Asyfeed module

Programming Guide

```
NoTool:=0;
NoPosRT:=1;

(*---Blend Definition---*)
TrajectoryBlend:= LoadData 'trajectory_blend_[mm]';
TrajectoryBlend:= TrajectoryBlend/1000;
NoBlend:=0;

(*---Speed Definition---*)
HighSpeed:= LoadData 'high_speed_[%]';
HighSpeed:= HighSpeed/100;
ApproachPickSpeed:= LoadData 'approach_pick_speed_[%]';
ApproachPickSpeed:= ApproachPickSpeed/100;
PickSpeed:= LoadData 'pick_speed_[%]';
PickSpeed:= PickSpeed/100;
TakePartAfterPickSpeed:= LoadData 'zway_after_pick_speed_[%]';
TakePartAfterPickSpeed:= TakePartAfterPickSpeed/100;
ApproachPlaceSpeed:= LoadData 'approach_place_speed_[%]';
ApproachPlaceSpeed:= ApproachPlaceSpeed/100;
PlaceSpeed:= LoadData 'place_speed_[%]';
PlaceSpeed:= PlaceSpeed/100;
DisengagementXSpeedPlace:= LoadData 'xway_after_place_speed_[%]';
DisengagementXSpeedPlace:= DisengagementXSpeedPlace/100;

 (*---Height with component height---*)
PartHeight:=LoadData 'part_height_[mm]';
PartHeight:=PartHeight/1000;
TrajectoryPickHeight:= LoadData 'trajectory_pick_height_[mm]';
TrajectoryPickHeight:= TrajectoryPickHeight/1000;
TrajectoryPickHeight:=TrajectoryPickHeight+PartHeight;
ApproachPickHeight:= LoadData 'approach_pick_height_[mm]';
```

Document	MODULE_ASYFEED_Programming_Guide_EN 000.100.498		
Version	C	**Date**	06.04.2018
Robot controller version	Since V4.6.0		
Asyview controller version	Since V4.0.0		

Table of Contents

1. Introduction

This document is the property of Asyril S.A.; it may not be reproduced, modified or communicated, in whole or in part, without our prior written authorisation. Asyril S.A. reserves the right to modify any information contained in this document for reasons related to product improvements without prior notice. Before using the product, please read this entire document in order to ensure that the product is used correctly. However, if you encounter difficulties when using the product, do not hesitate to contact our customer service department.

In this manual, the safety information that must be respected is split into three types: "Danger", "Important" and "Note". These messages are identified as follows:

DANGER!

Failure to respect this instruction may result in serious physical injury.

DANGER!

This instruction identifies an electrical hazard. Failure to respect this instruction may result in electrocution or serious physical injury due to an electric shock.

IMPORTANT!

Failure to respect this instruction may result in serious damage to equipment.

NOTE:

The reader's attention is drawn to this point in order to ensure that the product is used correctly. However, failure to respect this instruction does not pose a danger.

Reference …

For more information on a specific topic, the reader is invited to refer to another manual or another page of the current manual.

IMPORTANT!

Asyril cannot be held responsible for damage to property or persons caused by the failure to respect the instructions contained in the manual for your machine.

NOTE:

All dimensions and values are expressed in millimetres (mm) in this manual.

1.1. Related manuals

As described in the Table 1-1, this manual is an integral part of the Asyfeed Pocket Module documentation set. This manual covers the different instructions and their syntax which can be used to program the pick & place application within the Asyfeed Pocket Module.

Manual Title	Manual reference	Description of the content
Operating manual	MODULE_ASYFEED_Operating_Manual_EN MODULE_ASYFEED_Manuel_Instructions_FR MODULE_ASYFEED_Bedienungsanleitung_DE	Technical description of the product, electrical and mechanical interfaces, maintenance and transport information.
HMI manual		Accessible directly via the HMI
Programming Guide	MODULE_ASYFEED_Programming_Guide_EN	THIS MANUAL
User Guide	MODULE_ASYFEED_User_Guide_EN MODULE_ASYFEED_Manuel_Utilisation_FR	Description of the main functionalities of the machine and of the standard P&P and calibration cycles
User Guide	SMARTSIGHT_User_Guide_EN SMARTSIGHT_Manuel_Utilisation_FR SMARTSIGHT_Benutzerhandbuch_DE	Describes how to configure the feeding and the vision detection

Table 1-1 : Related manuals

2. General information

2.1. ARL, definitions and main characteristics

ARL (Asyril Robotics Language) is the programming language for the Asyril robot controller. It is a structured language, the syntax of which is derived from normalised language (IEC 61131-3 ST - Structured Text). Its role is to provide a means to automate any task performed by the controller. It is used to define sequences of actions performed during a recipe.

There are various types of data, (NUMBER, STRING, BOOLEAN, VECTOR,…) as well as conditional test instructions (IF, SWITCH), loops (FOR, WHILE, REPEAT) and calculation possibilities with expressions and mathematical functions (SIN, COS, etc.)
A set of commands enables various software modules to be controlled (robot, vision system, inputs/outputs, etc.)

2.2. ARL files

ARL is a language that is compiled in a format specific to the Asyril platform.
The compilation chain is set out below.
The ARL source program file in XML format is preprocessed and compiled in order to obtain an executable file that is stored in the random access memory.
Execution is ensured by the ARL interpreter integrated into the robot controller.

General compilation diagram:

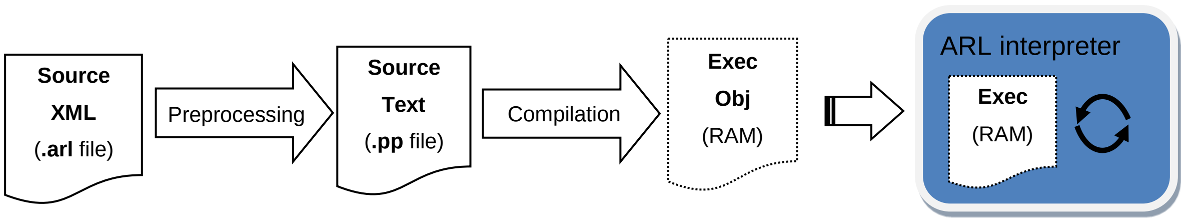


Figure 2-1: Compilation chain

<u>Source file (.arl extension)</u>
This XML file contains the text for the program as well as a set of "data" (resource) that can be accessed whilst the program is being executed.

<u>Preprocessed file (.pp extension)</u>

Files with the ".pp" extension are intermediate files that result from the compilation process for .arl programs. They are the result of the preprocessing operation for each program. These files are in plain text format.

Preprocessing covers operations associated with handling the program text. It includes processing macros (substitution) and removing comments from the source code.

These files are only created on the file system as a "trace" of the compilation process. They are not used directly by the compiler.

<u>Executable file</u>

Executable files are only stored in the memory. They are in a Asyril platform-specific format. They are the result of the actual compilation operation.

Only programs that successfully complete the compilation step are saved in the memory.

2.3. ARL program syntax

An ARL program is composed of one or more instructions that are evaluated and executed in turn; each instruction is a string of characters terminating in a semicolon (;). Comments identified by (* … *) are not evaluated.

3. Types of data

There are several types of data in ARL:

Data type	Example	Abbreviation used in this manual	Notation of variables used in this manual
Number	`125,458`	*real*	*r, for example rValue*
String	`'Arl'`	*String*	*s, for example sChain*
Vector	`(1,3,1,1,Var)`	*Vector*	*v, example vPosition*
Boolean	`true`	*bool*	*b, example bSlowSpeed*

The type is determined once a variable has been assigned. However, when a type has been assigned to a variable, it cannot be modified.

For example, the sequence of instructions below will generate a runtime error.

```
var:=1;
var:=(0.5,0.5,0.5);
```

The names of the variables must begin with an alphabetical character and contain no special characters (punctuation, etc.).

4. Operators

4.1. Summary table

The table below gives an exhaustive list of the operators that may be used in ARL. It describes the syntax and the type of variable that can be used for each of the operators described. A more precise description of each operator is given after this summary table.

Operator	Description	Syntax	Type of V1	Type of V2	Type of result
:=	**Assignation**	Var:= <V1>	real	-	real
			Vector	-	Vector
			String	-	String
			bool	-	bool
**	**Power**	<V1> ** <V2>	real	real	real
-	**Negation**	- <V1>	real	-	real
NOT	**Complement**	NOT <V1>	bool	-	bool
*	**Multiplication**	<V1> * <V2>	real	real	real
/	**Division**	<V1> / <V2>	real	real	real
MOD	**Modulo**	<V1>MOD<V2>	real	real	real
+	**Arithmetic addition**	<V1> + <V2>	real	real	real
	Vector addition		Vector	Vector	Vector
	Concatenation		String	String	String
-	**Arithmetic subtraction**	<V1> - <V2>	real	real	real
	Vector subtraction		Vector	Vector	Vector
<	**"Less than" comparison**	<V1> < <V2>	real	real	bool
>	**"Greater than" comparison**	<V1> > <V2>	real	real	bool
<=	**"Less than or equal" comparison**	<V1> <= <V2>	real	real	bool
>=	**"Greater than or equal" comparison**	<V1> >= <V2>	real	real	bool
<>	**Different from**	<V1> <> <V2>	real	real	bool
			Vector	Vector	bool
			String	String	bool
			bool	bool	bool

Operator	Description	Syntax	Type of V1	Type of V2	Type of result
=	**Equal to**	<V1> = <V2>	real	real	bool
			Vector	Vector	bool
			String	String	bool
			bool	bool	bool
AND	**"and" logic**	<V1> **AND** <V2>	bool	bool	bool
OR	**"or" logic**	<V1> **OR** <V2>	bool	bool	bool
XOR	**"exclusive or" logic**	<V1> **XOR** <V2>	bool	bool	bool

4.2. Detailed description

ASSIGNATION

Syntax: Variable**:=** <rValue>

Function: Assigns the **rValue** value to the **Variable** variable.

NOTE:
the value and the variables must be of the same type: boolean, vector with same number of coordinates, character string, number, etc.

Example:
```
PICKPOS:= (0.5,0.5,0,0,0,0);
```

See also:

POWER

Syntax: *<rValue>* ****** *<rPower>*

Function: Used to raise the **rValue** to the **rPower** power.

Parameters:

rValue	numerical expression
rPower	numerical expression

Example: `Val:= 2**10; (* result: val =` 2^{10} `= 1024*)`

See also:

NEGATION

Syntax: - *<rValue>*

Function: Used to return the opposite of the numerical expression **rValue**

Parameters:

rValue	numerical expression

Example:
```
Val:= -2; (* result: val = -2 *)
```

See also:

COMPLEMENT

Syntax: **NOT** *<rValue>*

Function: Used to establish the logical negation of **bValue**

Parameters:

bValue	boolean expression

Example:
```
A:= true;
B:= 5;
C:= 10;
Val:= NOT A; (* result: val = false *)
Val:= NOT (B > C); (* result: val = true *)
Val:= NOT (NOT A); (* result: val = true *)
```

See also:

MULTIPLICATION

Syntax: *<rValue1>* ***** *< rValue2>*

Function: Used to calculate the product of **rValue1** and **rValue2**

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 2 * 2; (* result: val = 4 *)
```

See also:

DIVISION

Syntax: *<rValue1>* ***/*** *< rValue2>*

Function: Used to calculate the quotient of **rValue1** and **rValue2**

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 5/2;  (* result: val = 2.5 *)
val:= 1/3; (* result: val = 0.3333333333333333 *)
```

See also:

MODULO

Syntax: *<rValue>* **MOD** *<rModulo>*

Function: Used to return **rValue** modulo **rModulo**. i.e. the rest of the euclidean division of **rValue** by **rModulo**. An error is generated if **rModulo** is zero.

Parameters:

rValue	numerical expression
rModulo	numerical expression

Example:

```
Val:= 15 MOD 10;  (* result: val = 5 *)
```

See also:

ARITHMETIC ADDITION, VECTOR ADDITION or CONCATENATION

Syntax: *<rvsValue1>* **+** *< rvsValue2>*

Function: Used to add two real numbers, two vectors (member to member addition) or to concatenate two character strings.

