ROS-I Motoman

Michaloski, John L. (Fed)

5/1/2017 11:04:00 AM

Motoman SIA-20-F-A00

Company: Yasakawa

Controller: fs100

DOF: 7

Joint names: joint\_s,joint\_l,joint\_e,joint\_u,joint\_r,joint\_b,joint\_t

Joint limits:

joint velocities:

Installing ROS-I Motoman (experimental development for indigo)

The github code can be found at: https://github.com/ros-industrial/motoman/tree/indigo-devel.

Having greped indigo and jade – same.

Clone the git source under a motoman\_ws/src.

michalos@woodsy:motoman\_ws> catkin init

michalos@woodsy:motoman\_ws> catkin build -DCMAKE\_BUILD\_TYPE=Debug

michalos@woodsy:motoman\_ws> source devel/setup.bash

Now we can display the motoman sia20 in rviz:

michalos@woodsy:motoman\_ws> roslaunch motoman\_sia20d\_support test\_sia20d.launch

robot\_state\_fs100.launch

<!--

Contoller specific version of 'robot\_interface.launch'.

Usage:

robot\_state\_fs100.launch robot\_ip:=<value>

-->

<launch>

<arg name="robot\_ip" />

<arg name="use\_bswap" value="true" />

<!--rosparam command="load" file="$(find motoman\_config)/?" /-->

<include file="$(find motoman\_driver)/launch/robot\_state.launch">

<arg name="robot\_ip" value="$(arg robot\_ip)" />

<arg name="use\_bswap" value="$(arg use\_bswap)" />

</include>

</launch>

robot\_state.launch

<!--

Wrapper launch file for the Motoman specific robot\_state node.

-->

<launch>

<!-- IP of robot (or PC running simulation) -->

<arg name="robot\_ip" />

<!-- Load the byte-swapping version of robot\_state if required -->

<arg name="use\_bswap" />

<!-- put them on the parameter server -->

<param name="robot\_ip\_address" type="str" value="$(arg robot\_ip)" />

<!-- load the correct version of the robot state node -->

<node if="$(arg use\_bswap)" name="joint\_state"

pkg="motoman\_driver" type="robot\_state\_bswap" />

<node unless="$(arg use\_bswap)" name="joint\_state"

pkg="motoman\_driver" type="robot\_state" />

</launch>

To run Motoman Sia20 connected to robot:

roslaunch moveit\_planning\_execution.launch sim=false controller=fs100 robot\_ip=????

..\motoman\_sia20d\_moveit\_config\launch\moveit\_planning\_execution.launch

<!-- the "sim" argument controls whether we connect to a Simulated or Real robot -->

<!-- - if sim=false, a robot\_ip and controller(fs100|dx100) arguments is required -->

<arg name="sim" default="true" />

<arg name="robot\_ip" unless="$(arg sim)" />

<arg name="controller" unless="$(arg sim)" />

. . .

<!-- run the "real robot" interface nodes -->

<!-- - this typically includes: robot\_state, motion\_interface, and joint\_trajectory\_action nodes -->

<!-- - replace these calls with appropriate robot-specific calls or launch files -->

<group unless="$(arg sim)">

<include file="$(find motoman\_sia20d\_support)/launch/robot\_interface\_streaming\_sia20d.launch" >

<arg name="robot\_ip" value="$(arg robot\_ip)"/>

<arg name="controller" value="$(arg controller)"/>

</include>

</group>

motoman\_sia20d\_support robot\_interface\_streaming\_sia20d.launch

<!--

Manipulator specific version of 'robot\_interface\_streaming.launch'.

