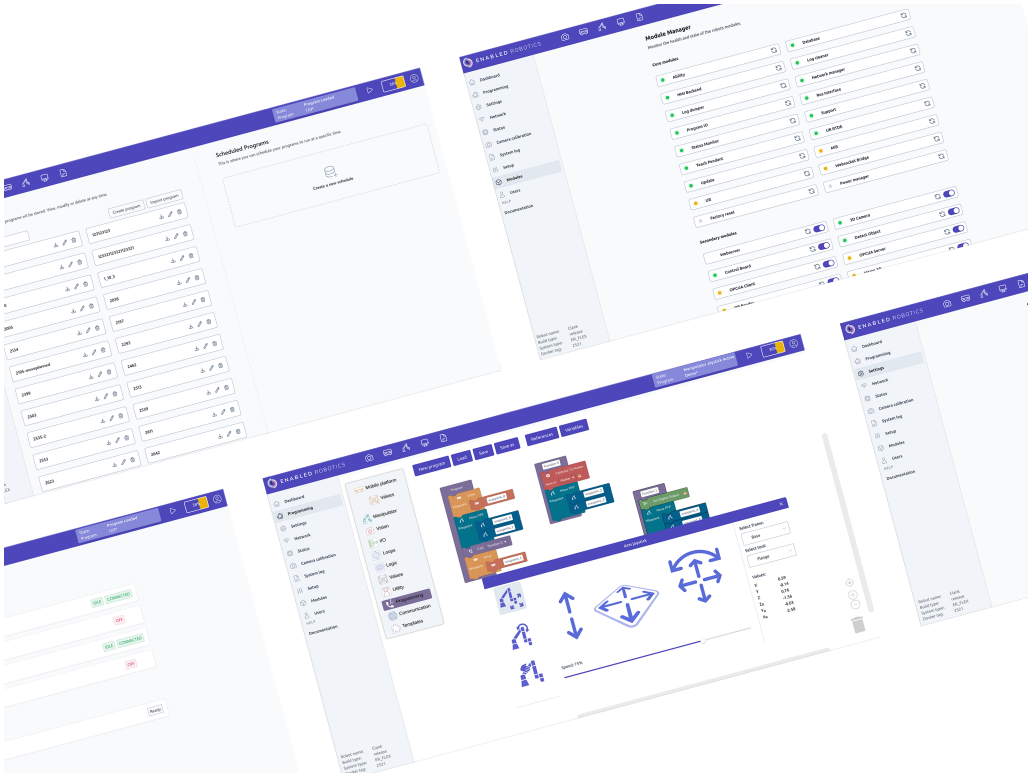
ENABLED ROBOTICS

ABILITY User Documentation

Blocks, Interfaces, System settings

Software Version 2.14.0



CONTENTS

CHAPTER

ONE

GENERAL CONTROL

There are two modes for controlling the robot: “Manual” and “Automatic”.

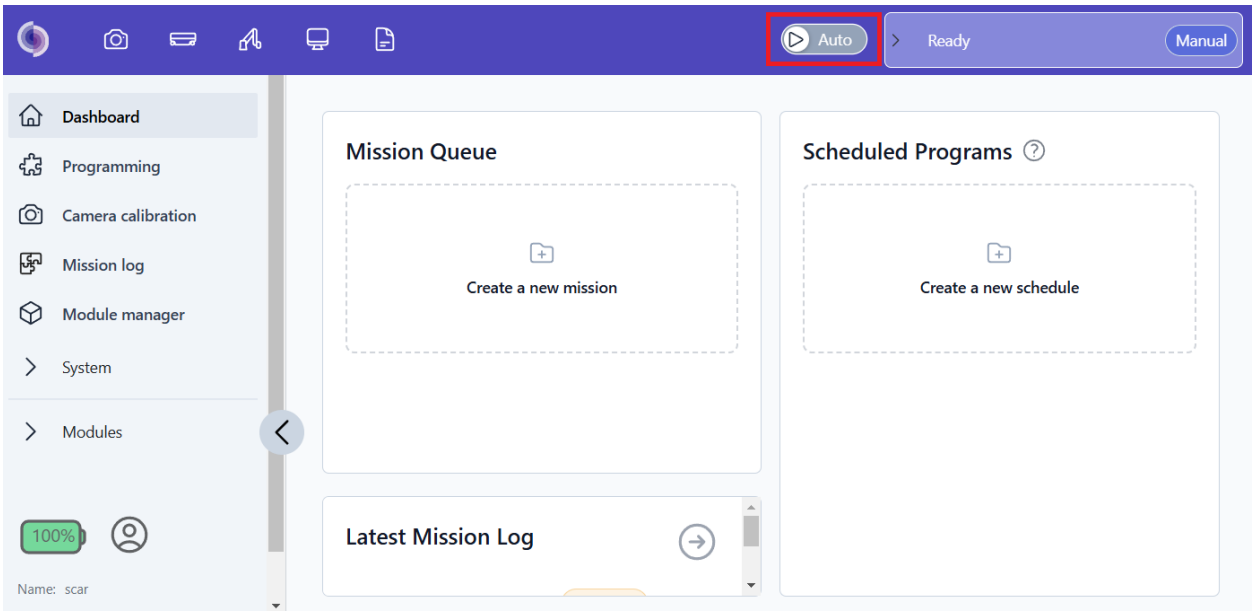


Fig. 1.1: The mode can be changed from the dashboard using the switch in the top bar.

Manual mode is used when setting up and programming the robot. It gives the user full control of the robot through the joysticks, as well as building programs through the “Programming” tab.

Automatic mode is used when the robot is to run automatically and enables the use of missions, such as when running in production.

Note: The mode selector in the Ability web interface is fully independent of the physical mode selector on the MiR. The mode selector on the MiR only enables/disables joystick control of the MiR. If the user wishes to joystick the MiR, both selectors should as such be in *manual* mode. In most cases when programming the robot the MiR selector will be in *automatic* and Ability in *manual*. When running in production or testing finished programs, both selector should be in *automatic*.

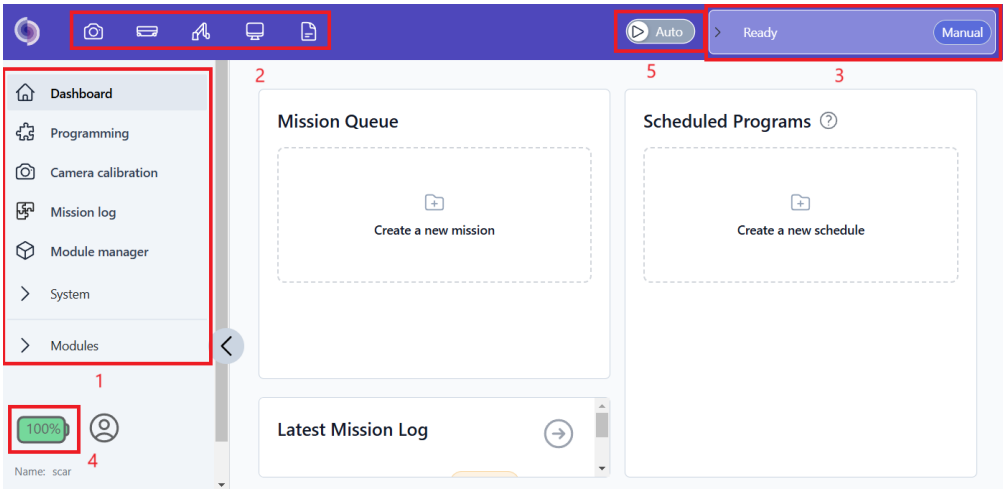
Controlling the robot in each mode is described in detail in the following sections.

1.1 Manual Mode

The manual mode is the default mode of the robot and is used when programming the robot, or controlling it using the joysticks.

1.1.1 Navigating the programming interface

When accessing the Ability programming interface, the first page shown is the dashboard. To the left of the dashboard is the side menu (1). Above, in the top bar is the toolbox (2), the system state (3) and the operation mode selector (5). In the lower left corner is the battery indicator (4). The side menu and top bar are visible from all pages of the web interface.



- **Side menu (1)** allows for navigating between the different pages of the web interface such as the programming view, camera calibration page and different system pages.
- **Toolbox (2)** contains different tools for controlling and interacting with the robot. The tools are described in detail in the following sections.
- **System state (3)** shows the current state of the robot, wether it is in automatic or manual and what program is currently running on the robot.
- **Battery indicator (4)** shows the current battery level of the robot. Clicking the indicator allows for sending the robot to any of the chargers defined on the internal map.
- **Operation mode selector** switches the operation mode of the robot between manual and automatic.

The contents of the dashboard itself are described in the section *Automatic Mode*.

1.1.1.1 The camera view

Pressing the camera view allows the user to get a camera view from the onboard camera. It functions as an overlay of the user interface and can therefore be kept open when e.g. programming the robot.

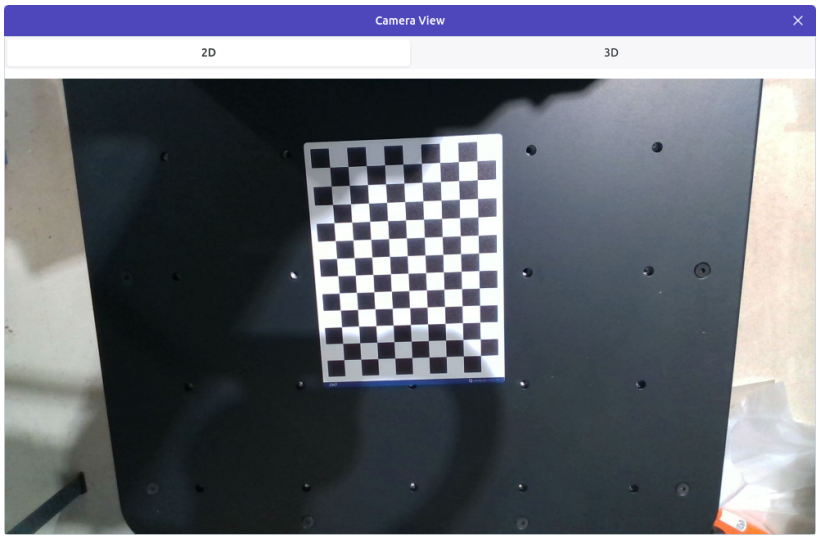


Fig. 1.2: Camera view

If the Vision 3D module is enabled, it is possible to inspect a point cloud of the current scene. The visualization includes the coordinate system of the current camera-location, but it is possible to select any available frame or tool.

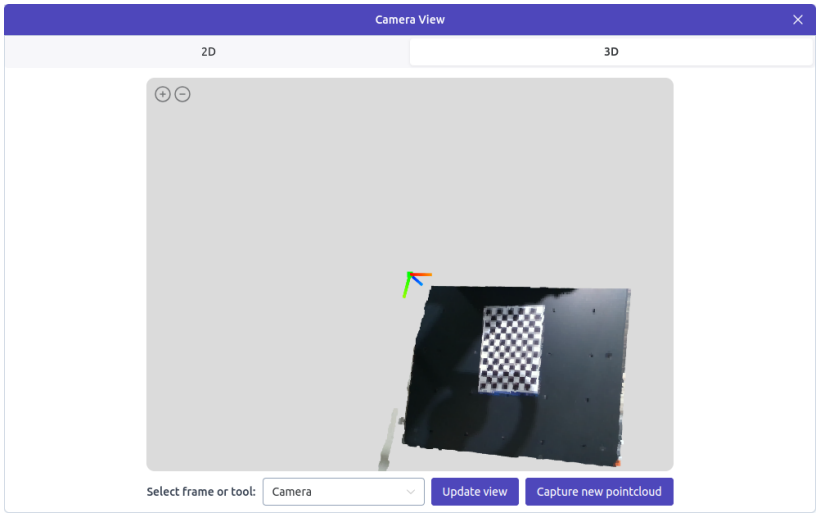


Fig. 1.3: Pointcloud view

1.1.1.2 The joystick control

Warning:

As the entire robot is moved with the joystick, make sure that no objects or humans are close to the robot before using this feature

 Fig. 1.4: MiR joystick button	 Fig. 1.5: MiR joystick button when MiR is blocked	 Fig. 1.6: UR joystick button	 Fig. 1.7: UR joystick button when arm is not in safe home

Pressing the mobile joystick (Fig. 1.4) in the top bar, allows the user to control the MiR robot with an on-screen joystick. The joystick can be activated when the user has claimed the robot token and the MiR is in manual mode. If the MiR joystick icon looks like Fig. 1.5, it means that the MiR is blocked and cannot be used. This can be resolved by moving the UR to a safe home position or disabling the MiR blockage feature in settings.

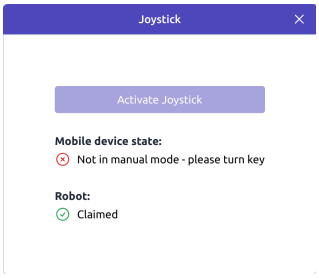


Fig. 1.8: Joystick for manually moving the MiR

Pressing the manipulator joystick (Fig. 1.6) allows the user to control the robot arm. The initial view is shown in Fig. 1.9. If a safe home position is configured on the UR teach pendant, it is possible to send the arm to the safe home position by clicking and holding down the “Move to safe home” button. If the button is released before the robot reaches its safe home position, the movement will stop immediately. The manipulator joystick icon looks like Fig. 1.7 whenever the UR is not in a safe home position.

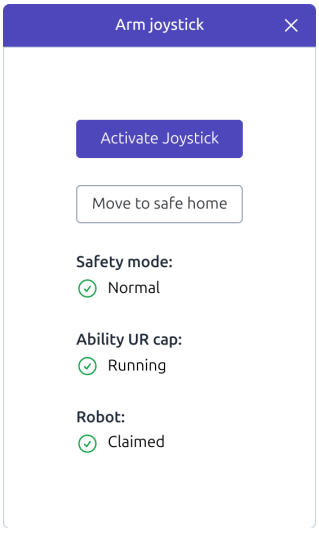


Fig. 1.9: Initial view when opening the manipulator joystick

Pressing “Activate Joystick” opens a view with three (two for the ER-MAX robot) modes of arm control. Default, shown in Fig. 1.10, enables cartesian movements of the tool. Pressing the middle button to the left enables moving the individual joints. Finally, holding the bottom button down for 2 seconds puts the robot into teach mode for 100 seconds, or until the button is pressed again.

Note: Teach mode is not available on the ER-MAX robot. To freedrive the ER-MAX consult section 4.4.2 of the user manual that was provided with the robot.

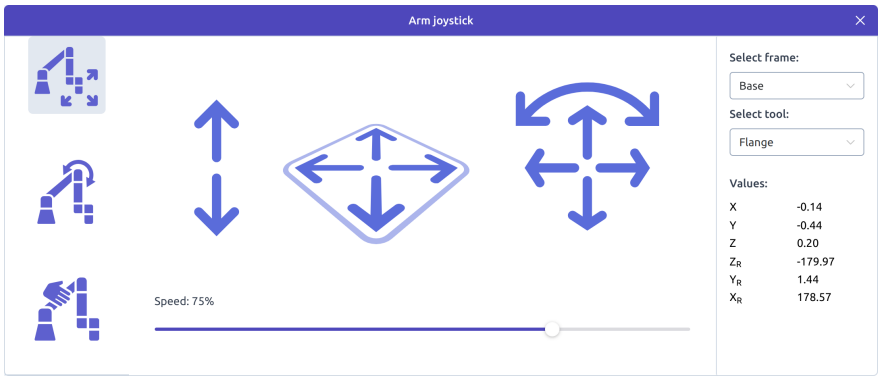


Fig. 1.10: Joystick for manually moving the UR

Whenever the teach mode is enabled all axes are free by default. It is possible to constrain the tool movement to specified axes. An example is shown in Fig. 1.11 where the Flange is constrained to translational movement in the Base frame.

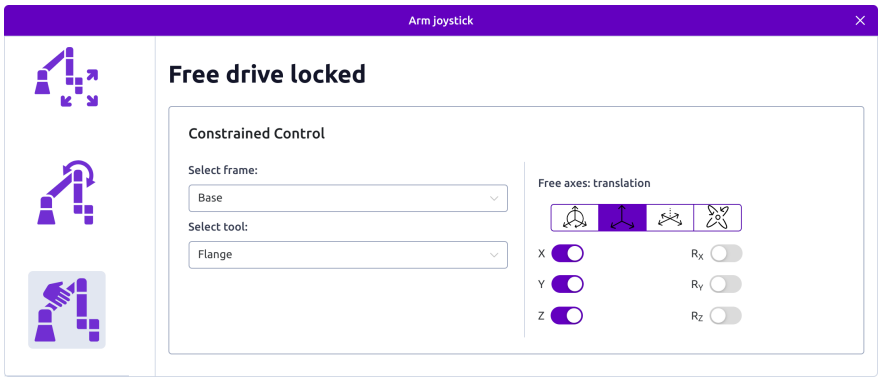


Fig. 1.11: Constrained free drive control

1.1.1.3 UR teach pendant view

The UR teach pendant view ( ) gives access to the content of the UR teach pendant within the Ability interface. This functionality is intended for giving experts easy access to configuring the arm and to define custom subroutines (events) within the UR Polyscope interface, which can then be activated by inserting a block in the Ability programming interface.

The UR teach pendant view is shown in Fig. 1.12.

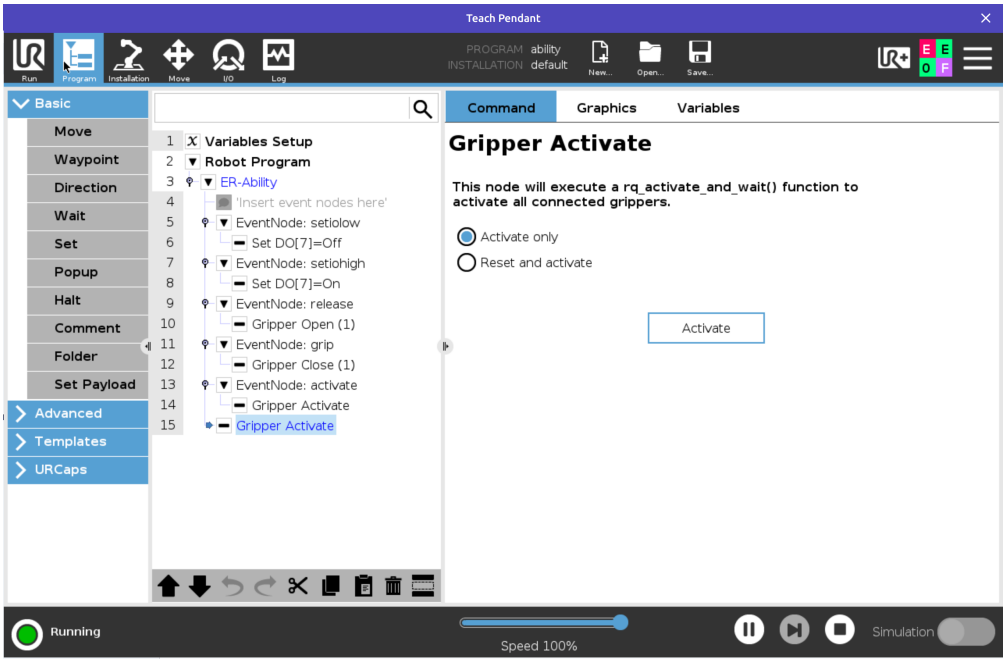


Fig. 1.12: UR Teach pendant view

1.1.1.4 Runtime Info

The current values of references and variables can be inspected through the 'Runtime Info modal'. Whenever a program is loaded the references are reset and all variables are cleared.

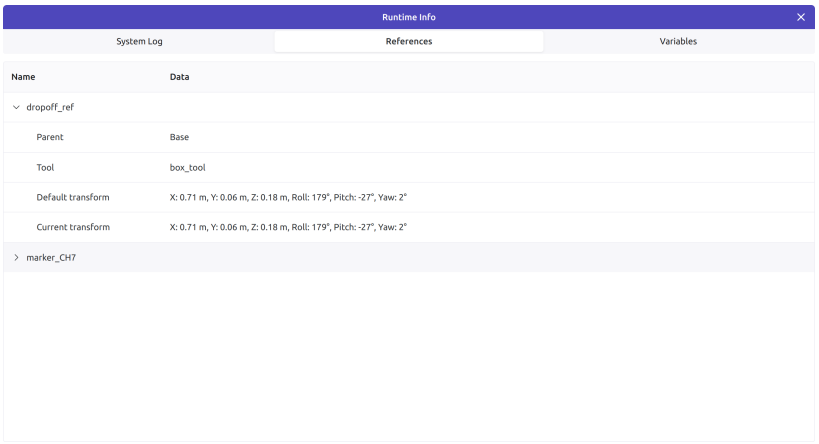


Fig. 1.13: Reference info tab

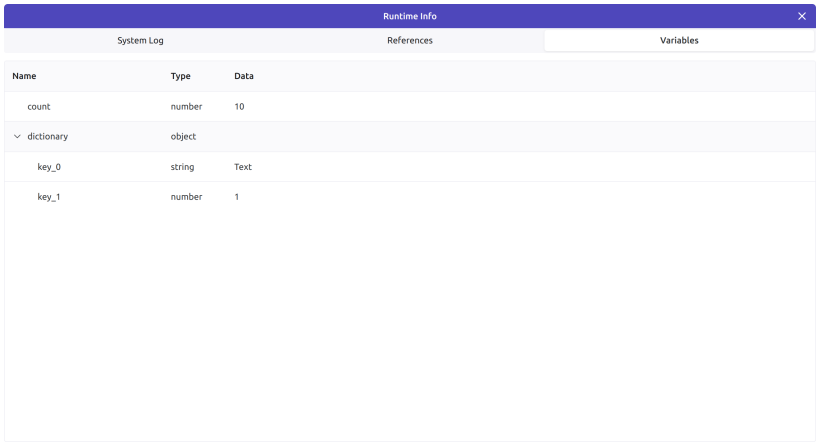


Fig. 1.14: Variable info tab

1.1.2 Programming Dashboard

The programming dashboard is found by going to the “Programming” item in the side menu. It gives an overview of all available programs.

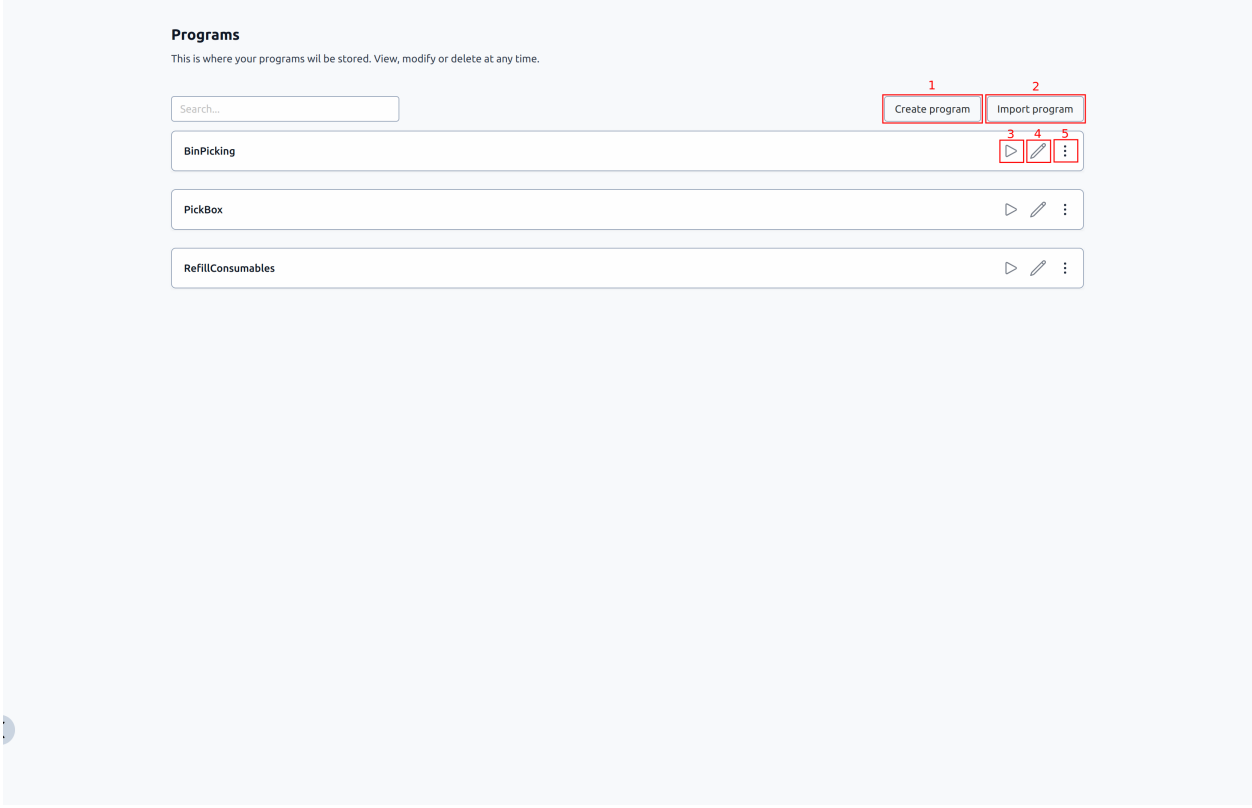


Fig. 1.15: Programming Dashboard

From the programming dashboard it is possible to create a new program (1), import a program from disk (2), play a program (3), edit a program (4), download a program and delete a program (5).

1.1.2.1 Import and export of programs

Pressing the export button (5) will download a compressed '.tar.gz' archive containing all data related to the program.

It may also contain other files or folders if certain blocks are used.

Pressing the "Import a program" button (2), lets the user choose a program (packed in a '.tar.gz' archive) from their device storage. Once chosen, the program is uploaded and validated. The name of the imported program will be equal to the folder-name inside the archive. If another program with the same name already exists, a pop-up will ask the user whether to overwrite the existing program or cancel the import.

When importing a program that has been created on another robot, please make sure that the configurations of the manipulators and mobile devices are similar. If the program makes use of any user defined entities such as custom tools or markers, the setup-file needs to be imported as well. This is done from the Setup-page under 'System'.

1.1.3 Programming basics

When a program is created, the programming workspace will be shown.

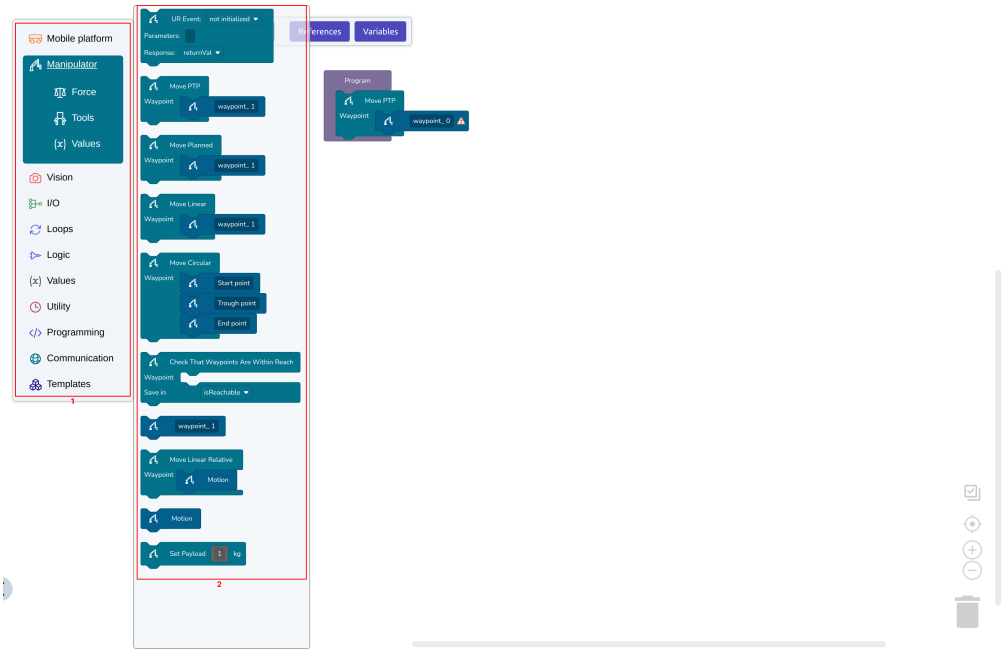


Fig. 1.16: Overview of the programming workspace. (1): Block categories. (2): Blocks

A program always contains the 'Program' block, into which further blocks can be inserted. Only blocks added inside of the 'Program' block will be executed. The only exceptions are the 'Function' and 'Error function' blocks that can be found under the *Programming* block category. Code inside a 'Function' block will be executed when a corresponding 'Call Function' block has been added in the main programming block. Similarly, code inside an 'Error function' block can be executed by a related block if it encounters an error.

1. Insert instructions. The different blocks are categorized and placed in different submenus. The blocks are color coded based on the category. For example, orange blocks refer to the *Mobile Platform*, while blue to the *Manipulator* and purple to *Programming* structures.

2. Configuring blocks. When a block has been inserted it can be configured. Click the block and a menu will appear to the right with the different settings available on the block. This is called the 'block menu'. For configuration, positions etc. the block menu will contain a button for reading in the current values of the robot. Remember to press 'Apply' to store the result before leaving the menu.
3. Click 'Save' to save the program.
4. Click the 'Play' button to execute the program.

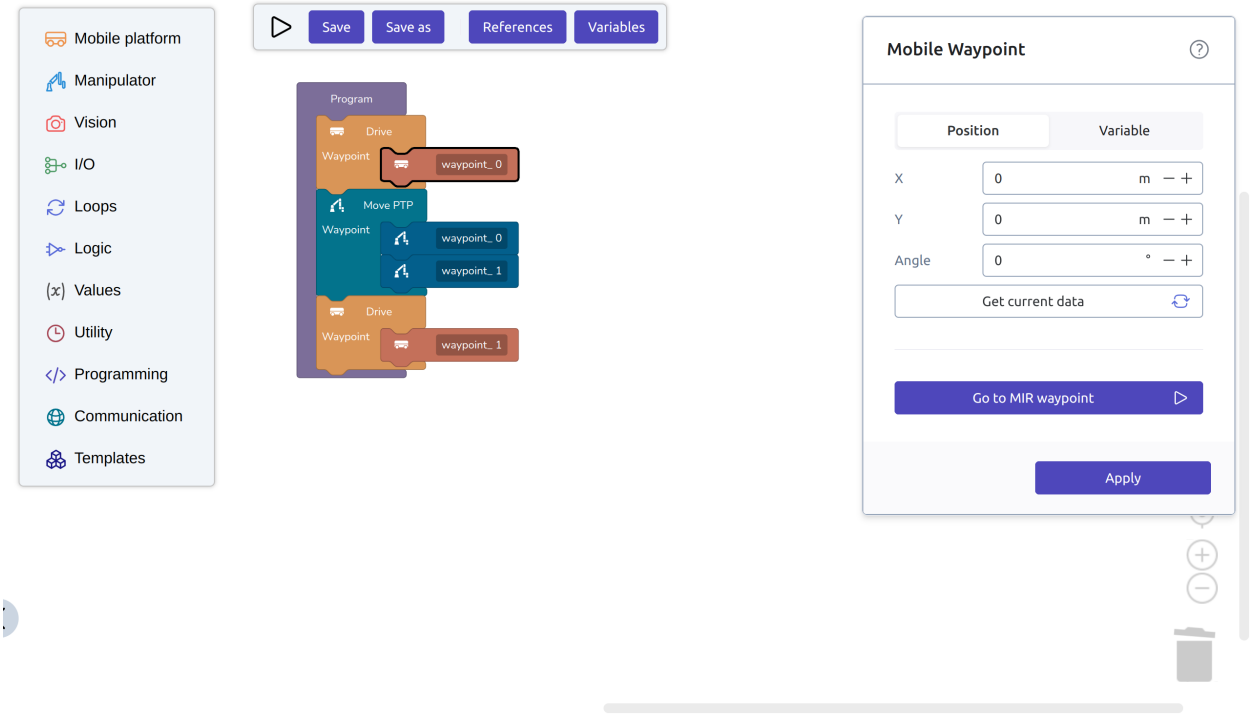


Fig. 1.17: Ability sample program

Danger:

When driving with the MiR always ensure that the UR and its tool is completely inside the footprint of the MiR. Failure to do so may cause hazardous situations as the MiR is unaware of the configuration of the UR and tool while driving.

Warning:

When using the vision system, movements of the arm may change based on the results of the vision system. It is therefore important to ensure that the robot has enough clearance to obstacles to avoid collision.

1.2 Automatic Mode

Enabling automatic mode starts the mission queue and will keep running missions until the queue is empty. The dashboard page offers functionality to control the robot's mode, add missions to the queue, and schedule programs.

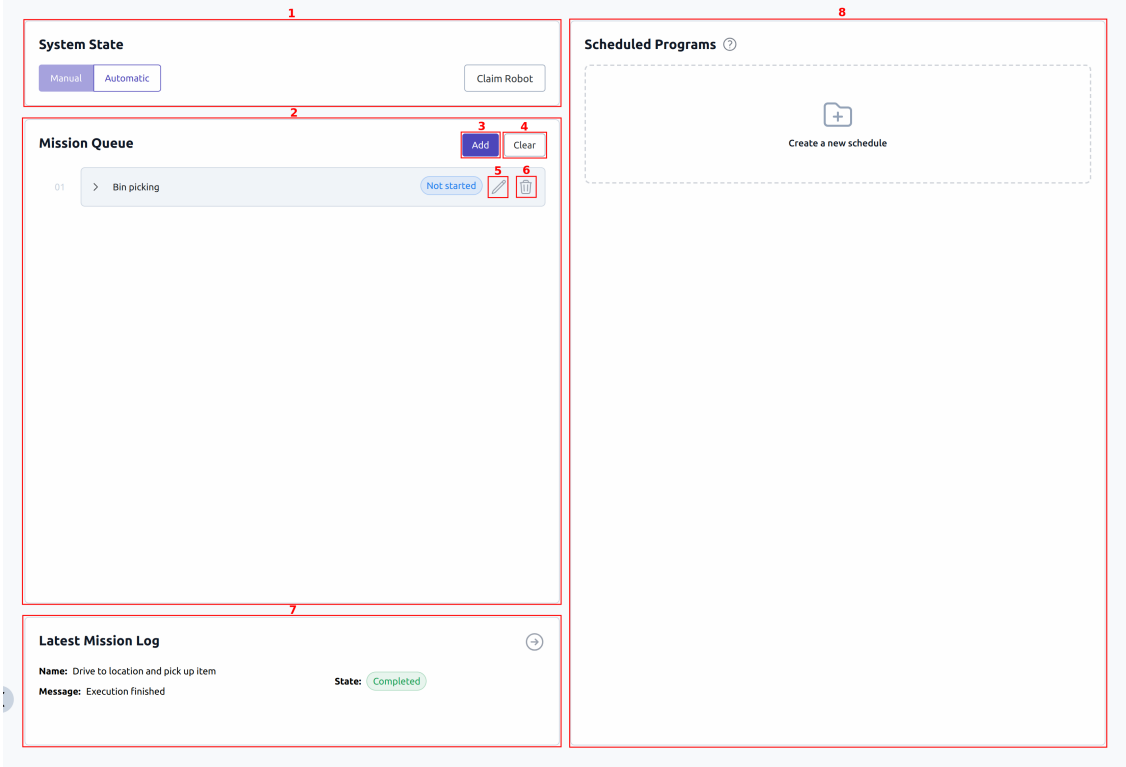


Fig. 1.18: Dashboard

- The 'System State' provides a visual representation of the robot's current mode and allows users to switch between modes using the switch (1).
- The 'Mission Queue' offers a comprehensive view of queued missions awaiting execution (2). Users can add new missions to the queue (3), clear the entire queue (4), and modify (5) or delete (6) individual missions. When the robot is in automatic mode, the queued missions are executed sequentially. Each mission may consist of multiple programs, allowing a series of programs to run in a predefined order.
- The 'Latest Mission Log' maintains a record of the last executed or canceled mission from the queue (7).
- The 'Scheduled Programs' provides an overview of upcoming scheduled programs (8) and enables users to schedule additional programs to be added to the mission queue. This feature enhances the planning and automation capabilities of the robot.

1.2.1 Mission builder

The mission builder allows users to create missions complete with programs and associated arguments, seamlessly adding them to the queue. Accessing the mission builder can be done by adding a new mission to the queue.

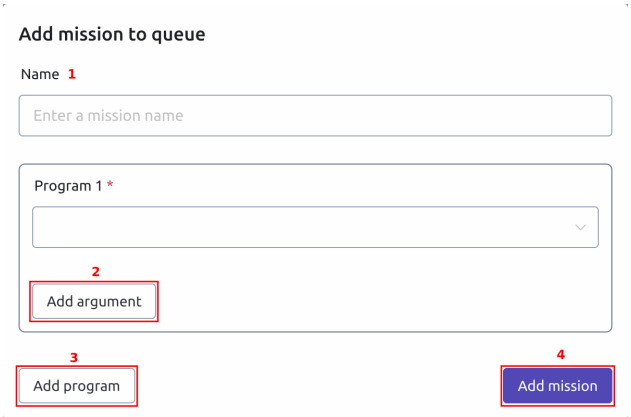


Fig. 1.19: Mission Builder

The mission can be given a name for identification (1), add as many programs as needed (3) and arguments to each program (2). By pressing the 'Add mission' button (4), the mission is add to the queue. The mission is executed if the robot is in automatic mode and the mission is next in line.

Although the mission builder is a great tool for testing, most often, missions are queued using the *REST Interface V2*. To learn more about how to use program arguments inside a program see *Program arguments*.

1.2.2 Scheduled Program

The program schedualer can be used if a program is to be executed at a spefic time in the future.

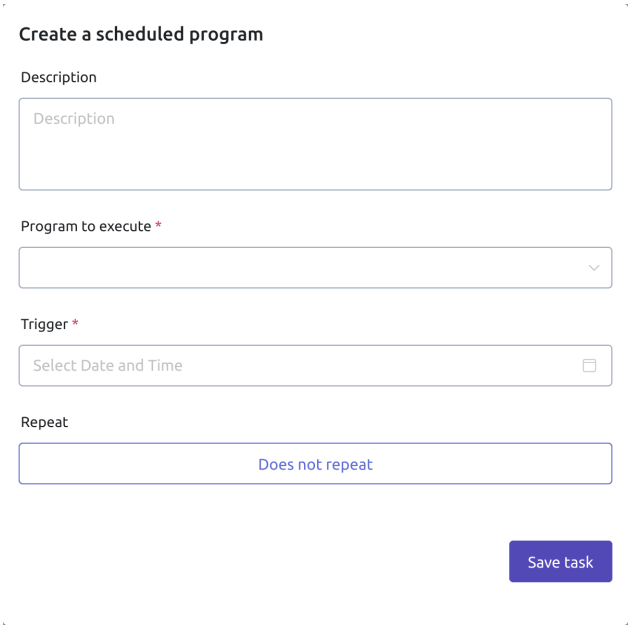


Fig. 1.20: Scheduled Program

When adding a new scheduled program, the configuration includes options to provide a description, specify the program for execution, and define a trigger. The trigger dictates the scheduling of the program, with an added repeat functionality to determine its frequency of schedulation. Upon the scheduled time, the program is added to the queue as part of a mission and executed when the robot is in automatic mode and the mission is next in line.

CHAPTER

TWO

HOW TO BUILD A PROGRAM

This section includes advice and explanations on how to build efficient and stable Ability programs. Part of the guide will be useful for people that have never built a program on the robot before, but the addition of tips and “best practices” makes the guide valuable to more experienced users as well. All the blocks mentioned and used in this guide have their own descriptive guides that can be found in the *Blocks* section of the Ability user documentation.

The guide is centered around building a single program. During the guide there will be suggestions, warnings and small tips that could be helpful when building a large program.

2.1 Introduction

Programming a mobile cobot like the ER-FLEX can be a complicated task. The combination of mobility and manipulation requires extra attention when building a program, but with the right tools it can be relativly straight forward. The goal of the Ability software is to provide such a tool.

The complexity of mobile cobots comes from the non-perfect accuracy of the mobile platform of the robot. If we wish to manipulate an object at a certain location, we might need an accuracy in the millimeter range, but if the robot is only able to position itself within a few centimeters, we are out of luck. That is, without a calibration/vision system.

The ER-FLEX comes with a built-in vision system that allows it to calibrate to local reference frames to gain millimeter accuracy for manipulating objects. This calibration step is one of the most important parts of every program.

