Motoplus-ROS Incremental Motion interface

Engineering Design Specifications

DOCUMENT NO:	M2092-EDS
DOCUMENT VER.:	1.2.0
DATE:	04/07/2017

Disclaimers

The information contained in this document is the proprietary and exclusive property of Yaskawa Motoman Robotics except as otherwise indicated. No part of this document, in whole or in part, may be reproduced, stored, transmitted, or used for design purposes without the prior written permission of Yaskawa Motoman Robotics.

The information contained in this document is subject to change without notice.

The information in this document is provided for informational purposes only. Yaskawa Motoman Robotics specifically disclaims all warranties, express or limited, including, but not limited, to the implied warranties of merchantability and fitness for a particular purpose, except as provided for in a separate software license agreement.

Privacy Information

This document may contain information of a sensitive nature. This information should not be given to persons other than those who are involved in the **ROS** project or who will become involved during the lifecycle

History

Revisions and Reviews			
Version	Person(s)	Description	Date
1.0.0	Tom Moolayil and Eric Marcil	Original version	11/01/2012
1.1.0	Eric Marcil	Revised version Target FS100 controller and rewrite the MotoPlus application in C.	03/14/2013
PDR		Eric Marcil, Ted Miller, Greg Morgan, Takeomi Hidata	3/18/2013
Review		Review with SwRI: Jeremy Zoss	3/19/2013
1.1.1	Eric Marcil	Integrated comments from the PDR and Review from SwRI. Main changes: redefinition of simple message and extra information on Motion_ready state	3/20/2013
1.1.2	Eric Marcil	Review section 2.7 Interpolation to reflect changes made to improve performance.	4/5/2013
1.1.3	Eric Marcil	Add RobotState message Change to architecture to check incoming joint trajectory data.	5/15/2013
1.1.4	Eric Marcil	Added I/O feedback signal in section 4.2 Added detail about DX100 version	8/14/2013
1.2.0	Eric Marcil	Updated document with new messages that have been added in the recent years. Section 2.4.6 to 2.4.11	4/7/2017

Document Approval

Motoman:

Yaskawa America, Inc.
Motoman Robotics Division
100 Automation Way
Miamisburg, OH 45342
937-847-6200

Customer:

Company Name
Address
City, State, Zip
Phone

Approvals:

#	Name	Title	Organization/Dept.
1			
2			
3			
4			
5			
6			

#	Signature	Date
1		
2		
3		
4		
5		
6		

Table of Contents

Index of Figures

1 Overview

The ROS-Industrial program, initiated by Southwest Research Institute (SwRI), enables new applications and reduces project costs for industrial robotics. ROS-Industrial leverages the advanced capabilities of the Robot Operating System (ROS) software for powerful new industrial applications. This platform is usually used to calculate possible robot IK solutions by creating a virtual world identical to that of the real robot and using the obstacle/work space information to plan an optimal path to perform a task.

The ROS industrial calculates a path and streams the way points using “Simple Message” to the MotoRos driver running in the Yaskawa controller (DX100 or later). The MotoRos driver interpolates between the way points to generate motion increments matching the controller motion interpolation clock and moves the robot through the waypoints.

1.1 Objectives

The overall objectives of this project are:

1. To enable the robot to execute externally generated trajectories at full speed and smoothing as is appropriate during the course of executing any trajectory.
2. To create a Motoplus application that used the mpExRcsIncrementMove function.
3. To define the communication interface of ROS to incorporate required data to define a trajectory and necessary function to operate the robot.
4. To implement restrictions on the incoming data from the PC to enforce safety and prevent damage to the robot.

2 Specifications

2.1 Architecture

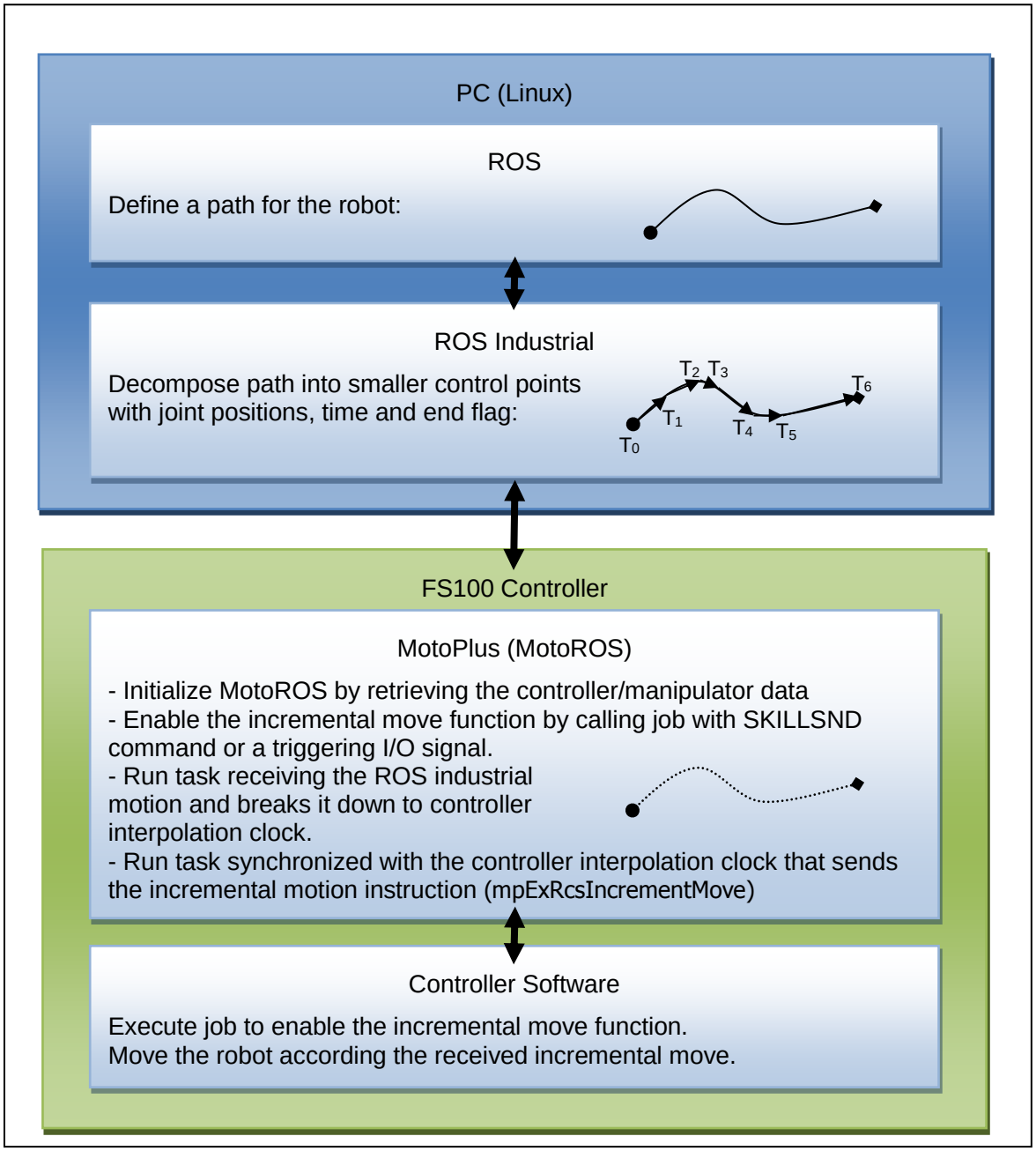


Figure 1: System Architecture

2.1.1 ROS to MotoROS

ROS-Industrial is responsible for generating the way-points and sending them to the robot controller. ROS will internally generate the way points and velocity information and send it via TCP/IP to a MotoPlus application, MotoROS, running on the controller to interpret and use them as it sees fit.

