Kortex Driver

Note: There have been many changes made between versions 1.1.7 and 2.0.0 of the ROS driver. You can view the changes and learn the steps to follow to adapt your code in this section.

Table of contents

Overview

This node allows communication between a ROS node and a Kinova Gen3 or Gen3 lite robot.

License

Author: Kinova inc.
Affiliation: Kinova inc.
Maintainer: Kinova inc. support@kinovarobotics.com

This package has been tested under ROS Noetic (Ubuntu 20.04).

For older ROS versions, checkout on corresponding branch :

- melodic-devel for ROS Melodic and Ubuntu 18.04 support.
- kinetic-devel for ROS Kinetic and Ubuntu 16.04 support, but the branch is no longer maintained (the melodic-devel branch might work for this configuration).

Usage

The `kortex_driver` node is the node responsible for the communication between the ROS network and the Kortex-compatible Kinova robots. It publishes topics that users can subscribe to. It also advertises services that users can call from the command line or from their own code to configure or control the robot arm or its sub-devices (actuators, vision module, interface module).

Arguments:

- **arm** : Name of your robot arm model. See the `kortex_description/arms` folder to see the available robot models. The default value is **gen3**.
- **dof** : Number of DOFs of your robot. The default value is for Gen3 is **7** and the default value for Gen3 lite is **6**. You will have to specify this value only if you have a Gen3 6DOF.
- **vision** : Boolean value to indicate if your arm has a Vision Module. The default value is for Gen3 is **true** and the default value for Gen3 lite is **false**. You will have to specify this value only if you have a Gen3 6DOF without a Vision Module. This argument only affects the visual representation of the arm in RViz.
- **gripper** : Name of your robot arm's tool / gripper. See the `kortex_description/grippers` folder to see the available end effector models (or to add your own). The default value is **""**. For Gen3, you can also put **robotiq_2f_85** or **robotiq_2f_140**. For Gen3 lite, the default (and only) gripper is **gen3_lite_2f**.
- **robot_name** : This is the namespace in which the driver will run. It defaults to **my_$(arg arm)** (so "my_gen3" for arm="gen3").
- **prefix** : This is an optional prefix for all joint and link names in the kortex_description. It is used to allow differentiating between different arms in the same URDF. It defaults to **an empty string**.
- **ip_address** : The IP address of the robot you're connecting to. The default value is **192.168.1.10**.
- **username** : The username for the robot connection. The default value is **admin**.
- **password** : The password for the robot connection. The default value is **admin**.
- **use_hard_limits** : [**Gen3 only**] If set to **true**, the arm's soft speed and acceleration limits are set to the hard limits and the MoveIt configuration uses those limits for the trajectories. If **false**, the default soft limit values are used. The default value for the parameter is **false**. **Be aware that setting this argument to true will set you arm's speed and acceleration limits to the maximum, so it will move way faster! Be cautious when using it for the first time as it may cause unwanted behaviour.**
- **cyclic_data_publish_rate** : Publish rate of the *base_feedback* and *joint_state* topics, in Hz. The default value is **40** Hz.
- **api_rpc_timeout_ms** : The default X-axis position of the robot in Gazebo. The default value is **0.0**.
- **api_session_inactivity_timeout_ms** : The duration after which the robot will clean the client session if the client hangs up the connection brutally (should not happen with the ROS driver). The default value is **35000** ms and is not normally changed.
- **api_connection_inactivity_timeout_ms** : The duration after which a connection is destroyed by the robot if no communication is detected between the client and the robot. The default value is **20000** ms and is not normally changed.
- **start_rviz** : If this argument is true, RViz will be launched. The default value is **true**.
- **start_moveit** : If this argument is true, a MoveIt! MoveGroup will be launched for the robot. The default value is **true**.

- **default_goal_time_tolerance** : The default goal time tolerance for the `FollowJointTrajectory` action server, in seconds. This value is used if no default goal time tolerance is specified in the trajectory. The default value is **0.5** seconds.
- **default_goal_tolerance** : The default goal tolerance for the `FollowJointTrajectory` action server, in radians. This value is used if no default goal tolerance is specified in the trajectory for the joint positions reached at the end of the trajectory. The default value is **0.005** radians.

To launch it with default arguments, run the following command in a terminal :

```
roslaunch kortex_driver kortex_driver.launch
```

To launch it with optional arguments, specify the argument name, then ":=", then the value you want. For example, :

```
roslaunch kortex_driver kortex_driver.launch ip_address:=10.0.100.239
start_rviz:=false robot_name:=terminator
```

You can also have a look at the roslaunch documentation for more details.

If everything goes well, you will see a "**The Kortex driver has been initialized correctly!**" message. If you also start MoveIt!, the `kortex_driver` output may be flooded in the `move_group` output, so pay attention to the warning and error messages! If the node fails to start for any reason, you will get an error message followed by a "**process has died**" message.

You will read below about the topics and services the driver offers. To read more about how to use those tools, go to the kortex_examples documentation.

Topics

Robot control topics (in)

You can publish on those topics for joint or Cartesian velocity control of the robot. You can also quickly stop its motion, apply the emergency stop (which will trigger a robot Fault state) and clear the Fault state with the corresponding topics.

Note: the robot's high level commands function at a rate of 40Hz. This will be improved in a future release.

- **`/your_robot_name/in/joint_velocity`**

  Publishing joint velocities (radians/second) on this topic will move the arm until it reaches a limit, or until you send a zero-velocity command or a Stop. You can see the message description here.

  From the command line, with your robot name being "my_gen3", you can publish a joint velocity to joint 0 to this topic like so:

  ```
  rostopic pub /my_gen3/in/joint_velocity kortex_driver/Base_JointSpeeds
  "joint_speeds:
  - joint_identifier: 0
  ```

```
    value: -0.57
    duration: 0"
```

- `/your_robot_name/in/cartesian_velocity`

  Publishing a Cartesian velocity (meters/second for linear, rad/second for angular) on this topic will move the arm until it reaches a limit, or until you send a zero-velocity command or a Stop. You can see the message description here.

  From the command line, with your robot name being "my_gen3", you can publish a twist to this topic like so:

```
rostopic pub /my_gen3/in/cartesian_velocity kortex_driver/TwistCommand
"reference_frame: 0
twist: {linear_x: 0.0, linear_y: 0.0, linear_z: 0.05, angular_x: 0.0,
angular_y: 0.0,
angular_z: 0.0}
duration: 0"
```

- `/your_robot_name/in/clear_faults`

  This clears the robot's fault state (if the faults are clearable).

  From the command line, with your robot name being "my_gen3", you can publish to this topic like so:

```
rostopic pub /my_gen3/in/clear_faults std_msgs/Empty "{}"
```

- `/your_robot_name/in/stop`

  This stops the robot's motion smoothly.

  From the command line, with your robot name being "my_gen3", you can publish to this topic like so:

```
rostopic pub /my_gen3/in/stop std_msgs/Empty "{}"
```

- `/your_robot_name/in/emergency_stop`

  This triggers a robot fault.

  From the command line, with your robot name being "my_gen3", you can publish to this topic like so:

```
rostopic pub /my_gen3/in/emergency_stop std_msgs/Empty "{}"
```

Robot feedback topics (out)

The robot feedback topics are always published by the `kortex_driver`. You don't have to activate them.

- `/your_robot_name/kortex_error`

  Every Kortex error will be published here. You can see the message description here.

- `/your_robot_name/base_feedback`

  The feedback from the robot is published on this topic at a rate of **cyclic_data_publish_rate**. You can see the message description here.

- `/your_robot_name/joint_state`

  The feedback from the robot is converted to a sensor_msgs/JointState and published on this topic at a rate of **cyclic_data_publish_rate**.

Notification topics (out)

The notification topics are only published by the `kortex_driver` if you activate them by first calling an activation service. Once activated, a notification topic will be activated until the node is shutdown.

Subscribing to all the notifications causes a heavy load on the robot CPU. That is why the notification topics were designed in such a way. The users also typically only use one or two notifications, if at all.

For example, if a user wants to subscribe to the **/my_robot_name/network_topic** (the message type is NetworkNotification), he will have to:

1. Call the **/my_robot_name/base/activate_publishing_of_network_notification** service to enable the publishing of the topic
2. Subscribe to the **/my_robot_name/network_topic** topic
3. Process the notifications when he receives them in his code

Services

Most of the services supported by this node are generated from the C++ Kortex API. You can find the documentation here.

Understanding packages

The *.srv* files are generated in different sub-folders depending on the sub-module they affect. For example, all the RPC calls used to configure the vision module are generated in `srv/generated/vision_config` and all the RPC calls common to all devices are generated in `srv/generated/device_config`. Here is a list of the packages with a short explanation of the services they have to offer:

- **actuator_config** : This package contains the functions used to configure a single actuator. **Note:** To choose the actuator you want to configure, you have to call the **/my_robot_name/actuator_config/set_device_id** service and specify the device identifier of the actuator you want to configure. You get the device identifiers of actuators when you launch the node, when you parse the output of the ReadAllDevices service or in the Kinova Kortex *Web App*.

- **base** : This package contains :

    - Services to read and update the configuration of the robot
    - Services to send high level commands to the robot
    - Services to read and update the Product Configuration
    - Services to activate the publishing of notifications
    - Services to read and update the user-related information **Note:** The base high level commands are treated every 25 ms inside the robot. High level control cannot be achieved at a rate faster than 40 Hz for now.

- **control_config** : This package contains the functions used to configure the control-related features on the robot. This includes :

    - Reading and setting the cartesian reference frame
    - Reading and setting the gravity vector
    - Reading and setting the payload information
    - Reading and setting the tool configuration

- **device_config** : This package contains the functions used to configure a generic Kortex device. This includes :

    - Reading and setting safety configurations
    - Reading general information on the specified device (software versions, serial numbers, MAC address, IPv4 settings, etc.) **Note:** To choose the device you want to configure, you have to call the **/my_robot_name/device_config/set_device_id** service and specify the device identifier of the device you want to configure. You get the device identifiers when you launch the node, when you parse the output of the ReadAllDevices service or in the WebApp.

- **device_manager** : This package contains ReadAllDevices service, which is used to get the list of connected device and various informations on each device.

- **interconnect_config** : This package contains the functions used to configure the interface module on the robot. **Note:** You don't have to call the `SetDeviceID` service before calling the **interconnect_config** services, because the `kortex_driver` node goes through the list of connected devices and automatically sets the correct device ID for the **interconnect_config** services.

- **vision_config** : This package contains the functions used to configure the vision module on the robot. **Note:** You don't have to call the SetDeviceID service before calling the **vision_config** services, because the kortex_driver node goes through the list of connected devices and automatically sets the correct device ID for the **vision_config** services.

Compatibility break between v1.1.7 and v2.0.0

Many things have been changed in the ros_kortex repository between versions 1.1.7 and 2.0.0 and you will have to modify your code if you don't want it to break.

- The `kortex_actuator_driver`, `kortex_vision_config_driver` and `kortex_device_manager` packages were removed and only the `kortex_driver` package remains (one driver to rule them all).
- Since we only have one driver and the ROS message generation does not deal with namespaces, the messages and services that are duplicated are now named differently. For example, the **Feedback** message exists within the `BaseCyclic`, `ActuatorCyclic`, `InterconnectCyclic` and

`GripperCyclic` Protocol Buffers .proto files. In ROS, this is now translated as a "PackageName_" prefix before the message name. So, for the **Feedback** message, the **BaseCyclic_Feedback**, **ActuatorCyclic_Feedback**, **InterconnectCyclic_Feedback** and **GripperCyclic_Feedback** ROS messages have been automatically generated. You may encounter build errors (in C++) or runtime errors (in Python) because of this change. You can just go in the `msg/generated` folder and look for the problematic message to find its new name to change the occurrences in your code.

- The services are now all **lowercase_with_underscores** instead of **UpperCase**.
- The services are now advertised in **/my_robot_name/my_package_name/desired_service** (see the Services section to learn about the packages). You can also visualize it if you start the node and type `rosservice list` in a terminal.
- The topics are now all **lowercase_with_underscores** instead of **UpperCase**.
- The **/my_robot_name/base_feedback/joint_state** topic is now advertised as **/my_robot_name/joint_state**.
- The kortex_driver launch file is now located in the `kortex_driver` package instead of the `kortex_bringup` package, which was deleted. Some arguments were added to the file.

About Conan

From release 2.2.0 onwards, the Kortex API is automatically downloaded from our Artifactory Conan server. The steps to install and setup Conan have been added to the root readme file. Conan downloads the binaries and header files in the Conan cache, by default situated in the `~/.conan/` directory.

If you want to learn more about Conan, you can read about it on their website.

If you still want to download the ZIP files for the API, you can find the link in the Kortex repository.

You will have to extract the API in the `kortex_api` folder as such:

```
kortex_api/
┬
├ include/
└ lib/
```

You will then have to build the catkin workspace and pass it the option to disable Conan so it links with your local API:

```
catkin_make -DUSE_CONAN=OFF
```

Support for multiple arms

The kortex_driver launch file is primarily used to define one-armed robots. Having more than one arm requires prefixing the joints and links to protect against ambiguity when refering those. To this end, a kortex_dual_driver launch file has been made to demonstrate running a two-armed robot. The same pattern could be repeated to describe a robot with any number of arms.

Note: This launch file is intended as a starting point showing how to launch a robot with two arms. When using this launch file for a specific robot, it should be modified to fit the robot's description.

The launch file uses the same parameters as the one-armed version, except that most parameters are to be defined individually for each arm (with the **left_** and **right_** prefixes).

The prefixes used in the joints and links names can be changed using the following parameters:

- `left_prefix`
- `right_prefix`

The following parameters are to be specified for each robot (using `left_` and `right_` prefixes):

- `arm`
- `dof`
- `vision`
- `gripper`
- `ip_address`
- `username`
- `password`
- `cyclic_data_publish_rate`
- `api_rpc_timeout_ms`
- `api_session_inactivity_timeout_ms`
- `api_connection_inactivity_timeout_ms`
- `default_goal_time_tolerance`
- `default_goal_tolerance`

The folowing parameters are **NOT** to be prefixed:

- `robot_name`
- `start_rviz`
- `start_moveit`
- `cyclic_data_publish_rate`

These parameters are common to both arms.

Example use : `roslaunch kortex_driver kortex_dual_driver.launch robot_name:=terminator left_ip_address:=192.168.1.11 left_arm:=gen3 left_gripper:=robotiq_2f_85 right_ip_address:=192.168.1.12 right_arm:=gen3 right_gripper:=robotiq_2f_85 start_moveit:=false`

Generation (**advanced**)

Some source code as well as most of the .MSG and .SRV files in this package are automatically generated, but the generated files are given on GitHub so that users don't have to generate them. However, if you have a special version of the Kortex API and want to generate those files yourself, it is possible. You will first need to follow the instructions to install Protocol Buffers.

The generation process is based on a custom `protoc` plugin. Basically, most of the generation process is in the scripts/ros_kortex_generator.py. Before launching the generation ensure that you have the Python

JINJA2 module installed.

To launch the generation of this package:

1. Open a terminal window.
2. Browse the /scripts directory of this package
3. Ensure that the generate_protobuf_wrapper_files.sh file can be executed. If not then run: `chmod +x generate_protobuf_wrapper_files.sh`
4. Run the command: `./generate_protobuf_wrapper_files.sh`
5. The result of the generation should be in the following folders:
    - `/include/kortex_driver/generated`
    - `/msg/generated`
    - `/src/generated`
    - `/srv/generated`

Protos files

The **protos** directory contains the Protobuf files from which the MSG, SRV and source files are generated. The content of this folder should not be modified.

Template files

The **templates** directory contains all the JINJA2 files needed by the `protoc` generator.