

S6.4 num **setMonitorSpeed**(num nSpeed)

Syntax

num setMonitorSpeed(<num nSpeed>)

Function

This instruction modifies the current monitor speed of the robot. It is effective only if the robot is in remote working mode and if the operator does not have access to the monitor speed.

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()

S6.4

string **getVersion**(string nComponent)

Syntax

num getVersion(<string nComponent>)

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
"VAL3"	Controller **VAL3** version, such as "s6.4 - Sep 27 2007 - 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"
License name	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)

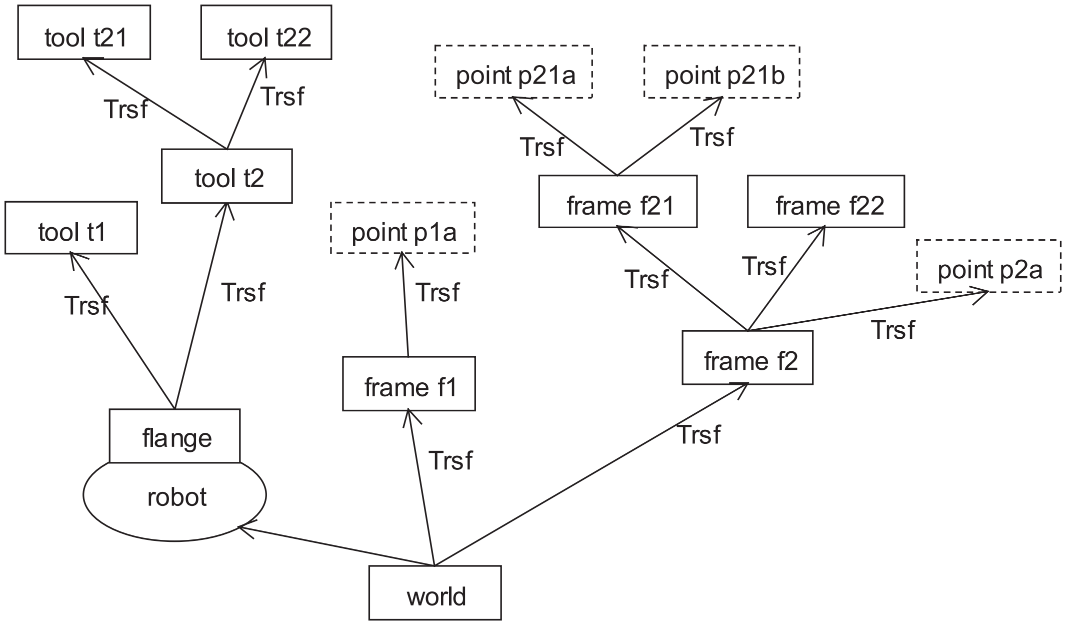
CHAPTER 8

ARM POSITIONS

8.1. INTRODUCTION

This chapter describes the **VAL3** data types used to program the arm positions used in a **VAL3** application.
Two position types are defined in **VAL3**: joint positions (**joint** type) that give the angular position of each revolute axis and the linear position for 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:

Organization chart: frame / point / tool / trsf



8.2. JOINT TYPE

8.2.1. DEFINITION

A joint location (**joint** type) defines the angular position of each revolute and the linear position of each linear axis robot axis.
The **joint** type is a structured type, with the following fields, in this order:

num j1	Revolute position of axis **1**
num j2	Revolute position of axis **2**
num j3	Revolute position of axis **3**
num j...	Revolute 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**.

8.2.2. OPERATORS

In ascending order of priority:

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

To avoid confusions between = and == operators, the = operator is not allowed within **VAL3** expressions used as instruction parameter.

8.2.3. INSTRUCTIONS

joint **abs**(joint jPosition)

Syntax

joint abs(joint jPosition)

Function

Returns the absolute value of a joint **Position**, field by field.

Parameter

jPosition	Joint expression

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**()

Syntax

joint herej()

Function

Return the current arm joint position.

Details

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)

Syntax

bool isInRange(joint jPosition)

Function

Test if a joint position is within the software joint limits of the arm.

Parameter

jPosition	Joint expression to be tested

Details

When the arm is out of the software joint limits (after a maintenance operation), it is not possible to move the arm with a **VAL3** 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 diInput) (CS8C only)

Syntax

void setLatch(dio diInput)

Function

Enable robot position latch on the next rising edge of the input signal.

Parameter

diInput	Dio expression defining the digital input to be used for latching

Details

The robot position latching is a hardware feature that is supported only by the fast inputs of the CS8C controller (io:fIn0, io:fIn1).

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 VAL3 instruction is executed.**

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

Example

```
// enable latch on first fast input (CS8C)
setLatch(io:fIn0)
```

See also

bool getLatch(joint& jPosition) (CS8C only)

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

Syntax

bool getLatch(joint& jPosition)

Function

Read the last latched robot position.

Parameter

jPosition	Joint expression defining the variable to update with the latched position

Details

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

```
// Wait for a latched position during 5 seconds.
bLatch = watch(getLatch(jPosition)==true, 5)
if bLatch==true
  putln("Successful position latch")
else
  putln("No latch signal was detected")
endif
```

See also

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

8.3. TRSF TYPE

8.3.1. 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 about the **x** axis
num ry	Rotation about the **y** axis
num rz	Rotation about the **z** axis

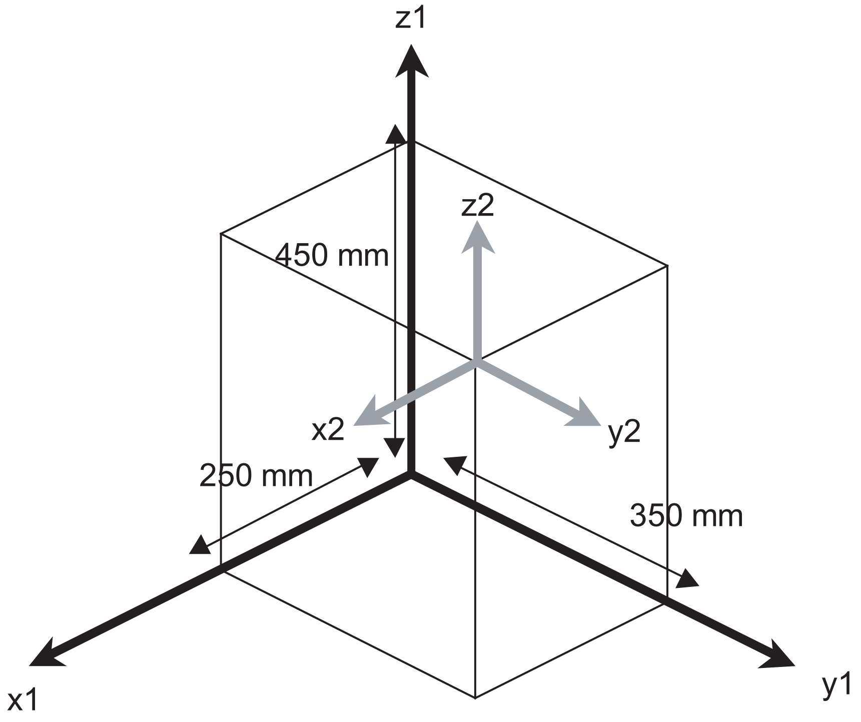
The **x, y** and **z** fields are expressed in the unit of length of the application (millimeter or inch, see the chapter entitled Unit of length). The **rx, ry** and **rz** fields are expressed in degrees.

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}**.