Parameters:

rvsValue1	numerical expression, vector or character string
rvsValue2	of the same type as **rvsValue1**

Example:

```
Val:= 5 + 3;  (* result: val = 8 *)
Val:= (1,0,1,1)+(1,1,5,1);  (* result: val = (2,1,6,2) *)
Val:= 'abc' + 'def';(* result: val = 'abcdef' *)
```

See also:

ARITHMETIC SUBTRACTION or VECTOR SUBTRACTION

Syntax: *<rValue1>* - *< rValue2>*

Function: Used to subtract two real numbers or two vectors (member to member subtraction)

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 5 - 3; (* result: val = 2 *)
Val:= (1,0,1,1)-(1,1,5,1); (* result: val = (0,-1,-4,0) *)
```

See also:

"LESS THAN"COMPARISON

Syntax: *<rValue1>* **<** *< rValue2>*

Function: Used to compare two numerical expressions. An instruction using this operator returns `true` if **rValue1** is strictly less than **rValue2** and `false` if not.

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 3 < 5; (* result: val = true *)
(*false if the result is greater than or equal to*)
```

See also:

"GREATER THAN" COMPARISON

Syntax: *<rValue1>* **>** *< rValue2>*

Function: Used to compare two numerical expressions. An instruction using this operator returns `true` if **rValue1** is strictly greater than **rValue2** and `false` if not.

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 5 > 5; (* result: val = false *)
(*false if the result is less than or equal to*)
```

See also:

"LESS THAN OR EQUAL TO" COMPARISON

Syntax: *<rValue1>* **<=** *< rValue2>*

Function: Used to compare two numerical expressions. An instruction using this operator returns `true` if **rValue1** is less than or equal to **rValue2** and `false` if not.

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 3 <= 5; (* result: val = true *)
(*false if the result is strictly greater than*)
```

See also:

"GREATER THAN OR EQUAL TO" COMPARISON

Syntax: *<rValue1>* **>=** *< rValue2>*

Function: Used to compare two numerical expressions. An instruction using this operator returns `true` if **rValue1** is greater than or equal to **rValue2** and `false` if not.

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 5 >= 5; (* result: val = true *)
(*false if the result is strictly less than*)
```

See also:

DIFFERENT FROM

Syntax: *<rbvsValue1>* **<>** *< rbvsValue2>*

Function: Used to compare two numerical expressions. An instruction using this operator returns `true` if **rValue1** differs from **rValue2** and `false` if not.

Parameters:

rbvsValue1	numerical expression, boolean, vector or character string
rbvsValue2	of the same type as **rbvsValue1**

Example:

```
Val:= 5 <> 5; (* result: val = false *)
Val:=  true <> false; (* result: val = true *)
Val:= 'test' <> 'test1'; (* result: val = true *)
Val:= (0,0,1,0) <> (0,0,0,0); (* result: val = true *)
```

See also:

EQUAL TO

Syntax: *<rValue1>* **=** *< rValue2>*

Function: Used to compare two numerical expressions. An instruction using this operator returns `true` if **rValue1** is equal to **rValue2** and `false` if not.

Parameters:

rValue1	numerical expression
rValue2	numerical expression

Example:

```
Val:= 5 = 5; (* result: val = true *)
Val:= true = false; (* result: val = false *)
Val:= 'test' = 'test1'; (* result: val = false *)
Val:= (0,0,1,0) = (0,0,0,0); (* result: val = false *)
```

See also:

AND LOGIC

Syntax: *<bValue1>* **AND** < *bValue2>*

Function: Used to establish the logical conjunction "and" between two expressions:

bValue1	bValue2	result
true	false	false
true	true	true
false	true	false
false	false	false

Parameters: *bValue1* boolean

bValue2 boolean

Example: `Val:= true AND true; (* result: val = true *)`

See also:

OR LOGIC

Syntax: *<bValue1>* **OR** < *bValue2>*

Function: Used to establish the logical disjunction "or" between two expressions:

bValue1	bValue2	result
true	false	true
true	true	true
false	true	true
false	false	false

Parameters: *bValue1* boolean

bValue2 boolean

Example: `Val:= true OR false; (* result: val = true *)`

See also:

EXCLUSIVE OR LOGIC

Syntax: *<bValue1>* **XOR** < *bValue2>*

Function: Used to establish the logical exclusion "exclusive or" between two expressions:

bValue1	bValue2	result
true	false	true
true	true	false
false	true	true
false	false	false

Parameters: *bValue1* boolean

bValue2 boolean

Example: `Val:= true XOR true; (* result: val = false *)`

See also:

5. Control instructions

(* COMMENT *)

Syntax: **(*** <string> ***)**

Function: Any character string flanked by the opening symbol (parenthesis star) and closing symbol (star parenthesis) will not be evaluated and the execution process continues after this string.

NOTE:
The use of comments is not authorised during programming from the robot web interface. Only the HMI interface enables comments to be used.

Example:
```
(* this is an example of a comment *)
```

See also:

Control instruction **IF**

Syntax:
IF <*bool* bCondition> **THEN**
<*instructions*>
ELSEIF <*bool* bCondition1> **THEN**
<*instructions*>
[…]
ELSEIF <*bool* bConditionN> **THEN**
<*instructions*>
ELSE
<*instructions*>
END_IF

Function: This **IF**, **THEN ELSEIF**, **THEN**, **ELSE**, **END_IF** sequence successively evaluates the bCondition conditions. If the result of the evaluation is true, the instruction sequence that follows is performed up to the next keyword (**ELSEIF**, **ELSE** or **END_IF**). If none of the **ELSEIF** conditions are met, the instructions following the keyword **ELSE** will be evaluated. The program then resumes after **END_IF**.

NOTE:
when different instructions are to be executed according to several possible values for one parameter, it is preferable to use the **SWITCH OF**, **CASE**, **ELSE**, **END_SWITCH** sequence.

Example:
```
IF ((i<>0 OR j<>0) AND (i<>14 OR j<>0)) THEN
(* do whatever you want *)
ELSEIF ((i <> 2) AND (j <> 2)) THEN
(* do something else *)
ELSE
(*do something else again *)
END_IF
```

See also: **SWITCH**

Control instruction **SWITCH**

Syntax: **SWITCH** <Variable> **OF**
CASE <*value1*>**:** <*instructions1*>
[…]
CASE <*valueN*>**:** <*instructionsN*>
ELSE
<*default instruction*>
END_SWITCH

Function: This **SWITCH**, **OF**, **CASE**, **ELSE**, **END_SWITCH** sequence successively evaluates the expressions indicated by the keyword **CASE** until a value equal to the initial variable is found between the keywords **SWITCH** and **OF**. The instructions that follow are then evaluated. If none of the **CASE** conditions are met, the instructions following the keyword **ELSE** will be evaluated. The program then resumes after **END_SWITCH**.

NOTE:
the values or variables following **CASE** must be of the same types as those following **SWITCH**.

Example:

```
(* take a part, wait or move to trash depending on the value of a flag *)

SWITCH PickFlag OF
 CASE 0: MOVETO Pick World ToolID HighSpeed NoBlend NoAction;
 CASE 1: SLEEP 200;
 ELSE MOVETO Trash World ToolID HighSpeed NoBlend NoAction;
END_SWITCH
```

See also: **IF**

Control instruction **FOR**

Syntax: **FOR** <nCounter>**:=** <nStart> **TO** <nEnd> **BY** <nStep>
DO
<instructions>
END_FOR

Function: This sequence executes the instructions between **DO** and **END_FOR** until the counter **nCounter** is equal to its end value **nEnd**. The counter is initialised by the **nStart** value and increased by **nStep** each time it passes through the loop.

Example:

```
(* fill a pallet of x_max * y_max positions *)
FOR x:=init_x TO x_max BY 1
DO
     FOR y:=init_y TO y_max BY 1
     DO
    (* pick parts, transport and place *)
     END_FOR
END_FOR
```

See also:

Control instruction **WHILE**

Syntax: **WHILE** <bCondition> **DO**
<instructions>
END_WHILE

Function: This sequence executes the instructions between **DO** and **END_WHILE** while the boolean condition **bCondition** is evaluated as true. If the condition is false after the first evaluation, the instructions are not executed, unlike the control instruction **REPEAT**.

Example:

```
(* fill a tube of 500 positions *)

PlaceCapacity:=500;
PlaceCounter:=0;
WHILE PlaceCounter <= PlaceCapacity DO
   (* pick part *)
   (* transport *)
   (* place part *)
   PlaceCounter:=PlaceCounter+1;
END_WHILE
```

See also: **REPEAT**

Control instruction **REPEAT**

Syntax: **REPEAT**
<instructions>
UNTIL <bCondition>
END_REPEAT

Function: This sequence executes the instructions between **REPEAT** and **UNTIL**, while the boolean condition **bCondition** is evaluated as true. If the condition is false after the first evaluation, the instructions will still be executed once, unlike the control instruction **WHILE**.

Example:

```
(* wait until the image has finished being analysed by Asyview
*)

REPEAT
sleep 20;
 S:=ModuleCmd'AsyView''GetState DeviceID=1 Name=SearchRunning';
UNTIL S=true;
END_REPEAT;
```

See also: **WHILE**

Control instruction **EXIT**

Syntax: **EXIT**

Function: Exit the function currently being executed

Example:

```
(* exit "while" if val:=1 *)
   WHILE true DO
   IF val:=1 THEN
     (*do something *)
      EXIT;
   END_IF
   (*do something else *)
END_WHILE
```

See also: **WHILE**

6. "Maths" instructions

real **COS** *rAngle*

Syntax: COS <*rAngle*>

Function: This function returns the cosine of **rAngle**. The result is between -1 and 1

Parameters: *rAngle* angle in radians

Example:
```
Val:= COS Pi/2; (* result: val = 0 *)
```

See also: **ACOS**

real **ACOS** *rValue*

Syntax: ACOS <*rValue*>

Function: This function returns the reverse cosine of **rValue** in radians. The result is between 0 and π radian. An error is generated if **rValue** is not between -1 and 1.

Parameters: *rValue* numerical expression

Example:
```
Val:= ACOS 0; (* result: val = π/2  *)
```

See also: **COS**

real **SIN** *rAngle*

Syntax: SIN <*rAngle*>

Function: This function returns the sine of **rAngle**. The result is between -1 and 1

Parameters: *rAngle* angle in radians

Example:
```
Val:= SIN Pi/2; (* result: val=1 *)
```

See also: **ASIN**

real **ASIN** *rValue*

Syntax: ASIN <*rValue*>

Function: This function returns the reverse sine of **rValue** in radians. The result is between -π/2 and + π/2 radians. An error is generated if **rValue** is not between -1 and 1.

Parameters: *rValue* numerical expression

Example:
```
Val:= ASIN 1; (* result: val = π/2 *)
```

See also: **SIN**

real **TAN** *rAngle*

Syntax: TAN <*rAngle*>

Function: This function returns the tangent of **rAngle**. An error is generated if **rAngle** is π/2 or - π/2.

Parameters: *rAngle, real type* angle in radians

Feedback:

Example:
```
Val:= TAN Pi/4; (* val: alpha = 1 *)
```

See also: **ATAN**

real **ATAN** *rValue*

Syntax: ATAN <*rValue*>

Function: This function returns the reverse tangent of **rValue** in radians. The result is strictly between - π/2 and + π/2 degrees.

Parameters: *rValue* numerical expression

Example: `Val:= ATAN 1; (* result: val = π/4 *)`

See also: **TAN**

real **ABS** *rValue*

Syntax: ABS <*rValue*>

Function: This function returns the absolute value of the numerical expression **rValue**

Parameters: *rValue* numerical expression

Example: `Val:= ABS -90; (* result: val = 90 *)`

See also:

void **SETVECTORVALUE** *vVector rPosition nValue*

Syntax: **SETVECTORVALUE** <vVector> <rPosition> <rValue>

Function: Assigns the value of **nValue** to the position **rPosition** of the vector **vVector**

Example:

```
(* modify the z coordinate by one point *)
desPos:= (0.2,-0.12,0,0);
Height:=0.05;
SETVECTORVALUE desPos 3 Height;
```

after this sequence, the vector `desPos` is: `(0.2,-0.12,0.05,0)`

See also: **GETVECTORVALUE**

real **GETVECTORVALUE** *vVector rPosition*

Syntax: **GETVECTORVALUE** <vVector> <rPosition>

Function: This function reads and sends the value of the n-th coordinate (**rPosition**) for the vector **vVector**

Example:

```
(* store the z coordinate for a point *)
ActualPos:= GETPOSITION;
Height:= GETVECTORVALUE ActualPos 3;
```

after this sequence, the `Height` variable will contain the actual coordinate of the robot for Z.

See also: **GETPOSITION**

7. Instructions by module

7.1. "Module" concept

It is possible to send a command to various modules via ARL. These modules are:

- The ROBOT module
- The ASYVIEW module
- The IOWAGO module
- The TURNING TABLE module
- The MATH module

Each module has specific instructions, which must be written using appropriate syntax. The following chapters describe each module, as well as the associated objects and instructions.

7.2. Robot module

7.2.1. Robot movements

The robot's platform executes the interpolated movements along a line. All the movement parameters associated with a trajectory are given in the system of coordinates (X,Y,Z). Two types of movement are possible:

✓ "Point to point" trajectory

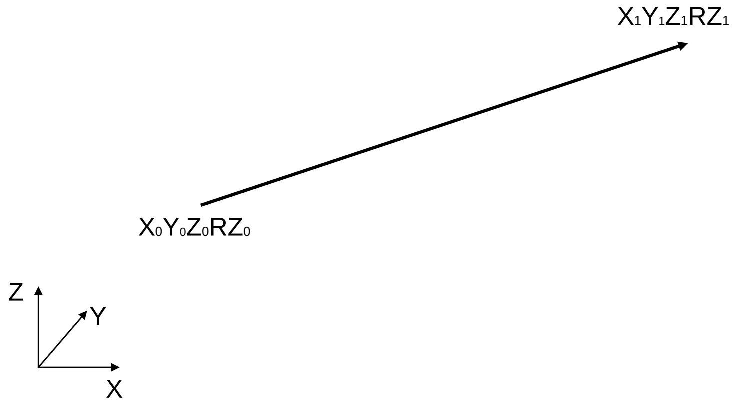


Figure 7-1: "Point to point" trajectory

During a point to point trajectory, the robot follows a straight line defined in a Cartesian reference from its current position ($X_0Y_0Z_0RZ_0$) up to the desired position ($X_1Y_1Z_1RZ_1$).

✓ Trajectory with blend

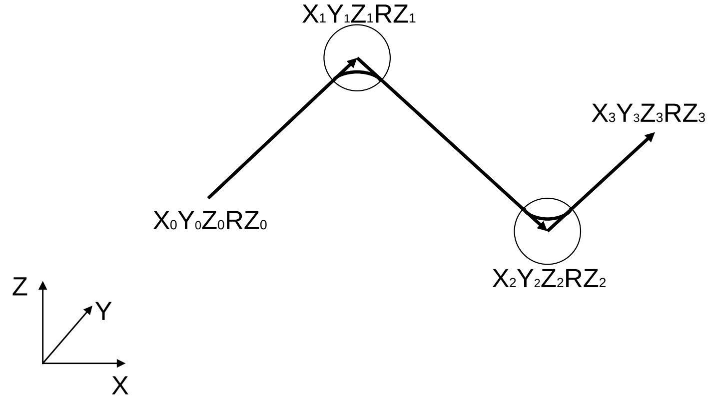


Figure 7-2: Multi-point trajectory with "blend"

A "blend" parameter is used to go from one target position to another without the robot stopping, thanks to curved trajectories during changes in direction. The "blend" factor represents a radius around the target position. When the robot enters the circle defined by this radius, the next position, stored in the task planner buffer, is calculated. The result is represented on the figure above.

7.2.2. Position in the work volume

The work volume of the robot may be subdivided thanks to four types of object:

- ✓ **The frames**, used to describe a geometric sub-reference (X' Y' Z') in the work volume of the robot (X Y Z).
- ✓ **The points**, used to represent a specific position in the work volume of the robot
- ✓ **The tools**, used to define the offsets caused by using a tool in relation to the robot's platform.
- ✓ **The motion sequence**, used to define a sequence of movements

Each of these objects can be saved in the robot controller so that they can be subsequently used. All of these elements are represented and defined in XML format.

IMPORTANT NOTE:

- ✓ *It is possible to define frames within frames*
- ✓ *The identifier 0 (frame, tool, point) represents the robot's world*
- ✓ *The identifiers permitted range from 1 to 99*

Type 0 FRAME

Definition: The type 0 frame is used to define an orthonormal reference from a vector and a plane as shown on the figure below:

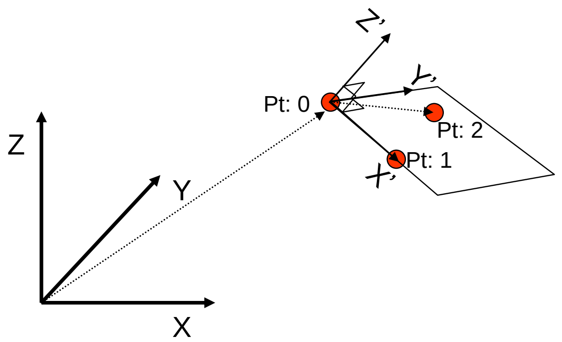


Figure 7-3: Defining a type 0 frame

Parameters:

ptconfig id='0'	defines the origin of the Cartesian system
ptconfig id='1'	defines the direction of the **x** axis of the reference
ptconfig id='2'	defines the orientation of the plane in the work volume
id	frame identifier

XML structure:

```
<world>
      <frame id='' type='0'>
            <ptconfig id='0' x='' y='' z='' rz=''>
            <ptconfig id='1' x='' y='' z='' rz=''>
            <ptconfig id='2' x='' y='' z='' rz=''>
      </frame>
</world>
```

Type 1 FRAME

Definition: The type 1 frame enables a reference to be defined from two vectors (i.e. three points). The reference is not necessarily orthonormal; the scales for the x and y axes are defined by the parameters xdw and ydw. The scale for the z axis remains metric (scale of the global robot reference). The direction of the z axis is defined orthogonally to the x and y axes, as shown on the figure below:

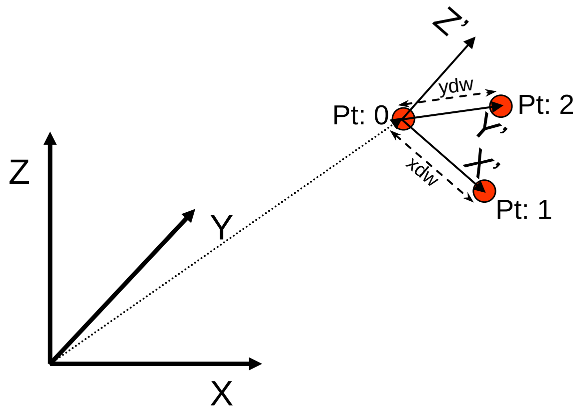


Figure 7-4: Defining a type 1 frame

Parameters:

ptconfig id='0'	defines the origin of the Cartesian system
ptconfig id='1'	defines the direction of the **x** axis of the reference
ptconfig id='2'	defines the orientation of the plane in the work volume
frame id	frame identifier
xwd	distance between points 0 & 1
ywd	distance between points 0 & 2

XML structure:

```
<world>
<frame id='' type='1'>
            <ptconfig id='0' x='' y='' z='' rz=''/>
            <ptconfig id='1' x='' y='' z='' rz=''/>
            <ptconfig id='2' x='' y='' z='' rz=''/>
            <calib xdw='0' ydw='0'/>
</frame>
</world>
```

Type 2 FRAME

Definition: The type 2 frame is used to define a reference from a list of at least three points for which the coordinates in the work volume (ptconfig) and in the new reference (ptcalib) must be provided. The reference is defined by adjusting the positions on a plane as shown in the figure below:

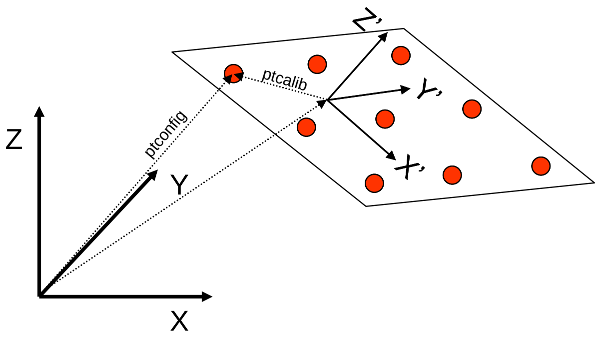


Figure 7-5: Defining a type 2 frame

Parameters:

ptconfig	Coordinates of each point in the robot's work volume
ptcalib	Theoretical coordinates for the corresponding point in the plane (x;y) of the new reference (X' Y' Z')
id	frame identifier

XML structure:

```
<world>
<frame id='' type='2'>
            <ptconfig id='0' x='' y='' z='' rz=''/>
            <ptconfig id='1' x='' y='' z='' rz=''/>
            <ptconfig id='2' x='' y='' z='' rz=''/>
            <ptcalib id='0' x='' y='' z='' rz=''/>
            <ptcalib id='1' x='' y='' z='' rz=''/>
            <ptcalib id='2' x='' y='' z='' rz=''/>
      </frame>
</world>
```

Type 3 FRAME

Definition:

The type 3 frame called a "meshframe" is a specific frame that is designed to correct the differences between a setpoint position and the final position of the robot in a plane.

It is used to define a "corrective chart" formed by a mesh of points.

This chart will be applied to any position included in the mesh by linear interpolation.

The frame is composed of three elements. Two "internal" frames and a mesh of points.

1) The internal calibration frame

This defines the mesh plane. It must be type 0 and defined in relation to the frame world. It defines the frame in which the mesh positions are expressed.

2) The internal work frame

This is optional and does not influence the correction quality. It is defined as a sub-frame of the calibration frame. It enables a work system to be freely defined.

It may be type 0, 1 or 2.

3) Point mesh

This defines a set of points, each of which must be located at an intersection of a rectangular grid having fixed dimensions. The points are defined in the internal calibration frame.

Each mesh point must be located exactly at the intersection of the grid but all of the points may form any surface without any restriction in size or "type" (with or without holes, etc.)

The origin of the calibration frame itself is not required to be joined with an intersection.

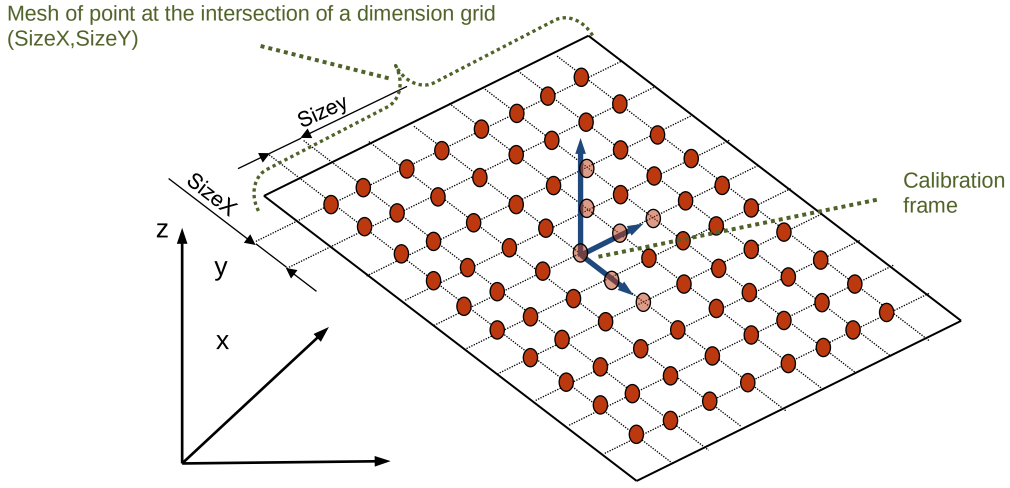
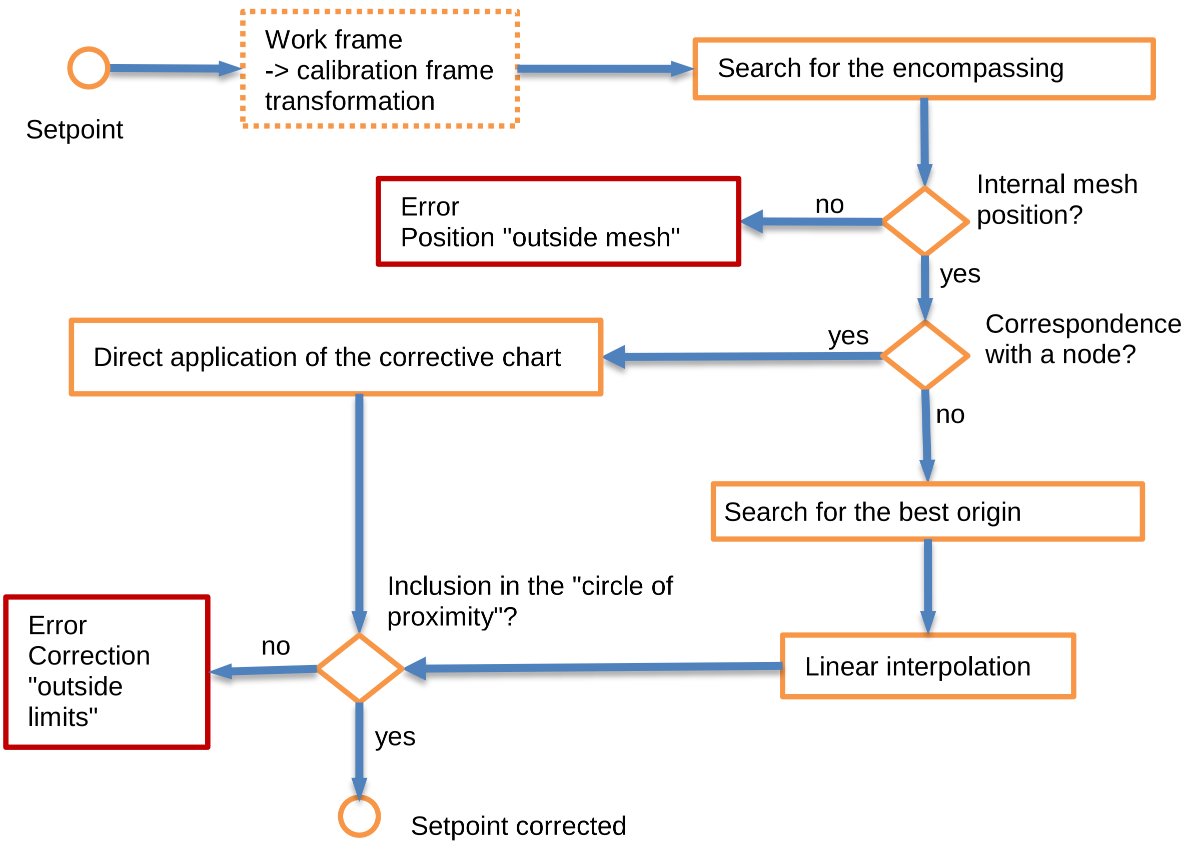


Figure 7-6: Defining a type 3 frame

Correction/interpolation principle



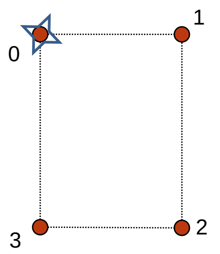


Fig 8.7
Direct application of the corrective chart.

The setpoint position corresponds exactly to node 0 of the mesh.
The correction is obtained directly by searching for the corresponding measured position.

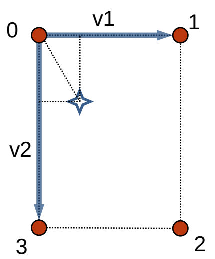


Fig 8.8
Linear interpolation case 1

The setpoint position does not form part of the chart.
Mesh node 0 is chosen as the origin as it is the closest.
The position is interpolated in the reference (v1,v2)

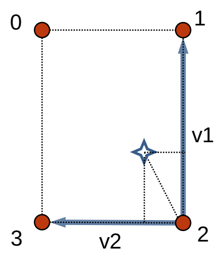


Fig 8.9
Linear interpolation case 2

The setpoint position does not form part of the chart.
Mesh node 2 is chosen as the origin as it is the closest.
The position is interpolated in the reference (v1,v2)

Parameters:	*ptconfig*	Coordinates of each point in the robot's work volume
	ptcalib	Theoretical coordinates for the corresponding point in the plane (x;y) of the new reference (X' Y' Z')
	Id	frame identifier.
	mesh	Definition of global mesh characteristics.
	Node	Point doublet (src, dst) defining a vertex of a mesh
	Src	(Source) Target position (setpoint) in the calibration frame reference.
	Dst	(Destination) Measured position that corresponds to the setpoint position Src

Definition of global mesh characteristics (mesh):

#	**Attribute**	**Signification**
1	type	Determines the type of mesh. The only value permitted with the current version is "constant". This corresponds to a rectangular mesh having a fixed dimension = sizex and sizey
2	sizex, sizey	Dimension of the mesh in direction x respectively y
3	filter	Spatial "filter" parameter for the mesh positions. Enables just some of the points defined in the mesh to be taken into account temporarily. The parameter is formed by a string of characters that defines the centre and the dimensions of the filter. The separator is the colon character ":". Two types of filter are currently possible. 1) ring – Corresponding to a ring. syntax: <centerX:centerY:innerRadius:OuterRadius> Example: "0.0:0.0:0.005:0.010" 2) rectangle – Corresponding to a rectangular window. syntax: <centerX:centerY:sizeX:sizeY> Example: "0.0:0.005:0.020:0.020"
4	proxradius	Proximity radius. Defines an upper limit for the difference (Euclidean standard) between the setpoint position (Src) and the measured position (Dst)

XML structure:

```
<frame id='<MeshFrameid>' type='3'>
      <frame id='<calibFrameId>' type='0' >
            <frame id='70' type='0' >
            <ptconfig id='0' x='' y='' z='' rz='' />
            <ptconfig id='1' x='' y='' z='' rz='' />
            <ptconfig id='2' x='' y='' z='' rz='' />

            <frame id='<workingFrameId>' type='0'
                  <ptconfig id='0' x='' y='' z='' rz='' />
                  <ptconfig id='1' x='' y='' z='' rz='' />
                  <ptconfig id='2' x='' y='' z='' rz='' />
            </frame>
      </frame>

      <mesh type='constant' sizex='' sizey='' filter=''proxradius=''>

            <node id='0' teached='' proxdistance='' >
                  <srcpos x='' y='' z='' rz='' />
                  <dstpos x='' y='' z='' rz='' />
            </node>

            <node id='1' teached='' proxdistance='' >
                  <srcpos x='' y='' z='' rz='' />
                  <dstpos x='' y='' z='' rz='' />
            </node>
            .
            .
            .

      </mesh>

</frame>
```

TOOL

Definition: TOOLs are used to define the offsets caused by using a tool in relation to the robot's platform.

Parameters:

x='_'	Offset according to robot axis X
y='_'	Offset according to robot axis Y
z='_'	Offset according to robot axis Z
rz='_'	Offset around robot axis RZ
id	Tool identifier

XML structure:

```
<world>
      <tool id='' x='' y='' z='' rz=''/>
</world>
```

POINTS

Definition: POINTS are objects used to represent a specific position in the work volume of the robot. They are not related to a specific frame, but they must be used in a frame.

Parameters:

x='_'	Position according to the X axis
y='_'	Position according to the Y axis
z='_'	Position according to the Z axis
rz='_'	Position around the RZ axis
id	Tool identifier

XML structure:

```
<world>
      <pt id='' x='' y='' z='' rz=''/>
</world>
```

7.2.3. I/O Action codes

It is possible to synchronise the digital outputs with the movements of the robot using "Action-Codes". The action-codes are sent with the robot command; they define what it must do and at what moment. The action to be performed is always coded using three digits and the value of the action (action duration, distance before action, etc.) is defined by a real number. An action code always takes the following format: XXX.XXXXXX

7.2.3.1. General table

	action-code				Action value
	1st digit	2nd digit	3rd digit	.	
No action	**0**	**0**	**0**	**.**	**0**
enter the value of an output, x metres before the target position	**1**	**output to activate: 0 to 9**	**output value: 0 or 1**	**.**	**Distance (in [m]) to the target position at which the output is activated**
Trigger an output x metres before the target position for 100 ms	**2**	**output to trigger: 0 to 9**	**output value for 100 ms: 0 or 1**	**.**	**Distance (in [m]) to the target position at which the output is triggered**
pick action: trigger suction x metres before the target position	**3**	**0**	**0**	**.**	**Distance (in [m]) to the target position at which suction is activated (output 0)**
place action: stop suction and blow for x milliseconds once in position	**4**	**0**	**0**	**.**	**Time (in [ms]) during which blowing is activated (output 1)**
Check the status of an input and stop the movement if the status of the input is not the desired status	**5**	**input to check: 0 to 9**	**desired input value: 0 or 1**	**.**	**none**
Synchronise the values of three outputs x metres before the target position	**6**	**0**	**0 to 7**	**.**	**Distance (in [m]) to the target position at which the output is triggered**
"Two-stage" place action with flow limitation	**7***	**(not used)**	**(not used)**		**Delay (in [ms]) between activation of blowing and activation of flow limitation.**

IMPORTANT NOTE:
Outputs 0 and 1 are fixed and cannot be modified. If a vacuum gripper is used, output 0 must be connected to the suction and output 1 to the blowing.

Details of action-code **60X.XXX**:

Code	*Output 0*	*Output 1*	*Output 2*
600.xxx	False	False	False
601.xxx	True	False	False
602.xxx	False	True	False
603.xxx	True	True	False
604.xxx	False	False	True
605.xxx	True	False	True
606.xxx	False	True	True
607.xxx	True	True	True

Details of action-code **7X.XXX**:
Warning: action code 7 is only active for Desktop type robots for which the "flow limitation" option has been installed.
This code works in a similar way to code 4 "place action". It works simultaneously on 3 dedicated outputs (0=vacuum, 1=blowing, 2=flow limiter) to create a two-stage blowing process.

The sequence is as follows:

1) the vacuum is deactivated and blowing triggered simultaneously.

2) after the given time frame, the flow limiter is triggered.

7.2.3.2. Examples

Code	Effect
121.001	sets output 2 to `true` when the robot is 1 mm from its target position
130.00001	sets output 3 to `false` when the robot is 10µm from its target position
211.001	sets output 1 to `true` when the robot is 1 mm from its target position. After 100 ms, output 1 is inverted (set to `false`)
230.00001	sets output 3 to `false` when the robot is 10µm from its target position. After 100 ms, output 3 is inverted (set to `true`)
300.001	sets output 0 to `true` when the robot is 1 mm from its target position
311.001	sets output 0 to `true` when the robot is 1 mm from its target position
400.030	sets output 0 to `false` and output 1 to `true` when the robot has reached its target position. After 30 ms, output 1 is inverted (set to `false`)
451.150	sets output 0 to `false` and output 1 to `true` when the robot has reached its target position. After 150 ms, output 1 is inverted (set to `false`)
510.010	after movement has begun, the robot controls whether the state of output 1 is `false`. Otherwise, the movement is stopped.
521.100	after movement has begun, the robot controls whether the state of output 2 is `true`. Otherwise, the movement is stopped.
600.001	sets outputs 0, 1 and 2 to `false` when the robot is 1 mm from its target position
607.00001	sets outputs 0, 1 and 2 to `true` when the robot is 10 µm from its target position

7.2.4. "Robot" module instructions

void ALARMMESSAGE *'sUserDefineMessage'*

Syntax: **ALARMMESSAGE** *'sUserDefineMessage'*

Function: This instruction generates a "warning" type information message which is added to the list of errors.

> **NOTE:**
> This instruction has no influence on the execution of the program.
> A message is generated but the program continues to run normally after this.

Parameters: *sUserDefineMessage* Message to be displayed

Example:

```
(* Warning if the robot is faulty *)
State:= GetState;
IF State = 0 THEN
  AlarmMessage 'Warning: the robot is faulty!';
END_IF
```

See also:

bool **DIN** *rInput*

Syntax: **DIN** <rInput>

Function: This instruction reads and sends the value of a specified digital input **nInput**.

Parameters: *nInput* numerical value referencing the input number read.

Example:

```
(* read the state of input 0 *)
Var:= DIN 0;
```

See also: DOUT

bool **DOUT** *rOutput bValue*

Syntax: **DOUT** <rOutput> <bValue>

Function: This instruction sets the value **bValue** for a specified digital output **nOutput**.

Parameters:

nOutput numerical value referencing the number of the output set.

bValue boolean corresponding to the value of the output set.

Example:

```
(* set output 3 to false *)
DOUT 3 false;
```

See also: DIN

vector **GETFRAMEPARAMS** *nFrameID sParamType nParamID*

Syntax: **GETFRAMEPARAMS**

Function: This instruction reads and sends the value for a particular parameter of a frame. The value sent depends on the type of parameter requested with sParamType.

NOTE:

This instruction can only currently be used with a type 3 frame = "meshFrame"

Parameters:

nFrameID Frame identifier (ID).

sParamsType Character string determining the type of parameter desired.
The only possible value is: "mesh". Which corresponds to the mesh of a type 3 frame = "meshFrame"

nParamID Mesh source "point" identifier

Example:

```
(* obtain "source" coordinate no 123 for frame 11 *)
Src:=GEFRAMEPARAMS 11 'mesh' 123;
```

See also: SETFRAMEPARAMS

vector **SETFRAMEPARAMS** *nFrameID sParamType vParam*

Syntax: **SETFRAMEPARAMS**

Function: This instruction enables the value of one of the parameters of a frame to be modified.

NOTE:

Warning: the value is only modified in the memory. To save the value to the disk, refer to the SAVEFRAME instruction.

Parameters:

nFrameID Frame identifier (ID).

sParamsType Character string determining the type of parameter desired.
The possible values are:

#	Value	Significance
1	"config"	"ptConfig" type parameter
2	"calib"	"ptCalib" type parameter
3	"scale"	"xdw, ydw" type parameter for type 1 frames.
4	"mesh"	"dst" type parameter (destination position for a node belonging to a mesh of a type 3 frame "meshFrame")

vParam Parameter to be modified. May be a value (real) or a vector depending on the type of parameter modified.

Example:

```
(* modify the value of ptcalib no 1 of frame 11 *)
SEFRAMEPARAMS 11 'calib' (1,2,3,4);
```

See also: GETFRAMEPARAMS**,** SAVEFRAME

real GETFRAMEPARAMSNUMBER *nFrameID sParamType*

Syntax: **GETFRAMEPARAMSNUMBER**

Function: This instruction sends the total number of parameters belonging to the frame. The value depends on the type of parameter requested with sParamType.

NOTE:

This instruction can only currently be used with a type 3 frame = "meshFrame"

Parameters:

nFrameID Frame identifier (ID).

sParamsType Character string determining the type of parameter desired. **The only possible value is: "mesh".** Which corresponds to the mesh of a type 3 frame = "meshFrame"

Example:

```
(* get the total number of mesh nodes for frame 11 *)
TotalNodeNumber:=GETFRAMEPARAMSNUMBER 11 'mesh';
```

See also: *SETFRAMEPARAMS*

real GETNEXTFRAMEPARAMSINDEX *nFrameID sParamType*

Syntax: **GETNEXTFRAMEPARAMSINDEX**

Function: This instruction moves the collection index to the next valid element. Only applicable with a type 3 frame = "meshFrame". The current implementation results in the next node being searched for, for which the attribute "teached" = false.

NOTE:

This instruction can only currently be used with a type 3 frame = "meshFrame"

Parameters:

nFrameID Frame identifier (ID).

sParamsType Character string determining the type of parameter desired. **The only possible value is: "mesh".** Which corresponds to the mesh of a type 3 frame = "meshFrame"

Example:

```
(* move the index to the next element for frame 11 *)
idxValue:=GETNEXTFRAMEPARAMSINDEX 11 'mesh';
```

See also: *SETFRAMEPARAMSINDEX*

*void **SETFRAMEPARAMSINDEX** nFrameID nIndexValue*

Syntax: **SETFRAMEPARAMSINDEX**

Function: This instruction is used to set the current value for the collection index.

NOTE:

This instruction can only currently be used with a type 3 frame = "meshFrame"

Parameters:

nFrameID Frame identifier (ID).

sParamsType Character string determining the type of parameter desired. **The only possible value is: "mesh".** Which corresponds to the mesh of a type 3 frame = "meshFrame"

Example:

```
(* reset the index of frame 11 *)
SETFRAMEPARAMSINDEX 11 'mesh' 0;
```

See also: *GETFRAMEPARAMSINDEX*

vector **GETPOINT** *nPointID*

Syntax: **GETPOINT**

Function: This instruction reads and sends the coordinates for a point defined by its identifier.

Parameters: *nPointID* numerical value referencing the identifier (ID) of the point for which the coordinates are desired.

Example:

```
(* get the coordinates of point 5 *)
pos:=GETPOINT 5;
```

See also: SETPOINT

vector **GETPOSITION** *frameID toolID*

Syntax: **GETPOSITION**

Function: This instruction reads the current position of the robot in the global reference system (frame 0).

Parameters: *nFrameID* numerical value referencing the identifier (ID) of the frame in which we want to read the coordinates. The parameter is optional; the default value is 0 (frame world).

nToolID numerical value referencing the identifier (ID) of the tool with which we want to read the coordinates. The parameter is optional. There are no default values; the tool correction is not applied if the parameter is absent.

Example:

```
(* set the current position in frame 1 with tool 2 in the variable pos *)
pos:=GETPOSITION 1 2;
```

See also:

real **GETTIME**

Syntax: **GETTIME**

Function: This instruction reads and sends the current time in milliseconds [ms]

Parameters: none

Example:

```
(* calculate the duration of a pick - place cycle *)

Tstart:= GETTIME;

(* pick - place loop *)

Tend:= GETTIME;
Ttotal:= Tend - Tstart;
```

See also:

real GETSTATE

Syntax: **GETSTATE**

Function: This instruction reads and return the state of the robot:

Ref	State	Description
-2	Alarm	The emergency stop button is triggered. NOTE: When the button is released, the robot switches to the "**error**" state.
-1	Error	An error has occurred on the robot; The possible causes are: - Movement too fast - Collision with an object NOTE: if an error occurs, the robot must be put in the "**off**" state in order to continue.
0	Off	In this state, the robot is "off" but ready to start
1	Initialisation	The robot is in an initialisation phase
2	On	The robot is initialised and its motors are regulated. But in this status, movements are not yet possible as the "path planner" and "task planner" are not in operation.
3	Idle	The robot is ready to move.

Parameters: *nOutput* numerical value referencing the number of the output set

Example:

```
(* initialise the robot *)

state:=getstate;
IF state<3 Then
      SETSTATE 0;
      WHILE state<>0 DO
      state:=getstate;
      sleep 20;
      END_WHILE

      SETSTATE 3;
      state:=getstate;
      WHILE state<>3 DO
            state:=getstate;
            sleep 20;
      END_WHILE
END_IF
```

See also: SETSTATE

bool **HOLDIFREQUESTED**

Syntax: **HOLDIFREQUESTED**

Function: This instruction triggers the OMAC "Holding" state, only when a pause request is in progress. ("pause" button pressed)
In this case it sends the "true" value.
If no request is in progress, the instruction has no effect and sends the "false" value

NOTE:

Parameters: none

Example:

```
(* Message *)
Pause:= false;
Pause:= HoldIfRequested;
IF pause = TRUE THEN
      AlarmMessage 'The program resumes after being paused!';
ELSE
      AlarmMessage 'The program has not been interrupted…';
END_IF
```

value **LOADDATA** *'sNameToDisplay'*

Syntax: **LOADDATA** *'sDataName'*

Function: This instruction enables data contained in the "data" part of an ARL program to be loaded into a variable.

NOTE:
This instruction can only be used in ARL programs from the HMI of an MFEED or AFEED module. It returns the value of the variable recorded in the table of dynamic values.

Parameters: *sDataName* Name of the resource

Example:

```
(* move to the pick position with an adjustable speed *)
pickPosition:= (0,0,0,0);
speed:= 'Pick_speed' loadValue;
moveto  pickPosition 1 1 speed;
```

See also:

void **SAVEFRAME** *nFrameID*

Syntax: **SAVEFRAME** <*nFrameID*>

Function: This instruction saves all of the parameters for the designated frame to the disk.

Parameters: *nFrameID* frame identifier.

Example:

```
(* Modification and saving of ptconfig no 0 of frame 11 *)
SETFRAMEPARAMS 11 'config" 0 (1,2,3,4);
SAVEFRAME 11; // optional, save the frame to the disk
```

See also: *SETFRAMEPARAMS*

vector **TRANSFORMPOINT***vPosition posFrameId transFrameId*

Syntax: **TRANSFORMPOINT** <vPosition> <nPosFrameId> <nTransFrameId>

Function: This instruction enables the coordinates of a point of a frame to be transformed to another.

Parameters:

vPosition	position to be transformed
nPosFrameId	Id of the frame of the position to be transformed (vPosition)
transFrameId	Id of the destination frame

Example:

```
(* transform the position (1,2,3,4) of frame 11 to frame 22 *)
pos:= TRANSFORMPOINT (1,2,3,4) 11 22;
```

See also: TRANSFORMVECTOR

vector **TRANSFORMVECTOR** *vPosition posFrameId transFrameId*

Syntax: **TRANSFORMVECTOR** <vPosition> <nPosFrameId> <nTransFrameId>

Function: This instruction is used to transform a vector $\vec{V}$ = (x,y,z) from one frame to another. This means that the origin $\vec{O}$ = (0,0,0,0) of the starting frame reference is also transformed in the incoming frame and it is the subtraction $\vec{V} - \vec{O}$ that is sent by the instruction.

Parameters:

vPosition	vector to be transformed
nPosFrameId	Id of the frame of the position to be transformed (vPosition)
transFrameId	Id of the destination frame

Example:

```
(* transform the position (1,2,3,4) of frame 11 to frame 22 *)
pos:= TRANSFORMVECTOR (1,2,3,4) 11 22;
```

See also: TRANSFORMPOINT

void **MOVEMS** *vPos rMotionSequenceID*

Syntax: **MOVEMS** *<vPos> <rMotionSequenceID>*

Function: This instruction causes the robot to move to the **vPosition** point by executing the movements programmed in the **nMotionID** movement sequence beforehand.

NOTE:
The robot must be in the "idle" state to accept movement commands.

Parameters:

vPos	vector representing the coordinates of the final target position. in which frame??
nMSID	numerical value referencing the identifier of the movement sequence to be performed.

Example:

```
(* perform a movement sequence *)

desPos:= (0.02,0.012,0.02,0);
MotionID:= 1;
MOVEMS desPos MotionID;
```

See also:

void **MOVETO** *vPos rFrameID rToolID rSpeedFactor rBlend rActionCode bRZtarget rRZspeedFactor*

Syntax: **MOVETO** *<vPos> <rFrameID> <rToolID> <rSpeedFactor> <rBlend> <rActionCode> <bRZtarget> <rRZspeedFactor>*

Function: This instruction causes the robot to move to the **vPos** point in the **nFrameID** reference with the **nToolID** tool.

NOTE:
The robot must be in the "idle" state to accept movement commands.

Parameters:

vPos — vector representing the coordinates of the final target position.

rFrameID — numerical value referencing the identifier for the frame in which to interpret the **vPos** vector.

rToolID — numerical value of the identifier for the tool loaded on the robot.

rSpeedFactor — dynamic multiplication factor between 0 and 1.

$$Speed_{desired} = Speed_{max} * \boldsymbol{nSpeedFactor}$$

rBlend — Optional parameter.
Numerical value in metres [m] of the blend curve radius.

rActionCode — Optional parameter
Numerical value representing the IO code to be performed.

bRZtarget — Optional parameter. ! *[Valid only for the robot type Desktop with a 4th axis working with the "end to end" mode.]*
Boolean value indicating if the 4th axis will be synchronized or not with that movement final position.

rRZSpeedratio — Optional parameter. ! *[Valid only for the robot type Desktop with a 4th axis working with the "end to end" mode.]*
Numerical value in the range [0..1] representing the dynamic traveling time factor for the 4th axis speed.
4thAxis travel time = MaxSpeedTime * rRZSpeedRatio.

Example:

```
(* perform a movement sequence *)
Pick:= GETPOINT 5PickHeight:= Pick + (0,0,0.0001,0);
FrameID:= 3;
ToolID:= 1;
HighSpeed:= 0.7;
RZSynchro := True;
RzSpeed   := 0.8;
Vacuum:= 300.050;
MOVETO PickHeight FrameID ToolID HighSpeed;
MOVETO Pick FrameID ToolID HighSpeed 0 Vacuum;
MOVETO Pick FrameID ToolID HighSpeed 0 Vacuum RZSynchro RzSpeed;
MOVETO PickHeight FrameID ToolID HighSpeed;
```

See also:

void **SETPOINTPARAMS** *rPointID vPoint*

Syntax: **SETPOINTPARAMS** <*nPointID*> <*vPoint*>

Function: This instruction sets the coordinates for a point referenced by its **nPointID** identifier at a specified value.

Parameters:

rPointID	numerical value referencing the identifier (ID) of the point for which the coordinates are to be set.
vPoint	vector of coordinates to be set in point **nPointID**

Example:

```
(* store the value of point1 in ID point n°1 *)
point1:= (0.03,0.02,0.01,0);
pointID:= 1;
SETPOINTPARAMS pointID point1;
```

See also: GETPOINT

void **SETSLOWSPEED** *bSlowSpeed*

Syntax: **SETSLOWSPEED** <*bSlowSpeed*>

Function: This instruction enables the robot to be set at slow speed.

Parameters: *bSlowSpeed* boolean indicating whether the robot should move at slow speed:

- True: movement at slow speed
- False: movement at programmed speed

Example:

```
(* set the robot at slow speed *)
SETSLOWSPEED true;
```

See also:

void **SETSTATE** *rState*

Syntax:	**SETSTATE <*rState*>**	
Function:	This instruction change the robot state	
Parameters:	*rState*	numerical value indicating the desired state of the robot:0 for "off", 2 for "on" and 3 for "idle"
Example:	see below	
See also:	GETSTATE	

```
(* initialise the robot *)

state:=getstate;
IF state<3 Then
      SETSTATE 0;
      WHILE state<>0 DO
      state:=getstate;
      END_WHILE
      SETSTATE 3;
      state:=getstate;
      WHILE state<>3 DO
            state:=getstate;
      END_WHILE
END_IF
```

void **SLEEP** *rTime*

Syntax:	**SLEEP** *<rTime>*	
Function:	This instruction is used to pause the program being executed for a certain duration in milliseconds [ms]	
Parameters:	*rTime*	Numerical value indicating a waiting time in [ms]

Example:	`(*trigger output 0, 200 ms after movement has begun*)`
	
	`MOVETO PickHeight FrameID ToolID HighSpeed NoBlend NoAction;`
	`SLEEP 200;`
	`DOUT 0 true;`	

See also:	WAITFORMOTIONEND	

void **STOP**

Syntax: **STOP**

Function: This instruction enables any robot movement in progress to be stopped.

Parameters: *none*

Example:

```
(* stop the program according to the value of a variable*)
IF var=5 THEN
MOVETO Trash FrameID ToolID HighSpeed NoBlend PlaceAction;
STOP;
END_IF;
```

See also:

void **WAITFORMOTIONEND**

Syntax: **WAITFORMOTIONEND**

Function: This instruction is used to pause the execution of a task until all of the tasks in progress have finished.

Parameters: *none*

Example:

```
(* wait one second between two movements *)

MOVETO PickHeight FrameID ToolID HighSpeed NoBlend NoAction;
WaitForMotionEnd;
SLEEP 1000;
MOVETO Pick FrameID ToolID HighSpeed NoBlend Vacuum;
```

See also: SLEEP

void ModuleCmd 'Robot' *'SetEnable value=<true|false> [timeout=<delay>] [settling=<delay>]'*

<table>
<tr><td>Syntax:</td><td colspan="2">SetEnable value=<true|false> [timeout=<delay>] [settling=<delay>]</td></tr>
<tr><td>Function:</td><td colspan="2">This instruction set the robot state to "IDLE" or "OFF".
This is meant to shorten the typical sequence of commands used to "initialize" the robot. It handles internally error clearing and timeout exception if the target state is not reached after a certain limit of time.</td></tr>
<tr><td>Parameters:</td><td>value</td><td>True = Clear error and block until state = "IDLE"
False = block until state = "OFF"</td></tr>
<tr><td></td><td>timeout</td><td>Optional parameter that specifies the maximum delay allowed for the state to be reached. Unit: [ms]. Default value = 6000[ms]</td></tr>
<tr><td></td><td>Settling</td><td>Warning:
This is an advance optional parameter. Do not modify that value unless specified by Asyril.
Optional parameter that specify the "settling time" applying to state change. This represent the minimum delay before a robot state is considered stable.
Unit: [ms]. Default value = 50[ms]</td></tr>
<tr><td>Example:</td><td colspan="2"><code>(* Set the robot state to IDLE *)</code>
<code>ModuleCmd 'Robot' 'SetEnable value=true';</code></td></tr>
<tr><td>See also:</td><td colspan="2">SETSTATE</td></tr>
</table>

7.1. "Asyview" module instructions

The Asyview module is the component that manages the communication with the Asyril's vision system SmartSight. (Asyview is the software controller that runs on a SmartSight)

The commands detailed here after describe the syntax of a subset of the SmartSight functions. These are the commands used in a typical pick and place application. For more information see the "SmartSight programming manual".

void ModuleCmd 'AsyView' '<modulePath> **START**'

Syntax: <modulePath> **START**

Function: This instruction starts the related module.

Parameters: *modulePath* Path of an Asyview module.

Example: This instruction starts the first module of the first cell.

```
ModuleCmd 'Asyview' 'A[0]/C[0]/M[0] Start';
```

See also: STOP

void ModuleCmd 'AsyView' '<modulePath> **SETPARAMETER name=WORKINGMODE WorkingMode**=<Wm>'

Syntax: <modulePath> **SetParameter WORKGINMODE workingMode=**< active | passive >

Function: This instruction sets the working mode of a module

Parameters: *modulePath* Path of an Asyview module.

workingMode Working mode keyword = "active" or "passive".

Example: This instruction set the first module to the "active" working mode

```
ModuleCmd 'Asyview' 'A[0]/C[0]/M[0] SetParameter
name=WorkingMode WorkingMode=active';
```

See also: FIELDOFVIEW

void ModuleCmd 'AsyView' '<modulePath> **STOP**"

Syntax: <modulePath> **STOP**

Function: This instruction stops the related module.

Parameters: *modulePath* Path of an Asyview module.

Example: This instruction starts the first module of the first cell.

```
ModuleCmd 'Asyview' 'A[0]/C[0]/M[0] Stop';
```

See also: START

void ModuleCmd ‘AsyView‘ ‘<imgAcqPath> SETPARAMETER name=FIELDOFVIEW imageConfigurationName=<ImgConfigNm> locked=<True|False>‘

Syntax: <imgAcqPath> **SETPARAMETER name=FIELDOFVIEW ImageConfigurationName=<ImgConfigNm> locked=**<True|False>

Function: This instruction locks or unlocks the field of view (FieldOfView) of the vision system When the field of view is locked, it prevents image acquisition. It is possible to lock/unlock the field of view of each device separately.

Parameters:

imgAcqPath	Path of an Asyview image acquisition manager.
ImgConfigNm	
locked	Boolean argument. “true” to lock. “false” to unlock.

Example: This instruction locks the field of view for the image of the first module. This means that it will not be possible to acquire an image while the field of view remains locked.

```
ModuleCmd 'Asyview' 'A[0]/C[0]/M[0]/I[0] SetParameter
name=FieldOfView ImageConfigurationName=default Locked=true';
```

See also: GETRESULT

vector ModuleCmd ‘AsyView‘ ‘<modulePath> GETRESULT ModelName=<modelID>‘

Syntax: <modulePath> **GETRESULT** ModelName=<modelID>

Function: This instruction triggers a position search on a particular module. The response is instantaneous if a position is already available. Otherwise, the instruction blocks until a position is found.

Parameters:

modulePath	Path of an Asyview module.
(modelID)	Optional: Identifier of the part model (it comes from the teaching interface). **Note**: *modelId* needs to be a number.

Feedback: This function returns a vector of six coordinates: For compatibility reason with the previous versions, some vector coordinate may be set to zero.

```
(posX, posY, 0, posTheta, modelID, positionID)
```

Example: A position search is requested on the first module. When a position is available, the vector of six coordinates corresponding to it is stored in the "`pickPos`" variable

```
pickPos:= ModuleCmd 'Asyview' 'A[0]/C[0]/M[0] GetResult';
```

```
pickPos:= ModuleCmd 'Asyview' 'A[0]/C[0]/M[0] GetResult
ModelName=3';
```

See also: REMOVERESULT

void ModuleCmd ‘AsyView‘ ‘**REMOVERESULT** NextModelName=<modelID>**’** *vPosition*

Syntax: **REMOVERESULT** NextModelName=<modelID> <vPosition>

Function: Requires one particular position to be removed from the Asyview's position buffer. The position to be removed must be given by a variable containing a vector returned by the GetResult instruction.

Parameters:

(modelID) Optional: Identifier of the part model (it comes from the teaching interface).

vPosition Vector with six coordinates returned by the "getResult" command `(posX, posY, 0, posTheta, modelID, positionID)` There is no need to specify a module path because it is embedded into the position vector returned by the getResult command. That path and the positionID are enough to unequivocally determine which position must be removed from the buffer.

Example: This instruction removes the "`pickPos`" vector from the position buffer.

```
ModuleCmd 'Asyview' 'RemoveResult' pickPos;
ModuleCmd 'Asyview' 'RemoveResult NextModelName=3' pickPos;
```

See also: GETRESULT

void ModuleCmd ‘AsyView‘ ‘<imgAcqPath> **ACQUIRE**‘

Syntax: <imgAcqPath> **ACQUIRE**

Function: Triggers the camera to order an image capture

Parameters: *imgAcqPath* Path of the image acquisition manager

Example: This instruction triggers an image acquisition on the first module

```
ModuleCmd 'AsyView' 'A[0]/C[0]/M[0]/I[0] Acquire';
```

See also: PROCESSMANAGERSTATE

String ModuleCmd 'AsyView' '<modulePath> **GETPARAMETER** name=PROCESSMANAGERSTATE modelName=<*modelID*>'

Syntax: **<**modulePath**> GETPARAMETER name=PROCESSMANAGERSTATE** modelName=<PartId>

Function: This instruction reads the state of the process manager

Parameters:

modulePath	Path of an asyview module.
modelID	Identifier of the part model (it comes from the teaching interface).

Feedback: The state keyword of the process manager as a string:

The list of possible keyword is listed here after. See the SmartSight documentation for the signification of each state.

"*Unknown*", "*Error*","*Unconfigured*","*IDLE*","*Starting*","*Running*"
"*Aborting*","*Stopping*","*Stopped*","*Resetting*","*Loading*","*Saving*"
"*Calibrating*","*Teaching*","*Configuring*","*Mixed*","*Reading*","*Writing*",
"*Clearing*","*Acquiring*","*NotDefined*","*Preparing*",
"*OpeningTeaching*","*ClosingTeaching*"

Example: This series of instructions is used to acquire an image and to wait until the image has been analysed before continuing the rest of the program

```
ModuleCmd 'AsyView' 'A[0]/C[0]/M[0]/I[0] Acquire';
processMgmtState:= 'running';
WHILE processMgmtState = 'running' DO
      processMgmtState:= ModuleCmd 'Asyview' 'A[0]/C[0]/M[0]
      GetParameter name=ProcessManagerState modelName=3';
      Sleep 100;
END_WHILE
```

See also: ACQUIRE

real ModuleCmd ‘AsyView‘ ‘ <modulePath> **GETPARAMETER name=AVAILABLERESULTS** imageConfigurationName=<ImgConfigNm>’

Syntax: <modulePath> **GETPARAMETER name=AVAILABLERESULTS** imageConfigurationName=<ImgConfigNm>

Function: This instruction counts the number of positions available (i.e. the parts accepted and not yet deleted) in the buffer of the device requested.

Parameters:

modulePath	Path of an Asyview module.
ImgConfigNm	Identifier of the image configuration.

Feedback: Number of positions still available in the buffer

Example: This program example is used to perform a specific series of instructions if only one position remains in the buffer.

```
PositionNbr:=ModuleCmd 'AsyView''A[0]/C[0]/M[0] GetParameter
name=AvailableResults imageConfigurationName=Default';
IF PositionNbr=1 THEN
(* … *)
END_IF
```

See also: ACQUIRE

obj ModuleCmd ‘AsyView‘ ‘ <modulePath> **GETPARAMETER name=<parameterName>**’ resultName

Syntax: <modulePath> **GETPARAMETER** name=<parameterName> resultName

Function: This generic instruction can be used to get the return value of any parameter.

Parameters:

modulePath	Path of an Asyview module.
parameterName	Name of the requested parameter.
resultName	Name of the String to look for in the response.

Feedback: Result value (it can be a String, a vector, a number, etc.).

Example: This instruction asks for the quantity of parts on the feeder (feeding information).