Defaults provided for sia20d:

- 7 joints

Usage:

robot\_interface\_streaming\_sia20d.launch robot\_ip:=<value> controller:=<fs100|dx100>

-->

<launch>

<arg name="robot\_ip" />

<!-- controller: Controller name (fs100 or dx100) -->

<arg name="controller"/>

<rosparam command="load" file="$(find motoman\_sia20d\_support)/config/joint\_names\_sia20d.yaml" />

<include file="$(find motoman\_driver)/launch/robot\_interface\_streaming\_$(arg controller).launch">

<arg name="robot\_ip" value="$(arg robot\_ip)" />

</include>

</launch>

# Create a Joint Position Streaming Interface utilizing TCP Socket Libraries

These notes apply to creating the joint position streaming interface on an industrial controller using the ROS-Industrial TCP socket libraries. This interface is good for basic motion and proof of concept integration.  
  
Controller Code

### Joint Position Streaming Interface

The controller acts as a server for all connections. All types are 32 bit types (floats and integers, byte order is "Little Endian" (see [bug write up](http://code.google.com/p/swri-ros-pkg/issues/detail?id=8&can=1) for more information).

There are two connections, one for joint commands and one of joint feedback. Here are the details:

### Joint Commands (Motion)

|  |
| --- |
| Socket type: TCP |
| Socket port: 11000 |
| [JointMessages](http://wiki.ros.org/JointMessages) (constant values shown in **bold**) |
| Client request: LENGTH(bytes-not including length specifier), **10** (MSG\_ID – JOINT MESSAGE), **2** (COMM\_TYPE - REQUEST), **0** (REPLY\_TYPE – N/A), SEQ\_NUMBER(see special values below), JOINT\_DATA[10] (in rads (floats)) |
| Server reply: LENGTH(bytes), **10**, **3**(COMM\_TYPE – RESPONSE), REPLY (1 = SUCCESS, 2 = FAILURE), UNUSED, UNUSED[10] |

SEQ\_NUMBER – The sequence number is the number assigned by ROS to each of the points (i.e. this is always 0 or positive). A value of -1 indicates the end of a trajectory, a value of -2 indicates a stop (in both cases the controller stops, **joint data is not valid for these special types**)

### Pseudo-Code

The client will send joint points one at a time. When it receives the server response it sends the next point. This allows the server (controller) to determine the rate at which points are sent and how much buffering is performed. In practice this results in slower motion than desired, but the point spacing can be adjusted in ROS to account for this.

Pseudo-code for the controller is shown below (code for specific controllers might look very different given the differences between controllers).

while(true)

if(tcp.isConnected())

{

tcp.receive(jointPositionMsg)

jointPosition = decode(jointPositionMsg)

MoveJ(jointPosition)

jointResponseMsg = encode(jointResponse)

tcp.send(jointResponseMsg)

}

else

{

tcp.connect()

}

}

## Joint Feedback (Joint States)

|  |
| --- |
| Socket type: TCP |
| Socket port: 11002 |
| [JointMessages](http://wiki.ros.org/JointMessages) (constant values shown in **bold**) |
| Server streams: LENGTH(bytes), **10**, **1**(COMM\_TYPE – TOPIC), **0** (REPLY\_TYPE – N/A), UNUSED, JOINT\_DATA[10] (in rads (floats)) |

The Joint States just stream rate is determined by the controller, 10-50Hz is an appropriate range.

##### **Pseudo-Code**

Pseudo-code for the controller is shown below (code for specific controllers might look very different given the differences between controllers).

while(true)

if(tcp.isConnected())

{

jointCurrentMsg = encode(jointCurrent)

tcp.send(jointCurrentMsg)

sleep(rate)

}

else

{

tcp.connect()

}

}

MOTOMAN SIA 20d OPERATION

To operate:

1. Undo ESTOP
2. Set TEACH MODE key
3. JOINT W PEDNTANT must have deadman switch on side of TP ½ way down
4. SPEEED: LOW MED HIGH
5. COORD put button joint (robot skeleton) World (3 axis (0,0,0) ) or Tool (gripper outline )
6. BUTTON PLAY – PUSH SERVO ON (light should go on) THEN SWTCIH KEY TO AUTO

Questions/Problems:

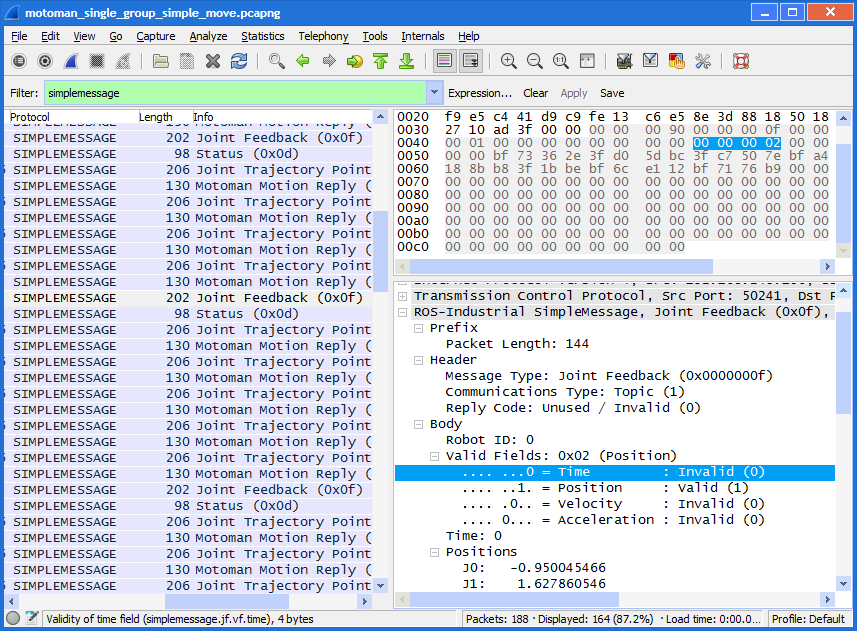
Topic list error?  
motion\_enabled = 0  
Parameter file on motoman must be updated – how?

# Wireshark Protocol Dissector

A Lua Wireshark dissector plugin for the simple message protocol is available from [ros-industrial/packet-simplemessage](https://github.com/ros-industrial/packet-simplemessage) at GitHub. See the readme for information on how to install it.

## packet-simplemessage v0.1.11 Overview

This is a Wireshark Lua dissector for the ROS-Industrial SimpleMessage protocol. For more information on the protocol, see [simple\_message](http://wiki.ros.org/simple_message). The current version of the dissector supports only the Groovy version of SimpleMessage (which is also used in Hydro and Indigo).

[](https://github.com/ros-industrial/packet-simplemessage/blob/master/sshot.png)

Packet types dissected:

* Ping
* Joint Position
* Joint Trajectory Point
* Status
* Joint Trajectory Point Full
* Joint Feedback
* Motoman Motion Control
* Motoman Motion Reply
* Motoman Read Single IO
* Motoman Read Single IO Reply
* Motoman Write Single IO
* Motoman Write Single IO Reply
* Motoman Joint Trajectory Point Full Extended
* Motoman Joint Feedback Extended

Tested on (but should work on other versions and OS as well):

* Windows
  + Wireshark 2.0.2 (from [wireshark.org/download](https://wireshark.org/#download))
* Linux (Ubuntu)
  + Wireshark 2.0.2 (from [ppa:wireshark-dev/stable](https://launchpad.net/~wireshark-dev/+archive/ubuntu/stable))

## Installation

Make sure the version of Wireshark you have installed was compiled with Lua support (see [wireshark.org/Lua](http://wiki.wireshark.org/Lua)).

If you're not interested in tracking development on the main branch, download the latest release from the [GitHub Releases](https://github.com/ros-industrial/packet-simplemessage/releases)page and extract the archive somewhere temporarily (Windows users will likely want to download the zip  archive).

## Linux (per user)

cd $PACKET\_SIMPLEMESSAGE

mkdir -p ~/.wireshark/plugins

cp packet-simplemessage.lua ~/.wireshark/plugins

## Windows (per user)

Open %USERPROFILE%\AppData\Roaming (Win7) or %USERPROFILE%\Application Data (WinXP) and open the Wireshark\pluginsfolder (if it doesn't exist, create it). Now copy packet-simplemessage.lua to the plugins folder.

