

Deployment Manual

1. Generating C++ Code

To begin, we need to create a greedy policy block for our trained RL agent. This block takes the observation signals as outputs and connects them through a MUX block to merge them into a vector. On the other side, we create an input for the action that the RL agent is trained to perform. This will be the model for which we will generate C++ code(Figure 1).

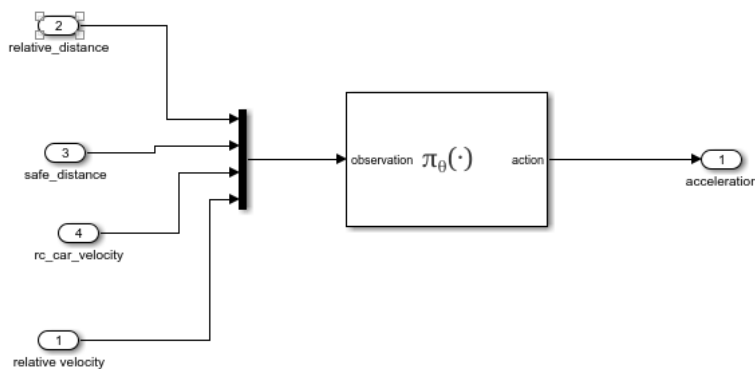


Figure 1 Simulink format used to generate C++ from trained RL agent

Before proceeding, we need to configure some settings. In the Simulink model navigate to the "Modeling" tab and access the "Model Settings" menu. In the hardware implementation section, set the Device Vendor to "ARM Compatible" and the Device Type to "ARM Cortex-A." These settings are specific to an NVIDIA Jetson Xavier-NX running Ubuntu 18.04. If you are using a different embedded controller, please refer to the device's documentation and operating system to determine the appropriate device type and vendor.

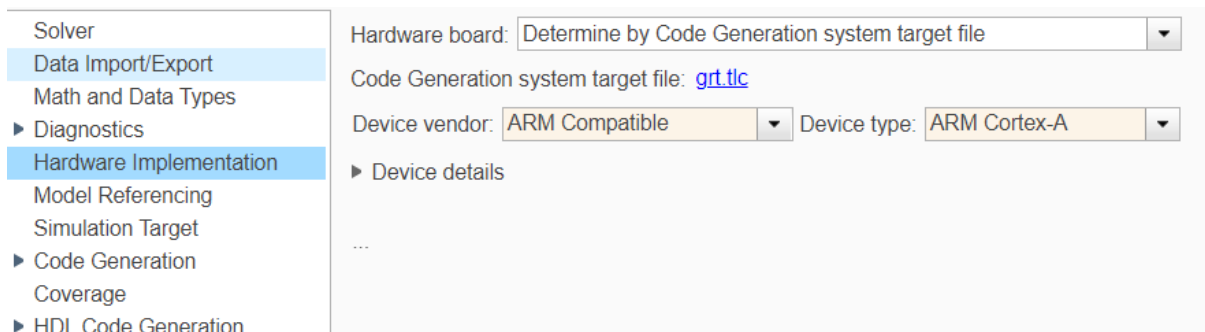


Figure 2 Hardware Implementation settings

Next, go to the Code Generation section and modify the following settings:

- Set the System Target File to "ert.tlc."
- Set the Language to "C++."
- Set the Language Standard to "C++11 (ISO)."
- The toolchain is typically selected automatically, so you don't need to worry about it. Just ensure that it is a C++ toolchain.

By adjusting these settings, we can proceed with generating C++ code for the model.