In general, most mobile cobot applications have the same overall flow in terms of programming logic:

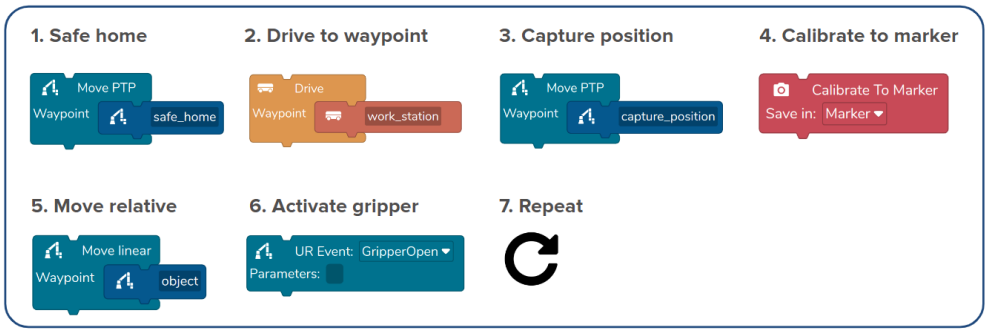


Fig. 2.1: The general program flow of a mobile cobot application

1. **Move the arm into a safe home configuration**

To avoid colliding with objects when driving, it is important for the arm to be in a safe configuration inside the footprint of the mobile platform. On newer hardware versions of the ER-FLEX, the safe home configuration is enforced through the safety PLC. This means the robot is only able to drive when the arm is in the specific configuration. The configuration can be changed in the URs interface.

2. **Drive to a pre-defined position on the map**

   During commissioning, an internal map is recorded on the robot by driving it around. When the map has been recorded, the robot can autonomously navigate between positions defined on the map.

3. **Move the arm into a capture position where the vision system can see the checkerboard marker**

   After arriving at the pre-defined position, the arm is moved into position to look for a checkboard marker. The versatility of the vision system means that the capture position does not have to be perfect. As long as the marker is within the camera view and inside a range of ~ 1 meter, the marker should be able to be detected.

4. **Calibrate to the marker to obtain a local reference frame.**

   The vision algorithm of the robot detects the marker, and calculates its position with high accuracy.

5. **Move the manipulator relative to the local reference frame**

   With a local reference frame obtained, the manipulator can now do relative movements to do different kinds of tasks: a. Load objects on/off the robot e.g. for intralogistics. b. Manipulate objects at the location e.g. for machine tending or product assembly.

6. **Activate the end-of-arm tool to manipulate object**

   At some point the robot will need to activate the end-of-arm tool to grasp/release the object.

7. **When manipulation of the objects is finished, return to #1 and repeat the process.**

Although this is the general flow of programming, one will often find that a program will contain a lot more logic than just these 7 steps. For instance, one might want the robot to wait for some external input before starting its task, and probably also report back when the task is done. Most programs will also need to implement error handling to make sure robot operation will contine in the event of an unknown scenario or error. Maybe the robot will need to interact with external machines to start and stop processes automatically.

> **Warning:** An individual risk assessment should always be done for every application involving the robot to ensure that local safety standards are met.

2.2 Before building a program

Before building a program there are a few setup steps that need to be done.

2.2.1 Map

Building a good map of the environment is important to the navigation of the mobile platform. Mapping is done automatically as described in the ER-FLEX manual. The automatically generated map could be used without further processing, but going through a few extra steps of editing the map can improve both navigation efficiency and safety e.g:

- Removing noise by deleting dynamic objects from the map and making sure the map corresponds to the real world static environment
- Adding forbidden, preferred, speed and critical zones

- Filling out “missing spots” by manually drawing in walls or floors
- Setting up charging stations and positions

A full guide on best practices for building maps can be found on MiR Academy. For more information on how to edit maps please refer to the MiR User Guide

---

Note: Even a small bit of noise on the map e.g. a wall 1 pixel wide, will cause the navigation system to navigate around it.

2.2.2 Safe home configuration

The safe home configuration should ensure that the UR, including any tool, is fully within the footprint of the robot. On certain hardware verions of the ER-FLEX, the safety configuration is setup such, that when the UR leaves the defined configuration, the MiRs brakes will be triggered.

The safe home configuration is setup through the teach-pendant view:

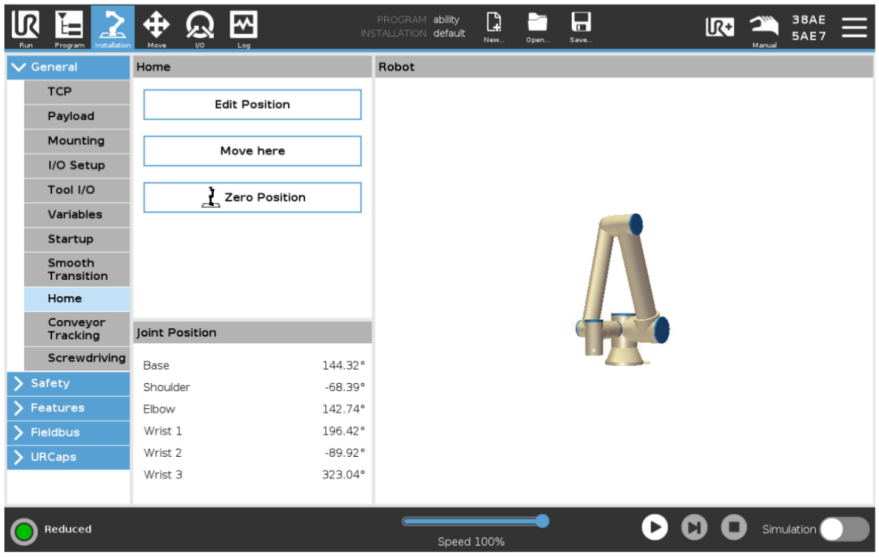


Fig. 2.2: Defining the home configuration is done in the URs interface under “Installation”

1. Define a general Home position by opening the teach-pendant view and go to Installation -> General -> Home. Press “Edit position” and teach in the desired position.
2. To sync the Home position to the Safe home position, go to Safety -> Safe home.
3. Enter the safety password (default is “enabled”), press “Sync from Home” and then apply.
4. After restarting the robot, a prompt will appear saying that the program needs to be confirmed with the new safety settings. To do this, reload the program by pressing “Open…” in the top bar and selecting the “Ability” program. Then save the program again by going to “Save…” and selecting “Save all”.

Note: Preferably, the safe home configuration should keep the CoG as low as possible

---

After setting up the safe home configuration on the UR, the current safe home state is now shown on the manipulator joystick button.

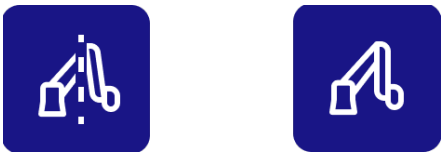


Fig. 2.3: The dashed line going through the manipulator joystick button shows that the arm is **not** currently in safe home.

Sending the robot to the safe home configuration can be done via the joystick.

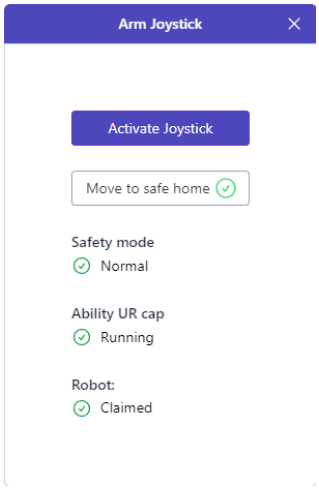


Fig. 2.4: Holding down the “Move to safe home” button will send the arm into the configuration defined on the UR teach-pendant.

2.2.3 Tools

When adding a tool to the robot like a gripper, defining the Tool Center Point (TCP), can be very useful. Defining a tool and its TCP makes it possible to make movements relative to the TCP rather than the flange of the robot. A *Tool* exists as a static configuration across different programs, this makes it easy to reuse. Tools are defined in *SYSTEM -> SETUP* using the *Tool* Setup Entity.

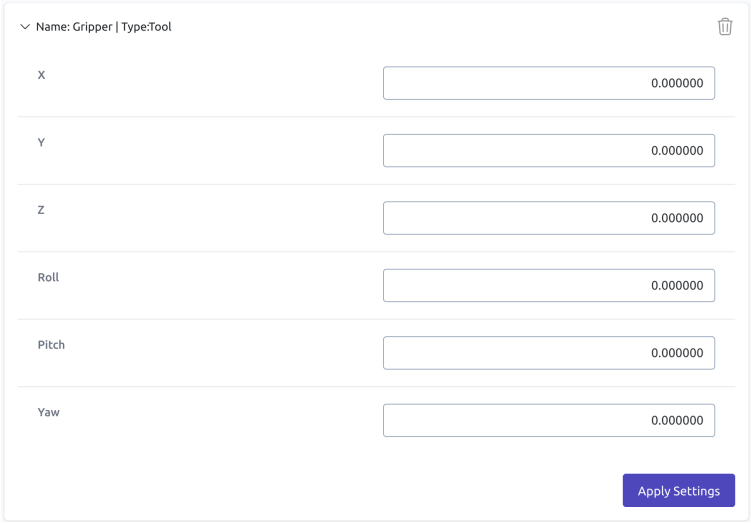


Fig. 2.5: The *Tool* Setup Entity is used to add tool configurations to the robot.

Note: Adding a Tool Setup Entity with the right TCP, while not strictly necessary, will make programming more intuitive when the tool has to rotate around another reference.

Finding the TCP of a tool can be achieved several ways. Often, the manufacturer of tools will provide the TCP in their documentation. For custom tools, the TCP can either be measured using a calliper or by using the UR's tool wizard. The tool wizard is found *Installation -> General -> TCP* on the teach-pendant. Please refer to the UR Manual for a guide on using the wizard.

Important: The UR tool wizard will give the position of the TCP in **millimeters [mm]** and **radians**. When adding the *Tool* Setup Entity on the *Setup* page the values should be converted to **meters [m]** and **radians**

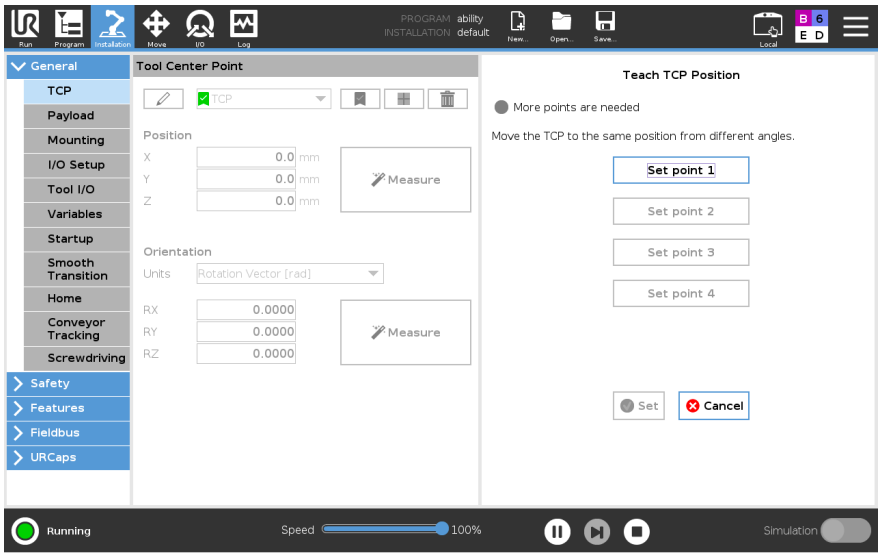


Fig. 2.6: The UR's tool wizard can be accessed using the teach-pendant

Note: Another advantage to using tool definitons is, that it makes handling physical modifications to the tool easy. Simply adjust the settings for the tool in the setup and all existing movements defined with the tool will be adjusted automatically.

2.2.4 Payload

The payload of the robot refers to the load mounted on the flange of the manipulator. A payload is defined by a combination of the Center of Gravity (CoG) being the distance in x,y,z from the flange as well as the size of the mass (in kg). Before building a program, an initial payload should be set on the UR corresponding to the tool mounted on the robot at the start of the program. Setting the payload is done by going to *Installation -> General -> Payload* on the teach-pendant. Finding the payload can also be done from here using the UR's payload wizard.

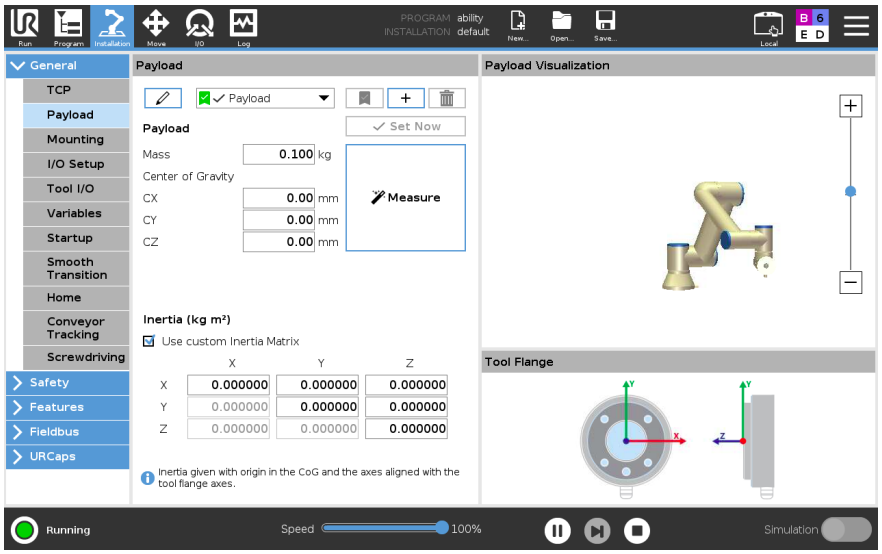


Fig. 2.7: The UR's payload page also provides a wizard for finding the current payload.

Important: In most programs the payload will change during the execution of the program e.g. when picking up objects. When picking up objects heavier than 1-2kg it is important to use the *Set Payload* block to overwrite the default payload set on the UR.

2.2.5 UR Events

Another recommended setup is adding UR Events relevant to the application on the teach-pendant. Common events often involve activating 3rd party URCaps e.g. for controlling a gripper. In the ER-Ability URCap it is possible to define event nodes, which are subprograms for the UR created in the UR Polyscope interface. The definition of an event node in the UR Polyscope interface can be seen in the example below. These event nodes can be called from the Ability user interface using the *UR Event* block.

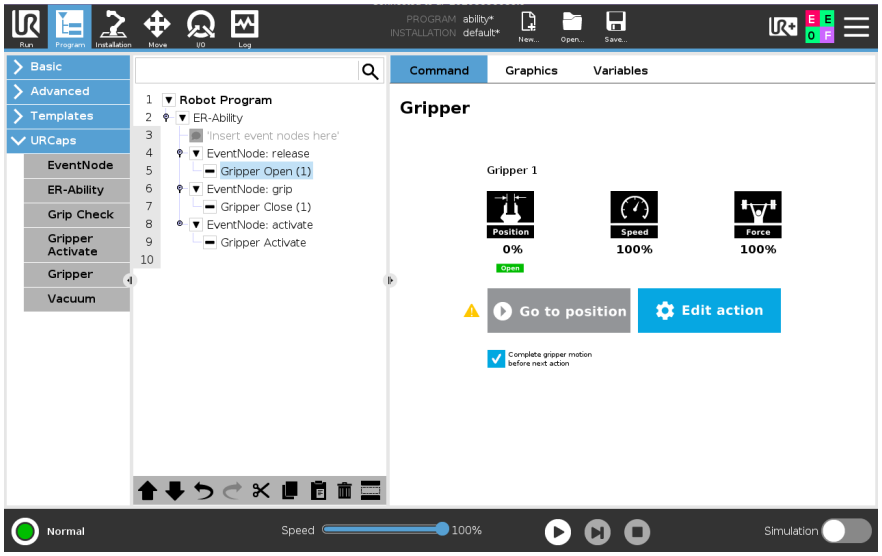


Fig. 2.8: Example of UR Events being used to activate, open and close a gripper.

Note: Make sure to save the UR program after defining events.

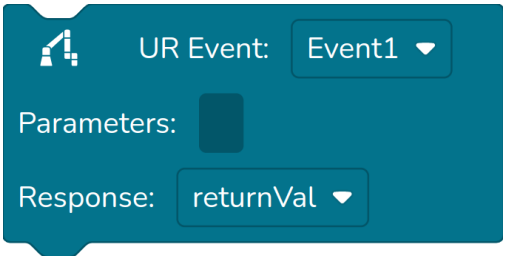


Fig. 2.9: UR Events defined on the teach-pendant will show up on the 'UR Event' block

When the desired UR Events have been setup on the UR it may be beneficial to save the program under a name that relates to the program instead of overwriting the default ability.urp program. It is possible to change the default UR program the robot runs on the *Setup* page.

2.2.6 Camera recalibration

Although the camera comes pre-calibrated from the factory, it might need to be recalibrated. This can be due to one of several reasons:

- The camera fixture has been detached from the flange while mounting a tool to the robot, and therefore may have slightly shifted in position/orientation when reattached.
- Although unlikely, the camera fixture might have been bumped during transport or unboxing of the robot causing the fixture to move or bend.
- Mounting the camera in a different position and orientation relative to the flange than it was from the factory.

In general, the camera needs to be recalibrated everytime it is detached from the flange, even if it is reattahced in the same position/orientation.

How to calibrate the camera is described in the *Camera calibration* section.

2.3 Building a simple program

2.3.1 The 'Program' block

The first block of a program is the 'Program' block. This is the root of any program. When running a program, all blocks defined inside this block will be executed. In conventional programming languages this would often be referred to as the 'main function'

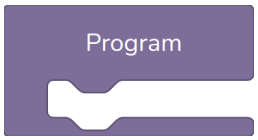


Fig. 2.10: The 'Program' block is the basis of all programs.

2.3.2 Moving the manipulator

There are two ways of moving the manipulator: *Move PTP and Move Linear*. Point To Point (PTP) movements will move the manipulator the shortest distance in joint-space. This is the most efficient kind of movement, as the joints have to travel the least distance. It often results in the tool moving in a curved trajectory. Linear movements will move the shortest distance in cartesian space, e.g. a straight line. This is essential when picking up or moving along objects.

Tip: When linear motion of the tool is not strictly necessary, it is recommended to use PTP movements.

Both the *Move PTP and Move Linear* blocks use the same type of *Manipulator Waypoint* block. A waypoint can be configured as either:

1. A joint configuration
2. A tool position relative to a reference
3. A variable (which can be either of the 2 above)

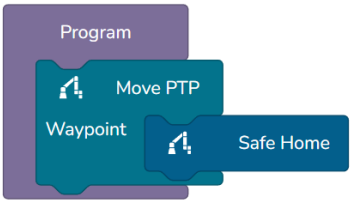
Joint configurations are important for making sure that each joint is in a specific configuration. Joint configurations are fully deterministic, this is not the case with tool positions as there can be several joint configurations for each tool position.

Tool positions are used for movements relative to a reference. E.g. when picking up objects or moving relative to a marker.

To begin any program, the arm must be moved into the *Safe home configuration* defined during setup of the robot.

Danger: If the manipulator including any tools is not entirely within the footprint of the mobile platform it might cause hazardous situations when the mobile platform is navigating the environment.

Start by adding a “Move” block to the program. Replace the *Manipulator Waypoint* inside with the *Safe Home* waypoint found in the manipulator category.



2.3.3 Driving

Driving to a location can be accomplished with three different blocks: *Drive to Waypoint*, Drive to position and by executing a MiR mission. The simplest and easiest to use is the *Drive to Waypoint* block. Simply use the mobile joystick to move the robot to the position of interest, and save the position in a *Mobile Waypoint* block.

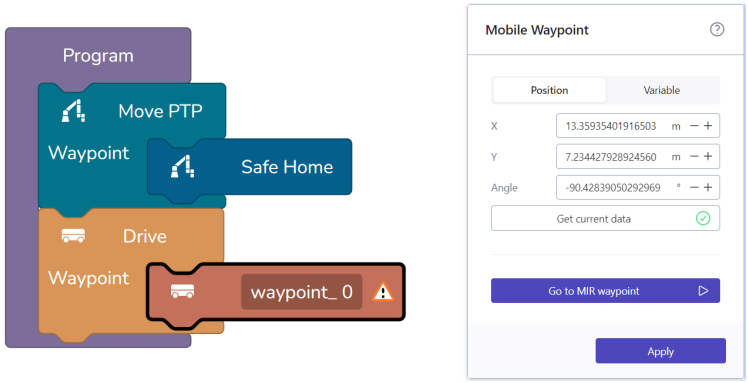


Fig. 2.11: On the *Mobile Waypoint* block, press “Get current data” to get the current position coordinates of the robot.

Tip: Using the ‘Position’ tab in the *Mobile Waypoint* block is useful if the robot is supposed to go to the position once in the entire loop of a program. If the same position is going to be used multiple times in different functions, consider adding a variable with the position that is initialized at the start of the program.

The Drive to position and MiR mission blocks can be useful if the user is already familiar with programming a MiR robot.

2.3.4 Vision

When arriving at a position with the robot, the next step is to find a local reference frame. This will allow the robot to make accurate manipulator movements in relation to said frame, e.g. a box, table, machine or workcell. There are multiple ways of finding local reference frames, the most common is to use the *Calibrate to Marker* block. This block is able to quickly and accurately find a reference frame using the Enabled Robotics chessboard *Markers*.

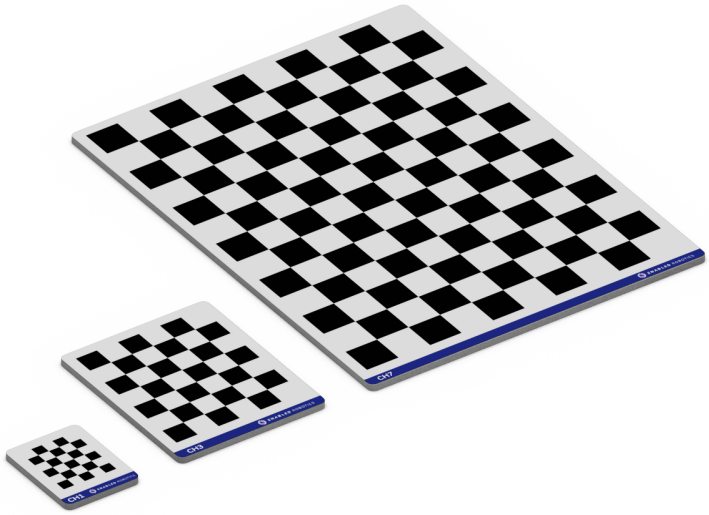


Fig. 2.12: The three types of standard markers available for the ER-FLEX: CH1, CH3 and CH7.

Before calibrating to a marker, the manipulator must first be moved into a capture position where the marker is within view of the camera. This is usually achieved by moving to a static joint configuration.

Note: When using joint angles, it dosen't matter which tool is selected on the "Move" block as the manipulator will always be moved to the same configuration independent of the tool.

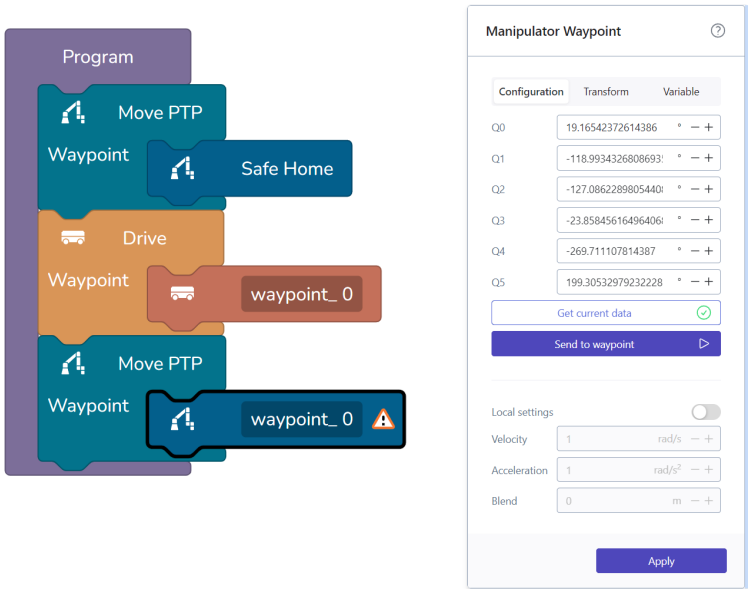


Fig. 2.13: In the "Configuration" tab, pressing "Get current data" will read the current angle of each joint and save it in the waypoint.

With the camera now pointing at the marker, a *Calibrate to Marker* block can be initialized.

To initialize a marker reference, go to the *Calibrate to Marker* block, click the "not initialized" dropdown and select "New reference".

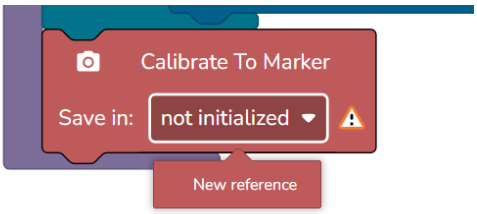


Fig. 2.14: When selecting "New reference" from the *Calibrate to Marker* block, the reference menu will appear in "marker mode", allowing for simple initialization of a marker reference.

From the reference menu, select the type of marker used in the "Marker" dropdown and press "Detect and create".

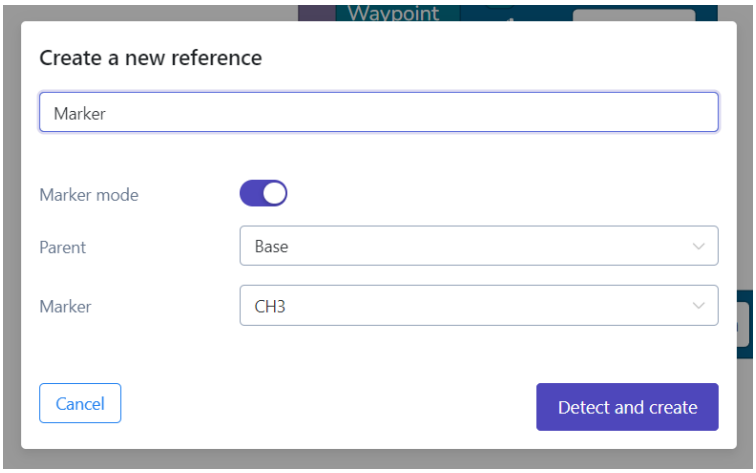
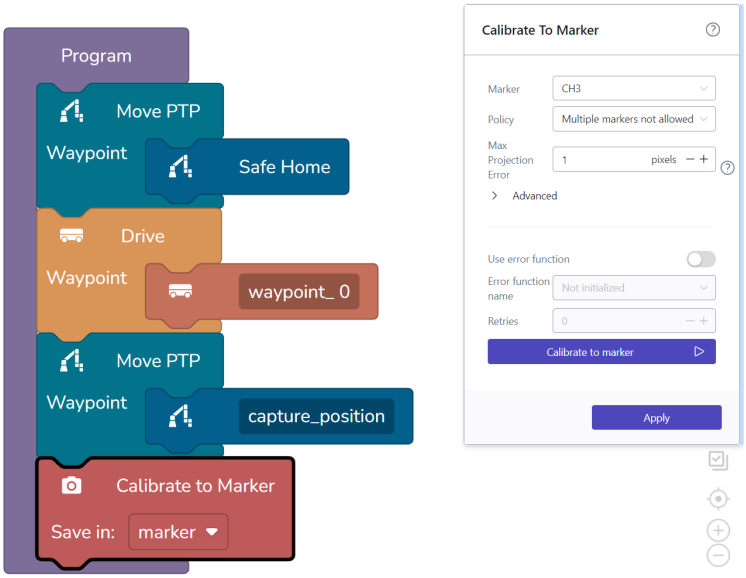


Fig. 2.15: When the reference menu is in marker mode, the only options are “Parent” and “Marker”. The parent is normally set to “Base” to ensure high accuracy when moving relative to the reference.

The reference will now be initialized and can be used in the *Calibrate to Marker* block. Make sure to select it in the dropdown on the block itself. Lastly, select the correct type of marker and policy in the block menu to the right.



2.3.5 Moving the manipulator relative to a reference

After obtaining a reference, manipulator movements can now be defined relative to it. To define relative movements, add a Move block to the program and select the desired tool to use for the relative movements.

Adding tools is done through the *Setup* menu as described in the previous section: *Before building a program*.

The transformations (translation and rotation) defined in the *Manipulator Waypoint* block, will be between the tool selected in the parent move block, and the reference selected in the waypoint menu.

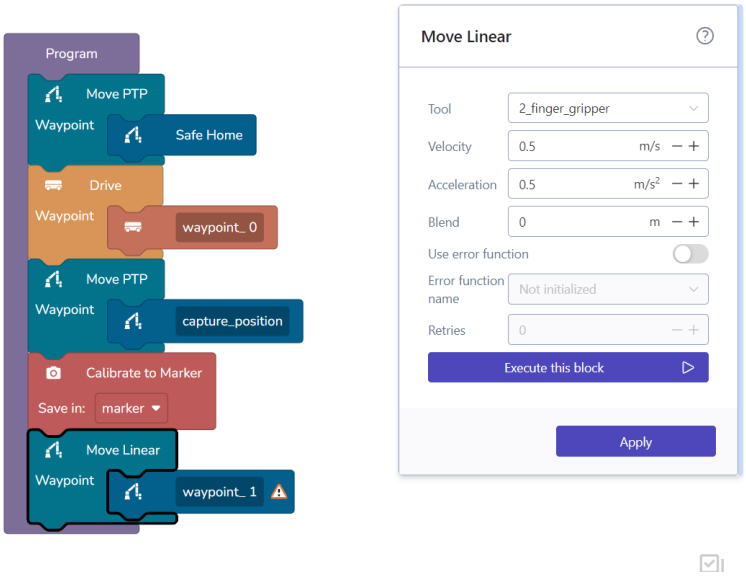


Fig. 2.16: Selecting the correct tool when making relative moves makes the transformations defined in the waypoint block more intuitive.

Tip: Using tool definitions not only makes the relative transformations more understandable, but also makes it easy to make changes to the phyiscal tool without breaking the robot program. It is simply a matter of updating the parameters on the *Tool*.

To set the relative transformation, first use the joystick to move the tool into *roughly* the desired end position. The position does not need to be perfect as it will be fine-tuned by manually setting the transformation in the block menu. After moving the tool into position, go to the *Manipulator Waypoint* block and select the Transform menu. From the reference dropdown menu, select the "Marker" reference created earlier and press the "Get current data" button to get the current transformation between the marker and the tool.

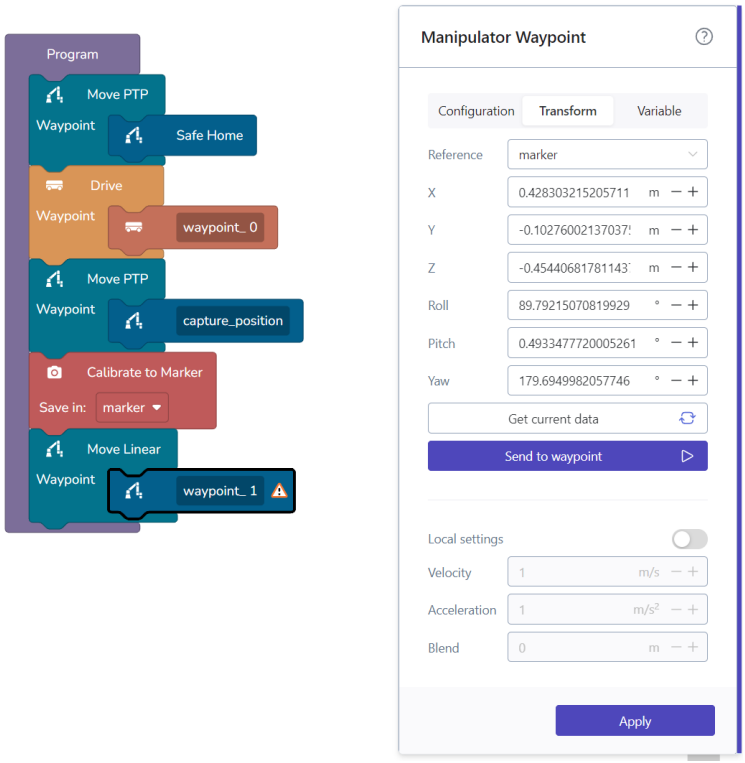


Fig. 2.17: While the desired transformation between the marker and tool can be put in directly, it can help getting the current transformation to better understand how the tool is currently positioned relative to the reference. The transformation can then be adjusted.

To fine-tune the transformation, it is important to understand the local reference frame of the marker used as seen below.

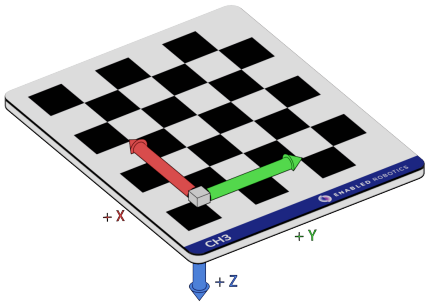
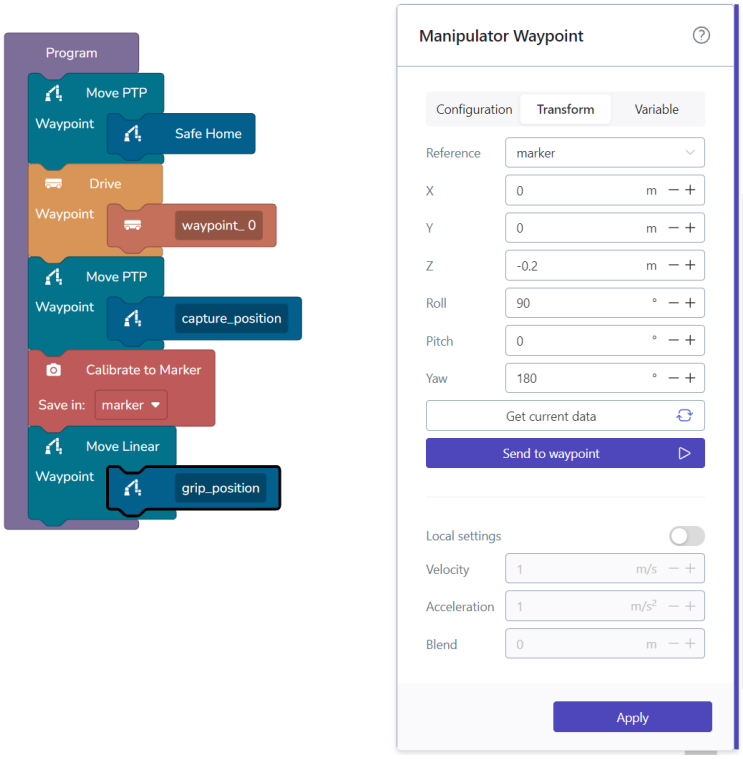


Fig. 2.18: Understanding the reference frame of the marker is important when defining the relative tool position. For all Enabled Robotics markers the z-axis points through the marker.

With the current transformation between the marker and the tool, it is simply a matter of adjusting each

parameter to fit the desired end position. In the example below, the tool is moved 20 cm away from the zero point of the marker.

Note: The z-axis of the Enabled Robotics markers points "through" the marker. When approaching the marker, the marker to tool transformation should therefore have a negative z-value.



Tip: In this example, the tool is moved directly into a gripping position. In most cases, an "approach" waypoint would be added before the grip position to avoid colliding with the object. Often this is just a case of offsetting the z-axis further away.

2.3.6 Activating tool

When the tool has been moved into position, activating the gripper is done by adding the *UR Event* block to the program and selecting the event that closes the gripper. This assumes that the gripper control events have been setup as described in the previous section: *Before building a program*.

2.3.7 Function blocks

To make programs readable it is important to break down the program into smaller sets of instruction. For this, the *Function* and *Call function* blocks can be used. The program could for example be broken down as follows.

Fig. 2.19: Example of defining separate functions and calling them within the main “Program”

Not only can these blocks be used to increase readability and reuseability, but they are also very useful for testing when building a program. The block menu of the *Function* block has a “Execute this function” button, to allow for testing smaller parts of a bigger program without being forced to play through the entire program.

Warning: When executing a function from the block menu it will use the current state of the programs variables and references. While this is very useful when testing and building the program, it can lead to issues if non-valid references are used, such as an old marker calibration, or if variables haven’t been initialized.

2.4 Variables

Variables are a fundamental part of the block-based programming found in the Ability software. They are used to store, manipulate and communicate values. Variables are given values and accessed using the *Get/Set variable blocks*.

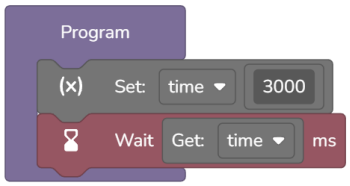


Fig. 2.20: A simple example of setting a variable and using it in another block.

Variables are often used in conjunction with the *If block* to check for certain values of the variable.

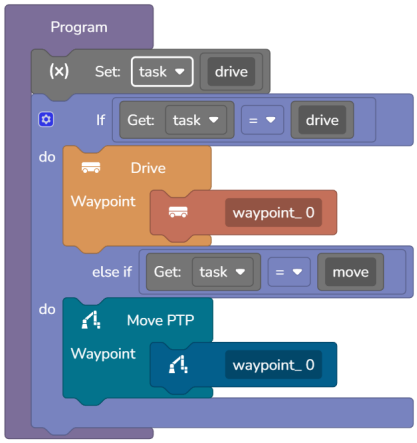


Fig. 2.21: A variable is given a string value and then compared against different possible values.

Tip: Variables are by default local within a programs workspace. This means they can be used across different functions in the same program. To use a variable, it simply has to have been initialized before the point in the program where it is used. Variables can be shared among programs by *Including programs* or by saving them globally using the *Save/Load variable blocks*.

Tip: Variables are by default non-persistent, meaning they are only stored during runtime of a program. If another program is loaded or the robot is rebooted, all the currently stored variables are cleared. To save variables persistently, the *Save/Load variable blocks* can be used.

To create a variable, go to the “Variables” button at the top of the programming workspace and press “Add new”. Then simply give it a name and press “Create”.

Fig. 2.22: A pop-up will show when creating or editing a variable. Variables can also be created by clicking the dropdown on the *Get/Set variable blocks*.

2.4.1 Value blocks

Value blocks hold a simple value of a certain type. They can be distinguished by the fact that they are rectangular without the jigsaw "tick" at the top and bottom of the block. This also means that they cannot be put into the program sequence with normal blocks.

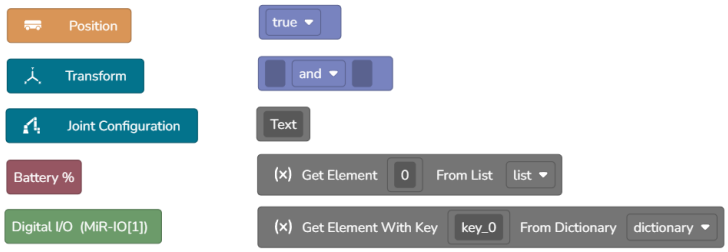


Fig. 2.23: A few examples of value blocks. The blocks return different types of values.

Value blocks are organized into sub-categories, e.g. *Position* is found in the values sub-category of *Mobile Platform* and Transform would be found under *Manipulator*. Value blocks that are not related to a specific category are found in the Values category block category.

2.4.2 Value fields

Value blocks only fit into value fields. Value fields are found on certain blocks indicated by a small rectangular darkened area. Most value fields expect a certain type of value, e.g. the Wait expects a double or integer and the *If block* expects a boolean value. Some value fields can accept many types of values, for example the Set variable block will accept any type of value and assign this to the selected variable.

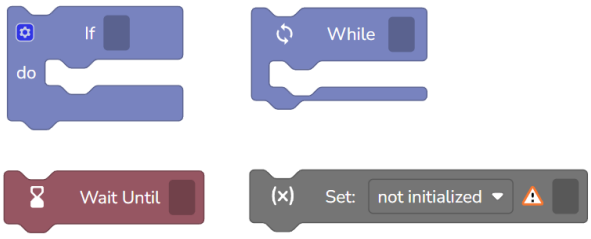


Fig. 2.24: A few examples of value fields. The fields expect different types of values.

Tip: To learn what type a value block returns or what type a value field expects, please refer to the specific block documentation in the *Blocks* section.

2.4.3 Value types

The following table shows the different value types available in the programming interface along with a description of what kind of data they hold.

Table 2.1: Value types table

Type	Description
Boolean	Can be either “True” or “False”
Integer	A whole number: “1”, “42”.
Double	A floating point number: “1.2”, “3.1415”
String	A piece of text including numbers, letters and symbols
List	Holds a series of values. The values can be of any type e.g. integers, doubles, strings. It is also possible to create a list of lists. Values in a list are retrieved from their “index”. Lists are 0-index meaning the first value has index 0.
Dictionary	Similar to a list it holds a series of values, but instead of retrieving the values using indexes, values are retrieved from a key which is given to the value when creating the dictionary.
Transform	List of 6 values: x, y, z, roll, pitch, yaw. Describes the transformation between two points in space, normally a reference and a tool.
Joint configuration	List of 6 values: q0, q1, q2, q3, q4, q5. Describes the angle of each joint of the manipulator. Normally used to send the manipulator to a deterministic position.
Position	List of 3 values: x, y, angle. Describes the position and orientation of the robot on its internal map.