8.3.2. ORIENTATION

Orientation

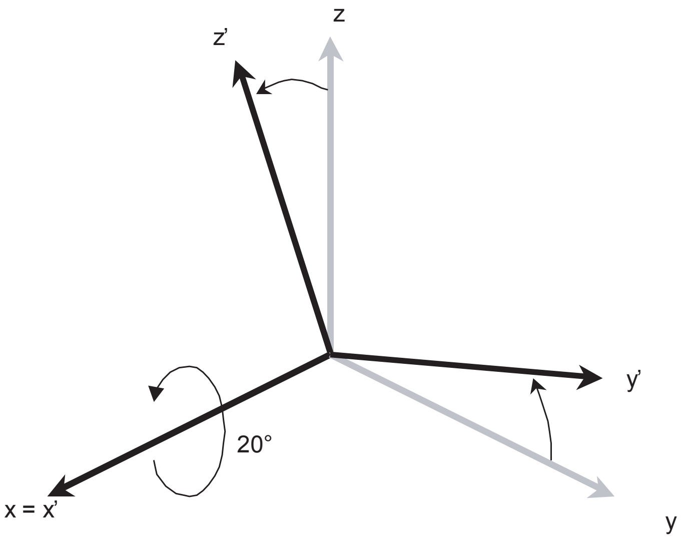


The position of frame **R2** (grey) relative to **R1** (black) is:
x = 250mm, y = 350 mm, z = 450mm, rx = 0°, ry = 0°, rz = 0°

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

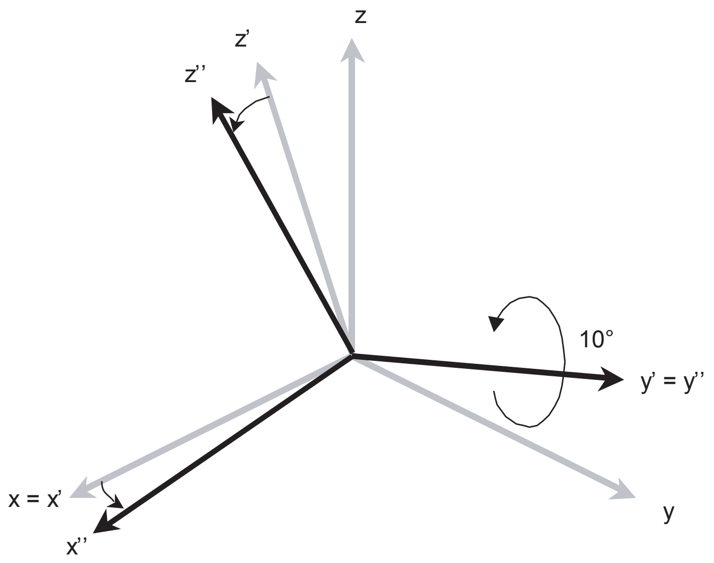
For example, orientation **rx = 20°, ry = 10°, rz = 30°** is obtained as follows. First, the frame **(x,y,z)** is rotated through **20°** about the **x** axis. This gives a new frame **(x',y',z')**. The **x** and **x'** axis coincide.

Frame rotation about the axis: X



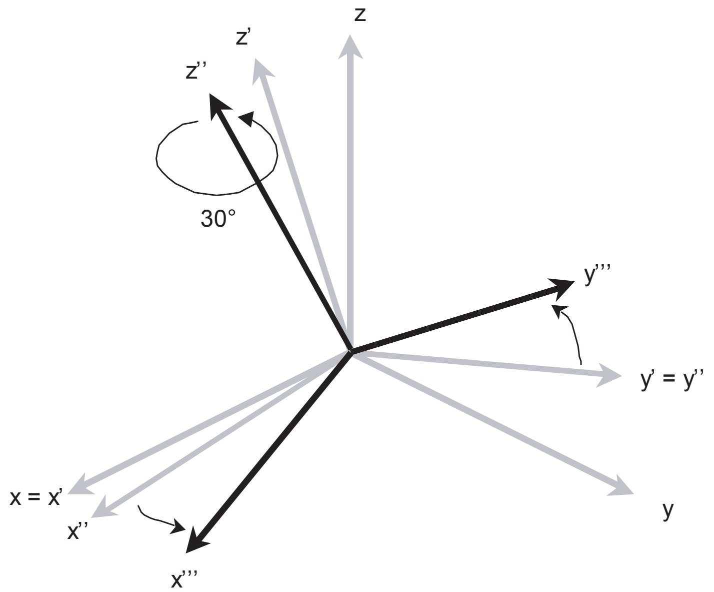
Then the frame is rotated through **20°** about the **y'** axis of the frame obtained at the previous step. This gives a new frame **(x'',y'',z'')**. The **y'** and **y''** axis coincide.

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 = 250mm, y = 350 mm, z = 450mm, rx = 20°, ry = 10°, rz = 30°

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.

8.3.3. OPERATORS

In ascending order of priority:

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

To avoid confusions between = and == operators, the = operator is not allowed within **VAL3** expressions used as instruction parameter.

8.3.4. INSTRUCTIONS

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

Syntax

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.**

Parameter

trsf trPosition1	Transformation type expression
trsf trPosition2	Transformation type expression

Example

```
// Displays the distance between two points, whatever their reference frames
putln(distance(position(point1, world), position(point2, world)))
```

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)

S6.4 trsf **interpolateL**(trsf trStart, trsf trEnd, num nPosition)

Syntax

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: trsf.x0 = trStart.x + (trEnd.x-trStart.x)*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 tFrame, frame fReference)

trsf position(tool tTool, tool tReference)

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

trsf align(trsf trPosition, trsf Reference)

S6.4 trsf **interpolateC**(trsf trStart, trsf trIntermediate, trsf trEnd, num nPosition)

Syntax

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 tFrame, frame fReference)

trsf position(tool tTool, tool tReference)

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

trsf align(trsf trPosition, trsf Reference)

S6.4 trsf **align**(trsf trPosition, trsf Reference)

Syntax

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 tFrame, 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)

8.4. FRAME TYPE

8.4.1. DEFINITION

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

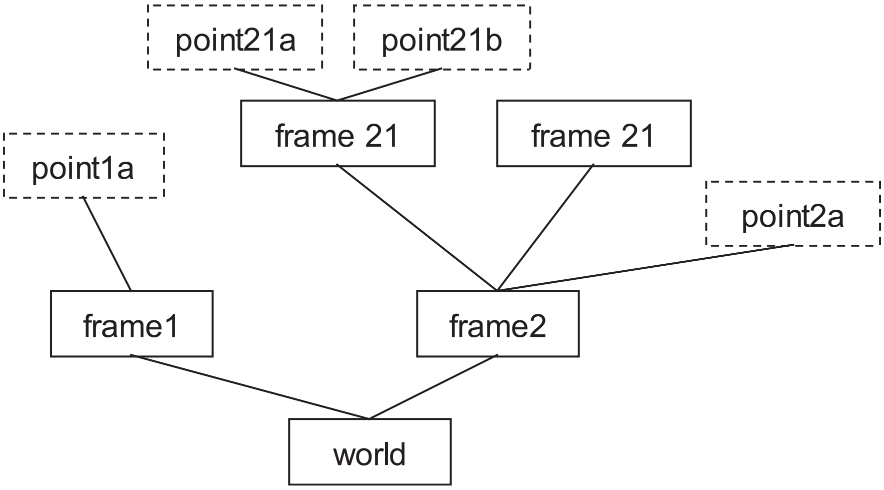
The frame type is a structured type with only one accessible field:

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, or via the **=** operator). The **frame** type **world** reference frame is always defined in a **VAL3** 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.

Links between reference frames



By default, a **frame** type variable uses **world** as its reference frame.

8.4.2. 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 cell taught point display 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 **VAL3** points to the frame. If the object is moved, simply reteach 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.

8.4.3. 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 **VAL3** expressions used as instruction parameter.

8.4.4. INSTRUCTIONS

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

Syntax

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

Function

Calculates 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.

S6.4 trsf **position**(frame tFrame, frame fReference)

Syntax

trsf position(<frame tFrame>, <frame fReference>)

Function

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

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

See also

trsf position(point pPosition, frame fReference)

trsf position(tool tTool, tool tReference)

8.5. TOOL TYPE

8.5.1. DEFINITION

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

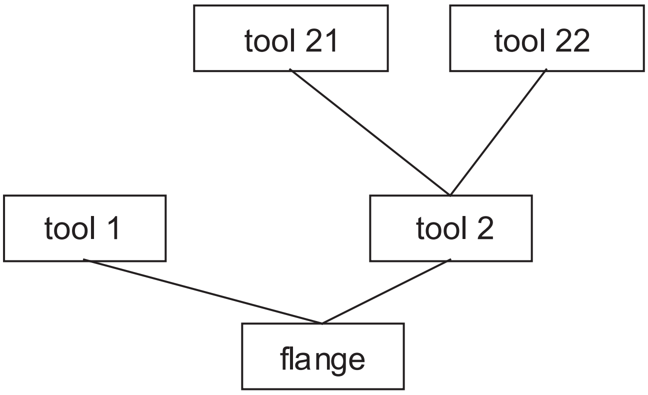
The **tool** type is a stuctured type with the following fields, in this order:

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 **basic tool** of a **tool** type variable is defined when it is initialized (via the user interface, or via the **=** operator). The **flange** basic tool, of **tool** type, is always defined in a **VAL3** 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.