## ROS-I Code that Implements ROS Client using Simple Message Protocol

The ROS Code to support this interface can be found in the dx100 package. This code can be reused for different controllers as it was developed genercially (**TODO: Move this generic code to a package under the ROS-Industrial stack**).



<launch>

<include file="$(find motoman\_sia20d\_support)/launch/load\_sia20d.launch" />

<param name="use\_gui" value="true" />

<node name="joint\_state\_publisher" pkg="joint\_state\_publisher" type="joint\_state\_publisher" />

<node name="robot\_state\_publisher" pkg="robot\_state\_publisher" type="robot\_state\_publisher" />

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find industrial\_robot\_client)/config/robot\_state\_visualize.rviz" required="true" />

</launch>

## MOTOMAN and INDUSTRIAL CORE TREE

├───src

│ ├───motoman

│ ├───motoman\_driver

│ │ ├───Inform

│ │ │ ├───DX100

│ │ │ │ ├───single\_arm

│ │ │ │ └───two\_arms\_no\_waist

│ │ │ ├───DX200

│ │ │ │ └───single\_arm

│ │ │ └───FS100

│ │ │ ├───dual\_arm\_sda

│ │ │ └───single\_arm

│ │ ├───MotoPlus

│ │ │ ├───\_buildLog

│ │ │ └───output

│ │ │ ├───DX100

│ │ │ │ └───MotoRosDX100

│ │ │ ├───DX200

│ │ │ │ └───MotoRosDX200

│ │ │ └───FS100

│ │ │ └───MotoRosFS100

│ │ ├───include

│ │ │ └───motoman\_driver

│ │ │ ├───industrial\_robot\_client

│ │ │ └───simple\_message

│ │ │ └───messages

│ │ ├───launch

│ │ └───src

│ │ ├───industrial\_robot\_client

│ │ └───simple\_message

│ │ └───messages

│ ├───motoman\_msgs

│ │ ├───msg

│ │ └───srv

│ ├───motoman\_sia20d\_moveit\_config

│ │ ├───config

│ │ └───launch

│ ├───motoman\_sia20d\_support

│ │ ├───config

│ │ ├───launch

│ │ ├───meshes

│ │ │ └───sia20d

│ │ │ ├───collision

│ │ │ └───visual

│ │ ├───test

│ │ └───urdf

│ └───motomangui

│ ├───src

│ │ └───rcsgui

│ ├───include

│ │ └───rcsgui

│ └───motomangui

│ └───NIST

├───devel

│ ├───etc

│ │ └───catkin

│ │ └───profile.d

│ ├───include

│ │ └───motoman\_msgs

│ ├───share

│ │ ├───catkin\_tools\_prebuild

│ │ │ └───cmake

│ │ ├───motoman\_msgs

│ │ │ └───cmake

│ │ ├───common-lisp

│ │ │ └───ros

│ │ │ └───motoman\_msgs

│ │ │ ├───srv

│ │ │ └───msg

│ │ ├───motoman\_driver

│ │ │ └───cmake

│ │ ├───motoman\_mh5\_support

│ │ │ └───cmake

│ │ ├───motoman\_sda10f\_support

│ │ │ └───cmake

│ │ ├───motoman\_sia10f\_support

│ │ │ └───cmake

│ │ ├───motoman\_sia10d\_support

│ │ │ └───cmake

│ │ ├───motoman\_sia20d\_support

│ │ │ └───cmake

│ │ ├───motoman\_sda10f\_moveit\_config

│ │ │ └───cmake

│ │ ├───motoman\_sia5d\_support

│ │ │ └───cmake

│ │ └───motoman\_sia20d\_moveit\_config

│ │ └───cmake

│ └───lib

│ ├───pkgconfig

│ ├───python2.7

│ │ └───dist-packages

│ │ └───motoman\_msgs

│ │ ├───srv

│ │ └───msg

│ └───motoman\_driver

├───logs

└───industrial\_core

├───industrial\_core

├───industrial\_deprecated

├───industrial\_msgs

│ ├───msg

│ └───srv

├───industrial\_robot\_client

│ ├───cmake

│ ├───config

│ ├───include

│ │ └───industrial\_robot\_client

│ ├───launch

│ ├───src

│ └───test

├───industrial\_robot\_simulator

│ └───launch

├───industrial\_trajectory\_filters

│ ├───include

│ │ └───industrial\_trajectory\_filters

│ └───src

├───industrial\_utils

│ ├───include

│ │ └───industrial\_utils

│ ├───src

│ └───test

└───simple\_message

├───cmake

├───include

│ └───simple\_message