2.5 References

References are local coordinate frames in the space around the robot. They are used to define how the robot should move in relation to them. References can either be detected using vision such as with the *Calibrate to Marker* block, or they can be created by the user.

References are defined by a *parent* and a *transform*. For example, when using the *Calibrate to Marker* block, the block essentially calculates the transformation from the base of the manipulator to the zero point on the marker. In this case the *parent* is the manipulator base and the *transform* is a list of x, y, z, roll, pitch, yaw values describing the position of the marker relative to the base. *Calibrate to Marker* creates a new reference which can now act as the *parent* for other references.

When creating a reference, the user must select what the parent should be. The default options are:

1. **Base** - The base of the UR
2. **World** - The zero point of the internal map on the MiR

After creating a new reference, either manually or using a *Vision* block, they will show up as available parents as well. When initially creating a reference, the transform that is defined is called the *default transform*. Blocks may update the current transform of selected references during program execution, but every time the program is loaded, the current transform is reset to the *default transform*.

2.5.1 Creating a reference

User defined references can be created through the Reference menu. Clicking 'Update' will update the default transform values based on the current location of the selected tool in the selected parent coordinate system. The values defined here are stored with the program data.

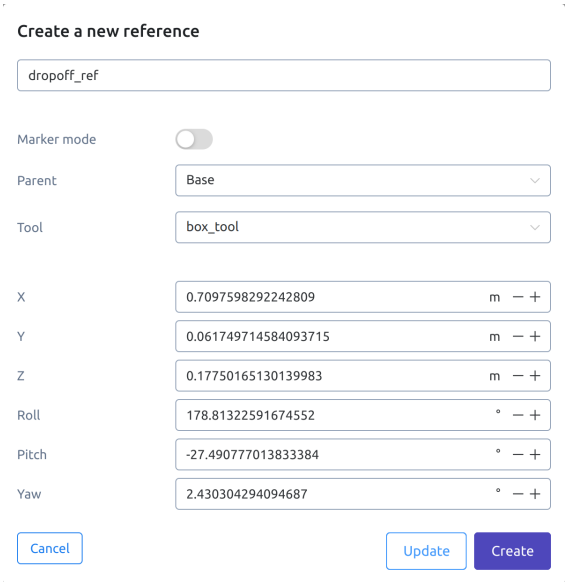


Fig. 2.25: Menu for creating a new reference

If 'Marker mode' is toggled, the default transform is computed automatically based on a marker detection. References created by the *Calibrate to Marker* block are initialized in marker mode.

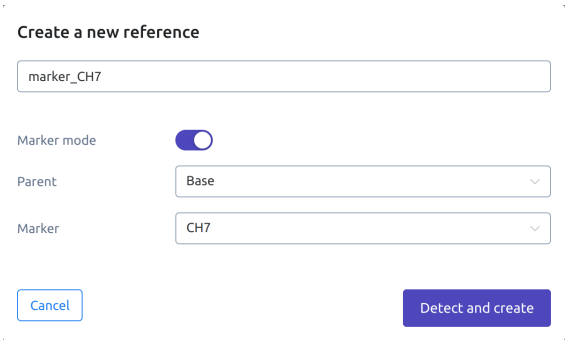


Fig. 2.26: Menu for creating a new marker reference

2.6 Including programs

In the block menu of the 'Program' block, it is possible to include other programs. Including other programs allows for calling functions and manipulating variables in those programs.

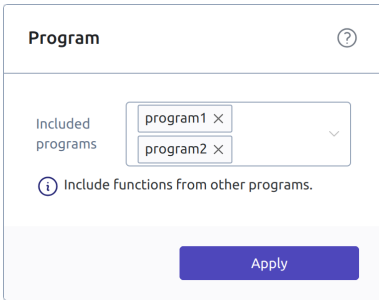


Fig. 2.27: Several programs can be included at once.

2.6.1 Calling functions

Functions from included programs can be called using the *Call Program Function* block.

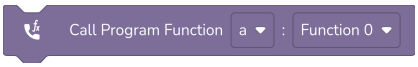


Fig. 2.28: The *Call Program Function* block has a field to select a program and a function.

2.6.2 Sharing variables

When including programs, all variables are shared between programs. This means that if "var1" is set in program A, and program A calls a function from program B, the function in program B will be able to use "var1".

Tip: When including a program, the variables of this program do not automatically show up in the variables tab, they need to be created manually. To make sure the correct variable is used, the names must be identical in each program, including letter case.

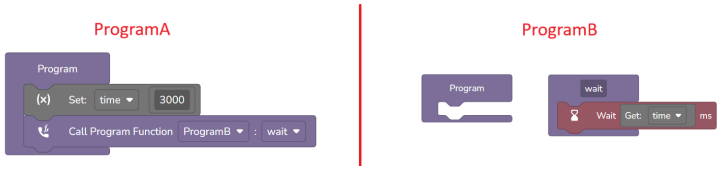


Fig. 2.29: A variable is set in one program and retrieved in another program.

Note: Programs that are included into other programs, normally have empty main "Program" blocks, as it is only the functions from the program that are called, not the program itself.

2.7 Program arguments

Program arguments allow for setting variables in a program at the time of running it. Arguments are passed into the program either from the *Mission builder* on the dashboard or through the *REST Interface V2*. Argu-

ments are often used to pass information such as a location ID for the Site layout manager. An argument, like a variable, consists of a *name*, *value* and *type*.

Note: Currently, program arguments are limited to be one of the four *Value types*: String, Double, Integer or Boolean.

2.7.1 Passing arguments into the program

In production environments, program arguments would normally be passed through the *REST Interface V2*, but for testing purposes the *Mission builder* on the dashboard can be used.

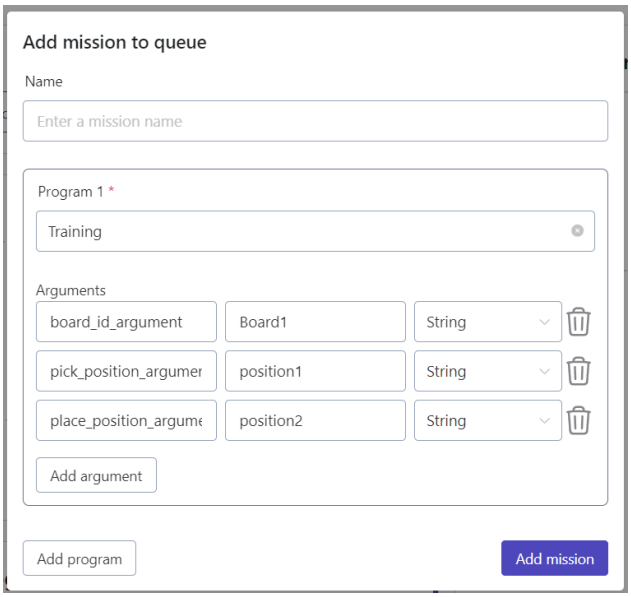


Fig. 2.30: Several program arguments can be passed at the same time.

When passing program arguments, the *name* is the unique identifier used in the program to access the argument.

2.7.2 Accessing arguments inside the program

Accessing arguments inside a program is done using the *Dictionaries* blocks. Program arguments are by default saved into a variable called **"arguments"**. This variable is of the type Dictionary and each key in the dictionary correpsonds to the *name* of each program argument. Using the Get element from dicitonary block, the value of the argument can be extracted from the dictionary based on its *name*.

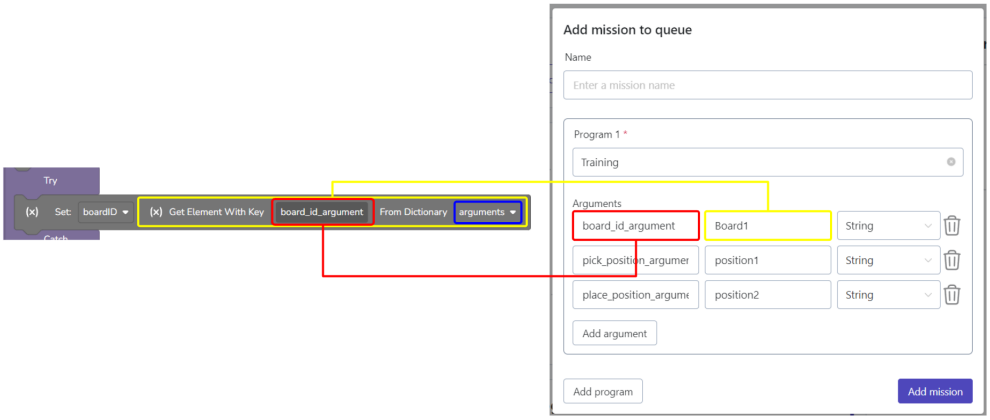


Fig. 2.31: The value of the argument is extracted from a dictionary called “arguments” using the *Get Element From Dictionary* block, and then saved into a variable.

CHAPTER

THREE

BLOCKS

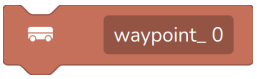
3.1 Mobile Platform

This section includes guides on how to use functions related to the mobile platform.

3.1.1 Mobile Waypoint

3.1.1.1 Overview

One of the fundamental building blocks of the mobile platform movement is the 'Mobile Waypoint' block. The waypoint block refers to a position on the internal map of the mobile platform. The block can be used in combination with the *Drive to Waypoint* block.



The block menu has two tabs: "Position" and "Variable"

- The **Position** tab defines a target position and stores it as x, y position in meters and an angle in degrees.
- The **Variable** tab makes it possible to define the waypoint at runtime and change it dynamically using a variable.

Tip: The current position of the mobile platform can be retrieved and used as the target position by pressing "Get current data" in the block menu.

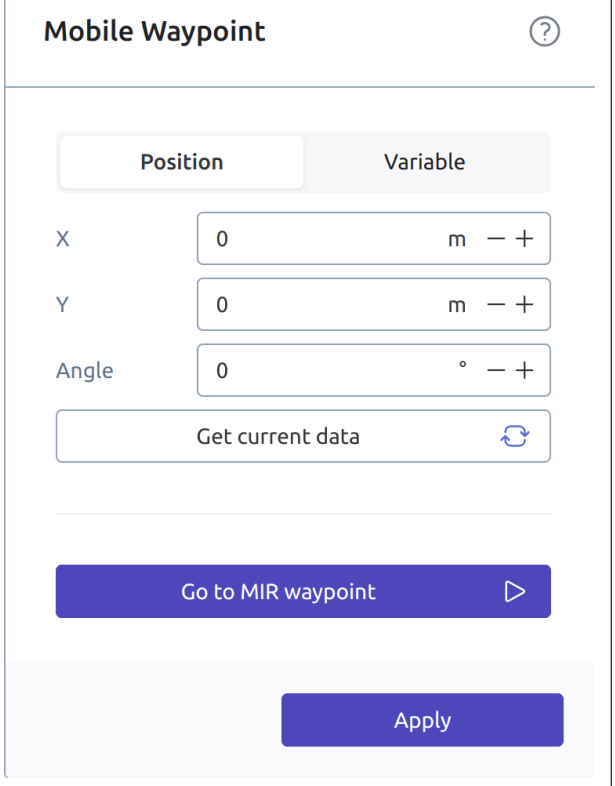


Fig. 3.1: The Position tab.

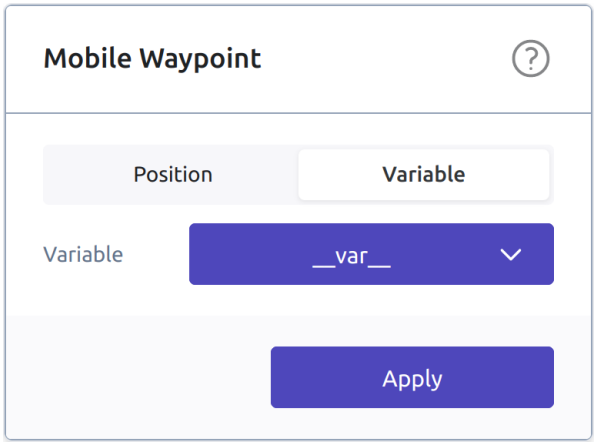


Fig. 3.2: The Variable tab.

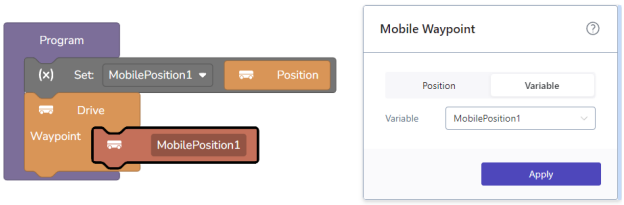
3.1.1.2 Examples

3.1.1.2.1 Example 1: Fixed position

This example stores the coordinates (x, y, angle) on the waypoint itself.

3.1.1.2.2 Example 2: Variable position

This example stores the coordinates (x, y, angle) in a variable called “MobilePosition1” using the *Position* block, and uses this variable for the waypoint.



3.1.2 Drive to Waypoint

3.1.2.1 Overview

The 'Drive to Waypoint' instruction block moves the mobile platform to the targted position. The path is planned between the current location of the mobile platform and the given waypoint.

The 'Drive to Waypoint' instruction block uses one or more *Mobile Waypoint* blocks. The *Mobile Waypoint* block describes a point on the internal map of the mobile platform. The mobile platform will move through all waypoints and stop at the last waypoint given.

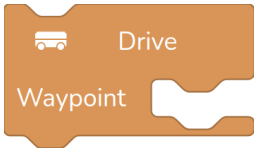


Fig. 3.3: The Drive to Waypoint block can hold any number of waypoints

The block menu has one field found under the 'Advanced' menu; **Planning Attempts**. **Planning Attempts** is used to determine how many times the mobile platform should retry to plan a path between the waypoints in case the initial path is blocked.

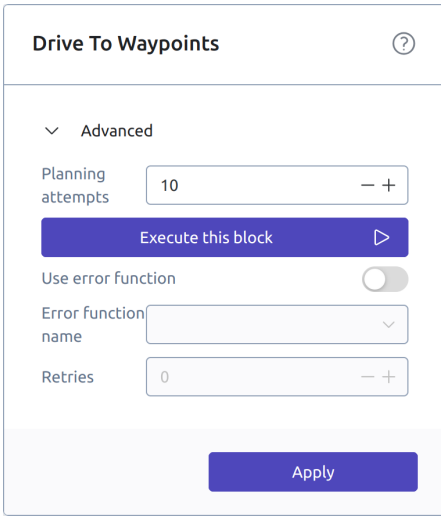


Fig. 3.4: The Drive to Waypoint block menu

Warning: With the 'Drive to Waypoint' block is it important to consider the area in which the robot operates. Areas with overhanging objects such as tables that the robot can collide with, can cause hazardous situations and areas which are dynamically changing can cause the robot to move slow and have trouble path planning.

3.1.2.2 Examples

For examples of use see *Mobile Waypoint*.

3.1.3 Drive to Position

3.1.3.1 Overview

Fig. 3.5: The Drive to Position block with a position dropdown.

The 'Drive to position' block is used to drive the mobile platform to a predefined position. The positions are retrieved from the mobile platform and can be added to the mobile platform map through the MiR interface. In the 'Advanced' menu, the **Planning Attempts** can be set. This defines the number of attempts the mobile platform will attempt at planning a path before going throwing an error in case the path is blocked.

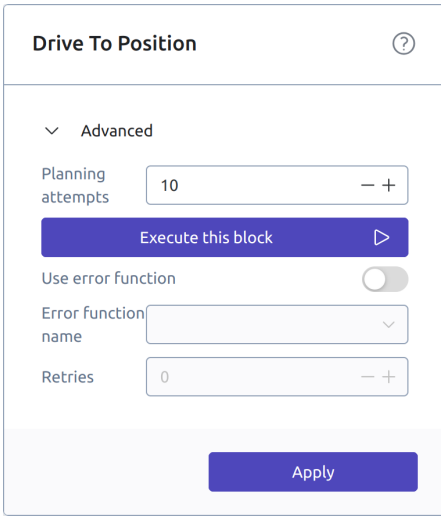


Fig. 3.6: Drive to Position block menu.

Warning: With the 'Drive to Position' block is it important to consider the area in which the robot operates. Areas with overhanging objects such as tables that the robot can collide with, can cause hazardous situations and areas which are dynamically changing can cause the robot to move slow and have trouble path planning.

3.1.4 Drive to Charging Station

3.1.4.1 Overview

Fig. 3.7: The 'Drive to Charging Station' with a charger dropdown.

The 'Drive to Charging Station' block is used to drive the mobile platform to a predefined charging station. The charging stations are retrieved from the mobile platform and can be added to the mobile platform map through the MiR interface.

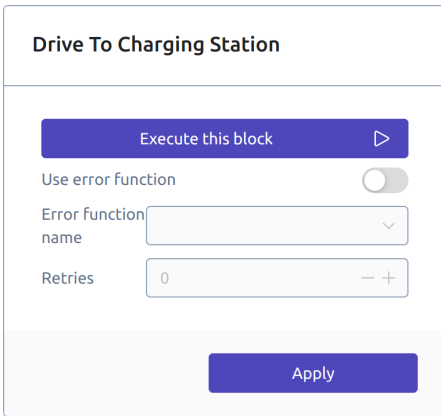
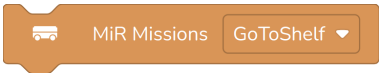


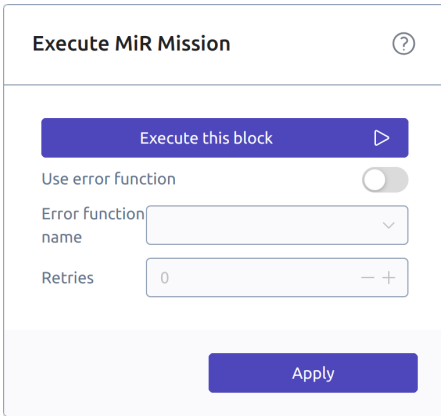
Fig. 3.8: The 'Drive to Charging Station' menu.

3.1.5 MiR Mission

3.1.5.1 Overview



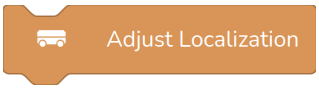
The 'MiR Mission' block is used to call missions created through the MiRs interface.



This block can be used to define subprograms as missions using the mobile platforms advanced functionality. An example could be to e.g. switch between maps at runtime.

3.1.6 Adjust Localization

3.1.6.1 Overview



The 'Adjust localization' block is an instruction block used to re-adjust the mobile platforms internal position. The block is useful in areas with many dynamic obstacles that can cause the position of the mobile platform to drift, or in areas with poor floor traction.

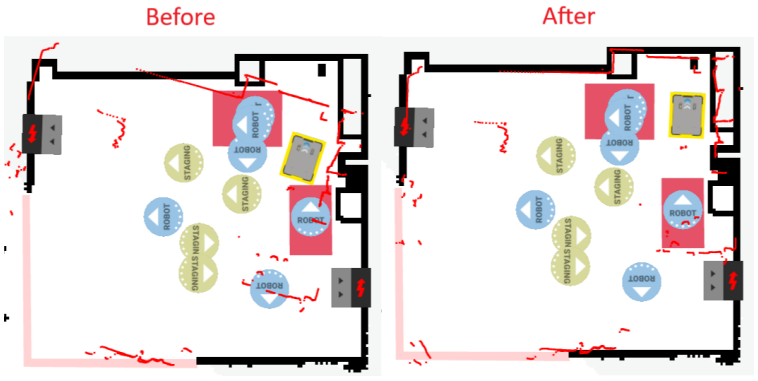
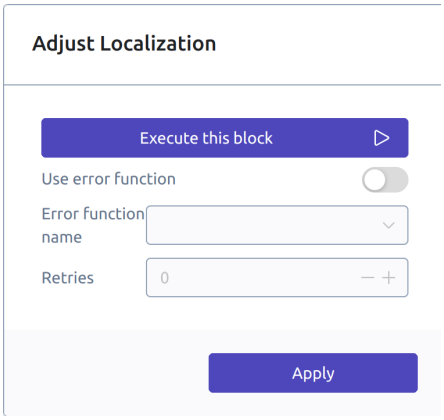


Fig. 3.9: The Adjust localization block attempts to align the laser scanner data of the mobile platform with the pre-recorded map of the environment.

Note: The block does not move the mobile platform, but only the internal position of the mobile platform.

The 'Adjust localization' block can be used in areas where drifting might occure and where it is crucial that the position of the mobile platform is precise.



3.1.7 Mute Protective Fields

3.1.7.1 Overview

The 'Mute Protective Fields' block mutes the protective fields around the mobile platform for the blocks that are within the scope of the block. The block affects the *Align Base*, *Drive Relative*, Drive to Position, Drive to Charging Station, and *Drive to Waypoint* blocks.

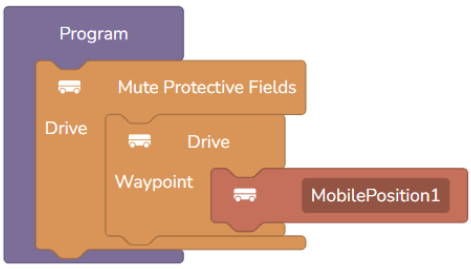
The protective fields are muted when one of the mentioned blocks are executed in the scope of the 'Muted Protective Fields'. The protective fields only reactivate when a *Mobile Platform* drive block is executed outside of the 'Mute Protective Fields' scope. The 'Mute Protective Fields' block can be used in areas where there is not enough space to maneuver or if the the mobile platform has to position itself close to an object or wall.

Note: This feature only works for MiR 250 and must be activated on the MiR interface. It can be found under settings/feature.

3.1.7.2 Examples

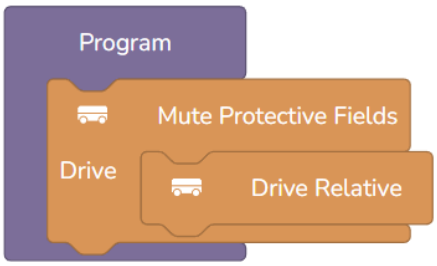
3.1.7.2.1 Example 1: Drive to waypoint

This example will drive the robot all the way from the current position to the defined waypoint with muted protective fields. If the robot is to drive a long way it is not recommended to mute the protective fields.



3.1.7.2.2 Example 2: Drive relative

This example mutes the protective fields to do a *Drive Relative*. This is common when trying to get the robot closer to objects.



3.1.7.2.3 Example 3: Align base

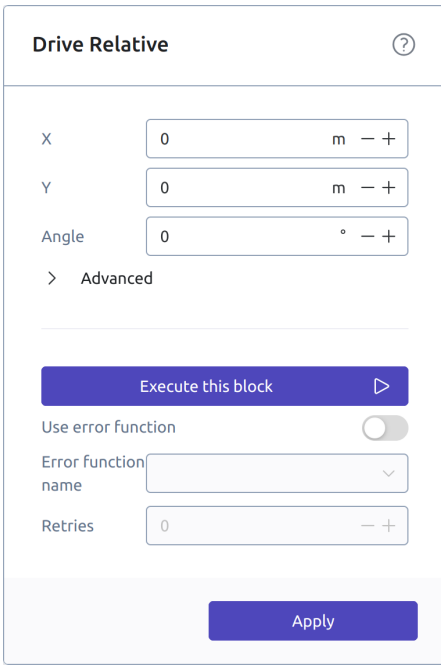
This example mutes the protective fields while running *Align Base*. *Align Base* does a series of relative movements of the mobile platform to align it to a reference like a marker.

Warning: Muting the protective fields can potentially result in hazardous situations as the robot will be able to move closer to objects.

3.1.8 Drive Relative

3.1.8.1 Overview

The 'Drive Relative' block is used to do a relative movement of the mobile platform.



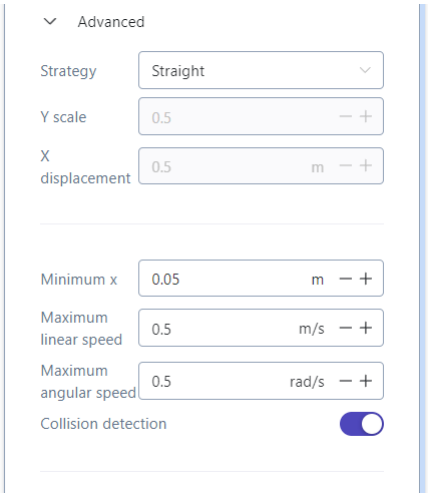
The relative movement is defined by a distance in **x** and **y**, and a **rotation**. The **x** parameter moves the robot forward or backward from its current position. A positive value moves the robot forward, and a negative value moves it backward. The values are in meters, so -0.5 moves the robot backward half a meter. The robot moves forward or backward from the way it is oriented when it receives the action, not along any axis on the map.

The **y** parameter moves the robot left or right from its current position. A positive value moves the robot to the left and a negative value moves it to the right. The values are in meters, so 0.5 moves the robot half a meter to the left.

Note: As the robot cannot move directly to the side, a movement where the **x** parameter is 0 will cause the robot to do several movements to obtain the desired relative movement. The "Strategy" parameter under advanced section defines how this movement is be done.

3.1.8.2 Advanced parameters

The 'Advanced' section of the block menu allows for modifying the nature of y-only movements, as well as the speed and safey scanner settings for the block.



- **Strategy**: This dropdown allows for selecting between “Straight” and “V-shape”. Selecting “Straight” will cause the robot to do a 90 degree turn around its own axis, move forwards the specified y-amount and then do another 90 degree turn to face in the original direction. Selecting “V-shape” will cause the robot to move forwards/backwards at an angle and then backwards/forward at an angle again to effectively move sideways in a “V” shape.
- **Y scale**: If “V-shape” is selected as the strategy, Y scale defines the shape of the V as shown below. The value must be between 0.001 and 1.

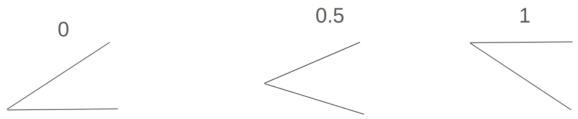


Fig. 3.10: The shape of the V movement is dependent on the y-scale parameter.

- **X displacement**: If “V-shape” is selected as the strategy, X displacement defines how far forwards and backwards the robot should move when performing the V shape. A positive value will make the robot move forwards and then backwards. A negative value will make the robot move backwards and then forwards. A value of 0.5 will as such move the robot forwards 0.5 meters and then backwards 0.5 meters. The value must be between -1 and 1. The lower the numerical value, the shallower the v shape will be.
- **Minimum x**: The threshold for when the chosen **Strategy** is used instead of a regular relative movement. If minimum x is set to 0.2, the **Strategy** will only be used if the x movement is less than 0.2. The y movement **Strategy** is only useful if the robot is to move more in y than in x, therefore this minimum x threshold might worth tweaking.
- **Maximum linear speed**: The maximum allowed speed of the robot in meters per second when moving forwards or backwards.
- **Maximum angular speed**: The maximum allowed speed of the robot in degrees per second when rotating.
- **Collision detection**: is selected as default. Collision detection makes the robot look for obstacles while it executes the action, and the robot will either stop or slow down to avoid colliding with the obstacle. Once the obstacle is no longer in the way, the robot will finish the action. In most situations, we recommend enabling Collision detection, but in cases where the robot needs to turn around its

own center in tight spaces, it can be a good idea to disable it to prevent the robot from stopping when it gets too close to surrounding obstacles or walls. Collision detection is not a safety function. If a person enters the Protective field of the robot while Collision detection is disabled, the robot will still enter Protective stop.

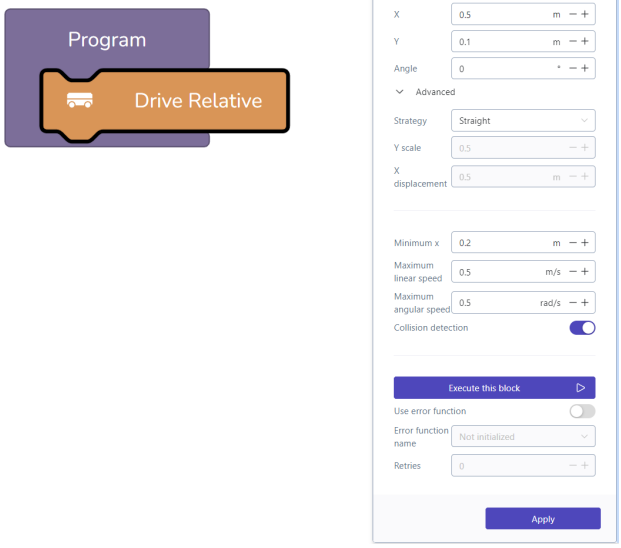
Warning: Changing the 'Advanced' settings can potentially result in hazardous situations as the robot will be able to move faster and closer to objects.

Important: When disabling the **Collision detection**, the laser scanners are still active, meaning the mobile robot will still stop if, e.g. a person walks in front of it.

3.1.8.3 Examples

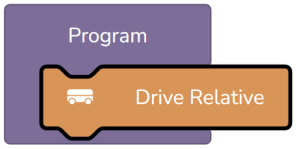
3.1.8.3.1 Example 1: Regular relative movement

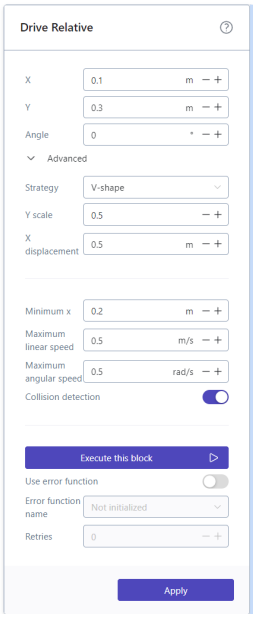
This example will have the robot do a regular relative move forward and a little to the left. The robot will rotate slightly to the left, move forwards until the desired x offset of 0.5 meters and y offset of 0.1 meters has been achieved, then rotate slightly to the right to point in the original direction. The strategy is ignored in this case as the x parameter is greater than the minimum x parameter.



3.1.8.3.2 Example 2: Y-movement strategy

This example will move the robot to the left and slightly forwards. In this case the x parameter is less than minimum x and therefore the selected strategy is used. In this case the v-shape. The v-shape will have the same angle on each side of the v as per the y-scale parameter. The robot will rotate to the left, move forwards 0.5 meters, rotate to the right, move backwards until the final x, y offset is reached and then finally do a rotation to point in the original direction.



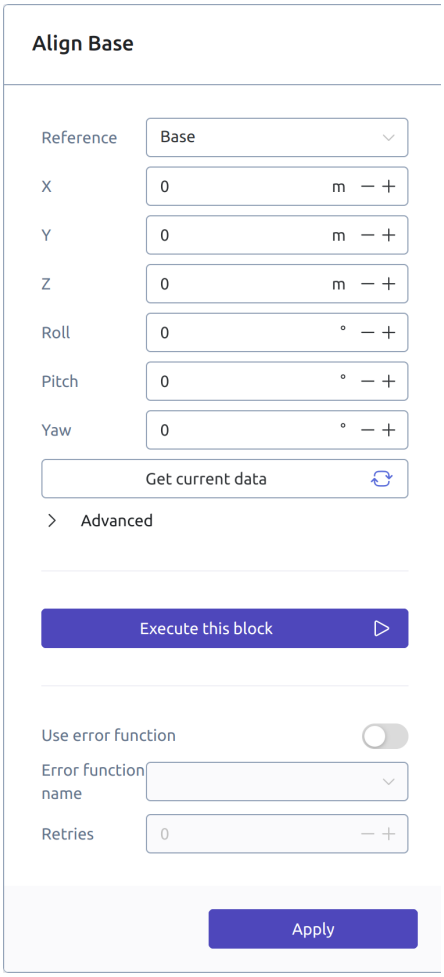


3.1.9 Align Base

3.1.9.1 Overview

The ‘Align Base’ block is used to align the mobile platform to a reference. It is often used in collaboration with *Calibrate to Marker*. This block is useful when higher accuracy is required for the positioning and orientation of the mobile platform, often in cases where reach of the manipulator is limited.

The block works by defining a desired transformation between the base of the manipulator and a chosen reference. When executing the block, the mobile platform will do a series of relative movements to obtain the desired transformation as accurately as possible. In many cases, a check is done beforehand with the Is mobile platform positioned block to check if it is actually necessary to run Align base.



The 'Align Base' block has the following settings:

- **Reference**: The reference used for the alignment. In most cases this would be a marker.
- **X**, **Y**, and **Z**: The position, in meters, of the base of the manipulator in the reference frame.
- **Roll**, **Pitch**, and **Yaw**: The orientation, in degrees, of the base of the manipulator in the reference frame.

Furthermore, the 'Advanced' section offers a series of parameters that allow for tweaking how the actual movement is executed. These parameters are similar to those found on the *Drive Relative* block. To learn more see: Drive relative advanced parameters.

Warning: Changing the 'Advanced' settings can potentially result in hazardous situations as the robot will be able to move faster and closer to objects.

3.1.9.2 Examples

3.1.9.2.1 Example 1: Align the mobile platform to a marker

The following is how to set up a simple program that aligns the mobile robot to a marker.

- **Step 1:** Position the mobile platform

  Use the *Mobile joystick* or the brake release of the mobile platform to accurately position the robot in the desired position relative to the marker. Save the mobile position in a waypoint.

- **Step 2:** Set up a capture position

  Use the *Arm joystick* to place the arm in a configuration so that it can see the marker in the camera view.

- **Step 3:** Initialize marker

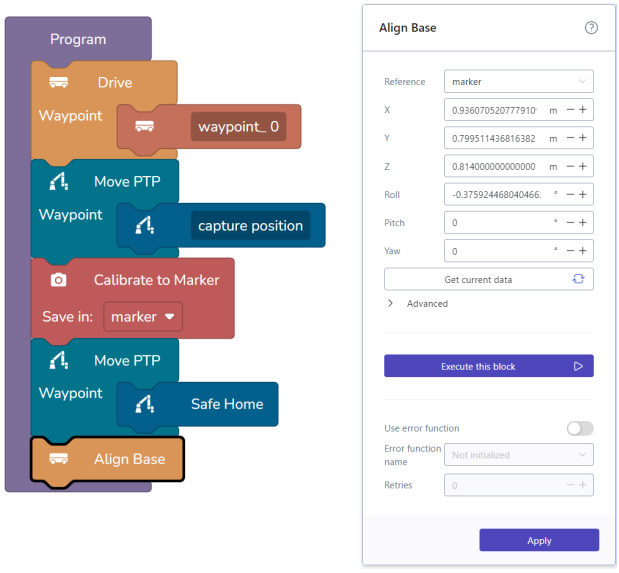
  Use the ' *Calibrate to Marker*' block to initialize the marker.

- **Step 4:** Move the arm back to safe home

  The Align base block will move the mobile platform, it is therefore important to make sure the arm is in a safe configuration before running the block.

- **Step 5:** Setup the 'Align Base' block

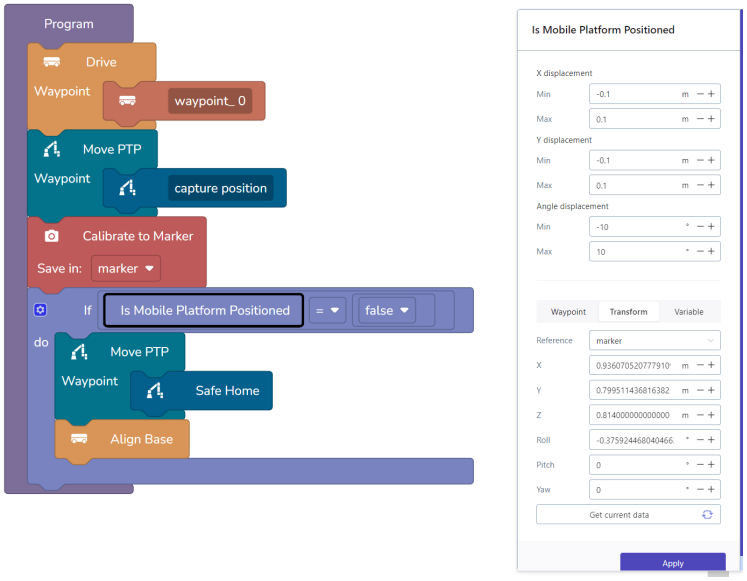
  Save the alignment position by opening the Align base block menu. Select the initialized marker reference and press the 'Get current data' button. Now the 'Align Base' block is ready to be used in the program.



3.1.9.2.2 Example 2: Use 'Is mobile platform positioned' to check if aligning is necessary

In this example, a check is done with the Is mobile platform positioned block to evaluate whether an alignment is needed. The Is mobile platform positioned block works by defining minimum and maximum allowed deviations in x, y and rotation from the defined transformation. The block returns a boolean value saying whether or not the current position is inside of these values or not. By adding an *If block*, the Align base block is only executed when the position is outside on or more of the values.

The reference and transform defined on the Align base block and those on the Is mobile platform positioned would normally be the same to ensure that the position that is checked against is also the one that the align base will try to achieve.



3.1.9.2.3 Example 3: Mute protective fields to align closer to objects

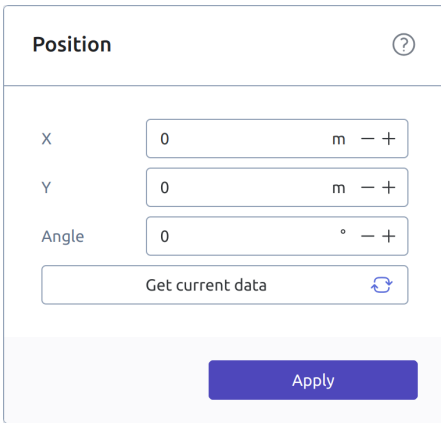
Another common use case for the Align base block is to align the robot closer to objects such as shelves, machines or workcells. To achieve this, it is combined with the Mute protective fields block, allowing the robot to navigate closer to the object. For this use case, it is recommended to use the “v-shape” strategy as described in Drive relative advanced parameters.

3.1.10 Position

3.1.10.1 Overview

The position block is a hardcoded mobile position value block.

The block menu contains fields for X, Y position in meters and rotation in degrees. It is normally used with the 'Set variable' block to save a mobile position in a variable.



3.1.10.2 Examples

For example of use see *Mobile Waypoint*.

3.1.11 Current Position

3.1.11.1 Overview

The current position block is a dynamic mobile position value block. At runtime the value is the current position of the mobile platform. The position is stored as the current position on the internal map of the robot with X, Y in meters and rotation in degrees.

3.1.12 Is Mobile Waypoint Valid

3.1.12.1 Overview

The 'Is mobile waypoint valid' is a boolean value block that checks if the waypoint given is valid according to the internal map and zones of the mobile platform. It can be useful when mobile positions are received from external systems that may not know the exact details of the map. By checking the validity of the waypoint before trying to drive there, time can be saved.

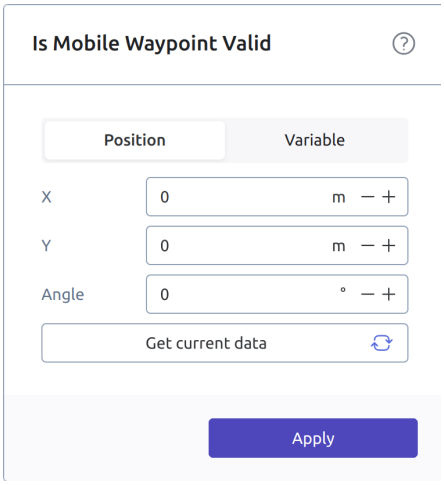
The result of the check will be **false** in the cases where the defined waypoint places the footprint of the robot such that it touches either of the following:

- An undefined piece of the map (no floor)
- A wall
- A forbidden zone

Note: The block does not check that the waypoint can actually be reached. E.g. if the position is clear but blocked off behind a wall, the block will still return **true**.

The block menu for the 'Is mobile waypoint valid' has two tabs: **Position** and **Variable**.

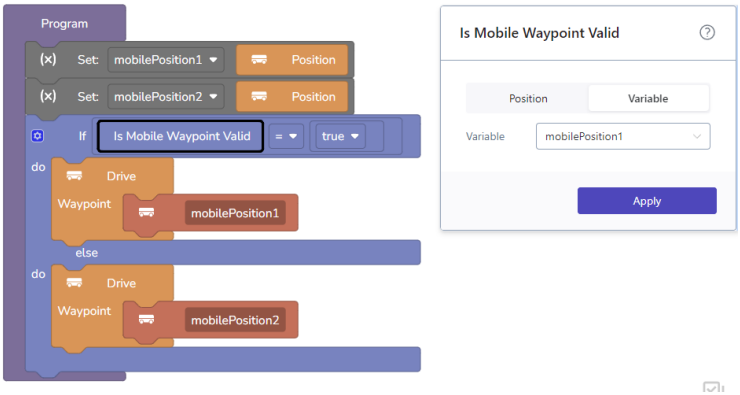
1. The **Position** tab is for using a user-defined waypoint and is defined by **X** and **Y** in meters and an **Angle** in degrees.
2. The **Variable** tab allows for having a waypoint that can be changed dynamically at run time. The variable must be of the type "mobile position". See *Variables* for more information on value types.