Links between tools



By default, the output of a tool is the system **io:valve1** output, the opening and closing times are **0** and the basic tool is **flange**.

8.5.2. 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 **VAL3** tool corresponding to the physical tool at the end of the arm.
- **To control tool wear or a tool change**
  To update the arm position, simply update the geometrical coordinates of the tool.

8.5.3. 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 **VAL3** expressions used as instruction parameter.

8.5.4. INSTRUCTIONS

void **open**(tool tTool)

Syntax

void open (tool tTool)

Function

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.

Parameter

tool tTool	Tool type expression

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)

Syntax

void close (tool tTool)

Function

Activates the tool (closing) by setting its digital output to **false**.

Before activating the tool, **open()** 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.

Parameter

tool tTool	Tool type expression

Example

```
// the close instruction is equivalent to:
waitEndMove()
tTool.gripper = false
delay(tTool.ctime)
```

See also

Type tool
void open(tool tTool)
void waitEndMove()

S6.4 trsf **position**(tool tTool, tool tReference)

Syntax

trsf position(<tool tTool>, <tool tReference>)

Function

This instruction returns the coordinates of the tool **tTool** in the reference frame of 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 tFrame, frame fReference)

8.6. POINT TYPE

8.6.1. DEFINITION

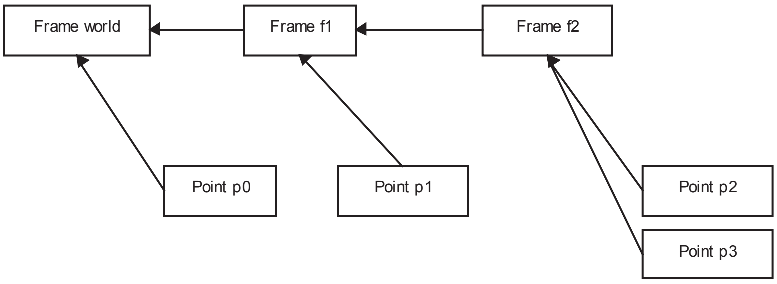
The **point** type is used to define the position and orientation of the robot tool in the cell.

The **point** type is a stuctured 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 **here()**, **appro()** and **compose()** instructions.

Point definition



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

CAUTION:
By default, a local point type variable has no reference frame. Before it can be used, it must be initialized from another point with the '=' operator, or via one of the here(), appro() and compose() instructions.

8.6.2. 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 **VAL3** expressions used as instruction parameter.

num **distance**(point pPosition1, point pPosition2)

Syntax

num distance(point pPosition1, point pPosition2)

Function

Returns the distance between **pPosition1** and **pPosition2**.

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

Parameter

point pPosition1	Point type expression
point pPosition2	Point type expression

Example

```
// Displays the distance between two points, whatever their reference frames
putln(distance(pPoint1, pPoint2))
```

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)

Syntax

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

Function

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.

Parameter

point pPosition	Point type expression
frame fReference	Reference frame type expression
trsf trTransformation	Transformation type expression

Example

```
point pResult
// 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)

Syntax

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.

Parameter

point pPosition	Point type expression
trsf trTransformation	Transformation type expression

Example

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

See also

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

point **here**(tool tTool, frame fReference)

Syntax

point here(tool tTool, frame fReference)

Function

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)

Syntax

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

Function

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 revolute **jPosition** position.

Parameter

tool tTool	Tool type expression
frame fReference	Reference frame type expression
joint jPosition	Revolute position type 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)

Syntax

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 revolute 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.

Parameter

tool tTool	Tool type expression
joint jInitial	Joint position type expression
point pPosition	Point type expression
joint& jResult	Joint position type variable

See also

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

trsf **position**(point pPosition, frame fReference)

Syntax

trsf position(point pPosition, frame fReference)

Function

Returns the coordinates of **pPosition** in **fReference** frame.

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

Example