2.1.2 MotoROS to ROS

Once MotoROS receives a point, it will send a reply to the ROS side to let it know it has received the way point and it is ready to receive subsequent points.

2.1.3 INFORM to Motoplus

There will be an inform job which has to be running in order for Motoplus to move the robot. It doesn't need to have any motion commands. The mpExRcsIncrementMove command only works while the cursor is on a WAIT command. The INFORM job will look as follows:

```
NOP
'reset the I/O signals
DOUT OT#(890) OFF
DOUT OT#(889) OFF
TIMER T=0.05
'
'signal ROS that the controller
'is ready to receive motion
DOUT OT#(889) ON
'
'wait for the signal that ROS is done
WAIT OT#(890)=ON
'
'turn off the controller ready signal
DOUT OT#(890) OFF
END
```

For the final product release, this code will probably be encapsulated in the macro function to facilitate the usage.

2.2 Communication Sequence and Tasks

2.2.1 Main Task

The main task is the initial task at start-up. Figure 2 shows the main task operation and interaction with the controller. It will create an instance of the controller structure which contains all the data (controller data, control group, queue…) of the MotoPlus application. It starts the Connection Server task and then enter in an endless loop that will monitor the controller status (alarm, error, servos, play…) and take required actions when the state of the controller changes. For example monitor conditions to detect when the controller is ready to receive motion from ROS.