3.1.12.2 Examples

3.1.12.2.1 Example 1: Checking waypoint is valid before driving to it

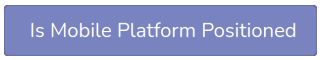
This example defines 2 mobile positions as variables and then checks that the first one is valid before driving to it. If it is not, the robot tries to drive to the other waypoint.



3.1.13 Is Mobile Platform Positioned

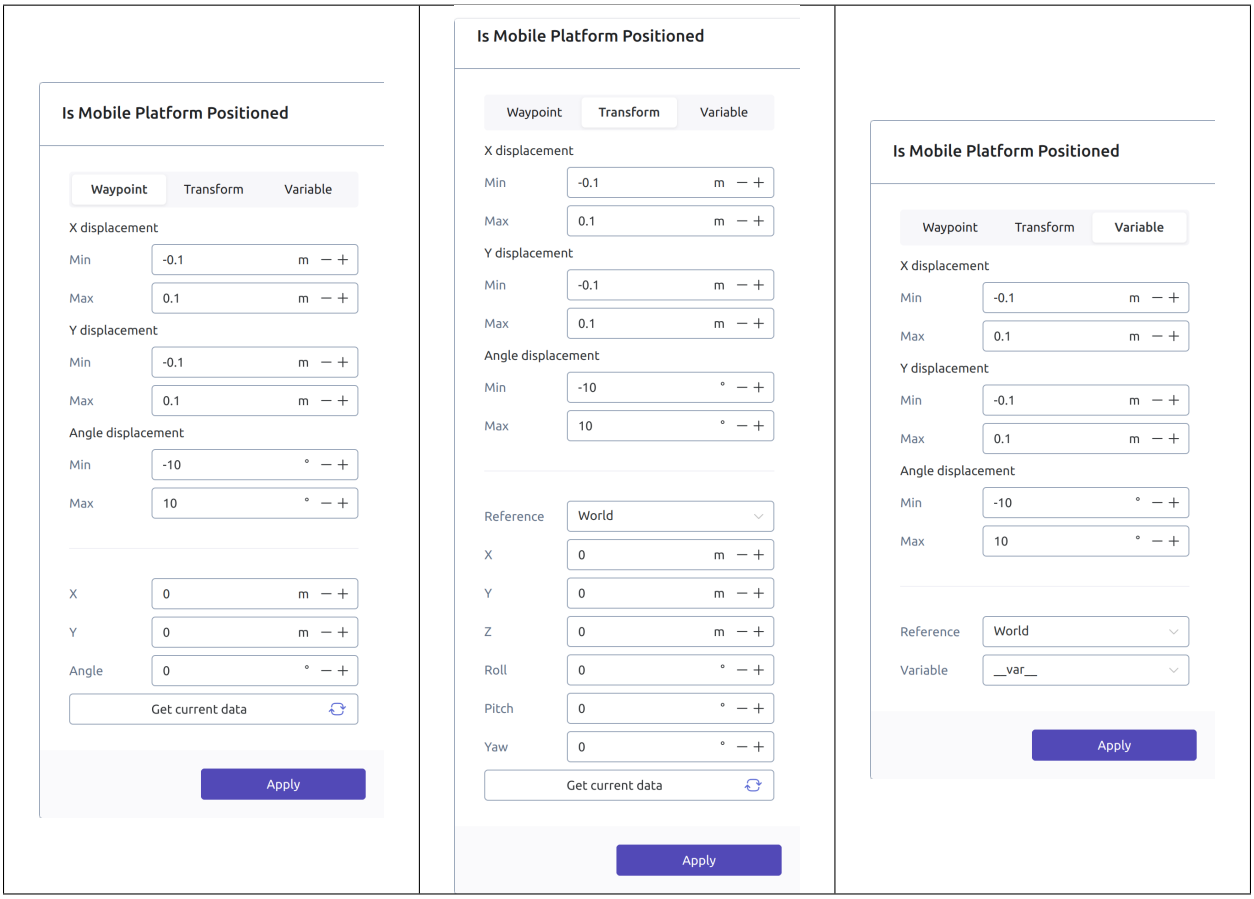
3.1.13.1 Overview

The 'Is mobile platform positioned' is a boolean value block that checks if the mobile platform is at or close to a given position.



The block menu for the 'Is mobile platform positioned' has three tabs: **Waypoint**, **Transform** and **Variable**.

All tabs have the three parameters **X displacement**, **Y displacement** and **Angle displacement**, which describe the minimum and maximum allowed difference between the position defined in the block and the actual position of the mobile platform.



- The **Waypoint** tab uses a mobile position defined by **X** and **Y** in meters and an **Angle** in degrees. In this case, the block checks against a position on the internal map and evaluates whether the current mobile position is within the displacement parameters. In this case the accuracy is limited by the resolution of the internal map being 5cm. E.g. setting the displacement parameters to values lower than this does not make sense as the robot is not able to locate itself that accurately.
- The **Transform** tab is for comparing against a reference, often times a marker. *Calibrate to Marker* can have an accuracy of down to +/- 1mm. This means using the Transform tab allows the "Is positioned" block to very accurately determine its current position versus the defined transform and evaluate

whether it is within the specified displacement parameters.

- The **Variable** tab allows for having a waypoint that can be changed dynamically at run time. The variable must be of the type "mobile position". See *Variables* for more information on value types.

3.1.13.2 Examples

For example of use, see example 2 in *Align Base*.

3.2 Manipulator

This section includes guides on how to use manipulator related blocks.

3.2.1 Manipulator Waypoint

3.2.1.1 Overview

The basic building block of the manipulator movement is the 'Waypoint' block. The waypoint block can be used in combination with different move blocks like *Move PTP and Move Linear*, the block defines a point in space the manipulator is supposed to move to/through.

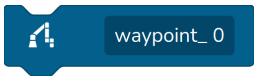


Fig. 3.11: The Waypoint block

The block menu for the waypoint has three tabs: **Configuration**, **Transform**, and **Variable**.

1. The **Configuration** tab is for saving a specific joint configuration, Q0 being the base joint and Q5 being the outermost wrist joint. Pressing "Get current data" will read in all the current angle of each joint. Joint configurations are deterministic and will alwyas result in the same manipulator pose.
2. The **Transform** tab is for saving a position of the *Tool* defined by the parent move block. The reference field represents what reference the tool position should be saved relative to. The default will be the Base of the robot arm and the other option will be the World. There are two different methods to get more *References*. 1: adding a new reference in the reference menu. 2: using a vision instruction block like *Calibrate to Marker*, see *The Transform block menu*. Pressing "Get current data" will read in the current Transform between the selected reference and the tool selected on the parent "Move" block. Transforms/Tool positions are not deterministic, meaning that several poses of the manipulator can achieve the same tool position. The manipulator will always go to the pose which results in the shortest joint path.
3. The **Variable** tab is for having a waypoint that can be changed dynamically at run time, the variable can be both a *Joint Configuration* and a Transform, see *Variables* for more information on variables. The 'reference' field found in the tab is only relevant for Transform variables. Like in the transform tab, it defines what reference the tool position should be saved relative to, see *The Variable block menu*.

Note: All three tabs have local settings where each waypoint can override the velocity, acceleration, and blend set by the parent Move instruction.

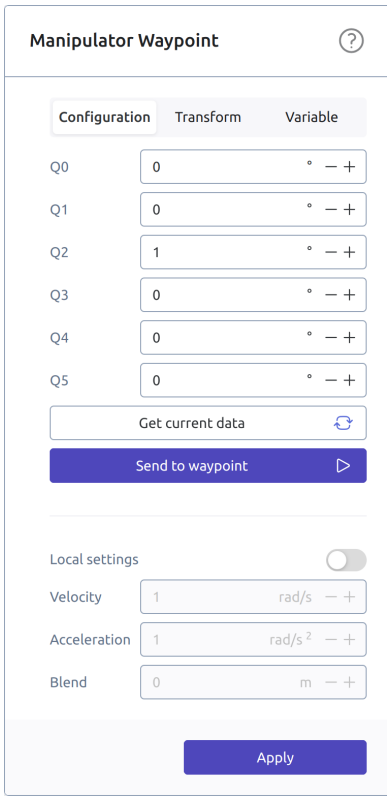


Fig. 3.12: The Configuration block menu

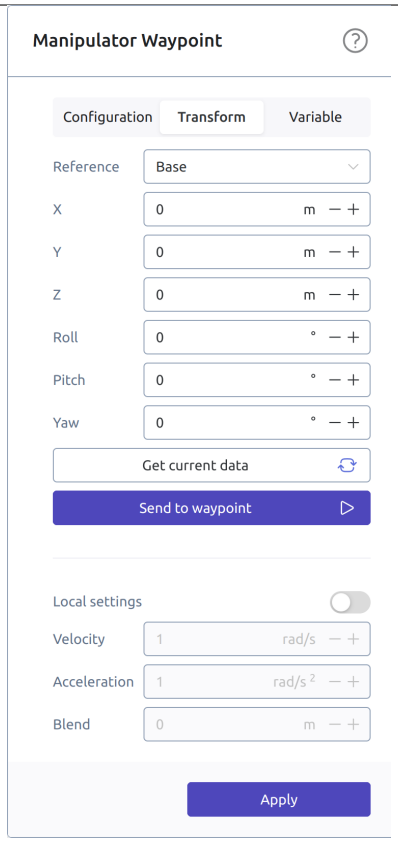


Fig. 3.13: The Transform block menu

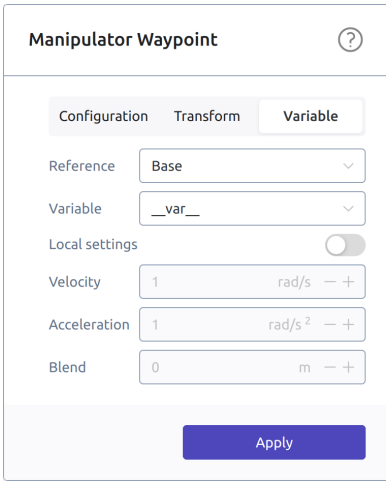


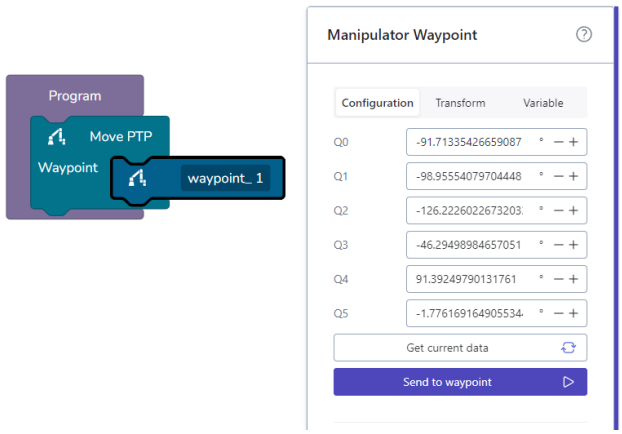
Fig. 3.14: The Variable block menu

Tip: Names of non-variable waypoints must be unique. If a waypoint is defined by a variable, the waypoint will take the name of the variable meaning variable waypoints can have the same name.

3.2.1.2 Examples

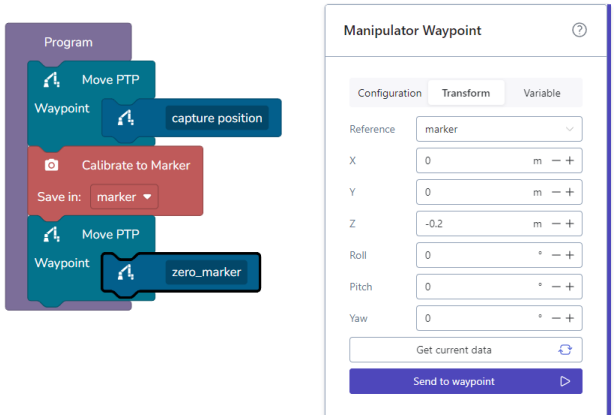
3.2.1.2.1 Example 1: Fixed joint configuration

This example stores a fixed joint configuration on the waypoint itself. This is a common way to save simple poses like capture positions for markers or intermediate positions when moving the arm between other positions.



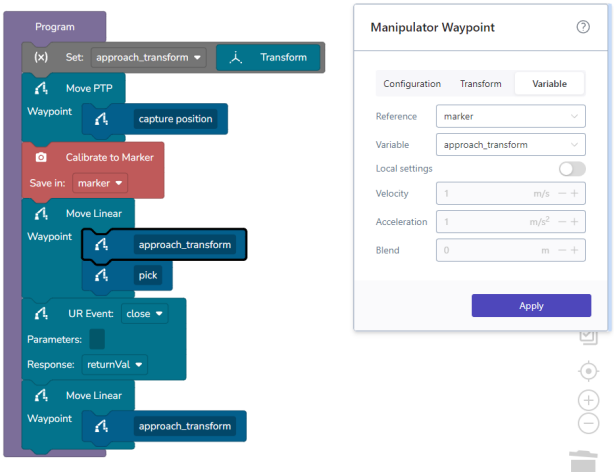
3.2.1.2.2 Example 2: Fixed transform relative to marker

A common use of the Transform tab is to define tool positions relative to markers or other references. The example below moves the selected tool (default is the tool flange) 0.2 meters above the zero point of the marker. The *Building a simple program* section covers this in more detail.



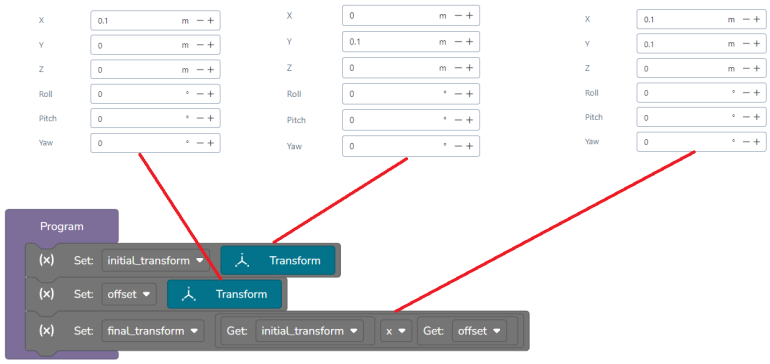
3.2.1.2.3 Example 3: Variable transform

Saving transforms to *Variables* allows for re-using the same tool position in several waypoints. Changing the saved Transform will change the tool position for all waypoints using that variable. An example use for this is "approach" positions, e.g. the position immediately before and after picking/placing an object as these postions are often the same.

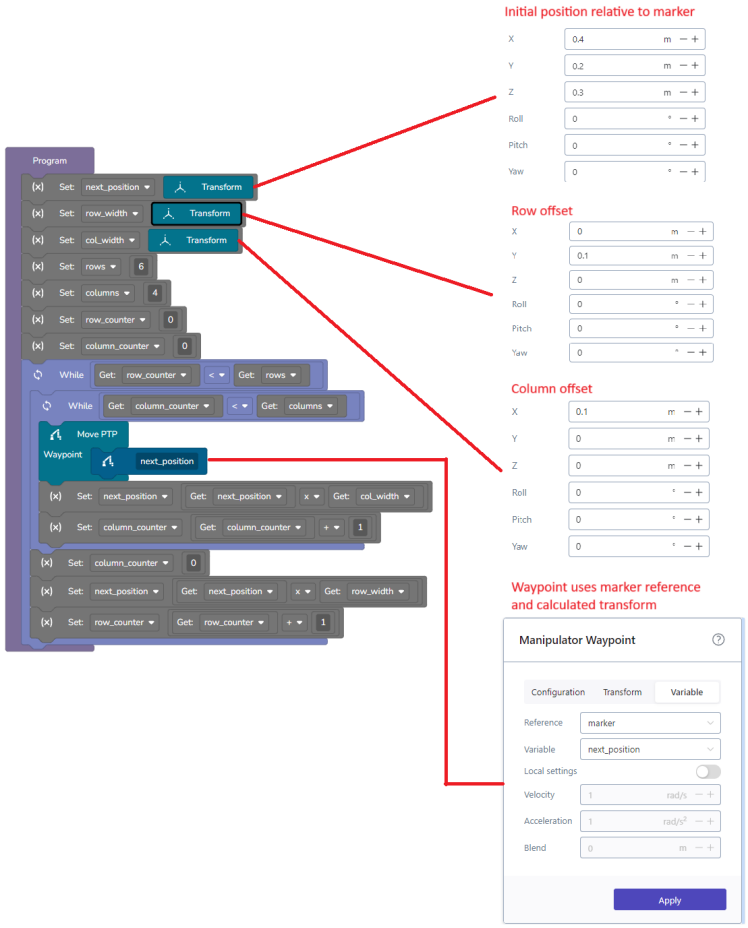


3.2.1.2.4 Example 4: Multiplying transforms to create offsets and palletizing patterns

Transforms can be multiplied using the Calculated value block to create dynamic tool postitions. Multiplying two transforms will combine their offsets as seen below.



Combining this with *Loops* allows for creating palletizing patterns as seen below by iteratively multiplying the same transform with x and y offsets. This example only palletizes in one layer, but layers could be added with an additional nested loop.



3.2.2 Safe Home

3.2.2.1 Overview

The 'Safe Home' block is a predefined waypoint block which corresponds to the safe home position defined on the UR.

Fig. 3.15: The Safe Home block

Note: The safe home position needs to be configured on the UR arm; otherwise, the block throws an execution error during execution. To learn more about how to set it up, see *Safe home configuration*

The block menu for the 'Safe Home' block allows the waypoint to override the velocity, acceleration, and blend set by the parent Move instruction.

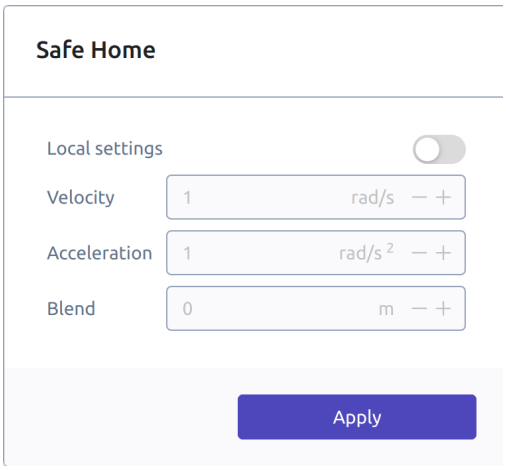
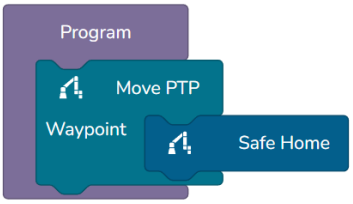


Fig. 3.16: The Safe Home block menu

3.2.2.1.1 Example 1: MovePTP to Safe Home

To use the 'Safe Home' block it must be put inside a 'Move' block.

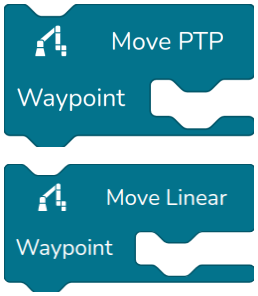


3.2.3 Move PTP and Move Linear

3.2.3.1 Overview

The 'Move PTP' and 'Move Linear' instruction blocks are the basic movement blocks for the manipulator of the robot. The 'Move PTP' block moves the manipulator the shortest distance in joint-space and the 'Move Linear' moves the shortest distance in cartesian space.

Both Move instructions use the same type of *Manipulator Waypoint* block. This block type describes a point in space that the robot has to move through or to. The Move instruction blocks can contain any number of *Manipulator Waypoint* blocks. The manipulator will move through all waypoints and stop at the last waypoint in the list.



3.2.3.2 Usage

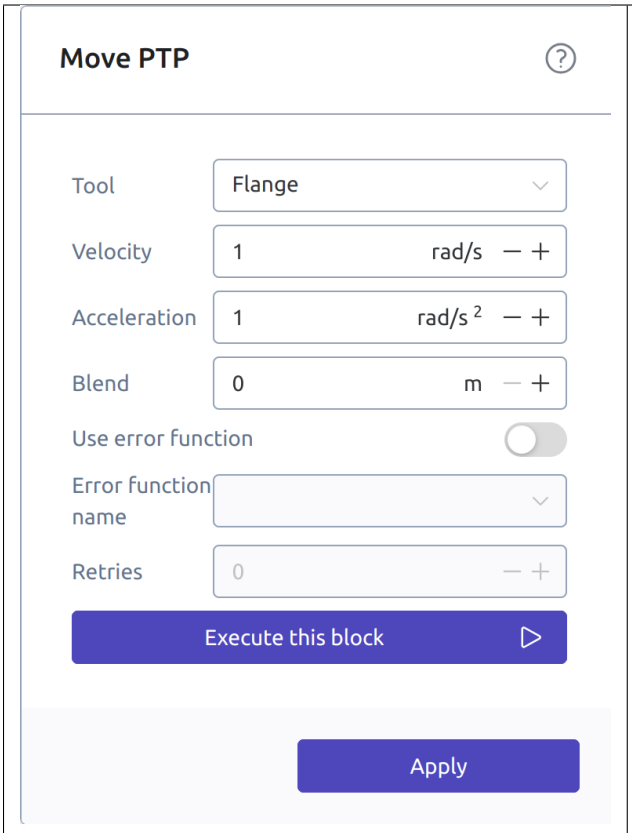


Fig. 3.17: Move PTP block menu

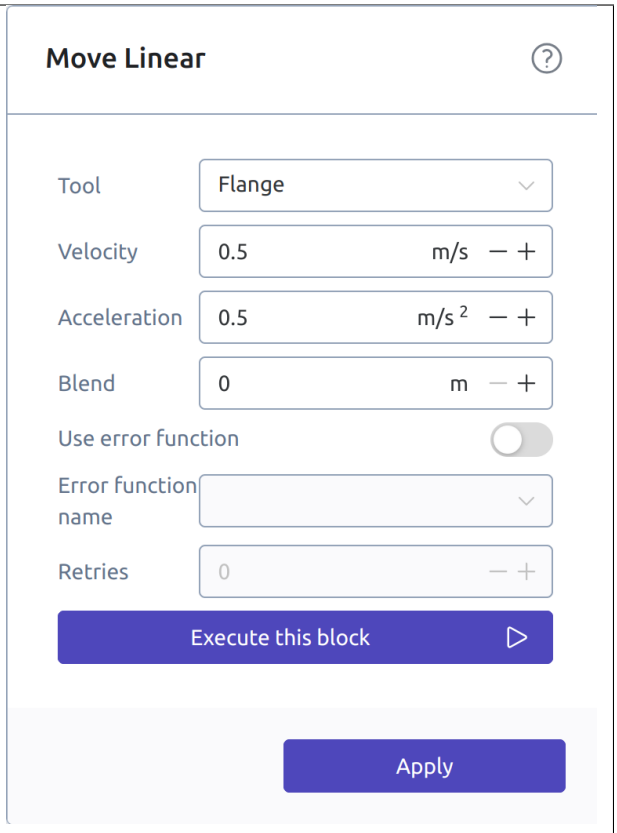


Fig. 3.18: Move Linear block menu

The block menus for Move PTP and Move Linear blocks are the same, the blocks have four fields: **Tool**, **Velocity**, **Acceleration**, and **Blend**.

- **Tool** defines what the TCP of the move applies to, as default, it will be the flange and have the camera as an option. To add more tools go to the *Setup* page.
- **Velocity** does not have the same definition for Move PTP and Move Linear, for Move PTP the velocity is represented by radians per second, for Move Linear it represents the velocity of the tool in meters per second.
- **Acceleration** does not have the same definition for Move PTP and Move Linear, for Move PTP the acceleration is represented by radians per second squared, for Move Linear it represents the acceleration of the tool in meters per second squared.
- **Blend**. If a blend radius is set, the robot manipulator trajectory will be modified to avoid the robot stopping at the point. This can make a movement through a series of waypoints smoother.

Warning: Changing these settings can potentially result in hazardous situations as the robot will be able to move and accelerate faster.

Important: If the blend region of a waypoint overlaps with the blend radius of the previous or following waypoints, the waypoint will be skipped. If it is important that the robot reaches a certain waypoint it is

recommended that the blend radius is set to 0 in the local settings of the waypoint. This way, the rest of the waypoints in the Move block will still be blended.

The 'Move PTP' block should be seen as the default manipulator movement instruction block, the manipulator will move straight in the configuration space, this means the robot will not need to move some joints much faster than other joints during the movement which can often happen when using the 'Move Linear' block.

Important: The 'Move Linear' block should only be used when the manipulator **must** move in a straight line, this is often the case when picking or placing objects.

Note: Another important feature of the 'Move PTP' block is that when combined with a *Manipulator Waypoint* saved as a configuration, it is the only method to guarantee that the robot is returned to the same joint configuration, therefore it is the correct choice for saving a safe home position for the robot manipulator.

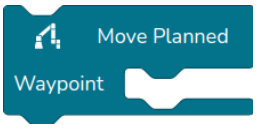
3.2.3.3 Troubleshooting

When using the 'Move Linear' block it is important to consider how the robot has to move in that straight line, a common issue is attempting to move the manipulator close to the base of the robot as this would require the base joint to move too fast. Or moving in a straight line while the flange of the robot is pointing vertically as this could require the third wrist to rotate at a high velocity. When using the move blocks with only tool waypoints the robot over time can hit a joint limit, the best way to avoid this is using a 'Move PTP' block and a configuration waypoint.

3.2.4 Move Planned

3.2.4.1 Overview

The 'Move Planned' block makes the manipulator move in a collision free path in joint-space. The block takes in a number of waypoints and plan a collision free path for the robot itself and the virtual box between the waypoints.



3.2.4.2 Usage

The 'Move Planned' block uses the *Manipulator Waypoint* block. This block type describes a point in space that the robot has to move through or to. The 'Move planned' block can contain any number of *Manipulator Waypoint* blocks. The arm will move through all waypoints and stop at the last waypoint in the list.

Warning: The 'Move Planned' block only plans a collision free path based on the internal model of the robot and not of the surrounding enviroment or added elements to the robot such as a tool or a camera.

The 'Move Planned' block menu, seen on image Fig. 3.19, have seven parameter fields: **Tool**, **Velocity**, **Acceleration**, **Add Collision Box**, **X dimensions**, **Y dimensions** and **Z dimensions**.

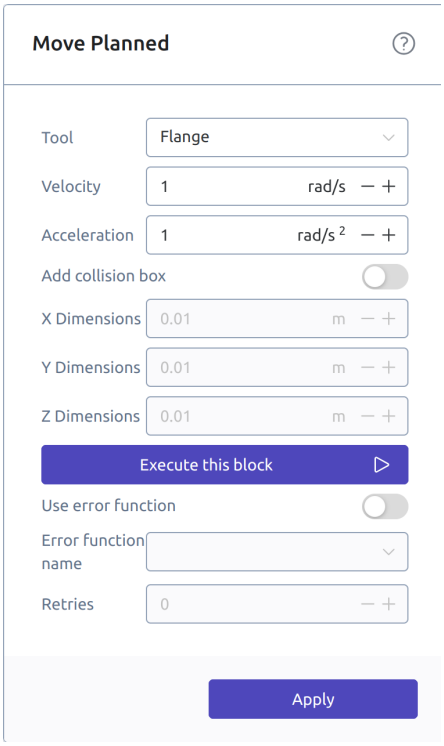


Fig. 3.19: The block menu

- **Tool** defines what the TCP of the move applies to, as default, it will be the flange and have the camera as an option. To add more tools go to the *Setup* page.
- **Velocity** is represented by radians per second.
- **Acceleration** is represented by radians per second squared.
- **Add Collision Box** adds virtual box to the flange of the robot.
- **Dimensions** is the dimensions of the virtual box and are represented by meters.

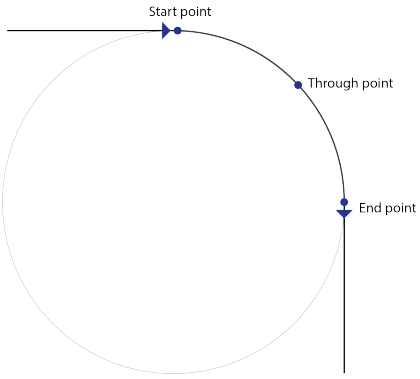
Warning: Changing these settings can potentially result in hazardous situations as the robot will be able to move and accelerate faster.

The virtual box is attached to the robots flange and follows the coordinate system of the flange.

3.2.5 Move Circular

3.2.5.1 Overview

The 'Move Circular' block makes the manipulator move in a circular arch. The block starts by doing a point-to-point movement to the starting point. It then starts moving in a circular ark by the through point and stopping at the endpoint.



For this reason, it is strictly necessary to have three and only three waypoints for the 'Move Circular' block, the current version of the block only supports the waypoints being of the tool type.

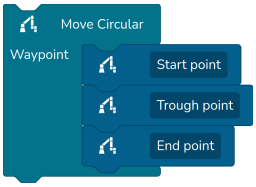


Fig. 3.20: The three waypoints of the 'Move Circular' block used for defining a circular motion

Important: The 'Move Circular' block should **only** contain the three specified waypoints

3.2.5.2 Usage

The 'Move Circular' block menu has the same four fields as the *Move PTP and Move Linear* blocks, please refer to these for more information about these fields. The block also has a toggle for setting the "Lock orientation" parameter.

The lock orientation toggle is used to lock the orientation of the tool at the starting point to the tangent of the circle arch. In other words, it will lock the starting orientation of the starting position to be pointing at the centre of the circle arch.

3.2.6 UR Event

3.2.6.1 Overview

The UR Event allows UR specific functionality to be executed from the block program. On the UR teach-pendant insert an Event Node under the ER-Ability program node. Give the event node a name. The event can now be called and executed from the block programming interface.

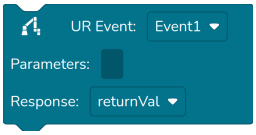


Fig. 3.21: The UR Event block

3.2.6.2 Usage

It is possible to pass arguments to the UR Event Node by adding a value (or a list of values) to the input value field of the block.

The following argument types are supported:

- **String**
- **Boolean**
- **Float**
- **Integer**
- **Pose (transform)**
- **Joint Configuration**

If the UR Event Node returns a value, it will be stored in the response variable.

The block menu for the UR event offers a button to execute the event.

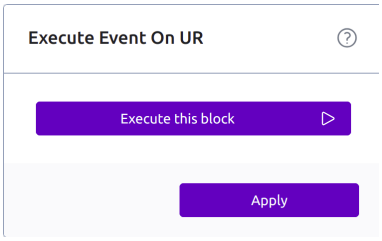


Fig. 3.22: The UR Event block menu

3.2.6.3 Setting up Event Nodes in the UR Polyscope interface

The Event Node is configured by defining a unique name (mandatory), specifying input arguments (optional) and selecting a return value (optional).

Input arguments are added using the “Add Row” button and removed using the “Remove Row” button. For each input argument it is required to specify a unique name and select a type.

The return value can be a global variable or any of the input arguments. It defaults to “none” which means “no return value”.

Note: Only events defined as sub-nodes to the ER-Ability node in the UR Polyscope interface can be selected from the ‘UR Event’ block.

An example of an UR Event Node is shown in Fig. 3.23. It can be called using the Event Block shown in Fig. 3.24.

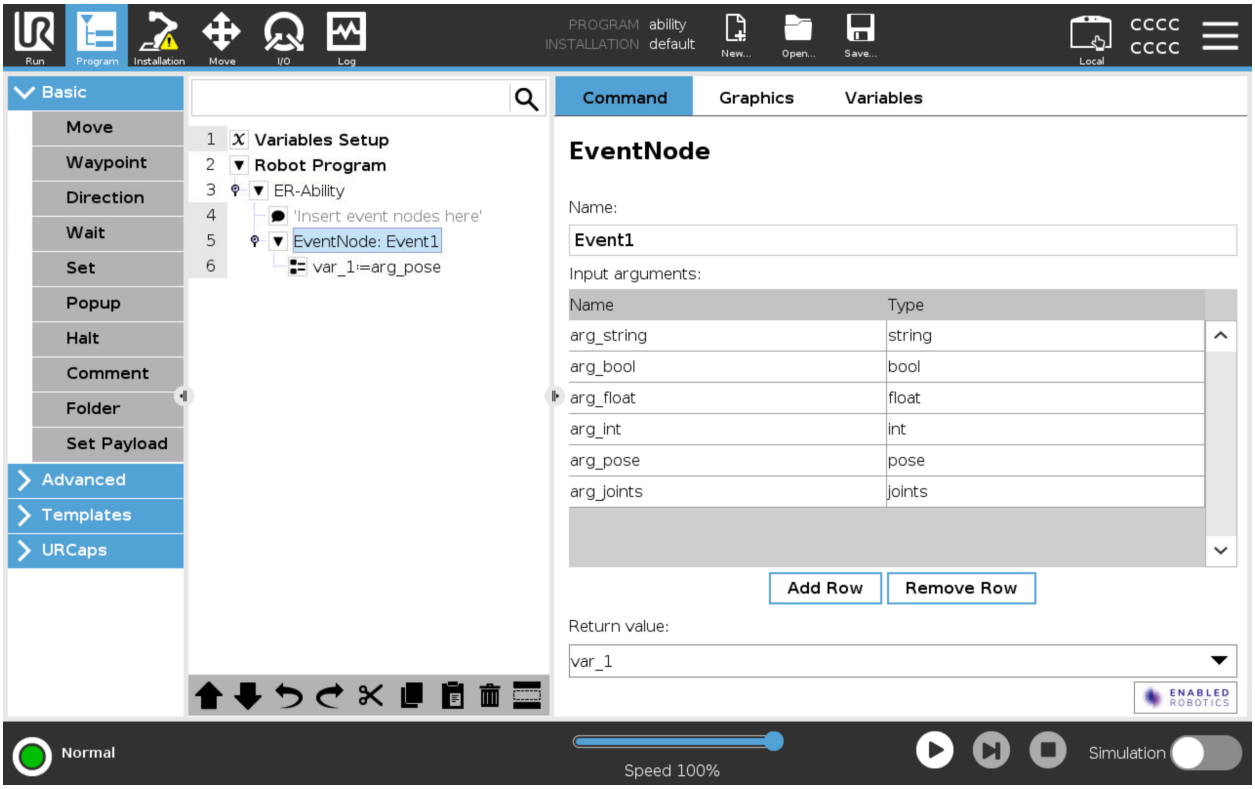


Fig. 3.23: UR Event Node example

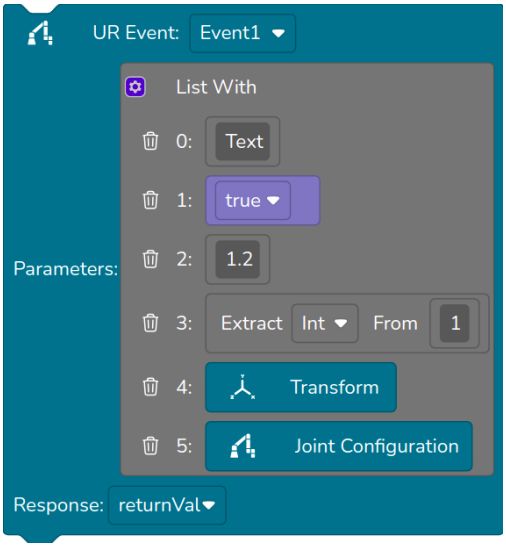


Fig. 3.24: UR Event block example

3.2.7 Check Reach

3.2.7.1 Overview

The 'Check That Waypoints Are Within Reach' block checks that all the waypoints inside are reachable from the current position of the mobile base. The result of the check are stored in a variable that can later be used in logic blocks as described in Logic

Fig. 3.25: Check Reach block

3.2.7.2 Usage

The block menu for the UR event offers no options. Just a button to execute the event now.

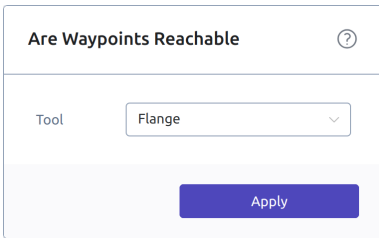


Fig. 3.26: The Check Reach block menu

3.2.8 Motion

3.2.8.1 Overview

The motion block describes a relative motion of the manipulator. The description is either given using two waypoints, where the difference between them is the motion or a direction XYZRPY relative to a given reference frame.

Fig. 3.27: The Motion block

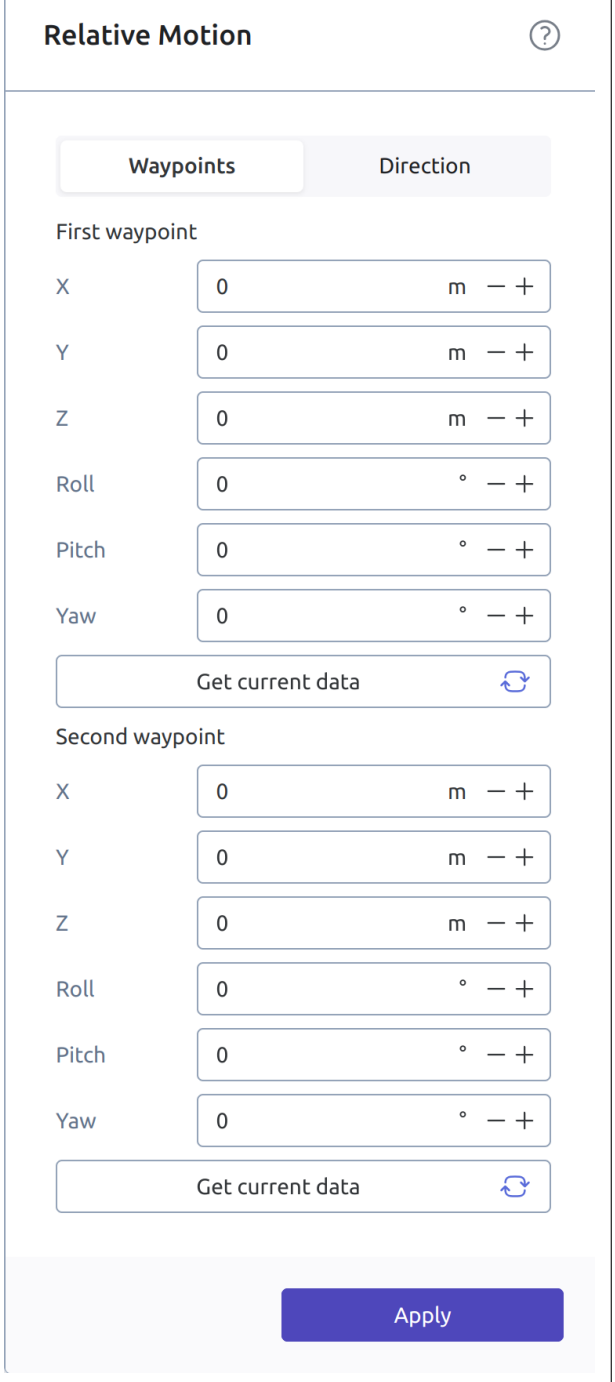


Fig. 3.28: Motion with waypoints.

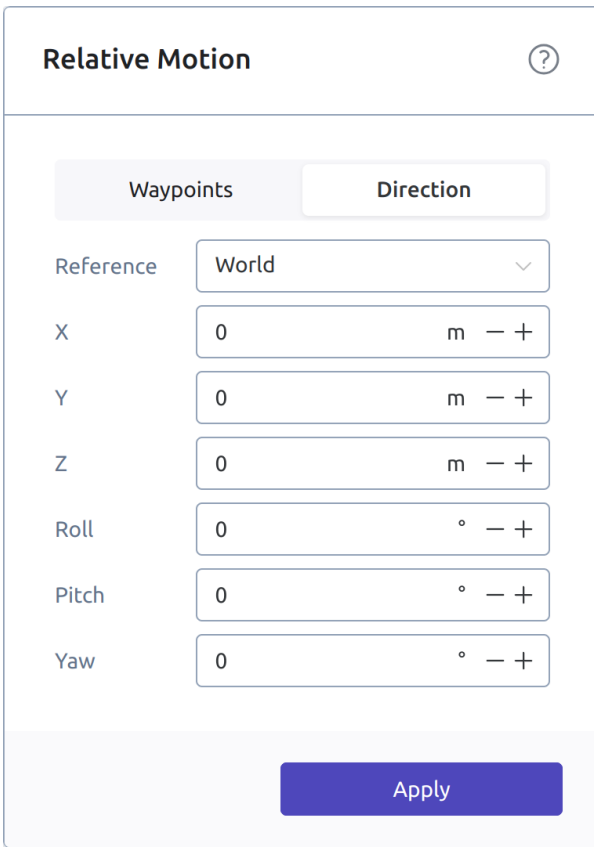


Fig. 3.29: Motion with direction.