```
// Displays the distance between two points, whatever their reference frames
putln(distance(position(pPoint1, world), position(pPoint2, world)))
```

See also

num distance(point pPosition1, point pPosition2)

trsf position(tool tTool, tool tReference)

trsf position(frame tFrame, frame fReference)

8.7. CONFIG TYPE

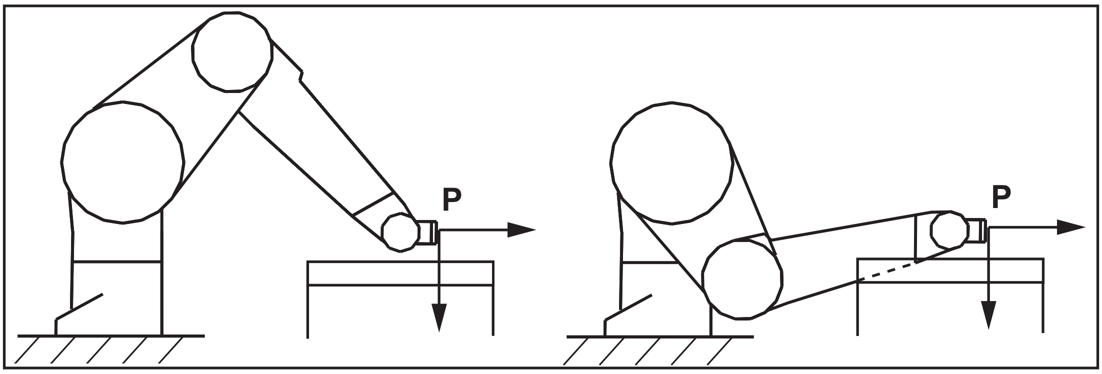
The configuration concept of a Cartesian point is an "advanced" concept; this chapter can be skipped the first time you read this document.

8.7.1. 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:

Two configurations that can be used to reach a given point: P



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.

8.7.2. 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** 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

8.7.3. 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.

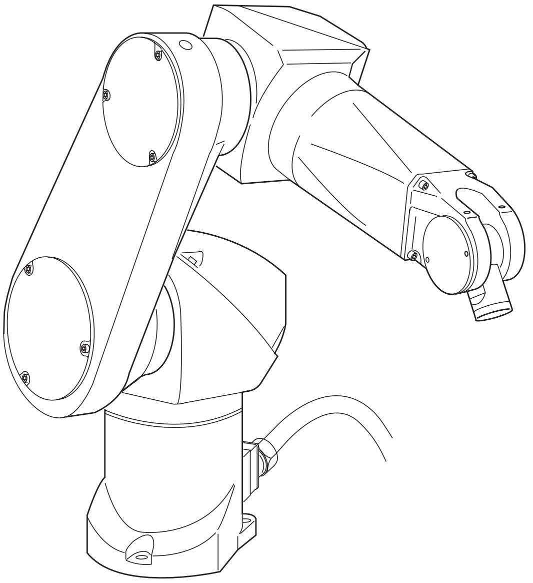
To avoid confusions between = and == operators, the = operator is not allowed within **VAL3** expressions used as instruction parameter.

8.7.4. CONFIGURATION (RX/TX ARM)

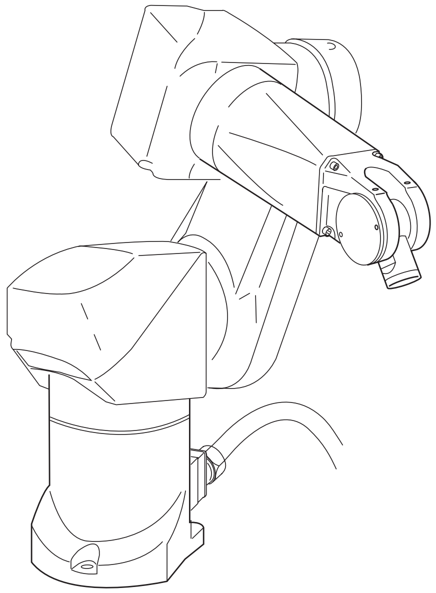
8.7.4.1. SHOULDER CONFIGURATION

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



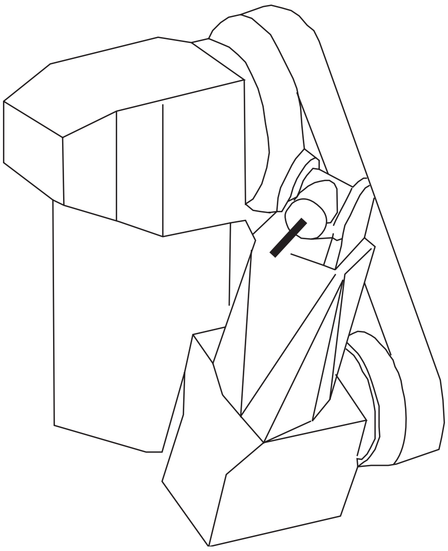
Configuration: lefty



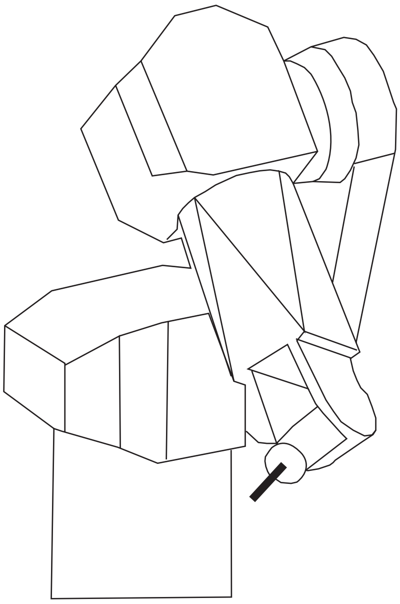
The righty configuration is defined by (d1 * sin(j2) + d2 * sin(j2+j3) + δ) < 0, and the lefty configuration is defined by (d1 * sin(j2) + d2 * sin(j2+j3) + δ) >= 0, where d1 is the length of the robot arm, d2 the length of the forearm, and δ the distance between axis 1 and axis 2, in the x direction.

8.7.4.2. ELBOW CONFIGURATION

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



Configuration: enegative



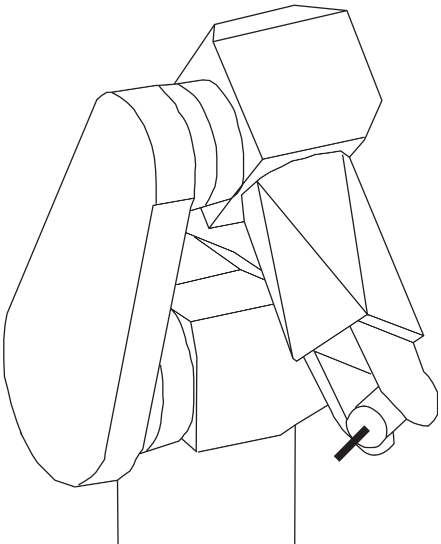
Configuration: epositive

The **epositive** configuration is defined by **j3 >= 0**.

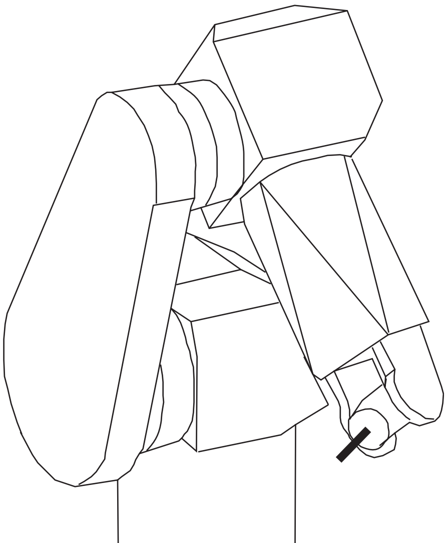
The **enegative** configuration is defined by **j3 < 0**.

8.7.4.3. 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

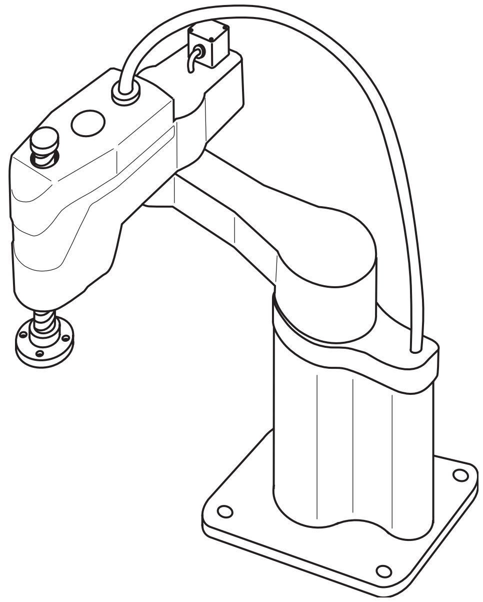
The **wpositive** configuration is defined by **j5 >= 0**.

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

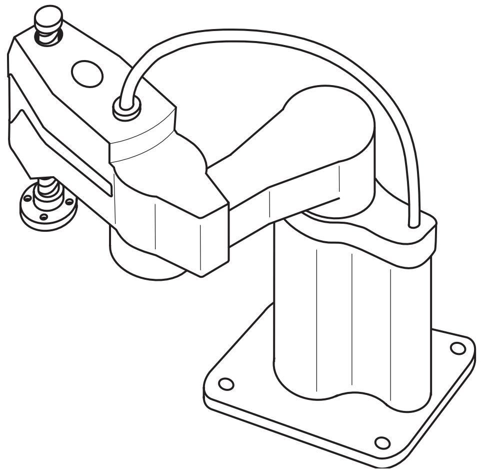
8.7.5. CONFIGURATION (RS ARM)

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) ≤ 0**.

8.7.6. INSTRUCTIONS

config **config**(joint jPosition)

Syntax

config config(joint jPosition)

Function

Returns the configuration of the robot for the joint **jPosition** position.

Parameter

joint jPosition	Joint position type expression

See also

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

CHAPTER 9

MOVEMENT CONTROL

9.1. TRAJECTORY CONTROL

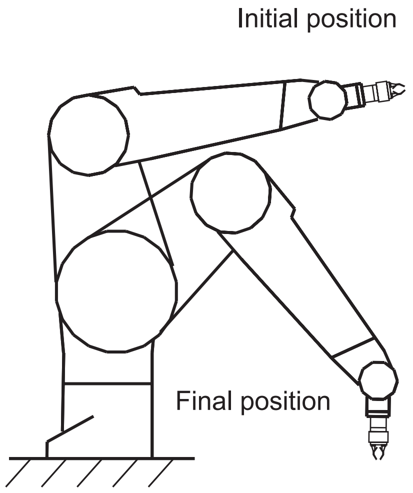
A succession of points is not sufficient to define the trajectory of a robot. It is also necessary to indicate the type of trajectory used between the points (curve or straight line), specify how the trajectories are linked together and define the movement speed parameters. This section therefore presents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

9.1.1. 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.

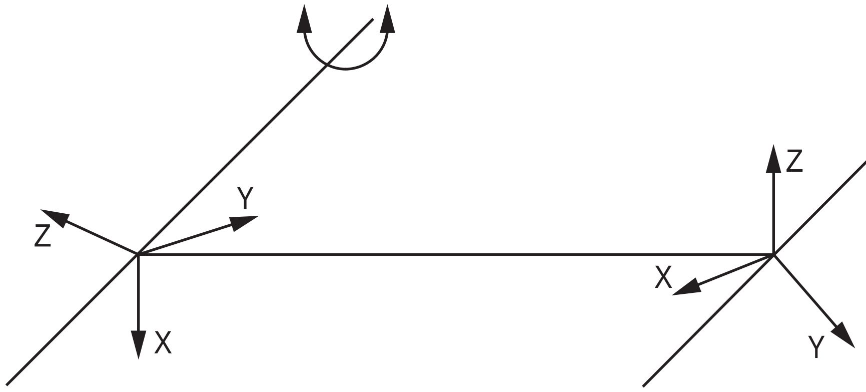
A point-to-point movement is a movement in which only the final destination (Cartesian or revolute point) is important. Between the start point and the end point, the tool center point follows a curve defined by the system to optimize the speed of the movement.

Initial and final positions



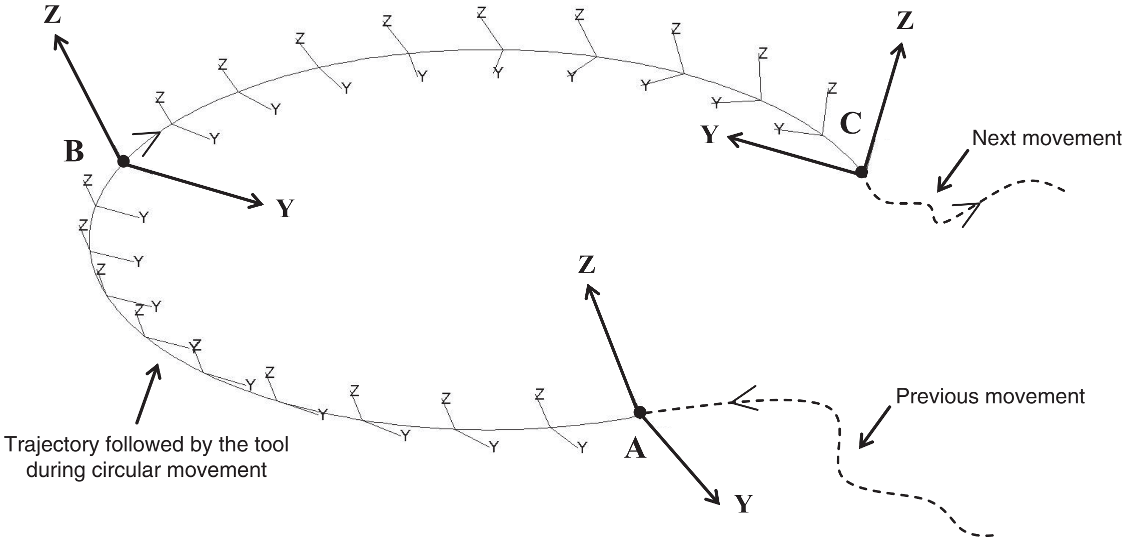
Conversely, in the case of a straight line movement, the tool center point moves along a straight line. The orientation is interpolated in a linear way between the initial and final orientation of the tool.

Straight line movement



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.

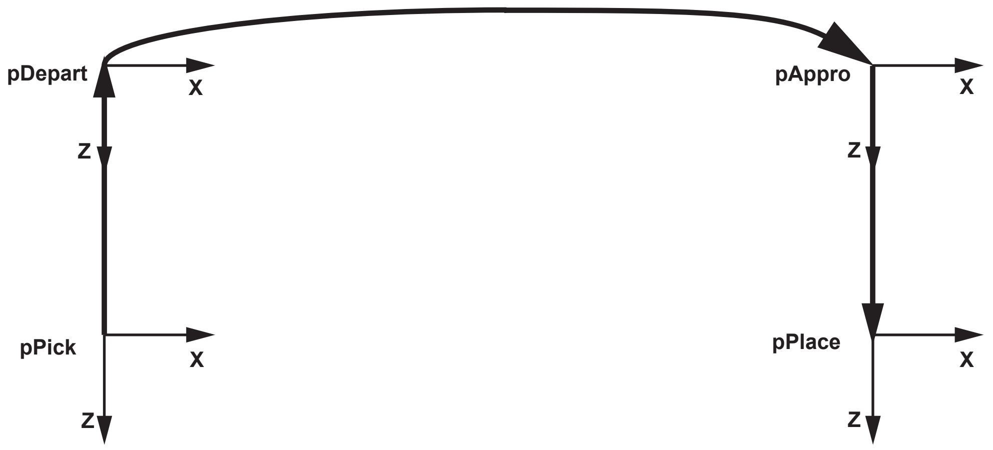
Circular movement



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**.

Cycle type: U



Let us assume that the robot is initially at the **pPick** point. The program required to execute the movement can be written as follows:

```
movel(pDepart, tTool, mDesc)
movej(pAppro, tTool, mDesc)
movel(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.*

9.1.2. MOVEMENT SEQUENCING

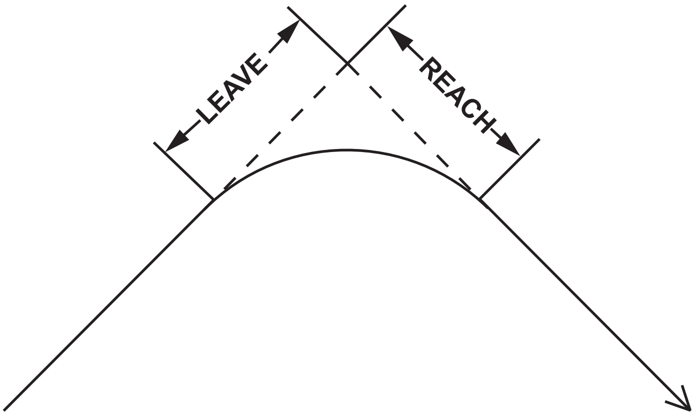
9.1.2.1. BLENDING

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**, 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**, 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'

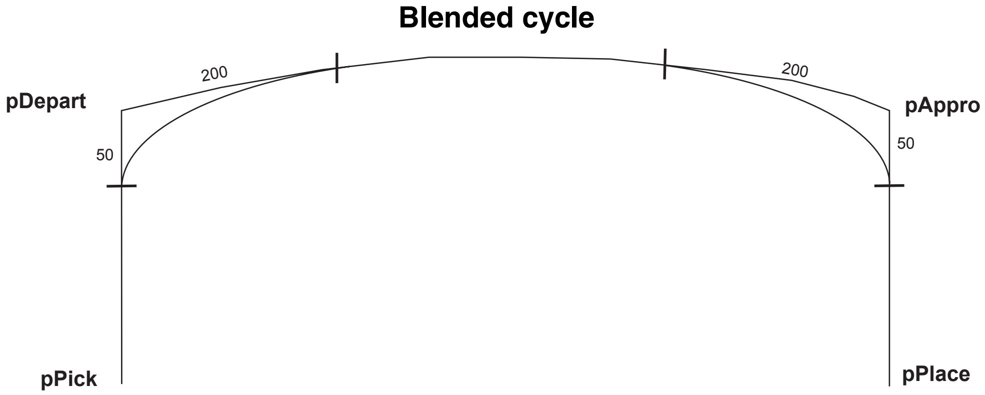


Example:

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
movel(pDepart, tTool, mDesc)
mDesc.leave = 200
mDesc.reach = 50
movej(pAppro, tTool, mDesc)
mDesc.blend = OFF
movel(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.

9.1.2.2. CANCEL BLENDING

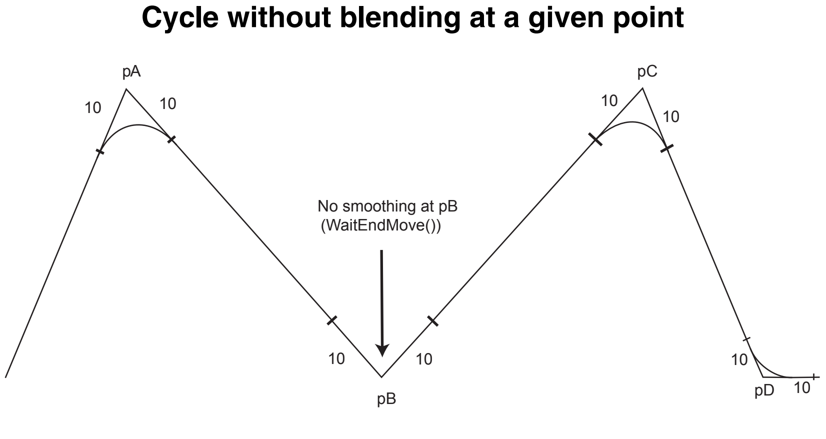
The **waitEndMove()** instruction is used to cancel the effect of blending. The robot then completes the last programmed movement as far as its arrival point, as if the movement descriptor **blend** field were set to **off**.