```
quantity:=ModuleCmd 'AsyView' 'A[0]/C[0]/M[0] GetParameter
name=PartsOnFeeder' 'PartsOnFeeder';
```

See also: other GETPARAMETER commands

void **ModuleCmd 'AsyView' '<modulePath> UNCALIBRATE imageConfigurationName=<ImgConfigNm>'**

Syntax: <modulePath> **UNCALIBRATE** imageConfigurationName=<ImgConfigNm>

Function: This command reset the coordinate transformation associated with the current calibration. It sets an identity transformation.

Parameters:

modulePath	Path of an asyview module.
ImgConfigNm	Identifier of the image configuration.

Example: This command resets the calibration data of the first module

```
ModuleCmd 'asyview' 'A[0]/C[0]/M[0] Uncalibrate
imageConfigurationName=Default';
```

See also: ADDPOINTPAIR

void ModuleCmd ‘AsyView‘ ‘<modulePath> ADDPOINTPAIR imageConfigurationName=<ImgConfigNm>‘ _vPosition rPosition_

Syntax: <modulePath> **ADDPOINTPAIR** imageConfigurationName=<ImgConfigNm> <vPosition> <rPosition>

Function: This instruction sets the correspondence between a vision and a robot position for calibration purpose.

Parameters:

modulePath	Path of an asyview module.
ImgConfigNm	Identifier of the image configuration.
vPosition	Vision position as a vector of at least 2 dimension `(VisXPos, VisYPos, … )`
rPosition	Robot position as a vector of at least 2 dimensions `(RobotXPos, RobotYPos, … )`

Example: This sequence of instructions firstly reads the current vision result and then set the correspondence with the robot position = (1.0,1.0,0.0,0.0)

```
VisionPos := ModuleCmd 'AsyView' 'A[0]/C[0]/M[0] GetResult';
ModuleCmd 'AsyView' A[0]/C[0]/M[0] AddPointPair
imageConfigurationName=Default' VisionPos (1.0,1.0,0.0,0.0);
```

See also: CALIBRATE

void ModuleCmd ‘AsyView‘ ‘<modulePath> **CALIBRATE** imageConfigurationName=<ImgConfigNm>‘

Syntax: <modulePath> **CALIBRATE** imageConfigurationName=<ImgConfigNm>

Function: This command calibrates by computing the coordinate transformation associated with the current calibration data.

Parameters:

modulePath	Path of an asyview module.
ImgConfigNm	Identifier of the image configuration.

Example: This command initiates a calibration process on the first module

```
ModuleCmd 'asyview' 'A[0]/C[0]/M[0] Calibrate
imageConfigurationName=Default';
```

See also: SAVECALIBRATION

void ModuleCmd ‘AsyView‘ ‘<modulePath> **SAVECALIBRATION** imageConfigurationName=<ImgConfigNm>‘

Syntax: <modulePath> **SAVECALIBRATION** imageConfigurationName=<ImgConfigNm>

Function: This command save the calibration data into the disk

Parameters:

modulePath	Path of an asyview module.
ImgConfigNm	Identifier of the image configuration.

Example: This command saves the calibration data of the first module

```
ModuleCmd 'asyview' 'A[0]/C[0]/M[0] SaveCalibration
imageConfigurationName=Default';
```

See also: CALIBRATE

void ModuleCmd ‘AsyView‘ ‘<modulePath> **<FUNCTION>** arguments‘

Syntax: <modulePath> **<FUNCTION>** <arguments>

Function: This generic command can be used to send almost any instruction to the Asyview

Parameters:

modulePath	Path of an asyview module.
FUNCTION	Generic Asyview instruction (refer to the specific documentation).
arguments	Generic optional arguments of the function.

Example: This command saves the latest images in BMP format in the D:\AsyrilData folder

```
ModuleCmd 'asyview' 'A[0] SaveLatestImages
folderpath=D&#58\AsyrilData:format=BMP';
```

7.2. I/O extension module instructions

The I/O extension module is an optional item that helps integrating the Asyfeed Module into a machine by increasing the number of digital I/O and having the possibility to use analog I/O. Inside the operating manual and inside the electrical diagram given with the module, more information can be found on how to interface the module and how to wire it.

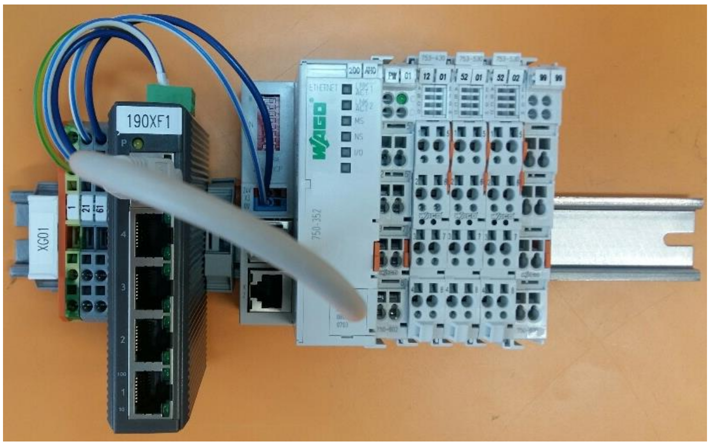


Figure 7-6 – Example of I/O Extension module

Each I/O terminal has an identifier given by the electrical diagram. For each type of terminal, the first two digits will give the type and the last two are incremental for each terminal of a given type.

In order to be able to communicate with the system, the name of the contact must be given. The string identifying the contact of one terminal will start with 200AH, then add the terminal identifier and finish with _XX1 where XX will be DI, DO, AI or AO depending of the type of the terminal and the number will be the position of the contact inside the terminal, see Figure 7-7 and the table below for examples.

Table 2 - Example of IO name for ARL

Terminal Type	*Label on I/O extension module*	*IO name for ARL*
8DI PNP	200AH1201 Input N°1	200AH1201_DI1
8DO PNP	200AH5201 Output N°8	200AH5201_DO8
2AI 4-20mA	200AH2201 Input n°1	200AH2201_AI1
2AO 4-20mA	200AH6201 Output n°2	200AH2201_AO2

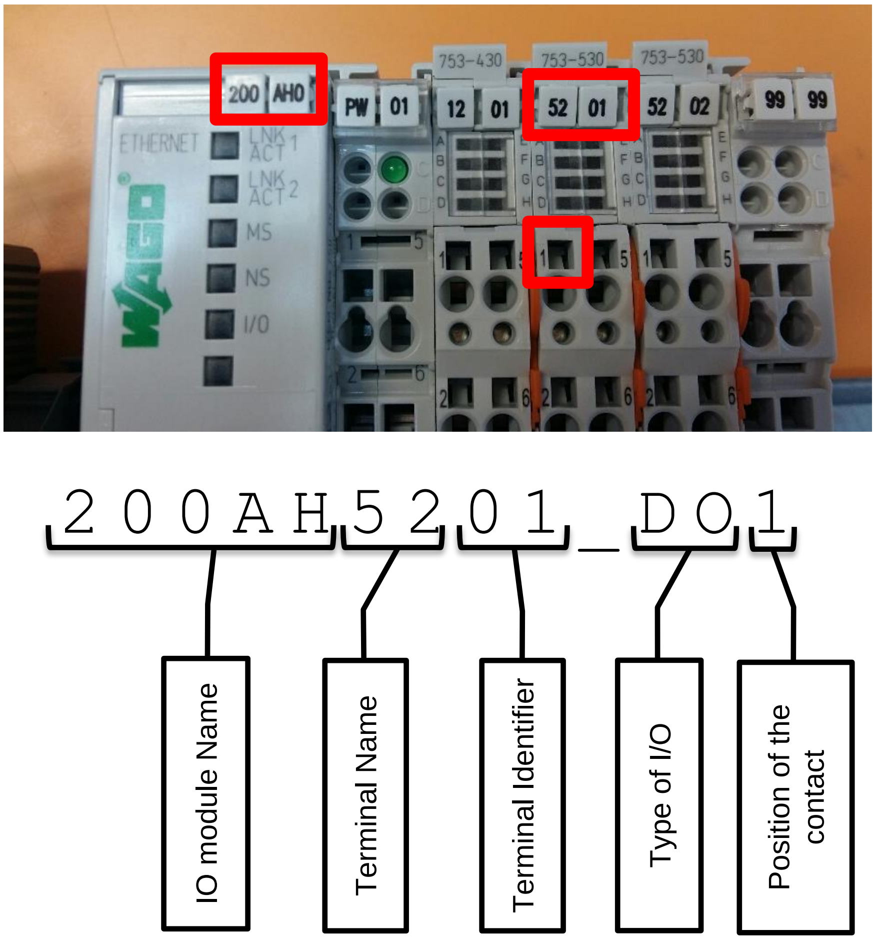


Figure 7-7 - Representation of the contact 200AH5201_DO1

7.2.1. Digital I/O

bool ModuleCmd 'iomodule' 'cmd=**READ** name=*sDigitalInput*'

Syntax: <cmd=**READ**> <name = sDigitalInput>

Function: This instruction reads and sends the state of a digital input identified by its **sDigitalInput** name.

> **NOTE:**
> The same command also enables the state of a digital input to be read.

Parameters: *sDigitalInput* character string identifying the name of the digital input.

Feedback: boolean corresponding to the state of the input at the time of interrogation.

Example: This instruction reads the state of the door sensor digital input which is wired to the input 1 of the card 1201 :

```
DoorsOpen:= ModuleCmd 'IOMODULE' 'cmd=Read name=200AH1201_DI1;
```

See also: WRITE

void **ModuleCmd 'iomodule' 'cmd=WRITE name= sDigitalOutput value=*bValue*'**

Syntax: <cmd=**WRITE**> <name= sDigitalOutput> <value= bValue>

Function: This instruction sets the state of a digital output identified by its **sDigitalOutput** name at a chosen **bValue** value.

Parameters:

sDigitalOutput character string identifying the name of the digital output.

bValue boolean corresponding to the desired state of the output.

Example: If we want to enable and disable the ionizer which is wired to the output 8 of the card 5201 :

```
(* enable the ionizer *)

ModuleCmd 'IOMODULE' 'cmd=Write name=200AH5201_DO8 value=true;

(* disable the ionizer *)

ModuleCmd 'IOMODULE' 'cmd=Write name=200AH5201_DO8 value=false;
```

See also: READ

7.2.2. Analog I/O

The commands used for the analog I/O are the same as the ones for the digital, but the type of the data will differ.

real **ModuleCmd 'iomodule' 'cmd=READ name=*sAnalogInput*'**

Syntax: <cmd=**READ**> <name = *sAnalogInput* >

Function: This instruction reads and sends the state of an analog input identified by its **sAnalogInput** name.

NOTE:

The same command also enables the state of an analog input to be read.

Parameters:

sAnalogInput character string identifying the name of the analog input.

Feedback: Real number corresponding to the state of the input at the time of interrogation.

This number is given in raw value and must be converted into interpretable value by using the following formula :

$$\frac{raw_{value}}{2^{15}} * range + offset$$

Example: This instruction reads the state of the analog pressure sensor connected to the input 1 of the card 2201 (2AI 4-20mA) :

```
(* Get the raw value from sensor*)

sensor_raw:= ModuleCmd 'IOMODULE' 'cmd=Read name=200AH2201_AI1;

(* Convert this value into mA*)

sensor_mA:= (PSensor/32768)*16+4;
```

See also: WRITE

void ModuleCmd ‘iomodule‘ ‘cmd=**WRITE** name= *sAnalogOutput* value=*bValue*‘

Syntax: <cmd=**WRITE**> <name= *sAnalogOutput* > <value= *bValue*>

Function: This instruction sets the state of an analog output identified by its **sAnalogOutput** name at a chosen **bValue** value.

Parameters:

sAnalogOutput	character string identifying the name of the analog output.
bValue	real number corresponding to the desired state of the output. The value is given in mA or V depending on the type of the output

Example: If we want to inject 10.4mA to a device connected to the output 2 of the card 6201 (2AO 4-20mA) :

```
ModuleCmd 'IOMODULE' 'cmd=Write name=200AH6201_AO2 value=10.4;
```

See also: READ

7.3. Turning table module instructions

void **ModuleCmd 'TurningTable' 'cmd=STARTHOMING blocking=*bBlock* homingOffset=*rAngle*'**

Syntax: <cmd=**STARTHOMING**> <blocking= bBlock> <homingOffset= rAngle>

Function: This instruction launches an initialisation sequence for the turning table.

Parameters:

bBlock optional boolean type argument, defining whether the execution of the rest of the ARL program should be blocked during the initialisation phase of the table or not.

rAngle optional number type argument, defining the offset applied in degrees after the reference is found

NOTE:

If this argument is used, the offset value chosen replaces any other value defined in a machine configuration file. (*.arc file)

Example: The example below launches an initialisation sequence for the turning table by preventing execution of the ARL program from continuing until the initialisation sequence is complete.