│ ├───messages

│ └───socket

├───src

│ ├───messages

│ └───socket

└───test

### DebugLevel.msg

# Debug level message enumeration. This may replicate some functionality that

# alreay exists in the ROS logger.

# TODO: Get more information on the ROS Logger.

uint8 val

uint8 DEBUG = 5

uint8 INFO = 4

uint8 WARN = 3

uint8 ERROR = 2

uint8 FATAL = 1

uint8 NONE = 0

### DeviceInfo.msg

# Device info captures device agnostic information about a piece of hardware.

# This message is meant as a generic as possible. Items that don't apply should

# be left blank. This message is not meant to replace diagnostic messages, but

# rather provide a standard service message that can be used to populate standard

# components (like a GUI for example)

string model

string serial\_number

string hw\_version

string sw\_version

string address

### RobotMode.msg

# The Robot mode message encapsulates the mode/teach state of the robot

# Typically this is controlled by the pendant key switch, but not always

int8 val

# enumerated values

int8 UNKNOWN=-1 # Unknown or unavailable

int8 MANUAL=1 # Teach OR manual mode

int8 AUTO=2 # Automatic mode

### RobotStatus.msg

# The RobotStatus message contains low level status information

# that is specific to an industrial robot controller

# The header frame ID is not used

Header header

# The robot mode captures the operating mode of the robot. When in

# manual, remote motion is not possible.

industrial\_msgs/RobotMode mode

# Estop status: True if robot is e-stopped. The drives are disabled

# and motion is not possible. The e-stop condition must be acknowledged

# and cleared before any motion can begin.

industrial\_msgs/TriState e\_stopped

# Drive power status: True if drives are powered. Motion commands will

# automatically enable the drives if required. Drive power is not requred

# for possible motion

industrial\_msgs/TriState drives\_powered

# Motion enabled: Ture if robot motion is possible.

industrial\_msgs/TriState motion\_possible

# Motion status: True if robot is in motion, otherwise false

industrial\_msgs/TriState in\_motion

# Error status: True if there is an error condition on the robot. Motion may

# or may not be affected (see motion\_possible)

industrial\_msgs/TriState in\_error

# Error code: Vendor specific error code (non zero indicates error)

int32 error\_code

### ServiceReturnCode./msg

# Service return codes for simple requests. All ROS-Industrial service

# replies are required to have a return code indicating success or failure

# Specific return codes for different failure should be negative.

int8 val

int8 SUCCESS = 1

int8 FAILURE = -1

### TriState.msg

# The tri-state captures boolean values with the additional state of unknown

int8 val

# enumerated values

# Unknown or unavailable

int8 UNKNOWN=-1

# High state

int8 TRUE=1

int8 ON=1

int8 ENABLED=1

int8 HIGH=1

int8 CLOSED=1

# Low state

int8 FALSE=0

int8 OFF=0

int8 DISABLED=0

int8 LOW=0

int8 OPEN=0

## robot\_state\_visualize.launch

Simple launch file starts a robot visualization of the current hardware state as reported by the actual robot via the robot state node parameters. Parameters include:

1. robot\_ip - ip address of robot state server
2. urdf\_path - urdf path and filename relative to current directory (including the extension)

