Bioscara

Sebastian Storz

Mar 04, 2026

CONTENTS:

This is the documentation for Bioscara DIY SCARA robot developed at the **DTU Arena for Life Science Automation (DALSA)**. See the table of contents below to see the available user guides.

- The *Getting Started* guides help you to install dependencies, build the ROS2 workspace, setup the network and finally operate the robot.
- The *Developer Guides* guides are targeted towards developers and contain other useful instructions, for example how to build this documentation or flash the joint firmware.
- The *Hardware* guides describe the hardware assembly instructions. **Note:** These require a consolidation!
- The *Source Code Documentation* links to the C++ API documentation of the ROS2 packages.

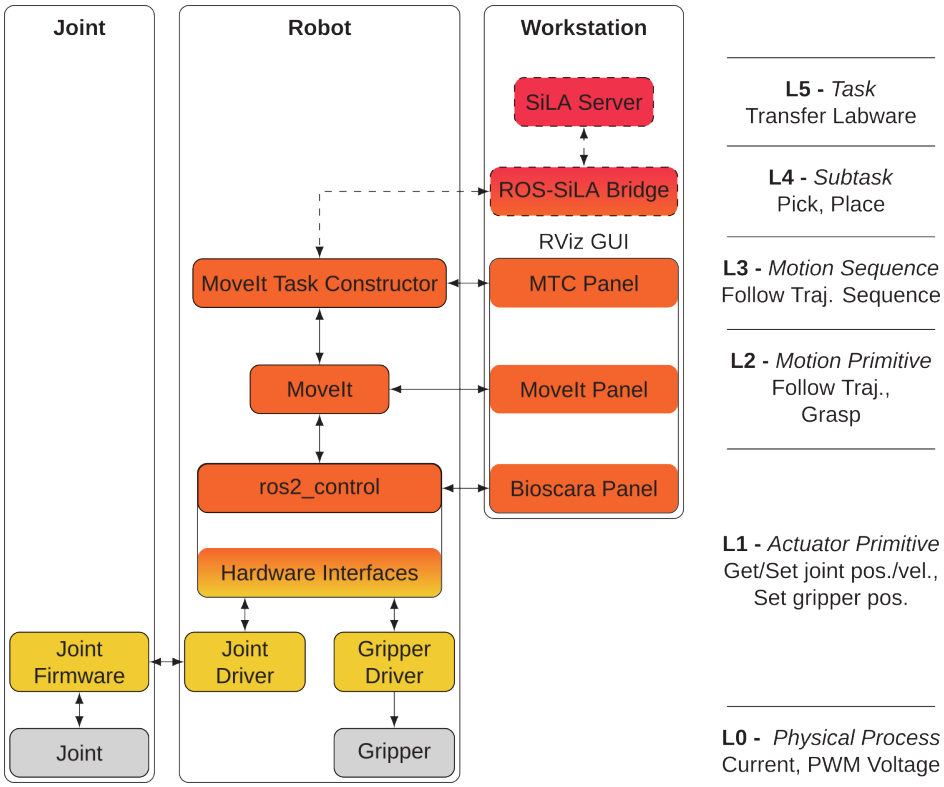
The PDF version of this documentation can be found here: `bioscara.pdf`

CHAPTER

ONE

ARCHITECHTURE

The control system architecture is schematically displayed in the figure below. The yellow colors whos custom C++ hardware specific code which is described in the *C++ API Documentation*. The orange colors are the ROS2 applications for motion control and trajectory generation. The red colors are SiLA applications, not yet implemented as indicated by the dashed lines.



1.1 Getting Started

1.1.1 Installation

This guide describes how to to install and setup all devices to operate the robot. At the first two chapters contain device specific details while the latter can be applied to both the robot controller and a control PC.

Important

This installation guide has not been tested in its entirety. It can thus happen that a prerequisite has been missed, use your own problem solving skills to fill the gap! If you find missing information, include it into this document!

Robot Controller (Raspberry Pi 4B 4GB) Specific

The following section describes the installation steps specific to the Raspberry Pi robot controller.

Creating the Boot Image

Using the latest version of the Raspberry Pi imager, flash an SD card (16GB, Class 10) with Ubuntu 24.04 LTS Server.

Note

Since the HDMI ports of the Raspberry Pi are no longer accessible when installed inside the robot, follow the next steps to configure remote access.

Flash the SD card with the boot image, you don't need to set custom settings, this will be done in the next step. **Do not boot the Raspberry Pi yet!**

Configuring the Network

When the flashing is finished, copy the *installation/network-config* file to the */boot/* partition of the SD card.

The *network-config* contains the initial netplan configuration. This file will only be read once per instance (during the first boot). The contents of *network-config* will be copied to */etc/netplan/50-cloud-init.yaml* when installed on the Raspberry Pi further information.

The netplan contains the following configutation:

- Sets the Netplan renderer (the backend for which Netplan is creating the configuration files) to NetworkManager.
- WiFi:
  - Saves an AP with the SSID: "DALSA_IOT" and pre-shared key: "dalsa_iot". See *networking guide* for a description how to use this fallback connection.
- Ethernet:
  - Configures static routing for the "eth0" interface for a LAN. See the network architecture *here*.
  - The static IP of the robot (the Raspberry Pi) is set to `10.10.10.2`.
  - The ethernet route gets assigned a higher metric to route internet traffic through the WiFi interface if it is connected, since the LAN has not internet access.

Troubleshooting

For unknown reasons, while testing theses instructions, it has happened that the Pi was not connecting to the AP nor ethernet. Logging into the Pi with a screen attached showed that the NetworkManager was not installed on the system, this should not happen on a clean installation. If that should happed try one of the following:

- Create a new boot SD card, but edit *network-config* to use 'networkd' (the systemd-networkd network service) instead of 'NetworkManager' as the Netplan renderer. Then, when booted up and connected to a network, install the NetworkManager (`sudo apt install network-manager`). Edit */etc/netplan/50-cloud-init.yaml* back to

'NetworkManager' as renderer and apply `sudo netplan apply`. The network should now be up and managed by the NetworkManager, not systemd-networkd.

- Instead of creating a new boot image, connect to the Pi's shell by attaching display and keyboard and change the renderer to 'networkd', apply `sudo netplan apply`, wait for connection `sudo netplan status -a`, then continue as described above with downloading the NetworkManager.

Configuring the User

Copy the *installation/user-data* file to the */boot/* partition of the SD card as well.

The *user-data* file contains mostly configuration for the user *scara* with the password 'dtubio'. The ubuntu cloud-init program manages these configurations during instance deployment. Many settings can be set here, the official README is given for reference:

> *"The user-data of the cloud-init seed. This can be used to customize numerous aspects of the system upon first boot, from the default user, the default password, whether or not SSH permits password authentication, import of SSH keys, the keyboard layout, the system hostname, package installation, creation of arbitrary files, etc. Numerous examples are included (mostly commented) in the default user-data. The format of this file is YAML, and is documented at: https://cloudinit.readthedocs.io/en/latest/topics/modules.html and https://cloudinit.readthedocs.io/en/latest/topics/examples.html"*

You can insert the SD card into the Raspberry Pi and **boot now!**

Wait for the installation to finish, it will take some time.

Open Remote Terminal Session

After the installation of the OS has succeeded, we need to log in remotely. The remaining guide assumes you are logged in via ssh to the robot controller.

To do so, first establish a wired or wireless connection to the robot controller as described in the *network guide* and then log in remotely via ssh:

```
ssh scara@<ip-adress>
```

Connect to the Internet

The following script requires an internet connection. Establish an internet connection as described in the *networking guide*.

Raspi-Config Installation

Installing the Raspi-Config tool on the Ubuntu OS brings some addtional hardware configuation options.

Note

The exact mechanism and effects of the tool are not fully clear.

Execute the *installation/scripts/raspi-config_install.sh* script. The source code was provided here.

```
bash installation/scripts/raspi-config_install.sh
```

To apply the changes, restart the robot.

PWM Activation

In order to use the PWM for servo control, it must be enabled in the */boot/firmware/config.txt* as a "dtoverlay=pwm". However there were issues that the */sys/class/pwm/pwmchip0* could only be used as root. For this reason the *installation/scripts/pwm_activation.sh* registers a udev rule that automatically changes the owner to the "plugdev" group, of which the scara user is a part of. This way the PWM can be controlled without root rights.

```
bash installation/scripts/pwm_activation.sh
```

Ensure Low-Latency

To ensure a low-latency operation some settings must be applied. Both of the following settings were already set correctly after the OS installation on the Raspberry Pi. Potentially from the Raspi-config installation(?). Follow the following instructions to ensure they are indeed enabled.

Note

This set of kernel configuration is also set when installing the "Low Latency Ubuntu" (`sudo apt install linux-lowlatency`). But this has not been tested.

Preemption Policy

In mainline Ubuntu the maximum preemption policy is PREEMPT. Ensure the CONFIG_PREEMPT is set:

```
cat /boot/<BUILD?> | grep CONFIG_PREEMPT
```

the last known <BUILD> was `config-6.8.0-1028-raspi`.

Find more information on the preemption policy here.

Interrupt Timer Resolution

Ensure that a 1000 Hz interrupt timer resolution is set:

```
cat /boot/<BUILD?> | grep CONFIG_HZ
```

the last known <BUILD> was `config-6.8.0-1028-raspi`.

> *"The timer interrupt handler interrupts the kernel at a rate set by the HZ constant. The frequency affects the timer resolutions as a 100 Hz value for the timer granularity will yield a max resolution of 10ms (1 Hz equating to 1000ms), 250Hz will result in 4ms, and 1000Hz in the best-case resolution of 1ms."* source

Adding the *realtime* Group

Add the user to the realtime group and give it the rights to set higher scheduler priotities as described here.

For real-time tasks, a priority range of 0 to 99 is expected, with higher numbers indicating higher priority. By default, users do not have permission to set such high priorities. To give the user such permissions, add a group named realtime and add the user controlling your robot to this group:

```
sudo addgroup realtime
sudo usermod -a -G realtime $(whoami)
```

Afterwards, add the following limits to the realtime group in /etc/security/limits.conf:

```
@realtime soft rtprio 99
@realtime soft priority 99
@realtime soft memlock unlimited
@realtime hard rtprio 99
@realtime hard priority 99
@realtime hard memlock unlimited
```

The limits will be applied after you log out and in again.