```
ModuleCmd 'TurningTable' 'cmd=startHoming blocking=false';
```

See also: GETHOMINGSTATUS; MOVE

bool **ModuleCmd 'TurningTable' 'cmd=GETHOMINGSTATUS'**

Syntax: <cmd= **GETHOMINGSTATUS**>

Function: This instruction is sent to establish the initialisation state of the turning table.

Return value: boolean corresponding to the initialisation state at the time of interrogation:

- True: "homing" is complete.
- False: "homing" has begun, but is not yet complete.

Example: The example below loops until the homing is done:

```
Status:=false;
 WHILE hoStatus = FALSE DO
      Status:=ModuleCmd 'TurningTable''cmd=getHomingStatus';
      Dbg 'homing loop...';
END_WHILE
```

See also: STARTHOMING; MOVE

<height> ModuleCmd 'TurningTable' 'cmd=**READHEIGHTSENSOR**'

Syntax: <cmd= **READHEIGHTSENSOR**>

Function: This instruction is sent to the height measurement sensor.

Return value: value in millimetres of the height read by the sensor. (the zero is defined by the ***RESETHEIGHTSENSOR*** command)

Example: The example below reads the value of the height sensor:

```
H1:= 0.0;
H1:= ModuleCmd 'TurningTable' 'cmd=ReadHeightSensor';
IF H1 > 0.5 THEN
(* do what you want *)
END_IF
```

See also: READHEIGHTSENSORSTATE *;* RESETHEIGHTSENSOR

void ModuleCmd 'TurningTable' 'cmd=**ENABLEHEIGHTSENSOR** value=bEnable'

Syntax: <cmd= **ENABLEHEIGHTSENSOR**>

Function: This instruction turn on or off the laser beam of the height sensor.

Parameters: *rEnable* boolean value defining if the laser beam should be active or not.

Example: The example below turns off the height sensor laser beam.

```
ModuleCmd 'TurningTable' 'cmd=EnableHeightSensor value=false;
```

See also: *READHEIGHTSENSOR*

<state> ModuleCmd 'TurningTable' 'cmd= **READHEIGHTSENSORSTATE**'

Syntax: <cmd= **READHEIGHTSENSORSTATE**>

Function: This instruction is sent to establish the state of the height sensor.

Return value: Integer between 0, 1, 2 and 3 corresponding to the state of the sensor:

Ref	State
0	Sensor faulty
1	Normal operation
2	Sensor below lower level (the value is below the minimum value)
3	Sensor exceeds upper level (the value is above the maximum value)

Example: The example below reads the state of the sensor:

```
State:=ModuleCmd'TurningTable''cmd=ReadHeightSensorState';
```

See also: READHEIGHTSENSOR

bool ModuleCmd ‘TurningTable‘ ‘cmd= **RESETHEIGHTSENSOR**‘

Syntax: <cmd= **RESETHEIGHTSENSOR**>

Function: This instruction resets the height sensor in the current state to zero.

The values subsequently measured using command **READHEIGHTSENSOR** will be the heights relative to this zero.

Return value: none

Example: The example below resets the sensor.

```
ModuleCmd 'TurningTable''cmd=ResetHeightSensor ';
```

See also: READHEIGHTSENSOR

void ModuleCmd ‘TurningTable‘ ‘cmd=**MOVE** value=*rAngle* | *sectorNo=iSectorNo*‘

Syntax: <cmd= **MOVE**> <value= rAngle>

Function: This instruction causes the turning table to pivot by a given angle in degrees.

Parameters:

rAngle numerical value defining the relative angle of rotation desired.
The value is interpreted differently depending of the table type.
Table of type "indexed" (stepper motor and mechanical index)
The angle is given as a set point to the stepper motor controller.
Table of type "servo" (servo motor workgin with absolute positioning)
The angle is translated (rounded) into an absolute sector number and added to the current sector value.
The move will always be an integer multiple of the angle between two sectors.

iSectorNo **Apply only for the turning table of type "servo".**
Integer value representing an absolute move to the given sector.
The keyword "current" can be use instead of a numerical value to move to the current sector. (when the table was move manually for ex.)

Example: The example below rotates the table by 45°:

```
ModuleCmd 'TurningTable' 'cmd=Move value=45.0';
```

The example below make an absolute move to the sector no=5.

```
ModuleCmd 'TurningTable' 'cmd=Move sectorNo=5';
```

The example below make an absolute move to the current sector

```
ModuleCmd 'TurningTable' 'cmd=Move sectorNo=Current';
```

See also: GETCURRENTSECTORNO

void ModuleCmd 'TurningTable' 'cmd=**STOP**'

Syntax: <cmd= **STOP**>

Function: This instruction causes the turning table to stop the current movement.

Parameters: *None*

Example: The example below stop the current movement

```
ModuleCmd 'TurningTable' 'cmd=Stop';
```

See also: *MOVE*

bool ModuleCmd 'TurningTable' 'cmd= **READERRORCODE**'

Syntax: <cmd= **READERRORCODE**>

Function: **! Apply only to the table of type "servo"**

This instruction is sent to the wago module and, where applicable, returns a specific error code.

NOTE:

for more information about the meaning of these error codes, please refer to the Wago documentation.

Return value: four-digit integer corresponding to a Wago error.

Example: The example below creates a line in the console during the initialisation phase:

```
ErrorCode:= ModuleCmd 'TurningTable' 'cmd=ReadErrorCode ';
```

See also:

void ModuleCmd 'TurningTable' 'cmd=**ENABLE** value=**bSTATE**'

Syntax: <cmd= **ENABLE**>

Function: **! Apply only to the table of type "servo"**

This instruction turns on or off the position control of the table.

Parameters: *bState* Boolean value representing the control state

True: Turn the control on

False: Turn the control off.

Example: The example below turns the control on

```
ModuleCmd 'TurningTable' 'cmd=Enable value=true';
```

See also: *MOVE*

bool ModuleCmd 'TurningTable' 'cmd= **GETCURRENTSECTORNO**'

Syntax: <cmd= **GETCURRENTSECTORNO**>

Function: **! Apply only to the table of type "servo"**

Return the current sector no.

Parameter None

Return value: The current sector number. Range from [0…<SectorNbr>]

<SectorNbr> The total number of sectors is a configuration setting that must match the actual mechanical design of the table. Due to I/O limitation the maximum manageable sector number is 16.

Example: The example below

```
CurSecNo:= ModuleCmd 'TurningTable' 'cmd=GetCurrentSectorNo';
```

See also: *MOVE*

7.4. MATH module instruction

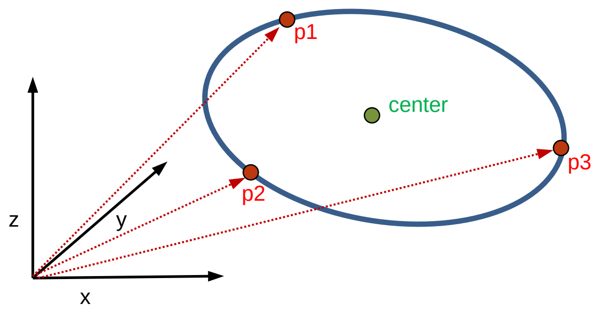
<ret> ModuleCmd 'MATH' 'cmd=CIRCLECENTER func=<FunctionCode>' **bSTATE'**

Syntax: <cmd=**CIRLCECENTER**> <func=FunctionCode>

Function: This instruction is used to calculate the centre of a circle passing through three points.

The centre calculated will be in the plane passing through the three points.

The instruction is used to accumulate the points (vector) into a "collection" in the memory to facilitate future calculations.



NOTE:
The calculation only takes into account the first three points of that collection. The additional points are ignored.

Parameters: *functionCode* Function code. May take the following values:

1) "clearPoints" – Reset the collection of points in the memory.
2) "addPoint" – Add a point to the collection.
3) "computeCenter" – Calculate the centre taking into account the first 3 collection points

Return value: Depends on the function code

Function code	Type of return
clearPoints	<void>
addPoint	<void>
computeCenter	<vector>

Example:

```
Complete example of calculating the centre of a circle
center:= (0,0,0,0);
p1:= (1,2,3,4); p2:= (5,6,7,8); p3:= (9,10,11,12);
ModuleCmd 'MATH' 'cmd=CIRCLECENTER func=clearpoints';
ModuleCmd 'MATH' 'cmd=CIRCLECENTER func=addPoint' p1;
ModuleCmd 'MATH' 'cmd=CIRCLECENTER func=addPoint' p2;
ModuleCmd 'MATH' 'cmd=CIRCLECENTER func=addPoint' p3;
center:= ModuleCmd 'MATH' 'cmd=CIRCLECENTER func=computeCenter';
```

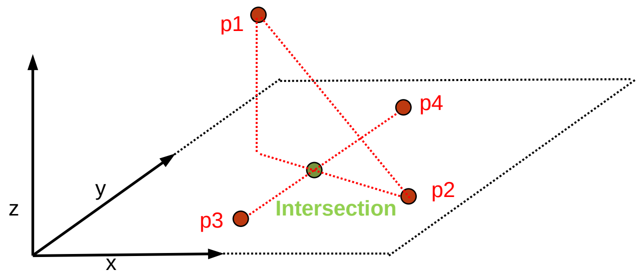
See also: LINEINTERSECT

<ret> ModuleCmd 'MATH' 'cmd=LINEINTERSECT func=<FunctionCode>'

Syntax: <cmd=**LINEINTERSECT**> <func=FunctionCode>

Function: This instruction is used to calculate the point of intersection of two straight lines defined by four points. <u>The calculation is performed in two dimensions along the x-y plane</u>

The instruction is used to accumulate the points (vector) into a "collection" in the memory to facilitate future calculations.



NOTE:
The calculation only takes into account the first four points of that collection. The additional points are ignored.

Parameters: *functionCode* Function code. May take the following values:
1) "clearPoints" – Reset the collection of points in the memory.
2) "addPoint" – Add a point to the collection.
3) "computeIntersect" – Calculation of the intersection taking into account the first 4 points

Return value: Depends on the function code

Function code	Type of return
clearPoints	<void>
addPoint	<void>
computeCenter	<vector>

Example: Complete example of calculating the centre of a circle

```
intersect:= (0,0,0,0);
p1:= (1,2,3,4); p2:= (5,6,7,8); p3:= (9,10,11,12);
ModuleCmd 'MATH' 'cmd=LINEINTERSECT func=clearpoints';
ModuleCmd 'MATH' 'cmd=LINEINTERSECT func=addPoint' p1;
ModuleCmd 'MATH' 'cmd=LINEINTERSECT func=addPoint' p2;
ModuleCmd 'MATH' 'cmd=LINEINTERSECT func=addPoint' p3;
intersect:= ModuleCmd 'MATH' 'cmd=LINEINTERSECT
func=computeIntersect';
```

See also: CIRCLECENTER

8. Technical Support

8.1.1. For a better service …

Have you read the FAQ and the check-list and still not found an answer your questions? Before contacting us, please note down the following information concerning your product:

- Serial number and product key for your equipment
- Software version(s) used
- Error message, alarm, or visual signals displayed by the interface.

8.1.2. Contact

You can find extensive information on our website: **www.asyril.com**

You can also contact our Customer Service department:

support@asyril.com +41 26 653 7190

Alphabetical index

Thematic index

OPERATORS

LOOPS

ROBOT

ASYVIEW

IOWAGO

TURNINGTABLE

MATH module instructions

Revision table

Rev.	Date	Author	Comments
A	31.07.2012	SiA	Initial version
A2	03.12.2012	ZuM	Addition of information related to the mesh frame, maths module and various other modifications
A3	13.02.2013	ZuM	Addition of the type 7xx.xxx action code
A4	26.01.2015	ZuM	Addition according to the software updates. New IODevice of type "Cylinder" and update of the MoveTo documenting the new 4^{th} axis related arguments.
B	25.06.2015	ZuM	Adding the complete new set of Asyview module commands to match the changes brought by the Asyview since version 3 (Asyview V3) Adding the new "setEnable" robot command.
B1	25.08.2016	DaM	Documentation and product name update
B2	03.02.2017	BeJ	Modification of §7.2 relating to the I/O extension module
C	13.02.2018	PeD	Modification of the Asyview module commands for the Asyview V4. It requires the robot ≥V4.6.0.

Asyril SA
Z.I. le Vivier 22
Ch-1690 Villaz-St-Pierre
Switzerland
Tel. +41 26 653 71 90
Fax +41 26 653 71 91
info@asyril.com
www.asyril.com