Creating vehicle interface

How to implement a vehicle interface

The following instructions describe how to create a vehicle interface.

1. Create a directory for vehicle interface

It is recommended to create your vehicle interface at `<your-autoware-dir>/src/vehicle/external`

```
cd <your-autoware-dir>/src/vehicle/external
```

2. Install or implement your own vehicle interface

If there is an already complete vehicle interface package (like `pacmod_interface`), you can install it to your environment. If not, you have to implement your own vehicle interface by yourself. Let's create a new package by `ros2 pkg create`. The following example will show you how to create a vehicle interface package named `my_vehicle_interface`.

```
ros2 pkg create --build-type ament_cmake my_vehicle_interface
```

Then, you should write your implementation of vehicle interface in `my_vehicle_interface/src`. Again, since this implementation is so specific to the control device of your vehicle, it is beyond the scope of this document to describe how to implement your vehicle interface in detail. Here are some factors that might be considered:

- Some necessary topic subscription of control commands topics from Autoware to control your vehicle:

Topic Name	Topic Type
/control/command/control_cmd	autoware_auto_control_msgs/msg/AckermannControlCom

Topic Name	Topic Type
/control/command/gear_cmd	autoware_auto_vehicle_msgs/msg/GearCommand
/control/current_gate_mode	tier4_control_msgs/msg/GateMode
/control/command/emergency_cmd	tier4_vehicle_msgs/msg/VehicleEmergencyStamped
/control/command/turn_indicators_cmd	autoware_auto_vehicle_msgs/msg/TurnIndicatorsCommar
/control/command/hazard_lights_cmd	autoware_auto_vehicle_msgs/msg/HazardLightsCommanc

Topic Name	Topic Type
/control/command/actuation_cmd	tier4_vehicle_msgs/msg/ActuationCommandStamped
etc.	etc.

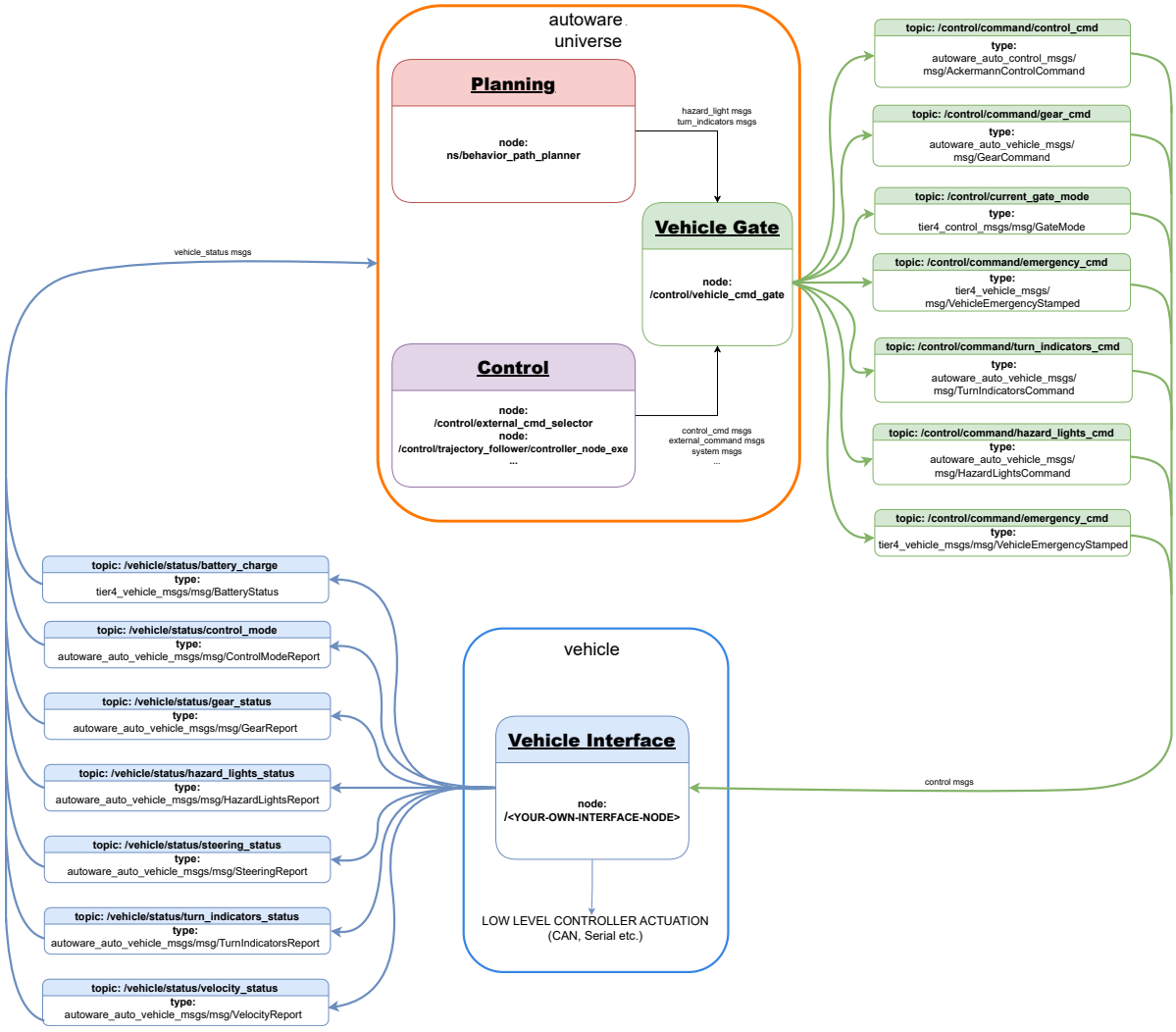
- Some necessary topic publication of vehicle status topics from vehicle interface to Autoware:

Topic Name	Topic Type
/vehicle/status/battery_charge	tier4_vehicle_msgs/msg/BatteryStatus
/vehicle/status/control_mode	autoware_auto_vehicle_msgs/msg/ControlModeReport

Topic Name	Topic Type
/vehicle/status/gear_status	autoware_auto_vehicle_msgs/msg/GearReport
/vehicle/status/hazard_lights_status	autoware_auto_vehicle_msgs/msg/HazardLightsReport
/vehicle/status/turn_indicators_status	autoware_auto_vehicle_msgs/msg/TurnIndicatorsReport
/vehicle/status/steering_status	autoware_auto_vehicle_msgs/msg/SteeringReport
/vehicle/status/velocity_status	autoware_auto_vehicle_msgs/msg/VelocityReport

Topic Name	Topic Type
etc.	etc.

This diagram as an example for communication of vehicle interface and autoware with describing sample topics and message types.



Sample demonstration of vehicle and autoware communication. There are some topics and types included in this diagram and it can be changed your desired

control command or autoware updates.

You must create a subscriber and publisher with these topics on your vehicle interface. Let's explain with the simple demonstration of subscribing `/control/command/control_cmd` and publishing `/vehicle/status/gear_status` topics.

So, your `YOUR-OWN-VEHICLE-INTERFACE.hpp` header file should be like this:

```
...
#include <autoware_auto_control_msgs/msg/ackermann_control_command.hpp>
#include <autoware_auto_vehicle_msgs/msg/gear_report.hpp>
...

class <YOUR-OWN-INTERFACE> : public rclcpp::Node
{
public:
    ...
private:
    ...
    // from autoware

rclcpp::Subscription<autoware_auto_control_msgs::msg::AckermannControlCommand>::S
haredPtr
    control_cmd_sub_;
    ...
    // from vehicle
    rclcpp::Publisher<autoware_auto_vehicle_msgs::msg::GearReport>::SharedPtr
gear_status_pub_;
    ...
    // autoware command messages
    ...
    autoware_auto_control_msgs::msg::AckermannControlCommand::ConstSharedPtr
control_cmd_ptr_;
    ...
    // callbacks
    ...
    void callback_control_cmd(
    const
autoware_auto_control_msgs::msg::AckermannControlCommand::ConstSharedPtr msg);
    ...
    void to_vehicle();
    void from_vehicle();
}
```

And your `YOUR-OWN-VEHICLE-INTERFACE.cpp` .cpp file should be like this:

```
#include <YOUR-OWN-VEHICLE-INTERFACE>/<YOUR-OWN-VEHICLE-INTERFACE>.hpp>
...
```

```
<YOUR-OWN-VEHICLE-INTERFACE>::<YOUR-OWN-VEHICLE-INTERFACE>()
: Node("<YOUR-OWN-VEHICLE-INTERFACE>")
{
  ...
  /* subscribers */
  using std::placeholders::_1;
  // from autoware
  control_cmd_sub_ =
create_subscription<autoware_auto_control_msgs::msg::AckermannControlCommand>(
    "/control/command/control_cmd", 1, std::bind(&<YOUR-OWN-VEHICLE-
INTERFACE>::callback_control_cmd, this, _1));
  ...
  // to autoware
  gear_status_pub_ =
create_publisher<autoware_auto_vehicle_msgs::msg::GearReport>(
    "/vehicle/status/gear_status", rclcpp::QoS{1});
  ...
}

void <YOUR-OWN-VEHICLE-INTERFACE>::callback_control_cmd(
  const autoware_auto_control_msgs::msg::AckermannControlCommand::ConstSharedPtr
msg)
{
  control_cmd_ptr_ = msg;
}

void <YOUR-OWN-VEHICLE-INTERFACE>::to_vehicle()
{
  ...
  // you should implement this structure according to your own vehicle design
  control_command_to_vehicle(control_cmd_ptr_);
  ...
}

void <YOUR-OWN-VEHICLE-INTERFACE>::to_autoware()
{
  ...
  // you should implement this structure according to your own vehicle design
  autoware_auto_vehicle_msgs::msg::GearReport gear_report_msg;
  convert_gear_status_to_autoware_msg(gear_report_msg);
  gear_status_pub_->publish(gear_report_msg);
  ...
}
```

- Modification of control values if needed
  - In some cases, you may need to modify the control commands. For example, Autoware expects vehicle velocity information in m/s units, but if your vehicle publishes it in a different format (i.e., km/h), you must convert it before sending it to Autoware.