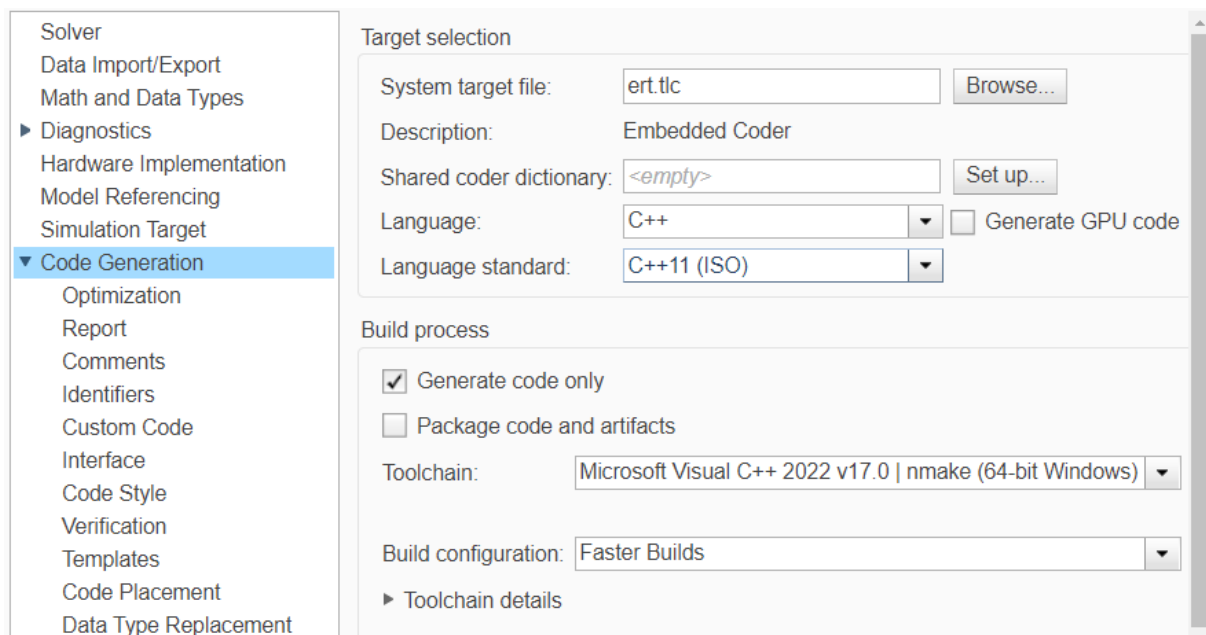


Figure 3 Code Generation Settings

Now, within your Simulink model, go to the "Apps" tab and click on the "Embedded Coder App". This will cause an additional section to appear in your Simulink model. Next, click on "Generate Code" and then click on "Generate Code" again, as illustrated in Figure 4.

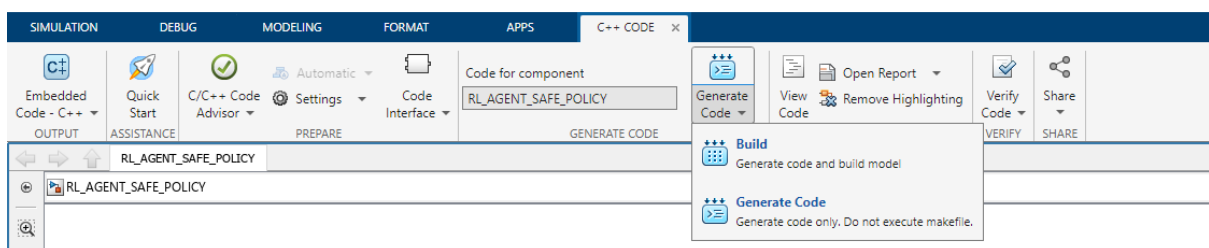


Figure 4 Embedded Coder App

If everything went smoothly, you should now have generated two C++ files and four header files, as depicted in Figure 5. The generated files, including the two C++ files and four header files, are stored in a folder that ends with "ert_rtw" within your current directory (See Figure 6). Although there may be additional files present, you only need to focus on the mentioned C++ and header files. To extract them, simply copy these six files onto a USB drive. These files contain the necessary code to execute our RL policy in C++, mirroring the functionality we had in Simulink during the testing setup in the simulation manual.

