robótica

Robótica
Departamento de Ingeniería de Sistemas y Automática
Universidad Politécnica de Valencia

EUROPEAN COMMISSION. DIRECTORATE GENERAL - JRC
Joint Research Centre – Ispra
Institute for the Protection and the Security of the Citizen (IPSC)

virtualrobot

virtualrobot modeller

virtualrobot simulator

virtualrobot translator

VIRTUAL ROBOT EXTERNAL ACCESS LIBRARY

October 2012, Version 6.7b (VREAL 6.7b)

Developed by

robótica
Department of System Engineering and Control (DISA)
Polytechnic University of Valencia (UPV)

Contact Address

vrs@isa.upv.es
http://robotica.isa.upv.es/virtualrobot
DISA - UPV
Camino de Vera s/n E-46022, Valencia (Spain)
Tf: +34-963.879.575 Fax: +34-963.879.579

Supported by

European Commission. Directorate General - JRC
Joint Research Centre - Ispra
Institute for the Protection and the Security of the Citizen (IPSC)

Index

1. Objective

The objective of this document is to explain a library developed to provide access to some of the functionality of *Virtual Robot Simulator* (*VRS*) from a user application. The library, called *VR External Access Library* (*VREAL*), is implemented as a Dynamic Link Library on the file "vreal.dll". The user application can manage, through this library, the elements defined on *VRS*.

VREAL can be compiled with a dynamic link on the user application. The "vreal.h" file contains the definition of the library, while the "vreal.lib" includes the library access specification. Both files must be used to compile and link the user application, while the "vreal.dll" file must be accessible for the user application execution.

2. Implementation

The *VR External Access Library* provides an interface formed by a set of functions whose objective is to allow the user to interact with the kernel of *VRS*.

In addition to constant and type definition, there are the following independent sets of functions:

- A set of functions to initialize and close the Library
- A set of functions to manage files
- A set of functions to edit robots and the environment
- A set of functions to define and modify robot parameters
- A set of functions for speed control
- A set of functions for tool handle
- A set of functions for robot motion
- A set of functions for robot attachment
- A set of functions for inputs and outputs
- A set of functions for environment
- A set of functions to handle with robot operation
- A set of functions to handle an auxiliary list of figures
- A set of functions to handle the display options of *VRS*
- A set of functions for video recording

3. Library-Kernel Communication

The communication between the user application and the kernel of *VRS* can be made in local mode or remote mode:

- In local mode, the user application must be run on the same computer than *VRS*. In fact, *VRS* will ask for the name of the user application to be run and will start it. The user application must initialize *VREAL* with the function provided for this (`alInitialize`) with no parameter at all.

Then, the user application has access to all the functionality of this library.

- In remote mode, the user application can be run on the same computer than *VRS* or in any other computer. In both cases, a TCP/IP protocol is used for application communication. The user must first activate on *VRS* the VREAL Remote Mode (**File>>Start Remote VREAL Server**). Then the user must start his/her application wherever it is and the user application must initialize *VREAL* with the function provided for this (`alInitialize`) with the appropriate parameter (the name or ip-address of the PC where *VRS* is running). Then, the user application has access to all the functionality of this library.

In any case, **THE LIBRARY MUST BE CLOSED** before the user application finishes.

4. Function Parameters

Some parameters are used in most of the functions and therefore are explained once in this section.

For example, three parameters are used as identifier in many functions:

- `robotId` is the robot identifier. It is returned only by the function `alLoadRobot` and used by any function that applies on a robot.
- `objectId` is the object identifier. It is returned only by the function `alGetObjectId` and used by any function that applies on an object.
- `partId` is the part identifier. It is returned only by the function `alGetPartId` and used by any function that applies on a part.
- `figureId` is the figure identifier. It is returned by any `Add` function and used by any function that applies on a figure.

A location is usually represented in a function be means of:

- The position represented with the values `x,y,z`.
- The orientation represented with the values `alpha, beta` and `gamma`. These three angles are the Euler Angles Type 2 (also called ZYZ).

According to these parameters, a location is computed with the following steps:

1. Rotation of alpha angle related to Z axis
2. Rotation of beta angle related to V axis (Y axis of mobile frame)
3. Rotation of gamma angle related to W axis (Z axis of mobile frame)
4. Translations of x,y,z values related to X, Y, Z axes (axes of fix frame)

There are some options on locations summarized as:

- The six values of the location can be grouped on an array as explained on Appendix A.
- In both cases, a different Euler angle type can be used as explained on Appendix B.
- The location can also be represented as a transformation matrix, as explained on Appendix C.

5. Constant Definition

The following constants are defined:

NUM_DOF		NUM_ROBOTS	
NUM_TOOLS	NUM_OBJECTS		NUM_PARTS
RESET		SYNCHRO	
POINTTOPOINT		LINEAR	
ABSOLUTE_MOVEMENT		RELATIVE_MOVEMENT	
ORIGIN	TOOL_FRAME		WORLD
CHECK_RANGE		NO_CHECK_RANGE	
CHECK_ORIENTATION		NO_CHECK_ORIENTATION	
	CLOSEST		
RIGHT_DOWN		RIGHT_UP	
LEFT_UP		LEFT_DOWN	
POSITIVE_WRIST		NEGATIVE_WRIST	
WIRED	SHADED		HIDDEN
ENVIRONMENT_LOADED		ENVIRONMENT_NOT_LOADED	
VISIBLE		INVISIBLE	
ACTIVE_TRACE		NO_ACTIVE_TRACE	
HIDE_TRACE		NO_HIDE_TRACE	
CHECK_COINCIDENCE		NO_CHECK_COINCIDENCE	
TRANSLATION		ROTATION	
X_AXIS	Y_AXIS		Z_AXIS
U_AXIS	V_AXIS		W_AXIS
EULER_ANGLES_TYPE_1	EULER_ANGLES_TYPE_2		EULER_ANGLES_TYPE_3
NUM_DIGITAL_INPUTS		NUM_DIGITAL_OUTPUTS	
NUM_ANALOGICAL_INPUTS		NUM_ANALOGICAL_OUTPUTS	
FIRST_VIEWPORT		SECOND_VIEWPORT	
THIRD_VIEWPORT		FOURTH_VIEWPORT	
	FRAME		
3DPOINT	LINE	DISK	TRIANGLE	3DFACE
BOX	PYRAMID	TRIANGULAR_PYRAMID	TENT	WEDGE
CONE	TUBE	SPHERE	DOME	TORUS
CONE_SPHERE	CONE_TWO_SPHERES	TENT_CYLINDER	TENT_TWO_CYLINDERS	

The following color inversion table gives the possible parameters and meanings for the color inversion functions:

PARAMETER	MEANING
NOT_INVERTED_COLOR	Proper object color
INVERTED_COLOR	Color inversion
INVERT_IN_WHITE_COLOR	Set color to white
INVERT_IN_YELLOW_COLOR	Set color to yellow
INVERT_IN_PINK_COLOR	Set color to pink
INVERT_IN_ORANGE_COLOR	Set color to orange
INVERT_IN_RED_COLOR	Set color to red
INVERT_IN_PURPLE_COLOR	Set color to purple
INVERT_IN_LIGHT_GREEN_COLOR	Set color to light green
INVERT_IN_DARK_GREEN_COLOR	Set color to dark green
INVERT_IN_LIGHT_BLUE_COLOR	Set color to light blue
INVERT_IN_DARK_BLUE_COLOR	Set color to dark blue
INVERT_IN_GREY_COLOR	Set color to grey
INVERT_IN_BLACK_COLOR	Set color to black

Their values are defined in one of the following files:

- "Includes\VReal\VRealExternalDefines.h".
- "Includes\VRealCommunications\VRealCommunicationsExternalDefines.h".
- "Includes\VRTransf\VRTransfExternalDefines.h".
- "Includes\VRMaths\VRMathsExternalDefines.h".

All these files are included in "Includes\VReal\VReal.h".

6. Type Definition

In "Includes\GeneralDefines.h", the following types are defined:

```
typedef char STRING[256]

typedef unsigned long COLORREF
```

7. Error Codes

Every function in the library has at least the following possible values to be returned (unless other case stated):

`RET_OK`	Successful execution
`RET_ERROR`	Error in execution

When a function of VREAL returns an error, it can be produced by any of the following modules:

- VRMaths
- VRTransf
- VRStdio
- VRealCommunications
- VREAL
- VRPROL
- Virtual Robot Simulator

To recognize the module that generates the error, the following constants are defined (as defined in the file **Includes\GeneralDefines.h**):

MODULE	CONSTANT	VALUE
VRMaths	`VR_MATHS_ERROR`	`1000`
VRTransf	`VR_TRANSF_ERROR`	`2000`
VRStdio	`VR_STDIO_ERROR`	`3000`
VrealCommunications	`VR_VREAL_COMM_ERROR`	`4000`
VREAL	`VR_VREAL_ERROR`	`5000`
VRPROL	`VR_VRPROL_ERROR`	`6000`
VRS	`VR_ERROR`	`7000`

The returned error will be the addition of a specific error and the module constant identifier, as defined in the header file of each module. A summary is included here:

MODULE	Header File	Errors
VRMaths	VRMathsError.h	
	#DEFINE INVALID_SIN_VALUE	VR_MATHS_ERROR + 1
	#DEFINE INVALID_COS_VALUE	VR_MATHS_ERROR + 2
VRTransf	VRTransfError.h	
	#DEFINE INVALID_NO_POINTS	VR_TRANSF_ERROR + 1
	#DEFINE INVALID_TRANSFORMATION	VR_TRANSF_ERROR + 2
	#DEFINE INVALID_AXIS	VR_TRANSF_ERROR + 3
	#DEFINE INVALID_EULER_TYPE	VR_TRANSF_ERROR + 4
	#DEFINE INVALID_OPERATION	VR_TRANSF_ERROR + 5
	#DEFINE INVALID_QUATERNION	VR_TRANSF_ERROR + 6
VRStdio	VRStdioError.h	
	#DEFINE INVALID_DIALOG_TYPE	VR_STDIO_ERROR + 1
VRealCommunications	VRealCommunictaionsError.h	
	#DEFINE ERROR_INI_FILE	VR_VREAL_COMM_ERROR + 1
	#DEFINE TIMEOUT_ERROR	VR_VREAL_COMM_ERROR + 2
VREAL	VrealError.h	
	#DEFINE INVALID_ROBOT_NAME	VR_VREAL_ERROR + 1
	#DEFINE INVALID_JOINT_NUMBER	VR_VREAL_ERROR + 2
	#DEFINE INVALID_FRAME	VR_VREAL_ERROR + 3
VRPROL	VRProlError.h	
	#DEFINE ERROR_IN_VREAL_FUNCTION	VR_VRPROL_ERROR + 1
VRS	VrealError.h	
	#DEFINE ROBOT_OUT_OF_RANGE	VR_ERROR + 6
	#DEFINE INVERSE_KINEMATICS_ERROR	VR_ERROR + 7
	#DEFINE ROBOT_INVALID_ID	VR_ERROR + 8
	#DEFINE ROBOT_INVALID_FREEDOM_DEGREES	VR_ERROR + 11
	#DEFINE ROBOT_INVALID_IKALGORITHM	VR_ERROR + 12
	#DEFINE ROBOT_INVALID_TOOL_FRAME	VR_ERROR + 16
	#DEFINE ROBOT_INVALID_POSITION	VR_ERROR + 19
	#DEFINE ROBOT_INVALID_ORIENTATION	VR_ERROR + 20
	#DEFINE ROBOT_INVALID_TYPE	VR_ERROR + 23
	#DEFINE WRIST_INVALID_TYPE	VR_ERROR + 24
	#DEFINE ARM_INVALID_TYPE	VR_ERROR + 25
	#DEFINE INVALID_SOLUTION_SELECTED	VR_ERROR + 26
	#DEFINE ROBOT_OUT_OF_FRAME	VR_ERROR + 27
	#DEFINE ROBOT_ERROR_LOCATION	VR_ERROR + 28

8. Initialization Functions

There are two functions to initialize and close the library:

- `alInitialize`

  a) For local mode:

  ```
  int alInitialize()
  ```

  This function initializes the VREAL library in local mode. It must be called before any other function of the library is used.

  b) For remote mode:

  ```
  int alInitialize(STRING host)
  ```

  The `host` parameter indicates the name or ip-address of the PC where *VRS* is running in VREAL Remote Mode. A network connection is required. The VREAL Remote Mode (**File>>Start Remote VREAL Server**) must be activated on *VRS*.

- `alClose`

  ```
  int alClose()
  ```

  This function closes the VREAL library. **It must be called before finishing the user application in any of the modes (local or remote).**

9. File Functions

This set of functions mainly allows to load and close robots and environment.

- `alLoadRobot`

```
int alLoadRobot(
     STRING fileName,
     int *robotId)
```

  This function loads on *VRS* a new robot from a rkf file (indicated on `fileName`) and gives back the robot identifier on `robotID`. When a robot with the same name exists, *VRS* adds a number (1,2,3,…) to the robot name. The maximum number of robots is defined by the constant `NUM_ROBOTS`. The loaded robot becomes the active robot. The file name starts from *VR-Path*.

- `alCloseRobot`

```
int alCloseRobot(int robotId)
```

  This function closes on *VRS* the robot specified with `robotID`. If the closed robot is the active robot, the first robot on the list of robots becomes the active robot.

- `alSaveRKFRobot`

```
int alSaveRKFRobot(int robotId , STRING rkfFileName)
```

  This function saves a robot as a robot kinematics file (RKF). An error is returned if there is no robot with this identifier. The file name starts from *VR-Path*.
  ********** NOT IMPLEMENTED **********

- `alSaveRGFRobot`

```
int alSaveRGFRobot(int robotId , STRING rgfFileName)
```

  This function saves a robot as a robot geometric file (RGF).
  An error is returned if there is no robot with this identifier.
  The file name starts from *VR-Path*.
  ********** NOT IMPLEMENTED **********

- `alLoadEnvironment`

  ```
  int alLoadEnvironment(STRING fileName)
  ```

  This function loads an environment (indicated on `fileName`) on *VRS*. As only one environment can be opened on *VRS*, the new environment will replace any other possible environment. The file name starts from *VR-Path*.

- `alCloseEnvironment`

  ```
  int alCloseEnvironment()
  ```

  This function closes the environment on *VRS*.

- `alSaveEnvironment`

  ```
  int alSaveEnvironment(STRING enfFileName)
  ```

  This function saves the environment as an environment file (ENF). An error is returned if the environment is empty. The file name starts from *VR-Path*.

- `alCloseAll`

  ```
  int alCloseAll()
  ```

  This function closes all the robots and the environment on *VRS*.

- `alGetVRPath`

  ```
  int alGetVRPath(STRING VRPath)
  ```

  This function gets the absolute path for the Virtual Robot Simulator installation (that is, where the VRS executable file is).

10. Edit Functions

This set of functions mainly allows placing robots and environment.

- `alGetRobotFrame`

```
int alGetRobotFrame(
     int robotId,
     double *x, double *y, double *z,
     double *alpha, double *beta, double *gamma)
```

This function obtains the location of robot frame related to world frame, that is, where is located the robot in the space.
The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:
- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alPlaceRobot`

```
int alPlaceRobot(
     int robotId,
     double x, double y, double z,
     double alpha, double beta, double gamma)
```

This function places a robot on VRS, that is, specifies where the robot is located in the space.
The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:
- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alGetEnvironmentFrame`

```
int alGetEnvironmentFrame(
     double *x, double *y, double *z,
     double *alpha, double *beta, double *gamma)
```

This function obtains the location of the environment frame related to world frame, that is, where is located the environment in the space.
The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:
- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alPlaceEnvironment`

```
int alPlaceEnvironment(
     double x, double y, double z,
     double alpha, double beta, double gamma)
```

This function places the environment on VRS, that is, specifies where the environment is located in the space.
The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:
- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

11. Robot Definition Functions

This set of functions is mainly designed to obtain and modify the robot configuration.

- `alSetActiveRobot`

```
int alSetActiveRobot(
     int robotId)
```

  This function actives an specific robot specified with the parameter. The information of the active robot is shown on the dynamic information field.

- `alGetActiveRobot`

```
int alGetActiveRobot(
     int *robotId)
```

  This function obtains in its parameter the robot identifier of the active robot.

- `alInvertRobotColor`

```
int alInvertRobotColor(
     int robotId,
     int invert)
```

  This function inverts the color of the robot specified in the first parameter. The second parameter can be any of the values indicated in the table of color inversions.

- `alInvertRobotLinkColor`

```
int alInvertRobotLinkColor(
     int robotId,
     int linkId,
     int invert)
```

  This function inverts the color the link specified in the first parameter of the robot specified in the second parameter. The third parameter can be any of the values indicated in the table of color inversions.

- `alInvertRobotAdapterColor`

```
int alInvertRobotAdapterColor(
     int robotId,
     int invert)
```

This function inverts the color of the adapter of the robot specified in the first parameter. The second parameter can be any of the values indicated in the table of color inversions.

- `alInvertRobotToolColor`

```
int alInvertRobotToolColor(
     int robotId,
     int toolId,
     int invert)
```

This function inverts the color of the tool specified in the second parameter of the robot specified in the first parameter. The third parameter can be any of the values indicated in the table of color inversions.

- `alGetAvailableRobots`

```
int alGetAvailableRobots(
     int robotIds[NUM_ROBOTS],
     STRING robotNames[NUM_ROBOTS],
     int *numberOfRobots)
```

This function obtains the arrays of identifiers and associated names of the available robots on *VRS*, that is, the robots loaded. The number of available robots is returned on the last parameter.

Parameters:
`robotIds` is an array with the identifiers of the available robots
`robotNames` is an array of names of the available robots
`numberOfRobots` is the number of available robots. If it is equal to 0, there is no robot available on *VRS*. It specifies the number of valid values on the arrays.

- `alGetRobotIdentifier`

```
int alGetRobotIdentifier(
     STRING robotName,
     int *robotId)
```

This function obtains the first robot identifier for a robot name. An error is returned if there is no robot with this name. The name is the Name field of GENERAL section in the RKF file of the robot. In order to avoid name duplications, when *VRS* loads a robot with the same name than an already loaded robot, changes the name adding a number to it. That is, if three robots with the same name (“robot”) are loaded, the names will be “robot”, “robot1”, “robot2”.
Parameters:
`robotName` is a string with the robot name.
`robotId` is the robot identifier returned by the function

- `alGetRobotConfiguration`

```
int alGetRobotConfiguration(
     int robotId,
     int *numberOfDOF,
     STRING robotType,
     STRING wristType,
     STRING joints)
```

This function obtains the configuration of a robot.

Parameters:
`numberOfDOF` is the number of degrees of freedom of the robot given by the `robotId` parameter. `robotType`, `wristType`, `joints` are the parameters of robot configuration as defined on Robot Kinematics File (RKF) description (see VRFFD document).

- `alGetRobotIKAlgorithm`

```
int alGetRobotIKAlgorithm(
     int robotId,
     STRING ikAlgorithm)
```

This function obtains the inverse kinematics algorithm for the robot indicated in the `ikAlgorithm` parameter.

- `alGetRobotDHFrame0`

```
int alGetRobotDHFrame0(
     int robotId,
     double dhFrame0[4][4])
```

This function obtains the transformation matrix for robot DH frame 0 related to robot frame.

- `alGetRobotDHTable`

```
int alGetRobotDHTable(
     int robotId,
     double dhTable[NUM_DOF][4])
```

This function obtains the DH Table for a robot.

- `alGetRobotUserSpace`

```
int alGetRobotUserSpace(
     int robotId,
     double userSpace[NUM_DOF][2])
```

This function obtains the user space definition for a robot.

- `alGetRobotRanges`

```
int alGetRobotRanges(
     int robotId,
     double jointRanges[NUM_DOF][2])
```

This function obtains the joints ranges of the robot that indicates the `robotId` parameter. The joint ranges are returned in the matrix named `jointRanges`.

- `alGetRobotSynchro`

```
int alGetRobotSynchro(
     int robotId,
     double jointSynchro[NUM_DOF])
```

This function obtains the synchronization configuration for a robot.

- `alGetRobotOrigin`

```
int alGetRobotOrigin(
     int robotId,
     double originFrame[4][4])
```

This function obtains the transformation matrix for robot programming Origin frame related to robot frame.

- `alGetRobotSettings`

```
int alGetRobotSettings(
     int robotId,
     int *armConf,
     int *wristConf)
```

This function obtains the robot configuration settings for a robot. This configuration will be used to determine robot configuration on robot motions. The first parameter indicates the robot whose configuration must be obtained. The rest of values give the configuration according to:

`armConf` may be:

`CLOSEST`, `RIGHT_DOWN`, `RIGHT_UP`, `LEFT_DOWN`, `LEFT_UP`

`wristConf` may be:

`POSITIVE_WRIST`, `NEGATIVE_WRIST`

- alGetRobotCheckRanges

```
int alGetRobotCheckRanges(
     int robotId,
     int *checkRange)
```

This function obtains the robot check range state. When checkRange is NO_CHECK_RANGE the robot moves as if no range limitation is applied. When checkRange is CHECK_RANGE the robot cannot move outside its ranges.

- alSetRobotCheckRanges

```
int alSetRobotCheckRanges(
     int robotId,
     int checkRange)
```

This function defines the robot check range state. When checkRange is NO_CHECK_RANGE the robot moves as if no range limitation is applied. When checkRange is CHECK_RANGE the robot cannot move outside its ranges. The default value is CHECK_RANGE.

- alGetRobotCheckOrientation

```
int alGetRobotCheckOrientation(
     int robotId,
     int *checkOrientation)
```

This function obtains the robot check orientation state. When checkOrientation is NO_CHECK_ORIENTATION the robot moves as if no orientation constraints are specified on movements. When checkOrientation is CHECK_ORIENTATION the robot moves according to orientations constraints.

- alSetRobotCheckOrientation

```
int alSetRobotCheckOrientation(
     int robotId,
     int checkOrientation)
```

This function defines the robot check orientation state. When checkOrientation is NO_CHECK_ORIENTATION the robot moves as if no orientation constraints are specified on movements. When checkOrientation is CHECK_ORIENTATION the robot moves according to orientations constraints. The default value is CHECK_ORIENTATION.

12. Speed Functions

This set of functions is mainly designed to control speed scale and robot speed.

- `alGetSpeedScale`

```
int alGetSpeedScale(
     double *speedScale)
```

This function sets the speed scale. The speed scale value will be between -1.0 and 1.0, where -1.0 indicates the minimum speed scale and 1.0 the maximum speed scale.

- `alSetSpeedScale`

```
int alSetSpeedScale(
     double speedScale)
```

This function sets the speed scale. The speed scale value must be between -1.0 and 1.0 (otherwise an error is returned and speed scale set to closest value), where -1.0 indicates the minimum speed scale and 1.0 the maximum speed scale. The initial default value is 0 (no speed scaling). The meaning of the speed scale is as follows:

- If the speed scale is set to a positive value `speedScale`, the speed of all the robots are mapped to the interval [`speedScale`,1]. That is, for a `speedScale` value of 0.25, the robot speeds will be mapped from [0,1] to [0.25,1] and a robot speed of 0.5 becomes 0.625. As extreme value, when the speed scale is set to 1, all the robot speeds will be scaled to 1.
- If the speed scale is set to a negative value `speedScale`, the speed of all the robots are mapped to the interval [0,1+`speedScale`]. That is, for a `speedScale` value of -0.25, the robot speeds will be mapped from [0,1] to [0,0.75] and a robot speed of 0.5 becomes 0.375. As extreme value, when the speed scale is set to -1, all the robot speeds will be scaled to 0.

Hence, positive speed scale will make faster the slow robot motions, while negative speed scale will make slower the fast robot motions.

- `alGetRobotSpeed`

```
int alGetRobotSpeed(
     int robotId,
     double *robotSpeed)
```

This function gets the speed of the robot given by `robotId` parameter. The `robotSpeed` value will be between 0.0 and 1.0, where 0.0 indicates the slowest speed and 1.0 the fastest speed.

- `alSetRobotSpeed`

```
int alSetRobotSpeed(
     int robotId,
     double robotSpeed)
```

This function sets the speed of the robot given by `robotId` parameter. The `robotSpeed` value must be between 0.0 and 1.0, where 0.0 indicates the slowest speed and 1.0 the fastest speed. The initial default value for any robot is 0.5.

The robot speed, after adjusted with the global speed scale, is the simulation robot speed used to compute delays in motions. That is, when the simulation robot speed is 1.0, no delay is generated for drawing while robot motion (drawing is as fast as the computer allows). On the other side, when simulation robot speed is lower than 1.0, delays are generated for drawing different interpolation steps in robot motion.

13. Tool Functions

This set of functions is mainly designed to manage the tool.

- `alGetNumberOfToolFrames`

```
int alGetNumberOfToolFrames(
     int robotId,
     int *numberToolFrames)
```

This function obtains the number of ToolFrames defined for a robot. The first parameter indicates the robot identifier. The second value gives the number of the defined ToolFrames for this robot. ToolFrame0 is not considered.

- `alGetActiveToolFrame`

```
int alGetActiveToolFrame(
     int robotId,
     int *toolFrameId)
```

This function obtains the active ToolFrame for a robot. The first parameter indicates the robot whose ToolFrame must be obtained. The second value gives the number of the ToolFrame that is active.

- `alSetActiveToolFrame`

```
int alSetActiveToolFrame(
     int robotId,
     int toolFrameId)
```

This function sets the active ToolFrame. The first parameter indicates the robot whose ToolFrame must be changed. The second value indicates the number of the ToolFrame that must be active after the function execution. If the ToolFrame is not defined, the function returns and the active ToolFrame is not changed.

- alGetToolFrameDefinition

```
int alGetToolFrameDefinition(
     int robotId,
     double *x, double *y, double *z,
     double *alpha, double *beta, double *gamma)
```

This function obtains the definition of the active ToolFrame. The first parameter indicates the robot identifier. The rest of parameters give the location of the actual ToolFrame related to ToolFrame0. When the active ToolFrame is ToolFrame0, the location is related to the last DH frame.

The location is represented with a position (x,y,z) and a orientation (alpha, beta and gamma) specified in Euler Angles type 2.

- alGetAvailableTools

```
int alGetAvailableTools(
     int robotId,
     int toolIds[NUM_TOOLS],
     STRING toolNames[NUM_TOOLS],
     int *numberOfTools)
```

This function obtains the arrays of identifiers and associated names of the available tools of the robotId parameter. The number of available tools for this robot is returned on the last parameter.

Parameters:
robotId is the identifier of the robot
toolIds is an array with the identifiers of the available tools
toolNames is an array of names of the available tools
numberOfTools is the number of available tools.

- alGetActiveTool

```
int alGetActiveTool(
     int robotId,
     int *toolId)
```

This function obtains the active tool for a robot. The first parameter indicates the robot whose tool must be obtained. The second value gives the number of the tool that is active. This value will be 0 if the robot has no tool.

- alSetActiveTool

```
int alSetActiveTool(
     int robotId,
     int toolId)
```

This function sets the active Tool. The first parameter indicates the robot whose Tool must be changed. The second value indicates the number of the Tool that must be active after the function execution. If the robot does not have this tool, an error is returned.

- `alGetNumberOfToolStates`

```
int alGetNumberOfToolStates(
     int robotId,
     int *noToolStates)
```

This function obtains the number of tool states of the active tool for a robot. The first parameter indicates the robot whose number of tool states must be obtained. The function returns in the second parameter the number of tool states of the active tool.

- `alGetToolStatus`

```
int alGetToolStatus(
     int robotId,
     double *toolStatus)
```

This function obtains the tool status of the active tool for a robot. The first parameter indicates the robot whose tool status must be obtained. The function returns in the second parameter the tool status of the active tool in the interval [0.0,1.0].

- `alSetToolStatus`

```
int alSetToolStatus(
     int robotId,
     double toolStatus)
```

This function sets the tool status for the active tool. The first parameter indicates the robot whose tool must be changed. The second value indicates the tool status of the active tool of this robot. The value must be in the interval [0.0,1.0], assigning the closest end value in other case.

14. Robot Motion Functions

This set of functions is designed to move robots.

- `alRobotReset`

```
int alRobotReset(
     int robotId,
     int resetType)
```

  This function moves the robot which identifier is equal to the first parameter to its reset configuration or to its synchronism configuration depending on the value of the second parameter. The value of this second parameter must be `RESET` or `SYNCHRO`.

- `alGetRobotJoints`

```
int alGetRobotJoints(
     int robotId,
     double joints[NUM_DOF])
```

  This function obtains the joints configuration of the robot that indicates the `robotId` parameter. The joint values are returned in the array named `joints`.

- `alSetRobotJoints`

```
int alSetRobotJoints(
     int robotId,
     double joints[NUM_DOF])
```

  This function moves the robot, which identifier is passed as the first parameter of the function, to the configuration where the values of its joints are the specified in the array that is passed as the second parameter of the function. No interpolation is applied for the movement, **producing a jump** from a robot configuration to the new one.

- `alMoveRobotJoints`

```
int alMoveRobotJoints(
     int robotId,
     double joints[NUM_DOF])
```

  This function moves the robot, which identifier is passed as the first parameter of the function, to the configuration where the values of its joints are the specified in the array that is passed as the second parameter of the function. A linear interpolation on Joint Space is generated for the movement.

- alMoveOneRobotJoint

```
int alMoveOneRobotJoint(
     int robotId,
     int joint,
     double jointValue,
     int absoluteMovement)
```

This function moves the specified joint of a robot to a new value, specified as an absolute or relative motion according to the value of the last parameter. The absoluteMovement parameter can have one of these two values:

ABSOLUTE_MOVEMENT	For Absolute movement
RELATIVE_MOVEMENT	For Relative movement

A linear interpolation on Joint Space is generated for the movement. An error is returned when the specified joint is not valid (less than 1 or greater than the DOFs of the robot).

- alGetRobotLocation

```
int alGetRobotLocation(
     int robotId,
     double *x, double *y, double *z,
     double *alpha, double *beta, double *gamma)
```

This function obtains the location of the active ToolFrame of the robot given by the robotId parameter. ToolFrame location is related to robot Programming Origin frame.

The location is represented with a position (x,y,z) and a orientation (alpha, beta and gamma) specified in Euler Angles type 2.

Options:

- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.
- A last parameter can be specified to obtain robot location from WORLD frame with the following function interface (always specifying Euler angles according to Appendix B):

```
int alGetRobotLocation(int robotId, double *x, double *y,
double *z, double *alpha, double *beta, double *gamma,
int frame, int eulerType)
```

An error is returned if the parameter used is TOOL_FRAME.

- alMoveRobot

```
int alMoveRobot(
     int robotId,
     double x, double y, double z,
     double alpha, double beta, double gamma,
     int linearMovement,
     int absoluteMovement,
     int frame)
```

This function moves a robot to a given position.

Parameters:
The parameter robotId is the robot to be moved.
The linearMovement parameter can have two possible values:

POINTTOPOINT	For PointToPoint Movement
LINEAR	For Linear Movement

When the value of this parameter is POINTTOPOINT, the robot moves from its current location to the destination location without following any special trajectory. When the value of this parameter is LINEAR, the robot moves from its current location to the destination location according to a linear trajectory between the two locations for the ToolFrame.
The absoluteMovement parameter can have one of these two values:

ABSOLUTE_MOVEMENT	For Absolute movement
RELATIVE_MOVEMENT	For Relative movement

If the value is ABSOLUTE_MOVEMENT, it means that the x, y, z, alpha, beta, gamma values represent an absolute movement. Otherwise if the value is RELATIVE_MOVEMENT the x, y, z, alpha, beta, gamma values represent a relative movement from the current location.
The frame parameter can have one of these values:

ORIGIN	For Origin Frame
TOOL_FRAME	For Tool Frame
WORLD	For World Frame

If the value is ORIGIN, the Programming Origin Frame is taken as the reference of the location. If the value of the frame parameter is TOOL_FRAME, the current location of the active ToolFrame is taken as the reference frame of the movement. If the value of the frame parameter is WORLD, the World Frame is taken as the reference frame of the movement. Therefore, the specified location represents the location of the active ToolFrame related to the robot programming Origin Frame (for ORIGIN), to the current location of the active ToolFrame (for TOOL_FRAME) or the world frame (for WORLD).

The location is represented with a position (x,y,z) and a orientation (alpha, beta and gamma) specified in Euler Angles type 2.

Options:

- The last three parameters can be avoided to consider a `POINTTOPOINT`, `ABSOLUTE_MOVEMENT` related to `ORIGIN` frame. Then the function can be called with this interface:
  ```
  int alMoveRobot(int robotId, double x, double y, double z,
  double alpha, double beta, double gamma)
  ```
- The location can be managed as an array according to Appendix A. Also the three parameters can be avoided as in previous option, giving the following interface:
  ```
  int alMoveRobot(int robotId, double location[6])
  ```
- A different type for the Euler angles can be specified according to Appendix B. This option is not compatible with the reduced version of first and second options.
- The location can also be managed as a transformation matrix, according to Appendix C. Also the three parameters can be avoided as in previous option, giving the following interface:
  ```
  int alMoveRobot(int robotId, double transformation[4][4])
  ```

- `alMove`

```
int alMove(
     int robotId,
     double x, double y, double z,
     double alpha, double beta, double gamma,
     int moveParameter)
```

This function is the same as the previous one but with the motion parameters in just one parameter as addition of them, as in the example:
`alMove(robotId,x,y,z,a,b,g,LINEAR+RELATIVE_MOVEMENT+WORLD)`

`POINTTOPOINT`, `ABSOLUTE_MOVEMENT` and `ORIGIN` are default values and must not be added. None of the parameters can be added twice. The options are the same than in previous function.

- `alSetRobotLocation`

```
int alSetRobotLocation(
     int robotId,
     double x, double y, double z,
     double alpha, double beta, double gamma,
     int absoluteMovement,
     int frame)
```

This function moves a robot to a given position without interpolation. The effect is that the robot disappears from its current configuration and appears on the specified configuration, without any transit from a configuration to the new one. The parameters and options are the same as in `alMoveRobot`, except `linearMovement`, which have no sense in this function.

- alApproxToLocation

```
int alApproxToLocation(
     int robotId,
     double x, double y, double z,
     double alpha, double beta, double gamma,
     int linearMovement,
     int frame,
     double xDistance,
     double yDistance,
     double zDistance)
```

This function approximates the robot indicated in `robotId` parameter to a location. The actual ToolFrame of the robot will result with the same orientation with the referred location but the position will be the same with an offset of `xDistance, yDistance, zDistance` values related to the robot origin frame (when `frame` is `ORIGIN`) or active ToolFrame (when `frame` is `TOOL_FRAME`) according to the `frame` parameter (`WORLD` cannot be used). The `linearMovement` parameter has the same meaning that in `alMoveRobot`.

The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2 **always referred to the robot origin frame**.

Options:

- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- alMoveToPart

```
int alMoveToPart(
     int robotId,
     int partId,
     int opFrameId,
     int linearMovement)
```

This function moves the robot indicated in `robotId` parameter to the location of the `opFrameId` of the `partId` parameter, making coincident the actual ToolFrame of the robot with the referred operation frame of the part, both in position and orientation. The `linearMovement` parameter has the same meaning that in `alMoveRobot`.

- `alApproxToPart`

```
int alApproxToPart(
     int robotId,
     int partId,
     int opFrameId,
     int linearMovement,
     double xDistance,
     double yDistance,
     double zDistance)
```

This function approximates the robot indicated in `robotId` parameter to the location of the `opFrameId` of the `partId` parameter. The actual ToolFrame of the robot will result with the same orientation with the referred operation frame of the part but the position will be the same with an offset of `xDistance`, `yDistance`, `zDistance` value in the axes of the operation frame. Note that usually negative values will be used for approximation. The `linearMovement` parameter has the same meaning that in `alMoveRobot`.

- `alGetRobotZone`

```
int alGetRobotZone(
     int robotId,
     double *robotZone)
```

This function obtains the zone for a robot. The zone value will be in the interval [0.0,1.0].

- `alSetRobotZone`

```
int alSetRobotZone(
     int robotId,
     double robotZone)
```

This function defines the zone for a robot. The zone value must be in the interval [0.0,1.0]. The default value is 0.5. This function has no visible effect on *VRS* but it will be transmitted to the robot controller when a robot is connected to *VRS*.

15. Robot Attachment Functions

This set of functions is designed to attach robots.

- `alAttachRobot2RobotToolFrame`

```
int alAttachRobot2RobotToolFrame(
     int robotAttachedId,
     int robotPlatformId,
     int toolFrameId)
```

  This function attaches a robot (its robot frame) to a Tool Frame of another robot. When the robot used as platform moves, the robot attached modifies its location according to this movement.

- `alFreeRobotAttachment`

```
int alFreeRobotAttachment(
     int robotId)
```

  This function frees a robot if it was attached to another robot.

16. Input/Output Functions

This set of functions is designed to control robot input/output signals. Each robot has defined a set of digital outputs, digital inputs, analogical outputs and analogical inputs defined in the constants:

```
NUM_DIGITAL_INPUTS
NUM_DIGITAL_OUTPUTS
NUM_ANALOGICAL_INPUTS
NUM_ANALOGICAL_OUTPUTS
```

- `alSetDigitalOutput`

```
int alSetDigitalOutput(
     int robotId,
     int digitalOutputNo)
```

  This function sets a digital output of a robot.

- `alResetDigitalOutput`

```
int alResetDigitalOutput(
     int robotId,
     int digitalOutputNo)
```

  This function resets a digital output of a robot.

- `alResetAllDigitalOutput`

```
int alResetAllDigitalOutput(
     int robotId)
```

  This function resets all the digital outputs of a robot.

- `alCheckDigitalOutput`

```
int alCheckDigitalOutput(
     int robotId,
     int digitalOutputNo,
     int *digitalStatus)
```

  This function obtains in `digitalStatus` the state of a digital output of a robot.

- `alGetAllDigitalOutputs`

```
int alGetAllDigitalOutputs(
     int robotId,
     int digitalValues[NUM_DIGITAL_OUTPUTS])
```

  This function reads all the values from the digital outputs of a robot.

- `alConnectDigitalInput`

```
int alConnectDigitalInput(
     int robotId,
     int digitalInputNo,
     int fromRobotId,
     int fromDigitalOutputNo)
```

  This function connects in a digital input of a robot the digital output of another robot.

- `alCheckDigitalInput`

```
int alCheckDigitalInput(
     int robotId,
     int digitalInputNo,
     int *digitalStatus)
```

  This function obtains in `digitalStatus` the state of a digital input of a robot.

- `alGetAllDigitalInputs`

```
int alGetAllDigitalInputs(
     int robotId,
     int digitalValues[NUM_DIGITAL_INPUTS])
```

  This function gets all the values from the digital inputs of a robot.

- `alSetAnalogicalOutput`

```
int alSetAnalogicalOutput(
     int robotId,
     int analogicalOutputNo,
     double value)
```

  This function sends a value to an analogical output of a robot.

- `alGetAnalogicalOutput`

```
int alGetAnalogicalOutput(
     int robotId,
     int analogicalOutputNo,
     double *Value)
```

This function gets a value from an analogical output of a robot.

- `alGetAllAnalogicalOutputs`

```
int alGetAllAnalogicalOutputs(
     int robotId,
     double analogicalValues[NUM_ANALOGICAL_OUTPUTS])
```

This function reads all the values from the analogical outputs of a robot.

- `alConnectAnalogicalInput`

```
int alConnectAnalogicalInput(
     int robotId,
     int analogicalInputNo,
     int fromRobotId,
     int fromAnalogicalOutputNo)
```

This function connects in an analogical input of a robot the analogical output of another robot.

- `alGetAnalogicalInput`

```
int alGetAnalogicalInput(
     int robotId,
     int analogicalInputNo,
     double *value)
```

This function gets a value from an analogical input of a robot.

- `alGetAllAnalogicalInputs`

```
int alGetAllAnalogicalInputs(
     int robotId,
     double analogicalValues[NUM_ANALOGCAL_INPUTS])
```

This function gets all the values from the analogical inputs of a robot.

17. Environment Functions

This set of functions is designed to allow handling the environment elements.

- `alExistsEnvironment`

```
int alExistsEnvironment(
       int *exists)
```

  This function obtains in `exists` if there is an environment loaded on *VRS* (`exists=ENVIRONMENT_LOADED`) or if there is no one (`exists=ENVIRONMENT_NOT_LOADED`).

- `alGetAvailableObjects`

```
int alGetAvailableObjects(
     int objectIds[NUM_OBJECTS],
     STRING objectNames[NUM_OBJECTS],
     int *numberOfObjects)
```

  This function obtains the arrays of identifiers and associated names of the available objects on the current environment of *VRS*, that is, the objects loaded in the environment. The number of available objects is returned on the last parameter.