3.2.9 Move Linear Relative

3.2.9.1 Overview

The 'Move Linear Relative' instruction block moves the robot relative to the current position of the arm. The block contains a 'Motion' instruction which is defined by either two waypoints or a direction in a reference frame.

Fig. 3.30: The Move Linear Relative block per default contains one Motion instruction but can contain several.

3.2.9.2 Usage

Unlike the other Move blocks, it is not possible to define the blend for the movement and it does not use the standard *Manipulator Waypoint* instruction blocks only 'Motion' blocks. Any amount of 'Motion' blocks can be added to the 'Move Linear Relative' block.

The Motion block menu has two categories. The first is the waypoint category, this category of motion defines a line between the two waypoints this line will then be applied to the current positions of the robot arm move to the end of the line. The second is the direction category, this category of motion is defined by moving a set distance or rotation in a reference frame.

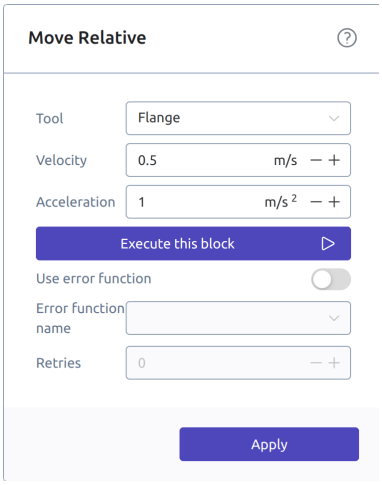


Fig. 3.31: The move relative block menu

3.2.9.3 Troubleshooting

A common issue when using the 'Move Linear Relative' block is the same as when using the standard 'Move linear' block. The robot can reach a joint limit if it does not get reset with a 'Move PtP' to a configuration.

3.2.10 Set Payload

3.2.10.1 Overview

The 'Set Payload' block is used to specify the payload for the manipulator. The payload is defined as the load mounted on the flange of the manipulator.

Fig. 3.32: The Set Payload block specifies the payload in kg.

3.2.10.2 Usage

The 'Set Payload' block has the following settings:

- **Payload**: The payload specified in kg, including tooling.
- **CoG X**, **CoG Y**, and **CoG Z**: The center of gravity of the payload relative to the flange of the robot. The CoG is specified in meters.

Warning: If the payload specified does not match the actual payload of the robot arm, the arm may enter into a protective stop.

Note: The Payload should be specified including the tooling on the robot.

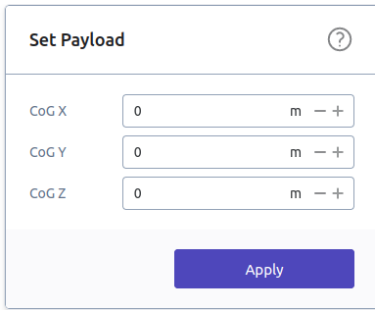
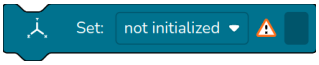


Fig. 3.33: The Set Payload block menu

3.2.11 Set Reference

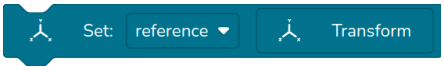
3.2.11.1 Overview

Updates the values of a reference. When a reference is updated all motions relative to this reference will be relative to the new values. This block enables to reuse same movements in a number of different locations. For instance, if having to execute the same assembly sequence in four locations, the movements can be programmed relative to a reference and only the reference will have to be updated for the four different locations.



3.2.11.2 Usage

Either drag a 'Transform', 'Current Transform' or 'Get Variable' block into the value field of the Set Reference block. Please note, the Set Reference block only updates the current transform of a reference. The parent frame can only be changed through the Reference menu in the top of the programming workspace.



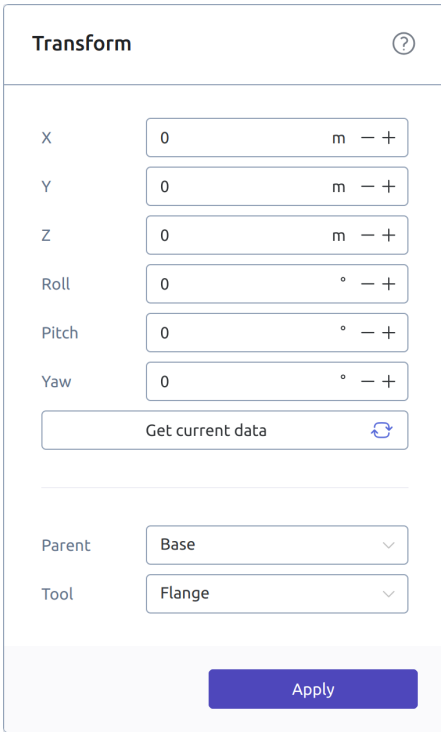
3.2.12 Transform

3.2.12.1 Overview

The Transform block is a hardcoded value block.

3.2.12.2 Usage

The block menu has the XYZ position in meters and the RPY angles in degrees.



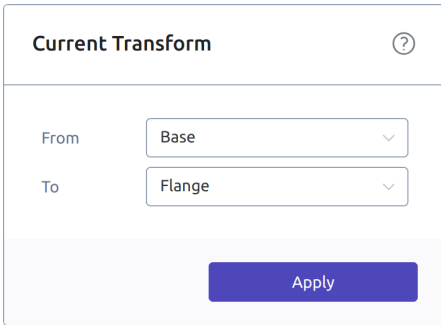
3.2.13 Current Transform

3.2.13.1 Overview

The current transform block is a dynamic transform value block. At runtime the value is the current transform between two references.

3.2.13.2 Usage

In the block menu, the two dropdowns define which two references to calculate the transform between.



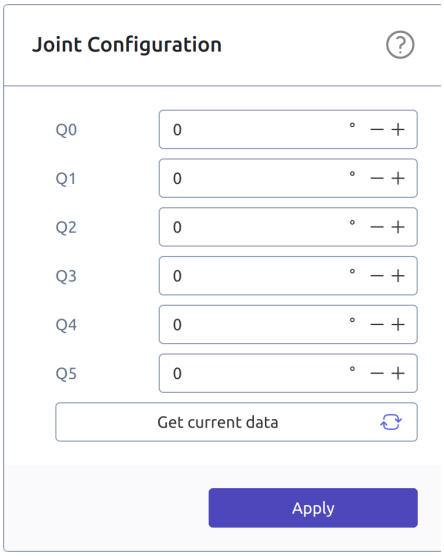
3.2.14 Joint Configuration

3.2.14.1 Overview

A hardcoded joint configuration value block

3.2.14.2 Usage

In the block menu, each joint can be defined in degrees.



3.2.15 Current Joint Configuration

3.2.15.1 Overview

The 'Current Joint Configuration' block is a dynamic joint configuration value block. At runtime the value is the current joint configuration of the manipulator.

3.2.16 Is Arm in Safe home

3.2.16.1 Overview

The 'Is Arm in Safe home' block is a boolean value block that returns true if the arm is in a safe home position otherwise false.

Note: The safe home position need to be configured on the UR otherwise the block will return true.

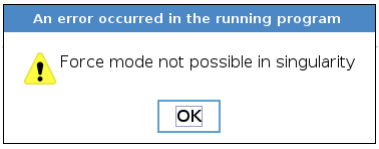
3.2.17 Force Blocks

3.2.17.1 Overview

The force blocks have the special ability to allow the manipulator to touch or collide with objects, without going into safety stop. This is particularly useful when aligning with objects or shapes that require very precise positioning of the robot. Typical applications are peg-in-hole problems, alignment to surface, screw-while-moving, etc.

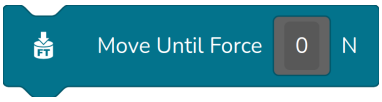
Warning: The force blocks utilize the internal force-torque sensor of the UR robot. This is only present in the e-series, and thus results may vary with the CB-series which use the motor torques to estimate TCP force.

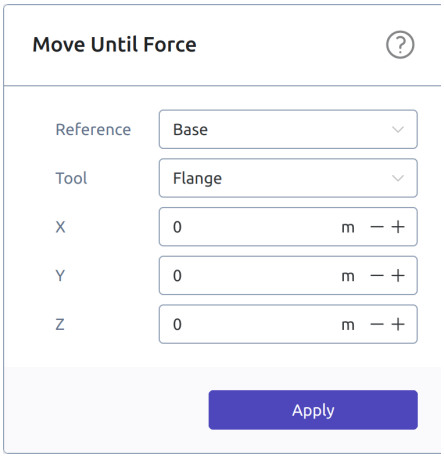
Note: If the error below shows up on the teach pendant, when executing a force block, it means that the manipulator's joint-configuration is near a singularity. Make sure the robot is far from any joint limits, and that no links are parallel, or just try a different configuration of the manipulator.



3.2.17.2 Move Until Force

The 'Move Until Force' block moves the tool in a specified direction until it meets a specified force.



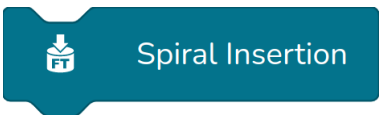


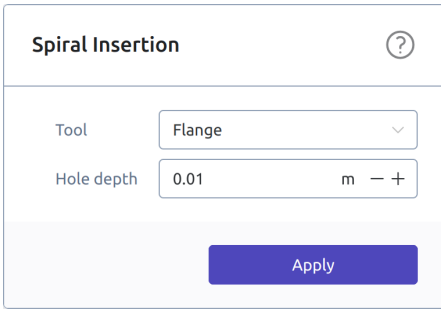
Use the block as follows:

- Specify the direction of movement by specifying a point relative to a reference in the right-hand block-menu.
- Specify the force needed for the manipulator to stop moving.

3.2.17.3 Spiral Insertion

The 'Spiral Insertion' block is used to insert a (preferably) round object into a hole of similar diameter.





The workflow is as follows:

- Create a waypoint that positions the tool such that it points directly at the plane in which the tool is to be inserted. It may be up to 10 cm away from the plane.
- Place the 'Spiral Insertion' block and specify the depth of the hole. This is used to determine if the tool is inside the hole.
- **The 'Spiral Insertion' block will perform the following motion:**
  - Move the tool along positive z-axis until collision with plane.
  - Start a spiral search movement until the hole is found.
  - Move tool into hole.
  - If the hole cannot be found, the tool will go up and out of the hole, and position itself in a slightly different entry position, before throwing an error. You can catch this error using the 'Try-Catch' block

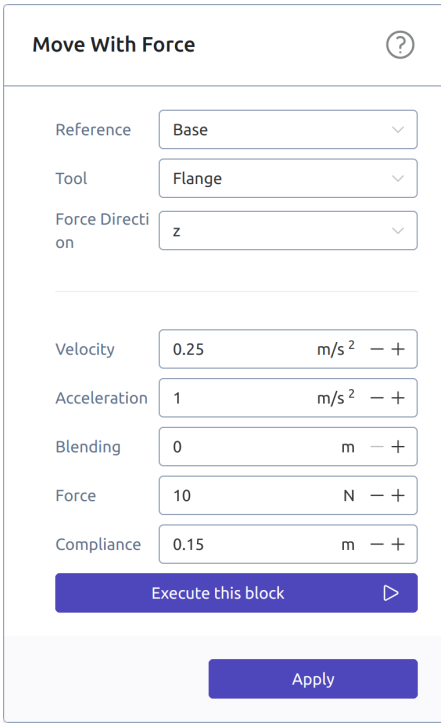
---

Note: The algorithm can only adjust for translational inaccuracies, but not rotational. This means that the tool will not rotate during insertion.

Warning: If dealing with sensitive or brittle objects/surfaces, beware that 'Spiral Insertion' will actuate +2 kg of pressure in peak conditions.

3.2.17.4 Move With Force

The 'Move With Force' block is used to move the tool from one waypoint to another, while maintaining a specified force in a direction.



Note: The 'Move With Force' block will move to the first waypoint with a normal point-to-point movement and then apply force.

The workflow is as follows:

- Set the manipulator in the starting configuration that you want to start applying force. Save this as the first waypoint.
- Move the manipulator into the next configuration and save this as a waypoint in the block. Additional waypoints can be added.

Important: The 'Move With Force' block moves in straight lines between waypoints as default.

The 'Move With Force' block takes the following parameters:

- **Reference**: The reference which the movement should be relative too.
- **Tool**: The tool applied to the manipulator.
- **Force direction**: The direction to apply force, i.e., the manipulator will apply force in the specified direction while moving from waypoint to waypoint.
- **Velocity**: The velocity of the movement in meter per second.
- **Acceleration**: The acceleration of the movement in meter per second squared.
- **Blending**: The blending radius of the movement between waypoints.

- **Force**: The force to apply in the desired force direction.
- **Complicance**: The compliance of the manipulator when applying force. The more the compliance, the more flexible the manipulator will be and vice versa.

3.2.18 Tool Blocks

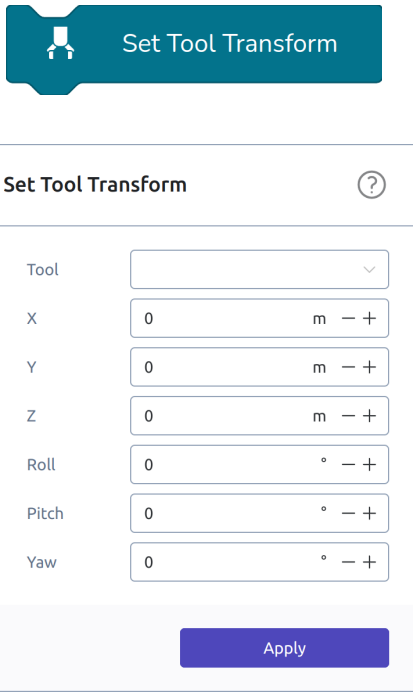
3.2.18.1 Overview

The tool blocks allow the user to set which tool that should be active and to update the parameters of the available tools.

The specification of tools is done in *System* > *Setup*, as described here: *Tool*.

3.2.18.2 Set Tool Transform

The 'Set Tool Transform" block allows for overriding the default tool trasformation for the selected tool. The tool transformation is reset to the default value every time the program is restarted.



The settings of the 'Set Tool Transform' block are:

- **Tool**: The tool for which to update the parameters.
- **X**, **Y**, and **Z**: The position, in meters, of the tool reference (TCP), relative to the flange of the manipulator
- **Roll**, **Pitch**, and **Yaw**: The orientation, in degrees, of the tool reference, relative to the flange of the manipulator. The orientation is specified as ZYX Euler Angles.

3.3 Vision

This section includes guides on how to use vision related functions of the robot.

3.3.1 Calibrate to Marker

3.3.1.1 Overview

The 'Calibrate to Marker' block is used to define local reference frames in the world around the robot. By placing an ER chessboard marker and calibrating to it, the robot is able to estimate the markers 3D pose. Manipulator waypoints can then be created relative to the marker. The ability to calibrate to a marker is essential when the robot is moving between workstations as the mobile base is only accurate within about 5cm.

3.3.1.2 Markers

Ability comes with 3 predefined chessboard markers: CH1, CH3 and CH7. Each marker is made up of a number of black and white squares.

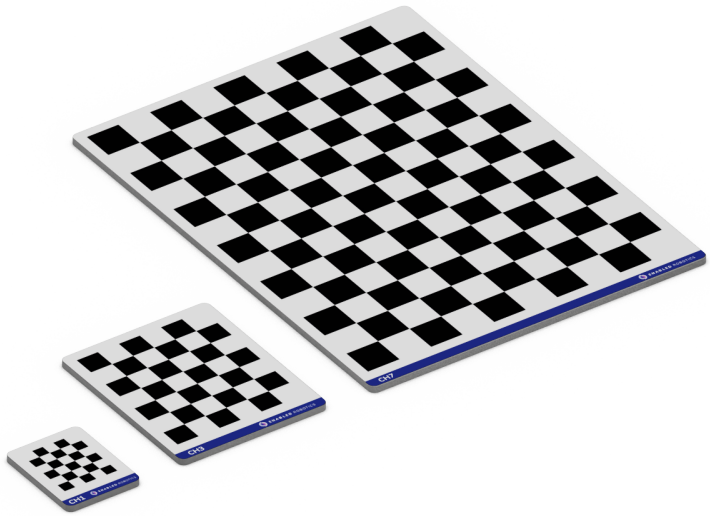


Fig. 3.34: The CH1, CH3 and CH7 markers

Which marker to use is dependent on the use case as they have different properties for certain situations:

- **CH1:** For cases where space is an issue. Will fit on very small surfaces and objects but provide less precision and requires the camera to be closer.

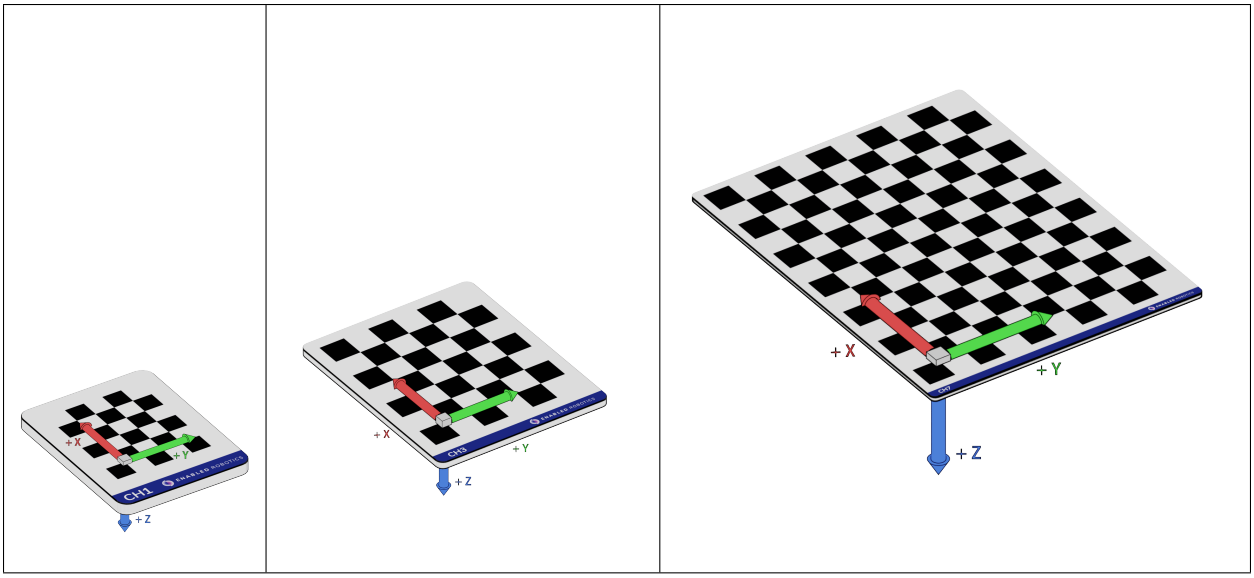
- **CH3:** For general use. Can be placed on most surfaces such as tables and shelves and will provide high precision.
- **CH7:** For initial camera calibration and for cases where precision is very important. Takes up a lot of space but provides very high precision and can be calibrated to from further away.

Marker	CH1	CH3	CH7
Modules	5x4	6x5	12x9
Module size	5mm	10mm	15mm
Physical size	30x35mm	70x80mm	165x210mm
Recommended range	20-30cm	20-100cm	50-150cm

3.3.1.2.1 Reference frame definitions

When calibrating to a marker, the estimated pose of the reference frame is defined in the bottom left saddle point of the marker with the z-axis pointing down as seen below.

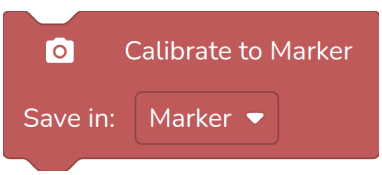
Table 3.1: CH marker reference frames

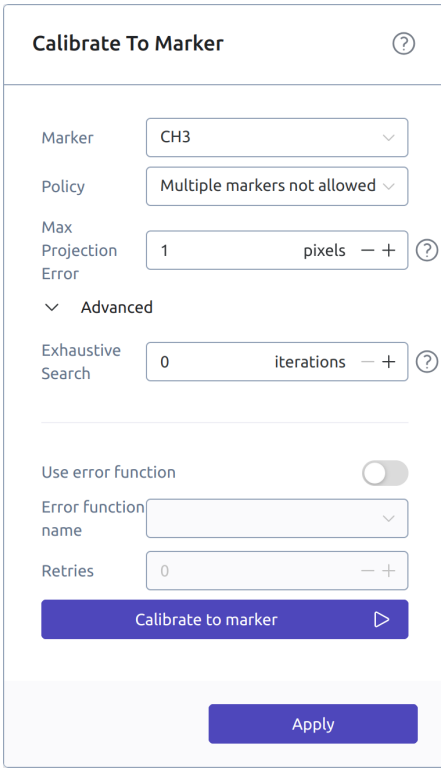


3.3.1.2.2 Custom Markers

Defining markers with a custom module size, number of rows and number of columns can be done using the *ChessboardMarker* Setup Entity. After configuring the Setup Entity, the custom marker will be available in the dropdown in the block menu of the 'Calibrate to Marker' block.

3.3.1.3 Usage

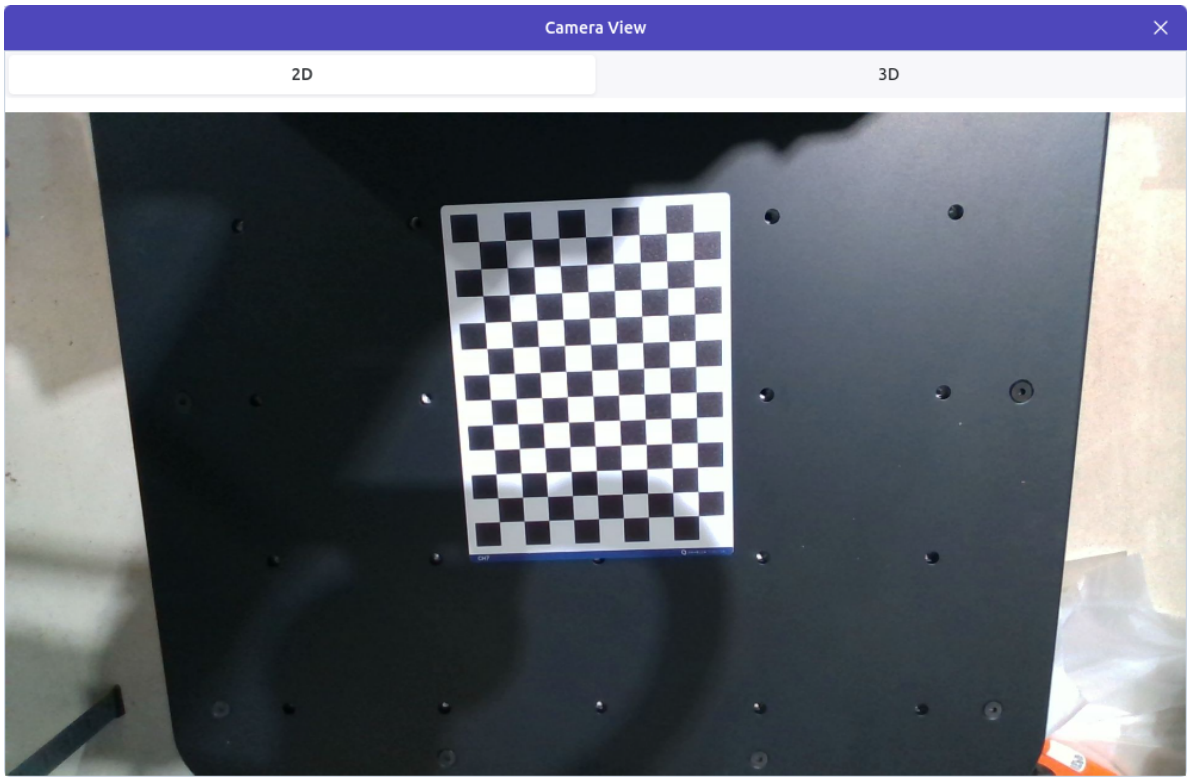




3.3.1.3.1 Initializing the marker

To use the 'Calibrate to Marker' block, we first need to initialize the blocks reference:

1. Make sure the marker is within view of the camera.



2. Go to the block menu and select the type of marker you wish to use.
3. Click 'Calibrate to marker'. The robot will search for markers in the image and if found, save the pose in the blocks reference. If no reference has been selected, a new one will be created. In the bottom of the side menu, detection feedback is shown, including the calculated mean projection error, the name of the reference, and a detection image.

Note: The Calibrate to Marker block also have the fields: **Policy** and **Precision** which can be used to improve the performance of the block.

- **Policy** determines the behaviour of the block if several markers are detected in the image. This can either be:
  - *Multiple markers not allowed*: Causes the block to throw an error if multiple markers are detected.
  - *First marker found*: Selects the first marker found by the detection algorithm. There is no specific pattern to which marker is detected first.
  - *Closest to camera*: Selects the marker closest to the camera in the real world.
  - *Closest to center of image*: Selects the marker closest to the center of image in pixels.
  - *Closest to camera Z-axis*: Selects the marker closest to the camera Z-axis in the real world.
  - *Lowest projection error*: Selects the marker with the lowest projection error.
- **Max projection error** determines the maximum allowed pixel projection error. The expected projection error depends on the marker-size and the distance to the marker. The larger percentage the marker takes up of an image, the larger pixel projection error can be expected. In general, a projection error above 1 is considered a bad calibration and can cause unexpected relative movements.

  If the projection error upon calibration is greater than the threshold selected, the block will throw an error. The error can subsequently be handled with an error function for repositioning the camera view

for a better result.

Tip: Using the *Select first marker* policy speeds up the runtime of the block about 5 times because no validation step is needed to make sure there are not other markers in view. This makes it a great option when it is certain that no other markers will be in view other than the desired one.

Note: The bigger the distance between camera and marker, the lower the precision of the estimated pose will be.

3.3.1.4 Troubleshooting

Failure to detect the marker or poor precision of the estimated pose can be due to several factors:

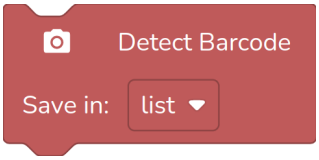
- The marker is too far away / at too steep of an angel
- The marker is damaged or dirty
- The marker is poorly lit
- The marker is warped

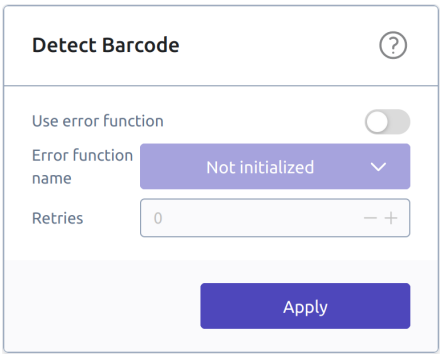
Tip: An easy way to obtain better precision with small markers like the CH1 is to do a 2 step calibration, by using the reference frame from the first calibration to move closer to the marker and do another calibration. This way the marker will fill out more pixels in the camera view and allow the robot to do a more precise calibration.

3.3.2 Detect Barcode

3.3.2.1 Overview

This block is used to detect and read one or more barcodes in the camera view, and save the data in a variable. Supported codes are: EAN/UPC, Code 128, Code 39, Interleaved 2 of 5 and QR Codes.





Note: Codes need to be atleast **40 mil** in resolution for the camera to reliably read them. This goes for both 1D and 2D codes.

3.3.2.2 Usage

To detect a barcode simply place the block in the point of the program where the barcode should be read and input a name for the variable on the block.

The data of the barcode/QR codes is stored as an **array of strings**. Even if only 1 code is detected. Therefore the result needs to be handled as a list. This can be done in several ways depending on the goal of the program.

1. Iterating all codes using “For Each Loop”
2. Reading a specific entry with “Get element”

Note: There is no specific pattern to the order that the barcodes are read and saved.

3.3.2.3 Troubleshooting

If codes are not detected, it could be due either of the following:

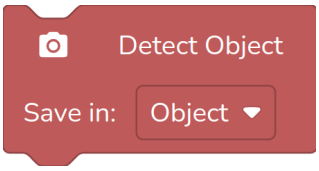
- Code is too small (less than 40 mil resolution)
- Code is too far away (recommended range is 20cm - 50cm for 40 mil codes)
- Code is damaged, warped or dirty.
- Code is of invalid type

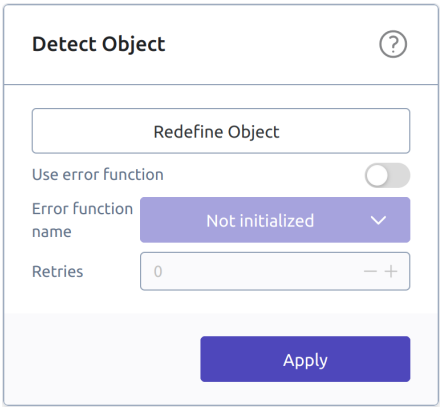
3.3.3 Detect Object

3.3.3.1 Overview

The ‘Detect Object’ block is made for detecting objects in 2D e.g. when picking things off a table. The process used for this is called template matching. Template matching is the process of matching the image of an object (the template) to a query image, and finding the location of the object in the query image. By also calibrating to the surface the object is to be detected on, we are able to determine the location of the object in 3D real world coordinates. Calibrating a reference to a template can be used in many scenarios,

but most often to pick up the template object. This section describes how to use the 'Detect Object' block to do just this.





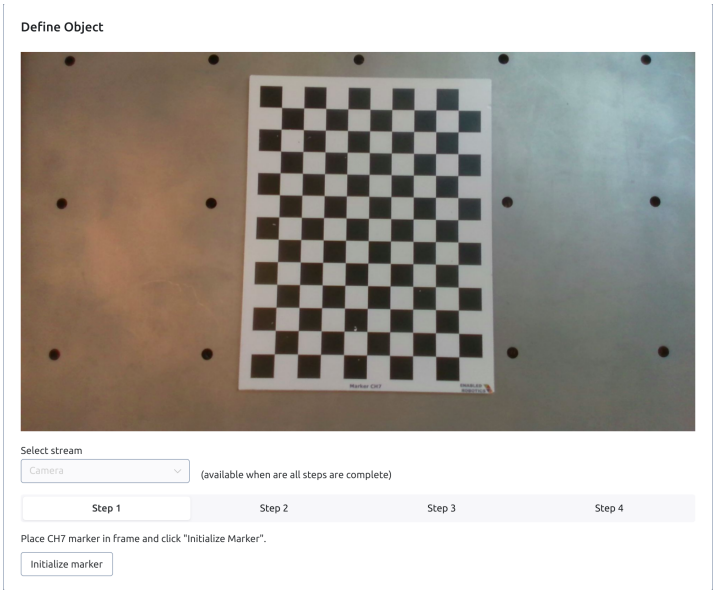
3.3.3.2 Usage

3.3.3.2.1 Defining a template object

Before calibrating to a template object, the object first needs to be defined. To do this, add a 'Detect Object' block in the programming interface and press 'Define Object' in the block menu. Defining an object is a 4 step process:

- **Step 1:** Initialize chessboard.

  To be able to calculate the real world coordinates of the object, we first need to define the plane that the object will be placed in. To do this just place an Enabled Robotics CH7 marker in the camera view and press 'Initialize marker'.



Note: Make sure the marker is entirely visible in the camera view

- **Step 2:** Take image of background.

  To find the template object we need to define the background it will be placed on. To do this, remove the marker and press 'Take image'

- **Step 3:** Take image of object.

  To define the template, place the template object in the middle of the camera view and press 'Take image'. Make sure the only difference between the background image and object image is the presence of the template object. If other objects than the template object have been placed or removed between taking the two images, template generation will likely fail.

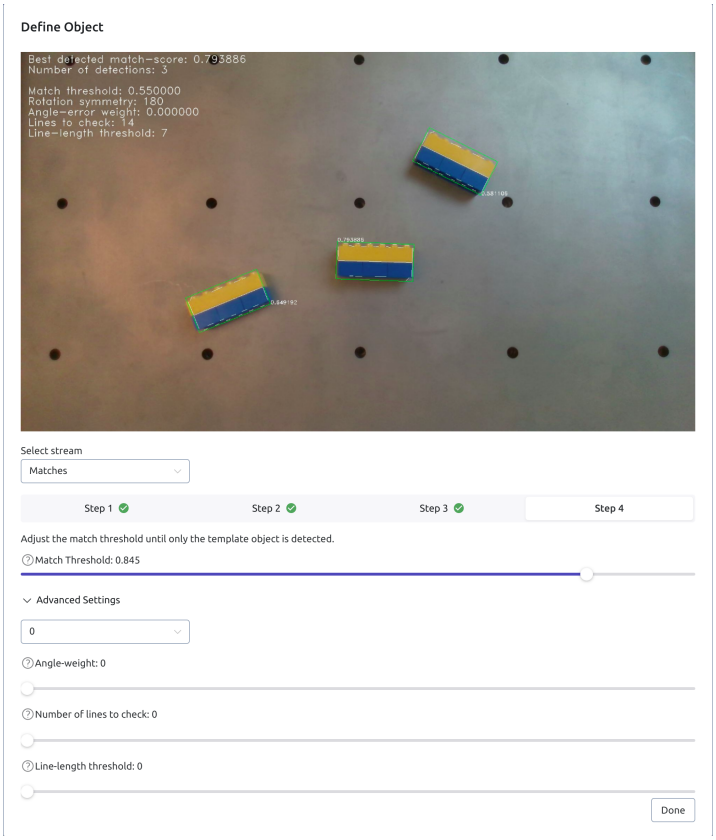
Note: Placing the template object right beneath the camera will yield better matching results

- **Step 4:** Adjust match score threshold.

  Finally the match score threshold has to be adjusted. The match score is a measure of how confident the detection algorithm is, that it found the right object. The threshold works as a cutoff to remove any detection below it.

Important: Setting the threshold too **high** will cause the algorithm to not find the template object even though it is present in the scene. Setting the threshold too **low** will cause the algorithm to detect objects in places where there are none.

  When adjusting the threshold, move the object around or place several objects as seen in `define_object_2`) to make sure the object is also detected in the peripheral of the camera view. Because of perspective change, the algorithm will be less accurate the further the object is from the center of the camera view. When satisfied with the match threshold, press 'Done'.



- **(Optional):** Advanced settings

  Under the advanced settings page, the following matching parameters can be tweaked:

  - **Rotation symmetry**

    This option denotes the rotation symmetry of the object in degrees. A perfectly rectangular object would for instance have 180 degrees symmetry whereas a square object would have 90 degrees symmetry.

  - **Angle-error weight**

    The match score of each detection is based on a weighted sum of error in angle between template and query lines, and error in position between template and query lines. This parameter controls the weight of the angle-error. A weight of 1, translates to 6 degrees angle-error weighing the same as 1 pixel of position-error.

  - **Number of lines to check**

    When matching, the longest lines on the template are considered first. This options denotes how many n'th longest lines should be checked. The more lines, the slower the algorithm.

  - **Line length threshold**

    Controls the threshold of how long lines must be to be taken into consideration. (In pixels).

3.3.3.3 Troubleshooting

A number of problems can occur when using the 'Detect Object' block. To get the best possible performance from the block, consider the following:

1. **Lighting**

   The detection algorithm will work best in well lit environments with relatively constant lighting.

2. **Contrast**

   Make sure the template object has enough color contrast with the background it will be detected on.

3. **Template lines**

   When adjusting the match threshold. Consider how the white template lines on the image appears. Do they follow the contour of the object? Are there large gaps in the contour? If the template lines do not seem very representative of the object, go back and redo step 2 and 3 of the wizard.

4. **Size of template object in camera view**

   The detection algorithm will be less accurate with small objects. If the object appears small in the camera view, consider moving the camera closer.

5. **Perspective change**

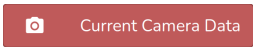
   When moving objects to the peripheral of the camera view, they will look different to the camera because of perspective change. This is especially true with taller objects. This perspective change will cause an error in the center estimation of the detected object. To combat this, make sure the objects to be detected are close to the center of the camera view.

3.3.4 Current Camera Data

3.3.4.1 Overview

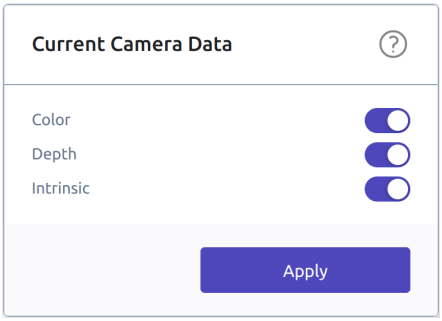
Current Camera Data holds 3 types of information:

1. **Color** A normal image (base64 encoded)
2. **Depth** A pointcloud (base64 encoded)
3. **Intrinsic** from the calibration



3.3.4.2 Usage

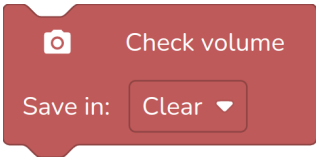
The block menu has 3 toggles for each type of information that should be included in the value.

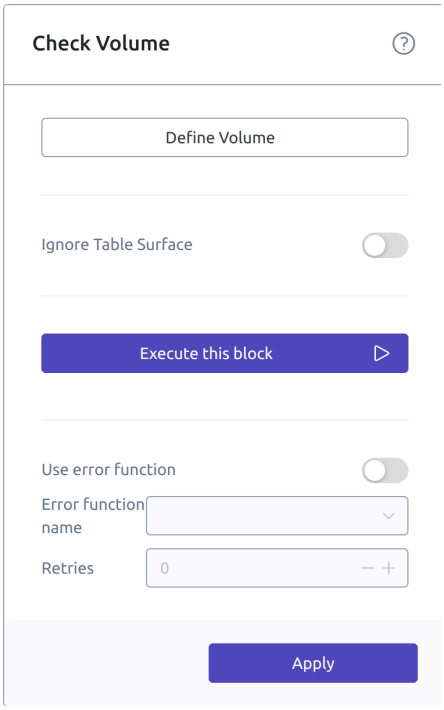


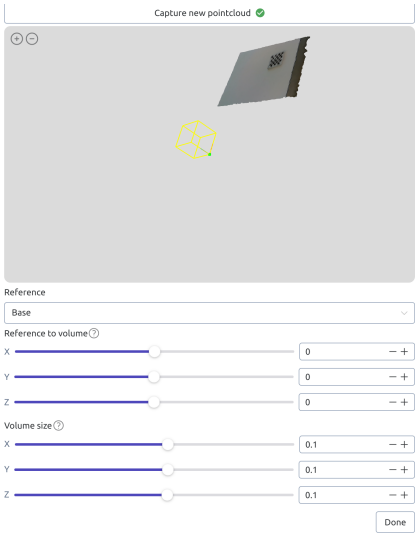
3.3.5 Check Volume

3.3.5.1 Overview

The 'Check Volume' block is used to determine if a volume is clear or not clear, e.g., when placing objects it can be useful to check if the area is clear before placing them. This is done by using the 3D camera to estimate if any object lies within the volume. The 'Save in' variable will be true if the area is clear or false if the area is not clear.







3.3.5.2 Usage

3.3.5.2.1 Defining the volume

- **Step 1:** Reference

  The reference can be either the base of the manipulator or a marker found using *Calibrate to Marker*

- **Step 2:** Reference to volume

  The reference to volume defines the offset from the reference to the volume. Use the sliders to position the volume.

- **Step 3:** Volume size

  The volume size defines the size of the volume in meters.

Note: the check will return true if the volume is not visible in the camera view or the volume is occluded

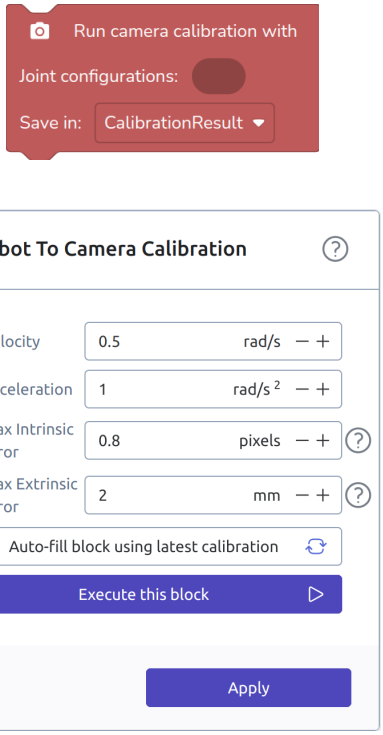
---

In some cases it may be necessary to enable the **Ignore Table Surface** option. If enabled, the most dominant surface (e.g. a table) within the volume is ignored. In this case the volume should be defined to include the surface, otherwise unexpected results may occur.