This **concludes** the robot control specific setup.

Control PC Specific

The control PC is a desktop PC running the Ubuntu 24.04 LTS Desktop operating system. Its purpose is to be the primary control interface for the user. It is in the same network as the robot controller and thus the ROS2 nodes on the robot controller and control PC can communicate with each other. Recommended use case (at the current development state):

- Run the RViz GUI with the Bioscara Panel, MoveIt and MTC Panel to control the hardware state, manual trajectory generation and sequence inspection
- Since the MTC is not realtime-critical, it can also run on the control PC

OS Installation

Install the latest Ubuntu 24.04 LTS Desktop release according to the offical guide.

Create the following user:

- **User:** *scara-dev*
- **Password:** *dtubio*

All following instructions assume you are logged in to the device.

Network Configuration

First assign a static IP by creating the */etc/netplan/99_config.yaml* file with following content:

```
network:
  version: 2
  renderer: networkd
  ethernets:
    eno1:
      addresses:
        - 10.10.10.3/24
      routes:
        - to: default
          via: 10.10.10.1
          metric: 700 # Increase the metric so that a the wifi connection (metric: 600)␣
↪is preffered for internet traffic
      nameservers:
          addresses:
            - 8.8.8.8
            - 8.8.4.4
```

And the following a simple WPA wifi access point:

```
...
  wifis:
    wlp0s20f3:
      dhcp4: true
      optional: true
      access-points:
        "DALSA_IOT":
          password: "dalsa_iot"
...
```

Then add a static hostname entry by adding the following line in the */etc/hosts* file:

```
10.10.10.3 scara-dev
```

Reboot the machine to apply all changes. Changes to the netplan manager can be applied like this:

```
sudo netplan apply
```

Connect to the Internet

Establish an internet connection as described in the *networking guide*.

Install I2C dependecies

In order to later compile the ROS2 workspace the only external dependency *lgpio* must be installed.

Execute the *installation/scripts/i2c_libraryAndTools_install.sh* script:

```
bash installation/scripts/i2c_libraryAndTools_install.sh
```

Install ROS2

Use the *installation/scripts/ROS2-Jazzy_install.sh* to install ROS2. The installation requires root priviliges, enter the password when prompted:

```
bash installation/scripts/ROS2-Jazzy_install.sh
```

Clone the Project Repository

Clone the project repository from Github into the *~/bioscara/* repository:

```
git clone https://github.com/DALSA-Lab/bioscara.git
```

Import further Dependecies

Next we are going to install a lot of dependencies, either as a binary or from source.

Dependecy Management Tools

The order of managing dependenices is according to this guideline the following:

1. **rosdep**: Installs missing binary dependecies specified in the packages via the systems package manager.
2. **vcstool**: Specify further source code repositoryies in a repository file, `vcstool` will then retrieve the repository and it can then be built with colcon.

3. **other**: Manually install dependecies. (Already finished after install of *lgpio*)

Clone further Source Code Repositories

This step pulls all dependecies that are not available as binaries.

Navigate to the ROS2 workspace *lib/ros2_ws*:

```
cd lib/ros2_ws
```

`vcstool` is found in many ROS2 packages to import dependencies that are not in a ROS or debian repository from a repository file. It should be automatically installed with the ROS2 install script. If not, its binary name is `python3-vcstool`.

Then invoke the `vcstool` to import the repositories specified in the *lib/ros2_ws/req.repos* file:

```
vcs import --recursive src < req.repos
```

This will pull the following repositories:

- ***single_trigger_controller*** branch: *main*
  - This repository hosted on the DALSA Github contains the SingleTriggerController used to trigger homing. Since the controller is generic, it is hosted as a seperate repository.
- ***moveit_task_constructor*** branch: *ros2*
  - The MTC is not available as a binary and must thus be built from source.

Important

The following repositories only need to included if the latest available binary MoveIt2 version is < 2.12.4! All newer version will include a bugfix that was not yet released at the time of writing the tutorial. **TO-DO:** Remove the following repositories from the *req.repos* file if the binary is available.

- ***moveit2*** branch: *main*
  - MoveIt2 is cloned from a DALSA Fork. Using a fork, we have full control over the versioning and it also includes the changes to the Pilz Motion Controller that allow it to be used with any planning pipeline.
- ***moveit_visual_tools*** branch: *ros2*
  - Only cloned from source due to installation conflicts with the binary
- ***pick_ik*** branch: *main*
  - Only cloned from source due to installation conflicts with the binary

Install even more Repositories

Important

Do this step ONLY if MoveIt is installed from source! This is NOT necessary if it can be installed as a binary.

Recursively import MoveIt's source code dependencies:

```
cd src
for repo in moveit2/moveit2.repos $(f="moveit2/moveit2_$ROS_DISTRO.repos"; test -r $f &&␣
↪echo $f); do vcs import < "$repo"; done
```

Binary Dependencies

This steps installs all packages that can be resolved through `rosdep` (all packages that have been released to the ROS2 package ecosystem and some debian packages):

```
cd lib/ros2_ws
sudo apt update
rosdep update
rosdep install -r --from-paths src --ignore-src --rosdistro $ROS_DISTRO -y
```

This command will recursively scan every package in the workspace for the <depend/> key and install missing packages.

As a last step, remove any conflicting MoveIt binaries. This should not be neccessary on a fresh install.

```
sudo apt remove ros-$ROS_DISTRO-moveit*
```

Build the Workspace!

Follow the *building instructions* to finally compile the entire workspace!

1.1.2 Building the ROS2 Workspace

After importing all dependencies the ROS2 workspace *lib/ros2_ws/* will contain a lot of packages, primarily MoveIt and MTC packages along the usual Bioscara packages. In order to be able to run them, they all have to be built. `colcon` is the ROS2 building tool that is used for this purpose.

Note

This guide assumes that you are in the *lib/ros2_ws/* workspace. If you are not, navigate to it:

```
cd lib/ros2_ws/
```

Building the Entire Workspace including MoveIt2 and MTC

Both MoveIt2 and the MTC are huge packages that will make the Raspberry Pi crash under the load. Even for normal computers there is a high chance of freezing. For this reason the `colcon build` command is invoked with very restrictive parameters to allow the compilation to finish at all. The build command that should be used to compile the entire workspace at once is:

```
MAKEFLAGS="-j1 -l1" colcon build --mixin release --executor sequential --symlink-install␣
↪--continue-on-error > log.out &
```

```
disown
```

The `MAKEFLAGS="-j1 -l1"` restrict the process to a single core, the `--executor sequential` flag ensurs that only one package at the time is built (otherwise up to 8 would be built), the `--continue-on-error` flag will try to keep the compilation going if one package fails. This is very important since the compilation can take up to 10 (!) hours on the Raspberry Pi 4B with 4GB memory.

The high memory usage will cause the Raspberry Pi to become unresponsive at times, potentially dropping the SSH connection and thus the user session that started the build proces. To avoid this the `colcon` output is being piped > `log.out` to a file and the process is being sent to the background `&`. Before disconnecting from the session the process must be disowned by calling `disown`. The build job will now continue even if the user session that started it terminates.

Building selected Packages

Luckil as long as you dont modify any of the MoveIt or MTC packages or accidentally delete the *build/* or *install/* directories, you will not have to build the entire workspace again. To rebuild a selected set of packages simply invoke `colcon` like this:

```
colcon build --mixin release --symlink-install --packages-select <package_name> <package_
↪name> <package_name>
```

or select the by a regex expression `--packages-select-regex <package_name_regex>` or any other (selection flag)[https://colcon.readthedocs.io/en/released/reference/package-selection-arguments.html]

Other Notes

Using the `--symlink-install` flag create symlinks instead of copying files from the source and build directories where possible. This way it is not necessary to recompile a package if a non-C++ file has been modified. Changing parameter yaml-files, Pthyon scripts or URDF files is thus simpler and faster.

The `--mixin release` makes the packages compile in release mode, which is very important for performance reasons. Not using this flag will make path planning very slow.

If compilation fails, in particular if a package's *CMakeLists.txt* has been modified, it can help to selectively clean the *build/* and *install/* directory:

```
rm -rf build/<package_name> install/<package_name>
```

1.1.3 Operate the Robot

This document explains how to bring the robot into operation for manual and programmatic control. The document will decribe how to start the robot hardware as well as its emulation.

Prerequisites

Make sure that following prerequisites are fullfilled:

- In order for the robot to operate, all software and system configurations must be installed as described in the *installation* guide.
- The ROS2 workspace must be compiled as described in the *build* guide.
- **For hardware operation:** The robot must be correctly assembled and all components are electrically connected. The latest hardware information can be found *here*
- A stable wired network connection between the control PC/developers PC and the robot controller is established as described in the *networking tutorial*.

Connect to the Robot Controller

After a connection has been established to the robot controller, open a ssh session from the control PC:

```
ssh scara@<ip-adress>
```

Type the password **dtubio** when prompted and hit enter. You should now have a terminal session on the robot controller and you can proceed.

Starting the Robot

Warning

You are about to enable the robot! Make sure it is securely mounted and its workspace is clear!

Change to the *lib/ros2_ws* directory:

```
cd lib/ros2_ws
```

All following commands are relative to this directory.

Source the workspace packages:

```
source install/local_setup.bash
```

Launch the robot control nodes

To start all ROS2 nodes simultaneously launch the complete launch file:

```
ros2 launch bioscara_arm_gripper_128_moveit_config complete.launch.py
```

This will start the ros2_control node responsible for the control implementation and the MoveIt2 move_group node for trajectory generation.

Note

With no access to the hardware it is possible to emulate it by passing the `use_mock_hardware:=true` argument to the launch file:

```
ros2 launch bioscara_arm_gripper_128_moveit_config complete.launch.py use_mock_
↪hardware:=true
```

Launch the RViz GUI

The RViz GUI is needed for vsizualization and to control the Bioscara hardware, to example home it. Since the robot controller does not have a desktop, the control GUI must either run on a seperate control PC or must be forwared via ssh. In both cases a stable and fast network connection is critical to ensure timely updates.

Using Window Forwarding

This method has the advantage that the external control PC does not need have ROS2 installed. A ssh client capable of X-forwarding is sufficient.

Open a second ssh session from the control PC to the robot controller with X-forwarding enabled:

```
ssh scara@<ip-adress> -X
```

Start the RViz GUI on the robot controller, it will be forwarded to the control PC:

```
ros2 launch bioscara_arm_gripper_128_moveit_config moveit_rviz.launch.py
```

Using an Networked PC with ROS2 installed

Using this method requires the control PC to have ROS2 installed, along with RViz2, the Bioscara RViz plugin, the MoveIt Panel and MTC Panel. The advantage is that the robot controller has less computation load, since it does not need to display the RViz process.

In this case it is sufficient to simply open a new local terminal session. Start the RViz GUI on the robot controller:

```
ros2 launch bioscara_arm_gripper_128_moveit_config moveit_rviz.launch.py
```

Alternative: Manually launching Nodes

If you experience problems with the complete launch file or to try differnent launch configurations, you can try to launch the components indiviudally. First launch the ros2_control node with the *scene_bringup* packages:

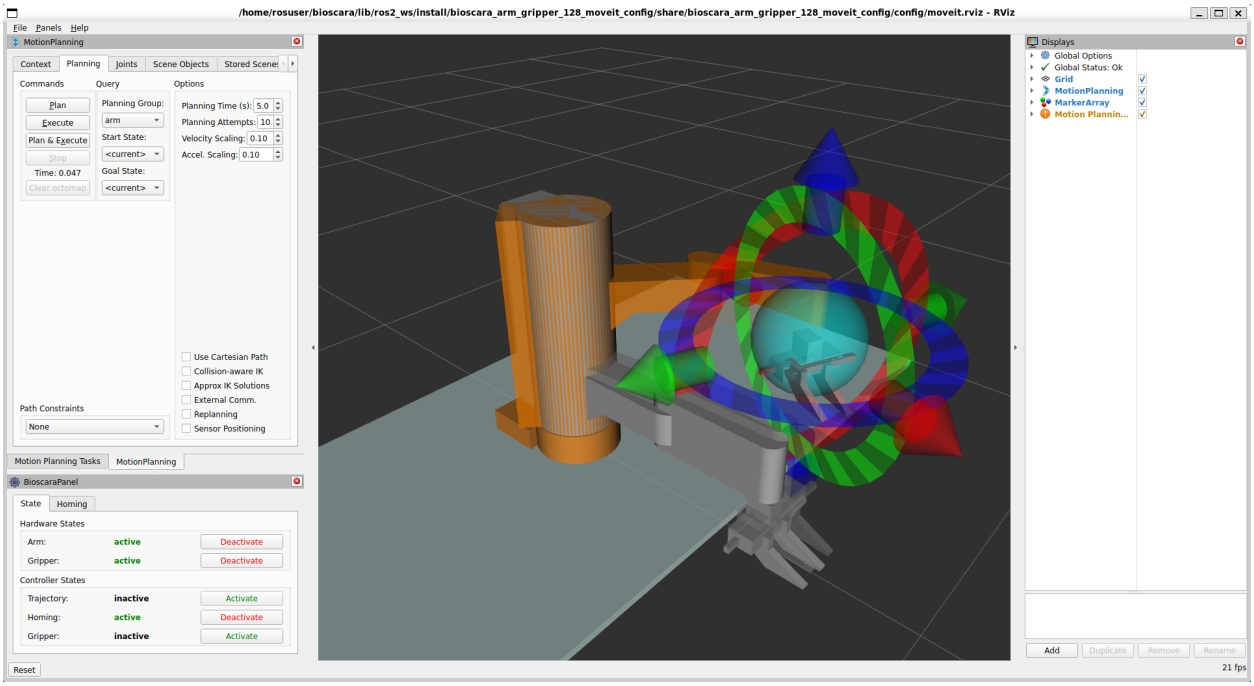
```
ros2 launch scene_bringup bioscara_arm_gripper128.launch.py use_mock_hardware:=true/false
```

Then launch the move_group seperately:

```
ros2 launch bioscara_arm_gripper_128_moveit_config move_group.launch.py
```

Preparing the Robot

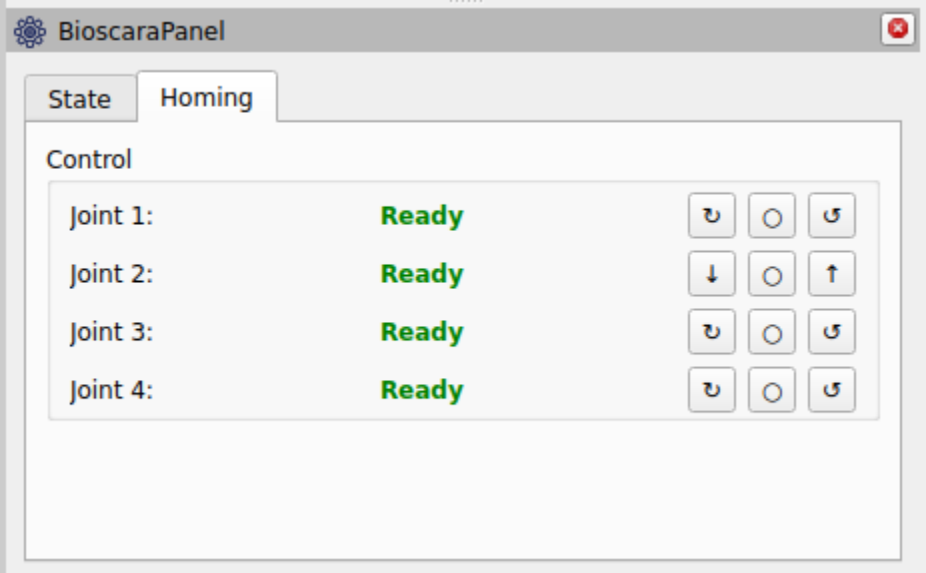
You should now see the RViz GUI like this



The important panel are on the left side. On the top left you can find the MoveIt *MotionPlanning* and MoveIt-Task-Constructor *Motion Plannining Tasks* Panel. But before we get to that the robot has to be configured for operation with the bottom left *Bioscara* Panel.

Homing the Robot

If the robot has been power-cycled, its joint's need to be homed again. To do this activate the homing controller under the "State" tab as displayed in the picture above. The navigate to the "Homing" tab where you should see all joints as "Not Homed". During homing the joint will slowly drive towards its upper or lower limit where it will detect the collission and zero its encoder. You can select the direction through the buttons right of the joint label as it suits you best. Make sure that the joint can move freely and that it actually stops at its endstop. Repeat this until all joints report "Ready" as shown in the picture below:



Enabling the Controllers

Now with the robot homed it is almost ready for operation. Under the "State" tab, deactivate the homing controller then activate the trajectory and gripper controller. The arm and gripper hardware must also be active. All joint motors are now engaged and can not be backdriven. To backdrive the robot deactivate the Arm hardware.

Manually Create Trajectories

It is possible to manually plan trajectories, execute them, compare path planner performance and more with the *MotionPlanning* panel. The official tutorial gives a good introduction hot to use it.

In short: Use the handles to move the orange goal robot state into a desired pose, then press "plan" and/or "execute" under the "Planning" tab. By default the OMPL RRTConnect path planner is used to plan the paths, alternatives can be selected under the "Context" tab.

Planning Scene

Under the "Scene Objects" tab the planning scene can be modified. Collision objects can be added, modified and removed from it, or a a setup can be imported. Always press "publish" to commit the changes to the planning scene.

Planning and Executing a MTC Task

The MoveIt-Task-Constructor (MTC) is used to plan more complex trajectory sequences. The tasks are created in the *dalsa_motion_plans* package. Python scripts are the easiest to edit and create, although it seems that not all C++ methods are fully supported through the Python bindings. The Python scripts are in the *dalsa_motion_plans/scripts* directory.

The MTC can be run either on the robot contoller or the control PC since it is not realtime-critical. Exectute the following steps on the robot controller or control PC respectively.

Adding a new MTC Task

A sort-of correct documentation for the Python bindings can be found here. Using these recipes complex applications can be programmed. Some examples can be found in the */scripts* dir, have fun!

To create a new Python script with a new MTC task the following needs to be done:

- Create the *cool_task.py* under *dalsa_motion_plans/scripts*
- Add the script to the *dalsa_motion_plans/CMakeLists.txt*. Find the following lines and append the the *cool_task.py* to the list of installed files.

```
install(PROGRAMS
        scripts/IAmNotARobot.py
        scripts/PickPlace.py
        scripts/PickPlace_continous.py
        scripts/PickPlace_markers.py
        scripts/cool_task.py                # <===== Append here
        DESTINATION lib/${PROJECT_NAME})
```

- In order to make the script available rebuild the *dalsa_motion_plans* package:

```
colcon build --symlink-install --mixin release --packages-select dalsa_motion_plans
```

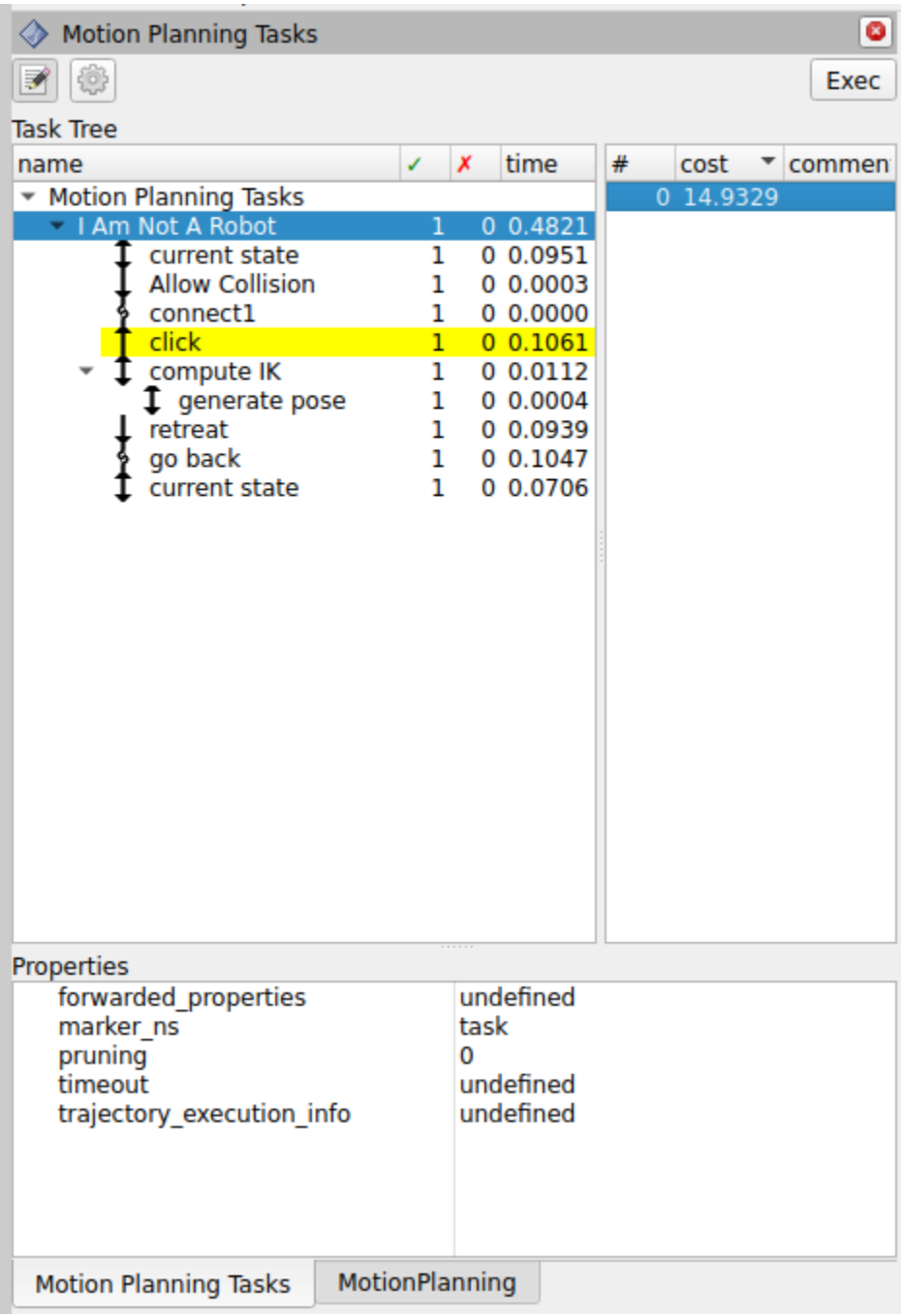
Executing the Task

From a new terminal, navigate to the *lib/ros2_ws* worspace and source it `source install/local_setup.bash`. Then you can execute any MTC task by invoking

```
ros2 launch dalsa_motion_plans run.launch.py exe:=cool_task.py
```

where the `exe` argument specifies the name of the script to execute.

If the task planned succesfully, it is published to the "/solution" topic and displayed in the *Motion Planning Tasks* MTC Panel shown below. For each task a number of solutions can be computed which can be indivdually inspected and, if one is selected, executed by pressing the Exec button.



Common Problems

Some common issues and remedies are described in the following.

General Launch Problems

If nodes fail to launch and no clear error message is printed in the console try one or a combination of the following:

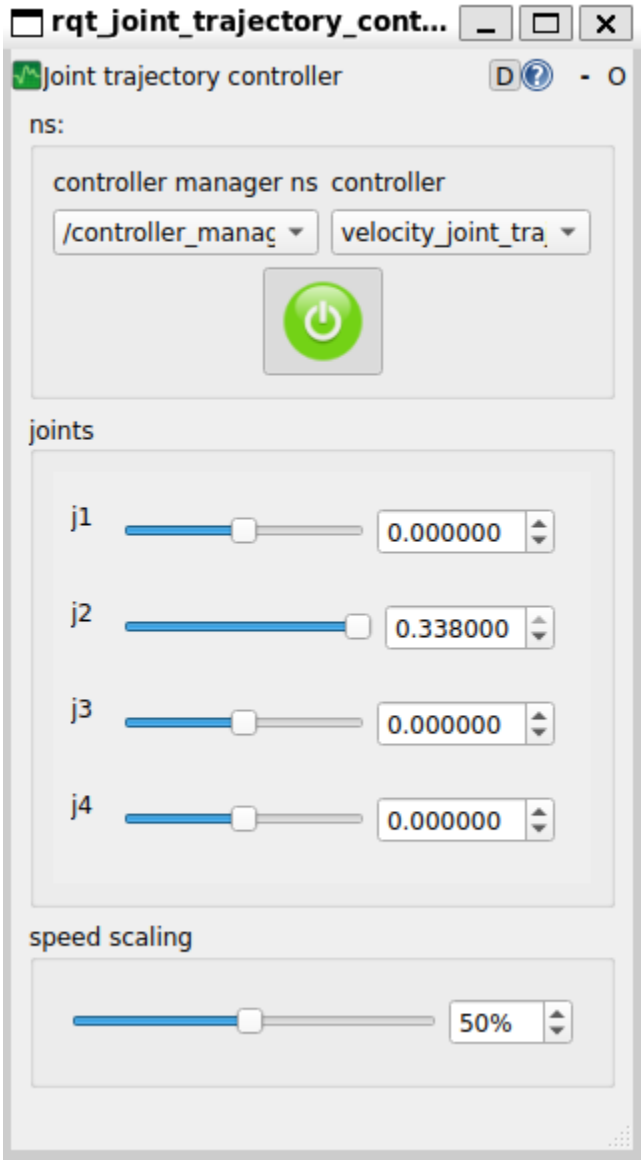
- Open a new terminal and source the workspace again
- Build the workspace or the affected packages again and execute the above step.

- Remove the affected packages from the *lib/ros2_ws/install/<package>* and *lib/ros2_ws/build/<package>* directory and execute the above steps.

Motion Planning Fails

This can have many reasons, the exact move_group error usually helps to identify the issue.

- **Start state out ouf bounds:** If a joint is just a tenth of a millimeter outside its limit, motion planning will fail. If the gripper is causing the violation, manually send a new width command through the console `ros2 action send_goal /gripper_controller/gripper_cmd control_msgs/action/ParallelGripperCommand "{command:{name:[gripper],position:[0.05]}}"`. If the problematic joint is on the robot arm and not J2, just deactivate the robot arm, move the joint away from the limit and reactivate the arm again. If the problematic joint is J2 it can not be moved by hand. Instead do the following: Open another terminal navigate to *lib/ros2_ws/*, source the workspace, and start the rqt_joint_trajectory_controller: `ros2 run rqt_joint_trajectory_controller rqt_joint_trajectory_controller`. Select the "controller_manager" and "velocity_joint_trajectory_controller", enable it on the red button and manually drag the J2 set point away from the limit. This GUI can also be used for other purposes but it should never be active in parallel with an ongoing trajectory through MoveIt.



- **Start state collision:** If a trajectory is planner immediatly after homing , the gripper links can be in a self-collision with link link_1, since the collision body is a primitive cylinder. Simply deactivate the arm and move the links out of collision. The collided links will be highlighted in red in RViz.

ros2_control Fails to Start

Sometimes, for yet unknown reasons, if the ros2_control node is launched from the complete launch file, it attempts to start the robot hardware despite the `use_mock_hardware:=true` has been set. If that happens while no hardware is connected, the node will fail to start and control is impossible. Launching the ros2_control node separately as described in *here* seems to resolve this issue.

Homing Sequence not Starting/Finishing

As described in this bug report, under yet unclear conditions, the homing sequence is immediatly aborted after it has been started. The joint driver however assumes it is sill ongoing while the firmware has already finished it. This issues can be fixed by:

1) Stop the homing sequence by pressing the o button
2) Restart homing

1.1.4 Network Setup

This document describes the network architecture and possibilities to establish inter-device connections.

Architecture

If the network configuration described in the *installation instructions* are followed, then the robot controller and control PC are set-up for the network architecture as follows:

The Laptop is optional, but useful during development. In case a Windows hotspot is used the IP addresses are 192.168.137.X (Subnetmask 255.255.255.0).

Permanent Ethernet LAN

If only the control PC and the robot controller are in the network, the switch can be omitted and the devices connected directly to each other via an Ethernet cable. The devices should see each other on their respective IP adress.

Note

Although this should work in theory, it has not been tested yet. A third developer PC has always been connected and a switch employed.

If a third device needs to join the network, a network switch is needed to route the trafic.

Temporary WiFi Connection

Both the robot controller and control PC should be configured to automatically connect to a WiFi network with the the following credentials:
SSID: DALSA_IOT
Password: dalsa_iot

It is possible to gain access wirelessly by creating such an access point. An easy solution is a smartphone or laptop WiFi hotspot. In the hotspot's settings the IP adress of the connected device will show up if it has sucessfully connected.

Semi-Permanent Internet Access through DTUsecure

There are two options to access the internet.

Through the temporary WiFi AP

Create the access point as described in the previous section, and the device will have internet access forwarded via the hotspot. This method is simple, yet the internet connection is slow and unstable. Not ideal to pull a large repository from Github for example.

Via DTUsecure

Connect the device to the DTUsecure WiFi network.

The easiest way is to simply execute the connection script *installation/scripts/connect_DTUsecure.sh*:

If the robot should be connected to DTUsecure, for example for a faster connection, run the provided script *installation/scripts/connect_DTUsecure.sh*. DTUsecure uses the WPA-EAP authethication mechanism with your DTU id and DTU password. The script has been made to simplify the setup.

Caution

Due the nature of the OS, your DTU accounts password will be saved in plain text in a netplan file on the system if the connection shall be saved persistently betweeen reboots! Follow the scripts output to see where.

then the script can be run simply by invoking:

```
sudo bash installation/scripts/connect_DTUsecure.sh
```

The script adds a connection via nmcli, and asks the user for DTU id and password and if the credentials should be saved.

Further Information

Managing networks on Linux can be confusing, for example if interfaces show up as 'unmanaged' the chances are high that a different network manager has control over the interface. This table gives a good overview which programs might be involved. Note that Netplan is not mentioned there, since Netplan only takes a standardized configuration file as input and then depending on the renderer creates the configuration for the backend. Currently Netplan only works with systemd-networkd and NetworkManager, the latter is much simpler to use with its CLI `nmcli`, and terminal UI `nmtui`.

1.2 Developer Guides

1.2.1 Standalone Gripper Operation

This document describes the simplest way to manually control the gripper. This is desireable for development and necessary after assembly to recalibrate the grippers reduction and offset. The current gripper is driven by a PWM controlled RC servo. The frequency is 50 Hz. The PWM should be a hardware generated PWM as software PWM has potentially high jitter which could cause overheating. The gripper is controlled through its C++ API defined in the *bioscara_gripper_hardware_driver::Gripper* object.

Connect to the Robot Controller

Follow the *networking tutorial* to establish a connection to the robot controller. When a connection has been established login to the controller via ssh:

```
ssh scara@<ip-adress>
```

Type the password **dtubio** when prompted and hit enter. You should now have a terminal session on the robot controller.

Manual Operation

The gripper can be manually controller by executing the `gripper_manual_control` program. First, ensure that the entire workspace is built by following the *build* instructions. At a minimum, the `gripper_manual_control` and `bioscara_gripper_hardware_driver` packages must be built. You can manually build them by executing:

```
cd lib/ros2_ws/
colcon build --symlink-install --packages-select gripper_manual_control bioscara_gripper_
↪hardware_driver
```

This system test package contains a simple programm that allows to manually control the gripper actuator. The program can be used in two ways:

1) Setting the desired width. This can only be used when the correct reduction and offset is known. **If they are unkown**, use the 2) mode and follow the steps described there.

2) Setting the servo angle directly. This should be used for testing, positioning when mounting and dismounting and to calculate reduction and offset (explained *here*).

Running the program

Open a new terminal session on the robot controller, navigate to the *lib/ros2_ws/* directory and source it.

```
cd lib/ros2_ws/
source install/local_setup.sh
```

Tip

Sourcing the executables is only is necessary once after opening a new terminal.

You can then run the executable:

```
ros2 run gripper_manual_control manual_control
```

Now follow the instructions given in the console output.

Calculating reduction and offset

The gripper has the reduction r and offset o parameters which are used to translate from a desired gripper width to the servo angle. These paramters must be supplied to the gripper hardware interface when used for normal operation.

Warning

The gripper will be damaged, if the reduction and offset paramters are incorrect. The parameters must always be calibrated after dissassembly and on initial commissioning.

The relationship between gripper width w and acutator angle α is as follows: $\$\alpha = r(w - o)\$$

To determine these parameters execute the following steps:

1. Manually set the gripper to an open position by setting an actuator angle. Be careful to not exceed the physical limits of the gripper since the actuator is strong enough to break PLA before stalling.
2. Measure the gripper width between the jaws w_1 and note the commanded actuator angle α_1.
3. Move the gripper to a more closed position that still allows you to accurately measure the width.
4. Measure the second width w_2 and note the corresponding angle α_2.
5. Calculate the offset o:
$$o = \frac{\alpha_1 w_2 - \alpha_2 w_1}{\alpha_1 - \alpha_2}$$
6. Calculate the reduction r:
$$r = \frac{\alpha_1}{w_1 - o}$$

1.2.2 Modifying and Flashing the Joint Firmware

The joint firmware is programmed in C++ using the Arduino framework since the uStepperS32 stepper library is released for that ecosystem.

Prerequisites

In order to flash firmware to the uStepperS32 stepper controllers some prerequisites must be met. The following guide is copied for reference from the uStepper website.

Install the Arduino IDE

Installing the uStepperS32 library and flashing the firmware is done through the Arduino IDE which needs to be installed on the developer's computer from here.

Install STM32 CUBE Programmer

The uStepper is based on an STM32F401 microcontroller. The STM32 CUBE Programmer is used behind the scenes to flash the firmware from the Arduino IDE. Install the programmer from here.

Add the uStepperS32 library to the Arduino IDE

In the Arduino IDE, go to *File->preferences* and append the URL `https://raw.githubusercontent.com/uStepper/uStepperSTM32Hardware/master/package.json` to the Additional Boards Manager URLs field at the bottom and accept with OK.

Then install the uStepper board through the board manager under *Tools->Board->Boards Manager*. Search for "ustepper" and the board should show up after the board manager has refreshed. Then press INSTALL.

Finally it should now be possible to select the uStepper STM32 boards under *tools->boards*.

Install the uStepperS32 Library

Go to *Sketch->Include Library->Manage Libraries*, search for "uStepperS32", and press INSTALL. Now both the hardware and the library are setup for compilation and flashing.

Flashing the Firmware

To flash the firmware on the uStepper board, open the *lib/joint_firmware/joint/joint.ino* firmware file with the Arduino IDE, connect the uStepper board with a USB-C cable to the computer, select the correct serial port and the board as "uStepper STM32 boards" under the "board and port" tab.

The follow the steps to flash the firmware:

1. Press and hold the BOOT button on the uStepper board
2. Keep the BOOT button is pressed and press the RESET button for at least 0.5 s
3. Release the BOOT button
4. Select UPLOAD in the Arduino IDE. The board is now in dowload mode and will load the new firmware.

Monitor the Arduino IDE output console for potential error messages.

Other Notes

The buttons on the uStepper board are very difficult to reach on J2 if it is mounted on the robot. A thin non-conductive stick can be used to press the BOOT button, the RESET button should be reachable by hand.

1.2.3 Building the Documentation locally

The documentation consists of two independent parts. The Doxygen-based C++ API documentation and these Sphinx user guides.

First Steps

All following commands are executed from the *~/bioscara/docs/* directory:

```
cd docs
```

Create the Virtual Environment and Install Dependencies

It is recommended to use a virtual environment to manage all dependencies. These steps need to be executed only once at the first time. Create the virtual python environment:

```
python -m venv .venv
```

Before installing the dependencies the environment needs to be activated:

```
source .venv/bin/activate
```

The we can install the required packages to build the documentation:

```
pip install -r requirements-docs.txt
```

This installs Sphinx and its packages.

Aditionally install doxygen

```
sudo apt-get update
sudo apt-get install doxygen
```

And install all LaTeX tools:

```
sudo apt-get install texlive texlive-font-utils texlive-fonts-recommended texlive-latex-
↪extra latexmk
```

Building the Documentation

First make sure the virtual environment is activated:

```
source .venv/bin/activate
```

Then run doxygen from the *docs/doxygen/* directory to produce HTML, XML and LaTeX output:

```
cd doxygen
doxygen
```

Compile the LaTeX to a PDF:

```
cd doxygen/latex
make
```

First compile it as a PDF (the HTML will link to it later)

```
sphinx-build -M latexpdf . sphinx/latex
```

Finally build the Sphinx documentation:

```
sphinx-build -b html . sphinx/html
```

1.2.4 Creating a IKFast Inverse Kinematics Plugin

Important

This guide can only serve as a starting point since the desired result could not be achieved!

The IKFast is an analytical inverse kinematics solver that automatically generates a robots IK solver code based on its URDF file. This guide documents which steps have been undertaken to generate the IKFast solver plugin for MoveIt.

The result however has always been that the robot did not move when dragging on its handles in MoveIt.

Background

Please refer to the original guide on the MoveIt documentation page here for the information about the individual commands.

Plugin Creation

make sure Docker is installed:

```
docker -v
```

If that is not the case install Docker by following the guide here.

if that is done, make sure your user is added to the "docker" group:

```
sudo usermod -aG docker ${USER}
```

log out and back in to apply the new group:

```
su -l ${USER}
```

navigate to the *lib/ros2_ws/src/dalsa_bioscara_arm* directory:

```
cd lib/ros2_ws/src/dalsa_bioscara_arm
```

export the arm as an ENV variable:

```
export MYROBOT_NAME="bioscara_arm"
```

Create a urdf file from the *scene.xacro* which will be parsed by the plugin generation script.

```
ros2 run xacro xacro -o $MYROBOT_NAME.urdf bioscara_arm_description/urdf/scene.xacro
```

Then create then call the plugin creation script:

```
ros2 run moveit_kinematics auto_create_ikfast_moveit_plugin.sh --iktype <type> $MYROBOT_
↪NAME.urdf arm base_link tool_flange
```

On first run this will take a while to pull and build the docker images.

Select one 4-DOF <type> from the following (source):

- **Transform6D** - end effector reaches desired 6D transformation
- **Rotation3D** - end effector reaches desired 3D rotation
- **Translation3D** - end effector origin reaches desired 3D translation
- **Direction3D** - direction on end effector coordinate system reaches desired direction
- **Ray4D** - ray on end effector coordinate system reaches desired global ray
- **Lookat3D** - direction on end effector coordinate system points to desired 3D position
- **TranslationDirection5D** - end effector origin and direction reaches desired 3D translation and direction. Can be thought of as Ray IK where the origin of the ray must coincide.
- **TranslationXY2D** - end effector origin reaches desired XY translation position, Z is ignored. The coordinate system with relative to the base link.
- **TranslationLocalGlobal6D** - local point on end effector origin reaches desired 3D global point. Because both local point and global point can be specified, there are 6 values.
- **TranslationXAxisAngle4D**, **TranslationYAxisAngle4D**, **TranslationZAxisAngle4D** - end effector origin reaches desired 3D translation, manipulator direction makes a specific angle with x/y/z-axis (defined in the manipulator base link's coordinate system)
- **TranslationXAxisAngleZNorm4D**, **TranslationYAxisAngleXNorm4D**, **TranslationZAxisAngleYNorm4D** - end effector origin reaches desired 3D translation, manipulator direction needs to be orthogonal to z, x, or y axis and be rotated at a certain angle starting from the x, y, or z axis (defined in the manipulator base link's coordinate system)

None of the last 6 types have yielded a successfull result yet.

Build the new Plugin Package

Navigate to the root of the workspace:

```
cd lib/ros2_ws
```

and build the new package:

```
colcon build --symlink-install --mixin release --packages-select bioscara_arm_ikfast_
↪plugin
```

Change the kinematics.yaml

Almost done, change the *lib/ros2_ws/src/moveit_configurations/bioscara_arm_gripper_128_moveit_config/config/kinematics.yaml* file to the new IK solver:

```
arm:
  kinematics_solver: bioscara_arm/IKFastKinematicsPlugin
  ...
```

Try it out

Start the robot (mock or hardware), move_group and rviz as described in the *guide*.

1.3 Hardware

1.3.1 Supplementary Assembly Instructions

The hardware has been modified in comparison the original Bioscara project.

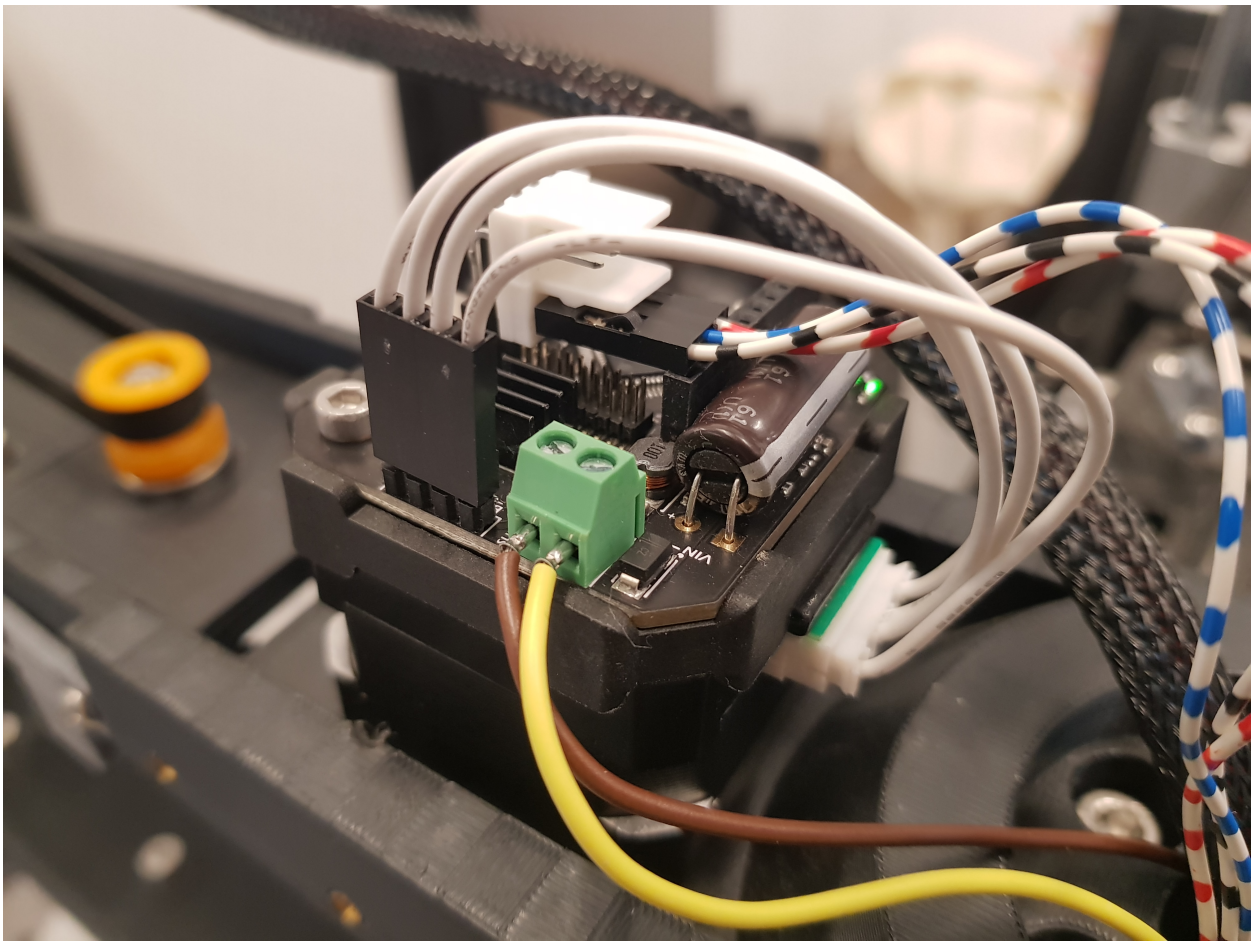
Key Modifications

The key changes were:

- Replacing the MKServo42C closed-loop stepper drivers with UStepper S32 drivers
- Removing the optical endstops and replacing them with physical stop blocks for sensor less homing
- Removing the associtated wiring to the optical endstops and MKServo drivers.
- The UStepper drivers only require two signal wires (SDA/SCL) and the 24V/GND power cables.
- The electronics compartment has been simplified since the 3D-printer motherboard could be removed.
- All wires have been replaced and properly sized and terminated.
- Power distribution is achieved through terminal block mounted on DIN-rails.
- The gripper has been redesgined.

Joint driver

The new joint driver is mounted on the back of the stepper motors just like the MKServo 42C drivers.

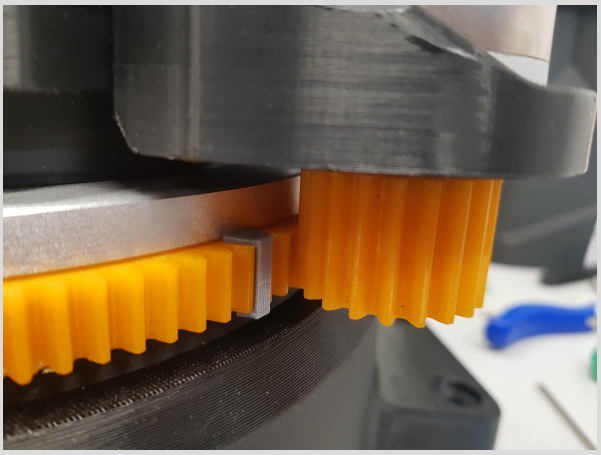


TO-DO:

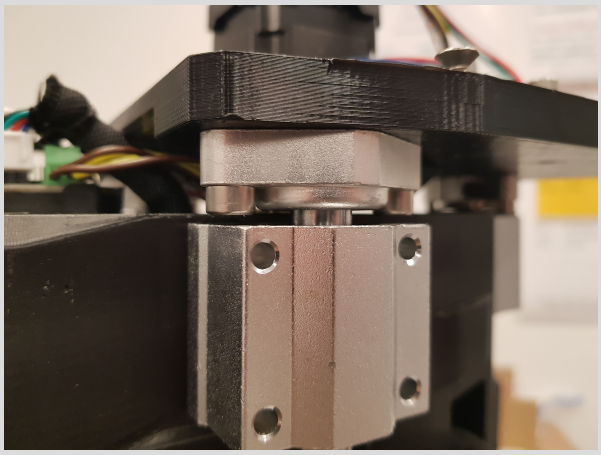
- Increase the cutout dimensions in the J2 covers and J1 Top cover to make room for the slightly larger UStepper mounts.

Endstops

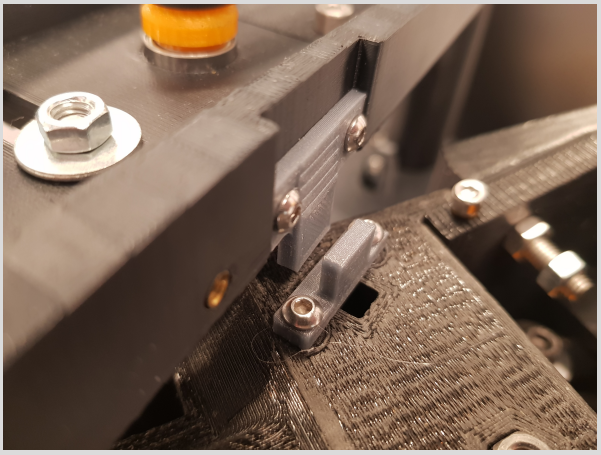
See the new endstops in the figures below. Their STL file can be found in the *hardware/meshes/* directory.



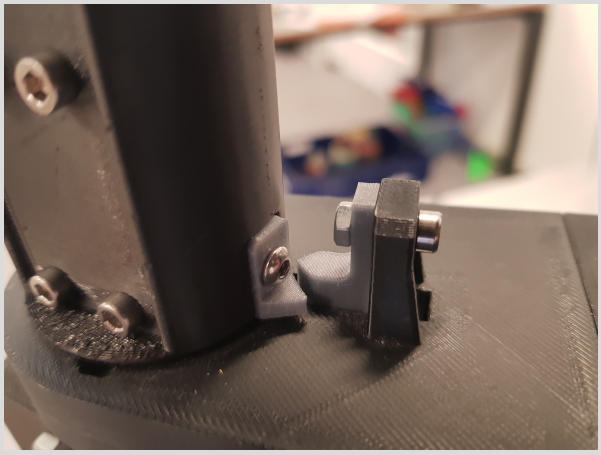
J1 Endstop



J2 Endstop



J3 Endstop



J4 Endstop

New Gripper Assembly

The new gripper was designed in June 2025. The results are presented in a presentation and report.

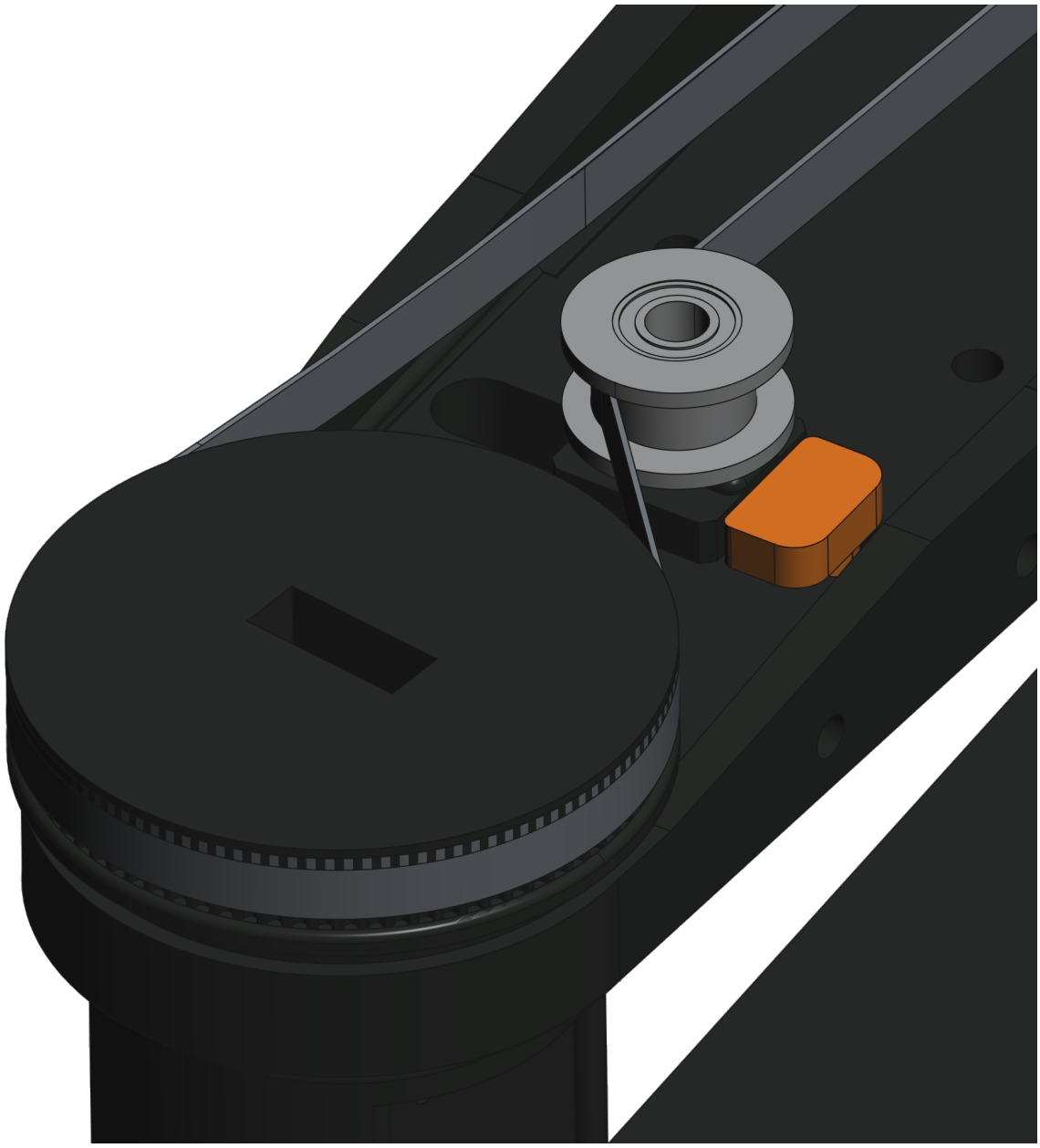
The CAD files can be found under the *hardware/CAD Model/gripper_v2* directory.

Tensioner Pulley Slipping

The J4 belt drive needs to be redesigned, since the tensioner pulley slips under load, releasing tension. The current workaround is the *tensioner_stop_block* that is inserted in the slot to prevent the pulley from sliding back. The stop block needs to be printed and inserted as shown in the figure.

TO-DO:

- Fix the design, for example by “knurling” the surface to increase its roughness to increase friction.



Electrical Design

The electrical schematic can be found here.

The schematic shows how to wire the new joint controllers and how to distribute the power in the terminal blocks. For indicative cable lengths and wire gauges refer to the schematic as well.

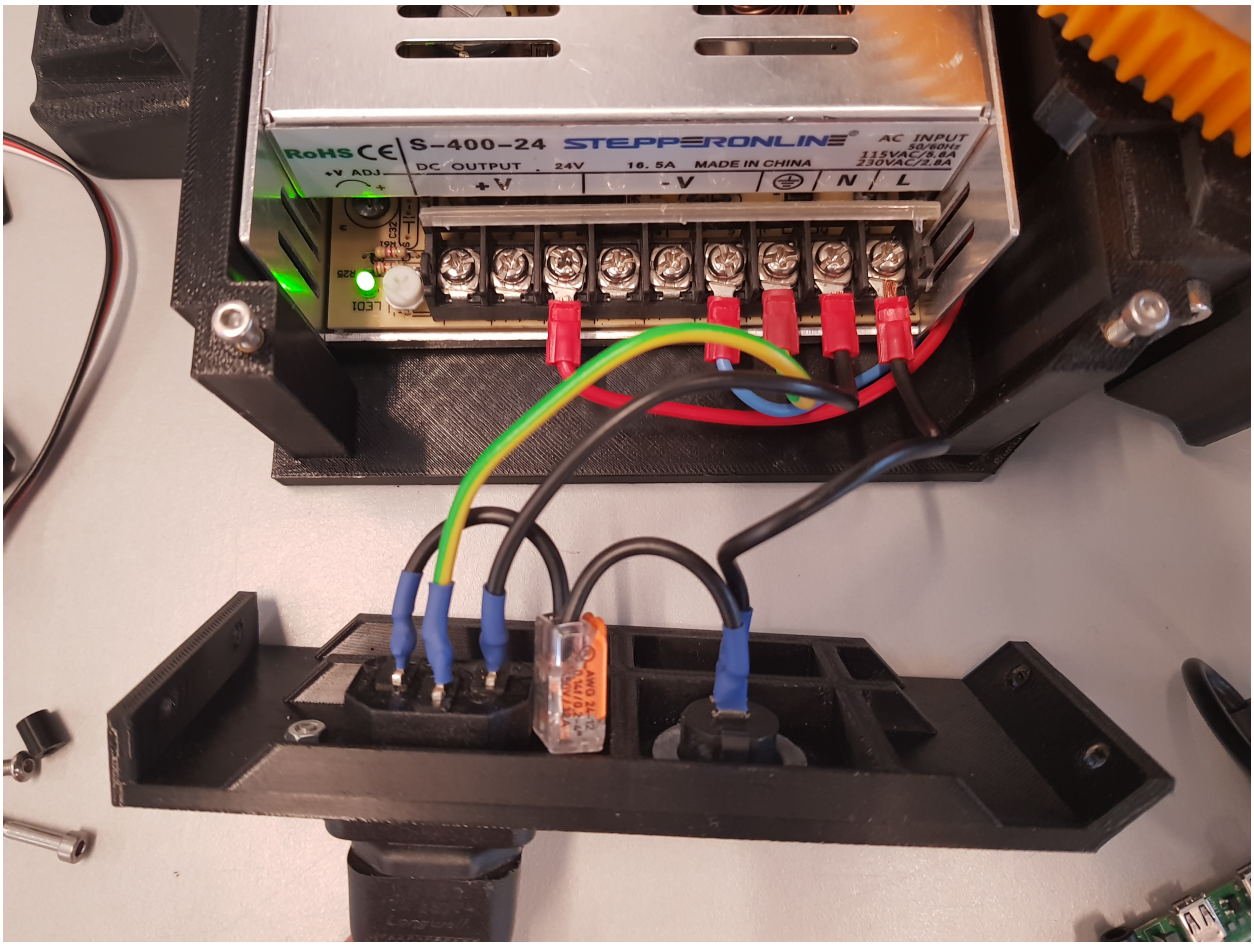
Important

All wires shall be appropriatly sized and terminated with spade connectors for the power supply and ferrules for screw terminals.

Note

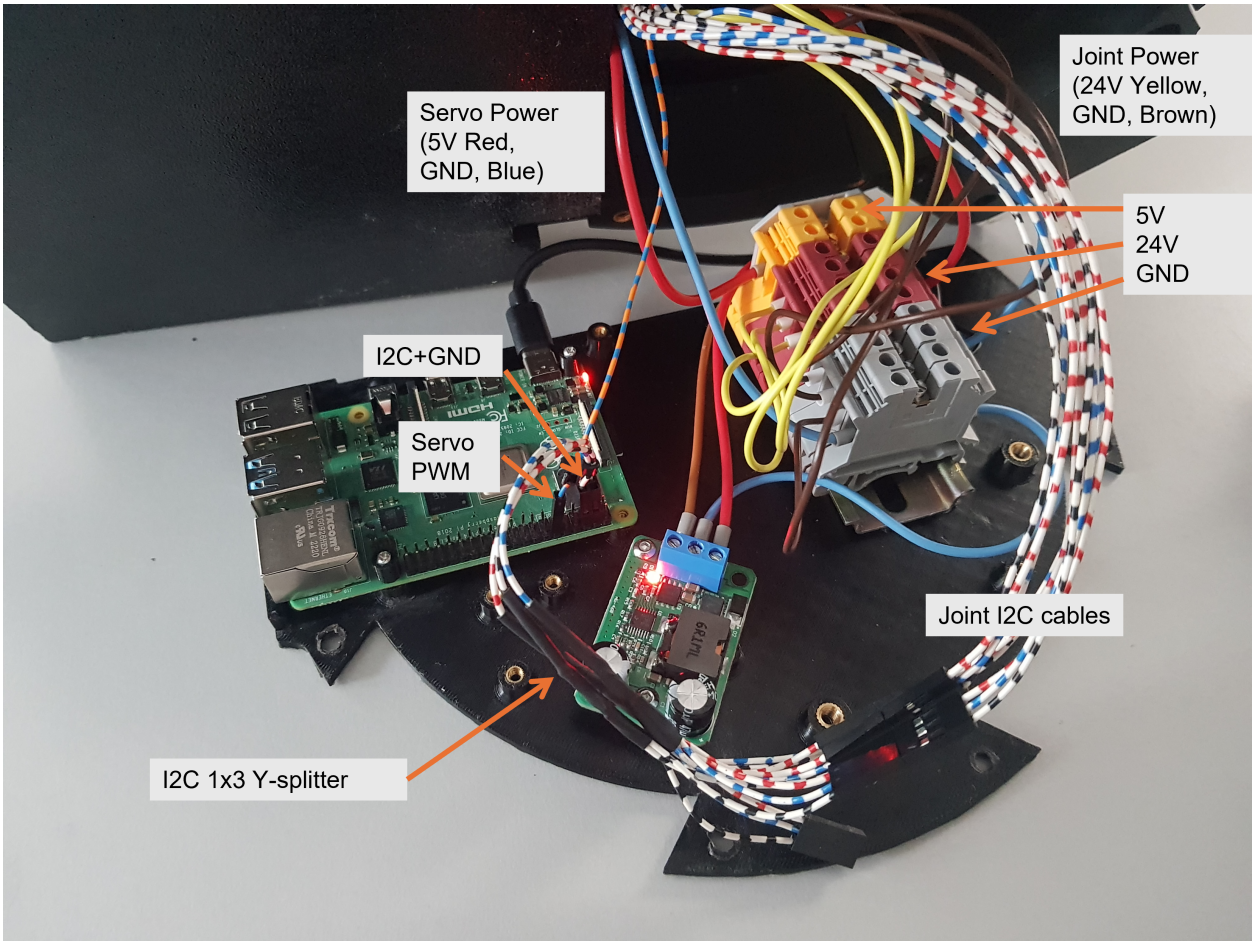
Some images may display a seperate GND connection with the I2C wires. This is no longer correct. The wiring as displayed in the schematic with only a single GND wired to the power terminals is correct.

The AC wiring was redone. Note that the wires are properly terminated. The power switch was replaced with a simple switch.



Simplified Electronics Compartment

The 3D-printer motherboard and level-shifter was removed from the base plate. The number of wires was reduced. The new design is shown in the figure below.



TO-DO:

- Redesigning the electronics base plate to be able to properly mount all components. The terminal block are currently just glued in place.

Assembly Instructions

The assmbley instructions are given in this pdf.

> **Tip**
>
> A video of the original assembly can be found in this YouTube video.

TO-DO:

- The instructions have only been preliminarily updated. A unified assembly guide with up to date images and information needs to be compiled.
- The current assembly guide is PowerPoint presentation. This file format is very poor for version control. The new assembly guide needs to full-fill the following criteria:
  - Platform independent: The new guide shall be easily maintainable from different platforms. PowerPoint is poorly portable between operating systems.
  - Version Control: The new guide shall be written in a text based format that supports version control such as this document or Latex.

CAD Model and STL Meshes

The complete STEP CAD model can be found in the *hardware/CAD_model/* directory. The STEP file is compressed as a ZIP archive to be able to upload it to Github. The STL meshes for 3D-printing are in the *hardware/meshes/*.

Tip

The original, now outdated, Bioscara_v1 STEP files can be found here

TO-DO:

- The coordinate systems of the CAD model are completely wrong and need to be fixed
- The CAD model does not include the simplified arm shape used in the robot description URDF files, this remains to be added.

1.4 Source Code Documentation

You can find the detailed C++ API documentation and a further information about the ROS2 workspace here.

1.4.1 ROS2 Package Architecture

The *lib/ros2_ws/src* directory in the ROS2 workspace contains all packages needed to develop, test and operate the Bioscara robot. Logically coherent software is combined in a package which is useful for distribution and dependecy management.

Remarks on Nomenclature

- The robot arm (without any gripper) is refered to as the "arm" or "bioscara_arm".
- a custom bioscara gripper is refered to as the "gripper" or "bioscara_gripper_<type>"
- The assembly of an arm and a gripper is a "robot" or "bioscara_<arm>_<gripper>" .
- An environment with a robot and potentially other objects is a "scene".

Design goals

The package architecture is designed on modularity, since it is expected future variants of hard- and software will be developed. The modular approach allows to mix and swap old the software. Modularity achieved two fold:

- Standalone packages for arms and grippers.
- Grippers that are simply variants but utilize the same hardware interface, just different hardware description and hardware parameters grouped in a package but with standalone description macros.
- Packages and description macros can be swapped via command line arguments or set in scene specific bringup files.

The architecture may serve as a blueprint for the future integration of more DALSA robots (custom or commercial) into ROS2.

Additional MoveIt2 packages

Unfortunately it was necessary to build MoveIt2 from source to use features from the MoveIt Task Constructor, installing MoveIt from source adds many packages to the workspace. To build the workspace follow the *compilation instructions*.

Meta Packages

Packages that logically belong together are grouped in meta packages. Meta packages do not contain source code or configuration files but simplify dependency management. For example a hardware meta package may contain the hardware's description, driver, interface, controller and bringup packages. Other packages that depend on that hardware can simply specify the meta package as a dependency and automatically inherit the remaining sub-dependencies.

The following meta packages are available:

dalsa_bioscara_arm: Contains all packages related to the custom bioscara arm hardware component

dalsa_bioscara_grippers: Contains all packages related to the custom bioscara gripper hardware component and its variants.

dalsa_controllers: Contains custom, hardware agnostic controllers.

dalsa_moveit_configurations: Contains MoveIt2 configurations.

system_test_packages: Contain different simple programs to execute system tests.

Overview

The diagram below shows a simple package architecture and dependency diagram. It does not yet include the MoveIt configuration package. The individual packages are grouped by their meta package.

Hardware component package

Hardware component meta packages are structured according to the RTW Package Structure refrence, developed by dstogl, a key developer of ros2_control. Each hardware component *dalsa_bioscara_grippers* and *dalsa_bioscara_arm*, i.e. the arm and the gripper have meta package structured as follows, using gripper as an example:

```
dalsa_bioscara_grippers/
├── bioscara_gripper_bringup                                # The bringup package is␣
↪used to launch a hardware component, either standalone or from another bringup package
│   ├── CMakeLists.txt
│   ├── config
│   │   ├── bioscara_gripper_controller_manager.yaml        # ros2_control controller␣
↪manager paramters if used stand-alone
│   │   └── bioscara_gripper_controllers.yaml               # Definition of controllers␣
↪for this hardware component, gripper controller in this case
│   ├── launch
│   │   └── bioscara_gripper.launch.py                      # Use this launch file used␣
↪to start the component stand-alone, including GUI and ros2_control stack.
│   └── package.xml
├── bioscara_gripper_descriptions                           # The description package␣
↪contains all files to assemble a component description with Xacro
│   ├── CMakeLists.txt
│   ├── config
│   │   ├── bioscara_gripper_128_parameters.yaml            # Describes static component␣
↪parameters parsed in the robot description, like joint limits
│   │   └── bioscara_gripper_<variant_X>_parameters.yaml    # Parameters of another␣
↪gripper variant
│   ├── launch
│   │   └── view.launch.py                                  # Use this launch file to␣
↪simple view and inspect the assembled component.
│   ├── meshes                                              # Resources for the URDF file
│   ├── package.xml
```

(continues on next page)

(continued from previous page)

```
│   ├── rviz
│   │   └── display.rviz                                    # Saved display␣
↪configuration for the stand-alone GUIs
│   └── urdf
│       ├── bioscara_gripper_128.ros2_control               # ros2_control hardware␣
↪information for the 128 mm variant
│       ├── bioscara_gripper_128.urdf                       # Base URDF file containing␣
↪kinematic, visual and collision info for the 128 mm variant
│       ├── bioscara_gripper_128.xacro                      # Contains the "load_gripper
↪" macro, combining the .urdf and .ros2control parts for the 128 mm variant
│       ├── bioscara_gripper_<variant_X>.ros2_control
│       ├── bioscara_gripper_<variant_X>.urdf
│       ├── bioscara_gripper_<variant_X>.xacro              # Variants of the gripper␣
↪using the same hardware interface can be added here
│       ├── materials.xacro
│       └── scene.xacro                                     # bringup/launch file uses␣
↪this scene for the stand-alone start of the component.
├── bioscara_gripper_hardware_driver                        # The hardware driver␣
↪contains the hardware specific implementation, like PWM generation for the gripper
│   ├── CMakeLists.txt
│   ├── include
│   │   └── bioscara_gripper_hardware_driver
│   ├── package.xml
│   └── src
│       ├── mBaseGripper.cpp                                # Base implementation
│       ├── mGripper.cpp                                    # Hardware implementation
│       └── mMockGripper.cpp                                # Mock implementation for␣
↪testing
├── bioscara_gripper_hardware_interface                     # The hardware interface␣
↪abstracts the hardware driver and creates a hardware componen plugin for ros2_control
│   ├── CMakeLists.txt
│   ├── bioscara_gripper_hardware_interface.xml
│   ├── include
│   │   └── bioscara_gripper_hardware_interface
│   ├── package.xml
│   └── src
│       └── gripper_hardware.cpp
└── dalsa_bioscara_grippers                                 # Meta package
```

The name of the gripper component package is written in plural since it can include multiple variants of the bioscara gripper which use the same hardware interface but might have differen geometries.

Bringup Packages

The bringup packages inside hardware components contain the ros2_control controller configurations for that specific hardware component, as well as a controller manager configuration if the hardware component is launched stand-alone with the launch file included in the bringup package. Stand-alone launch includes the ros2_control stack and can be used for example hardware interface testing, without having to also start higher level applications like MoveIt.

Description Packages

Description packages are a core building block of the modular principle. It contains robot description files and control configuration files. A robot description file is an integral part in many ROS2 applications. ROS2 uses an XML based format called Unified Robot Description File (URDF) which is "*a format to describe the kinematics, dynamics, and geometries of robots, independently of software program*" [1], in detail this includes a systems:

- Kinematic description (joints and frame definitions)
- Visual properties for viszualization
- Collision properties defining physical boundaries for trajectory generation
- Mass and Inertial properties for physical simulation (not utilized in this project)
- The URDF can be expanded for specific applications, in this project it addtionally includes:
- ros2_contol configuration
  - hardware interface plugin
  - command and state interfaces
  - joints
  - gpios

The URDF files can be combined using Xacro, a XML macro extension. With Xacro a virtual robot assembly can be created, allowing to flexibly match and mix hardware components. A desription of the Xacro macros follows in the next section.

The description package also contains a stand-alone *scene* and launch file which can be used to simply display the geometry visuals and collision bodies, test joint limits using joint state broadcaster and frame transformations.

Dynamic Robot Assembly

The entire robot is assembled in the *scene_descriptions* package, in particular the *load_robot* macro defined in *scene_descriptions/robot.xacro". The diagram below tries to viszualize the different Xacro macros that make up the robot. The robot can be modified simply either by passing diffferent arguments to the macros, loading different parameters from the paramter files or swapping entire macros. For example the *scene.xacro* file takes a gripper macro path as an argument, thus an entirely different gripper can be loaded as long as it supports the same input arguments. Alternatively one can decide to not mount a gripper at all to the robot by passing an empty gripper macro name.

After assembling the bioscara robot in its default state the following kinematic chain is created:

Root Level Scene Packages

Scene Description

The main *scene.xacro* file describes the robot and its environment, in this case a table top. It loads the *load_robot* macro defined in the *robot.xacro* file. The resulting robot has been described in the *Robot Assembly section*.

Scene Bringup

Contains the launch files of a specific robot configuration.

Tip

This package contains the main launch file to start a robot configuration including the ros2_control stack using the *<RobotConfiguration>.launch.py* launch file. Start Bioscara with the currently mounted 128 mm gripper, execute

```
ros2 launch scene_bringup bioscara_arm_gripper128.launch.py use_mock_hardware:=true/
false gui:=true/false.
```

Note

This launch files might be superseeded by a single launch file that additionally launches the MoveIt move_group from a hardware specific dalsa_moveit_configurations package.

1.4.2 C++ API Documentation

The C++ API documentation is generated with doxygen and can be found here.

The PDF version of the API documentation can be dowloaded here: `bioscara_api.pdf`