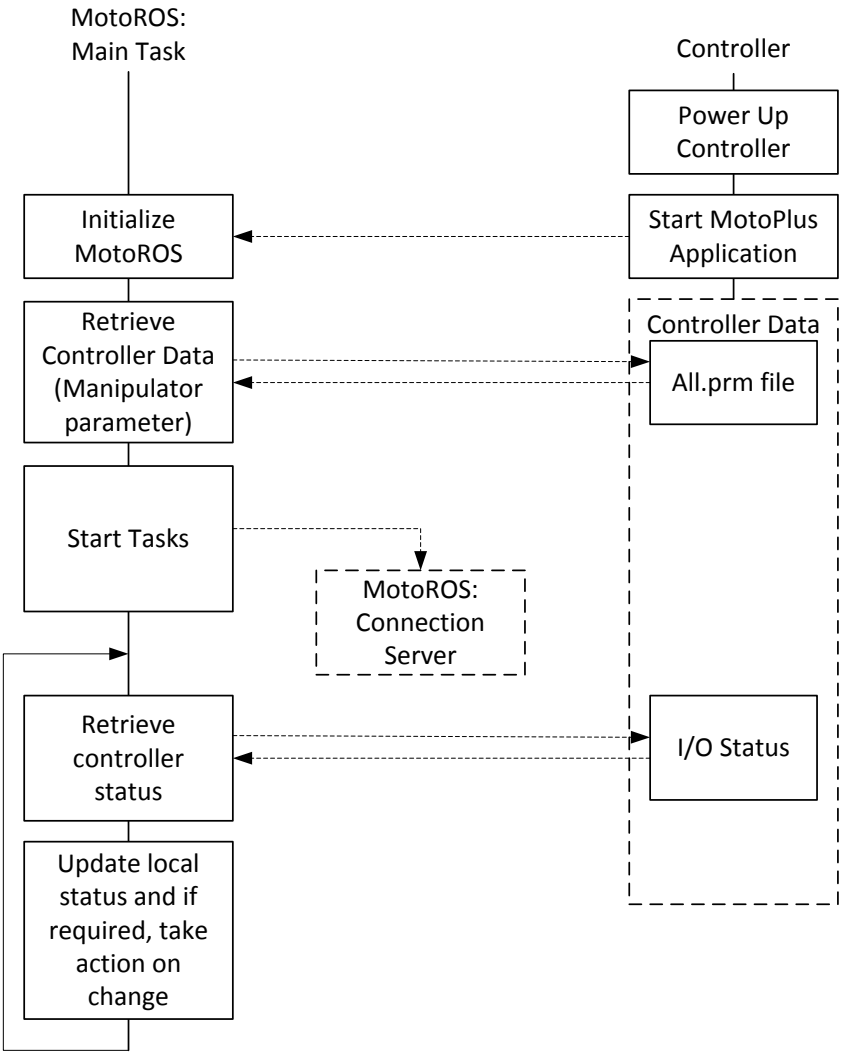


Figure 2: Main Task Start-up

2.2.2 Connection Server Task

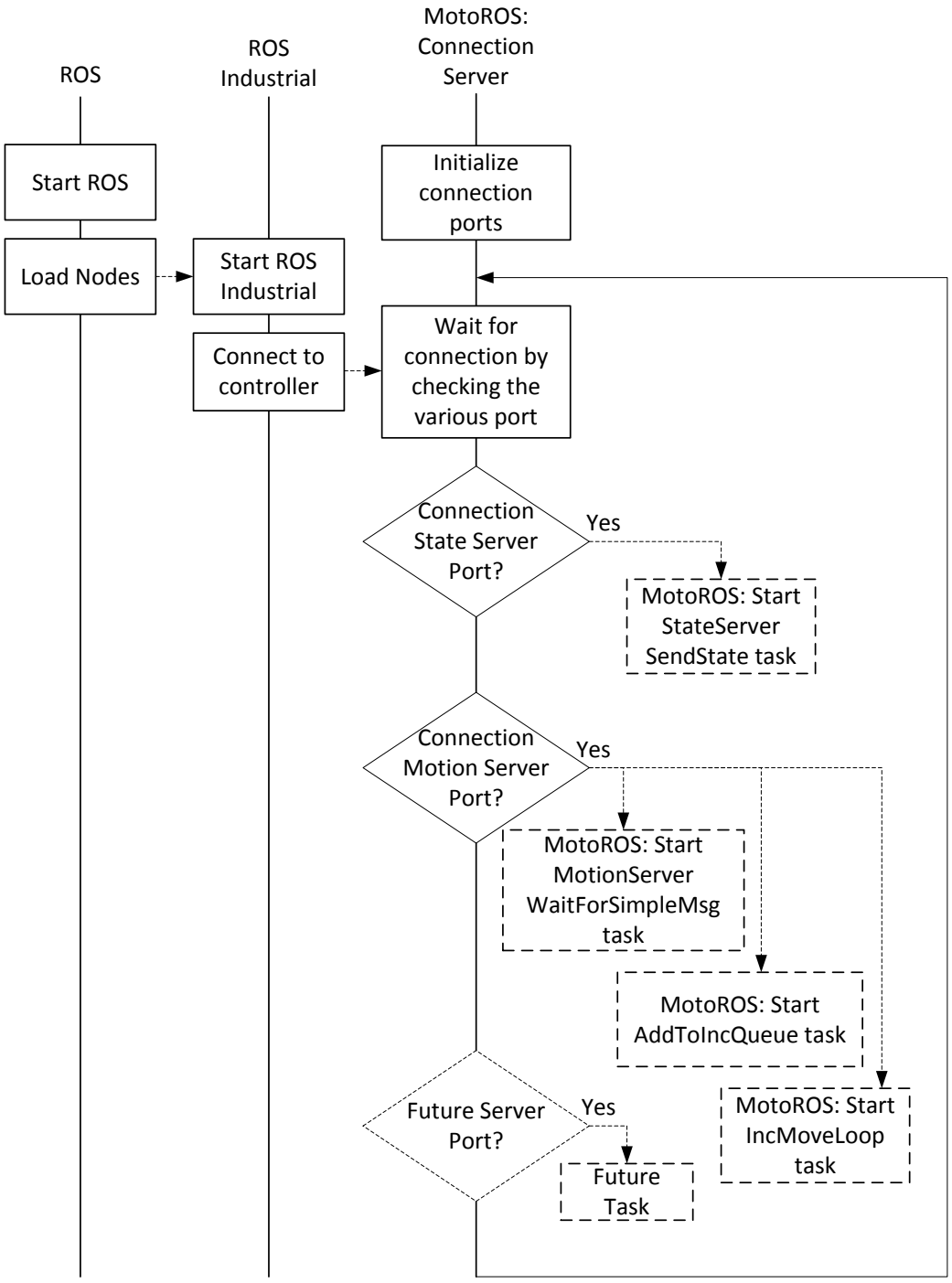


Figure 3: Connection Server Task

The connection server task (figure 3) opens connections on various ports (state, motion…) and then waits for a client to connect on one of the port. When a connection is detected on a port, a corresponding new task is started to handle communication to the connected client. The architecture is such that multiple clients can connect on a same server. They are however a maximum number of allowable connections and the connection request will be denied if all connections are already in use. Initially the supported ports and corresponding server tasks are the State and the Motion servers but this could be expended to other type of service server in the future. It is important to be mindful of the number of the overall tasks in the system.

2.2.3 State Server Task

The state server task (figure 4) retrieves the controller state (robot position) and sends it to the clients connected to the state server port. It doesn't wait on any request from the client, the moment the connection is established it will automatically start sending the information at regular interval. Thus there may be multiple clients listening, there will be a single instance of the state server task.

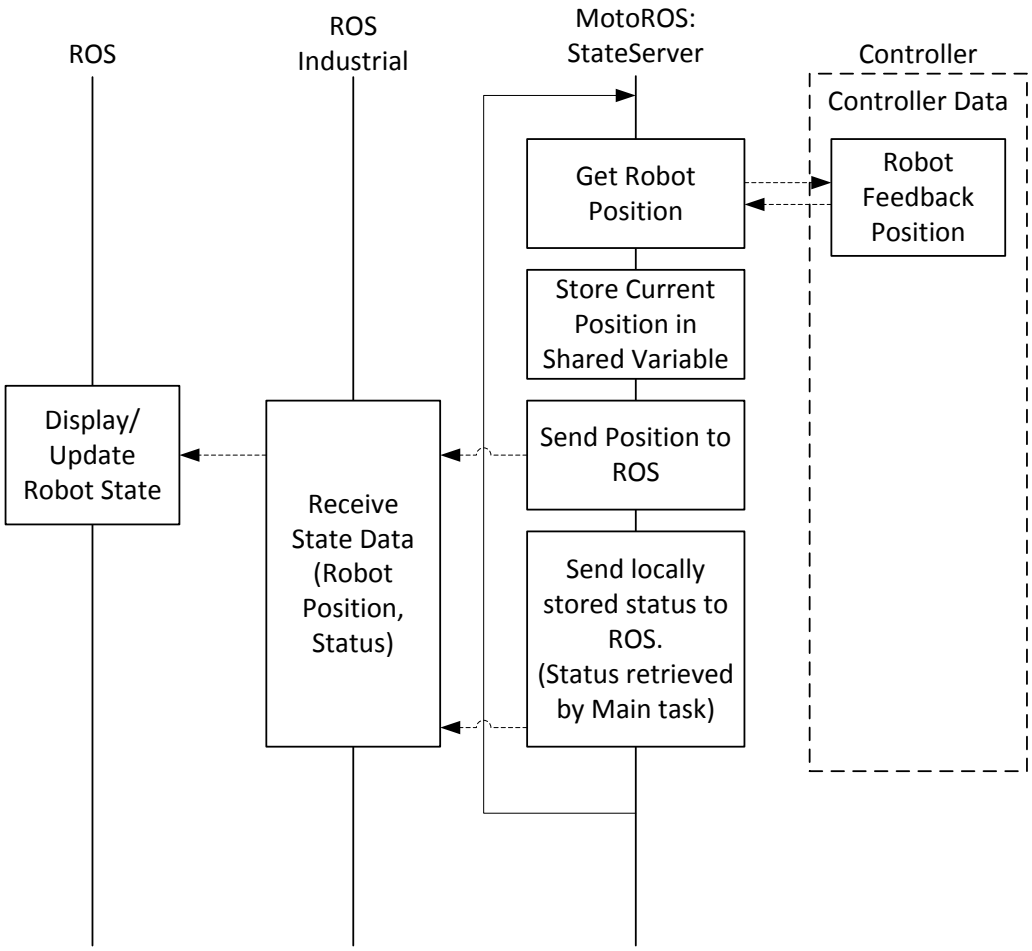


Figure 4: State Server Task

2.2.4 Motion Server Task

A new motion server task is started for each connection to the motion server port. But multiple connections should never attempt to control the same control group. So in the initial implementation which will focus on a single robot, only one connection should be allowed. When multiple robots are to be controlled, if the motion message sent contains the motion for all control groups, then there should also be only one connection allowed. But if each control group is controlled by its own node sending messages only for that group, then a connection for each group should be allowed.