3.3.6 Run Camera Calibration

3.3.6.1 Overview

This block is used to run a full camera calibration sequence. First it clears the current calibration, then it moves the robot arm to each of the specified joint configurations and add the measurement to a new calibration. Based on the new measurements, the robot-to-camera calibration is updated.



Note: If you want to repeat the latest calibration sequence, use the "Auto-fill block using latest calibration"-button in the block menu which will generate a list of joint configurations corresponding to the ones used in the latest calibration.

3.3.6.2 Usage

To run the camera calibration block, first create a *list* of joint configurations and drag it into the value input of the Run Camera Calibration block. The CH7 marker should be fully visible in all configurations. See *Camera calibration* for more guidance on how to position the camera.

In the side menu of the block, it is possible to specify velocity and acceleration of the robot arm movements. It is also possible to change the maximum allowed intrinsic and extrinsic error. Keeping the default values should in general result in a fair calibration.

If the calibration errors exeed any of the specified error thresholds, the new calibration is discarded and the previous calibration is recovered if it exists. The previous calibration is also recovered if the calibration process is stopped before it finishes (e.g. if the user presses the stop-button or any execution error occurs).

3.3.6.3 Troubleshooting

If the camera calibration fails, it could be due to either of the following:

- One of the joint configurations is not reachable.
- The marker is not properly detected in one of the views.
- The calculated intrinsic and/or extrinsic errors exeeds the specified thresholds.
- Any other issue as specified in *Camera calibration*.

3.4 I/O

3.4.1 Overview

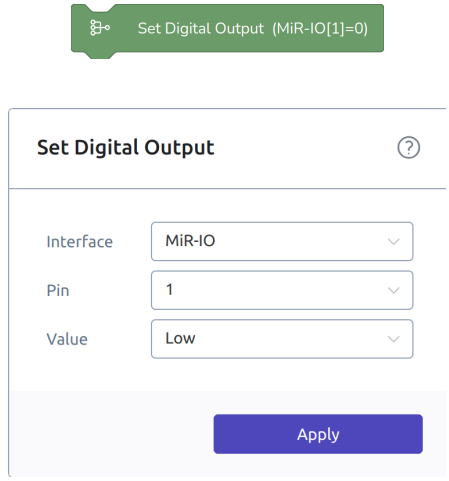
The I/O category contains blocks for setting and reading I/O functions on the UR and MiR, both digital and analog. Each block has a dropdown to select between the UR and the MiR I/O interface. When selecting the UR, it is the physical I/O ports on the UR controller that are set. When selecting the MiR, it is the internal PLC registers that are set.

Table 3.2: Available I/O

I/O	Digital Input	Digital Output	Analog Input	Analog Output
UR	CI0->CI7, DI0->DI7, CO0->CO7, DO0->DO7, TO0, TO1	CO0->CO7, DO0->DO7, TO0, TO1	AI0, AI1, AI2, AI3, AO0, AO1	AO0, AO1
MiR	PLC Registers 1-100	PLC Registers 1-100	PLC Registers 1-100	PLC Registers 1-100

3.4.2 Setting I/O values

The blocks in the I/O category are of two types: instruction blocks for setting an I/O value and value blocks for getting an I/O value.



The block menus for the two “Set” instruction blocks are similar, they have three fields:

1. The **Interface** field is for picking between the UR and MiR I/O interface.
2. The **Pin** field is for picking the specific output pin the instruction should set, this list is generated based on what interface has been chosen to reflect the pins available on the interface.
3. The **Value** field defines what the pin should be set to. For the ‘Set Digital Output’ block this will be a dropdown with either true or false and for the ‘Set Analog Output’ block it is a number (double) between 0 and 1.

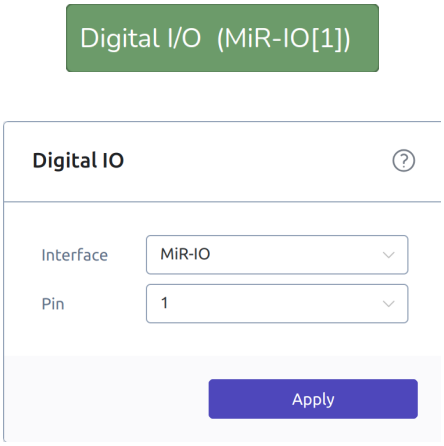
Important: UR analog I/O value ranges: On the UR teachpendant, the analog input and output value domains can be switched between voltage (0->10V) and current (4->20mA). No matter which domain is selected, the value range on the I/O blocks in Ability are always 0->1. This means setting analog I/O to

0 correpsonds to either 0V or 4mA depending on the selected domain. Similarly setting analog I/O to 1 corresponds to either 10V or 20mA. Values for each domain are interpolated between 0 and 1 (0.5 -> 5V or 12mA). Getting analog I/O returns values in a similar way.

Note: Setting a MiR PLC register to a digital true or false will simply set it to 1 or 0. Similarly, when getting the value using "Get digital I/O" the returned value will be false if the register is 0 and true for all other values.

3.4.3 Getting values

The two value blocks are for reading the values of either an analog or digital, that data can then be saved in a variable or compared to another value.



The block menus for the two "Get" value blocks are similar, they have two fields:

1. The **Interface** field is for picking what IO hardware the value is supposed to read from.
2. The **Pin** field is for picking the specific I/O pin the value should read, this list is generated based on what interface has been chosen to reflect the pins available on the interface.

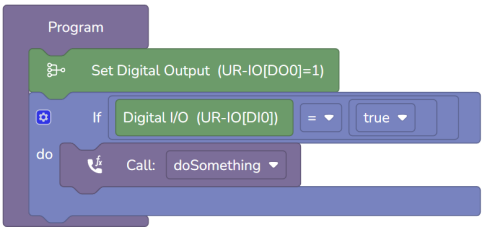
The return *Value types* are:

- Get digital I/O -> **Boolean**
- Get analog I/O -> **Double** (a value between 0 and 1)

3.4.4 Examples

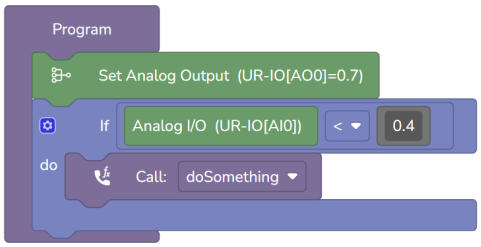
3.4.4.1 Example 1: Setting and getting digital I/O

This example sets digital output 0 on the UR to high and then checks if digital input 0 is high.



3.4.4.2 Example 2: Setting and getting analog I/O

This example sets analog output 0 on the UR to 0.7 (7V or 15.2mA) and then checks if analog input 0 is less than 0.4 (4V or 10.4mA).



3.5 Loops

3.5.1 Overview

The loop category contains instruction blocks that allow the program to:

1. Repeat instructions a certain number of times.
2. Execute the same instructions until a certain condition is met.
3. Loop through a list of elements.

3.5.2 Usage

The pattern for using the Loop instruction blocks are the same. All blocks defined inside the statement input for the loop will be run as long as the condition for looping is met.

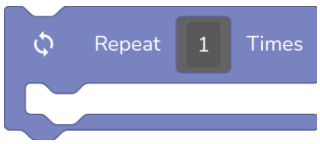


Fig. 3.35: The 'Repeat' block takes a number as value input

The 'Repeat X Times' block repeats the code defined in the statement input the number of times defined in the value input. The value can either be a fixed number or a variable that is set somewhere else in the program.

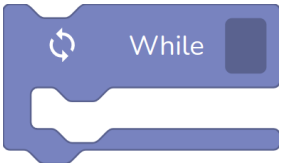


Fig. 3.36: The 'While' block takes a boolean expression as value input

The 'While' instruction will keep looping while the input value is true. Common usages for this block are repeating a task several times until an exit criteria is met. For more advanced usages of the block, it can be used in conjunction with the try-catch block to build error handling patterns, see the *Programming* section about the try-catch block.

Fig. 3.37: The 'For Each' block iterates the **list** variable and returns each element in the **element** variable

The "For Each" instruction is a useful block when looping through a list of items, e.g. a list of barcodes obtained by the *Detect Barcode* block, an array of data from a *OPC-UA Client* block or a user created list.

3.6 Logic

3.6.1 If block

The 'If' block is used when a program needs a branching condition, this means that whenever the robot needs to make a decision during a task, the 'If' block can be used to compare values and make that decision.

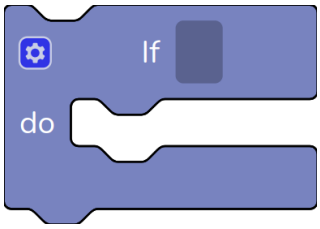


Fig. 3.38: The 'If' block takes a boolean variable as value input

The 'If' block is mutable which means that it is possible to modify the block using the cog on the block itself. Clicking the cog allows for adding multiple "else if" and a single "else" instruction input.

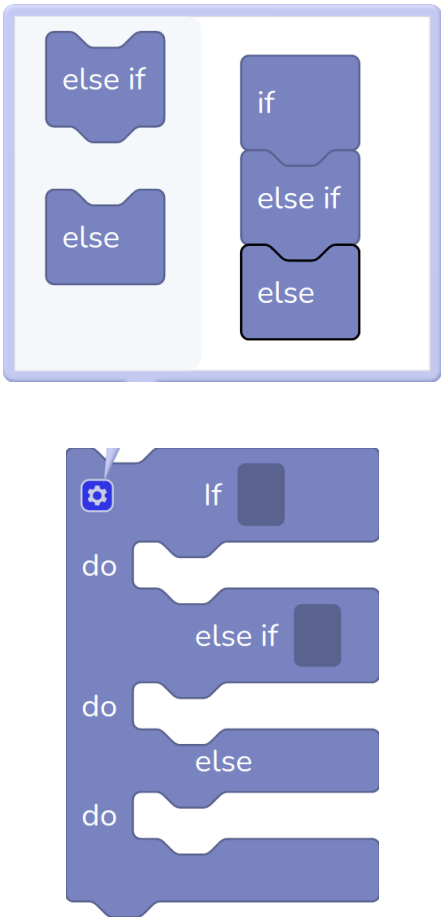
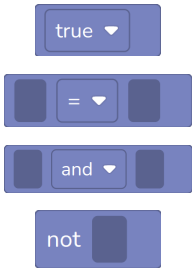


Fig. 3.39: Adding an ‘else if’ and an ‘else’ clause to the ‘If’ block

3.6.2 Logic values

The value blocks in the logic category are the ‘Boolean value’, the ‘Comparator’, the ‘Logical operator’ and the ‘Not’ block. All of these value blocks return a boolean value and can be used with the ‘If’ block.



3.7 (x) Values

This section includes documentation for different value types including basic types, lists and dictionaries.

3.7.1 (x) Basic values

The value category is filled with basic value types sorted into 4 groups:

1. Primitive value blocks
2. Computed value blocks
3. Get/Set value blocks
4. Save/Load value blocks

3.7.1.1 Primitive value blocks

There are two primitive value blocks, string and number. Each one is a simple hard-coded value.

3.7.1.2 Computed value blocks

Fig. 3.40: Calculated value

Fig. 3.41: Inverse value

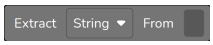


Fig. 3.42: Extracted value

Fig. 3.43: String replace

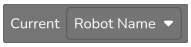


Fig. 3.44: Current system value

Calculated value performs one of 4 mathematical operation (addition, subtraction, division, multiplication) on 2 numbers.

Inverse value Gives the inverse of a transform

Extracted value Casts to a value selected in the dropdown

String replace Performs a Search-Replace operation on a string

Current system value returns the current nickname or IP address of the robot

3.7.1.3 Get/Set variable blocks

The Get/Set blocks stores a value in a variable or pulls the value of a variable. Variables are cleared whenever a program is loaded.

3.7.1.4 Save/Load variable blocks

The Save block saves a variable and it's value to disk.

The Load block retrieves a variable and it's value from disk. An optional default value can be specified.

The Save/Load blocks are useful for having persistant data across program runs, such as counters. As default the persistent variable is only accessible by the same program. If it's needed to share data between programs, select "global" in the dropdown on the block.

3.7.2 (x) Lists

3.7.2.1 The List Block

The List block acts like a value block which defines a list. The block can be mutated by clicking the blue gear icon to change the size of the list. Elements can be deleted by clicking the trash icon.

Fig. 3.45: Empty list

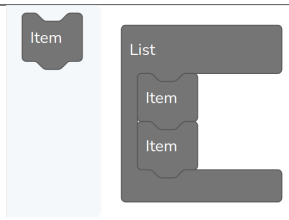


Fig. 3.46: List mutator

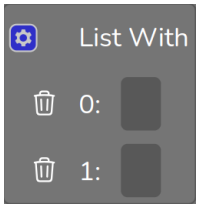


Fig. 3.47: List with 2 elements

3.7.2.2 The Set List Block

The Set List block is identical to other set value blocks. It assigns a list to a variable.

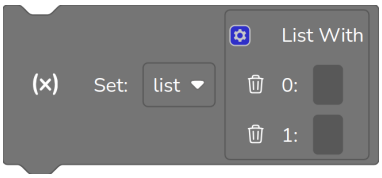


Fig. 3.48: Set List block

3.7.2.3 Get Element From List

The Get Element block gets a specific element in a list by it's index. List are zero-indexed. An error is raised if the index is out of range.

Fig. 3.49: Get Element From List block

3.7.2.4 Add Element to List

The Add Element block appends an element to the end of a list.

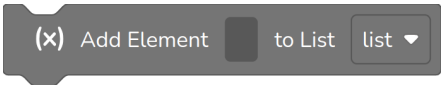


Fig. 3.50: Add Element to List block

3.7.2.5 Length of List

The "Length of List" value block returns the number of items in the list. This is useful in a for-loop to iterate all items in a list.

Fig. 3.51: Get Length of List

3.7.3 (x) Dictionaries

3.7.3.1 The Dictionary Block

The list block acts like a value block which defines a dictionary. The block can be mutated by clicking the blue gear icon to change the size of the dictionary. Key-Value pairs can be deleted by clicking the trash icon.

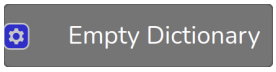


Fig. 3.52: Empty Dictionary

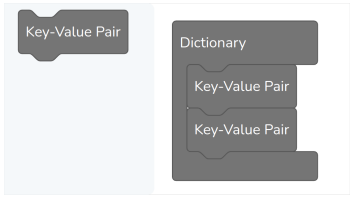


Fig. 3.53: Dictionary mutator

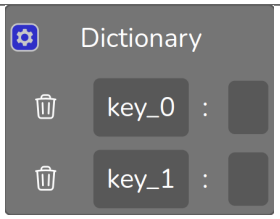


Fig. 3.54: Dictionary with 2 elements

3.7.3.2 The Set Dictionary Block

The Set Dictionary block is identical to other set value blocks. It assigns a dictionary to a variable.

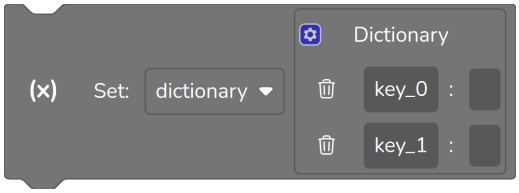


Fig. 3.55: Set dictionary block

3.7.3.3 Get Element From Dictionary

The Get Element block gets a specific element in a dictionary by it's key. An error is raised if the key doesn't exist.

Fig. 3.56: Get Element From Dictionary block

3.7.3.4 Set Key in Dictionary

The Set Key block can be used to add a new key-value pair to a dictionary. The value is overwritten if the key already exist.

Fig. 3.57: Set Key-Value Pair in Dictionary

3.7.3.5 Is Key in Dictionary

The “Is key in Dictionary” value block can be used to check whether a specific key is in the dictionary.

Fig. 3.58: Is Key in Dictionary block

3.7.3.6 Get Keys in Dictionary

The “Get Keys in Dictionary” value block returns a list of keys in the dictionary. This is useful in combination which the “For each” block, if it is desired to loop through all key-value pairs of a dictionary.

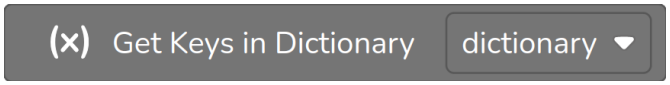


Fig. 3.59: Get Keys in Dictionary block

3.7.3.7 Create Dictionary From String

The ‘Create Dictionary From String’ block creates a dictionary from a string. The default behaviour of this block creates a dictionary variable named “dictionary” from a string variable named “string” where semi-colon is interpreted as line-breaks and comma is interpreted as key-value seperator.

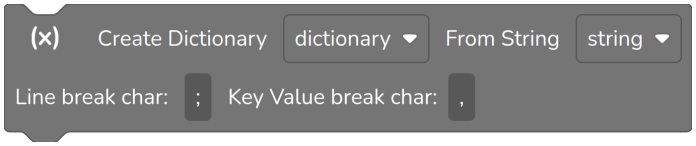


Fig. 3.60: Create Cictionary From String block

3.7.3.8 Export dictionary using the Data Grabber module

If the Data Grabber module is activated, the Dictionary category will also contain the “Save dictionary as json” block. This is useful for collecting sequences of data for later use.

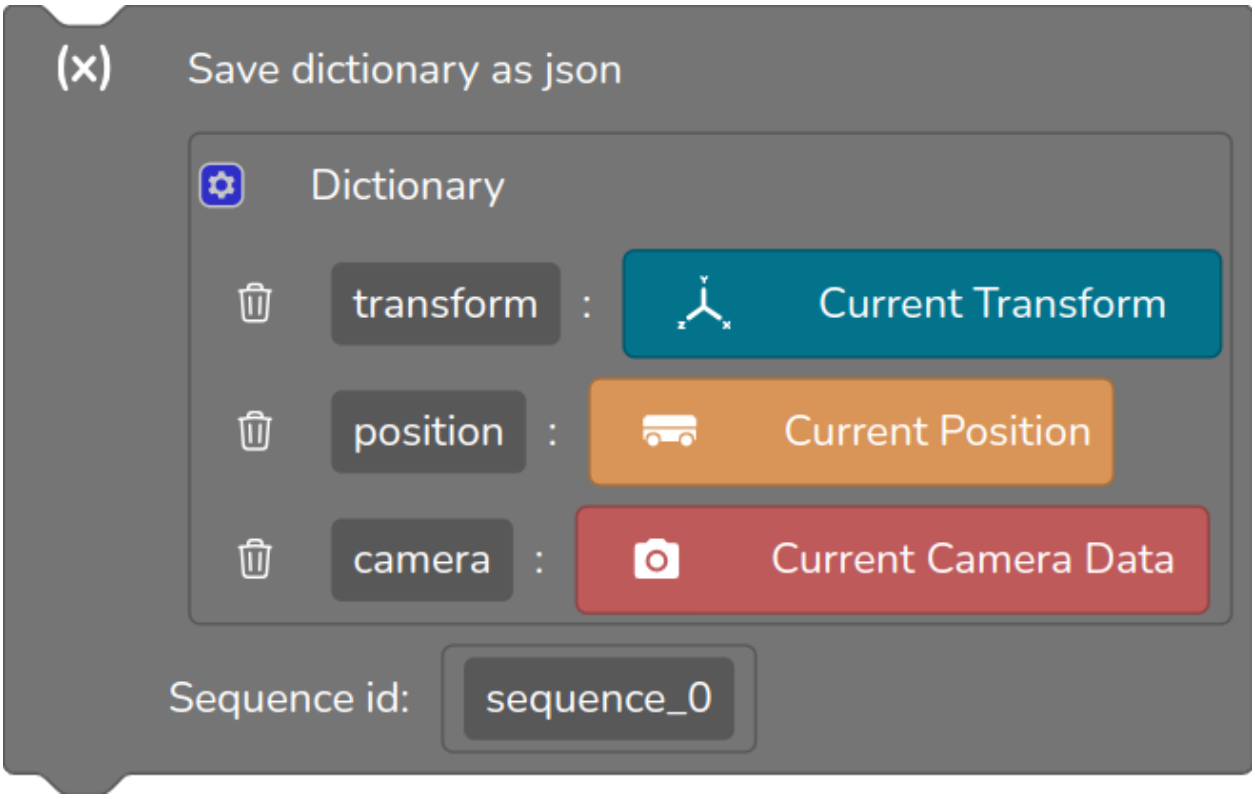


Fig. 3.61: The “Save dictionary as json” block with a template dictionary of robot data

The “Save dictionary as json” block exports a specified dictionary to json data and stores it to the current program folder. The data is tagged by the program name, sequence id and a timestamp, and can be downloaded later from the “Download Data” page.

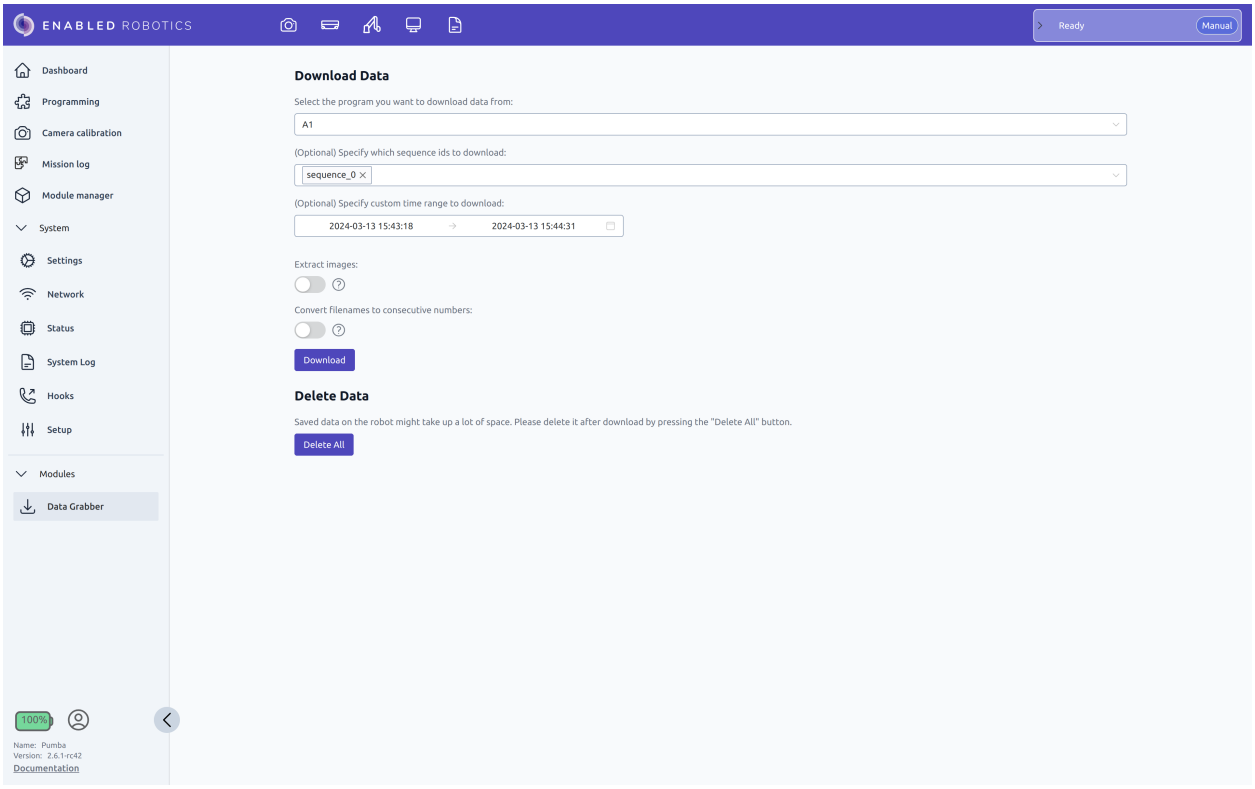


Fig. 3.62: Using the default options, all data from a specific program is downloaded as raw json files

In order to download the captured data, select the desired program name in the dropdown list and hit “Download”. Using the default options, all captured data will be downloaded as raw json files which will be organized in sub-folders named after the sequence ids. Please note, if images are added to the exported data, it may take some time to open the json files, since the images are stored as base64 strings.

There is a couple of options to consider to adjust the downloaded data according to your needs:

- **Sequence ids**: can be changed so only specific sequence ids are downloaded.
- **Time range**: can be changed to download data from a specific time period. The default values are the first and last captured data timestamps respectively.
- **Extract images**: if enabled, the key “camera” will be interpreted as a “Current Camera Data” value, and images will be stored as images-files in separate folders.
- **Convert filename to consecutive numbers**: if enabled, the output filenames will be converted to consecutive numbers (“000000”, “000001”, “000002” etc).

Important: Please remember to press **Delete Data** when all desired data has been downloaded.

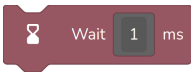
3.8 Utility

3.8.1 Overview

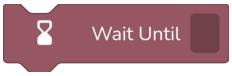
The utility category has multiple utility blocks for different purposes.

The PopUp Message block creates a pop up in the user interface with two buttons “Okay” and “Cancel”. The users’ response will then be saved in the result variable as a boolean, “Cancel” storing as false and “Okay” storing as true.

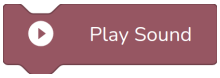
The Wait ms block waits for the set number of ms.



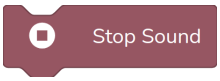
The Wait Until block waits till some boolean expression returns true, this could be an io port or the time being some specific value.



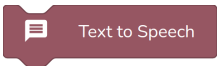
The Play Sound block will play a sound. It is possible to choose between the sounds currently available on the MiR. New sounds can be added through the MiR interface. It is possible to change the volume, and play the sound in different modes: full length, looping, or with a custom duration. The block is non-blocking so other blocks can be executed while a sound is playing.



The Stop Sound block will stop any playing sound.



The Text to Speech block synthesizes written text to sound that is played on the robot. It is possible to choose between different voices and to adjust the volume. The block is non-blocking so other blocks can be executed while the robot is speaking.



The Battery % value block returns state of charge of the battery .

3.9 </> Programming

The programming category contains blocks for achieving programming functionality commonly found in regular programming languages.

3.9.1 Function

This block creates a function that can be called from the *Call function* block or the *Call Program Function* block.

The special 'Program' function acts as the entrypoint for a program. In regular programming languages this is often called the 'main function'

Fig. 3.63: The function block

The special 'Program' function acts as the entrypoint for a program. In regular programming languages this is often called the 'main function'

Fig. 3.64: The function block

3.9.2 Run Script

The 'Run script' block can execute small snippets of Python code.

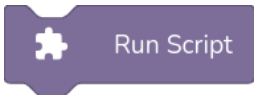


Fig. 3.65: Run Script block

The execution environment is very limited, core Python features are removed such as `print` and `import`. The execution environment has a predefined variable `knowledgebase` which can be used to interract with the variables defined in the programmring workspace.

Only primitive variables of the following type can be set and read:

- Bool
- Int
- Double
- String

The following code snippet illustrates the usage:

```
i = knowledgebase.get_int("my_integer")
i += 10
knowledgebase.set_int("my_integer", i)
```

Attempts to Get a variable of a wrong type, or any syntax error in the snippet, will result in unsuccessful execution and put the robot in an error state.

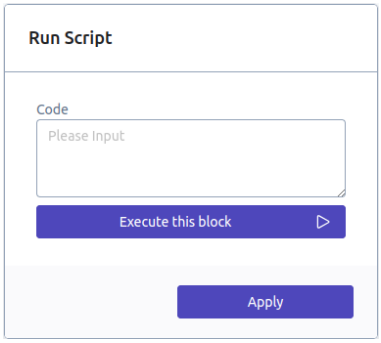


Fig. 3.66: Run Script menu

Note: The 'Run script' block uses Python version 3.8.10

3.9.3 Call function

This block executes the selected function. This is usefull to reduce duplicated blocks.

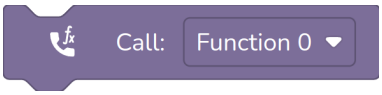


Fig. 3.67: The call block

3.9.4 Call Program Function

This block executes a function defined in a different program. The block has a field to select between included programs. When a program is selected, all functions defined inside this program will be available to select from the other dropdown.

Splitting up a program in multiple sub-programs is useful if functionality is shared across programs.

Note: The block menu in the main 'Program' block defines which other programs to load when executing a program. See *Including programs* for more information.

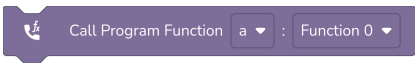
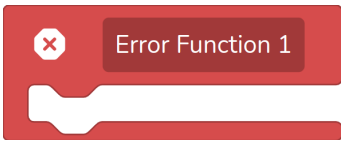


Fig. 3.68: The call program function block

3.9.5 Error Function

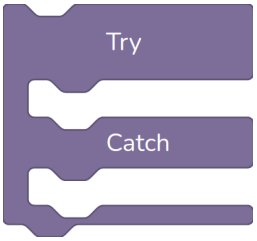
This block defines a function that works as an error handler. From the block menu of any instruction block, you can configure the block to use an error handling function if execution fails. This can for example be used

when calibrating to a marker.



3.9.6 Try Catch

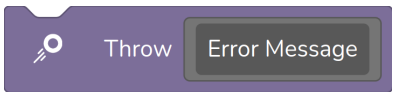
This block attempts execution of blocks placed inside the “Try” section. If any of the blocks throw an error, the “catch” section of the block is executed.



Note: If the block in the try section has an error function associated with it. This error function is called instead.

3.9.7 Throw

This block can raise an error that can be caught in a try catch statement. If the error is not caught, execution is stopped and the robot is put in an error state, showing the error message defined on the block.



3.9.8 Exit

This block stops execution of the current program and mission. If the mission contains more programs, those will not be started. Hitting an exit block is considered a successful finish of execution and the robot is ready to execute the next mission.

3.10 Communication

This section includes guides on how to use communication related functions of the robot

3.10.1 OPC-UA Client

3.10.1.1 Overview

OPC-UA is an industrial client-server communication protocol. OPC-UA is made for both data handling and control of systems such as machines and robots. Ability hosts an OPC-UA Client which can be accessed through a number of blocks. The client can be used to access variables and call methods on other OPC-UA servers.

3.10.1.2 OPC-UA Client Usage

The OPC-UA client is used through the three communication blocks in the web interface: 'Read variable', 'Write variable' and 'Call method'.

3.10.1.2.1 Read Value

The 'Read Value' block will read the value attribute of a given node and save it to a variable in the web-interface. The block needs the endpoint of the server and the address (namespace index and node id) of the node. The block can be executed from the block menu, in this case the result will be displayed underneath the 'Execute' button.

Fig. 3.69: Using the Read Value Attribute block to read the robots own program state.

3.10.1.2.2 Write Value

The 'Write Value' block will write to the value attribute of a given node. The block needs the endpoint of the server, the address (namespace index and node id) of the node as well as the type and value that should be written to the node. The block can handle arrays of all datatypes and dimensions. These should be input using square brackets eg: `[[1,2],[3,4]]` for a two dimensional array of integers or `["one","two","three","four"]` for a one dimensional array of strings. The block can also be executed from the block menu.

Fig. 3.70: Using the Write Value Attribute block to write an integer.

3.10.1.2.3 Call Method

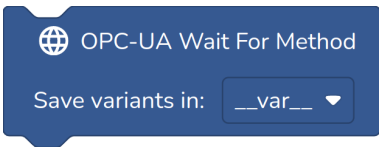
The 'Call Method' block needs both the address of the object that contains the method and the address of the method itself. The method needs to be called with the arguments in the right order and type (An error

will be displayed underneath the 'Execute' button if done incorrectly). As with the 'Write Value' block, to pass an array as argument, square brackets are used.

3.10.1.2.4 Wait For Method

The 'Wait For Method' adds a new method to the OPC-UA server. The signature of the can be customised in the block menu. The returned value is a dictionary with each entered value in the signature. The keys are 'arg1', 'arg2', 'arg3'… for each argument.

Note: The method is removed from the OPC-UA interface after being called once.



3.10.2 REST Communication

3.10.2.1 Overview

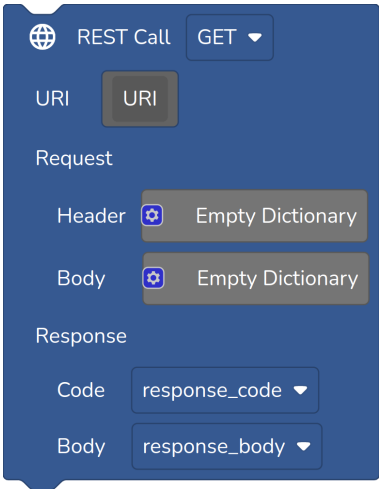
The Ability software offers two ways of interacting with the robot using REST: #. The **REST Interface** is used for communication outside program execution, e.g. loading and starting programs. #. the **REST Communication blocks** are used for REST communication during program execution, e.g. for handling data exchange of orders and tasks between the robot and a centralized system such as a WMS or ERP system.

This part of the documentation describes the use of the REST Communication blocks. For a description of the REST Interface see here: *REST Interface V2*.

Note: REST Communication is a secondary module and needs to be enabled in the Module Manager before it can be used.

3.10.2.2 REST Call

The 'REST Call' block is used for making calls to a REST resource.



The input fields are:

- URI: the full identifier for the resource, including port.
- Header: a dictionary with headers to add to the call.
- Body: the 'payload' of the call.

The returned values are:

- Code: the response code from the server.
- Body: the response body if any.

The data is a list of dictionaries, and will have to be parsed using the *Lists* and *Dictionaries* blocks.

3.10.2.3 REST Receive

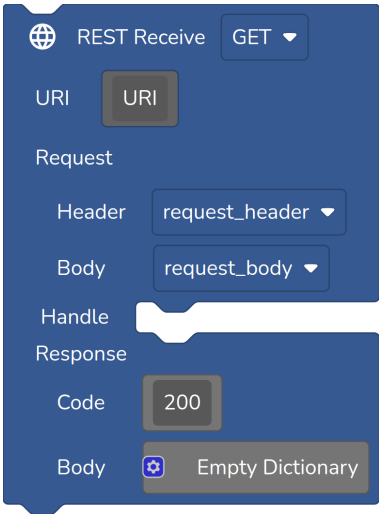
The 'REST Receive' block spins up a temporary rest server and waits for a call.

To set up the block, enter the desired endpoint on the block in the format: `http://<robot-ip>:<port>/<path>`.

Note: The port 8082 cannot be used as it is already occupied by the *REST Interface V2* running on the robot. For a full list of occupied ports, see: *Ports*.

The "handle" section on the block defines what should happen when the endpoint is called.

A timeout period needs to be set in the block menu of the 'REST Receive' block. Setting it to -1 will make the block wait indefinitely. If the block times out, it will throw an error.



3.10.2.4 Troubleshooting

- **Connection**: Make sure that the robot has a connection to the client/server that is to be communicated to. This can be tested using a simple ping command from the client or server.
- **Ports**: Make sure that the ports used are open on the network. Keep in mind that 8082 is used for the *REST Interface V2*, and therefore cannot be used to set up endpoints.
- **Data parsing**: If an error is thrown during parsing, it is likely because the data is parsed incorrectly. E.g. a list is assumed to be a dictionary or vice versa. To inspect the formatting of the data, a tool such as Postman[1] can be used.
- **http and https**: While the 'REST Call' block supports both http and https, the 'REST Receive' block only supports creating endpoints with http.

3.10.3 Websocket Client

3.10.3.1 Overview

The Websocket Client module can be used to comminicate to external systems through a websocket. Robot data such as current position, transform values, and camera data can be sent to the external system through a websocket request. The response can be passed to a variable and then as Waypoints in Manipulator and Mobile blocks.

3.10.3.2 Websocket Client Usage

The Websocket client has one block for calling an external ROS service exposed through Rosbridge. In order to call an external ROS service, the external system needs to run a Rosbridge server on the same network where the service is exposed.

A tutorial for setting up a Rosbridge server can be found here: https://wiki.ros.org/rosbridge_suite/Tutorials/RunningRosbridge

[1] https://www.postman.com/

Note: The Websocket Client module is disabled by default. It can be activated from the **Module Manager** page.

3.10.3.2.1 Call ROS Service

The following parameters has to be defined:

- **URI**: the websocket address exposed by the external Rosbridge server.
- **Service**: the name of the ROS service you want to call.
- **Args**: a dictionary of key-value pairs containing the arguments required by the ROS service. If it takes no arguments, leave the input field empty.
- **Response**: the variable where the service response is saved to.

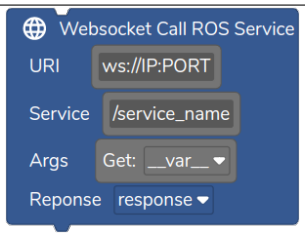


Fig. 3.71: The block has tree input fields: URI, service name and arguments, and one output-field where the response is stored.

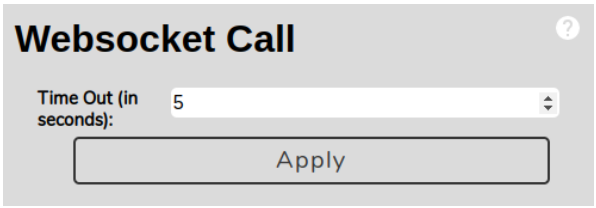


Fig. 3.72: The default timeout of 5 seconds can be changed in the block menu.

3.10.3.2.2 Example program

Fig. 3.73 shows an example of how to call an external system using the "Websocket Call ROS Service" block. In this example, the external system exposes a ROS service called */estimate_object_pose*. The service takes an image and current camera position as inputs and returns a transform.

The program is built of six blocks:

1. Move to capture position of object
2. Create a dictionary *current_data* which contains *Current Camera Data* and *Current Transform* from *Base* to *Camera*.
3. Call the external ROS service */estimate_object_pose* with *current_data* as request arguments and save response to *response* variable.
4. Set the *grab_object_position* to the *transform* value from the service response.
5. Convert the *grab_object_position* to "xyzrpy" format (so it can be passed as a Waypoint to a MovePTP block).
6. Finally, move to the desired position, calculated by the external system.

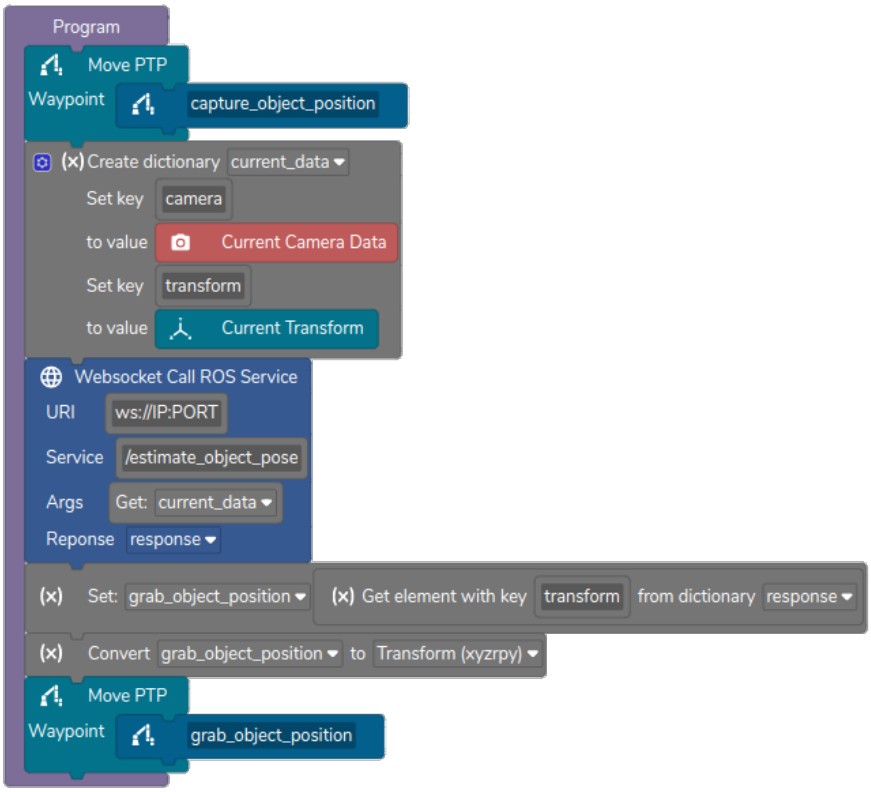


Fig. 3.73: Example program to use an external system for pose estimation

CHAPTER

FOUR

SYSTEM

This section describes the functionality of the System menu

4.1 Users

4.1.1 Setting up users

The credentials used to access the Ability web interface can be modified in the *USERS* menu.

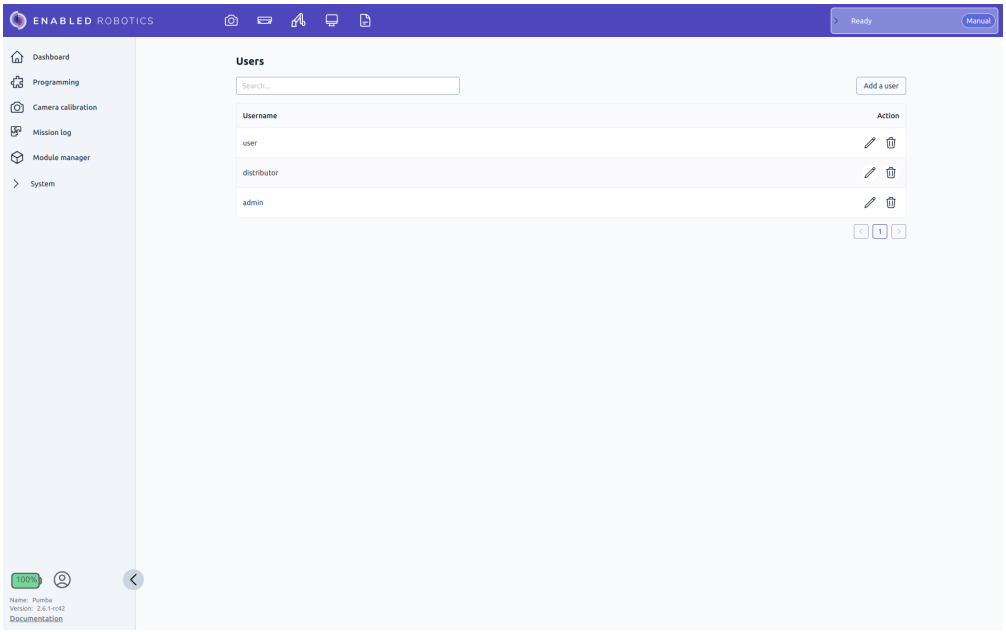


Fig. 4.1: Users page

From the users settings page it is possible to create new users and modify existing users.

When creating or modifying a user, just fill out the fields as shown in Fig. 4.2.

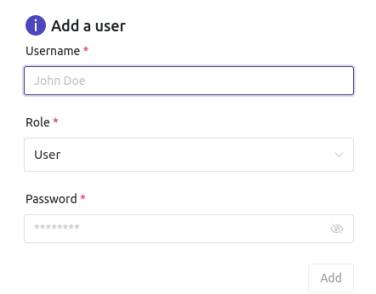


Fig. 4.2: Add user dialog box

4.2 Settings

The settings page holds general settings about the robot split into different tabs.

4.2.1 Robot

Robot nickname

On the robot tab it is possible to set a nickname for the robot. The nickname will be displayed in the bottom left corner of the webinterface.

If your device and network supports zeroconf you can also enable the visible toggle. When visible is enabled the robot will respond to “<robotname>.local”

Workcell settings

On the robot tab it is also possible to configure the physical properties of the robot:

- Top module configures the size of the front module. This is used for path planning.
- Mobile platform type is used both for path planning and starting a compatible driver.
- Arm type is used both for path planning and starting a compatible driver.
- Arm position is used for path planning.

Safety setting

Under the safety setting, it is possible to configure the blocking of the mobile device when the arm is not in the safe home. This means that when the blockage is enabled the execution of the mobile device blocks is not allowed when the arm is not in the safe home position. This also affects the use of the mobile device joystick. If the blockage is disabled, the mobile device blocks can be executed no matter if the arm is in the safe home position or not.

Note: This does not configure the safety setup on the PLC, but only changes if the mobile device blocks are allowed to be used if the arm is not in a safe home position.

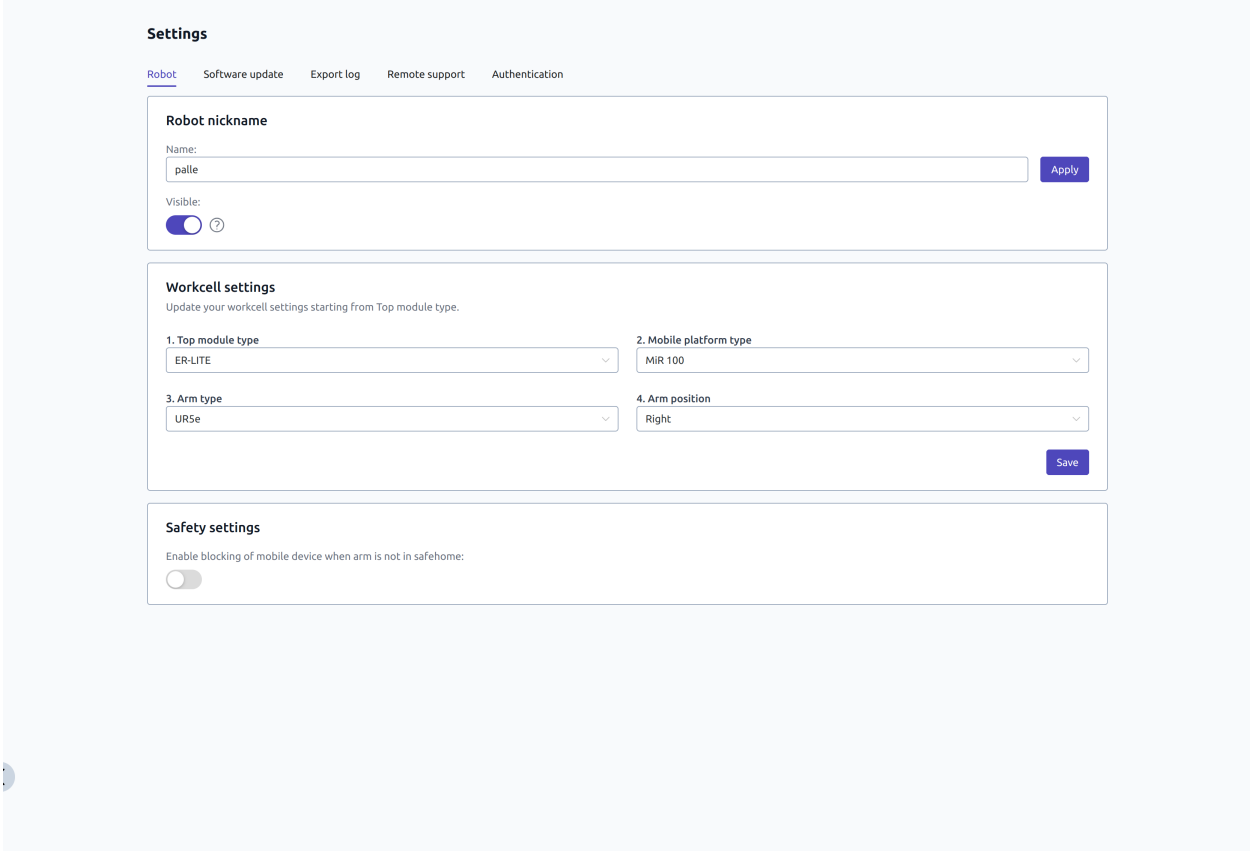


Fig. 4.3: Robot settings

4.2.2 Software Update

Updating the robot is a 2 step process.

1. Downloading the new release

Note: Downloading the update requires that the robot is connected to the internet. Connecting to a WiFi hotspot is described in the *Network* section.

In the “Cloud window” select the software version you wish to download. Click refresh to get a new list of available software versions. Click “Download” to download the selected version.

Once the download is complete, select the software version in the “Software update” window, then click “Apply”. Restart the robot by turning it off and on again as described in the manual.

Note: The Software Releases Forum on enabled-robotics.com has information on the latest software versions.

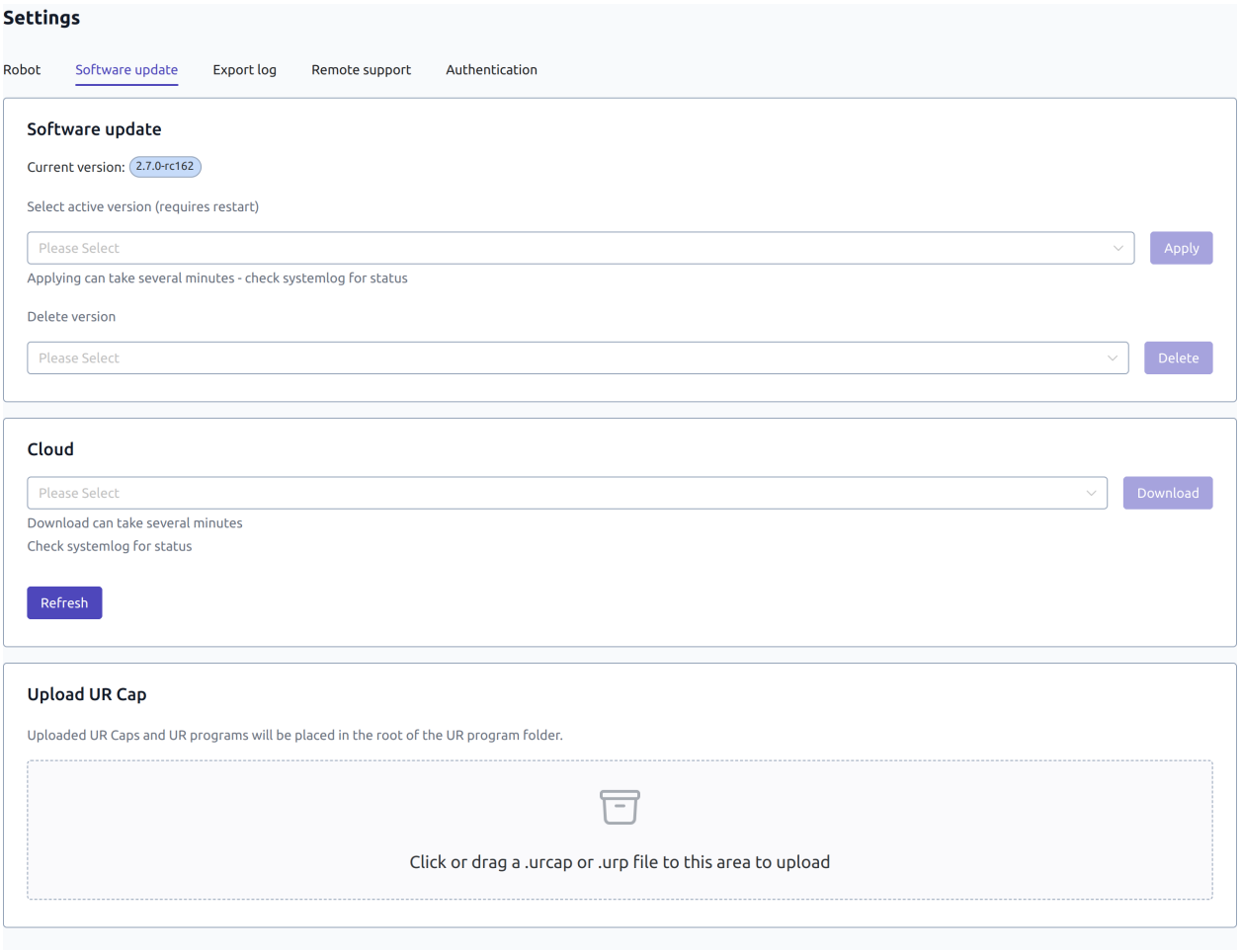


Fig. 4.4: Software update

Note: Releases are about 9 GB so the download might take a while depending on your internet connection.

2. Updating the Ability URCap

Go to the Downloads page on enabled-robotics.com and download the corresponding version of the Ability URCap.

Upload the URCap to the robot from the “Upload URCap” window. The URCap will be placed in the root of the UR program folder.

Another option is to save the Ability URCap to a USB-stick and insert it into the teach-pendant on the robot.

Press the “Hamburger” menu in the top right corner of the teach-pendant and go to Settings -> System -> URCaps.

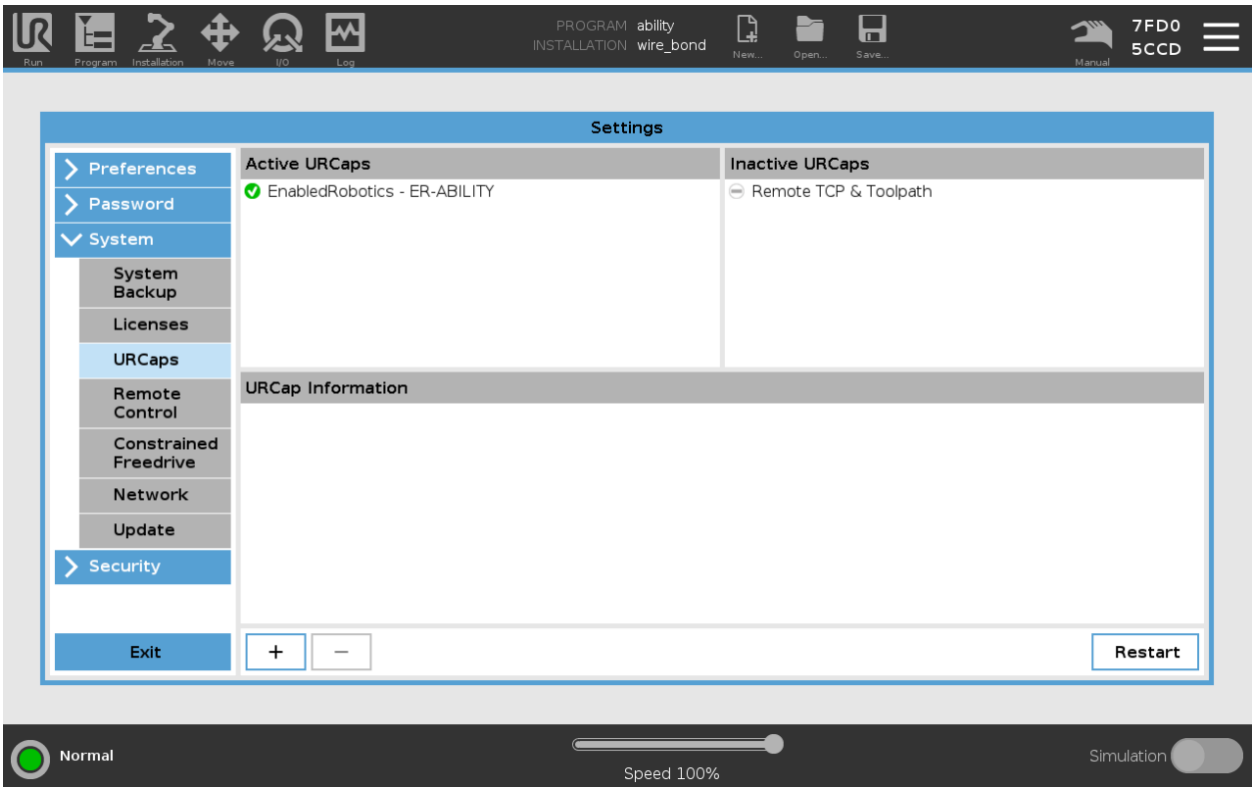


Fig. 4.5: Installing the ability URCaps through the UR's interface

From here press the "+" to add the new URCap. Open them one by one in the file system and press "Restart" to activate the new versions.

The robot is now updated. Confirm the update by connecting to the robot and checking the version number in the lower left corner of the web-interface. Make sure to check out the release notes for the update on the Software Releases Forum found at enabled-robotics.com

4.2.3 Export log

The export log page allows you to export all log entries in a given time window. Simply select the to and from and click "Confirm" to download a zipped archive of log files directly to your device.

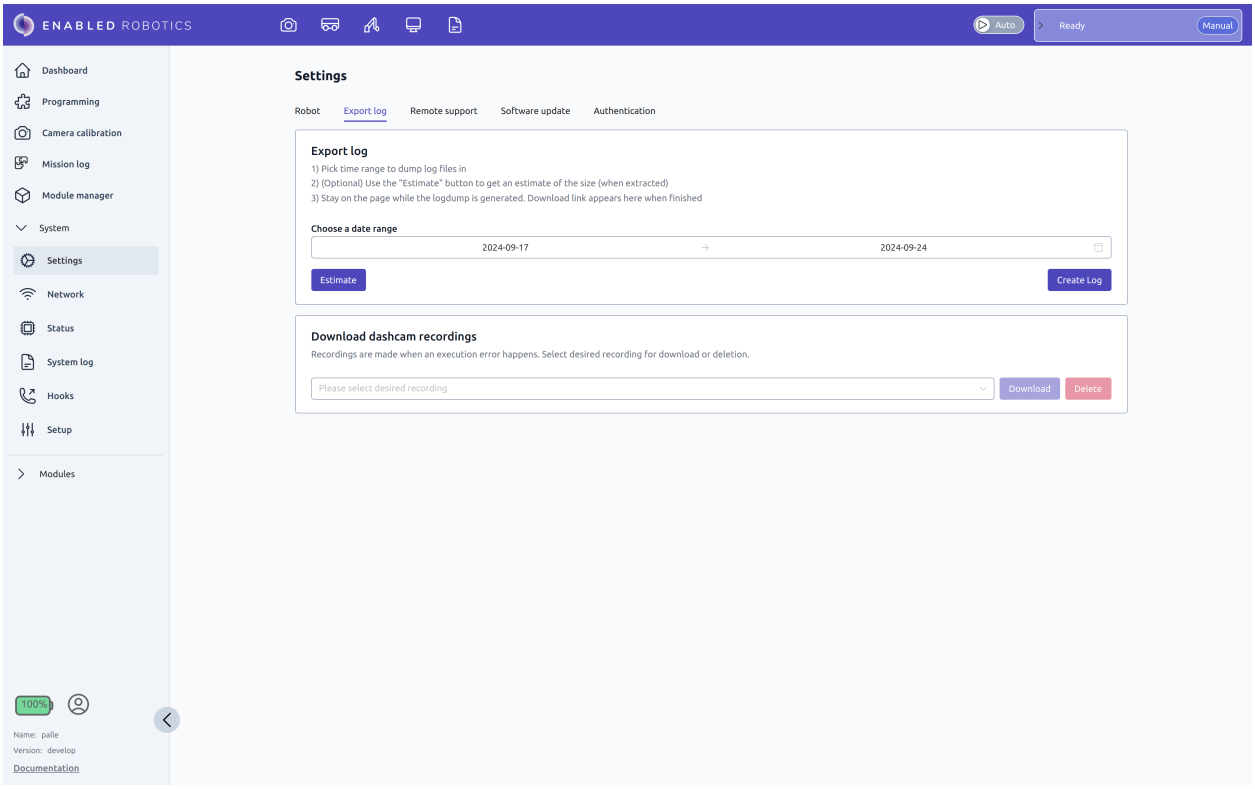


Fig. 4.6: Export log

You can also download and delete recordings made by the Dashcam Module. Read more about the Dashcam Module in the *Dashcam* section.

4.2.4 Remote support

The remote support tab gives an employee at Enabled Robotics a basic terminal access to the robot to help diagnose problems.

Enter a session ID given by the Enabled Robotics employee and click connect.

Remote support of course requires that the robot is connected to the internet. Connecting to a WiFi hotspot is described in the *Network* section. A network connection status icon indicates if the robot has internet access.

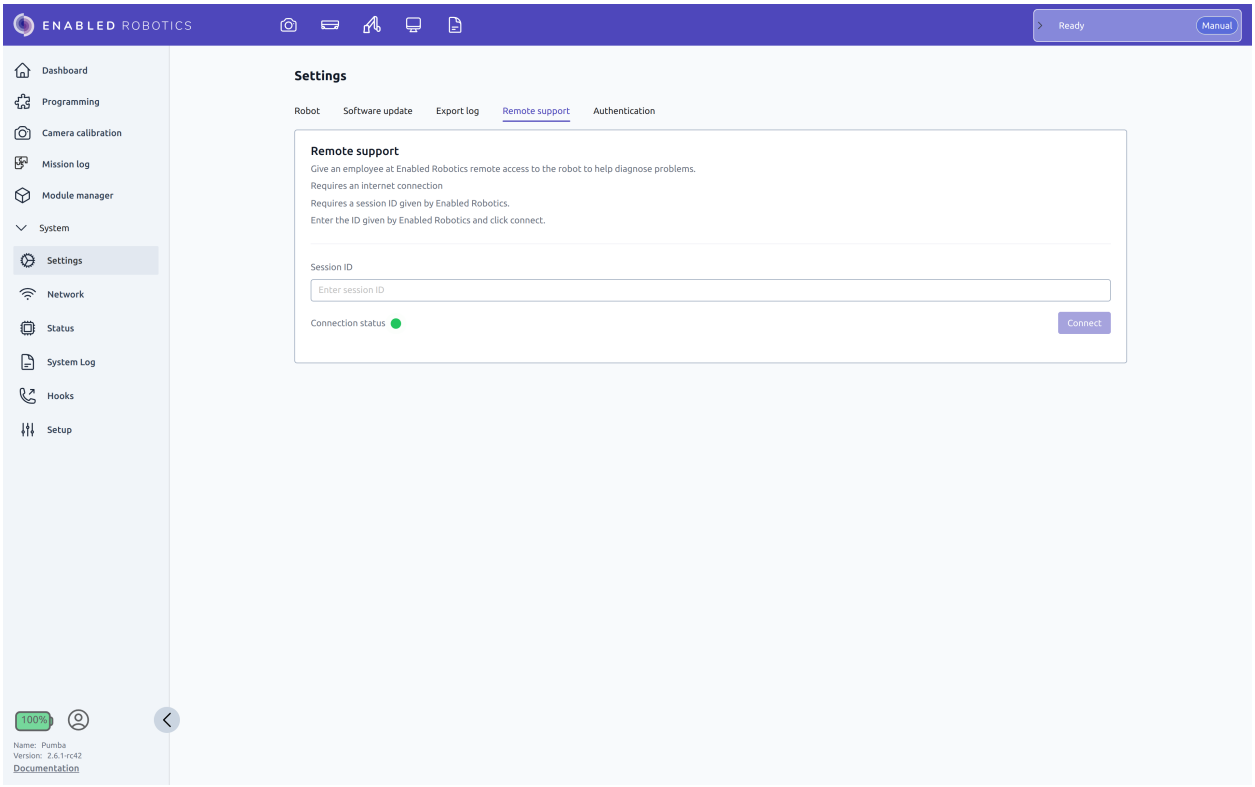


Fig. 4.7: Remote support

4.2.5 Authentication

If your organization runs an oauth service, it is possible to configure the robot to authenticate users against this service.

Enter the URLs, Client ID and Client secret.

Once configured you have the option to use local credentials or OAuth credentials on the login screen.

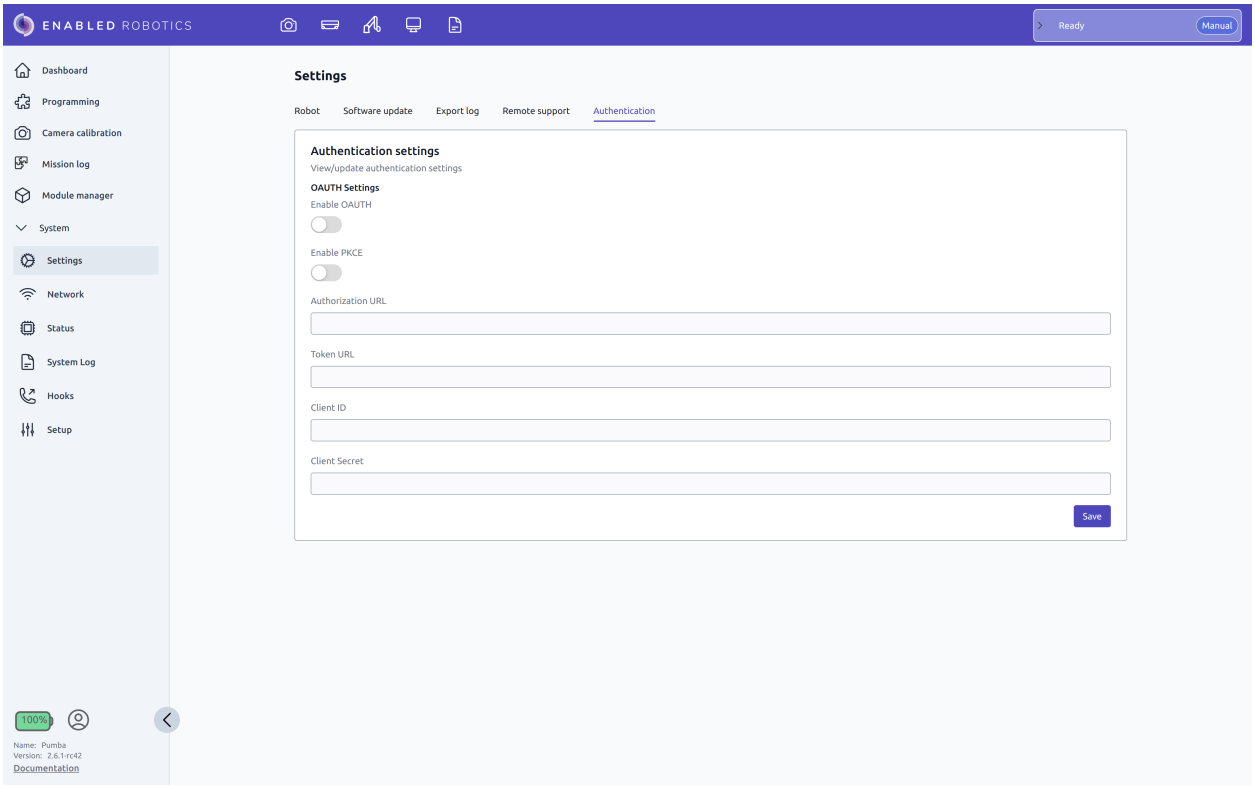


Fig. 4.8: Authentication settings

4.3 Network

4.3.1 Connection to Wi-Fi hotspot

The robot can be connected to an existing wireless network. To connect go to *Network* and press 'New connection' to get the screen shown in Fig. 4.9.

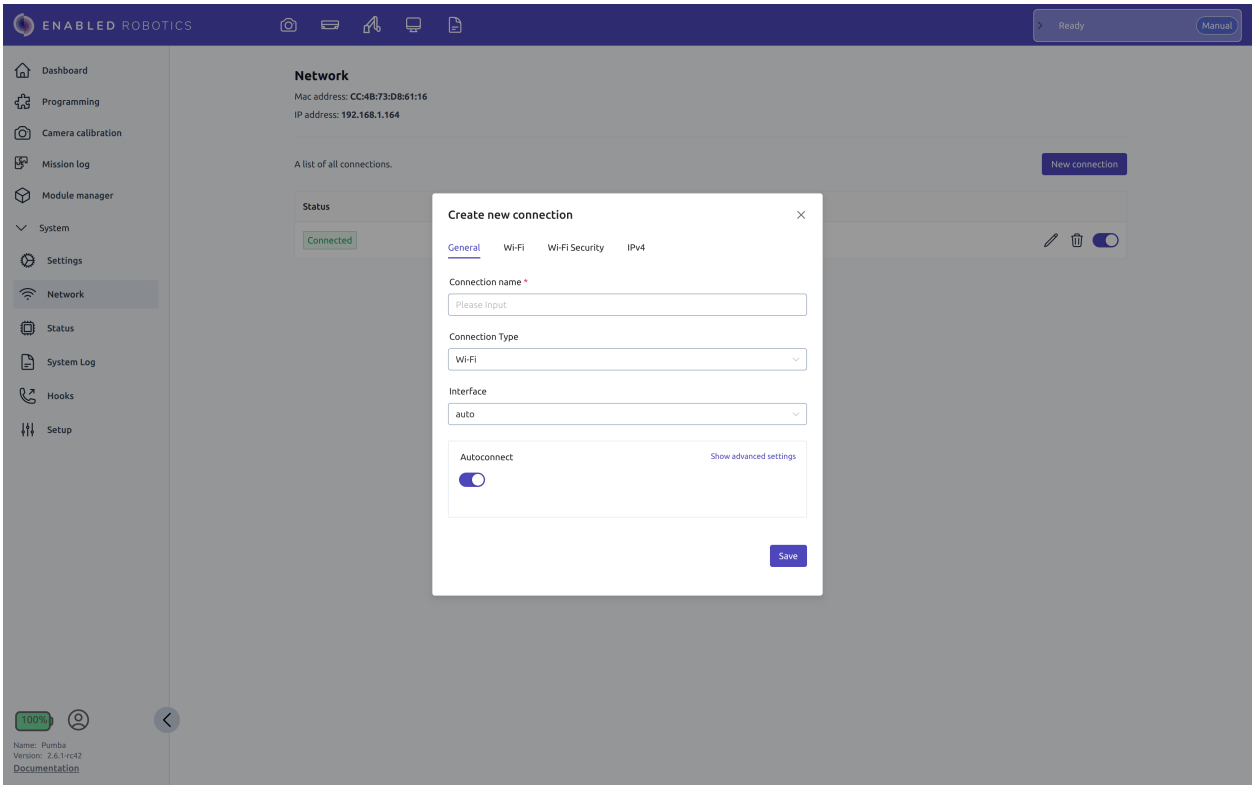


Fig. 4.9: Setup connection to a network.

Name the connection and select 'Connection Type'. Autoconnect can be selected if you want the robot to automatically connect to the given network whenever available.

In the 'Wi-Fi' tab, select your network from the list of available SSIDs. Go to the 'Wifi Security' tab, select security type and enter relevant information. Press 'Submit' to have the robot connect to the network. Once connected, the connectivity status will be displayed to the left on the new connection.

Note: When editing a connection, the security details must be filled out again.

4.3.2 Ports

The ports listed below need to be open for their corresponding services to be available on the network. Notice, that if the service is not required, the port does not have to be open.

- 22 - SSH: Used for remote support sessions.
- 80 - HTTP: Web server used to serve the web interface.
- 443 - HTTPS: Used for software updates.
- 4840 - OPC-UA: Used to expose the OPC-UA Interface.
- 8082 - REST: Used to expose the REST API.
- 8090 - Video stream: Used to show camera view in web interface.
- 9090 - Websocket: Used for communication between web interface and the robot.

4.4 Camera calibration

Before using the vision package, the camera needs to be calibrated. This calibration includes both the intrinsic calibration of the camera parameters as well as the robot to camera calibration.

Note: Every time the location of the camera on the arm changes, it should be calibrated again to function properly. Even small changes may impact the accuracy of the results.

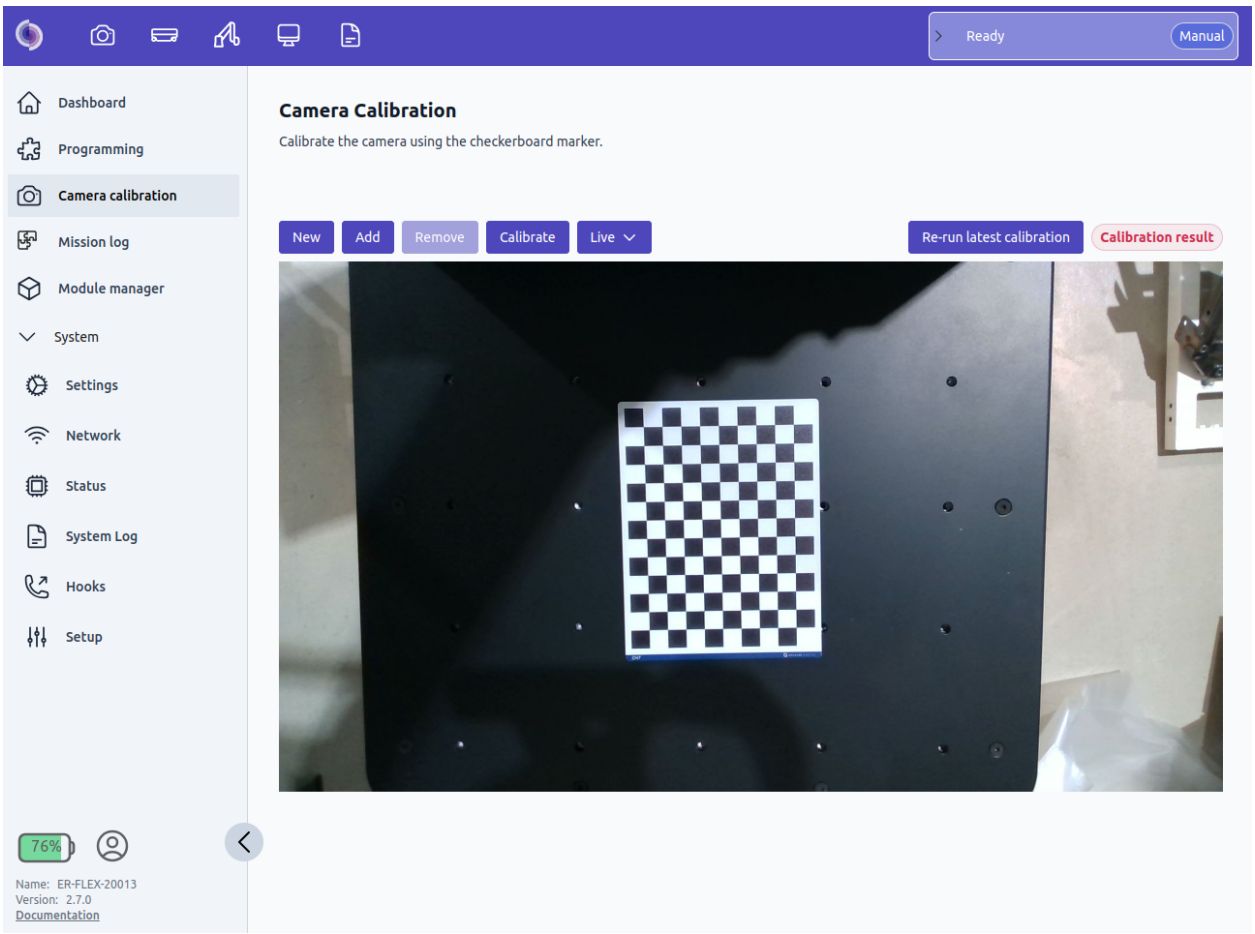


Fig. 4.10: Camera calibration page

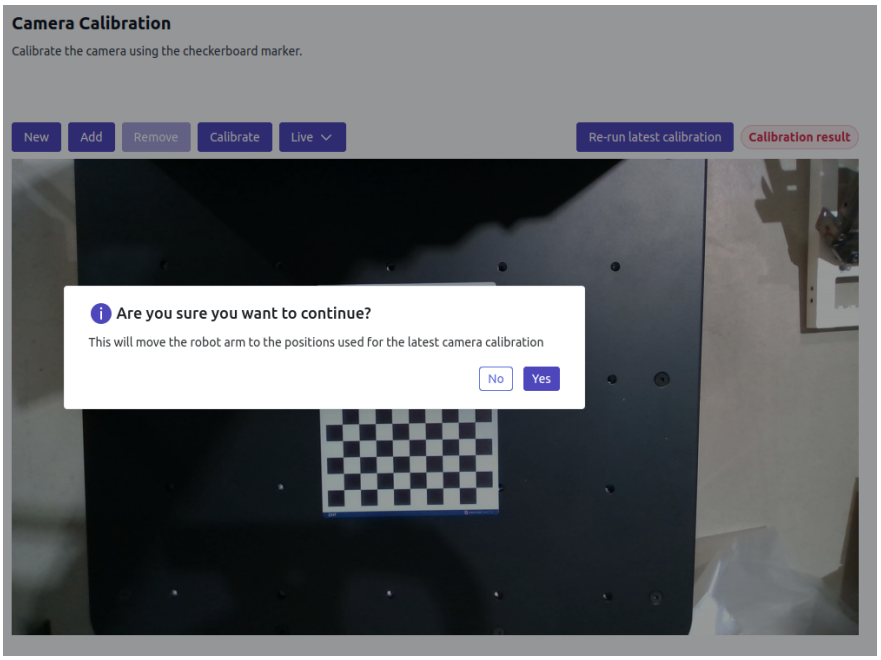
To collect calibration measurements, follow the steps:

1. Place the calibration board in a stable position where it is possible for the camera to see it from many different angles (e.g. on top of the module).
2. Press 'New' to create a new calibration
3. Move the robot to a configuration where the complete calibration board is within the camera view and press 'Add'. If the calibration board was detected the screen displays the recognition result for 2 seconds, before returning to the live view. You can always click the dropdown box to view the individual recognition results and check the current measurement count.
4. Repeat until 15 or more measurements are added. It is important to vary both the camera's rotation

and position relative to the marker. The maximum recommended angle between camera image plane and calibration board is 45°.

5. Press 'Calibrate'. Once calibrated you will see the calibration result displayed in the upper right part of the screen. The icon in the upper right corner of the screen will change color indicating the quality of the calibration (Green=Good, Yellow = Fair, Red = Poor). Under normal conditions the intrinsic error should be less than 0.5 and the extrinsic less than 1.5.

Re-run latest camera calibration If you need to re-calibrate the camera, you can press the "Re-run latest calibration" button. This will repeat the latest calibration by moving to the same robot arm positions, capturing new marker images and finally updating the intrinsic and extrinsic camera calibration. Before using this feature, make sure that it is safe to move the robot arm and that the marker is placed at the same position as last time the calibration was done.



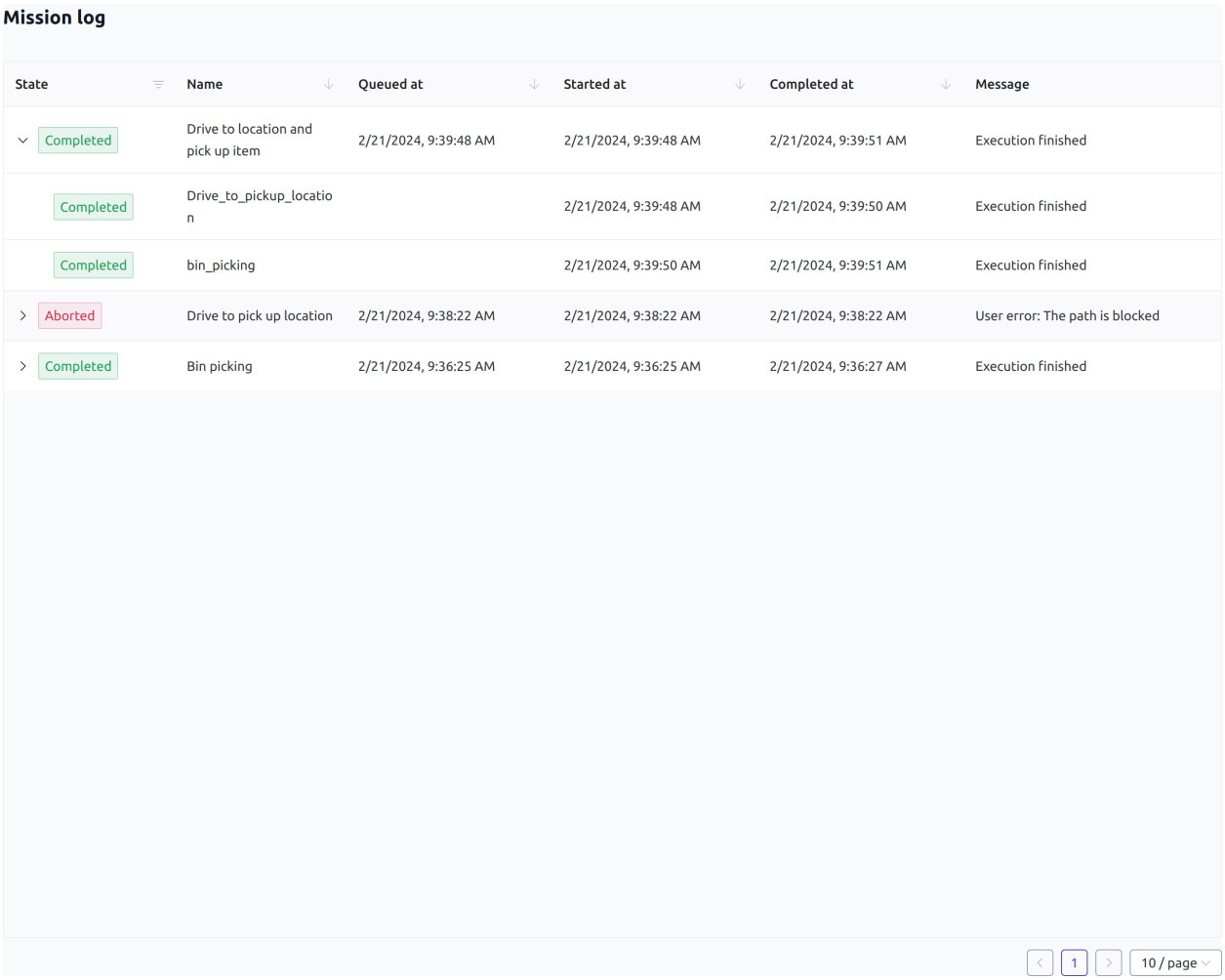
Troubleshooting

If the calibration does not show a satisfactory result it may be due to

1. Too many measurements with very steep angles between the calibration board and the camera or too large distance between camera and calibration board.
2. Too little variation in the orientation of the camera relative to the calibration board.
3. Too little variation in the position of the camera relative to the calibration board.
4. That the camera is out of focus.
5. The calibration board has moved relative to the robot during calibration.