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:



9.1.3. 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 **VAL3**'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.

9.1.4. PARTICULARITIES OF CARTESIAN MOVEMENTS (STRAIGHT LINE, CIRCLE)

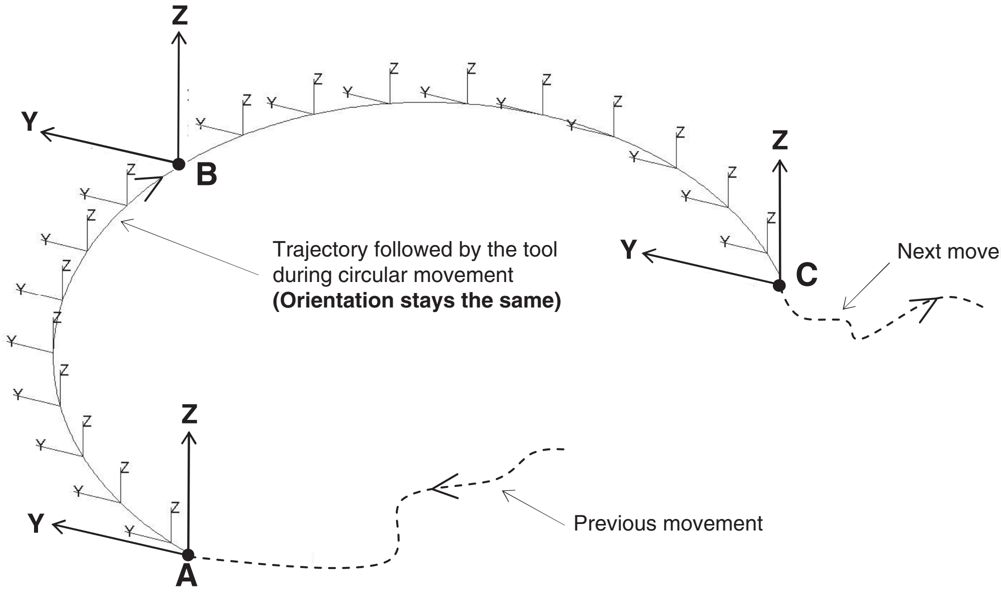
9.1.4.1. INTERPOLATION OF THE ORIENTATION

The **VAL3** 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.

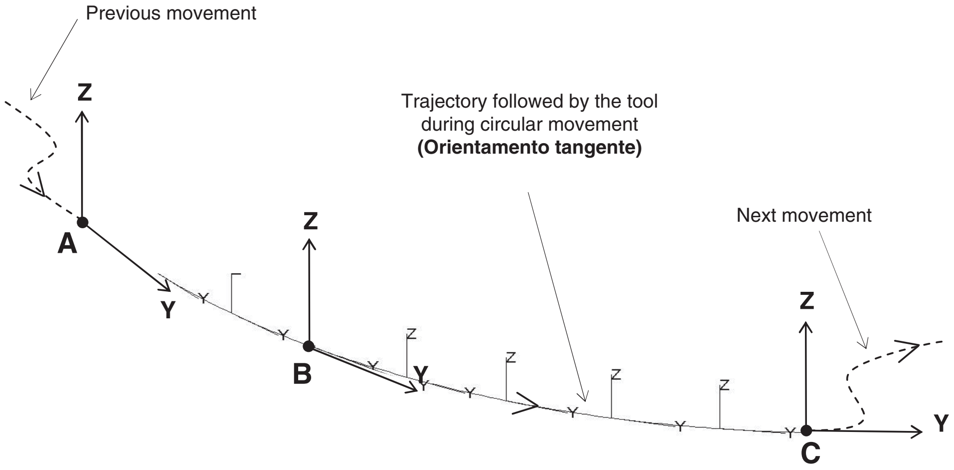
- For a constant orientation, the initial and final positions, and the intermediate position for a circle, must have the same orientation.

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 inital 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

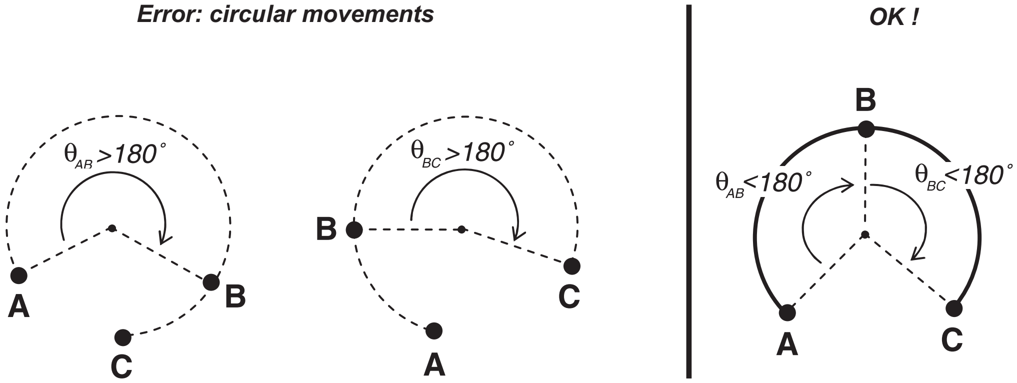


This results in a limitation for circular movements:

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.

Ambiguity as to the intermediate orientation

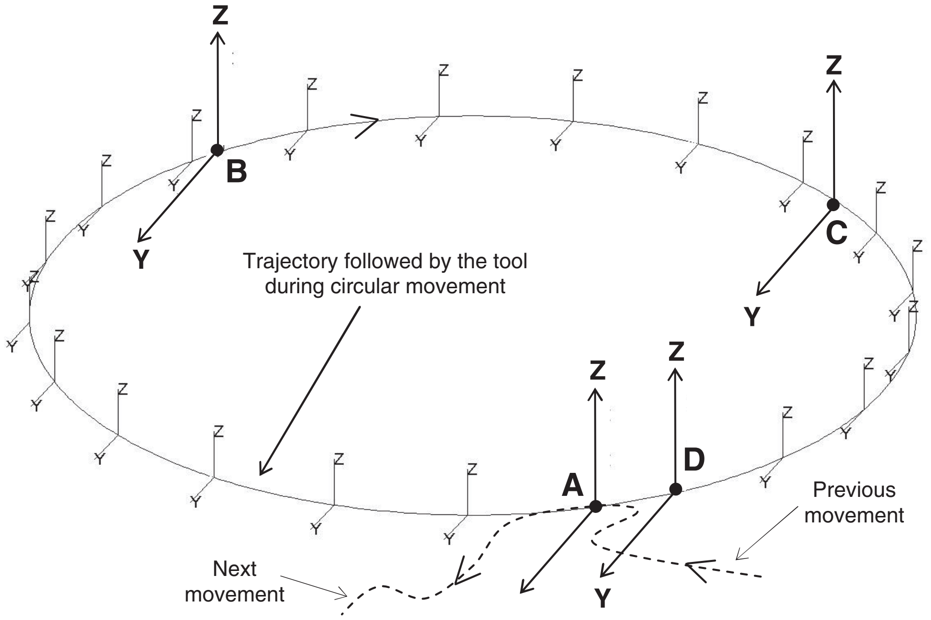


In particular, programming a full circle involves **2 movec** instructions:

movec (B, C, Tool, mDesc)

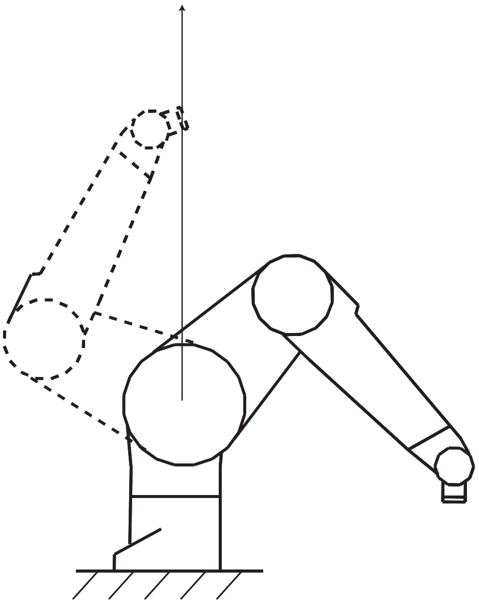
movec (D, A, Tool, mDesc)

Full circle



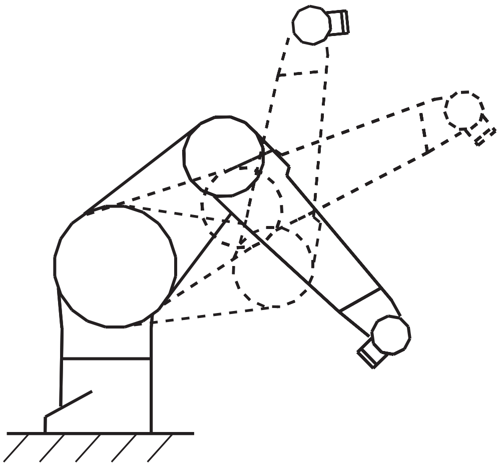
9.1.4.2. CONFIGURATION CHANGE (ARM RX/TX)

Configuration change: righty / lefty



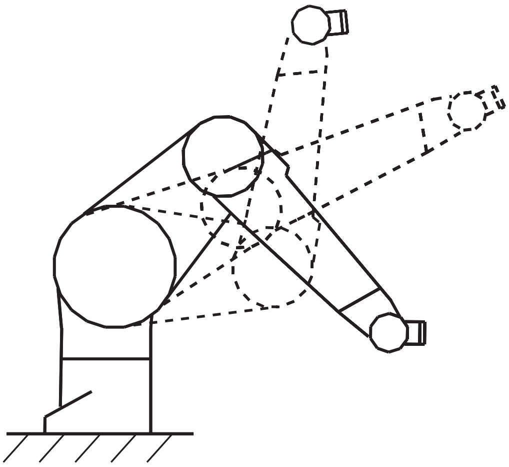
During a change of shoulder configuration, the centre of the robot's wrist has to pass vertically through axis **1** (but not exactly in the case of offset robots).

Positive/negative elbow configuration change



During a change of elbow configuration, the arm has to go through the straight arm position **(j3 = 0°)**.

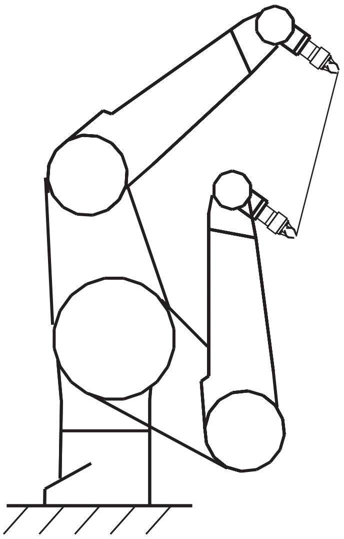
Positive/negative wrist configuration change



During a change of wrist configuration, the arm has to go through the straight wrist position **(j5 = 0°)**.

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

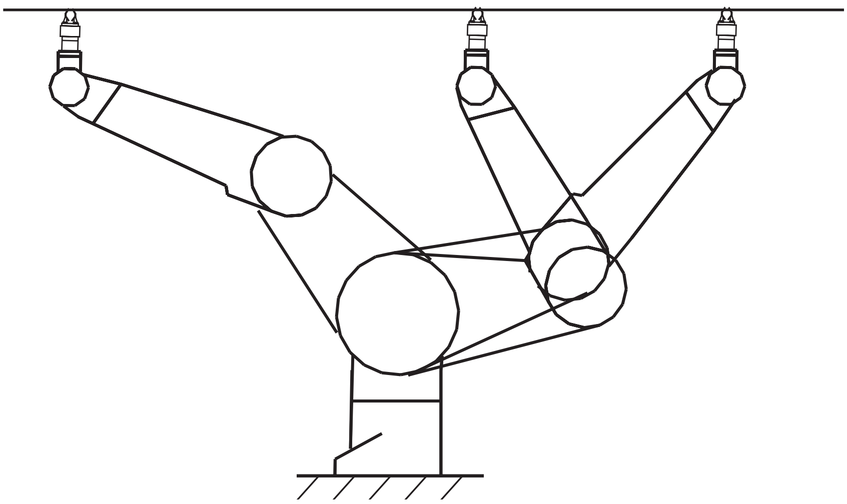


In other words, during a straight-line or circular movement, we can only impose a configuration if it is compatible with the initial position: it is therefore always possible to specify a free configuration, or one that is identical to the initial configuration.

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.

Shoulder configuration change possible



9.1.4.3. 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 axis 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.

9.2. MOVEMENT ANTICIPATION

9.2.1. 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.

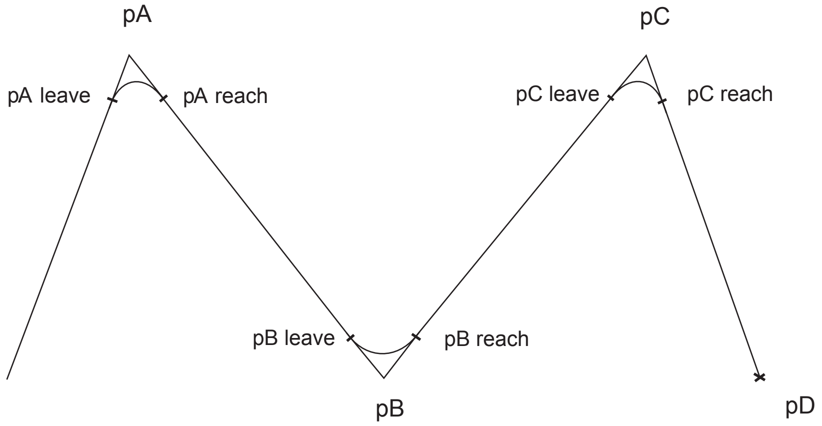
9.2.2. ANTICIPATION AND BLENDING

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).

Blended cycle



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.

9.2.3. SYNCHRONIZATION

The anticipation mechanism causes desychronization between the **VAL3** instruction lines and the corresponding robot movements: the **VAL3** 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.

Thus in the following program:

```
movej(A, tool, mDesc)
movej(B, tool, mDesc)
waitEndMove()
movej(C, tool, mDesc)
movej(D, tool, 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 comlete its movements before activating the tool.

9.3. SPEED MONITORING

9.3.1. 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.

9.3.2. 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.
4. If the speed is not sufficient, increase the acceleration and deceleration parameters

9.3.3. 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.

9.3.4. 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.

9.4. 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 only be adjusted via the robot's manual control pendant, and not in a **VAL3** application. However, the **speedScale()** instruction allows an application to know the current monitor speed and hence, if necessary, ask the user to reduce it when the cycle resumes, or set it to **100%** during production.
- 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.

9.5. MDESC TYPE

9.5.1. 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), or **joint** (blending).
num leave	In **joint** 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** 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 at **{100,100,100,9999,9999,off,50,50}**.

9.5.2. 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.

9.6. MOVEMENT INSTRUCTIONS

void **movej**(joint jPosition, tool tTool, mdesc mDesc)

void **movej**(point pPosition, tool tTool, mdesc mDesc)

Syntax

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

Function

Records a command for a joint movement towards the **pPosition** or **jPosition** positions, using the **tTool** and the **mDesc** movement parameters.

CAUTION:
The system does not wait for the movement to be completed before proceeding to the next VAL3 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).

Parameter

tool tTool	Tool type expression
mdesc mDesc	Movement descriptor type expression
joint jPosition	Joint type expression
point pPosition	Point type expression

See also

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

void **movel**(point pPosition, tool tTool, mdesc mDesc)

Syntax

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

Function

Records a command for a linear movement towards the **pPosition** point, using the **tTool** tool and the **mDesc** movement parameters.

> **CAUTION:**
> **The system does not wait for the movement to be completed before proceeding to the next VAL3 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).

Parameter

point pPosition	Point type expression.
tool tTool	Tool type expression.
mdesc mDesc	Movement descriptor type expression.

See also

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

void **movec**(point pIntermediate, point pTarget, tool tTool, mdesc **mDesc**)

Syntax

void movec(point pIntermediate, point pTarget, tool tTool, mdesc mDesc)

Function

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**.

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 VAL3 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).

Parameter

point pIntermediate	Point type expression.
point pTarget	Point type expression.
tool tTool	Tool type expression.
mdesc mDesc	Movement descriptor type expression.

See also

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

void **stopMove**()

Syntax

void stopMove()

Function

Stops the arm on the trajectory and suspends authorization of the programmed movement.

> **CAUTION:**
> **This instruction returns immediately: the VAL3 task does not wait for the movement to be completed before proceeding to the next instruction.**

The kinematic parameters 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

```
wait (diSignal==true)         // waits for a signal
stopMove()                    // stops movements along the trajectory
<instructions>
restartMove()                 // restarts movements along the trajectory
```

See also

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

void **resetMotion**(), void **resetMotion**(joint jStartingPoint)

Syntax

void resetMotion()
void resetMotion(joint jStartingPoint)

Function

Stops the arm on the trajectory and cancels all the stored movement commands.

> **CAUTION:**
> **This instruction returns immediately: the VAL3 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** revolute 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 revolute 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()

void **restartMove**()

Syntax

void restartMove()

Function

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 intruction has no effect.

See also

void stopMove()
void resetMotion(), void resetMotion(joint jStartingPoint)

void **waitEndMove**()

Syntax

void waitEndMove()

Function

Cancels the blending of the last movement commmand 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 instructions sent to the variable speed drives correspond 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

```
waitEndMove()
putln(sel(isEmpty(),1,-1))      // displays 1, no more commands in progress
putln(sel(isSettled(),1,-1))    // May display -1, the robot is not necessarily already stabilized
watch(isSettled(), 1)           // Waits for the robot to stabilize for 1 s maximum
```

See also

bool isSettled()
bool isEmpty()
void stopMove()
void resetMotion(), void resetMotion(joint jStartingPoint)

bool **isEmpty**()

Syntax

bool isEmpty()

Function

Returns **true** if all the movement commands have been executed, returns **false** if at least one command is still being executed.

Example

```
// If commands are in progress
if ! isEmpty()
  //Stop the robot and cancel the commands
  resetMotion()
endIf
```

See also

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

bool **isSettled**()

Syntax

bool isSettled()

Function

Returns **true** if the robot is stopped, and **false** if its position is not yet stabilized.

The position is considered as stabilized if the position error for each joint remains less than 1% of the maximum authorized position, for 50 ms.

See also

bool isEmpty()
void waitEndMove()

void **autoConnectMove**(bool bActive), bool **autoConnectMove**()

Syntax

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)

S6.4 num **getSpeed**(tool tTool)

Syntax

num getSpeed(<tool tTool>)

Function

This instruction returns the current Cartesian translation speed at the extremity 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)

S6.4 num **getPositionErr**()

Syntax

num 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)

S6.4 void **getJointForce**(num& nForce)

Syntax

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

num getPositionErr()

CHAPTER 10

OPTIONS

10.1. COMPLIANT MOVEMENTS WITH FORCE CONTROL

10.1.1. 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 **VAL3**'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.

10.1.2. 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.

10.1.3. 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.

10.1.4. LIMITATIONS

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.

10.1.5. INSTRUCTIONS

void **movejf**(joint jPosition, tool tTool, mdesc mDesc, num nForce)

Syntax

void movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)

Function

Records a compliant joint movement command towards the **jPosition** position using the **tTool** tool, the **mDesc** movement parameters, and a **nForce** force 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 VAL3 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).

Parameter

tool tTool	Tool type expression.
mdesc mDesc	Movement descriptor type expression
joint jPosition	Revolute position type position
num nForce	Numerical type expression

See also

void movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)
bool isCompliant()

void **movelf**(point pPosition, tool tTool, mdesc mDesc, num nForce)

Syntax

void movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)

Function

Records a compliant linear movement command towards the **pPosition** position using the **tTool** tool, the **mDesc** movement parameters and the **nForce** force 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 VAL3 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).

Parameter

point pPosition	Point type expression.
tool tTool	Tool type expression
mdesc mDesc	Movement descriptor type expression
num nForce	Numerical type expression

See also

void movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)
bool isCompliant()

bool **isCompliant**()

Syntax

bool isCompliant()

Function

Returns **true** if the robot is in compliant mode, otherwise returns **false**.

Example

```
movelf(pPosition, tTool, mDesc, 0)
wait(isCompliant())               // Waits for the robot to actually be in compliant mode
diEjection = true                 // Commands press ejection
waitEndMove()                     // Waits for the end of compliant movement
movej(jDepart, tTool, mDesc)      // Continues with a standard movement
```

See also

void movelf(point pPosition, tool tTool, mdesc mDesc, num nForce)
void movejf(joint jPosition, tool tTool, mdesc mDesc, num nForce)

10.2. ALTER: REAL TIME CONTROL ON A PATH

Cartesian Alter

10.2.1. 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.

10.2.2. 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 alterMovel(), 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. VAL3 execution may therefore resume after a waitEndMove even if the arm is still moving because of a changing alteration.

10.2.3. 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 **VAL3** 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.

10.2.4. 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.

10.2.5. 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.

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).

10.2.6. INSTRUCTIONS

void **alterMovej**(joint jPosition, tool tTool, mdesc mDesc)

void **alterMovej**(point pPosition, tool tTool, mdesc mDesc)

Syntax

void alterMovej(joint jPosition, tool tTool, mdesc mDesc)
void alterMovej(point pPosition, tool tTool, mdesc mDesc)

Function

Register an alterable joint move command (a line in the joint space)

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.

void **alterMovel**(point pPosition, tool tTool, mdesc mDesc)

Syntax

void alterMovel(point pPosition, tool tTool, mdesc mDesc)

Function

Register an alterable linear move command (a line in the Cartesian space)

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 movel instruction, except that it enables the alter mode for the move. See movel for more details.

void **alterMovec**
(point pIntermediate, point pTarget, tool tTool, mdesc mDesc)

Syntax

void alterMovec(point pIntermediate, point pTarget, tool tTool, mdesc mDesc)

Function

Register an alterable circular 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)

Syntax

num alterBegin(frame fAlterReference, mdesc mMaxVelocity)
num alterBegin(tool tAlterReference, mdesc mMaxVelocity)

Function

Initialize 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 centre point. This mode must be used when the deviation of the path is known or measured relatively to the tool centre 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, tvel 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, VAL3 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()**

Syntax

num alterEnd()

Function

Exit 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**)**

Syntax

num alter(trsf trAlteration)

Function

Specify a new alteration of the alterable path.

Parameter

trAlteration	Trsf expression defining the alteration to apply until the next alter instruction.

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 that 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()**

Syntax

num alterStopTime()

Function

Return 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)

S6.1 **10.3. OEM LICENCE CONTROL**

10.3.1. PRINCIPLES

An OEM licence is a controller-specific key that makes it possible to restrict the use of a **VAL3** project on some selected robot controllers.

A tool is provided with Stäubli Robotics Studio(*) 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 **VAL3** emulator, the "demo" key is enough to fully enable the OEM licence.

The **getLicence()** instruction is a **VAL3** 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 licence.exe executable delivered with the **VAL3** emulator, requires a specific SRS licence to be used.

10.3.2. INSTRUCTIONS

string **getLicence**(string sOemLicenceName, string sOemPassword)

Syntax

string getLicence(string sOemLicenceName, string sOemPassword)

Function

Returns the status of the specified OEM licence:

"oemLicenceDisabled"	The **VAL3** 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

CHAPTER 11

APPENDIX

11.1. RUNTIME ERROR CODES

Code	Description
-1	There is no task created by this application or library with the specified name
0	No runtime error
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 **VAL3** 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 **VAL3** 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.

11.2. CONTROL PANEL KEYBOARD KEY CODES

Without **Shift**

3	Caps	Space				
283	-	32				Move
					Ret.	-
2	Shift	Esc	Help			Run
282	-	255	-		270	-
	Menu	Tab	Up		Bksp	Stop
	-	259	261		263	-
1	User	Left	Down		Right	
281	-	264	266		268	

With **Shift**

3	Caps	Space				
283	-	32				Move
					Ret.	-
2	Shift	Esc	Help			Run
282	-	255	-		270	-
	Menu	UnTab	PgUp		Bksp	Stop
	-	260	262		263	-
1	User	Home	PgDn		End	
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

INDEX