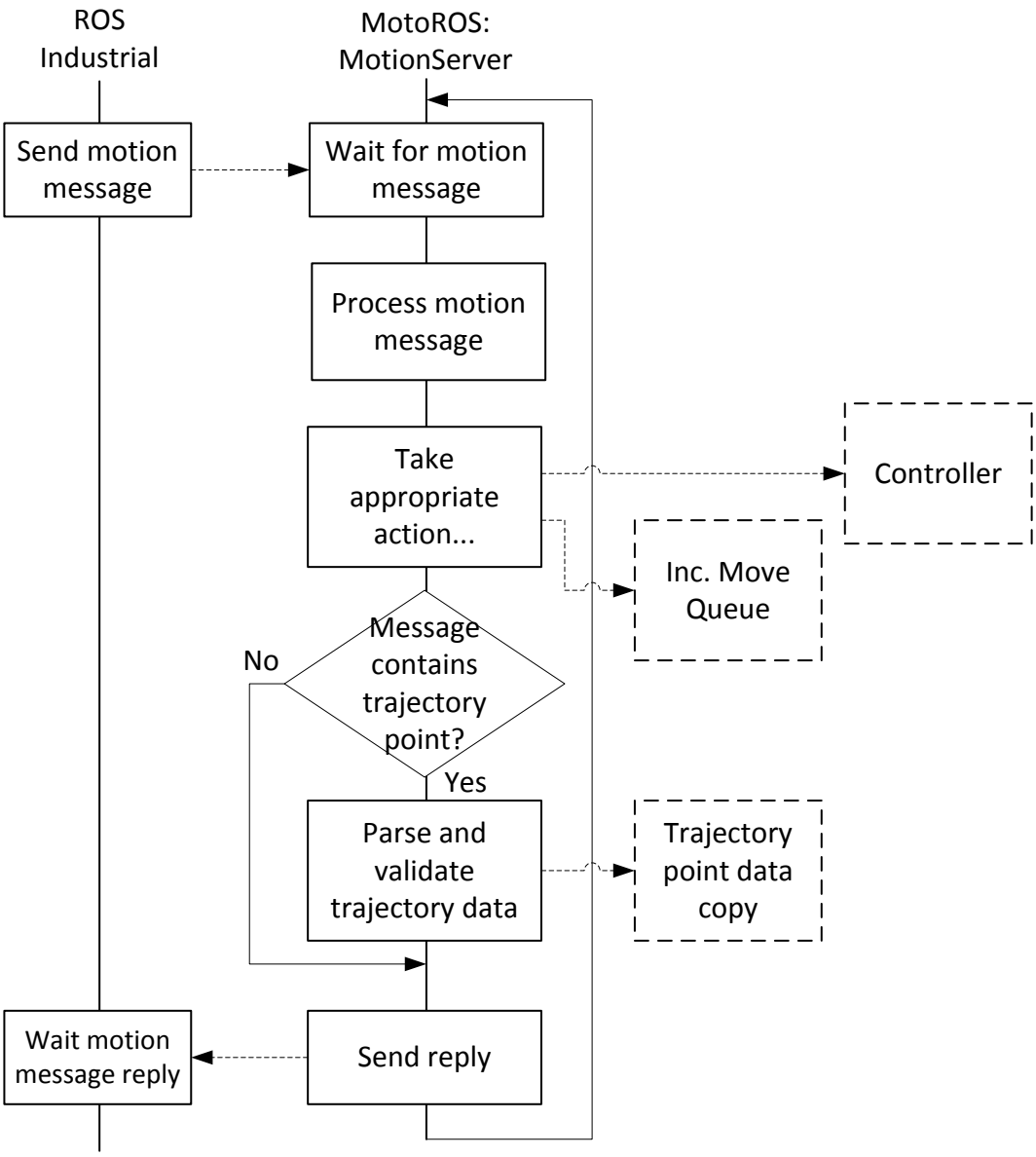


Figure 5: Motion Server Task

The motion server task (figure 5) will wait for motion message from the client. When receiving a message it will identify the message type and purpose, then take action accordingly before replying back to the client with the result of the processing (success, failure…) in the MOTION_REPLY message.

In the case of motion message that adds a trajectory point used to generate incremental motion in the queue, if the queue is full the reply could be delayed. This is undesirable because it would prevent other message such as stop motion to come in. So in order to prevent this situation, the message is parsed and its data validated, the point data is then temporarily copied to be processed by the AddToIncQueueProcess background task. So a success reply on such instruction only indicates that the message was accepted and not that the motion was completely processed into incremental moves. If the background task is already processing a point, a busy reply will be return to the client and the client will have to resend the message again.

2.2.5 Add to Inc Move Queue Task

The AddToIncQueueProcess task (figure 6) is a background tasks that take a motion message's trajectory data and breaks it down to incremental motion segments that matches the controller interpolation period. In the case of multiple motion server connections, you would have one AddToIncMoveQueue task for each motion server task.

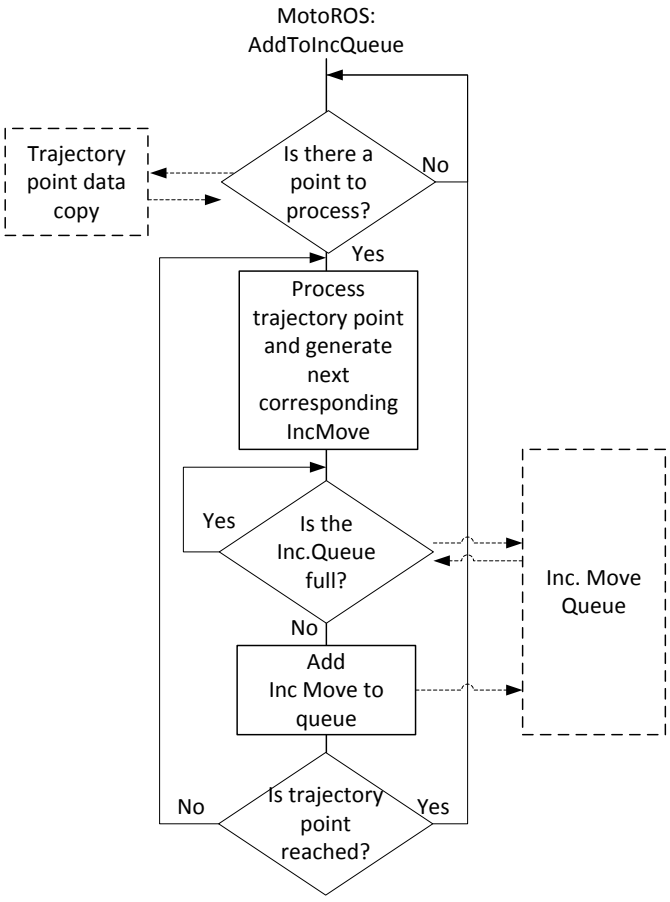


Figure 6: Add to Increment Move Queue Task

2.2.6 Inc Motion Task

The incremental motion task (figure 7) synchronizes with the controller internal clock and transfers the incremental move for one motion interpolation period to the controller with the mpExRcsIncrementMove. This is what regulates the motion to obtain the proper motion speed. Regardless of the number of motion server tasks there should only be one instance of the incremental motion task.