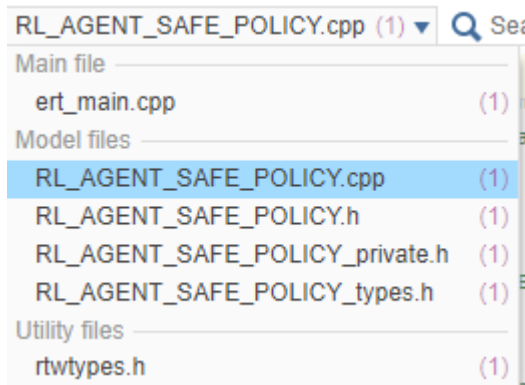


Figure 5 Generated files from embedded coder app

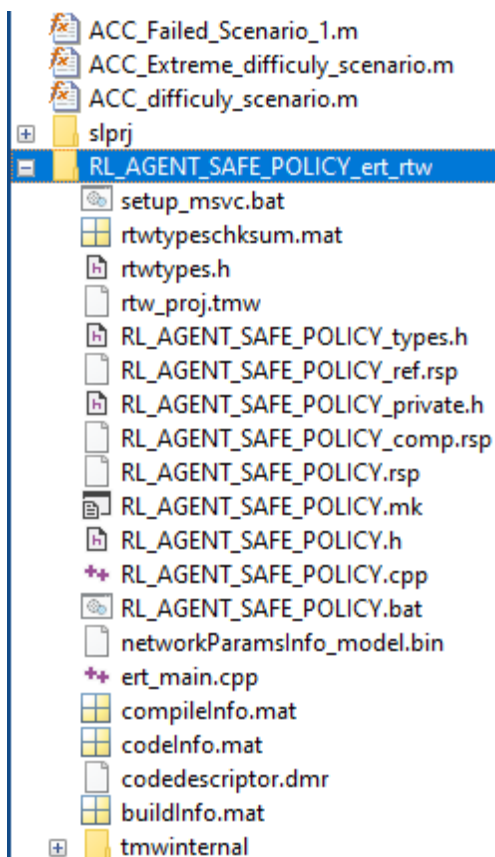


Figure 6 C++ Folder

2. Deploying C++ Code

2.1. Installation Guide

If you are using the non-formatted NVIDIA-jetson Xavier-Nx I worked on you can skip this section.

2.1.1. Installing ROS

- `sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'`

- `sudo apt install curl` # if you haven't already installed curl
- `curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -`
- `sudo apt install ros-melodic-ros-base`

`nano ~/.bashrc`

and add these lines in the end of the bashrc file.

```
source /opt/ros/melodic/setup.bash
source ~/catkin_ws/devel/setup.bash
```

2.1.2. Create catkin workspace

- `mkdir -p ~/catkin_ws/src`
- `cd ~/catkin_ws`
- `catkin_make`

2.1.3. Installing Fltenth system (for VESC controller)

- `cd ~/catkin_ws`
- `git clone https://github.com/fltenth/fltenth_system/tree/melodic`
- `mkdir -p fltenth_ws/src`
- `cp -r fltenth_system fltenth_ws/src`
- `sudo apt-get update`
- `sudo apt-get install ros-melodic-driver-base`
- `catkin_make`

2.1.4. Install RPLIDAR package for ROS

- `cd ~/catkin_ws/src`
- `git clone https://github.com/robopeak/rplidar_ros.git`

- `cd ~/catkin_ws/src`
- `catkin_make`

2.2. Test hardware

Before merging our C++ code with our hardware, we need to ensure that our hardware is communicating correctly through ROS topics. To verify this, I have created some test files to check the relative distance readings from the RPLIDAR A3 and the controllability of the motor using the VESC controller. These test files will help us validate the proper functioning of our hardware components before integrating them with the C++ code.

To test the hardware run the following commands

In the first Terminal

- `roscore`

In a second Terminal

For Lidar hardware check

- `roslaunch my_lidar_package lidarscan`

For Motor hardware check

- `roslaunch my_lidar_package motorcontrol`

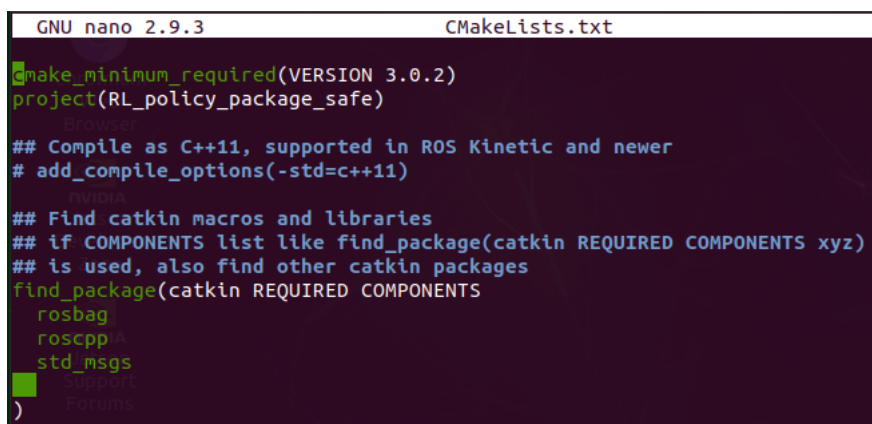
2.3. Deploy C++ code

If the hardware is functioning properly, we have reached the final step of implementing our C++ code on the NVIDIA Jetson. Take the USB stick where you saved the generated C++ and header files and transfer them to the NVIDIA-jetson. After this is done, we will create our catkin workspace.

- `cd ~/catkin_ws/src`
- `catkin_create_pkg packagename`
- `cd ~/catkin_ws/src/packagename`
- `mkdir src`

The second step involves taking our two C++ files and four header files generated using the Matlab Embedded Coder App and placing them inside **`~/catkin_ws/src/packagename/src/`**

We also need to alter the CMakeLists.txt file of our catkin package. We need to include certain packages, such as rosbag, which allows us to store our observation and action signals in a rosbag file. This enables us to graph these values at a later time without the need to visualize them in real-time. If you don't want to use rosbag you can leave it out of the required components section. But roscpp and std_msgs are crucial to include.



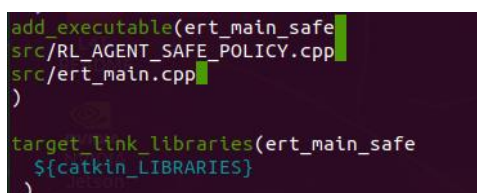
```
GNU nano 2.9.3 CMakeLists.txt
cmake_minimum_required(VERSION 3.0.2)
project(RL_policy_package_safe)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  rosbag
  roscpp
  std_msgs
)
)
```

Figure 7 Required Components CMakeLists.txt

Next, you will need to add the **`add_executable`** section to your code. Where you will add the two C++ files as executables as show in Figure 8. The name `ert_main_safe` is the name of the executable which we will use to run our code. Additionally, remember to include the **`target_link_libraries`** section in your code, as illustrated in Figure 8.



```
GNU nano 2.9.3 CMakeLists.txt
add_executable(ert_main_safe
  src/RL_AGENT_SAFE_POLICY.cpp
  src/ert_main.cpp
)

target_link_libraries(ert_main_safe
  ${catkin_LIBRARIES}
)
)
```

Figure 8 add executable CMakeLists.txt

Once you have completed all the necessary steps, return to the **catkin_ws** directory and run **catkin_make** again. This command will compile the code you have just added and create an executable. It's important to note that whenever you make changes to your code, you will need to go back to the **catkin_ws** directory and execute the **catkin_make** command to compile the code once again.

2.4. Run the complete deployment setup.

It's important to highlight that significant changes have been made to the generated **ert_main.cpp** and **RL_AGENT_SAFE_POLICY.cpp** files to establish communication with the hardware through ROS and incorporate other necessary logic for creating a self-driving car. To fully understand how to integrate ROS communication and other essential logic into these generated C++ files, I recommend visiting my GitHub repository, where you can find the complete code and detailed explanations.

https://github.com/DanielSmetankin/MasterThesis_SelfDrivingCar.git

To run the complete setup, you will need four terminals. Terminal 1 to set up roscore, which starts a ROS Master, a ROS Parameter Server, and a roslaunch node. Essentially, roscore functions as a router for our ROS topics. Terminal 2 for running the RPLIDAR node, which reads out the relative distance using the RPLIDAR_A3 sensor. Terminal 3 to run the VESC controller node, which manages several topics. However, in my case, I will only be utilizing one topic to control the motor speed. Terminal 4 to run our custom-made node that contains our RL greedy policy and other necessary logic for controlling the self-driving car.

By using these four terminals, you can set up and run the complete system, integrating ROS communication, sensor readings, motor control, and the custom logic for autonomous driving.

1st Terminal

```
roscore
```

2nd Terminal

```
roslaunch vesc_drive vesc_driver_node.launch
```

3th Terminal

```
Roslaunch rplidar_ros rplidar_a3.launch
```

4th Terminal

```
roslaunch RL_policy_package ert_main
```

uses Agent A RL Policy

uses Agent S RL Policy



I strongly advise all readers to refer to the official VESC configuration manual for VESC configuration.

The settings that I used to setup the motor with a **Traxxass 25C 11.1V 5000mAh 3cell Lipo Velineon 3500 brushless (3351R)** Are the following :

Usage : **Generic**
Motor : **Small Outrunner**

Battery :

MOTOR

BATTERY

S

Battery Type

BATTERY_TYPE_LIION_3_0_4

CURRENT

DEFAULT

HELP

Battery Cells Series

3

CURRENT

DEFAULT

HELP

Battery Capacity

5.000 Ah

CURRENT

DEFAULT

HELP

☒ Advanced (0 = defaults)

Battery Current Regen: -5.0 A

Battery Current Max: 80.0 A

PREVIOUS

NEXT

Figure 10 VESC Battery Settings

Setup :

BATTERY

SETUP

DIR

Gear Ratio

☒ Direct Drive

Motor Pulley: 13

Wheel Pulley: 36

Wheel Diameter

66.00 mm

CURRENT

DEFAULT

HELP

↓ Only change if needed ↓

Motor Poles

4

CURRENT

DEFAULT

HELP

PREVIOUS

RUN DETECTION

Figure 11 VESC Setup Settings

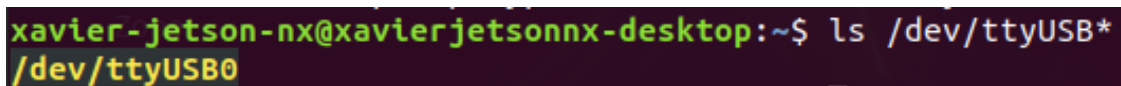
4. Debugging hardware problems

If you encounter any difficulties while trying to connect the Lidar or the VESC controller to the NVIDIA Jetson. You can try this debugging method.

Initially, connect only one device at a time, either the VESC or the Lidar. Then execute this command

```
ls /dev/ttyUSB*
```

This command will return the ID of the USB device like in Figure 12.



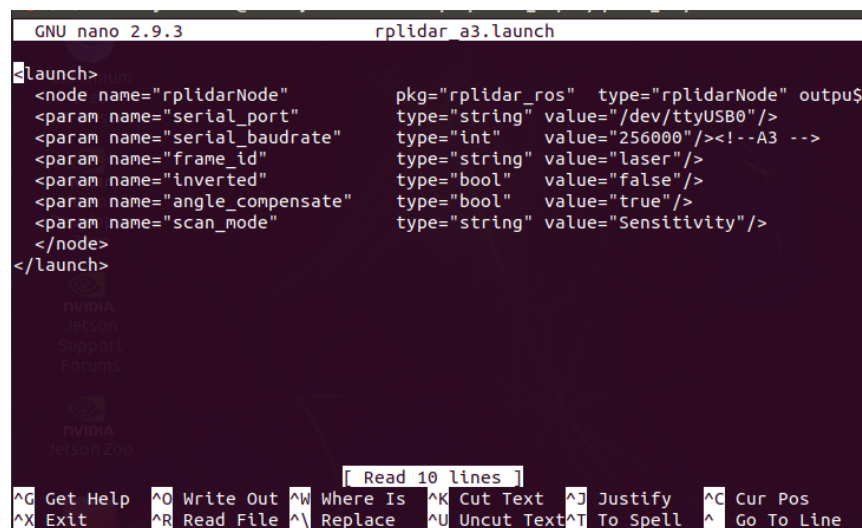
```
xavier-jetson-nx@xavierjetsonnx-desktop:~$ ls /dev/ttyUSB*  
/dev/ttyUSB0
```

Figure 12 `ls /dev/ttyUSB*` output

Next, navigate to the device launch file and verify that the USB ID specified inside the launch file matches the output obtained from the `ls /dev/ttyUSB*` command.

