**The NIST ISD Collaborative Robot Programming Interface**

**v 1.3**

**Updated 10 October, 2019**

**Table of Contents:**

1 Introduction: The NIST ISD Collaborative Robot Programming Interface 3

2 Data Structures 4

2.1 robotAxes 4

2.2 IO 4

2.3 robotPose 4

3 Creating a CRPI Robot Instance 5

3.1 Robot Types 5

3.2 Configuration XML Files 5

3.2.1 ROBOT 5

3.2.2 TCP\_IP 6

3.2.3 Serial 6

3.2.4 ComType 7

3.2.5 Mounting 7

3.2.6 ToWorld 7

3.2.7 CoordSystem 7

3.2.8 Tool 8

4 CRPI Syntax 9

4.1 ApplyCartesianForceTorque (robotPose robotForceTorque, vector<bool> activeAxes)) 9

4.2 ApplyJointTorque (robotAxes robotJointTorque) 9

4.3 Couple (char\* targetID) 9

4.4 CrclXmlHandler (char\* targetID) 9

4.5 CrclXmlResponse (char\* targetID) 9

4.6 CrpiXmlHandler (char\* targetID) 10

4.7 CrpiXmlResponse (char\* targetID) 10

4.8 FromSystem (char\* targetID) 10

4.9 FromWorld (robotPose \*in, robotPose \*out) 11

4.10 GetRobotAxes (robotAxes \*axes) 11

4.11 GetRobotForces (robotPose \*forces) 11

4.12 GetRobotIO (IO \*io) 11

4.13 GetRobotPose (robotPose \*pose) 11

4.14 GetRobotSpeed (robotAxes \*speeds) 12

4.15 GetRobotSpeed (robotPose \*speeds) 12

4.16 GetRobotTorques (robotAxes \*torques) 12

4.17 Message (char\* message) 12

4.18 MoveAttractor (robotPose pose) 12

4.19 MoveStraightTo (robotPose pose) 12

4.20 MoveThroughTo (robotPose\* poses, int numPoses, double\* accelerations, double\* speeds, double\* tolerances) 13

4.21 MoveTo (robotPose\* pose) 13

4.22 MoveToAxisTarget (robotAxes\* axes) 13

4.23 SaveConfig (char\* targetID) 14

4.24 SetAbsoluteAcceleration (double acceleration) 14

4.25 SetAbsoluteSpeed (double speed) 14

4.26 SetAngleUnits (char\* UnitName) 14

4.27 SetAxialSpeeds (double\* speed) 14

4.28 SetAxisUnits (char\*\* UnitNames) 14

4.29 SetEndPoseTolerance (Pose tolerances) 15

4.30 SetIntermediatePoseTolerance (robotPose\* tolerances) 15

4.31 SetLengthUnits (char\* UnitName) 15

4.32 SetParameter (char\* ParamName, void\* value) 15

4.33 SetRelativeAcceleration (double percent) 15

4.34 SetRelativeSpeed (double percent) 16

4.35 SetRobotDO (dig\_out, val) 16

4.36 SetRobotIO (IO \*io) 16

4.37 SetTool (double percent) 16

4.38 StopMotion (integer condition) 17

4.39 ToSystem (char\* targetID) 17

4.40 ToWorld (char\* targetID) 17

4.41 ToWorld (char\* targetID) 17

4.42 UpdateSystemTransform (char\* targetID) 18

4.43 UpdateSystemTransform (char\* targetID) 18

4.44 UpdateWorldTransform (char\* targetID) 18

4.45 UpdateWorldTransform (char\* targetID) 19

5 Installation Instructions 20

5.1 KUKA (KRC2 Controller) 20

5.1.1 Copying the source code to the controller 20

5.1.2 Running the CRPI client 20

5.2 ABB (IRC5 Controller) 20

5.2.1 Copying the source code to the controller 21

5.2.2 Installing the system modules 21

5.2.3 Running the CRPI Server 22

6 Example CRPI Program 23

# Introduction: The NIST ISD Collaborative Robot Programming Interface

Originally based on the Canonical Robot Command Language (CRCL), the NIST ISD Collaborative Robot Programming Interface (CRPI) is intended to provide an architecture to support the metrology of collaborative robot performance by means of commanding a myriad of robot platforms simultaneously with a singular command structure. The CRPI was developed to be an instantiation of the conceptualized model of the CRCL, and is subject to expansion, as real-world applications require additional functionality. Current development of the CRPI is focused on enabling and supporting application-driven multi-robot coordination and collaboration.

All functions provide an enumerated return type upon evaluation of the commanded action. This return type specifies when an action has been executed successfully or when there is an error. The return values are as follows:

|  |  |
| --- | --- |
| **Return Value** | **Description** |
| 0 – Success | The commanded function has been executed successfully. In terms of motion commands, the functions will return “success” when the robot is at or near (within tolerance) the specified configuration. |
| 1 – Failure | The commanded function has been executed but resulted in a failure. Failures include collisions during motion commands, missing parts during object scans, or illegal options for configuration changes (e.g., setting length units to “degree,” or setting tolerance values that the robot cannot achieve). |
| 2 – Reject | The robot rejected the commanded function prior to attempting execution. Rejections may include situations such as the robot being sent a motion command outside of the robot’s work volume, or being commanded to open its gripper when no gripper is attached. |
| 3 – Running | The robot is still executing a previously issued command. If the robot supports returning such a value, it may be used to assess whether the robot is currently capable of accepting any new commands, or may be used to identify erroneous functional states as the robot may be waiting for some event to occur. |

The coordinate frame for all Cartesian motion commands is that of the robot’s base. In multiple robot systems, there may be multiple bases. When it is necessary for the robots to communicate with one another for coordinated motions, they must be aware of where in space those bases are relative to their own bases. These origins may not be accurately known. It is therefore suggested that all Cartesian coordinates be given in reference to a common, known world space coordinate frame. Individual robots will know their origins relative to this world coordinate frame for local trajectory planning.

# Data Structures

A number of non-standard data structures are used throughout the CRPI library to facilitate the exchange and interpretation of data. These data structures are described presently:

## robotAxes

**Description**:

Data structure that captures the current or desired axis values for a robotic platform.

**Members**:

axis – An array of double-precision floating point values denoting current or desired axis values

axes – An integer specifying the size of the array, axis.

## IO

**Description**:

Data structure that encapsulates the digital and analog input and output values from the robot’s controller.

**Members**:

axis – An array of double-precision floating point values denoting current or desired axis values

axes – An integer specifying the size of the array, axis.

## robotPose

**Description**:

Data structure that captures the current or desired Cartesian pose for a robotic platform.

**Members**:

x – Cartesian X axis value

y – Cartesian Y axis value

z – Cartesian Z axis value

xrot – Rotation around the Cartesian X axis

yrot – Rotation around the Cartesian Y axis

zrot – Rotation around the Cartesian Z axis

status

turns

# Creating a CRPI Robot Instance

All robots are considered CRPI objects. To create a new CRPI robot instance, you must include the *crpi\_robot.h* header file, and the header file for the specific robot you wish to command. For instance, creating a handler for a KUKA LWR, include the *crpi\_kuka\_lwr.h* header. The CRPI robot is a templated class, and you must indicate which robot will be commanded:

crpi\_robot::CrpiRobot<*robot\_type*> rob(*path\_to\_config\_xml*);

Here, *robot\_type* is a specific class instance of the robot (in the case of a KUKA LWR, this is crpi\_robot::CrpiKukaLWR), and *path\_to\_config\_xml* is a plaintext string that points to the location of the robot’s configuration XML file.

## Robot Types

## Configuration XML Files

Each robot instantiation must have a corresponding XML configuration file that informs the CRPI how it should communicate with the robot. This XML file also includes information for coordinate space transformations, mounting information, and tool definitions. The contents of an example XML file are as follows:

<ROBOT>

<TCP\_IP Address="169.254.152.46" Port="30002" Client="true"/>

<Serial Port="COM7" Rate="57600" Parity="Even" SBits="1" Handshake="None"/>

<ComType Val="TCP\_IP"/>

<Mounting X="0" Y="0" Z="0" XR="135" YR="0" ZR="0"/>

<ToWorld X="236.72" Y="678.641" Z="-722.799" XR="0.508279" YR="-0.319192" ZR="81.3383" M00="0.150599" M01="-0.988563" M02="0.00793089" M03="236.72" M10="0.988579" M11="0.150546" M12="-0.00684316" M13="678.641" M20="0.00557093" M21="0.00887088" M22="0.999945" M23="-722.799" M30="0" M31="0" M32="0" M33="1"/>

<CoordSystem Name="Table1" X="2335.14" Y="471.0" Z="661.0" XR="0.0" YR="0.0" ZR="90.0" M00="0.0" M01="0.0" M02="0.0" M03="0.0" M10="0.0" M11="0.0" M12="0.0" M13="0.0" M20="0.0" M21="0.0" M22="0.0" M23="0.0" M30="0.0" M31="0.0" M32="0.0" M33="0.0"/>

<Tool ID="4" Name="gripper\_parallel" X="0" Y="0" Z="140" XR="0" YR="0" ZR="0" Mass="0.9" MX="0" MY="0" MZ="71"/>

<Tool ID="9" Name="gripper\_parallel\_plastic" X="0" Y="0" Z="108.5" XR="0" YR="0" ZR="0" Mass="0.9" MX="0" MY="0" MZ="71"/>

</ROBOT>

In this example, the robot has two communication mechanisms available for use, TCP/IP and serial, but is being told to connect via TCP/IP. In addition to a transformation to the world coordinate system, an additional coordinate system, “Table1” is defined. Also, two different tools are defined for use by this robot, gripper\_parallel and gripper\_parallel\_plastic. Each XML element and its arguments are described as follows:

### ROBOT

Encapsulation of the robot definition.

### TCP\_IP

Configuration information regarding TCP/IP communications with the robot.

#### Address

IP address of the robot. This element takes as an argument a string containing the IPv4 address of the target robot controller.

#### Port

IP socket port to either connect to or open, depending if the host PC is connecting to the robot as a client, or is acting as a server to which the robot connects. This element takes as an argument a character string containing the specified port number.

#### Client

Whether the host PC is connecting to the robot as a client. This element takes as an argument a character string specifying either true or false. If “true,” the PC will establish communications with the robot at the specified IP address and socket port. If “false,” the PC will create a TCP/IP server, open the specified socket port, and await a connection originating from the robot.

### Serial

Configuration information regarding serial communications with the robot.

#### Port

Communications port on the PC, through which the serial connection is established. Ports are specified as “COM1,” “COM2,” etc.

#### Rate

Transmission baud rate.

#### Parity

Set the parity bit to be either “Even” or “Odd.”

#### SBits

Set the number of bits in the transmission terminator. Valid values are “1” and “2.”

#### Hand-shake="None"/>

*[Not currently used]* Set the hardware flow control signal.

### ComType

Specify which protocol (Serial or TCP\_IP) to use to initiate communications with the robot.

#### Val

Identify the communication protocol, TCP\_IP or Serial.

### Mounting

Specify the mounting configuration of the robot to compensate for orientation and translation adjustments.

#### X, Y, Z, XR, YR, ZR

Cartesian location and orientation offsets.

### ToWorld

Transformation from the robot’s base coordinate system to the world coordinate system origin.

#### X, Y, Z, XR, YR, ZR

Cartesian transformation from the robot’s base coordinate system to the world coordinate system center.

#### M00, M01, …, M33

*[Optional]* The 4x4 homogeneous transformation matrix from the robot’s base coordinate system to the world coordinate system center. If the Cartesian transformation is provided in the XML file, the 4x4 transformation matrix is automatically generated on first execution, and the XML file is updated. If the transformation changes, either use the UpdateWorldTransform function, or remove M00 through M33 manually, modify the Cartesian transformation, and then re-load the XML file.

### CoordSystem

Define a transformation from the robot’s base coordinate system to a separate defined coordinate system origin.

#### Name

Name by which the coordinate system is referenced.