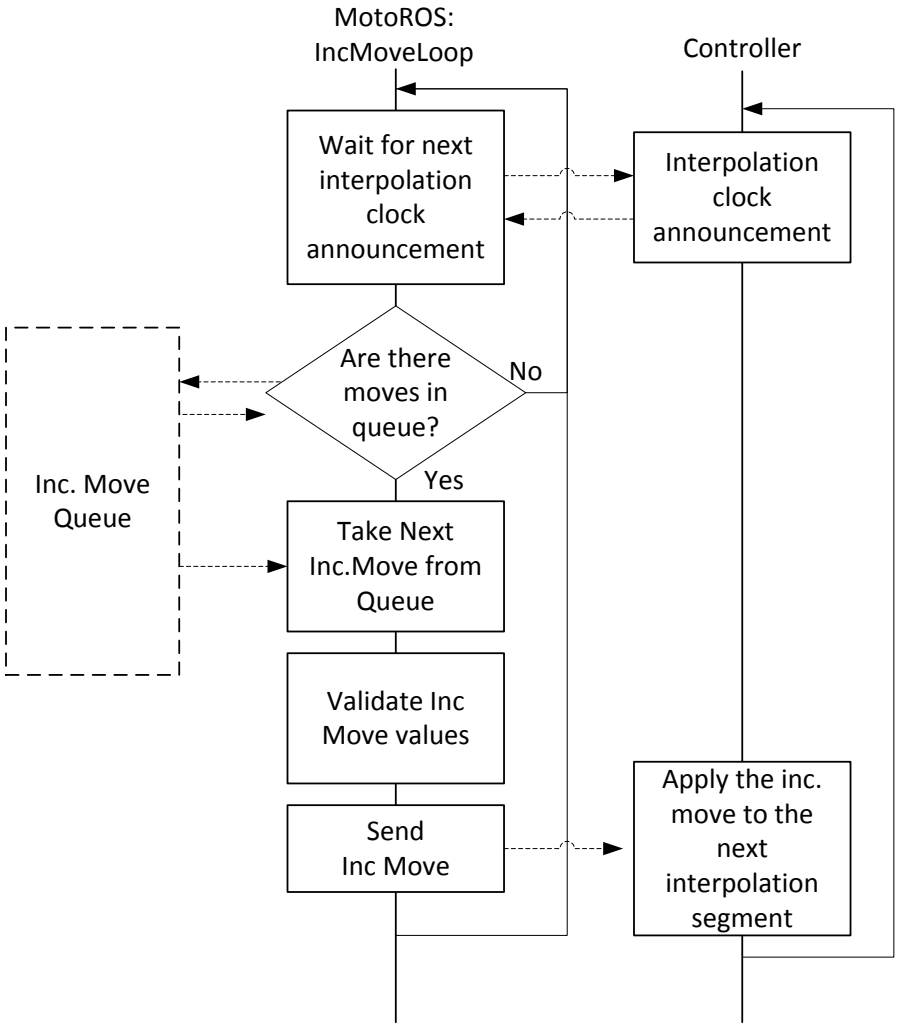


Figure 7: IncMotionTask

2.3 Data Structures

2.3.1 Controller Structure

The controller structure is the main data structure that contains all the data of the MotoROS application. It is instantiated and initialized with the controller data at the start of the MotoROS application. It holds the controller specific information such as the interpolation period, the number of control groups, pointers to the various tasks and so on. The pointer to this structure is normal passed as a parameter to most functions.

In the initialization phase, the controller data is populated by calling functions from the ParameterExtraction library.

2.3.2 Control Group

One of the main substructures to the controller is the control group structure (ctrlGroup) which contains the data for a specific control group. A control group is a logical grouping of multiple axes that represent a mechanism such as a robot, a base or a station. For example, in the case of a single arm robot there should be only one group R1; but for a dual arm robot you would have R1, R2, B1, B2 (which is physically the same as B1).

Each control group will hold the data specific to its axes: the number of axes, the pulse ratios, speed limits… It will also have its own incremental motion queue, so that one group can be move independently from other groups.

2.3.3 Incremental Motion Queue

The incremental motion queue is a circular queue of a fixed size that is constantly reused. It allows taking a trajectory that is define by a few points and sent to the MotoRos at a slower rate and break it down so smaller increments corresponding to the interpolation period. The incremental motion queue holds those smaller increments that can be sent to the controller through the mpExRcsIncrementMove function at the interpolation rates.

The incremental queue also includes the variables to control its access form multiple threads and keeps track of its usages (index, data count).

2.4 Simple Messages

ROS Industrial uses simple message structure to communicate between the ROS and the MotoROS application running on the controller. Note that the FS100 uses the Big Endian convention when parsing network communication and that the corresponding ROS communication library needs to be use.

The simple message is composed of prefix which contains the length of the message (header + data), a header containing the message type, com type and reply and then the data which may vary depending on the message type.

Prefix contains the following information:

- **Length**: signed 32 bit integer (4 bytes);

Header contains the following information:

- **MessageType**: signed 32 bit integer (4 bytes);
- **CommType**: signed 32 bit integer (4 bytes);
- **ReplyType**: signed 32 bit integer (4 bytes);

ROS Industrial already has defined some basic message type, but we find that some of those messages lack data to work properly with Motoman controller. Motoman specific message in the 2000 range were added but we've tried to follow the standard message model as much as possible.

Note that angular values are normally in radians. The ROS maximum number of axes is normally 10 axes but the Motoman maximum axes for a group is 8. ROS axes are define in the sequential order of the joint, whereas Motoman axes order for 7-axes robots places the 7th axis (E) value at index 2 between the 2nd (L) and 3rd (U) joints.

2.4.1 Message type 13: ROBOT_STATUS

The message called ROBOT_STATUS type which reflects the state of the robot/controller is designated as type 13.

Its data contains the following information:

Type	**Name**	**Description**
Int32	Drives_powered	Servo power -1 = Unknown; 1 = ON; 0 = OFF
Int32	E-stopped	Controller E-Stop state -1 = Unknown; 1 = TRUE(ON); 0 = FALSE(OFF)
Int32	Error_code	Alarm code of the first current alarm -1 = Unknown; 0 = No alarm; Other = Alarm#
Int32	In_error	There is at least one active alarm -1 = Unknown; 1 = TRUE(ON); 0 = FALSE(OFF)
Int32	In_motion	The controller is executing a job (program) -1 = Unknown; 1 = TRUE(ON); 0 = FALSE(OFF)
Int32	Mode	Controller/Pendant mode -1 = Unknown; 1 = Manual(Teach); 0 = Auto (Play or Remote)
Int32	Motion_possible	Controller can receive motion for ROS -1 = Unknown; 1 = Enabled; 0 = Disabled

2.4.2 Message type 14: JOINT_TRAJ_PT_FULL

The message called JOINT_TRAJ_PT_FULL type which includes position, velocity, acceleration and time is designated as type 14. This type of message contains sufficient trajectory data to accurately reproduce the trajectory generated by ROS with matching speed profile.

Its data contains the following information:

Type	**Name**	**Description**
Int32	Robot_id	Robot/group ID 0 = 1st robot
Int32	Sequence	Index of point in trajectory 0 = Initial trajectory point, which should match the robot current position.
Int32	Valid_fields	Bit-mask indicating which “optional” fields are filled with data. 1 = time, 2 = position, 4 = velocity, 8 = acceleration MotoROS expects all values, so this value should be set to 7.
Real32	Time	Timestamp associated with this trajectory point Units: in seconds
Real32	Positions[10]	Desired joint positions in radian. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.
Real32	Velocities[10]	Desired joint velocities in radian/sec. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.
Real32	Accelerations[10]	Desired joint accelerations in radian/sec^2. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.

2.4.3 Message type 15: JOINT_FEEDBACK

The message called JOINT_FEEDBACK type which includes position, velocity, acceleration and time is designated as type 15. At this time only the position field will be valid.