- `alGetObjectId`

```
int alGetObjectId(
     STRING objectName,
     int *objectId)
```

  This function obtains the first object identifier for a object name. An error is returned if there is no object with this name.

- `alGetObjectLocation`

```
int alGetObjectLocation(
     int objectId,
     double *x, double *y, double *z,
     double *alpha, double *beta, double *gamma)
```

  This function obtains the location of the object given by the `objectId` parameter. The location is related to the environment frame. An error is returned if there is no object with this identifier.

  The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:
- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alSetObjectLocation`

```
int alSetObjectLocation(
     int objectId,
     double x, double y, double z,
     double alpha, double beta, double gamma)
```

This function places the object given by the `objectId` parameter. The location is related to the environment frame. An error is returned if there is no object with this identifier.

The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:
- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alDelObject`

```
int alDelObject(int objectId)
```

This function deletes an object from the environment. An error is returned if there is no object with this identifier.

- `alInvertObjectColor`

```
int alInvertObjectColor(
     int objectId,
     int invert)
```

This function inverts the color of the object specified in the first parameter. The second parameter can be any of the values indicated in the color inversion table.

- `alGetAvailableParts`

```
int alGetAvailableParts(
     int partIds[NUM_PARTS],
     STRING partNames[NUM_PARTS],
     int *numberOfParts)
```

This function obtains the arrays of identifiers and associated names of the available parts on the current environment of *VRS*, that is, the parts loaded in the environment. The number of available parts is returned on the last parameter.

- `alGetPartId`

```
int alGetPartId(
     STRING partName,
     int *partId)
```

This function obtains the first part identifier for a part name. An error is returned if there is no part with this name.

- `alGetPartLocation`

```
int alGetPartLocation(
     int partId,
     double *x, double *y, double *z,
     double *alpha, double *beta, double *gamma)
```

This function obtains the location of the part given by the `partId` parameter. The location is related to the environment frame. An error is returned if there is no part with this identifier.

The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:

- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alSetPartLocation`

```
int alSetPartLocation(
     int partId,
     double x, double y, double z,
     double alpha, double beta, double gamma)
```

This function places the part given by the `partId` parameter. The location is related to the environment frame. An error is returned if there is no part with this identifier.

The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:
- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alDelPart`

```
int alDelPart(int partId)
```

This function deletes a part from the environment. An error is returned if there is no part with this identifier.

- `alInvertPartColor`

```
int alInvertPartColor(
     int partId,
     int invert)
```

This function inverts the color of the object specified in the first parameter. The second parameter can be any of the values indicated in the color inversion table.

- `alGetNumberOfOpFrames`

```
int alGetNumberOfOpFrames(
     int partId,
     int *opFrameNumber)
```

This function gets the number of operation frames of the part indicated by `partId` parameter, in the `opFrameNumber` parameter.

- `alGetPartOperationFrame`

```
int alGetPartOperationFrame(
     int partId,
     int opFrame,
     double *x, double *y, double *z,
     double *alpha, double *beta, double *gamma)
```

This function gives the location of an operation frame indicated by `opFrame` of a part indicated by `partId` parameter. The location is related to the part frame.

The location is represented with a position (`x,y,z`) and a orientation (`alpha, beta` and `gamma`) specified in Euler Angles type 2.

Options:

- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

18. Robot Operation Functions

This set of functions is designed to allow robot operation, including the interaction with the environment.

- `alActiveTrace`

```
int alActiveTrace(
     int robotId,
     int active)
```

This function activates or deactivates the trace of the robot given by the `robotId` parameter. If `active` parameter is `ACTIVE_TRACE`, the trace will be activated but if it is `NO_ACTIVE_TRACE`, the trace will be deactivated. The default value is `NO_ACTIVE_TRACE`, that is, the trace must be activated to be generated. When the trace is active, all locations for active ToolFrame trajectory are stored on the trace according to robot motions. Once a trace has been generated it will be drawn until it is hidden or deleted with one of the next two functions.

Option:

The second parameter can be avoided to use the `ACTIVE_TRACE` value. Then the function can be called with this interface:

```
int alActiveTrace(int robotId)
```

- `alHideTrace`

```
int alHideTrace(
     int robotId,
     int hide)
```

This function hides or shows the trace of the robot given by the `robotId` parameter. If `hide` parameter is `HIDE_TRACE`, the trace will be hidden but when it is `NO_HIDE_TRACE`, the trace will be shown (if there is some trace). The default value is `NO_HIDE_TRACE`, that is, the function must be called with a `HIDE_TRACE` parameter to hide the trace.

Option:

The second parameter can be avoided to use the `HIDE_TRACE` value. Then the function can be called with this interface:

```
int alHideTrace(int robotId)
```

- `alDeleteTrace`

```
int alDeleteTrace(
     int robotId)
```

This function deletes all the locations in the trace of the robot given by the `robotId` parameter, resulting in an empty trace.

- `alSetColorTrace`

```
int alSetColorTrace(
     int robotId,
     COLORREF color)
```

This function changes the color of the trace of the robot given by the `robotId` parameter. The `color` parameter indicates the new color for the draw of the trace.

- `alPickPart`

```
int alPickPart(
     int robotId,
     int partId,
     int opFrameId,
     double toolStatus,
     int checkOpFrame)
```

This function makes the robot indicated in `robotId` parameter to pick the part indicate in `partId` according to the spatial relation between the active ToolFrame of the robot and the operation frame of the part indicated with `opFrameId`.

When `checkOpFrame` is:

- `CHECK_COINCIDENCE` the part will be picked only if the active ToolFrame is coincident (with a tolerance of 10^{-3}) with the operation frame.
- `NO_CHECK_COINCIDENCE` the part will be picked anyway, keeping constant the transformation from the active ToolFrame to the part operation frame.

In any case, the active tool status will be changed to the `toolStatus` parameter.

The effect of the function is that the part will be attached to the robot until the robot is forced to place the part.

Options:

Different tolerances can be defined adding new parameters. The new parameters will be (in this order): tolerance in X, tolerance in Y, tolerance in Z, tolerance in Alpha, tolerance in Beta, tolerance in Gamma. Not all must be used, but the order is always as specified above. Then the part will be picked only when this tolerance is valid for all the specified values. The function is defined as:

```
int alPickPart(int robotId, int partId, int opFrameId,
      double toolStatus, int checkOpFrame,
      double xTolerance, double yTolerance,
      double zTolerance, double alphaTolerance,
      double betaTolerance, double gammaTolerance)
```

The tolerance is represented related to a position (x,y,z) and a orientation (alpha, beta and gamma) specified in Euler Angles type 2.

A different type for the Euler angles can be specified according to Appendix B (only when all the parameters are used).

- alPickPart

```
int alPickPart(
     int robotId,
     double toolStatus)
```

This function (reduced version of previous function) makes the robot indicated in robotId parameter to pick the first part that accomplishes the spatial relation between the active ToolFrame of the robot and any operation frame of the part.
In any case, the active tool status will be changed to the toolStatus parameter.
The effect of the function is that the found part will be attached to the robot until the robot is forced to place the part.

- alPlacePart

```
int alPlacePart(
     int robotId,
     int partId,
     double toolStatus)
```

This function makes the robot indicated in robotId parameter to place the part indicate in partId. In addition, the active tool status will be changed to the toolStatus parameter. The effect of the function is that the part will be detached from the robot. An error is returned if the specified part is not attached to the specified robot.

- alPlacePart

```
int alPlacePart(
     int robotId,
     double toolStatus)
```

This function (reduced version of previous function) makes the robot indicated in robotId parameter to place the part attached to the active tool frame of the robot. In addition, the active tool status will be changed to the toolStatus parameter. The effect of the function is that the part will be detached from the robot. An error is returned if there is no part attached to the active tool frame of the specified robot.

19. Functions for Auxiliary List of Figures

This set of functions mainly allows to manage an auxiliary list of figures. The primitives and their parameters are defined on Virtual Robot Primitive Definition document (VRPD). The location is always related to the World Frame.

- `alAddFrame`

```
int alAddFrame(
     double size,
     int visibility,
     int *figureId)
```

This function adds a coordinate system frame to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`size` is the size to draw the frame.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddPoint`

```
int alAddPoint(
     double x, double y, double z,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a point to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`x, y, z` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddLine`

```
int alAddLine(
     double x1, double y1, double z1,
     double x2, double y2, double z2,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a line to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`x1, y1, z1, x2, y2, z2` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddDisk`

```
int alAddDisk(
     double mainRadius,
     double minorRadius,
     double angle,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a disk to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`mainRadius`, `minorRadius`, `angle` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddTriangle`

```
int alAddTriangle(
     double x1, double y1, double z1,
     double x2, double y2, double z2,
     double x3, double y3, double z3,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a triangle to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`x1,y1,z1,...,x3,y3,z3` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAdd3dFace`

```
int alAdd3dFace(
     double x1, double y1, double z1,
     double x2, double y2, double z2,
     double x3, double y3, double z3,
     double x4, double y4, double z4,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a 3d face to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`x1,y1,z1,...,x4,y4,z4` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddBox`

```
int alAddBox(
     double length,
     double width,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a box to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`length,width, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddPyramid`

```
int alAddPyramid(
     double length,
     double baseWidth,
     double topWidth,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a pyramid to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`length,baseWidth,topWidth,height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddTriangularPyramid`

```
int alAddTriangularPyramid(
     double baseEdge,
     double topEdge,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a triangular pyramid to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`baseEdge, topEdge, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddTent`

```
int alAddTent(
     double baseLength,
     double baseWidth,
     double topLength,
     double topWidth,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds tent a to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`baseLength, baseWidth, topLength, topWidth, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- alAddWedge

```
int alAddWedge(
     double baseLength,
     double width,
     double topLength,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a wedge to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`baseLength,width,topLength,height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- alAddCone

```
int alAddCone(
     double baseRadius,
     double topRadius,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a cone to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`baseRadius, topRadius, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- alAddTube

```
int alAddTube(
     double mainRadius,
     double minorRadius,
     double angle,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a tube to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`mainRadius, minorRadius, angle, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddSphere`

```
int alAddSphere(
     double radius,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a sphere to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`radius` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddDome`

```
int alAddDome(
     double baseRadius,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a dome to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`baseRadius, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddTorus`

```
int alAddTorus(
     double mainRadius,
     double minorRadius,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a torus to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`mainRadius,minorRadius` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddConeSphere`

```
int alAddConeSphere(
     double baseRadius,
     double topRadius,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a conesphere to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`baseRadius, topRadius, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddConeTwoSpheres`

```
int alAddConeTwoSpheres(
     double baseRadius,
     double topRadius,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a conetwospheres to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`baseRadius, topRadius, height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddTentCylinder`

```
int alAddTentCylinder(
     double width,
     double baseLength,
     double topLength,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a tentcylinder to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`width,baseLength,topLength,height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddTentTwoCylinders`

```
int alAddTentTwoCylinders(
     double width,
     double baseLength,
     double topLength,
     double height,
     COLORREF color,
     int visibility,
     int *figureId)
```

This function adds a tenttwocylinders to the auxiliary list of figures and returns a primitive identifier.

Parameters:

`width,baseLength,topLength,height` as defined in VRPD.
`color` is the color of the primitive in RGB scale.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`figureId` figure identifier in the list, returned by the function.

- `alAddOBFFile`

```
int alAddOBFFile(
     STRING obfFileName,
     int visibility,
     int *lastId)
```

This function adds the figures defined in an OBF file to the auxiliary list of figures and returns the primitive identifier of the last figure. The file can contain any kind of primitives.

Parameters:

`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`lastId` is last figure identifier in the list, returned by the function.

- `alAddPAFFile`

```
int alAddPAFFile(
     STRING pafFileName,
     int visibility,
     int *lastId)
```

This function adds the figures defined in a PAF file to the auxiliary list of figures and returns the primitive identifier of the last figure. The file can contain any kind of primitives. The operation frames of the part become frame figures in the auxiliary list of figures.

Parameters:

`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)
`lastId` is last figure identifier in the list, returned by the function.

- `alAddVDAFile`

```
int alAddVDAFile(
     STRING vdaFileName,
     int thickness,
     int curveSections,
     int surfaceSections,
     COLORREF color,
     int visibility,
     int *lastId)
```

This function adds the figures defined in a VDA file to the auxiliary list of figures and returns the primitive identifier of the last figure. The file can contain surfaces, curves, circular arcs and points made with VDA_CURVE, VDA_SURF, DISK and 3DPOINT primitives.

Parameters:

`thickness, curveSections, surfaceSections` as defined in VRPD.

`color` is the color of the primitive in RGB scale.

`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)

`lastId` is last figure identifier in the list, returned by the function.

- `alGetFigureDimensions`

```
int alGetFigureDimensions(
     int figureId,
     int *figureType,
     double figureDims[12])
```

This function gives the figure type in `figureType` and the figure dimensions in `figureDims` of the figure specified in `figureId`. This function cannot be used with any of the figures created with a VDA file (an error is returned). If the figure does not exists, an error is returned. The figure type and the dimensions are as shown in the following table:

FigureType (defined with constant `FIGURE_TYPE_*`)	**figureDims[.]**											
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
FRAME	size											
3DPOINT	x	y	z									
LINE	x1	y1	z1	x2	y2	z2						
DISK	mainradius	minorradius	angle									
TRIANGLE	x1	y1	z1	x2	y2	z2	x3	y3	z3			
3DFACE	x1	y1	z1	x2	y2	z2	x3	y3	z3	x4	y4	z4
BOX	length	width	height									
PYRAMID	length	basewidth	topwidth	height								
TRIANGULAR PYRAMID	baseedge	topedge	height									
TENT	baselength	basewidth	toplength	topwidth	height							
WEDGE	baselength	width	toplength	height								
CONE	baseradius	topradius	height									
TUBE	mainradius	minorradius	angle	height								
SPHERE	radius											
DOME	baseradius	height										
TORUS	mainradius	minorradius										
CONE_SPHERE	baseradius	topradius	height									
CONE_TWO_SPHERES	baseradius	topradius	height									
TENT_CYLINDER	width	baselength	toplength	height								
TENT_TWO_CYLINDERS	width	baselength	toplength	height								

- `alSetFigureDimensions`

```
int alSetFigureDimensions(
     int figureId,
     double figureDims[12])
```

This function sets the figure dimensions of the figure specified in `figureId` to the values specified in `figureDims`. This function cannot be used with any of the figures created with a VDA file (an error is returned). If the figure does not exists, an error is returned. The figure

dimensions according to the figure type are as shown in the previous table.

- `alDelFigure`

```
int alDelFigure(
     int figureId)
```

This function deletes a figure of the auxiliary list of figures.

Parameter:

`figureId` is the figure identifier in the list.

- `alDelFigureList`

```
int alDelFigureList()
```

This function deletes all the figures of the auxiliary list of figures.

Parameter:

No parameter is required.

- `alInitFigureTransformation`

```
int alInitFigureTransformation(
     int figureId)
```

This function initializes the transformation of a figure.

Parameter:

`figureId` is the figure identifier in the list.

- `alGetFigureTransformation`

```
int alGetFigureTransformation(
     int figureId,
     double transformation[4][4])
```

This function obtains the transformation matrix of a figure.

Parameters:

`figureId` is the figure identifier in the list.

`transformation` is the 4x4 transformation matrix.

Options:

- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alSetFigureTransformation`

```
int alSetFigureTransformation(
     int figureId,
     double transformation[4][4])
```

This function applies a transformation matrix to a figure.

Parameters:

`figureId` is the figure identifier in the list.

`transformation` is a 4x4 Homogenous Transformation Matrix.

Options:

- The location can be managed as an array according to Appendix A.
- A different type for the Euler angles can be specified according to Appendix B.
- The location can also be managed as a transformation matrix, according to Appendix C.

- `alApplyFigureOperation`

```
int alApplyFigureOperation(
     int figureId,
     int kind,
     int axis,
     double value)
```

This function applies a basic transformation to a figure.

Parameters:

`figureId` is the figure identifier in the list.

`kind` is the kind of transformation to add in the list:

`TRANSLATION`	transformation of translation
`ROTATION`	transformation of rotation

`axis` is the axis on which the transformation must be applied:

`X_AXIS, Y_AXIS, Z_AXIS`	X,Y,Z axis of global frame
`U_AXIS, V_AXIS, W_AXIS`	U,V,W axis of local frame

`value` is the value of the transformation (in millimeters or degrees).

- `alGetFigureVisibility`

```
int alGetFigureVisibility(
       int figureId,
       int *visibility)
```

This function gives on `visibility` the visibility state for the figure indicated on `figureId`.

Parameters:

`figureId` is the figure identifier in the auxiliary list.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)

- `alSetFigureVisibility`

```
int alSetFigureVisibility(
        int figureId,
        int visibility)
```

This function sets the visibility state of a given figure in the auxiliary list of figures. Only the visible figures will be drawn.

Parameters:

`figureId` is the figure identifier in the list.
`visibility` is the figure visibility state (`VISIBLE`, `INVISIBLE`)

- `alGetFigurecolor`

```
int alGetFigurecolor(
     int figureId,
     COLORREF *color)
```

This function gives the color of a figure in the auxiliary list of figures.

Parameters:

`figureId` is the figure identifier in the list.
`color` returns the color of the primitive in RGB scale.

- `alSetFigurecolor`

```
int alSetFigurecolor(
     int figureId,
     COLORREF color)
```

This function sets the color of a given figure to an RGB value.

Parameters:

`figureId` is the figure identifier in the auxiliary list of figures.
`color` is the color in RGB scale.

- `alGetFigureRenderMode`

```
int alGetFigureRenderMode(
     int figureId,
     int *renderMode)
```

This function obtains the render mode of a figure.

Parameters:

figureId is the figure identifier in the list.
renderMode is the render mode (WIRED, SHADE or HIDDEN).

- alSetFigureRenderMode

```
int alSetFigureRenderMode(
     int figureId,
     int renderMode)
```

This function defines the render mode of a figure. Independently of the render configuration, the figure will be drawn always in this render mode.

Parameters:

figureId is the figure identifier in the list.
renderMode is the render mode (WIRED, SHADE or HIDDEN).

- alGetNumberOfFigures

```
int alGetNumberOfFigures(int *numberOfFigures)
```

This function returns the number of figures in the auxiliary list of figures.

- alSetFigureEffects

```
int alSetFigureEffects(
     int figureId,
     bool blend, int srcFnc, int dstFnc,
     STRING textureFile, int envFnc, STRING maskFile)
```

This function sets the visual effects of a figure in the auxiliary list of figures specified with figureId. The parameter meaning is as follow:

- If blend is true, the figure becomes a transparent figure, applying srcFnc and dstFnc as source and destination functions as transparency parameters. If blend is false, the figure becomes opaque. The transparency parameters srcFnc and dstFnc can be:
  GL_ZERO, GL_DST_COLOR, GL_SRC_COLOR,
  GL_ONE_MINUS_DST_COLOR, GL_ONE_MINUS_SRC_COLOR,
  GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA,
  GL_ONE_MINUS_DST_ALPHA, GL_SRC_ALPHA_SATURATE
- The textureFile parameter, if not empty, must be a BMP file with path related to VRS path (spaces must be avoided in file name and path) of a texture applied to the figure when the figure is displayed in RENDERED mode. The texture file will be applied to every face of the primitive. This function cannot be applied to the torus. It is highly recommended to have all textures in path *Models\Textures*.

- The `envFnc` parameters gives the texture application mode. When a texture file is specified for a primitive, the texture application mode can be specified with one of the following values:
  `GL_DECAL, GL_MODULATE, GL_BLEND, GL_REPLACE`
- The `maskFile` parameter, if not empty, must be a monochrome BMP file with path related to VRS path (spaces must be avoided in file name and path) of a mask. When the primitive is displayed in RENDERED mode with a texture file applied, this mask file will be applied as a mask. This function cannot be applied to the torus. It is highly recommended to have all masks in path *Models\Textures*.

  The parameters used are as defined in OPEN-GL. Please refer to OPEN-GL reference books for more information.

- `alMoveFiguresToObject`

```
int alGetFigureEffects(
     int figureId,
     bool *blend, int *srcFnc, int *dstFnc,
     STRING textureFile, int *envFnc, STRING maskFile)
```

  This function obtains the visual effects of a figure in the auxiliary list of figures specified with `figureId`. The parameter meaning is that of previous function.

- `alMoveFiguresToObject`

```
int alMoveFiguresToObject(
     STRING objectName,
     int *objectId)
```

  This function moves the auxiliary list of figures to the environment as a new object with the name specified in `objectName`. The frames in the auxiliary list of figures will not be moved. The auxiliary list of figures becomes empty. The object identifier is returned on the last parameter.

- `alCopyFiguresToObject`

```
int alCopyFiguresToObject(
     STRING objectName,
     int *objectId)
```

  This function copies the auxiliary list of figures to the environment as a new object with the name specified in `objectName`. The frames in the auxiliary list of figures will not be copied. The auxiliary list of figures is not modified. The object identifier is returned on the last parameter.

- `alMoveFiguresFromObject`

  ```
  int alMoveFiguresFromObject(int objectId)
  ```

  This function moves the primitives in the object of the environment specified in the parameter to the auxiliary list of figures. All the previous figures in the auxiliary list are not modified. The object is deleted. An error is returned if there is no object with this identifier.

- `alCopyFiguresFromObject`

  ```
  int alCopyFiguresFromObject(int objectId)
  ```

  This function copies the primitives in the object of the environment specified in the parameter to the auxiliary list of figures. All the previous figures in the auxiliary list are not modified. The object is not modified. An error is returned if there is no object with this identifier.

- `alMoveFiguresToPart`

  ```
  int alMoveFiguresToPart(STRING partName, int *partId)
  ```

  This function moves the auxiliary list of figures to the environment as a new part with the name specified in `partName`. All the frames in the auxiliary list of figures will be copied as operation frames of the part. If there is no frame on the list of figures, an operation frame is created with identity transformation. The auxiliary list of figures becomes empty. The part identifier is returned on the last parameter.

- `alCopyFiguresToPart`

  ```
  int alCopyFiguresToPart(STRING partName, int *partId)
  ```

  This function copies the auxiliary list of figures to the environment as a new part with the name specified in `partName`. All the frames in the auxiliary list of figures will be copied as operation frames of the part. If there is no frame on the list of figures, an operation frame is created with identity transformation. The auxiliary list of figures is not modified. The part identifier is returned on the last parameter.

- `alMoveFiguresFromPart`

  ```
  int alMoveFiguresFromPart(int partId)
  ```

  This function moves the primitives in the part of the environment specified in the parameter to the auxiliary list of figures. All the previous figures in the auxiliary list of figures are not modified. The operation frames of the part become frame figures in the auxiliary list of figures.

The part is deleted. An error is returned if there is no part with this identifier.

- `alCopyFiguresFromPart`

  ```
  int alCopyFiguresFromPart(int partId)
  ```

  This function copies the primitives in the part of the environment specified in the parameter to the auxiliary list of figures. All the previous figures in the list are not modified. The operation frames of the part become frame figures in the auxiliary list of figures. The part is not modified. An error is returned if there is no part with this identifier.

- `alSaveAsObject`

  ```
  int alSaveAsObject(STRING obfFileName)
  ```

  This function saves the auxiliary list of figures as an object file (OBF). An error is returned if there is no figure in the auxiliary list of figures. The file name starts from *VR-Path*.

- `alSaveAsPart`

  ```
  int alSaveAsPart(STRING pafFileName)
  ```

  This function saves the auxiliary list of figures as a part file (PAF). An error is returned if there is no figure in the auxiliary list of figures. The file name starts from *VR-Path*.

20. Functions for Display

This set of functions mainly allows the manage of different aspect for display. Usually the display is controlled by means of *VRS* options in such way that while the user application is running, the user can manage the display from the available options of *VRS*. With the functions explained in this section, the user application can take the control of the display and manage some parameters for this purpose. This will be possible only if the function `alGetDisplayControl` has been successfully executed. The `viewport` parameter specifies the viewport used for the function (possible values are defined with constants `FIRST_VIEWPORT`, …, `FOURTH_VIEWPORT`).

The functions are:

- `alGetDisplayControl`

```
int alGetDisplayControl(
     int viewport)
```

  This function gives the display control to the user application by means of VREAL functions. The following actions are executed on *VRS* before the display control is given to the VREAL library:
  - The display is configured with only one window
  - The display is initialized in perspective and shaded display.
  - The rest of parameters (point of view, reference point, …) are decided by *VRS*
  - The following display options become disabled on *VRS*:
    - 1 Window, 2 Horizontal Windows, 2 Vertical Windows, 4 Windows, Maximize
    - Wired, Shaded
    - Zoom, Scroll, Point of View, Reference Point
    - Perspective (Front, Back, Left, Right, Up, Down, Perspective)

- `alFreeDisplayControl`

```
int alFreeDisplayControl(
     int viewport)
```

  This function gives back the display control to *VRS* and the user can manage display by means of the available options on the menus which are enabled again. None of the following functions will produce any action after this function is called.

- `alGetCameraPosition`

```
int alGetCameraPosition(
     int viewport,
     double *x, double *y, double *z)
```

This function returns in `x,y,z` the camera position, i.e. the point of view from where the image is taken. The position is related to the world frame.

- `alSetCameraPosition`

```
int alSetCameraPosition(
     int viewport,
     double x, double y, double z)
```

This function defines the camera position in `x,y,z`, i.e. the point of view from where the image is taken. The position is related to the world frame. An error is returned if the camera is attached to a Tool Frame.

- `alGetPolarCameraPosition`

```
int alGetPolarCameraPosition(
     int viewport,
     double *tecta, double *fi, double *ro)
```

This function returns in `tecta,fi,ro` the polar coordinates of the camera position, i.e. the point of view from where the image is taken. `tecta` is the longitude, `fi` is the latitude and `ro` is the distance between the point of view and the reference point, having a zoom effect. The polar coordinates are related to the reference point.

- `alSetPolarCameraPosition`

```
int alSetPolarCameraPosition(
     int viewport,
     double tecta, double fi, double ro)
```

This function defines the polar coordinates of the camera position in `tecta,fi,ro`, i.e. the point of view from where the image is taken. `tecta` is the longitude, `fi` is the latitude and `ro` is the distance between the point of view and the reference point, having a zoom effect. The polar coordinates are related to the reference point. An error is returned if the camera is attached to a Tool Frame.

- `alGetPointRef`

```
int alGetPointRef(
     int viewport,
     double *x, double *y, double *z)
```

This function returns in `x,y,z` the coordinates of the reference point, i.e. where the camera is looking at. The position is related to the world frame.

- `alSetPointRef`

```
int alSetPointRef(
```

```
        int viewport,
        double x, double y, double z)
```

This function defines the coordinates of the reference point in `x,y,z`, i.e. where the camera is looking at. An error is returned if the camera is attached to a Tool Frame. The position is related to the world frame.

- `alGetCameraParameters`

```
int alGetCameraParameters(
        int viewport,
        int *nearPlane,
        int *farPlane,
        double *fov)
```

This function returns the following camera parameters:
- `nearPlane` is the near plane of the camera model
- `farPlane` is the far plane of the camera model
- `fov` is the field of view angle, in degrees

- `alSetCameraParameters`

```
int alSetCameraParameters(
        int viewport,
        int nearPlane,
        int farPlane,
        double fov)
```

This function defines the following camera parameters:
- `nearPlane` is the near plane of the camera model
- `farPlane` is the far plane of the camera model
- `fov` is the field of view angle, in degrees

- `alGetRenderConfig`

```
int alGetRenderConfig(
        int viewport,
        int *renderKind)
```

This function returns the current render configuration, which can be `WIRED`, `SHADED`, or `HIDDEN`.

- `alSetRenderConfig`

```
int alSetRenderConfig(
        int viewport,
        int renderKind)
```

This function defines the render configuration, which can be `WIRED`, `SHADED`, or `HIDDEN`.

- `alAttachCamera2ToolFrame`

```
int alAttachCamera2ToolFrame(
     int viewport,
     int robotId,
     int toolFrameId)
```

This function attaches the camera to the ToolFrame indicated on the second parameter of the robot indicated on the first parameter. An error is returned if the ToolFrame does not exist. The optic axis of the camera is assigned to the Z-axis of the ToolFrame and the vertical axis of the camera will be the Y-axis of the ToolFrame. A depth similar to far plane is assigned.

Option:
A depth parameter (to indicate distance from camera position to reference point) can be defined as fourth parameter with the following interface (if not specified, the default depth value is the far plane):

```
int alAttachCamera2ToolFrame(
     int viewport,
     int robotId,
     int toolFrameId,
     double depth)
```

- `alFreeCamera`

```
int alFreeCamera(
     int viewport)
```

This function frees the camera if it is attached. The camera will be restored as it was before attached.

- `alGetCameraDepth`

```
int alGetCameraDepth(
     int viewport,
     double *depth)
```

This function gives back the camera depth on the parameter. `depth` is the distance from camera position to reference point. This parameter is only used when the camera is attached. Its default value is the far plane.

- `alSetCameraDepth`

```
int alSetCameraDepth(
     int viewport,
```

```
        double depth)
```

This function defines the camera depth on the parameter. `depth` is the distance from camera position to reference point. This parameter is only used when the camera is attached. Its default value is the far plane.

- `alDisplayDynamicInfo`

```
int alDisplayDynamicInfo (
     int dynamicInfoVisible)
```

This function makes visible the dynamic info in *VRS* when the parameter is `VISIBLE` and invisible when the parameter is `INVISIBLE`.

21. Distance Functions

This set of functions is designed to compute distances to elements in VRS.

