WAYNE STATE UNIVERSITY

Instruction Documentation

---

e.DO Robot Operating ROS2 Wrapper

Comau

NVIDIA

1 Dec, 2020

Table of Contents

Introduction

This document is intended to provide instructions for finding the relevant source code for the e.DO ROS2 wrapper class packages and installing the required programs, packages and dependencies to run e.DO_manual_ctrl (referred to as the wrapper class). The following package source code location information will be described in the document: ROS1; ROS2; the ROS bridge; e.DO_Manual_Ctrl (The Wrapper Class); and dependencies.

The document will also provide build instructions for the various packages found above including what must be sourced in each terminal to properly build and run the packages and instructions for running the program with robot signals virtualized, and instructions for running the program with robot integration and an explanation of the functions and how to use them.

Packages, Programs & Source Code

Package	Location
ROS	Debian Command: sudo apt install ros-melodic-desktop-full
ROS2	Debian Command: sudo apt install ros-eloquent-desktop
NCURSES	Debian Command: sudo apt-get install libncurses5-dev libncursesw5-dev
Building the bridge	https://github.com/ros2/ros1_bridge
Building eDO-ROS2	https://www.youtube.com/watch?v=9bkkFNz5Bs8&t=201s

1.1 Package Setup and Build Instructions

1.1.1 Installation of ROS1 Melodic

This tutorial assumes that you are installing ROS1 Melodic Distribution on Ubuntu 18.04

1. Go to the ROS Wiki Installation page
2. Click on the "Distributions" link
3. Find "ROS Melodic Morenia" and click on the link
4. Click on the "Installation Instructions"
5. Click on Ubuntu icon
6. Open up a new terminal in Ubuntu and go through the installation instructions carefully
7. Configure your Ubuntu repositories to allow "restricted", "universe", and "multiverse".
    a. Setup your sources.list
    sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
    b. Setup your keys
    sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
    c. Install the full desktop build with the following command
    sudo apt install ros-melodic-desktop-full
    d. Review available packages that were installed
    apt search ros-melodic
8. Setup your environment
    a. To automatically add your ROS environment variables to every bash session when a new terminal is launched
    echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
    source source ~/.bashrc
    b. To manually source the distribution, use the following command
    source /opt/ros/melodic/setup.bash

9. Install dependencies
    a. To install "rosinstall" , "rosdep" and other command line tools you will need for this project, run the following

sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool build-essential

1.1.2 Install ROS2 Eloquent

This tutorial assumes that you are installing ROS2 Eloquent Elusor Distribution on Ubuntu 18.04

1. Go to the ROS Wiki Installation page
2. Scroll down to "Binary Packages" and click on "Debian packages"
3. Scroll to "Setup Sources" to add the ROS2 apt repositories to your system
    a. Run *sudo apt update && sudo apt install curl gnupg2 lsb-release*
    b. Run *curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -*
    c. Run *sudo sh -c 'echo "deb [arch=$(dpkg --print-architecture)] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" > /etc/apt/sources.list.d/ros2-latest.list'*
4. Install the ROS2 Packages
    a. Update your apt repository caches after setting up the repositories
    sudo apt update
    b. Install desktop version which includes ROS, RViz, demo, and tutorials
    sudo apt install ros-eloquent-desktop
    c. Install ROS-Base which is a package of communication libraries, message packages, and command line tools
    sudo apt install ros-eloquent-ros-base
5. 5. Environment setup
    a. To source the ROS2 Eloquent distribution
    source /opt/ros/eloquent/setup.bash

1.1.3 Cloning comau-na/edo-ROS2 Git Repository

1. Navigate to the desired directory to clone the GitHub repository
2. Run the following command in your to clone the edo ROS2 repository
    git clone https://github.com/comau-na/edo-ROS2.git

3. Ensure the the directory was cloned by running using `ls`
4. In the directory you should have the following folders
    - joint_state_pub

- ros1_bridge
- ROS
- ROS2

1.1.4 Setting up a ROS1 Build Workspace

For this project there are two primary packages that you will use ROS1 for:

- **eDO_core_msgs** - These are the messages used by e.DO to publish key messages on the ROS network. In this project these need to be built on both the ROS1 and the ROS2 side as they will be bridged between the edo Core Package (e.DO source code) and the ROS2 wrapper class
- **joint_state_pub** - This will only be required when running a terminal simulation without usage of the e.DO hardware. This is needed to publish JointState and JointStateArray messages which are required for the wrapper class to function properly in a simulated environment

1. To create a catkin workspace run the following commands

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

This will create a CMakeLists.txt file in your 'src' folder. In addition, you should have a 'build' and 'devel' folder. In the 'devel' folder there will be several setup.*sh files. You will need to sources these files to overlay this workspace on top of your environment.

2. Whenever you build a package in ROS1, you will need to run the following source command:

```
source devel/setup.bash
```

1.1.5 Setting up a ROS2 Build Workspace

For this project there are two primary packages that you will use ROS1 for:

- edo_core_msgs - Same message package as ROS1, but with the syntax changed for ROS2

- edo_manual_ctrl- e.DO wrapper class used to control the e.DO robot arm using ROS2

1. To create a colcon (ROS2) workspace run the following commands

```
mkdir -p ~/<name of colcon_ws>/
cd ~/<name of colcon_ws>/
colcon build
```

This will create a 'build', 'install', and 'log' folder in your colcon workspace

To build an individual package in the workspace, you can run the following command:

```
colcon build --packages-select <name of package>
```

2. Whenever you build a package in ROS2, you will need to run the following source command inside your colcon workspace:

```
. install/setup.bash
```

Now that your workspace has been added to your path, you will be able to use your new package's executables

1.1.6 Installing the 'ncurses' Library

Note - ncurses is a software API for controlling writing to the console screen under Unix, Linux and other operating systems. You can create text-based user interfaces on a Linux or Unix-like system using ncurses library. This library is used in our e.DO Manual Control Wrapper Class

To install, follow the simple directions below:

1. Open a new terminal
2. Type the following:

```
sudo apt-get install libncurses5-dev libncursesw5-dev
```

The link below has more detail on what ncurses can do:
How To Install ncurses Library on a Linux

1.1.7 Building the Bridge

1. Open up a third terminal
2. Navigate to your bridge workspace
3. Create a 'src' folder inside the bridge workspace
4. Copy the ros1_bridge folder from the edo-ROS GitHub root folder to the 'src' folder
5. Navigate back up to the bridge workspace
6. Source ROS1 and ROS2

```
. /opt/ros/melodic/setup.bash
. /opt/ros/eloquent/setup.bash
```

7. Source the messages packages that were compiled above for both ROS1 and ROS2
Example from my directory, but this will be wherever you build your messages package:

```
. ~/ros1_bridge_sandbox/ros1_msgs_ws/install_isolated/setup.bash
. ~/ros1_bridge_sandbox/ros2_msgs_ws/install/local_setup.bash
```

8. Check for any dependencies by running the following command

```
rosdep install --from-paths ~/ros1_bridge_sandbox/ros1_msgs_ws --ignore-src
```

Note: If you get an error saying rosdep keys cannot be resolved, use the following command:

```
rosdep install --from-paths src --ignore-src --skip-keys python-wxtools --rosdistro eloquent -y
```

9. Compile the ros1_bridge

```
colcon build --packages-select ros1_bridge --cmake-force-configure
```

Note: You may get a warning after the build that "1 of the mappings can not be generated due to missing dependencies- edo_core_msgs/JointFwVersionArray <-> edo_core_msgs/JointFwVersionArray: - edo_core_msgs/JointFwVersion". Currently, the wrapper class does not use this message, so there is no need to map it to ROS2

10. Source the newly created local_setup file in the install folder

. install/local_setup.bash

11. Test the mapping by printing out the paired messages

ros2 run ros1_bridge dynamic_bridge --print-pairs

Note: This should list out all the edo_core_msg mappings including the custom mapping rules. Please check to make sure all mappings are included before proceeding

1.1.8 Building edo_manual_ctrl

1. Open up a new terminal and source your ROS2 distro

. /opt/ros/eloquent/setup.bash

2. Navigate to your ROS2 workspace

3. Build the package

colcon build --packages-select edo_manual_ctrl

Note: you may get some warnings, but they build will still be successful and will not affect functionality

1.2 Mapping Rules for Custom Messages

1.2.1 How to create map custom messages manually using yaml:

When to make a custom mapping rule:

Most of the time, messages are automatically mapped by the dynamic bridge. If messages do not share the same name or have different field names, you must create a yaml file to tell the bridge to pair the messages and fields. It is sometimes necessary to have different names for the ROS and ROS2 versions of a message because ROS allows capital letters in message names while ROS2 does not.

Format for mapping rules:

```
-
  ros1_package_name: 'edo_core_msgs'
  ros1_message_name: 'JointState'
  ros2_package_name: 'edo_core_msgs'
  ros2_message_name: 'JointState'
  fields_1_to_2:
    position: 'position'
    velocity: 'velocity'
    current: 'current'
    commandFlag: 'command_flag'
    R_jnt: 'r_jnt'

-
  ros1_package_name: 'edo_core_msgs'
  ros1_message_name: 'JointFwVersion'
  ros2_package_name: 'edo_core_msgs'
  ros2_message_name: 'JointFwVersion'
  fields_1_to_2:
    id: 'id'
    majorRev: 'major_rev'
    minorRev: 'minor_rev'
    revision: 'revision'
     svn: 'svn'
```

TIPS:

- You have to start the file with a '-'.
- If you would like to map multiple messages in a file they must be separated with a '-'.
- If the names of the fields are the same, you do not have to include anything after "ros2_message_name".
- The mapping rule file must be of type .yaml.

Where to put the mapping rule:

-Your file structure should look like this:

.

```
├─ ros1_msgs_ws
│  └─ src
│     └─ bridge_msgs
│        └─ msg
│           └─ JointCommand.msg
├─ ros2_msgs_ws
│  └─ src
│     └─ bridge_msgs
│        ├─ msg
│        │  └─ JointCommand.msg
│        └─ # YAML file of your custom interfaces that have non-matching
                  names
└─ bridge_ws
   └─ src
      └─ ros1_bridge
         └─ CMakeLists.txt
         # this is the cmakelist.txt that that you have to edit
         └─ package.xml
         # this is the package.xml file that you have to edit
```

Now that you have a mapping rule, rebuild your ROS2 message package using this command: colcon build --packages-select YOUR PACKAGE NAME

Add these lines to the Package.xml file of the ros1_bridge package:

```
<export>
    <build_type>ament_cmake</build_type>
    <ros1_bridge mapping_rules="mapping_rule.yaml"/>
</export>
```

NOTE: there should already be a section in the package.xml that looks like this, add the extra line to the existing export tag:

```
<export>
    <build_type>ament_cmake</build_type>
</export>
```

Add these lines to the cmakelists.txt file of the ros1_bridge:

```
install(
    FILES mapping_rule.yaml
    DESTINATION share/${PROJECT_NAME})
```

Final step, rebuilding the bridge:

-The general structure of how to build the bridge with custom messages is:

1. Source ROS1
2. Source ROS2
3. Source ROS1 message package
4. Source ROS2 message package
5. Build the bridge

-The specific commands for rebuilding will look similar to this:

```
. /opt/ros/melodic/setup.bash
. /opt/ros/eloquent/setup.bash
. <workspace-parent-path>/ros1_msgs_ws/install_isolated/setup.bash
. <workspace-parent-path>/ros2_msgs_ws/install/local_setup.bash
cd <workspace-parent-path>/bridge_ws
colcon build --packages-select ros1_bridge --cmake-force-configure
```

-Now, run this command to ensure that the message was paired: ros2 run ros1_bridge dynamic_bridge --print-pairs

- your message should now show up on this list

2.1 Running the Program (Virtualized)

2.1.1 Robot Signal Proxying

The robot will likely not be available as frequently as you would like while you are working on your Capstone Project, to get around this logistical problem the previous team developed this method for testing your implementation when the robot is not available to you. By proxying signals expected back from the robot via ROS publishers created as needed, you can publish expected robot signal data and test your implementation's logical functionality.

2.1.2 Sourcing Terminals & Scripts

Every package you build or run will need to source it's dependencies.

Sourcing	
ROS	source /opt/ros/melodic/setup.bash
ROS2	source /opt/ros/eloquent/setup.bash
Bridge	source Location_of_bridge_package.../install/setup.bash
eDO Core Messages (ROS)	source Location_of_package.../devel/setup.bash
eDO Core Messages (ROS2)	source Location_of_package.../install/setup.bash

The previous team developed the following scripts to make sourcing terminals and running the bridge easier for you:

- ./start_bridge.sh
- ./run_manual_control.sh
- ./testing.sh

2.1.3 roscore

When the robot is not available to you, you must run roscore to initialize the core that ROS will use to run topics on and carry messages over. This step is not necessary when working with the robot as eDO Core will have a running roscore active.

1. Open a new terminal
2. Command: source /opt/ros/melodic/setup.bash
3. Command: roscore

Alternatively, you can run the shell script named start ros bridge. Command: ./start_bridge.sh

2.1.4 ROS bridge

The main purpose of this wrapper class is to work as a layer to take ROS2 commands then issue them to the eDO robot (the package eDO runs on is called eDO Core Package and is written in ROS). Until the Comau developers based in Italy update the eDO core package to run ROS2 natively, the ROS bridge must be used to run the wrapper class.

1. Open a new terminal
2. Source the bridge. Command: source Location of bridge package.../install/setup.bash
3. Run the Bridge. Command: ros2 run ros1_bridge dynamic_bridge --bridge-all-topics

Alternatively, you can run the shell script named start ros bridge. Command: ./start_bridge.sh

2.1.5 Launching the Program

Once the roscore and bridge are running you are ready to run the eDO Manual Control program.

1. Open a new terminal.
2. Source ROS2. Command: source /opt/ros/eloquent/setup.bash
3. Source the bridge. Command source Location of bridge package.../install/setup.bash
4. Start the program. Command: ros2 run edo_manual_ctrl sample_cli

Alternatively, you can run the shell script named run manual control. Command: ./run_manual_ctrl.sh

2.1.6 Program Functionality

Once sample_cli is running you will be greeted with the initialization screen. When the program launches it will be waiting for a machine state signal back from the robot.

1. Open a new terminal.
2. Command: source /opt/ros/melodic/setup.bash
3. Command: . devel_isolated/setup.bash
4. Command: rostopic pub -r 10 /machine_state edo_core_msgs/MachineState '{current_state: 1, opcode: 4}'

Alternatively, you can run the shell script named testing.sh. Command: ./testing.sh

The following chart shows functionality of the program options:

Calibrate			
Calls Jog			
Jog			
Jog			
Move			
Joint to Joint	Joint to Cartesian	Cartesian to Joint	Cartesian to Cartesian
Recalibrate			
Calls Jog			
Data Display			
Data Display			

2.1.6.1 Calibrate

After the initialization is complete the robot will enter calibration mode. Follow the commands to enter a calibration jog. The purpose of this is to line up the joints of the robot to the zero position so the robot knows where it is in real space. For virtualization you can skip aligning the joints and press escape to exit the jog menu and press y to send calibration to the robot. The program will be waiting for a subscriber to continue so follow the commands below:

1. Open Terminal
2. Command: Source /opt/ros/eloquent/setup.bash
3. Command: ros2 topic echo /bridge_init

2.1.6.2 Jog

The Jog command moves the robot in real time sort of like a RC Car. As long as you hold the button down the robot will move that joint in the direction you decide. When you select this option a key map will be provided for you telling you which button controls which joint (+/-).

You can also change the velocity the joints move at, also provided on the key map. Since this is a virtual version you can follow the following commands to see feedback of the messages being sent to eDO from the wrapper class:

1. Open a new terminal
2. Command: source /opt/ros/eloquent/setup.bash
3. Command: ros2 topic echo /bridge_init

2.1.6.3 Move

The Move command schedules robot pose destinations you want the robot to move to. So far the original inherited wrapper (ROS 1) and tablet only had functional testing for Joint-to-Joint moves so that is all that is working currently. Dr. Rushaidat should have more information for you regarding this as he is in communication with Italy trying to determine the status of these commands with the eDO Core development team. Select Joint to Joint from the move menu.

Select the number of Commands you want to enter, the amount of time the robot should wait before executing the command and how many times you want it to repeat (loops).

For virtualized testing to show the message being published to the topic use the following commands:

1. Open a new terminal
2. Command: source /opt/ros/eloquent/setup.bash
3. Command: ros2 topic echo /bridge_move

The program will be waiting for acknowledgement from the robot that the move was received and executed. To clear these queues to return to the main menu use the following commands:

1. Open a new terminal

2. source /opt/ros/eloquent/setup.bash
3. Command: ros2 topic pub /machine_movement_ack edo_core_msgs/msg/MovementFeedback '{type: 2, data: 0}'
4. Command: ros2 topic pub /machine_movement_ack edo_core_msgs/msg/MovementFeedback '{type: 0, data: 0}'
5. Command: ros2 topic pub /machine_movement_ack edo_core_msgs/msg/MovementFeedback '{type: 2, data: 0}'
6. Command: ros2 topic pub /machine_movement_ack edo_core_msgs/msg/MovementFeedback '{type: 0, data: 0}'

2.1.6.4 Recalibrate

The purpose of this function is to line up the joints of the robot to the zero position so the robot knows where it is in real space. For virtualization you can skip aligning the joints and press escape to exit the jog menu and press y to send calibration to the robot. The program will be waiting for a subscriber to continue so follow the commands below:

1. Open a new Terminal
2. Command: Source /opt/ros/eloquent/setup.bash
3. Command: ros2 topic echo /bridge_init

2.1.6.5 Data Display

Data display returns crucial information from the robot to the user. When you select this option the Machine State, Cartesian Pose and Joint state information is returned to the user. For virtual testing you need to follow these commands to proxy robot signals and see the display:

1. Open a new Terminal
2. Command: Source /opt/ros/melodic/setup.bash
3. Command: rostopic pub -r 10 /machine_state edo_core_msgs/MachineState '{current_state: 1, opcode: 4}'
4. Open a new Terminal
5. Command: Source /opt/ros/melodic/setup.bash
6. Command: rostopic pub -r 10 /cartesian_pose edo_core_msgs/CartesianPose '{x: 1.00, y: 2.00, z: 1.00, a: 2.00, e: 1.00, r: 1.00}'
7. Open a new Terminal
8. Command: Source /opt/ros/melodic/setup.bash
9. Command: rosrun edo_core edo_jnt_handler

Alternatively, you can run the shell script named testing.sh. Command: ./testing.sh

2.2 Running the Program (Robot Integration)

2.2.1 Connecting to the Robot

Connect to edoWifi. The password for edo Wifi is “edoedoedo”. For terminals communicating with eDO (Bridge, sample_cli) use the following commands.

1. Open a new terminal
2. Command: export ROS_MASTER_URI=http://IP_ADDRESS_OF_RASPBERRY_PI_IN_THE_EDO:11311
3. Command: export ROS_IP=YOUR IP ADDRESS

2.2.2 Sourcing Terminals & Scripts

Every package you build or run will need to source it’s dependencies.

Sourcing	
ROS	source /opt/ros/melodic/setup.bash
ROS2	source /opt/ros/eloquent/setup.bash
Bridge	source Location_of_bridge_ package.../install/setup.bash
eDO Core Messages (ROS)	source Location_of_package.../devel/setup.bash
eDO Core Messages (ROS2)	source Location_of_package.../install/setup.bash

The previous team developed the following scripts to make sourcing terminals and running the bridge easier for you:

- ./start_bridge.sh
- ./run_manual_ctrl.sh

2.2.3 ROS bridge

The main purpose of this wrapper class is to work as a layer to take ROS2 commands then issue them to the eDO robot (the package eDO runs on is called eDO Core Package and is written in ROS). Until the Comau developers based in Italy update the eDO core package to run ROS2 natively, the ROS bridge must be used to run the wrapper class.

1. Open a new terminal
2. Source the bridge. Command: source Location of bridge package.../install/setup.bash
3. Run the Bridge. Command: ros2 run ros1_bridge dynamic_bridge --bridge-all-topics

Alternatively, you can run the shell script named start ros bridge. Command: ./start_bridge.sh

2.2.4 Launching the Program

Once you are connected to the eDO robot and the bridge is running you are ready to run the eDO Manual Control program.

5. Open a new terminal.
6. Source ROS2. Command: source /opt/ros/eloquent/setup.bash
7. Source the bridge. Command source Location of bridge package.../install/setup.bash
8. Start the program. Command: ros2 run edo_manual_ctrl sample_cli

Alternatively, you can run the shell script named run manual control. Command: ./run_manual_ctrl.sh

2.2.5 Program Functionality

Once sample_cli is running you will be greeted with the initialization screen.

The following chart shows functionality of the program options:

Calibrate			
Calls Jog			
Jog			
Jog			
Move			
Joint to Joint	Joint to Cartesian	Cartesian to Joint	Cartesian to Cartesian
Recalibrate			
Calls Jog			
Data Display			
Data Display			

2.2.5.1 Calibrate

After the initialization is complete the robot will enter calibration mode. Follow the commands to enter a calibration jog. The purpose of this is to line up the joints of the robot to the zero position so the robot knows where it is in real space.

2.2.5.2 Jog

The Jog command moves the robot in real time sort of like a RC Car. As long as you hold the button down the robot will move that joint in the direction you decide. When you select this option a key map will be provided for you telling you which button controls which joint (+/-). You can also change the velocity the joints move at, also provided on the key map.

2.2.5.3 Move

The Move command schedules robot pose destinations you want the robot to move to. So far the original inherited wrapper (ROS 1) and tablet only had functional testing for Joint-to-Joint moves so that is all that is working currently. Dr. Rushaidat should have more information for you regarding this as he is in communication with Italy trying to determine the status of these commands with the eDO Core development team. Select Joint to Joint from the move menu.

Select the number of Commands you want to enter, the amount of time the robot should wait before executing the command and how many times you want it to repeat (loops).

2.2.5.4 Recalibrate

The purpose of this is to line up the joints of the robot using the jog function to the zero position to ensure correct calibration of eDO so the robot knows where it is in real space.

2.2.5.5 Data Display

Data display returns crucial information from the robot to the user. When you select this option the Machine State, Cartesian Pose and Joint state information is returned to the user. For virtual testing you need to follow these commands to proxy robot signals and see the display:

3.1 Work Moving Forward

3.1.1 Subscriber for NVIDIA Package

Towards the end of the semester, NVIDIA released a machine learning package intended to be a stretch goal for the team working on this project that you have inherited. Due to the Covid-19 pandemic, the team before you lost approximately 35% of testing time with the robot and was unable to perform testing on this package.

A subscriber, vision_subscriber, has been created for integration with the NVIDIA package. The Subscriber needs to be fully implemented with the software and tested. The goal of the whole ROS2 migration of the wrapper class we completed was access to improved AI capability for NVIDIA.

To summarize the goal of the software briefly, The NVIDIA vision package can identify objects to be manipulated by eDO, determines object location (x and y for now, picking objects from a flat plane, or constant z depth from fixed camera location) and sends a move command to the robot arm to find the object and perform an operation such as a sort depending on what the identified object was determined to be. Your job is to take the subscriber we wrote (takes the message data passed from the NVIDIA publisher and passes to the move command) and test that the eDO robot arm can receive the data, and process the motion to pick it up.

This subscriber has not yet been tested in any sense with the robot integration so consider it more of an outline than the complete solution for subscription to the NVIDIA package.

4.1 Known Issues

4.1.1 Data Display over Bridge - Joint State Array

4.1.1.1 Background

When working with the e.DO robot hardware in the lab, it was noted that 5 messages were not being properly bridged. Therefore, this required us to use the custom mapping rules to correctly the bridge the messages. Even after this step was completed, it was observed that the Print e.DO data function (DataDisplay class) was getting caught in a loop waiting for the */machine_algo_jnt_state_bridge* to be available to subscribe to. This was odd behavior, because the dynamic bridge showed that it was being properly paired, but the parameters were not being passed when DataDisplay (Print e.DO data in the command line) was being called.

Another important note is that when Print e.DO data works fine in a simulated environment and behaves as expected. **This only occurs when connected to the e.DO hardware itself.**

4.1.1.2 Suggested Next Steps

Our troubleshooting led us down the following two paths:

- There was a package that we didn't include as a dependency when porting over from the original ROS1 wrapper class that may need to be included. Our original thought was that this wasn't needed, but would be a good next step to troubleshoot. The dependency is:

  message_runtime

  It would be a good idea to try this first before proceeding with troubleshooting

- The other troubleshooting step would be to visualize or record the */machine_algo_jnt_state_bridge* topic to understand its behavior when the

edo_manual_ctrl is operating. Our time in the lab was limited this semester, so we didn't get much of a chance to diagnose the issue on the hardware itself.

4.1.2 Move Command - Cartesian Moves

Our team completed migration of the old wrapper from ROS to ROS2 mirroring the commands from the older ROS package. This means functionality from the old wrapper and the included eDO tablet that were not working, are also not working in our implementation.

We are not sure exactly why the Cartesian XYZAER commands are not working for our implementation, the tablet, or the old ROS Wrapper; but we believe it has to do with either the frame of the command (Do these move commands use the World Frame or a specific eDO arm User Frame?), or the range/combination of the inputs for 3D space. Dr. Rushaidat has been in communication with the design team in Italy for advice on the frame, range of acceptable inputs and how to select a combination of cartesian inputs that the robot can understand.

The move command functionality is paramount for what NVIDIA wants to do with their image recognition package so we recommend a solution be found for this issue as soon as possible.

4.1.3 Firmware Upgrade

The team before you was in the process of upgrading the control tablet provided by Comau and the eDO itself to version 3. Covid lockdown restrictions came into place before the team was able to complete the update. Currently the tablet has been upgraded however the eDO is still running on the lower firmware revision. Dr Rushaidat may have already completed this before you begin your part on the project so he should be able to advise you on the state of the Firmware.

5.1 Glossary and Acronyms

The following table provides definitions for terms relevant to this document and/or helpful definitions to know as you educate yourself on ROS and ROS2:

Term	Definition
e.DO	Modular, multi-axis articulated robot developed by Comau
eDO_core	ROS pkg responsible for internal operations of e.DO
eDO_core_msgs	ROS pkg responsible for messaging between ROS nodes
eDO_manual_ctrl	ROS wrapper package that allows control and communication between Ubuntu and ROS
ROS	Open-source robot operating system
ROS2	Version 2 of the open-source robot operating system
ROS Migration	The process the team will take to migrate codebase from ROS to ROS2
Node	Communication point responsible for modular purpose
Topic	A communication bus for nodes to exchange messages
Messages	Data passed between nodes over a topic
Publisher	Node that sends messages over a topic
Subscriber	Node that subscribes to and receives messages over a topic
Pub/Sub	Name for the publisher-subscriber topic communication model
Services	Alternate communication method based on call-and-response model instead of pub/sub, data only provided when called by client node.
Workspace (ws)	Area reserved for ROS packages within the file structure of host OS (Ubuntu)
Wrapper Class	ROS package developed to allow control of the e.DO robot from Ubuntu Command Line Interface
ros1_bridge	ROS Package used to bridge communication between ROS and ROS2
NVIDIA Jetson TX2	NVIDIA Embedded Autonomous/AI computing device

NVIDIA Jetson Xavier	NVIDIA Embedded Autonomous/AI computing device
Raspberry Pi	Small, single-board computer used as axis motor controller in the e.DO robot
Stakeholder	Any person interacting with e.DO who is not the user
User	Someone who interacts with e.DO
Client	Comau and NVIDIA
Command Line Interface (CLI)	Text-based user interface to view and manage computer files, and operation.
OS	Operating System
Host OS	Ubuntu
Ubuntu	Linux Distribution
Package.xml	Provides meta information about the package and required for building a package
CMake.txt	Standard file used to describe how to build the code and where to install it. This needs to be updated before building a package.

6.1 Getting Started Resource List

This section provides a list of links helpful for learning ROS, ROS2, Dependencies and Libraries used in this project:

Resources	
https://index.ros.org/doc/ros2/	ROS2
https://wiki.ros.org	ROS
https://github.com/jshelata/eDO_manual_ctrl	ROS e.DO wrapper
https://github.com/ros2/ros1_bridge	Bridge for ROS -ROS2
https://www.allisonthackston.com/articles/bridging_ros_ros2.html	ROS- ROS2 Bridging

https://github.com/mabelzhang/ros1_bridge_sandbox	ROS-ROS2 Bridging
https://github.com/ros2/ros1_bridge/blob/master/doc/index.rst	Message mapping ROS-ROS2
https://developer.nvidia.com/blog/implementing-robotics-applications-with-ros-2-and-ai-on-jetson-platform-2/	NVIDIA Jetson with ROS2
http://design.ros2.org/articles/changes.html	Differences between ROS1 and ROS2
https://index.ros.org/doc/ros2/Contributing/Migration-Guide/	Migration Guide for

6.1.1 Learning ROS & ROS2

A good starting point for your project is complete the ROS tutorials featured on ROS.org and the wiki page of ROS.org. It will be difficult to understand the code you are receiving without a good understanding of the fundamentals of ROS and ROS2.

Jack Shelata's GitHub page for the original ROS wrapper class is also a good resource for seeing where the previous team started as well as a practical example of ROS programming.

After you understand the fundamentals of the Pub/Sub communication protocol and have learned how messages are passed over topics, It is especially important to pay attention to the CMakeLists and Package.XML tutorials to learn what is needed to instruct the build tools in how to build your new packages. Referencing your class notes from CSC 4420 (Operating Systems) may also be helpful in understanding how to build packages in Linux.