Its data contains the following information:

Type	**Name**	**Description**
Int32	Robot_id	Robot/group ID 0 = 1st robot
Int32	Valid_fields	Bit-mask indicating which "optional" fields are filled with data. 1 = time, 2 = position, 4 = velocity, 8 = acceleration MotoROS only send the position, so this value should be set to 2.
Real32	Time	Timestamp associated with this trajectory point Units: in seconds
Real32	Positions[10]	Desired joint positions in radian. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.
Real32	Velocities[10]	Desired joint velocities in radian/sec. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.
Real32	Accelerations[10]	Desired joint accelerations in radian/sec^2. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.

2.4.4 Message type 2001: MOTO_MOTION_CTRL

The message called MOTO_MOTION_CTRL is designated as type 2001. This message is used to send motion commands to control and manage the overall motion.

Its data contains the following information:

Type	Name	Description
Int32	Robot_id	Robot/group ID 0 = 1st robot
Int32	Sequence	Optional message tracking number that will be echoed back in the respond.
Int32	Command	Desired command: 200101 = CHECK_MOTION_READY 200102 = CHECK_QUEUE_CNT 200111 = STOP_MOTION 200121 = START_TRAJ_MODE 200122 = STOP_TRAJ_MODE
Real32	Data[10]	Reserved for future command use.

Command details:

CHECK_MOTION_READY (200101): Checks if the MotoROS/Controller side is ready to receive external motion data from ROS.

CHECK_QUEUE_CNT(200102): Return the number of motion increment currently in the queue. A return of 0 indicates that the queue is empty and all previous motion has been sent to the controller.

STOP_MOTION (200111): Stops robot motion immediately. Note after a stop, the current trajectory will be cleared and a new trajectory will need to be initiated but it doesn't necessarily turn off the MOTION_READY state.

START_TRAJ_MODE (200121): Signals MotoROS to set the controller in trajectory receiving mode so that ROS can start sending trajectory points.

STOP_TRAJ_MODE (200122): Signals the MotoROS hands controller back to the controller's INFORM job. Note that motion needs to complete or stopped before sending this command.

2.4.5 Message type 2002: MOTO_MOTION_REPLY

MotoROS sends this reply message each time it receives a joint trajectory message or a motoman motion control command.

Its data contains the following information:

Type	Name	Description
Int32	Robot_id	Robot/group ID 0 = 1st robot
Int32	Sequence	Reference to the sequence number that is being responded to.
Int32	Command	Reference to the command or message type that is being responded to. 14 = JOINT_TRAJ_PT_FULL 200101 = CHECK_MOTION_READY 200102 = CHECK_QUEUE_CNT 200111 = STOP_MOTION 200121 = START_TRAJ_MODE 200122 = STOP_TRAJ_MODE
Int32	Result	High level command result code: 0=SUCCESS/TRUE; 1=BUSY; 2=FAILURE/FALSE; 3=INVALID; 4= ALARM; 5=NOT_READY; 6=MP_FAILURE
Int32	Subcode	More detailed result code (optional)
Real32	Data[10]	Reserved for future command use.

Result details:

SUCCESS/TRUE (0): The message was processed successfully or the state is true.

BUSY (1): The message couldn't be processed at this time. Resend the message.

FAILURE/FALSE (2): The message couldn't be processed properly or the state is false.

INVALID (3): The message type is invalid or the data is incorrect.

ALARM (4): An alarm is currently active on the controller.

NOT_READY (5): The controller is not in ready for motion.

MP_FAILURE (6): MotoPlus API failure. Subcode=MotoPlus Error Code

2.4.6 Message type 2003: MOTO_ READ_SINGLE_IO

The message called MOTO_READ_SINGLE_IO is designated as type 2003. This message is used to read the current state of a specific I/O point of the controller.

Its data contains the following information:

Type	Name	Description
Int32	ioAddress	Address of the controller I/O signal to be read. Values from 00010 to 1000559, please refer to the controller Concurrent I/O Manual for details on addresses.

2.4.7 Message type 2004: MOTO_READ_SINGLE_IO_REPLY

MotoROS sends this reply message each time it receives a MOTO_READ_SINGLE_IO message.

Its data contains the following information:

Type	Name	Description
Int32	value	State of the I/O: 0 = OFF 1 = ON
Int32	resultCode	High level command result code: 1 = SUCCESS 2 = FAILURE

2.4.8 Message type 2005: MOTO_ WRITE_SINGLE_IO

The message called MOTO_ WRITE _SINGLE_IO is designated as type 2005. This message is used to read the current state of a specific I/O point of the controller.

Its data contains the following information:

Type	Name	Description
Int32	ioAddress	Address of the controller I/O signal to be written. Values from 10010 to 1000559; Note that some addresses are read only. Please refer to the controller Concurrent I/O Manual for details on addresses.
Int32	value	State of the I/O: 0 = OFF 1 = ON

2.4.9 Message type 2006: MOTO_READ_SINGLE_IO_REPLY

MotoROS sends this reply message each time it receives a MOTO_ WRITE _SINGLE_IO message.

Its data contains the following information:

Type	Name	Description
Int32	resultCode	High level command result code: 1 = SUCCESS 2 = FAILURE

2.4.10 Message type 2016: MOTO_ JOINT_TRAJ_PT_FULL_EX

The message called MOTO_JOINT_TRAJ_PT_FULL_EX type is designated as type 2016. It includes position, velocity, acceleration and time for multiple control groups. This type of message contains sufficient trajectory data to accurately reproduce the trajectory generated by ROS with matching speed profile.

Its data contains the following information:

Type	**Name**	**Description**
Int32	numberOfValidGroups	Indicates the amount data contained in the array that is valid. This normally correspond to the number of control group defined on the controller.
Int32	sequence	Index of point in trajectory 0 = Initial trajectory point, which should match the robot current position.
Array	jointTrajPtData	Array of JointTrajPtExData structure containing data for each control group motion. Array length= 4

JointTrajPtExData structure:

Type	**Name**	**Description**
Int32	Robot_id	Robot/group ID 0 = 1st robot
Int32	Valid_fields	Bit-mask indicating which “optional” fields are filled with data. 1 = time, 2 = position, 4 = velocity, 8 = acceleration MotoROS expects all values, so this value should be set to 7.
Real32	Time	Timestamp associated with this trajectory point Units: in seconds
Real32	Positions[10]	Desired joint positions in radian. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.
Real32	Velocities[10]	Desired joint velocities in radian/sec. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.
Real32	Accelerations[10]	Desired joint accelerations in radian/sec^2. Ordering matches the sequential joint order: SLURBT for 6 axis robot and SLEURBT for 7 axis robots.

2.4.11 Message type 2017: MOTO_ JOINT_FEEDBACK_EX

The message called MOTO_JOINT_FEEDBACK_EX type is designated as type 2017. It includes position, velocity, acceleration and time for multiple control groups. At this time only the position field will be valid.

Its data contains the following information:

Type	**Name**	**Description**
Int32	numberOfValidGroups	Indicates the amount data contained in the array that is valid. This normally correspond to the number of control group defined on the controller.
Array	jointTrajPtData	Array of JointFeedback structure containing data for each control group position. Array length= 4

JointFeedback structure is the same as the one used for the JOINT_FEEDBACK message. Please refer to section 2.4.3.