#### X, Y, Z, XR, YR, ZR

Cartesian transformation from the robot’s base coordinate system to the world coordinate system center.

#### M00, M01, …, M33

*[Optional]* The 4x4 homogeneous transformation matrix from the robot’s base coordinate system to the named coordinate system center. If the Cartesian transformation is provided in the XML file, the 4x4 transformation matrix is automatically generated on first execution, and the XML file is updated.

### Tool

End-of-arm tooling definition.

#### ID

Identification number of the tool (may be robot specific).

#### Name="gripper\_parallel"

Unique identifying name for the tool, by which the tool may be specified and its parameters sent to the robot controller via the Couple command.

#### X, Y, Z, XR, YR, ZR

Cartesian transformation from the robot’s tool flange to the tool center point.

#### Mass

Mass (in kg) of the tool.

#### MX, MY, MZ

Center of mass (in mm) of the tool, relative to the tool flange connection.

# CRPI Syntax

Alphabetically, the commands’ functionalities are as follows.

QQQQQQ

## ApplyCartesianForceTorque (robotPose robotForceTorque, vector<bool> activeAxes))

**Functionality**:

Submit 6 DoF Cartesian force and torque commands at the TCP, expressed in the robot base coordinate system. The axes for active force and torque control can be toggled, enabling the possibility for hybrid position and force control. **Parameters**:

robotForceTorque – The target 6 DoF Cartesian force-torque command

activeAxes – Flags for each axis that enable or disable force control

QQQQQQ

## ApplyJointTorque (robotAxes robotJointTorque)

**Functionality**:

Submit joint-level torque commands to the robot.

**Parameters**:

robotJointTorque – The target joint torque specified for every axis of the robot.

## Couple (char\* targetID)

**Functionality**:

Configure the robot with a specified tool or gripper. The couple command sets the robot’s TCP to the defined tool point, and sends mass information for gravity compensation purposes. The target tool’s name should match with an entry in the associated XML configuration file for the robot.

**Parameters**:

targetID – The name of the object with which the robot should dock.

QQQQQQ

## CrclXmlHandler (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

## CrclXmlResponse (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQ

## CrpiXmlHandler (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQ

## CrpiXmlResponse (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQ

## FromSystem (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQ

## FromWorld (robotPose \*in, robotPose \*out)

**Functionality**:

Transforms the incoming 6 DoF Cartesian pose of the robot as expressed in the World coordinate system to the robot’s base coordinate system.

**Parameters**:

in – The current 6 DoF Cartesian pose of the robot as expressed in the World’s coordinate system.

out – The current 6 DoF Cartesian pose of the robot as expressed in the robot’s base coordinate system.

## GetRobotAxes (robotAxes \*axes)

**Functionality**:

Get feedback from the robot regarding its current robot axis configuration. Axis values are reported in axial units.

**Parameters**:

axes – The current collection of axis values, populated by the function.

## GetRobotForces (robotPose \*forces)

QQQQQ

**Functionality**:

Get feedback from the robot regarding the 6 DoF Cartesian forces and torques emitted by the end-effector on the environment.

**Parameters**:

forces – The current 6 DoF Cartesian forces and torques, populated by the function.

## GetRobotIO (IO \*io)

**Functionality**:

Get feedback from the robot’s digital and analog inputs.

**Parameters**:

io – The collection of digital and analog inputs, populated by the function.

## GetRobotPose (robotPose \*pose)

**Functionality**:

Get feedback from the robot regarding its current position in Cartesian space. Location and orientation are reported in terms of length and angle units, as specified in SetLengthUnits and SetAngleUnits, respectively.

**Parameters**:

pose – The current 6DOF pose for the robot’s TCP in Cartesian space coordinates, populated by the function.

QQQQQQ

## GetRobotSpeed (robotAxes \*speeds)

**Functionality**:

Get feedback from the robot regarding its current robot axis configuration. Axis values are reported in axial units.

**Parameters**:

axes – The current collection of axis values, populated by the function.

QQQQQQQ

## GetRobotSpeed (robotPose \*speeds)

**Functionality**:

Get feedback from the robot regarding its current robot axis configuration. Axis values are reported in axial units.

**Parameters**:

axes – The current collection of axis values, populated by the function.

QQQQQQ

## GetRobotTorques (robotAxes \*torques)

**Functionality**:

Get feedback from the robot regarding its current robot axis configuration. Axis values are reported in axial units.

**Parameters**:

axes – The current collection of axis values, populated by the function.

## Message (char\* message)

**Functionality**:

Display a message on the operator console.

**Parameters**:

message – The plain-text message to be displayed on the operator console.

## MoveAttractor (robotPose pose)

**Functionality**:

Move a virtual attractor to a specified coordinate in Cartesian space. The virtual attractor command starts force-based, potential field, or virtual force-based motion control of the robot if it is not already on. Specialized motion control is subsequently deactivated when a normal Cartesian or axial motion command is issued to the robot.

**Parameters**:

pose – The target 6DOF pose for the robot’s virtual attractor in Cartesian space coordinates.

## MoveStraightTo (robotPose pose)

**Functionality**:

Move the controlled point in a straight line from the current pose to the given pose, and stop there.

**Parameters**:

pose – The target 6DOF pose of the TCP in Cartesian space

## MoveThroughTo (robotPose\* poses, int numPoses, double\* accelerations, double\* speeds, double\* tolerances)

**Functionality**:

Move the controlled point along a trajectory passing through or near all but the last of the given poses, and stop at the last of the given poses. The MoveThroughTo function generates a trajectory based on a least-squares cubic spline fit to the target poses. Defining accelerations, speeds, and intermediate and end pose tolerances in this function will not overwrite the defined default values.

**Parameters**:

poses – An array of 6DOF poses through/near which the robot is expected to pass

numPoses – The number of poses in the submitted array

accelerations – (optional) An array of acceleration profiles for each motion associated with the target poses. If the array of length numPoses is not provided, assumes system-wide accelerations are used.

speeds – (optional) An array of speed profiles for each motion associated with the target poses. If the array of length numPoses is not provided, assumes system-wide speeds are used.

tolerances – (optional) An array of 6DOF tolerances in length and angle units for the specified target poses. If the array of length numPoses is not provided, assumes system-wide tolerances are used.

## MoveTo (robotPose\* pose)

**Functionality**:

Move the controlled point along any convenient trajectory from the current pose to the given pose, and stop there.

**Parameters**:

pose – The target 6DOF pose for the robot’s TCP in Cartesian space coordinates

## MoveToAxisTarget (robotAxes\* axes)

**Functionality**:

Move the robot axes to the specified values. When applicable, each axis moves at the speeds set using the SetAbsoluteAxisSpeed or SetRelativeAxisSpeed command, to the target configuration. As applied to mobile robots with, for example, skid steer or Ackerman steering with speed control, a mapping of array values to the speed and steering outputs must be defined. If target axis values are beyond the physical capabilities of the robot, the command is rejected.

**Parameters**:

axes – An array of target axis values specified in the current axial unit.

QQQQQ

## SaveConfig (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

## SetAbsoluteAcceleration (double acceleration)

**Functionality**:

Set the acceleration for the controlled point to the given value in length units per second per second.

**Parameters**:

acceleration – The target TCP acceleration in length/rotation.

## SetAbsoluteSpeed (double speed)

**Functionality**:

Set the speed for the controlled point to the given value in length units per second.

**Parameters**:

speed – The target TCP speed.

## SetAngleUnits (char\* UnitName)

**Functionality**:

Set angle units to the unit named by the UnitName. All commands that use angle units (for orientation or orientation tolerance) are in terms of those angle units. Existing values for orientation are converted automatically to the equivalent value in new angle units. The default angle unit is "degree".

**Parameters**:

UnitName – The name of the angle units in plain text. Available options include “degree” and “radian.”

## SetAxialSpeeds (double\* speed)

**Functionality**:

Set the axis-specific speeds for the motion of axis-space motions to the given value in axial units per second.

**Parameters**:

speed – The array of target axial motion speeds.