The usage is "robot\_state\_visualize.launch robot\_ip:=<value> urdf\_path:=<value>

For example: robot\_state\_visualize.launch robot\_ip:=192.168.1.10 urdf\_path:= urdf/M16iB20.urdf

<launch>

<arg name="robot\_ip" />

<arg name="urdf\_path" />

<!-- Populate parameters -->

<param name="robot\_description" textfile="$(arg urdf\_path)" />

<param name="robot\_ip\_address" type="str" value="$(arg robot\_ip)"/>

<!-- Required nodes -->

<node name="state\_publisher" pkg="robot\_state\_publisher" type="state\_publisher"/>

<node name="robot\_state" pkg="industrial\_robot\_client" type="robot\_state"/>

<node name="rviz" pkg="rviz" type="rviz" args="-d $(find industrial\_robot\_client)/config/robot\_state\_visualize.rviz"/>

</launch>

The class RobotStateInterface is a generic template that reads state-data from a robot controller and publishes matching messages to various ROS topics. Users should replace the default class members to implement robot-specific behavior.

#ifndef ROBOT\_STATE\_INTERFACE\_H

#define ROBOT\_STATE\_INTERFACE\_H

#include <vector>

#include <string>

#include "simple\_message/smpl\_msg\_connection.h"

#include "simple\_message/message\_manager.h"

#include "simple\_message/message\_handler.h"

#include "simple\_message/socket/tcp\_client.h"

#include "industrial\_robot\_client/joint\_relay\_handler.h"

#include "industrial\_robot\_client/robot\_status\_relay\_handler.h"

namespace industrial\_robot\_client

{

namespace robot\_state\_interface

{

using industrial::smpl\_msg\_connection::SmplMsgConnection;

using industrial::message\_manager::MessageManager;

using industrial::message\_handler::MessageHandler;

using industrial::tcp\_client::TcpClient;

using industrial\_robot\_client::joint\_relay\_handler::JointRelayHandler;

using industrial\_robot\_client::robot\_status\_relay\_handler::RobotStatusRelayHandler;

namespace StandardSocketPorts = industrial::simple\_socket::StandardSocketPorts;

/\*\*

\* \brief Generic template that reads state-data from a robot controller

\* and publishes matching messages to various ROS topics.

\*

\* Users should replace the default class members

\* to implement robot-specific behavior.

\*/

//\* RobotStateInterface

class RobotStateInterface

{

public:

/\*\*

\* \brief Default constructor.

\*/

RobotStateInterface();

/\*\*

\* \brief Initialize robot connection using default method.

\*

\* \param default\_ip default IP address to use for robot connection [OPTIONAL]

\* - this value will be used if ROS param "robot\_ip\_address" cannot be read

\* \param default\_port default port to use for robot connection [OPTIONAL]

\* - this value will be used if ROS param "~port" cannot be read

\*

\* \return true on success, false otherwise

\*/

bool init(std::string default\_ip = "", int default\_port = StandardSocketPorts::STATE);

/\*\*

\* \brief Initialize robot connection using specified method.

\*

\* \param connection new robot-connection instance (ALREADY INITIALIZED).

\*

\* \return true on success, false otherwise

\*/

bool init(SmplMsgConnection\* connection);

/\*\*

\* \brief Initialize robot connection using specified method and joint-names.

\*

\* \param connection new robot-connection instance (ALREADY INITIALIZED).

\* \param joint\_names list of joint-names for ROS topic

\* - Count and order should match data sent to robot connection.

\* - Use blank-name to skip (not publish) a joint-position

\*

\* \return true on success, false otherwise

\*/

bool init(SmplMsgConnection\* connection, std::vector<std::string>& joint\_names);

/\*\*

\* \brief Begin processing messages and publishing topics.

\*/

void run();

/\*\*

\* \brief get current robot-connection instance.