1. Lidar

```
cd ~/catkin_ws/src/rplidar_ros/launch  
nano rplidar_a3.launch
```



```
GNU nano 2.9.3 rplidar_a3.launch  
[Launch]  
<node name="rplidarNode" pkg="rplidar_ros" type="rplidarNode" output$  
<param name="serial_port" type="string" value="/dev/ttyUSB0"/>  
<param name="serial_baudrate" type="int" value="256000"/><!--A3 -->  
<param name="frame_id" type="string" value="laser"/>  
<param name="inverted" type="bool" value="false"/>  
<param name="angle_compensate" type="bool" value="true"/>  
<param name="scan_mode" type="string" value="Sensitivity"/>  
</node>  
</launch>  
[ Read 10 lines ]  
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos  
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

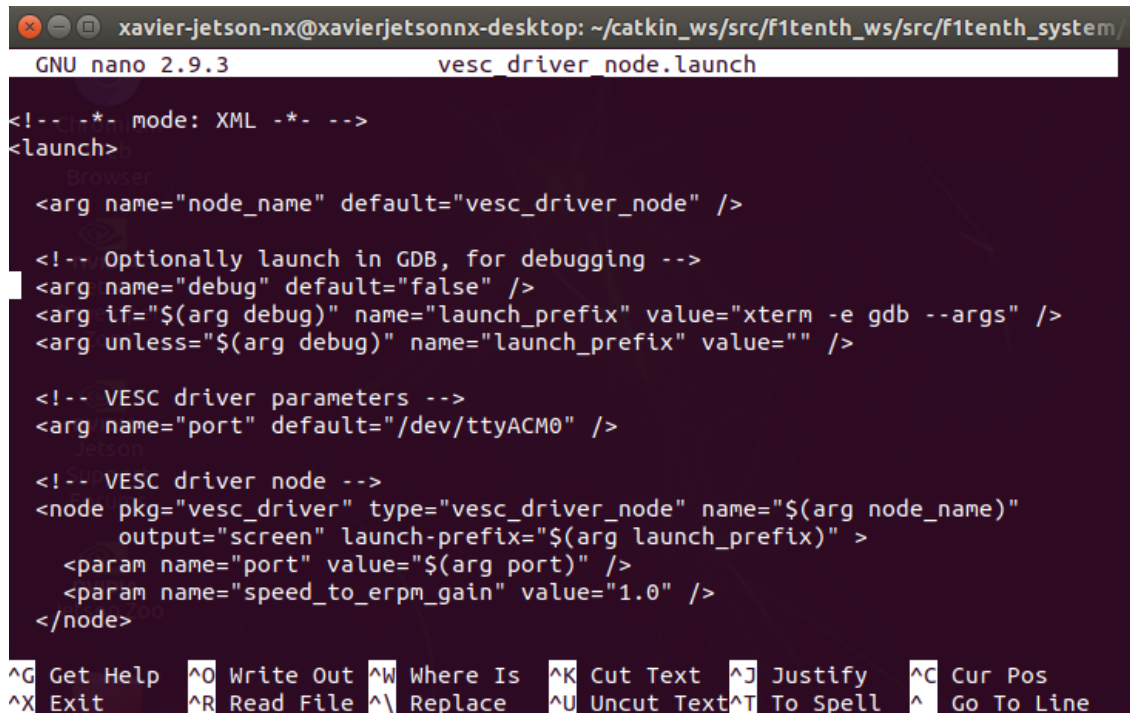
Figure 13 Contents of `rplidar_a3.launch` file

The VESC controller utilizes ttyACM instead of ttyUSB. However, the debugging method remains the same. Ensure that the output of the `ls /dev/ttyACM*` command matches the ttyACM specified in the `vesc_driver_node.launch` file.

```
ls /dev/ttyACM*
```

2. VESC

```
cd ~/catkin_ws/src/fltenth_ws/src/fltenth_system/vesc/vesc_driver/launch
nano vesc_driver_node.launch
```



```
xavier-jetson-nx@xavierjetsonnx-desktop: ~/catkin_ws/src/fltenth_ws/src/fltenth_system/
GNU nano 2.9.3 vesc_driver_node.launch

<!-- -*- mode: XML -*- -->
<launch>

  <arg name="node_name" default="vesc_driver_node" />

  <!-- Optionally launch in GDB, for debugging -->
  <arg name="debug" default="false" />
  <arg if="$ (arg debug)" name="launch_prefix" value="xterm -e gdb --args" />
  <arg unless="$ (arg debug)" name="launch_prefix" value="" />

  <!-- VESC driver parameters -->
  <arg name="port" default="/dev/ttyACM0" />

  <!-- VESC driver node -->
  <node pkg="vesc_driver" type="vesc_driver_node" name="$ (arg node_name)"
    output="screen" launch-prefix="$ (arg launch_prefix)" >
    <param name="port" value="$ (arg port)" />
    <param name="speed_to_erpm_gain" value="1.0" />
  </node>

^G Get Help  ^O Write Out ^W Where Is  ^K Cut Text  ^J Justify  ^C Cur Pos
^X Exit      ^R Read File ^\ Replace  ^U Uncut Text ^T To Spell  ^_ Go To Line
```