The functions are:

- `alDistanceToObject`

```
int alDistanceToObject(
        int objectId,
        double location[6],
        int *interfer,
        double *distance)
```

This function computes the distance to the object specified in `objectId` related to Z-axis of a location, returning the interference state and the distance. Possible result cases are specified in the following table:

Case	`interfer` value	`distance` value
There is no interference between Z-axis of location frame and object	`NOT_INTERFER`	0
The location frame is inside the object	`IS_INSIDE`	0
The object is in the negative direction of Z-axis of location frame	`IS_IN_BACK`	0
The object is in the positive direction of Z-axis of location frame	`IS_IN_FRONT`	distance value

- `alDistanceToPart`

```
int alDistanceToPart(
        int partId,
        double location[6],
        int *interfer,
        double *distance)
```

This function computes the distance to the part specified in `partId` related to Z-axis of a location, returning the interference state and the distance. Possible result cases are specified in the following table:

Case	`interfer` value	`distance` value
There is no interference between Z-axis of location frame and part	`NOT_INTERFER`	0
The location frame is inside the part	`IS_INSIDE`	0
The part is in the negative direction of Z-axis of location frame	`IS_IN_BACK`	0
The part is in the positive direction of Z-axis of location frame	`IS_IN_FRONT`	distance value

- `alDistanceToRobot`

```
int alDistanceToRobot(
        int robotId,
        double location[6],
        int *interfer,
        double *distance)
```

This function computes the distance to the robot specified in `robotId` related to Z-axis of a location, returning the interference state and the distance. Possible result cases are specified in the following table:

Case	`interfer` value	`distance` value
There is no interference between Z-axis of location frame and robot	`NOT_INTERFER`	0
The location frame is inside the robot	`IS_INSIDE`	0
The robot is in the negative direction of Z-axis of location frame	`IS_IN_BACK`	0
The robot is in the positive direction of Z-axis of location frame	`IS_IN_FRONT`	distance value

- `alGetClosestObject`

```
int alGetClosestObject(
        double location[6],
        int *objectId,
        double *distance)
```

This function computes the closest object related to Z-axis of a location. Possible result cases are specified in the following table:

Case	`objectId` value	`distance` value
No object is detected	`NOTHING`	0
The location frame is inside an object	`objectId`	0
At least an object is detected and the closest one is computed	`objectId`	distance value

- `alGetClosestPart`

```
int alGetClosestPart(
        double location[6],
        int *partId,
        double *distance)
```

This function computes the closest part related to Z-axis of a location. Possible result cases are specified in the following table:

Case	`partId` value	`distance` value
No part is detected	`NOTHING`	0
The location frame is inside a part	`partId`	0
At least a part is detected and the closest one is computed	`partId`	distance value

- `alGetClosestRobot`

```
int alGetClosestRobot(
        double location[6],
        int *robotId,
        double *distance)
```

This function computes the closest robot related to Z-axis of a location. Possible result cases are specified in the following table:

Case	`robotId` value	`distance` value
No robot is detected	`NOTHING`	0
The location frame is inside a robot	`robotId`	0
At least a robot is detected and the closest one is computed	`robotId`	distance value

Options:
A robot can be avoided to consider detection with an optional parameter, its identifier specified in `robotToAvoid`, with the following function interface:

```
int alGetClosestRobot(double location[6], int *robotId,
                      double *distance, int robotToAvoid)
```

- `alGetClosestElement`

```
int alGetClosestElement(
        double location[6],
        int *elementId,
        int *elementType,
        double *distance)
```

This function computes the closest element (object, part or robot) related to Z-axis of a location. Possible result cases are specified in the following table:

Case	elementId value	elementType value	distance value
No element is detected	NOTHING	NOTHING	0
The location frame is inside an object	objectId	IT_IS_OBJECT	0
The location frame is inside a part	partId	IT_IS_PART	0
The location frame is inside a robot	robotId	IT_IS_ROBOT	0
At least an element is detected and the closest one computed is an object	objectId	IT_IS_OBJECT	distance value
At least an element is detected and the closest one computed is a part	partId	IT_IS_PART	distance value
At least an element is detected and the closest one computed is a robot	robotId	IT_IS_ROBOT	distance value

Options:

A robot can be avoided to consider detection with an optional parameter, its identifier specified in `robotToAvoid`, with the following function interface:

```
int alGetClosestElement(double location[6],
            int *elementId, int *elementType,
            double *distance, int robotToAvoid)
```

22. Video Functions

This set of functions is designed to record videos[1].

The functions are:

- `alVideoStart`

```
int alVideoStart(
     STRING fileName)
```

  This function specifies the file name where a video is going to be recorded in an uncompressed AVI format. The function does not start video recording.

- `alVideoRecord`

```
int alVideoRecord()
```

  This function starts the video recording (or continue video recording if paused).

- `alVideoPause`

```
int alVideoPause()
```

  This function makes a pause on video recording.

- `alVideoEnd`

```
int alVideoEnd()
```

  This function ends video recording and closes the video file.

- `alGetImage`

```
int alGetImage(
     int sizeX, int sizeY,
     char *image)
```

  This function returns an image of size `sizeX` x `sizeY` specified as a matrix.

[1] Only for Windows 95/98/ME

23. Collision Check Functions

This set of functions is designed for collision check.

The functions are:

- `alStartCollisionCheck`

  ```
  int alStartCollisionCheck(int period)
  ```

  This function starts the collision check module with a period (in milliseconds). Collisions are checked between every robot and every object and part of the environment. Only those robot links, objects and parts defined with envelops are considered (see VRFFD). If a collision is detected, the background color is set to red color. When the collision check is activated, neither a robot nor the environment can be loaded or closed.

- `alEndCollisionCheck`

  ```
  int alEndCollisionCheck()
  ```

  This function ends the collision check module.

24. VREAL.INI File

The VREAL.INI file allows the configuration of the communication between *VRS* and the user's application. This file must be in the windows main directory.

The content of the file is as follow:

```
[GENERAL]
TimeOut=5
LoadTimeOut=40
;Communication Timeout in seconds
```

The TimeOut field means the timeout in seconds for application communication. The LoadTimeOut fields means the timeout in seconds for the load instructions. The second timeout usually will be greater than general timeout.

Appendix A. Location Over-Definition as an Array

In most of the functions where a location is required, six parameters are used. The six parameters can be grouped in a 6-component array. The definition of the function can be any of the following:

With six parameters	With array
..., double x, double y, double z, double alpha, double beta, double gamma, ...	..., double location[6], ...
..., double *x, double *y, double *z, double *alpha, double *beta, double *gamma, ...	..., double location[6], ...

The equivalence is given by the order, that is:

location[0] is x	location[1] is y	location[2] is z
location[3] is alpha	location[4] is beta	location[5] is gamma

This option is available on the following functions:

alGetRobotFrame	alSetRobotLocation
alPlaceRobot	alApproxToLocation
alGetEnvironmentFrame	alGetObjectLocation
alPlaceEnvironment	alSetObjectLocation
alGetRobotLocation	alGetPartLocation
alMoveRobot	alSetPartLocation
alMove	alGetPartOperationFrame
alGetFigureTransformation	alSetFigureTransformation

As an example, consider the first function, `alGetRobotFrame`, which can be called using any of the two following interfaces:

- `int alGetRobotFrame(int robotId, double *x, double *y, double *z, double *alpha, double *beta, double *gamma)`
- `int alGetRobotFrame(int robotId, double location[6])`

Please, note that the rest of the parameters must follow the location, as in `alApproxToLocation`:

```
int alApproxToLocation(int robotId, location[6],
int linearMovement, int frame, double xDistance,
double yDistance, double zDistance)
```

Appendix B. Euler Angles Type Selection

In any function where an orientation is specified with Euler angles the default type is 2 (also called ZYZ), that is, with the following rotations:

- Rotation of alpha angle related to Z axis
- Rotation of beta angle related to V axis
- Rotation of gamma angle related to W axis

A different Euler angle type can be specified with a new parameter on the function. The parameter `eulerType` is used and can take the following values (from Constant Definition section):

- `EULER_ANGLES_TYPE_1` for Euler angles type 1 (ZXZ: alpha for Z, beta for U, gamma for W)
- `EULER_ANGLES_TYPE_2` for Euler angles type 2
- `EULER_ANGLES_TYPE_3` for Euler angles type 3 (XYZ: alpha for X, beta for Y, gamma for Z)

This option is available on the following functions:

`alGetRobotFrame`	`alSetRobotLocation`
`alPlaceRobot`	`alApproxToLocation`
`alGetEnvironmentFrame`	`alGetObjectLocation`
`alPlaceEnvironment`	`alSetObjectLocation`
`alGetRobotLocation`	`alGetPartLocation`
`alMoveRobot`	`alSetPartLocation`
`alMove`	`alGetPartOperationFrame`
`alGetFigureTransformation`	`alSetFigureTransformation`

As an example, consider the first function, `alGetRobotFrame`, which can be called using this new parameter for any of the two available interfaces:

- `int alGetRobotFrame(int robotId, double *x, double *y, double *z, double *alpha, double *beta, double *gamma, int eulerType)`
- `int alGetRobotFrame(int robotId, double location[6], int eulerType)`

A special function for this option is `alPickPart`. For this function, a special orientation tolerance can be defined with optional parameters. Only when the six parameters are specified, a different Euler Angles Type can be used. In this case the function interface becomes according to:

```
int alPickPart(
      int robotId, int partId, int opFrameId, double toolStatus,
      int checkOpFrame, double xTolerance, double yTolerance,
      double zTolerance, double alphaTolerance, double betaTolerance,
      double gammaTolerance, int eulerType)
```

Appendix C. Location Over-Definition as Transformation

In most of the functions where a location is required, six parameters are used (even grouped as commented on Appendix A), representing three translation values for position and three angle values for orientation. An equivalent form to represent a location is by means of a 4x4 transformation matrix. The definition of the function can be any of the following:

With six parameters	With Transformation Matrix
..., double x, double y, double z, double alpha, double beta, double gamma, ...	..., double transformation[4][4], ...
..., double *x, double *y, double *z, double *alpha, double *beta, double *gamma, ...	..., double transformation[4][4], ...

This option is available on the following functions:

alGetRobotFrame	alSetRobotLocation
alPlaceRobot	alApproxToLocation
alGetEnvironmentFrame	alGetObjectLocation
alPlaceEnvironment	alSetObjectLocation
alGetRobotLocation	alGetPartLocation
alMoveRobot	alSetPartLocation
alMove	alGetPartOperationFrame
alGetFigureTransformation	alSetFigureTransformation

As an example, consider the first function, `alGetRobotFrame`, which can be called using any of the two following interfaces:

- `int alGetRobotFrame(int robotId, double *x, double *y, double *z, double *alpha, double *beta, double *gamma)`
- `int alGetRobotFrame(int robotId, double location[6])`
- `int alGetRobotFrame(int robotId, double transformation[4][4])`

Please, note that the rest of the parameters must follow the location, as in `alApproxToLocation`:

```
int alApproxToLocation(int robotId, transformation[4][4],
int linearMovement, int frame, double xDistance,
double yDistance, double zDistance)
```