\*

\* \return current robot connection object

\*/

SmplMsgConnection\* get\_connection()

{

return this->connection\_;

}

/\*\*

\* \brief get active message-manager object

\*

\* \return current message-manager object

\*/

MessageManager\* get\_manager()

{

return &this->manager\_;

}

std::vector<std::string> get\_joint\_names()

{

return this->joint\_names\_;

}

/\*\*

\* \brief Add a new handler.

\*

\* \param new message-handler for a specific msg-type (ALREADY INITIALIZED).

\* \param replace existing handler (of same msg-type), if exists

\*/

void add\_handler(MessageHandler\* handler, bool allow\_replace = true)

{

this->manager\_.add(handler, allow\_replace);

}

protected:

TcpClient default\_tcp\_connection\_;

JointRelayHandler default\_joint\_handler\_;

RobotStatusRelayHandler default\_robot\_status\_handler\_;

SmplMsgConnection\* connection\_;

MessageManager manager\_;

std::vector<std::string> joint\_names\_;

};//class RobotStateInterface

}//robot\_state\_interface

}//industrial\_robot\_cliet

#include "industrial\_robot\_client/robot\_state\_interface.h"

#include "industrial\_utils/param\_utils.h"

using industrial::smpl\_msg\_connection::SmplMsgConnection;

using industrial\_utils::param::getJointNames;

namespace StandardSocketPorts = industrial::simple\_socket::StandardSocketPorts;

namespace industrial\_robot\_client

{

namespace robot\_state\_interface

{

RobotStateInterface::RobotStateInterface()

{

this->connection\_ = NULL;

this->add\_handler(&default\_joint\_handler\_);

this->add\_handler(&default\_robot\_status\_handler\_);

}

bool RobotStateInterface::init(std::string default\_ip, int default\_port)

{

std::string ip;

int port;

// override IP/port with ROS params, if available

ros::param::param<std::string>("robot\_ip\_address", ip, default\_ip);

ros::param::param<int>("~port", port, default\_port);

// check for valid parameter values

if (ip.empty())

{

ROS\_ERROR("No valid robot IP address found. Please set ROS 'robot\_ip\_address' param");

return false;

}

if (port <= 0)

{

ROS\_ERROR("No valid robot IP port found. Please set ROS '~port' param");

return false;

}

char\* ip\_addr = strdup(ip.c\_str()); // connection.init() requires "char\*", not "const char\*"

ROS\_INFO("Robot state connecting to IP address: '%s:%d'", ip\_addr, port);

default\_tcp\_connection\_.init(ip\_addr, port);

free(ip\_addr);

return init(&default\_tcp\_connection\_);

}

. . .

void RobotStateInterface::run()

{

manager\_.spin();

}

} // robot\_state\_interface

} // industrial\_robot\_client

During the method init, RobotStateInterface assigns message handlers(e.g., joint\_trajectory, robot\_status, ping) are assigned to the manager. RobotStateInterface als need to assign :

* TcpClient
* JointRelayHandler - is a message handler that relays joint positions (converts simple message types to ROS message types and publishes them). It inherits from public industrial::message\_handler::MessageHandler.
* RobotStatusRelayHandler - It requires a SmplMsgConnection connection i.e., simple message connection, that will be used to send replies (TO WHOM?)
* SmplMsgConnection
* MessageManager
* joint\_names\_ - the names of the robot joints. ROS depends on the robot joint names to resolved values to joints.

RobotStateInterface runs it uses the "manager" to handle the messages.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Topic | Exe | In | Out |  |
| joint\_states   ([sensor\_msgs/JointState](http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html)) | joint\_state\_publisher |  | X |  |
| \*  e.g., | joint\_state\_publisher |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Exe | In | Out | Parameters |
| joint\_state\_publisher | joint\_states   ([sensor\_msgs/JointState](http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html)) |  |  |
| industrial\_robot\_simulator | joint\_path\_command | joint\_states  feedback\_states | controller\_joint\_names  initial\_joint\_state  motion\_update\_rate pub\_rate |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |