**Extensions to the Canonical Robot Command Language for Multi-Robot**

**Collaboration**

**v 1.0**

**Table of Contents:**

1. Introduction 3

2. The CRCL 4

 CloseGripper () 5

 CloseToolChanger () 5

 Dwell (double time) 5

 EndCanon (int reason) 6

 InitCanon () 6

 Message (string message) 6

 MoveStraightTo (Pose \* pose) 6

 MoveThroughTo (Pose \*\* poses, int numPoses) 7

 MoveTo (Pose \* pose) 7

 OpenGripper () 7

 OpenToolChanger () 8

 SetAbsoluteAcceleration (double acceleration) 8

 SetAbsoluteSpeed (double speed) 8

 SetAngleUnits (string UnitName) 8

 SetEndAngleTolerance (double tolerance) 9

 SetEndPointTolerance (double tolerance) 9

 SetIntermediatePointTolerance (double tolerance) 9

 SetLengthUnits (string UnitName) 9

 SetRelativeAcceleration (double percent) 10

 SetRelativeSpeed (double percent) 10

 StartObjectScan (string name) 10

 StopMotion (integer isEmergency) 10

 StopObjectScan () 11

3. Modifications to the Base CRCL 12

 CloseGripper () 13

SetTool (double percent) 13

 CloseToolChanger () 14

Couple (string targetID) 14

 Dwell (double time) 14

Dwell (int \*events, double \*parameters, int numEvents) 14

 MoveThroughTo (Pose \*\* poses, int numPoses) 15

MoveThroughTo (Pose \*\* poses, double \* accelerations, double \* speeds, double \*\* tolerances, int numPoses) 15

 OpenGripper () 16

 OpenToolChanger () 16

Decouple (string targetID) 16

 SetEndAngleTolerance (double tolerance) 16

 SetEndPointTolerance (double tolerance) 16

SetEndPoseTolerance (double \* tolerances) 16

 SetIntermediatePointTolerance (double tolerance) 17

SetIntermediatePoseTolerance (double \* tolerances) 17

 StartObjectScan (string name) 17

StartObjectTrack (string name, int trackType, Pose \* approach, bool retry) 17

 StopMotion (integer isEmergency) 18

StopMotion (integer condition) 18

 StopObjectScan() 19

StopObjectTrack (bool retry) 19

4. Proposed New Functionality 20

 GetRobotAxes () 20

 GetRobotPose () 20

 MoveAttractor (Pose \* pose) 20

 MoveToAxisTarget (double \* axes) 21

 RunProgram (string programName, string \* params, int numParams) 21

 SetAxialSpeeds (double \* speed) 21

 SetAxisUnits (string \* UnitNames) 22

 SetParam (string \* ParamName, void \* value) 22

# Introduction

The current description of the Canonical Robot Command Language (CRCL) provides base functionality of commanding high-level Cartesian motions of robot systems. The original construction of the CRCL was such that a given set of robot programs is expected to be reducible in syntax to this base set of functions. It is clear that some commands are actually superfluous (e.g., separate commands for opening and closing grippers), and the base CRCL can be further reduced to a more condensed form. This further reduction, however, is not seen as a critical need. Instead, it has been determined that the CRCL is in need of expansion to accommodate complex use cases of robot systems.

On the surface, the interpretation of many commands appears to be applicable to a myriad of robot platform designs including open-chain manipulator arms and automated guided vehicles (AGVs). However, it is clear that not all robots (even the relatively small sample size represented by NIST’s collection) are compatible with the current CRCL syntax. Moreover, it is clear that the CRCL syntax is not conducive to multi-robot workcells or distributed control structures.

This document focuses on redefining the CRCL to accommodate collaborative robot configurations consisting of a variety of robot classes. A number of extensions and modifications are suggested to accommodate the needs of implementing CRCL in multi-robot workcells. Commands are presented in C++ syntax, but may be implemented in any number of ways.

Graphically, the new CRCL is shown in Figure 1. The base CRCL syntax, as described in Section 2, provides the basis for the new CRCL. Section 3 describes refinements, redefinitions, and deletions of some of the base functions. Section 4 proposes new CRCL syntax

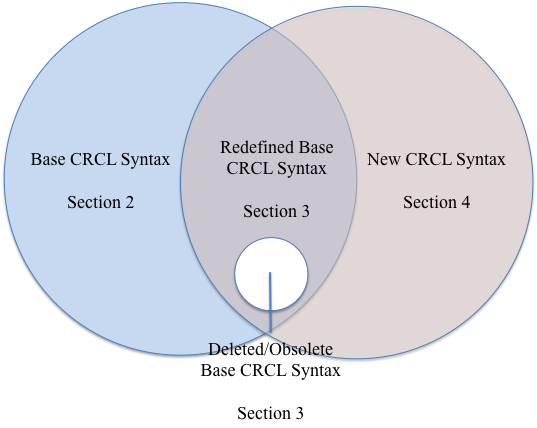


Figure 1: The proposed, extended CRCL consists of the base functionality (Section 2) with some modifications and deletions (Section 3) combined with new functionality (Section 4) to accommodate a variety of collaborative manufacturing tasks.

# The CRCL

The CRCL is centered on the observation that a common syntax is needed for describing industrial robot programs without relying on platform-specific languages or proprietary planning formats. The result is a minimal collection of functions that every robot is expected to be able to interpret.

A robot system capable of interpreting CRCL syntax is expected to consist of 1) a robot controller that accepts and executes CRCL functions, 2) a plan interpreter that reads a planning domain definition language (PDDL) file and outputs one or more CRCL commands to the controller, and 3) a world state database for the robot system. The plan interpreter issues commands to the robot controller in the form of CRCL syntax. The robot controller updates the world state, while both the robot controller and the plan interpreter can receive information from the world state. Visually, the topology of the robot system is illustrated in Figure 2, where the arrows indicate the flow of information between system components.

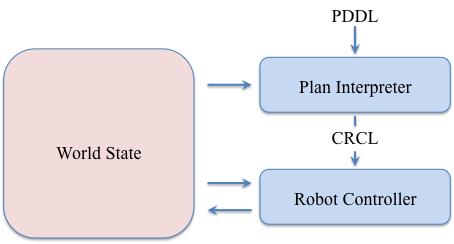


Figure 2: The canonical robot command topology. The plan interpreter extracts a plan from a PDDL file, defines a series of CRCL syntax commands that achieve the plan and submits them to the robot controller. The robot controller then executes the CRCL program, updating the world state as necessary.

From the original canonical robot command perspective, a physical robot system consists of a single industrial robot arm capable of positioning itself within its Cartesian work volume (within some set level of tolerance). The speed and acceleration of the robot’s motions are controllable, and a single end effector (presumably some form of simple gripper) can be attached to the robot’s tool flange at a time. Additional end effectors are stored on a tool changing station. It is assumed that the robot can move the arm or the gripper at any time, but not both simultaneously.

Although there is not a perfect one-to-one mapping between CRCL and the set of all industrial robot application commands, a sufficient analogue can be derived through creative interpretation. For example, a planner may employ the CloseGripper and OpenGripper functions to toggle the active state of a robot’s welding tip. Similarly, certain CRCL functionalities are interpreted liberally. For example, the original CRCL description specified robot poses in terms of 6 degrees of freedom (6DOF) Cartesian coordinates of the tool relative to the robot’s workstation. If no tool is attached, then the system interprets the motion commands to be relative to the tool flange. We relax this definition slightly such that poses are in terms of the tool center point (TCP), which, for AGVs and associated onboard equipment, is defined to be relative to the AGV’s origin.

When given commands that they are otherwise unable to execute (e.g., sending CloseToolChanger to a robot without a tool flange), the robot controller is expected to take appropriate action. No specific actions are given, however, and it is generally expected that the remainder of the plan will be ignored.

In this section, we provide an overview of the CRCL functions and, when applicable, discuss how they may be used in coordinating multi-robot collaborative workcells. The functionality of these commands assumes that they will be faithfully executed by the robot controller, and thus do not provide feedback to acknowledge receipt or signal completion. Moreover, it is assumed that a given robot will remain completely motionless unless commanded otherwise through the CRCL. As such, the pose specified in the most recent motion command is expected to be the current pose of the robot. New syntax will be introduced in in Sections 3 and 4 to relax this requirement.

The functions are presented here alphabetically for easier reference.

## CloseGripper ()

**Functionality**:

Close the gripper.

**Parameters**:

None.

**Collaboration notes**:

The CloseGripper command acts as a boolean activation signal, which may be applied to any number of possible robot-controlled tools that have a binary action state. For these tools, we will assume that the CloseGripper is analogue to the binary “on” state. When collaborating with other robots (e.g., robot hands), the CloseGripper command may be used to signal digital I/O semaphores for state coordination. A proposed alternative definition to the CloseGripper function is discussed in Section 3.

## CloseToolChanger ()

**Functionality**:

Close the tool changer on the robot so that it attaches to a tool. The robot must be in an appropriate position with respect to the tool for the changer mechanism on the robot to attach to the tool.

**Parameters**:

None.

**Collaboration notes**:

Although not every robot system is equipped with a tool changer (e.g., AGVs and robot hands), the CloseToolChanger function may be employed to signal some other part or tool acquisition functionality. This may include part hand-offs, docking, or calibration or registration procedures. A proposed alternative definition to the CloseToolChanger function is discussed in Section 3.

## Dwell (double time)

**Functionality**:

Stay motionless for a specified amount of time.

**Parameters**:

*time* – The time, in seconds, the robot shall remain motionless.

**Collaboration notes**:

The Dwell command may be utilized to coordinate the initiation of motions of multiple robots, and may likewise be used as a monitored stop for robot safety (provided the appropriate safeguards are taken to ensure that the robot will not move). Outside of these, the Dwell command has relatively little functional use for most collaborative industrial tasks.

## EndCanon (int reason)

**Functionality**:

Stop executing canonical robot commands. No specific action is required. The robot controller should not execute any canonical robot command except InitCanon after executing EndCanon, and should signal an error if it is given one.

**Parameters**:

*reason* – Numerical representation of enumerated reasons for halting execution. No set of enumerated reasons is specified in the CRCL other than a value of 0, which indicates that execution of a plan has completed successfully. A positive value of reason indicates otherwise.

**Collaboration notes**:

The EndCanon function may be interpreted as a signal for executing some automated functionality outside the purview of the CRCL lexicon. For instance, the EndCanon command may be used to signal a “gravity compensation” mode that enables hand-guiding of the robot by a human operator. Otherwise, the EndCanon may be employed as a safety functionality that prevents unintentional motions of the robot prior to an error state being cleared by an outside agent.

## InitCanon ()

**Functionality**:

Do whatever is necessary to get ready to move. Length and angle units are set to the default units. This command will normally be given when the plan interpreter opens a plan to be executed.

**Parameters**:

None.

**Collaboration notes**:

The InitCanon function may be interpreted as a signal for halting autonomous functionality and entering into a slave mode for external commands. The InitCanon function may likewise be used to remotely recover from an error state.

## Message (string message)

**Functionality**:

Display a message on the operator console.

**Parameters**:

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

**Collaboration notes**:

With its intended functionality, the Message command offers little for collaboration other than a use interface. However, this functionality may be usurped to relay task-relevant, collaborative information between robots, and between humans and robots.

## MoveStraightTo (Pose \* 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

**Collaboration notes**:

This function was specifically intended for industrial robots with kinematic solutions for placing the TCP in a specific location. However, this same functionality may also be applied to AGVs and relevant onboard equipment. It is assumed that the robot system will move from its original position to the new pose using a linear interpolation, which may be exploited for coordinated robot moves in Cartesian space. Additional timing and speed control will be needed for this extended functionality, however. It is assumed that some level of error handling will be in place such that target positions outside of the robot’s working volume are rejected.

## MoveThroughTo (Pose \*\* poses, int numPoses)

**Functionality**:

Move the controlled point along a trajectory passing near all but the last of the given poses, and stop at the last of the given poses.

**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

**Collaboration notes**:

None. It is assumed that the robot will follow a least-effort trajectory to achieve the target pose (see MoveTo command). An extension of this method must be defined to mandate path linearity for it to be useful for collaboration due to the otherwise unpredictable path of the TCP between Cartesian poses. It is assumed that some level of error handling will be in place such that target positions outside of the robot’s working volume are rejected.

## MoveTo (Pose \* 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

**Collaboration notes**:

None. It is assumed that the robot will follow a least-effort trajectory to achieve the target pose. Specifically, the motion will be that which results in the linear interpolation of joint angles between the current pose and the target pose. Although the joints will be interpolated linearly, the path of the TCP is expected to be non-linear and unpredictable. It is assumed that some level of error handling will be in place such that target positions outside of the robot’s working volume are rejected.

## OpenGripper ()

**Functionality**:

Open the gripper.

**Parameters**:

None.

**Collaboration notes**:

The OpenGripper command acts as a boolean activation signal, which may be applied to any number of possible robot-controlled tools that have a binary action state. For these tools, we will assume that the OpenGripper is analogue to the binary “off” state. When collaborating with other robots, the OpenGripper command may be used to signal digital I/O semaphores for state coordination.

## OpenToolChanger ()

**Functionality**:

Open the tool changer on the robot so that it releases the end effector. This is normally done after the end effector attached to the robot has been moved into an end effector changer.

**Parameters**:

None.

**Collaboration notes**:

Although not every robot system is equipped with a tool changer (e.g., AGVs and robot hands), the OpenToolChanger function may be employed to signal some other part or tool acquisition functionality. This may include part hand-offs, docking, or calibration or registration procedures. A proposed alternative definition to the CloseToolChanger function is discussed in Section 3.

## 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.

**Collaboration notes**:

Robots with coordinated motions should be operating at roughly the same target acceleration, though motions over long distances will be dominated by the speed term.

## 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.

**Collaboration notes**:

Robots with coordinated motions should be operating at the same target speed, though motions over short distances will be dominated by the acceleration term.

## SetAngleUnits (string 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.”

**Collaboration notes**:

None, however it is imperative that all collaborative robots use the same angle units to avoid misalignment.

## SetEndAngleTolerance (double tolerance)

**Functionality**:

Set the tolerance for the orientation of the TCP to the given value in current angle units.

**Parameters**:

tolerance – The user-defined tolerance of the orientation of the TCP at the end of a move command

**Collaboration notes**:

None.

## SetEndPointTolerance (double tolerance)

**Functionality**:

Set the tolerance for the position of the TCP to the given value in current length units.

**Parameters**:

tolerance – The user-defined tolerance of the position of the TCP at the end of motion commands.

**Collaboration notes**:

None.

## SetIntermediatePointTolerance (double tolerance)

**Functionality**:

Set the tolerance for smooth motion near intermediate points to the given value in current length units.

**Parameters**:

tolerance – The maximum distance offset in length units the TCP is allowed to be during MoveThroughTo motions

**Collaboration notes**:

None.

## SetLengthUnits (string 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.”

**Collaboration notes**:

None, however it is imperative that all collaborative robots use the same length units to avoid misalignment.

## 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]

**Collaboration notes**:

None. It should be noted that the CRCL plan interpreter does not have feedback from the robot, so the interpreter must have *a priori* knowledge of the robot’s maximum acceleration prior to issuing this command.

## 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]

**Collaboration notes**:

None. It should be noted that the CRCL plan interpreter does not have feedback from the robot, so the interpreter must have *a priori* knowledge of the robot’s maximum speed prior to issuing this command.

## StartObjectScan (string name)

**Functionality**:

Activates[[1]](#footnote-1) the object sensor, if it isn't already, and adds the given name to the list of parts being searched for. When the object sensor detects the part, it removes the part from the list of parts to search for and writes a sequence of canonical commands to "output\_commands.txt" (as a placeholder for updating the mySQL database). When the list of parts to search for becomes empty, the object sensor stops motion as soon as possible, and the controller ignores further movement commands until the sensor is deactivated.

**Parameters**:

name – The plaintext name of the part being searched for.

**Collaboration notes**:

Additional requirements and restrictions are necessary for the name parameter given the potential for collaborative tasks. There must be a means of distinguishing between commands associated with a specific object (i.e., a unique identifier) versus a single instantiation of a larger collection of objects. For example, the CRCL must be able to distinguish between commands involving “a white sphere” and “the autographed baseball.”

## StopMotion (integer isEmergency)

**Functionality**:

Stop the robot’s motions.

**Parameters**:

isEmergency – Whether or not the StopMotion command is an emergency. If isEmergency is 0, then the robot shall stop as soon as possible. Otherwise, the robot shall come to a graceful stop.

**Collaboration notes**:

Mechanisms for targeting and propagating emergency StopMotion commands are necessary if robots are expected to collaborate with one another. Currently the function parameter, isEmergency, is underutilized, and may be expanded to incorporate halting conditions (e.g., time to stop or event-based stopping conditions). Moreover, current robot safety standards distinguish between different types of emergency stops, and it would be worth attempting some level of analogue here.

## StopObjectScan ()

**Functionality**:

Deactivates the object sensor.

**Parameters**:

None.

**Collaboration notes**:

None.

# Modifications to the Base CRCL

The CRCL is largely sufficient for describing tasks that involve only a single industrial robot. However, when multiple robots (or robots of classes other than industrial robots) are expected to coordinate their motions to complete some collaborative task, the CRCL falls short of the requisite capabilities. In fact, by adding more robots, we find that the original robot system topology is not sufficient to enable coordinated robot control. In a single robot system, the plan interpreter needs only to talk to a single robot, which may be accomplished asynchronously through the world state database. Coordinating motions between multiple robots requires a more complex topology with bidirectional communications.

Rather than a one-to-one mapping of plan interpreter to robot controller, we propose that a new robot system topology be considered. In this new topology, a single plan interpreter issues targeted CRCL syntax commands to multiple robots. These robots execute the CRCL command functions, and feed status update to a shared world state database. The lines of communication are improved in that the robot controllers may feed directly back to the plan interpreter rather than indirectly sharing critical state information through the world state database (which has no guarantee of speed or accurately representing the current state of the robots). Moreover, multiple robots are given the ability to communicate with one another through shared bidirectional data streams, allowing the robots to coordinate their motions independent of the plan interpreter. To accommodate this, each robot is assigned a plaintext identifier, which the other robots can use to address messages, dock, and look up capabilities or custom functions.

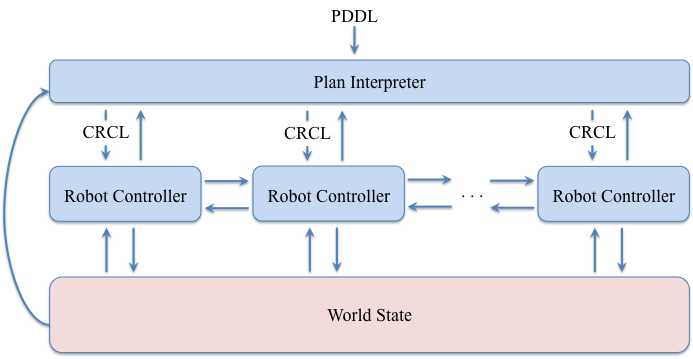


Figure 3: The proposed new canonical robot command topology. The plan interpreter extracts a plan from a PDDL file, defines a series of CRCL syntax commands that achieve the plan and submits them to targeted robot controllers. The robot controller then executes the CRCL program, sending feedback to the plan interpreter, and updating the world state as necessary.

In this section we highlight commands discussed in Section 2 that require modifications for more dynamic use cases of robot systems. In most cases, identified functions require minor adjustments to the defined functionality to accommodate different robot classes or use scenarios. In some cases, however, additional parameters are required for the functions to be useful. Some functions must be removed entirely. Functions not already part of the CRCL will be described in Section 4.

In general, it is recommended that all commands be given an enumerated return type. This return type will specify when an action has been executed successfully or when there is an error. The suggested 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. |

As mentioned previously in Section 2, the coordinate frame for all Cartesian motion commands is that of the robot’s workstation. In multiple robot systems, there may be multiple workstations. When it is necessary for the robots to communicate with one another for coordinated motions, they must be aware of where in space those workstations are relative to their own workstations. 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.

Alphabetically, the commands with suggested adjustments to functionality are as follows.

## CloseGripper ()

**Modification class**:

Rename and functional scope change.

**Reason**:

Not all robot classes have standard open/close grippers. Some, in fact, may not have grippers at all, but instead have attached components or tools. As such, the generic “gripper” functionality needs to be extended for variable-output tooling.

**Suggested change**:

### 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.

## CloseToolChanger ()

**Modification class**:

Rename and functional scope change.

**Reason**:

Not every robot is equipped with a tool changer, but may require some physical coupling for tool acquisition. The physical connection between different robots, stations, or tools is not intuitively captured with a boolean tool changer command.

**Suggested change**:

### Couple (string 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

## Dwell (double time)

**Modification class**:

Functional scope change.

**Reason**:

The current Dwell command is limited in application such that only a timer event can restart motion. In some cases, a robot’s motions may need to be resumed based on some other event (e.g., sensor inputs or user-generated signals).

**Suggested change**:

### Dwell (int \*events, double \*parameters, int numEvents)

**Functionality**:

Stay motionless until an event occurs. If multiple events are defined for the dwell command, motion resumes whenever one event occurs. Events and their associated parameters are classified as follows:

|  |  |  |
| --- | --- | --- |
| **Event ID** | **Event Description** | **Parameter range : description** |
| 0 – Timer Event | Resume motion after a set amount of time has passed | [0, ∞) : The amount of time, in seconds, to wait |
| 1 – Force Event | Resume motion after a force threshold has been exceeded | (-∞, ∞) : The absolute force threshold, in Newtons. If ≥ 0, motion resumes if measured force is greater than threshold. If < 0, motion resumes if measured force is less than threshold. |
| 2 – Relative force Event | Resume motion after a force threshold relative to the current force has been exceeded | (-∞, ∞) : The force threshold, in Newtons, relative to the current measured force. If ≥ 0, motion resumes if measured force is greater than threshold. If < 0, motion resumes if measured force is less than threshold. |
| 3 – External digital signal | Resume motion after an external digital signal is received | [0, ∞) : The digital channel (where the channel # = ⎣parameter⎦) to monitor for a digital “high” signal. |

**Parameters**:

events – An array of event identifiers to signal that motion is to resume

parameters – An array of parameters associated with the array of events that bound the response to different events

numEvents – The size of the events array and the parameters array.

## MoveThroughTo (Pose \*\* poses, int numPoses)

**Modification class**:

Functional scope change.

**Reason**:

The current MoveThroughTo command assumes that the motion profile throughout a multi-target trajectory will use the same accelerations, speeds, and tolerances set at the system level. In many cases, this assumption is false. Also, the current definition does not specify the manner in which the robot is expected to interpolate between target poses.

**Suggested change**:

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

**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

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.

numPoses – The number of poses in the submitted array

## OpenGripper ()

**Modification class**:

Deletion.

**Reason**:

Deprecated functionality, see CloseGripper.

## OpenToolChanger ()

**Modification class**:

Rename and functional scope change.

**Reason**:

Not every robot is equipped with a tool changer, but may require some physical coupling for tool acquisition. The physical connection between different robots, stations, or tools is not intuitively captured with a boolean tool changer command.

**Suggested change**:

### Decouple (string targetID)

*Functionality*:

Undock with a specified target object (e.g., tool, gripper, or station). The Decouple command performs a physical, electrical, and/or software severance action based on the known capabilities and requirements for the defined object. The robot rejects the Decouple function if it is not already coupled with the specified object or if it does not know how to decouple with the specified object (e.g., if no undocking procedure is defined for the object).

Parameters:

targetID – The name of the object from which the robot should undock

## SetEndAngleTolerance (double tolerance)

**Modification class:**

Deletion.

**Reason**:

Depreciated functionality, see SetEndPointTolerance.

## SetEndPointTolerance (double tolerance)

**Modification class:**

Rename and functional scope change.

**Reason**:

Some applications will require tighter tolerances along different axes with lower tolerance in others, and the current definition does not accommodate this requirement. Additionally, it makes more sense for the position and orientation tolerances to be defined in a single function rather than creating separate functions for both.

**Suggested change**:

### SetEndPoseTolerance (double \* 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 – An array of user-defined tolerances of the 6DOF pose of the TCP at the end of a move command

## SetIntermediatePointTolerance (double tolerance)

**Modification class:**

Rename and functional scope change.

**Reason**:

Some applications will require tighter tolerances along different axes with lower tolerance in others, and the current definition does not accommodate this requirement. Also, the original CRCL does not provide for intermediate orientation tolerances. It makes more sense for the position and orientation tolerances to be defined in a single function rather than creating separate functions for both.

**Suggested change**:

### SetIntermediatePoseTolerance (double \* 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

## StartObjectScan (string name)

**Modification class:**

Rename and functional scope change.

**Reason:**

The current definition of StartObjectScan implies visual servoing for the acquisition of stationary objects. While this may be sufficient for most applications of machine vision for robotics, it lacks the capacity for dynamic object tracking (e.g., picking objects from a moving conveyor belt) and robot coordination (i.e., the robot should continue moving even after “acquiring” the target object).

**Suggested change**:

### StartObjectTrack (string name, int trackType, Pose \* approach, bool retry)

**Functionality**:

Activates the object sensor, if it isn’t already, and adds the given name to the list of objects being searched for and tracked. The robot will then perform visual servoing to identify and locate the specified object. Depending on the defined track type, the robot will perform different actions. These actions are as follows:

|  |  |
| --- | --- |
| **trackType** | **Action behavior** |
| 0 – Stationary object acquire | Identifies and locates the specified stationary object. When the object sensor detects and locates the specified object, it removes the object from the list of objects to track, and writes a sequence of canonical commands to acquire the object. |
| 1 – Dynamic object acquire | Identifies and locates the specified moving object. When the object sensor detects and locates the specified object, the robot continues visual servoing to grasp the object while attempting to maintain the specified approach angle. Once the robot acquires the object, it removes the object from the list of objects to track. If the object moves beyond the robot’s work volume while the robot is attempting to acquire it, the robot can either attempt to find another equivalent object or return an error. |
| 2 – Dynamic object following | Identifies and locates the specified moving object. When the object sensor detects and locates the specified object, the robot continues visual servoing to maintain a specified pose relative to the tracked object. |

The object tracking can be aborted at any time by means of the STopObjectTrack command.

**Parameters**:

name – The plaintext name of the object being searched for.

tracktype – Which action behavior the robot should exhibit during the object tracking motion.

approach – The approach/follow position relative to the surface normal of the located object. The vector between the object’s origin and the approach position defines the vector of approach through which the robot will orient itself to acquire/follow the object. This value is optional for the object acquire actions, and if it is not provided the robot will attempt to acquire the object using the shortest approach vector from the pose at which the object was originally identified.

retry – Whether or not the robot should attempt to automatically find another object if the current tracked object is lost or moves beyond the robot’s reach.

## StopMotion (integer isEmergency)

**Modification class:**

Refinement of definition.

**Reason:**

The current definition of the StopMotion function underutilizes the addition of the isEmergency parameter, and can easily be modified for use in several anticipated stopping conditions. Moreover, use the value of 0 for isEmergency is counterintuitive because the variable name implies that the value of 0 indicates that it is not an emergency.

**Suggested change**:

### 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.

## StopObjectScan()

**Modification class:**

Rename and functional scope change.

**Reason:**

Adjusted to follow the changes suggested for the StartObjectScan command.

**Suggested change**:

### StopObjectTrack (bool retry)

**Functionality**:

Abort the current effort to acquire or follow an object. If so desired, the robot may automatically attempt another acquire action. Aborting a follow command removes the current object being followed from the list of objects being tracked.

**Parameters**:

retry – Whether or not the robot should attempt to automatically find another object to acquire after aborting the current track effort. Not used when aborting object follow actions.

# Proposed New Functionality

A number of robot applications require functionality that is not supported in the current CRCL syntax. Although an exhaustive effort to extend the CRCL to support all foreseeable uses of robot systems is not feasible, a number of manufacturing-relevant task applications can be accommodated by means of a few simple extensions. In particular, these extensions facilitate the integration and control of non-arm robots such as robotic hands and AGVs. We also provide mechanisms for closing the loop in what is otherwise an open-loop system.

Even though additional sensor-based functions are added to the CRCL, there is currently no mechanism planned to retrieve sensor information other than joint encoder/resolver feedback. The reason is that the types and configurations of available sensors for use in robot applications are innumerable.

In this section, we discuss these extensions and the reasons for their inclusion into the CRCL. Returned types and values, when different from the suggested values in Section 3, are also provided. Alphabetically, the new functions are as follows.

## GetRobotAxes ()

The current CRCL is lacking any mechanism for returning robot information to higher level processes and planners. In many cases, verification that the robot is where it is expected to be is vital to the performance of the system as a whole. When associated with a timestamp, this command may also be used to characterize the performance of individual robot axes.

**Functionality**:

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

**Parameters**:

None.

## GetRobotPose ()

The current CRCL is lacking any mechanism for returning robot information to higher level processes and planners. In many cases, verification that the robot is where it is expected to be is vital to the performance of the system as a whole. When associated with a timestamp, this command may also be used to characterize the motion of the robot through Cartesian space.

**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**:

None.

## MoveAttractor (Pose \* pose)

The original CRCL syntax does not have any mechanism for force-based motion control. There are several force control algorithms available, and it is not our intention to prescribe which of these algorithms a robot should use. Instead, it is preferable to only describe the performance functionality. A robot using force control shall follow a virtual attractor, and the force applied by the robot is proportional to the difference between the robot’s current pose and the pose of the attractor. The robot will attempt to minimize the positional error between the current TCP location and the virtual attractor location using stiffness control based on the user-specified virtual springs and dampers (defined at the robot level). For instance, moving the attractor just below the surface of an object will cause the robot to maintain constant contact with the object, but will limit the force applied to the object’s surface. This same function may be used for other control mechanisms such as potential fields and virtual forces for use in obstacle avoidance. If the robot system is incapable of handling any of these control methods, the robot rejects the MoveAttractor command.

**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.

## MoveToAxisTarget (double \* axes)

The original CRCL syntax explicitly forbade the ability to command motions in terms of axis angles or distances. The resulting mandate required that all motions be specified in terms of Cartesian coordinates. Cartesian coordinates are meaningless to some robot systems such as robotic hands or robots without inverse kinematic solutions.

**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.

## RunProgram (string programName, string \* params, int numParams)

Some robots are expected to have application-specific functionality that is not easily described in terms of CRCL syntax. This function allows for platform-specific programs to be executed without requiring an analogue in CRCL or expecting performance capabilities beyond those that are reasonable for issuing high-level commands. For example, force-based assembly search functions cannot be defined without also mandating a tight control loop with the robot and high fidelity sensor feedback.

**Functionality**:

Run a specific, pre-written program or function on the robot that is otherwise indescribable using the CRCL (e.g., force-based assembly searches). It is expected that the robot will read and cast the parameters to the appropriate units per the function definition. The robot will reject the command if the specified program is not defined or if the number of parameters is incorrect.

**Parameters**:

programName – The name of the program/function to be executed by the robot.

params – An array of parameters to be applied to the program call.

numParams – The number of parameters contained in the params array.

## SetAxialSpeeds (double \* speed)

The axis-space motion commands require parameters separate from the Cartesian motion commands. Some robot motions are expected to require independent control over the input parameters (e.g., if the robot’s axes include independently controllable treads for skid steering).

**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 (string \* UnitNames)

The axis-space motion commands require parameters separate from the Cartesian motion commands. Robots with independent control over specific axes (e.g., speed and steering) will have different axial units.

**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.

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

Many robots require the configuration of internal variables to function properly (e.g., to switch between control modes). Because the parameters types are both variable and numerous, it is necessary to provide a generic interface for setting these values.

**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

1. Activate and deactivate are used loosely, because the controller is still always subscribed to the sensor topic. But, the sensor callback returns immediately if the sensor is deactivated. [↑](#footnote-ref-1)