3. Prepare a launch file

After you implement your vehicle interface, or you want to debug it by launching it, create a launch file of your vehicle interface, and include it to `vehicle_interface.launch.xml` which included in `<VEHICLE_ID>_vehicle_launch` package that we forked and created at creating vehicle and sensor model page.

Do not get confused. First, you need to create a launch file for your own vehicle interface module (like `my_vehicle_interface.launch.xml`) **and then include that to `vehicle_interface.launch.xml` which exists in another directory.** Here are the details.

1. Add a `launch` directory in the `my_vehicle_interface` directory, and create a launch file of your own vehicle interface in it. Take a look at Creating a launch file in the ROS 2 documentation.
2. Include your launch file which is created for vehicle_interface to `<YOUR-VEHICLE-NAME>_launch/<YOUR-VEHICLE-NAME>_launch/launch/vehicle_interface.launch.xml` by opening it and add the included terms like below.

vehicle_interface.launch.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<launch>
    <arg name="vehicle_id" default="$(env VEHICLE_ID default)"/>
    <!-- please add your created vehicle interface launch file -->
    <include file="$(find-pkg-share
my_vehicle_interface)/launch/my_vehicle_interface.launch.xml">
    </include>
</launch>
```

Finally, your directory structure may look like below. Most of the files are omitted for clarity, but the files shown here needs modification as said in the previous and current process.

```
<your-autoware-dir>/
└─ src/
    └─ vehicle/
        ├─ external/
+       │   └─ <YOUR-VEHICLE-NAME>_interface/
+       │       ├─ src/
+       │       └─ launch/
+       │           └─ my_vehicle_interface.launch.xml
+       └─ <YOUR-VEHICLE-NAME>_launch/ (COPIED FROM sample_vehicle_launch)
+           ├─ <YOUR-VEHICLE-NAME>_launch/
+           │   ├─ launch/
+           │   │   └─ vehicle_interface.launch.xml
+           │   ├─ CMakeLists.txt
```

```
+           │   └─ package.xml
+           ├─ <YOUR-VEHICLE-NAME>_description/
+           │   ├─ config/
+           │   ├─ mesh/
+           │   ├─ urdf/
+           │   │   └─ vehicle.xacro
+           │   ├─ CMakeLists.txt
+           │   └─ package.xml
+           └─ README.md
```

4. Build the vehicle interface package and the launch package

Build three packages `my_vehicle_interface`, `<YOUR-VEHICLE-NAME>_launch` and `<YOUR-VEHICLE-NAME>_description` by `colcon build`, or you can just build the entire Autoware if you have done other things.

```
colcon build --symlink-install --cmake-args -DCMAKE_BUILD_TYPE=Release --
packages-select my_vehicle_interface <YOUR-VEHICLE-NAME>_launch <YOUR-VEHICLE-
NAME>_description
```

5. When you launch Autoware

Finally, you are done implementing your vehicle interface module! Be careful that you need to launch Autoware with the proper `vehicle_model` option like the example below. This example is launching planning simulator.

```
ros2 launch autoware_launch planning.launch.xml
map_path:=$HOME/autoware_map/sample-map-planning vehicle_model:=<YOUR-VEHICLE-
NAME> sensor_model:=<YOUR-VEHICLE-NAME>_sensor_kit
```

Tips

There are some tips that may help you.

- You can subdivide your vehicle interface into smaller packages if you want. Then your directory structure may look like below (not the only way though). Do not forget to launch all packages in `my_vehicle_interface.launch.xml`.

  ```
  <your-autoware-dir>/
  └─ src/
      └─ vehicle/
          ├─ external/
          │   └─ my_vehicle_interface/
  ```

```
        │       ├─ src/
        │       │   ├─ package1/
        │       │   ├─ package2/
        │       │   └─ package3/
        │       └─ launch/
        │           └─ my_vehicle_interface.launch.xml
        ├─ sample_vehicle_launch/
        └─ my_vehicle_name_launch/
```

- If you are using a vehicle interface and launch package from a open git repository, or created your own as a git repository, it is highly recommended to add those repositories to your `autoware.repos` file which is located to directly under your autoware folder like the example below. You can specify the branch or commit hash by the version tag.

  autoware.repos

  ```
  # vehicle (this section should be somewhere in autoware.repos and add the
  below)
  vehicle/external/my_vehicle_interface:
    type: git
    url: https://github.com/<repository-name-B>/my_vehicle_interface.git
    version: main
  ```

  Then you can import your entire environment easily to another local device by using the `vcs import` command. (See the source installation guide)