## SetAxisUnits (char\*\* UnitNames)

**Functionality**:

Set specific axial units to the units named by the UnitNames. All axis motion commands are given in terms of those axial units. Available units include “radian,” “degree,” and “percentage” (which is the percentage of the total range between the axis’ minimum and maximum value; e.g., speed, current, or torque). The default length unit is “degree.”

**Parameters**:

UnitNames – The array of axis-specific names of the axial units in plain text.

## SetEndPoseTolerance (Pose tolerances)

**Functionality**:

Set the default 6DOF tolerances for the pose of the TCP to the given value in current length and angle units.

**Parameters**:

tolerance – Set of user-defined tolerances of the 6DOF pose of the TCP at the end of a move command

## SetIntermediatePoseTolerance (robotPose\* tolerances)

**Functionality**:

Set the default 6DOF tolerance for smooth motion near intermediate points to the given value in current length and angle units.

**Parameters**:

tolerance – An array of user-defined tolerances of the 6DOF poses during MoveThroughTo motions

## SetLengthUnits (char\* UnitName)

**Functionality**:

Set length units to the unit named by the UnitName. All commands that use length units (for location, tolerance, speed, and acceleration) are given in terms of those length units. Existing values for speed, position, acceleration, etc. are converted automatically to the equivalent value in new length units. The default length unit is millimeters, "mm".

**Parameters**:

UnitName – The name of the length units in plain text. Available options include “inch,” “mm,” and “meter.”

## SetParameter (char\* ParamName, void\* value)

**Functionality**:

Set specified parameters to the given values. Parameter names are robot specific, and must be accounted for in local documentation. The parameter values must be cast to the appropriate value within the function definition.

**Parameters**:

ParamName – The name of the variable to be set.

value – The new value to be set to the specified variable

## SetRelativeAcceleration (double percent)

**Functionality**:

Set the acceleration for the controlled point to the given percentage of the robot's maximum acceleration.

**Parameters**:

percent – The percentage of the robot’s maximum acceleration in the range of [0, 1]

## SetRelativeSpeed (double percent)

**Functionality**:

Set the speed for the controlled point to the given percentage of the robot's maximum speed.

**Parameters**:

percent – The percentage of the robot’s maximum speed in the range of [0, 1]

QQQQQQQ

## SetRobotDO (dig\_out, val)

**Functionality**:

Get feedback from the robot regarding its current robot axis configuration. Axis values are reported in axial units.

**Parameters**:

axes – The current collection of axis values, populated by the function.

## SetRobotIO (IO \*io)

**Functionality**:

Set the outputs on the robot controller’s digital and analog outputs.

**Parameters**:

io – The collection of digital and analog outputs to be set.

## SetTool (double percent)

**Functionality**:

Set the attached tool to a defined output rate. Different tool classes can be addressed through this function. Some example interpretations are as follows:

|  |  |  |
| --- | --- | --- |
| **Tool class** | **Input range** | **Parameter Interpretation** |
| Pneumatic gripper | [-1, 1] | Values < 0 close the gripper, while values > 0 open the gripper. A value of 0 indicates no air input. |
| Position-addressable gripper or onboard equipment | [0, 1] | Value indicates target open state as a percentage of “fully open/extended” (1.0). |
| Variable air output | [-1, 1] | Values > 0 indicate air pressure as percentage of maximum pressure output. Values < 0 indicate partial vacuum as percentage of maximum vacuum output. A value of 0 indicates neutral/no output. |
| Rotary tool | [0, 1] | Value indicates spindle rotation as a percentage of maximum rotational speed. |

**Parameters**:

percentage – The desired output rate for the robot’s tool as a percentage of the maximum output.

## StopMotion (integer condition)

**Functionality**:

Stop the robot’s motions based on predefined stopping rules. These stopping rules are as follows:

|  |  |
| --- | --- |
| **Condition** | **Description** |
| 0 – Stop category 0 | The robot’s drives are deactivated immediately and the brakes are applied. |
| 1 – Stop category 1 | The robot and any external axes are brought to a fast, controlled stop. The drives are deactivated after 1 s, and the brakes are applied. |
| 2 – Stop category 2 | The robot and any external axes are stopped using a normal braking ramp. The drives are not deactivated, and the brakes are not applied. |

**Parameters**:

condition – The rules by which the robot is expected to stop.

## ToSystem (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQ

## ToWorld (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQQ

## ToWorld (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQQ

## UpdateSystemTransform (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQ

## UpdateSystemTransform (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQ

## UpdateWorldTransform (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

QQQQQQ

## UpdateWorldTransform (char\* targetID)

**Functionality**:

Dock with a specified target object (e.g., tool, gripper, or station). The Couple command performs a physical, electrical, and/or software connection action based on the known capabilities and requirements for the defined object. The robot must be in an appropriate position with respect to the specified object for the Couple action to be successful. The robot rejects the Couple function if it is not capable of coupling with the specified object, if it is not in an appropriate coupling position, or if it does not know how to couple with the specified object (e.g., if no docking procedure is defined for the object, or if the object is to be grasped rather than physically attached to the robot).

**Parameters**:

targetID – The name of the object with which the robot should dock

# Installation Instructions

Control programs are included in the CRPI repository that are used by certain robot systems to enable the use of the CRPI at the application layer. Note that not all robots require local software to be run on the controller.

## KUKA (KRC2 Controller)

JAM: TO DO: This is barebones. Do it right, Jeremy!

Assumes the XML for KRL option installed

Assumes the Serial (CREAD\_CWRITE) option installed

BACK UP ALL FILES ON YOUR CONTROLLER FIRST!

### Copying the source code to the controller

Copy the contents of the INIT, Program, and System folders to the respective folders on the KRC2 controller.

Open the XmlApiConfig.xml in the INIT folder. Modify the entry TCPI\_IP to use the IP address of the computer running the KUKA CRPI server.

Restart the controller.

### Running the CRPI client

To run the program, using the navigation pane, select Program->SerialCom->RUNFILE. Switch to Automatic mode, enable the drives, and press the Play button.

## ABB (IRC5 Controller)

The RAPID code that defines the CRPI server running on the ABB IRC5 controller was written with the assumption that the robot(s) being used are 7 degrees-of-freedom (DOF). This code has not been tested on robots with 6DOF or fewer.

It is assumed the ABB IRC5 controller is already configured for normal operation. If not, please refer to your robot’s Operating Manual, provided when you purchased your robot (or contact ABB support to acquire this manual if you do not currently have one).

It is also assumed that the following controller options are installed:

623-1 Multitasking

672-1 Socket Messaging or 616-1 PC Interface

For multi-arm systems, you will also need

604-1 MultiMove Coordinated

To determine which options are currently installed, using the FlexPendant, open the menu and select the Control Panel option. From the Control Panel, select the Installed Systems option. Select the robot(s) on the left, and verify the options are installed.

### Copying the source code to the controller

The following files are included in the CRPI repository:

CRPI\_Handler\_Left.mod

CRPI\_Handler\_Right.mod

For dual-arm systems using the MultiMove controller option (e.g., the ABB IRB14000 “YuMi” robot), these files must be copied to the ABB IRC5 controller via a memory stick using the USB port on the FlexPendant. Single-arm systems may choose either the CRPI\_Handler\_Left.mod or CRPI\_Handler\_Right.mod file. For the sake of simplicity, we assume the CRPI\_Handle\_Left.mod file is used in single-arm systems.

Note that the repository also includes CRPI\_StateServer\_Left.mod and CRPI\_StateServer\_Right.mod files. These are used to provide state updates via a separate socket connection, and are not required for the CRPI to function. Installing these options can be accomplished by modifying the steps in the following subsection, substituting the respective file names as necessary.

### Installing the system modules

Before the code can be used by the robot, task modules must be created using the source code copied to the IRC5 controller.

On the FlexPendant, open the menu and select the **Control Panel** option. From the **Control Panel**, select the **Configuration** option.

At the bottom of the screen, use to **Topics** list to select the **Controller** option. Open the **Task** screen (either by double-pressing the **Task** object, or by selecting **Task** and then pressing the **Show All** button).

Each arm controlled by the IRC5 controller requires a motion task to operate. It is likely your controller already has a robot task created (T\_ROB1 for single-arm systems, and T\_ROB\_L and T\_ROB\_R for the YuMi). This task will need to be modified to accommodate the CRPI module. If this task does not already exist, create a new task by selecting the Add button.

Modify the motion task(s) as follows:

Set *Main entry* as “CRPI\_Main\_Left” (or “CRPI\_Main\_Right” for the right arm in a two-arm system).

If you intend to use the CRPI\_StateServer modules, create new non-motion tasks with the following:

*Name*: CRPI\_StateServer\_Left (or CRPI\_StateServer\_Right)

*Type*: SEMISTATIC

*Main entry*: CRPI\_State\_Left (or CRPI\_State\_Right)

Check unsolved references: 1

*Trust Level*: NoSafety

*MotionTask*: NO

*Use Mechanical Unit Group*: rob\_l (or rob\_r)

*Hidden*: NO

*RMQ Type*: None

*RMQ Mode*: Interrrupt

Once these modifications are complete, press the **Back** button to return to the **Controller** topic.

Open the **Automatic loading of Modules** screen. Create a new entry for each robot:

HOME:/CRPI/CRPI\_Handler\_Left.mod, and, for two-arm systems

HOME:/CRPI/CRPI\_Handler\_Right.mod

Set the *Task* entry to T\_ROB1 (or T\_ROB\_L or T\_ROB\_R, depending on the robot motion task).

Also create entries for the CRPI\_StateServer modules, if they are being used.

When all entries have been made, select the **YES** option to restart the controller and apply the changes.

### Running the CRPI Server

For each robot, ensure the controller is set to run the programs in continuous mode. Consult the FlexPendant Manual for instructions on switching between continuous and step modes. The StateServer modules, if used, automatically start on the controller’s power-up.

In the **Production Window** (or **Program Editor**) screen, press the “**PP to Main**” soft key to move the program pointer to the main entry point for each arm.

***IMPORTANT: Make sure the work area is clear of all personnel and obstructions prior to operating the robot.***

To run the server in manual mode:

1. Hold the FlexPendant **Enable Switch** (drives should power on, see notes below)
2. Press the **PLAY** soft key on the FlexPendant to run the program
3. Execute the desired motion from the CRPI PC application

Note that while operating in manual mode, the enabling switch must be held continuously for the robot to move.

***IMPORTANT: On the YuMi, the enable switch is disabled in system software, and does not impact the execution of the program. As such, do not rely on the enabling device for operator safety.***

To run the server in automatic mode:

1. Press the **Drives On** button on the controller (for the YuMi, this is accomplished through the FlexPendant)
2. Press the **PLAY** soft key on the FlexPendant to run the program
3. Execute the desired motion from the CRPI PC application

# Example CRPI Program

#include <stdlib.h>

#include <iostream>

#include <time.h>

#include "crpi\_robot.h"

#include "crpi\_universal.h"

#include "ulapi.h"

using namespace crpi\_robot;

using namespace std;

void main()

{

//! Create the robot object

CrpiRobot<CrpiUniversal> arm("universal\_ur5.xml");

//! Configure the default units

arm.SetAngleUnits("degree");

arm.SetLengthUnits("mm");

//! Couple the robot with the parallel gripper tool

arm.Couple("gripper\_parallel");

//! Temporary storage for current and target poses

robotPose curPose, newPose;

//! Get the current pose of the robot

arm.GetRobotPose(&curPose);

curPose.print();

//! Set a new pose target for 10 mm higher than the current position

newPose = curPose;

newPose.z += 10.0f;

//! Move robot

if (arm.MoveStraightTo (newPose) == CANON\_SUCCESS)

{

//! Set the robot digital output on channel 0 to high

arm.setRobotDO(0, 1)

//! Display the updated position

arm.GetRobotPose(&curPose);

curPose.print();

//! Set the robot digital output on channel 0 to low

arm.setRobotDO(0, 0);

}

else

{

//! Oops, something went wrong

cout << “Could not move robot.” << endl;

}

}