2.5 Motion Ready State

In order for the controller to accept external incremental motion, it must be in the MOTION_READY state. This state indicates that the controller is ready to receive trajectory from ROS. In order to be ready for motion, the following condition must be met:

- **Operating:** Job is playing back (PLAY mode + START or TEACH mode and INTERLOCK + TEST START)
- **WAIT instruction**: Job is executing a WAIT instruction. There is no explicit way of checking for the WAIT instruction other than actually sending mpExRcsIncrementMove and checking the return value for error.
- **Handshaking I/O signal is ON**: Since it is difficult to detect the execution of the WAIT instruction, an I/O signal is set just before the WAIT instruction to signal that the WAIT instruction has been reached.
- **REMOTE**: System is in REMOTE. The mpExRcsIncrementMove doesn't actually require the controller to be in REMOTE but other MotoPlus functions use to manage the system may require it and also for safety purpose, this condition is added.

The MOTION_READY state can be checked by sending a MOTO_MOTION_CTRL message with the command 1: CHECK_MOTION _READY.

Most perturbation of the system will cause the MOTION_READY state to turn off. Such perturbation include: Alarm, Error, E-stop, switching between REMOTE, PLAY and TEACH mode, Servo Off.

When the MOTION_READY state drops:

- Any further joint trajectory point messages are refused and the NOT_READY response code will be sent back.
- The incremental queue is cleared and no further increment will be sent to the controller.

To restart, the MOTION_READY state will need to be reestablish and a new trajectory will need to be generated by ROS using the current position of the robot as a start position.

If the system is not in the MOTION_READY state, sending a MOTO_MOTION_CTRL message with the command 200121: START_TRAJ_MODE command will initiate the following sequence to attempt to put the controller in the proper state:

- Reset alarm or errors
- Servo on
- Select the job :INIT_ROS (job containing sequence from section 2.1.3 or equivalent)
- Start the job execution

2.6 MotoROS to Controller communication

MotoROS exchanges data with the controller using the MotoPlus API. Please refer to the MotoPlus FS100 Language Reference manual and the mpExRcsIncrementMove document for details.

2.7 Interpolation of Pulse Increment

The MotoROS will receive trajectory points from the ROS Industrial. The trajectory points will include sequence number, time stamp, position (absolute) and velocity for each joint using angular radians units. The points maybe spread along a path at different spacing or time interval. The MotoROS application will need to interpolate the path between those points and determine the corresponding incremental move to send to the controller at the controller set interpolation period. The acceleration is modeled as a linear equation that will be resolved using the position, velocity and time data. The following are the calculation to be implemented for this interpolation.

2.7.1 Algorithm

For each new trajectory point:

Calculate acceleration equation coefficients.

While time is smaller than new ROS point time

Increment calculation time by next interpolation period

If next interpolation period is smaller than the controller interpolation period, make it equal.

If calculation time is smaller than new ROS point time

Set new time to calculation time

For each axis

Calculate position for the current calculation time

Calculate velocity for the current calculation time

Else (if calculation time is equal or larger than new ROS point time)

If calculation time is larger than new ROS point time

Set the next interpolation increment to the different between the two

Set the calculation time equal to the new ROS point time

For each axis

Set position to new ROS point position

Set velocity to new ROS point velocity

Convert new position in pulses

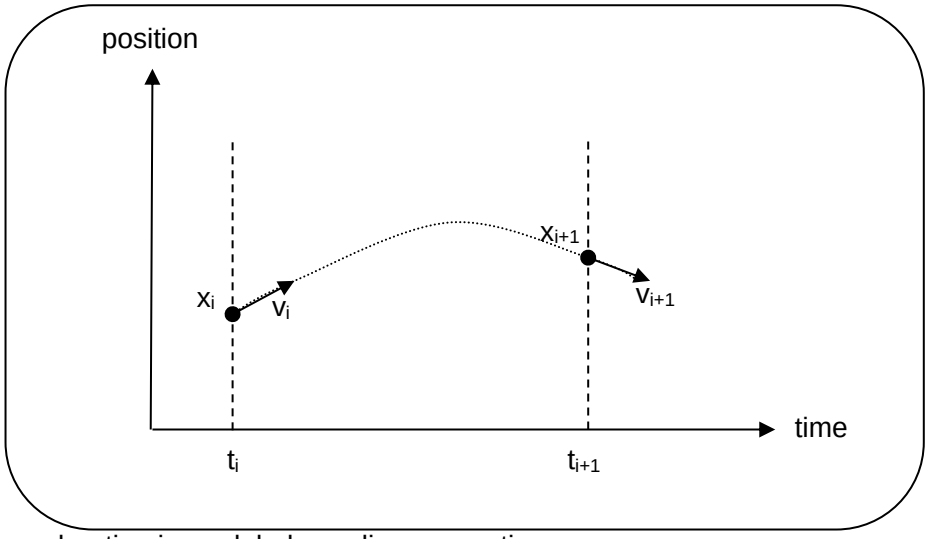
Calculate new pulse increment by subtracting previous pulse position from new pulse position.

Check incremental pulse validity

Add new pulse increment to queue

2.7.2 Calculation

This section covers the calculation made to interpolate points between two ROS control point. The known data is the position (x_i and x_{i+1}) and velocity (v_i and v_{i+1}) at the control points at the time t_i and t_{i+1} (received in the trajectory message from ROS).



The acceleration is modeled as a linear equation:

$$a = a_1 + a_2 t$$

The velocity at any given time is then determined by the equation:

$$v = v_i + \int_{t_i}^{t} a\, dt = v_i + \int_{t_i}^{t} (a_1 + a_2 t)\, dt$$

$$\boldsymbol{v = v_i + a_1(t - t_i) + \frac{a_2 (t-t_i)^2}{2}}$$

And the position by the equation:

$$x = x_i + \int_{t_i}^{t} v\, dt = x_i + \int_{t_i}^{t} (v_i + a_1 t + \frac{a_2 t^2}{2})\, dt$$

$$\boldsymbol{x = x_i + v_i(t - t_i) + a_1 \frac{(t-t_i)^2}{2} + \frac{a_2 (t-t_i)^3}{6}}$$

Using the control point data to resolve the equations, we find at time t_i that:

$$v_{i+1} = v_i + a_1(t_{i+1} - t_i) + \frac{a_2 (t_{i+1}-t_i)^2}{2}$$

And

$$x_{i+1} = x_i + v_i(t_{i+1} - t_i) + a_1 \frac{(t_{i+1}-t_i)^2}{2} + \frac{a_2 (t_{i+1}-t_i)^3}{6}$$

Using the two equations above, we can resolve the acceleration coefficients a_1 and a_2:

$$a_1 = \frac{(v_{i+1}-v_i)}{(t_{i+1}-t_i)} - \frac{a_2(t_{i+1}-t_i)}{2}$$

$$x_{i+1} = x_i + v_i(t_{i+1}-t_i) + \left(\frac{(v_{i+1}-v_i)}{(t_{i+1}-t_i)} - \frac{a_2(t_{i+1}-t_i)}{2}\right)\frac{(t_{i+1}-t_i)^2}{2} + \frac{a_2(t_{i+1}-t_i)^3}{6}$$

$$(x_{i+1}-x_i) = \frac{(v_{i+1}+v_i)(t_{i+1}-t_i)}{2} - \frac{a_2(t_{i+1}-t_i)^3}{12}$$

$$\frac{a_2(t_{i+1}-t_i)^3}{12} = -(x_{i+1}-x_i) + \frac{(v_{i+1}+v_i)(t_{i+1}-t_i)}{2}$$

$$\boldsymbol{a_2 = \frac{-12(x_{i+1}-x_i)}{(t_{i+1}-t_i)^3} + \frac{6(v_{i+1}+v_i)}{(t_{i+1}-t_i)^2}}$$

$$a_1 = \frac{(v_{i+1}-v_i)}{(t_{i+1}-t_i)} - \left(\frac{12(x_i-x_{i+1})}{(t_{i+1}-t_i)^3} + \frac{6(v_{i+1}+v_i)}{(t_{i+1}-t_i)^2}\right)\frac{(t_{i+1}-t_i)}{2}$$

$$\boldsymbol{a_1 = \frac{6(x_{i+1}-x_i)}{(t_{i+1}-t_i)^2} - \frac{2(v_{i+1}+2v_i)}{(t_{i+1}-t_i)}}$$

With those coefficients, the position and velocity for any time between t_i and t_{i+1} can be determined.

2.7.3 Check for incremental move validity

Check should be made to insure that any extreme value sent via ROS doesn't cause damage to the manipulator.

Initial test indicates that the controller already does checking for excessive segment and soft limits. Further testing should be done to determine what other validate might be required or not.

Check that might be required would be:

Acceleration:

If abs(newIncPulse[i] - prevIncPulse[i]) > MaxAccelPulse[i] generate error

3 Conclusion

The current design will give a good base for controlling robot motion through the ROS Industrial. It attempts to make an architecture that will allow expending functionality in the future. There are a few things that have not been fully detailed and will need to be refined as ROS industrial grows. Below are some suggestion of future developments and improvements.

3.1 Future development

In the case of multiple control group motion, it will have to be determined if the data for multiple robots/groups is pass together in one message to one motion server; or separately for each robot/group to separate motion server. Depending on the approach, better synchronization between motion groups may need to considered and elaborated further.

Currently the trajectory requires full details of time, positions and velocities. It would be possible to elaborate other cases to handle incomplete data by setting default values or behavior. Different trajectory data and profile could be added for simple motion to a point or for directly feeding incremental moves to the controller. The mpExRcsIncrementMove also supports Cartesian coordinate system which might facilitate data input.

The state server currently only broadcasts the joint feedback, this could also be elaborated on. As more options gets developed, simply broadcasting all the information might generate excessive traffic, so it would be desirable that the client be able to send request for the desired state, either subscribing to a broadcast or request a single reply message with a specific information.

4 Appendix

4.1 Port Numbers

The MotoPlus applications need to use a specific range of the ports on the controller. To avoid conflict between various MotoPlus application the ports 50240 to 50243 have been reserved for the MotoRos application.

```
TCP_PORT_MOTION    50240
TCP_PORT_STATE     50241
```

Ports 50242 and 50243 are not currently used but are reserve for future development.

4.2 IO Feedback

Controller universal output group 112 is being used by the MotoRos application to handshake with the controller job and to indicate the state of the MotoRos application.

```
IO_FEEDBACK_WAITING_MP_INCMOVE      11120  //output# 889
IO_FEEDBACK_MP_INCMOVE_DONE         11121  //output# 890
IO_FEEDBACK_MP_INITIALIZATION_DONE  11122  //output# 891
IO_FEEDBACK_CONNECTSERVERRUNNING    11123  //output# 892
IO_FEEDBACK_MOTIONSERVERCONNECTED   11124  //output# 893
IO_FEEDBACK_STATESERVERCONNECTED    11125  //output# 894
IO_FEEDBACK_FAILURE                 11127  //output# 896
```

The output #889 signals that the controller is ready to receive motion from ROS and should be set by the controller job.

The output #890 signals that ROS is done with moving the robot and the job execution can be resumed. The MotoRos turns on this output when the command STOP_TRAJ_MODE is received. The controller job normally turns it OFF before the next handshake.

The output #891 to #896 are set by MotoRos and should not be changed by the controller job (or operator). The output #891 confirms that the initialization was completed. The output #892 indicates that the server threads are running properly. Output #893 and #894 will only turn on when at least one client is connected and should turn off is all the clients disconnect. If output #896 is ON, a failure occurred and the controller will have to be rebooted in order to reset the MotoRos application.

4.3 Result Codes

```
ROS_RESULT_SUCCESS = 0,
ROS_RESULT_TRUE = 0,
ROS_RESULT_BUSY = 1,
ROS_RESULT_FAILURE = 2,
ROS_RESULT_FALSE = 2,
ROS_RESULT_INVALID = 3,
ROS_RESULT_ALARM = 4,
ROS_RESULT_NOT_READY = 5,
ROS_RESULT_MP_FAILURE = 6
```

4.4 Result subcodes

```
ROS_RESULT_INVALID_UNSPECIFIED = 3000,
ROS_RESULT_INVALID_MSGSIZE= 3001,
ROS_RESULT_INVALID_MSGHEADER = 3002,
ROS_RESULT_INVALID_MSGTYPE = 3003,
ROS_RESULT_INVALID_GROUPNO = 3004,
ROS_RESULT_INVALID_SEQUENCE = 3005,
ROS_RESULT_INVALID_COMMAND = 3006,
ROS_RESULT_INVALID_DATA = 3010,
ROS_RESULT_INVALID_DATA_START_POS = 3011,
ROS_RESULT_INVALID_DATA_POSITION = 3012,
ROS_RESULT_INVALID_DATA_SPEED = 3013,
ROS_RESULT_INVALID_DATA_ACCEL = 3014,
ROS_RESULT_INVALID_DATA_INSUFFICIENT = 3015

ROS_RESULT_NOT_READY_UNSPECIFIED = 5000,
ROS_RESULT_NOT_READY_ALARM = 5001,
ROS_RESULT_NOT_READY_ERROR = 5002,
ROS_RESULT_NOT_READY_ESTOP = 5003,
ROS_RESULT_NOT_READY_NOT_PLAY = 5004,
ROS_RESULT_NOT_READY_NOT_REMOTE = 5005,
ROS_RESULT_NOT_READY_SERVO_OFF = 5006,
ROS_RESULT_NOT_READY_HOLD = 5007,
ROS_RESULT_NOT_READY_NOT_STARTED = 5008,
ROS_RESULT_NOT_READY_WAITING_ROS = 5009
```

4.5 Reply Codes

Replies used for I/O Read and Write messages

```
ROS_REPLY_INVALID = 0,
ROS_REPLY_SUCCESS = 1,
ROS_REPLY_FAILURE = 2,
```

4.6 DX100 adaptation

For the DX100 version, the mpExRcsIncrementMove is replaced by the function mpMeiIncrementMove. The mpMeiIncrementMove API requires that the controller Job executes a SKILLSND instruction before the WAIT instruction. So, the instructions SKILLSND “ROS-START” and SKILLSND “ROS-STOP” need to be added before and after the WAIT OT#(890)=ON instruction in the RIS_INIT job. An extra task and related variables are added for the DX100 version to listen for the SKILLSND instruction.