4.5 Mission Log

The mission log page give you an overview of all prior executed and cancelled missions.



It is possible to expand the mission and see underlying programs. This can help when debugging larger missions that might have failed, to see which part failed and determine how far the robot got.

4.6 Hooks

The hooks page makes it possible for external systems to register for notifications when certain events occurs on the robot.

It is possible to register for notifications based on REST calls, also called webhooks. The webhooks can be configured and monitored on the hooks page.

The webhooks can be set up in two steps:

- Configure a connection to the REST server
- Subscribe to one of the available events and select the endpoint to send the notification to.

This page covers the following topics:

4.6.1 Events

The following events are available on the robot:

4.6.1.1 State

Receive an update for every state change on the robot.

4.6.1.2 Automatic Mode

A tristate event that has one of the following values:

Automatic Mode	Description
off	The robot is in manual mode.
ready	The robot is in automatic mode and there is nothing in the queue.
running	The robot is running a mission or about to (it has something in the queue).

4.6.2 Connections

A connection groups all communication destined for the same server together. When entering the Hooks page, a collapsed list of all configured connections are shown:

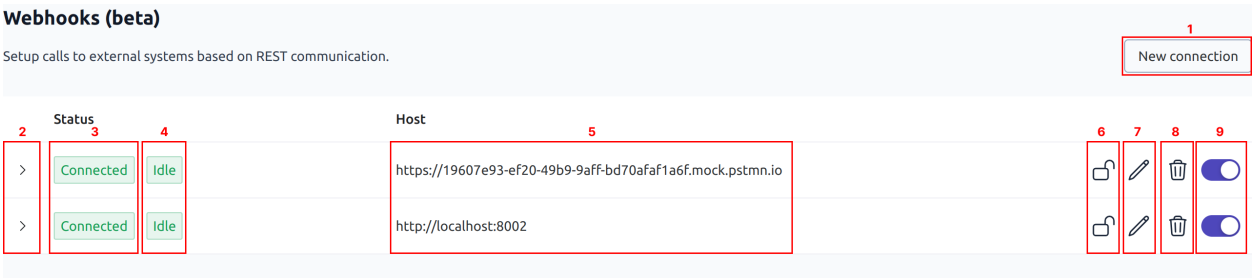


Fig. 4.11: Example of Hooks page with two webhook connections configured.

- To create a connection to a new server, click the "New connection" button (1). See section *Create Connection* for more information.
- Click the arrow (2) to expand a connection. This will show the list of subscriptions for a given server. See section *Subscriptions* for more information.
- The connection state (3) can be monitored in the Status column. See section *Connection States* for more information.
- The execution state (4) can be monitored in the Status column. See section *Execution States* for more information.
- The hostname for the server is shown in the Host column (5).
- If the connection is using TLS and verifying certificates, the lock symbol (6) will show a closed lock.
- Change the connection settings (7). See section *Edit connection* for more information.
- Delete the connection (8). This will remove all subscriptions and delete the connection.
- Enable or disable the connection (9). See section *Connection States* for more information.

4.6.2.1 Create Connection

Pressing the "New connection" button (1 in Fig. 4.11) opens a dialog for adding a new webhook server.

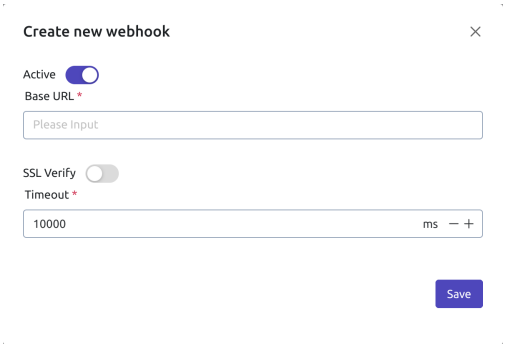


Fig. 4.12: Create new webhook connection.

The following fields must be set when adding the server:

- **Active**: It is possible to disable the connection initially if the server is not yet available and the connection should first be enabled later.

- **Base URL**: The hostname of the server. This address is used as the basis of all subscriptions. It is possible to add a path to the URL that will be used as the base path for all subscriptions under this connection. See more details about this URL in section *Subscriptions*.
- **SSL Verify**: Enable checking of the server certificate when using HTTPS.
- **Timeout**: Set the timeout in milliseconds (default is 10 seconds). If the server does not respond within this time limit, the connection is assumed to have failed. This timeout is used for subscriptions marked as critical. See more details about the timeout in section *Subscriptions*.

4.6.2.2 Edit connection

It is possible to edit existing connections (7 in Fig. 4.11). The values are the same as when creating a new connection, except that the active status can not be changed in the dialog. This is instead changed directly by using the toggle (9 in Fig. 4.11).

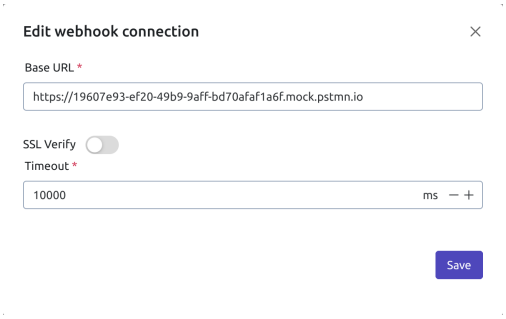


Fig. 4.13: Edit webhook connection.

4.6.2.3 Connection States

The connection state field (3 in Fig. 4.11) shows the current status of each connection.

Table 4.1: Overview of possible connection states.

Connection state	Description
Disconnected	Means that connection was lost, network issues, a notification timed out, or that the user explicitly disabled the connection.
Connecting	Trying to transition into the Connected state.
Connected	The connection is enabled and no communication errors have occured.

Note: If the user explicitly disables a connection, this does not imply that subscriptions are disabled. It works as an override that can be used to simulate a connection failure. If the connection has active subscriptions and the connection is disabled, these subscriptions can still trigger an error on the robot. To avoid this, manually disable all subscriptions before disabling the connection.

See section *Execution States* about the connection execution state field (4 in Fig. 4.11).

4.6.3 Subscriptions

A subscription will connect one of the available events to a specific server. Every time the event happens, the server will be notified. Subscriptions are either in critical or non-critical state. Critical events must reach the

server, and the server must acknowledge that it received the notification. If the notification fails for a critical subscription, the robot goes into entity error and aborts any running mission. Non-critical subscriptions will never cause an error on the robot.

Each connection has a list of subscriptions. These are shown by expanding the connection:

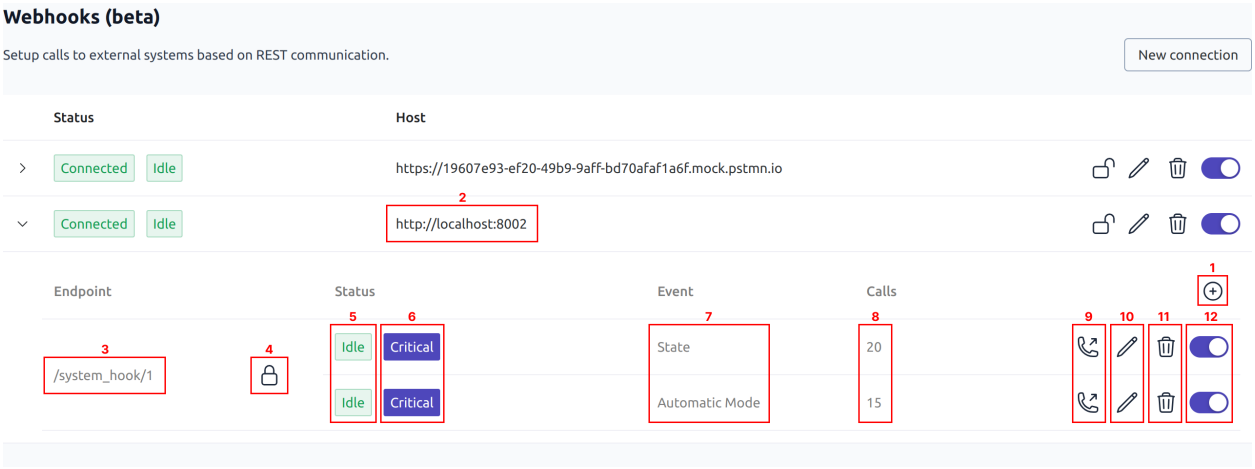


Fig. 4.14: Example of webhook subscriptions.

- To create a new subscription, click the "+" button (1). See section *Create Subscription* for more information.
- The notification is sent to the endpoint path (3). The full path is the concatenation of host url (2) and endpoint (3).
- With multiple subscriptions, it is possible to reuse the same endpoint (as shown on the image).
- It is possible to configure a secret token for each endpoint. If such a secret is set, the subscription is shown with a closed lock (4). The secret is a token that will be sent in the notification messages to the server. The server can check the secret token for added security.
- Each subscription execution state can be monitored (5), similar to the connection execution state. See section *Execution States* for more information about the possible states.
- A subscription can be either critical or non-critical. Click to toggle (6). Critical subscriptions can cause the robot to go into entity error if a notification does not reach the server.
- The event type that the subscription is for (7).
- The number of notifications sent to the server for a specific subscription (8).
- A button (9) to manually trigger a notification. This can be used for testing purposes. The number of calls increases (8) and the state (5) changes to "Executing" until the server has responded.
- Edit the subscription (10). See section *Edit Subscription* for more information.
- Delete the subscription (11).
- Enable/disable the subscription (12).

Warning: State subscriptions will never block the robot, while the Automatic mode subscriptions can block the robot if marked as critical. This means that the robot can appear unresponsive if there are large network delays or if the server does not respond fast enough. A timeout value can be set on the connection. In general, avoid marking subscriptions as critical, unless it is absolutely vital that external

systems are updated with the current information from the robot. For monitoring purposes, subscriptions should be marked non-critical.

4.6.3.1 Create Subscription

Click the "+" button (1 in Fig. 4.14) to add a new subscription.

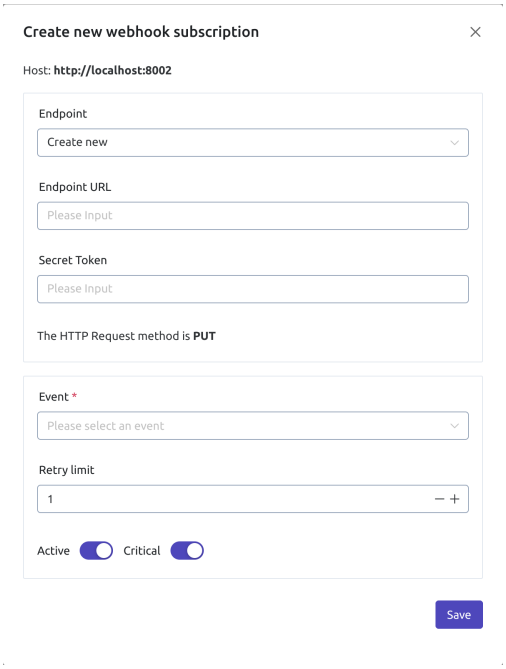


Fig. 4.15: Dialog for adding a new webhook subscription.

There are two parts of the subscription to configure:

Endpoint Insert the **Endpoint URL** (relative to the **Base URL** of the connection). Optionally provide a **Secret Token** with each notification sent to the server. If there are existing endpoints on the connection, it is also possible to reuse these by selecting one in the **Endpoint** dropdown at the top. This dropdown is not shown if there are no existing endpoints available on the connection.

Event Select the event to receive notifications from using the **Event** dropdown. It is not possible to select an event if there already is a subscription for this event on the endpoint. If a call failes the number of retry attempts can be set by setting the **Retry limit**. The default value is set to 1, meaning if the first call failes, one retry attempt is made. It is possible to select if the subscription is **Active** initially. If inactive, the subscription can be configured now and enabled later. The **Critical** mark can also be enabled or disabled.

4.6.3.2 Edit Subscription

It is possible to edit a subscription using the pencil icon (10 in Fig. 4.14). The endpoint section of the subscription can be edited, but notice that this affects all subscriptions on the same endpoint. It is possible to edit the subscription directly from the overview, clicking (6) or (12).

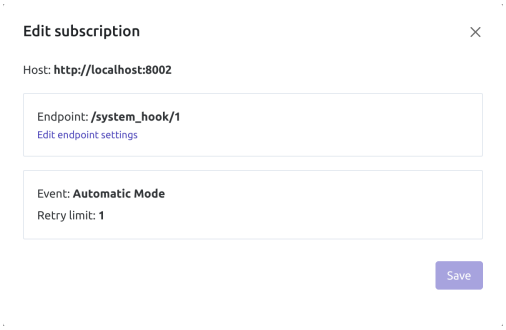


Fig. 4.16: Dialog for editing an existing webhook subscription.

4.6.3.3 Recovering from Error

In case a critical subscription fails, the robot will go into entity error. The topbar will show an error with instructions to resolve the error on the Hooks page:

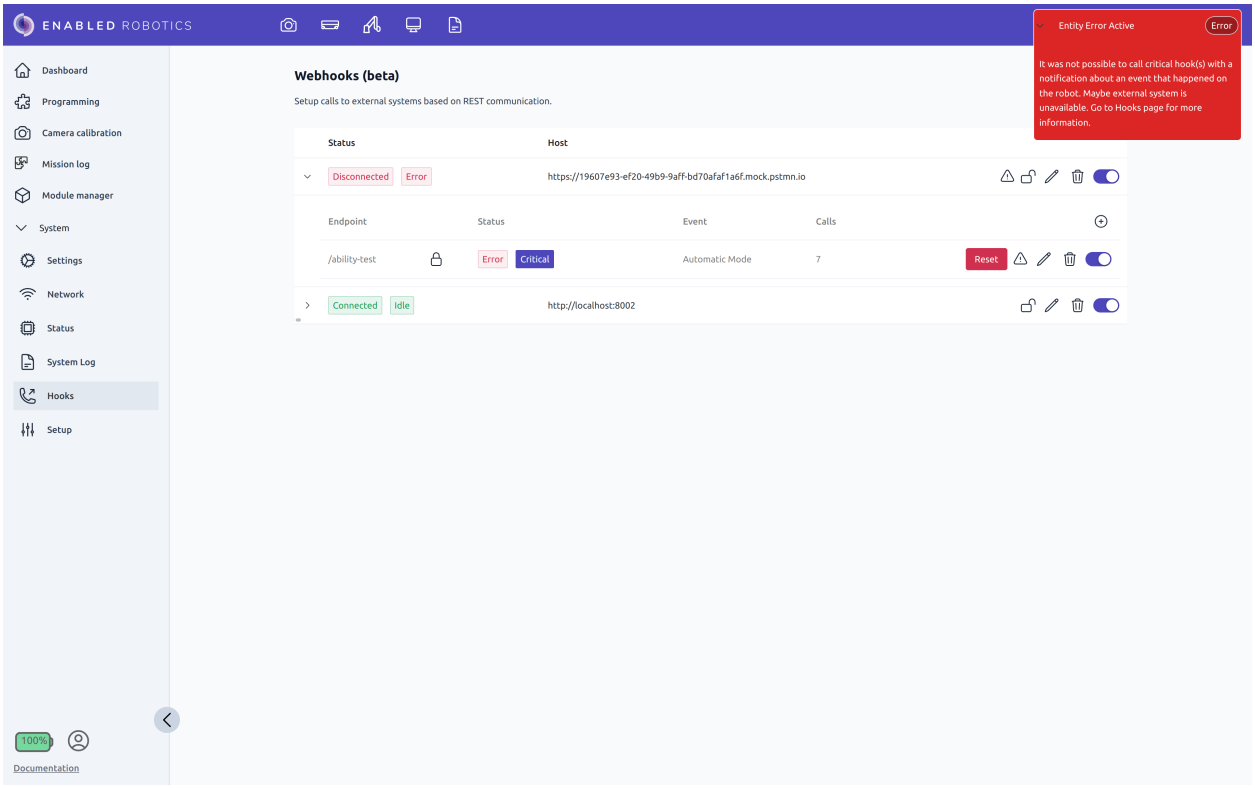


Fig. 4.17: Example of a critical subscription failing, causing the robot to go into error.

To resolve the error it is necessary to click the red “Reset” button. This button is only visible for failed subscriptions. On the connection level, a warning triangle is shown. This triangle indicates that one of the subscriptions under the connection have failed and needs to be reset.

If the subscription continues to fail, there is a connection or server error that must be resolved. The robot can not be used as long as this problem persists. To use the robot without integration with the external system, the subscription can be disabled, or the subscription can be marked as non-critical. Notice that it is

still necessary to reset the subscription after disabling or remarking it, but the robot should not go into error next time the event is triggered. Remember to reenable the subscription when the error is resolved.

4.6.4 Execution States

The execution states gives insight into the current state of all connections (4 in Fig. 4.11) and subscriptions (5 in Fig. 4.14). The connection execution state covers multiple subscriptions, so typically it is necessary to monitor the execution state on each individual subscription to get detailed information about the state.

Table 4.2: Overview of possible connection states.

Execution state	Description
Off	If the connection is explicitly disabled when booting the robot, or when the connection is first added. The Off state will never occur again after transitioning to any other state.
Starting	The user has requested that the connection is enabled. This is a transition state to Idle.
Idle	The connection is ready to send notifications to the server.
Executing	A notification is currently being sent to the server.
Stopping	User has requested that the connection is disabled. Waiting for running notifications to finish or abort.
Suspended	The connection has been disabled by user request.
Error	An error occured when trying to notify the server.
Resetting	The connection is trying to recover after an error.

Note: Notice that a warning triangle is shown on the connection when one or more subscriptions have failed. Expand the connection (2 in Fig. 4.11) to identify the failing subscription(s) and correct the issue(s).

4.7 Setup

A Setup is a way to define a configuration of the robot, which can be shared among multiple programs. It is comprised of a number of Setup Entities which can have different names and types. When a program starts, all values will be initialized to the values defined in the Setup, despite these having changed in the last program execution.

To configure or create a Setup go to *Setup*.

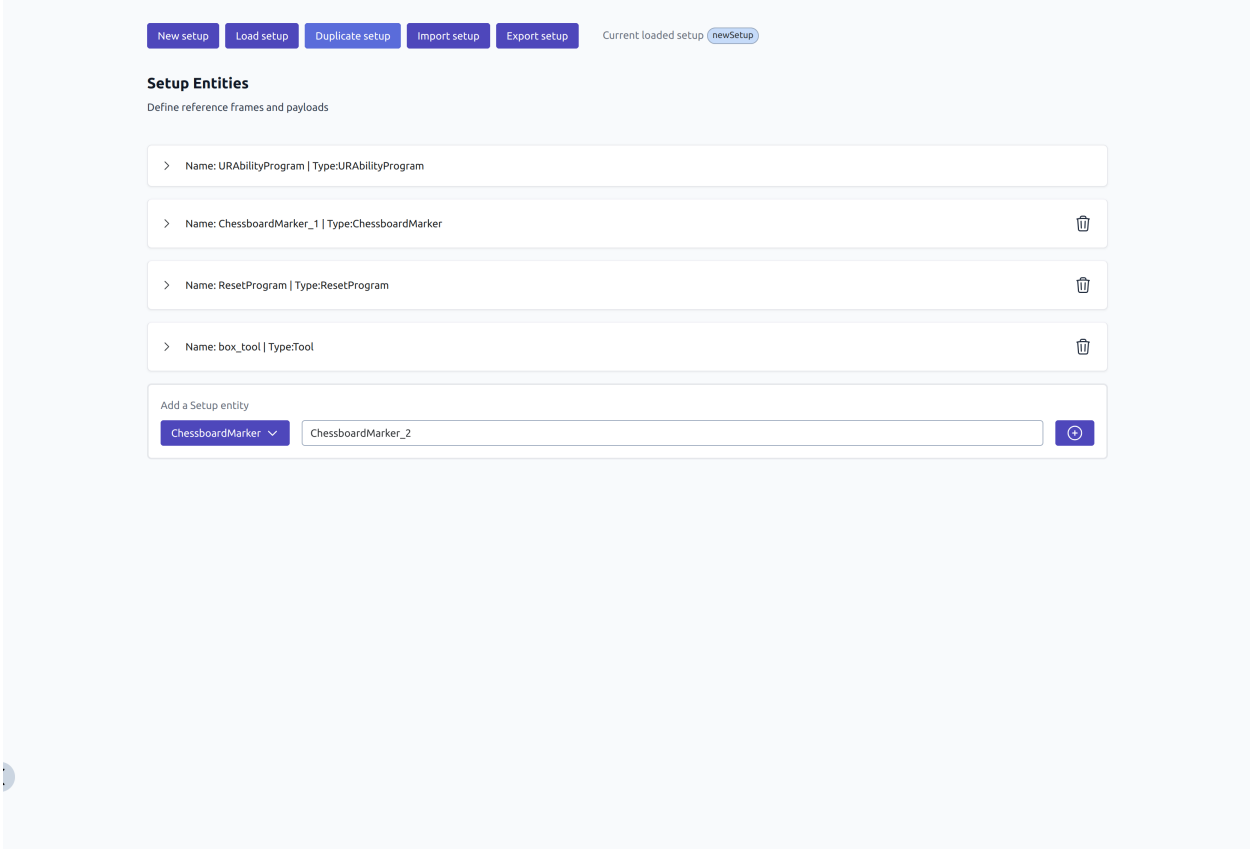


Fig. 4.18: Adding a Setup Entity in the setup menu

When defining a new Setup you can either press the 'New' button or choose to duplicate the current setup.

4.7.1 Setup Entities

Setup Entities are static software configurations that can be used across all programs. To add new Setup Entity you need to:

1. Provide a name.
2. Select which type of Setup Entity you wish to define.
3. Press the '+' on the left hand side.
4. Open the entity item created, enter the desired value(s) and press 'Apply Settings'.

Below is a description of each type of Setup Entity.

4.7.1.1 URAbilityProgram

The URAbilityProgram Setup Entity is used to define which UR program should be started on the UR. If the UR program is stopped, a pop-up will appear in the Ability interface asking to start the program defined in the Setup Entity. Unlike the other Setup Entities, only **1** URAbilityProgram can be active at a time.

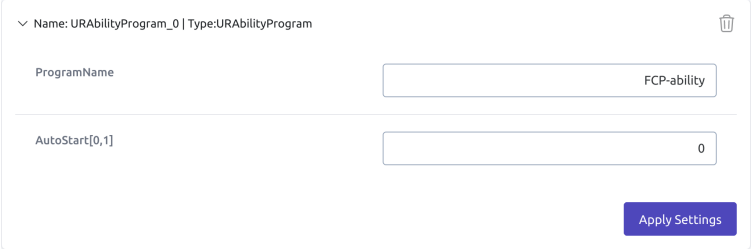


Fig. 4.19: The URAbilityProgram Setup Entity settings

Parameters:

- **Name**: The name of the UR program to be loaded (and run) on startup. Keep in mind, this UR Program must include an ER-Ability program node.
- **AutoStart[0,1]**: Setting this option to 1 will enable auto-starting the UR program in case it is stopped e.g. when resetting a protective stop. In this case, no pop-up will appear in the webinterface. Setting it to 0 will disable auto-starting, and a pop-up will instead appear asking to start the program in case it is stopped.

4.7.1.2 Tool

The Tool Setup Entity is used to define different tools in the Ability interface. These tools can then be used in several of the *Manipulator* blocks when programming the robot. Adding a Tool Setup Entity allows for making movements relative to the TCP of the mounted tool instead of the robots flange.

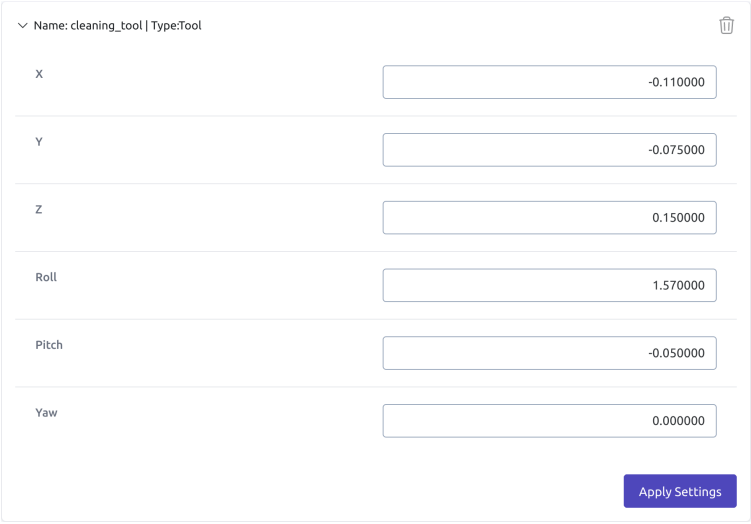


Fig. 4.20: The Tool Setup Entity settings

Parameters:

- **X,Y,Z**: The translation between the manipulators tool-flange and the TCP of the tool. **Given in meters [m]**.
- **Roll, Pitch, Yaw**: The rotation around the Z, Y and X axes respectively, between the tool-flange and the TCP of the rool. **Given in radians [rad]**.

4.7.1.3 ChessboardMarker

The ChessboardMarker Setup Entity is used to define custom chessboard markers. Custom chessboard markers allows for creating markers that can have any rectangular dimensions, for example, see below our standard CH3 marker (6x5) versus a custom box marker (3x8).

Fig. 4.21: Left: Standard CH3 marker (6x5). Right: Custom box marker (3x8).

After adding a new ChessboardMarker Setup Entity, the custom defined marker can be found in the drop-down in the *Calibrate to Marker* block menu.

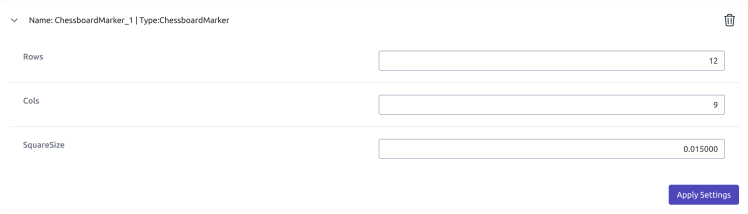


Fig. 4.22: The ChessboardMarker Setup Entity settings

Parameters:

- **Rows**: The number of rows of modules on the marker.
- **Cols**: The number of columns of modules on the marker.
- **SquareSize**: The width/height of 1 module. **Given in meters [m]**.

Note: The number of rows and columns is counted as the saddlepoints between the black and white squares.

Important: Markers must be asymmetric e.g. uneven number of rows, even number of columns. This ensures rotational invariance.

Important: The SquareSize of a custom marker must exactly match the size of the physical, printed marker. Even a 1mm error can cause major distortion in the estimated pose of the marker when using the *Calibrate to Marker* block. Similar to the Enabled Robotics markers, a custom marker needs to have some whitespace

around the chessboard to allow robust detection. We recommend 5mm of whitespace. Furthermore, the square size should be above 5 mm and the number of rows and columns should be at least 3. In general, the more rows and columns the marker has, the better detection.

4.7.1.4 ResetProgram

The ResetProgram Setup Entity is used to define a reset program that is executed as a mission when switching from manual to automatic mode. If missions are added before the robot is in automatic mode, the reset program will still be executed as the first mission when entering automatic mode.

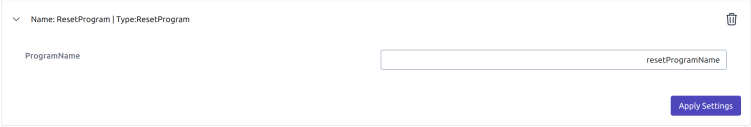


Fig. 4.23: The ResetProgram Setup Entity settings

Parameters:

- **ProgramName**: The name of the reset program to be executed. Keep in mind, the program must be available on the robot or a loading error will occur.

4.7.2 Import and export

It is possible to export the current setup by clicking the 'Export' button. This will download a file which defines all current Setup Entities and Hardware Entities.

It is also possible to upload a setup file using the 'Import' button. This is useful if you have multiple robots and want to use the same setup configuration on all robots.

Note: When importing a setup from another robot, please be aware that some entities might need to be manually updated. For instance, if the manipulator configurations are different, it is likely that user defined tools need to be updated.

CHAPTER

FIVE

INTERFACES

This section describes different interface for communication between robot and external systems.

5.1 ROS Interface

5.1.1 Overview

Ability provides a series of ROS topics and services that 3rd party developers can use to interface with the robot.

The interface is compatible with `ROS Melodic` and `ROS Noetic`.

All publicly supported topics and services are exposed under the `/er/` namespace.

The interface is divided in 3 sub namespaces:

1. `system`

   For Core system functionality.

2. `manip`

   For functionality regarding the manipulator.

3. `mobile`

   For functionality regarding the mobile platform.

5.1.2 Installation

The ROS package `er_public_msgs` is required to use the ROS interface and can be installed with:

`apt install ros-melodic-er-public-msgs` or `apt install ros-noetic-er-public-msgs`

5.1.3 System

Namespace	Name	Description	Type
`/er/system/`			
	`programs`	A list of programs on the system	Topic
	`status`	The current status of the robot	Topic
	`execute_program`	Load and execute a program on the robot	Service
	`stop`	Stop execution and reset errors (if any)	Service

Note: State definitions:

- NOT_CONNECTED = -1
- NO_PROGRAM = 0
- READY = 1
- EXECUTING = 2
- PAUSED = 3
- ERROR = 4

5.1.4 Manipulator

Namespace	Name	Description	Type
/er/manip/			
	events	A list of event nodes on the manipulator	Topic
	move_lin_to_joint	Perform a linear move to a joint configuration	Service
	move_lin_to_transform	Perform a linear move to a transform (base to tcp)	Service
	move_ptp_to_joint	Perform a Point-to-Point move to a joint configuration	Service
	move_ptp_to_transform	Perform a Point-to-Point mvoe to a transform	Service
	execute_event	Execute an event_node on the manipulator	Service
	execute_program	Load and start a program on the manipulator	Service
	activate	Initialize and brake-release the manipulator	Service
	set_payload	Set the payload of the manipulator	Service
	set_tcp	Set the tcp of the manipulator	Service

Note:

- Joint configuration is specified in **radians**
- Transform is specified in **meters**
- Speed in **rad/s**
- Acceleration in **rad/s^2**

5.1.5 Mobile platform

Namespace	Name	Description	Type
/er/mobile/			
	coordinates	The current coordinates of the mobile platform in the current map	Topic
	positions	A list of defined positions in the current map	Topic
	charging_stations	A list of defined charging stations in the current map	Topic
	missions	A list of missions on the mobile platform	Topic
	move_to_coordinates	Move to a set of coordinates	Service
	move_to_position	Move to a position defined on the map	Service
	move_to_chargign_station	Dock to a charging station	Service
	execute_mission	Execute a mission	Service

5.2 OPC-UA Interface

5.2.1 Overview

OPC-UA is an industrial client-server communication protocol. OPC-UA is made for both data handling and control of systems such as machines and robots. Ability hosts an OPC-UA Server that can be accessed to load, start and stop programs on the robot as well as monitor the battery percentage and program state.

5.2.2 OPC-UA Interface Usage

The OPC-UA server hosted by the robot can be accessed by its endpoint-url: `opc.tcp://<robot-ip>:4840`. Currently no security protocols are in place. The server contains 6 nodes. 4 methods and 2 variables which can be used to monitor and control the robot. Each node is identified by a namespace index (ns) and a node id (id). The structure of the Ability server can be seen in Fig. 5.1.

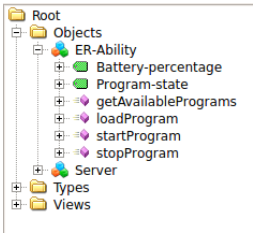


Fig. 5.1: Ability OPC-UA server overview.

Variables

Battery percentage and Program state are read only variables that are continuously updated.

ns	id	Type	Description
1	`bat-percent`	integer	Current battery percentage of the robot
1	`pro-state`	string	Current program state of the robot

Methods

To identify a method you need both the ns-index and node id of the method node itself, but also of the object node that contains it. In the case of the Ability OPC-UA server, all nodes are contained by the object node "ER-Ability" (ns=1, id=er-ability).

The methods on the server have the following signatures:

ns	id	Signature	Description
1	`getAvailablePrograms`	`string[] getAvailablePrograms()`	Returns available programs on the robot.
1	`loadProgram`	`void loadProgram(string program)`	Loads a specified program
1	`startProgram`	`void startProgram(string[] parameters)`	Starts the loaded program with the specified parameters
1	`stopProgram`	`void stopProgram()`	Stops the currently running program

5.2.3 UAExpert

An easy way to access and browse the robots OPC-UA server is to use the free OPC-UA client UAExpert[2] which can be found for both windows and linux. After starting UAExpert a server on the same network can be accessed by the following steps:

1. Click the "+" button in the top bar
2. Add server under "Custom Discovery"
3. Enter servers endpoint url. `opc.tcp://<robot-ip>:4840`
4. The server will now pop up in the top left menu called "Project", right click and connect. The server can now be browsed in the "Address space" menu.

5.2.4 Troubleshooting

A full description of the possible error codes can be found here:

https://open62541.org/doc/current/statuscodes.html

5.3 REST Interface V2

5.3.1 Overview

Ability exposes a REST interface. It provides information about the system state and also allows the user to affect this state.

[2] https://www.unified-automation.com/products/development-tools/uaexpert.html

The interface can be accessed at `<robot-ip>:8082/v2`
For development and experimentation, an interactive interface is availabel at `<robot-ip>:8082/v2/ui`.
Alternatively Postman[3] can be used to query the endpoints.

Endpoint	Method	Description
`/programs`	`GET`	Get all available programs on the robot
`/programs/current`	`GET`	Get the currently loaded program
`/programs/current`	`PUT`	Load a new program
`/status`	`GET`	Get the current status of the robot
`/status`	`PUT`	Change the state of the robot
`/references`	`GET`	Get all defined references on the system
`/transform/{reference_id}`	`GET`	Get transform to a specific reference (XYZRPY) (Meters and Radians)
`/help/datatypes`	`GET`	Get a dictionary from supported datatypes to IDs

5.3.2 Endpoints

5.3.2.1 `GET /programs`

Get all available programs on the robot

Responses

`200: Success`

```
[
  "Program1",
  "Program2",
  "Program3"
]
```

`500: Server error`

`503: No connection to backend`

5.3.2.2 `GET /programs/current`

Get the currently loaded program

Responses

`200: Success`

```
{
    "name": "string",
    "arguments": [
        {
```

(continues on next page)

[3] https://www.postman.com/

(continued from previous page)

```
            "name": "string",
            "type": 0,
            "value": "string"
        }
    ]
}
```

204: No program is currently loaded

500: Server error

503: No connection to backend

5.3.2.3 PUT /programs/current

Load a new program

Request body

```
{
    "name": "string",
    "arguments": [
        {
            "name": "string",
            "type": 0,
            "value": "string"
        }
    ]
}
```

Note: The integer in the `type` field specifies the datatype of the `value` field. For a list of supported datatypes query the `/help/datatypes` endpoint

Responses

200: Success

```
{
    "name": "string",
    "arguments": [
        {
            "name": "string",
            "type": 0,
            "value": "string"
        }
    ]
}
```

400: General user error

500: Server error

503: No connection to backend

5.3.2.4 GET /status

Get the current status of the robot

Responses

200: Success


```
{
    "state": "Idle",
    "current_program": {
        "name": "string",
        "arguments": [
            {
                "name": "string",
                "type": 0,
                "value": "string"
            }
        ]
    },
    "battery": 0
}
```


500: Server error

503: No connection to backend

5.3.2.5 PUT /status

Change the state of the robot

Request body


```
{
    "state": "Idle"
}
```


Note: `state` must be one of `Idle` `Executing` or `Paused`

Responses

200: Success

```
{
    "state": "Idle",
    "current_program": {
        "name": "string",
        "arguments": [
            {
                "name": "string",
                "type": 0,
                "value": "string"
            }
        ]
    },
    "battery": 0
}
```


400: General user error

500: Server error

503: No connection to backend

5.3.2.6 GET /references

Get all defined references on the system

Responses

200: Success


```
{
    "names": [
        "Base",
        "World"
    ],
    "uids": [
        "Base",
        "World"
    ]
}
```


500: Server error

503: No connection to backend

5.3.2.7 GET /transform/{reference_id}

Get transform to a specific reference (XYZRPY) (Meters and Radians)

Path Parameters

`reference_id` (required)

Query Parameters

`_from` (optional)

Responses

`200: Success`

```
{
    "transform": [
        0.8008282,
        0.8008282,
        0.8008282,
        0.8008282,
        0.8008282,
        0.8008282
    ]
}
```

`500: Server error`

`503: No connection to backend`

5.3.2.8 GET /help/datatypes

Get a dictionary from supported datatypes to IDs

Responses

`200: Success`

```
{
    "Boolean": 3,
    "Double": 1,
    "Integer": 2,
    "String": 0
}
```

5.4 REST Interface V3

5.4.1 Overview

Ability exposes a REST interface. It provides information about the system state and also allows the user to affect this state. It also provides endpoints to manipulate the mission queue.

The interface can be accessed at `<robot-ip>:8082/v3`
For development and experimentation, an interactive interface is availabel at `<robot-ip>:8082/v3/ui`.

Alternatively Postman[4] can be used to query the endpoints.

Endpoint	Method	Description
`/programs`	`GET`	Get all available programs on the robot
`/status`	`GET`	Get the current status of the robot
`/status`	`PUT`	Change the state of the robot
`/references`	`GET`	Get all defined references on the system
`/transform/{ref_id}`	`GET`	Get transform to a specific reference (XYZRPY) (Meters and Radians)
`/help/datatypes`	`GET`	Get a dictionary from supported datatypes to IDs
`/queue`	`GET`	Get the current queue on the robot
`/queue`	`POST`	Add a mission to the queue
`/queue`	`DELETE`	Clear mission queue
`/queue/{mission_id}`	`GET`	Get mission from queue
`/queue/{mission_id}`	`PUT`	Modify a mission in queue
`/queue/{mission_id}`	`DELETE`	Delete a mission in the queue
`/mission/current`	`GET`	Get the currently executing mission
`/log`	`GET`	Get the mission log
`/log/{mission_id}`	`GET`	Get the log for a mission

5.4.2 Endpoints

5.4.2.1 `GET /programs`

Get all available programs on the robot

Responses

`200: Success`

```
[
  "Program1",
  "Program2",
  "Program3"
]
```

`500: Server error`

`503: No connection to backend`

5.4.2.2 `GET /status`

Get the current status of the robot

Responses

`200: Success`

[4] https://www.postman.com/

```
{
    "state": {
        "safety": "Idle",
        "system": "Idle",
        "execution": "Idle",
        "joystick": "Idle"
    },
    "current_mission": {},
    "battery": 0,
    "message": ""
}
```


500: Server error

503: No connection to backend

5.4.2.3 PUT /status

Stop an executing mission.

This endpoint will stop a mission and take the robot out of automatic mode.

Request body


```
{
    "state": "Idle"
}
```


Responses

200: Success


```
{
    "state": {
        "safety": "Idle",
        "system": "Idle",
        "execution": "Idle",
        "joystick": "Idle"
    },
    "current_mission": {},
    "battery": 0,
    "message": ""
}
```


400: General user error

500: Server error

503: No connection to backend

5.4.2.4 GET /references

Get all defined references on the system

Responses

200: Success


```
{
    "names": [
        "Base",
        "World"
    ],
    "uids": [
        "Base",
        "World"
    ]
}
```


500: Server error

503: No connection to backend

5.4.2.5 GET /transform/{ref_id}

Get transform to a specific reference (XYZRPY) (Meters and Radians)

Path Parameters

ref_id (required)

Query Parameters

_from (optional)

Responses

200: Success


```
[
    -0.002428792212254366,
    1.3217533789434908,
    0.8140000000000001,
    -1.7617055621694448,
    0,
    0
]
```


500: Server error

503: No connection to backend

5.4.2.6 GET /help/datatypes

Get a dictionary from supported datatypes to IDs

Responses

200: Success


```
{
    "String": 0
    "Double": 1,
    "Integer": 2,
    "Boolean": 3,
}
```


5.4.2.7 GET /queue

Get the current queue on the robot

Responses

200: Success


```
[
    {
        "completed_at": "2000-12-31T12:00:00.0Z",
        "id": "00000000-0000-0000-0000-000000000000",
        "message": "Mission state message",
        "name": "mission_name",
        "programs": [
            {
                "arguments": [
                    {
                        "name": "string",
                        "type": 0,
                        "value": "string"
                    }
                ],
                "completed_at": "2000-12-31T12:00:00.0Z",
                "in_line_program": "string",
                "message": "Program state message",
                "name": "program_name",
                "program_id": "00000000-0000-0000-0000-000000000000",
                "started_at": "2000-12-31T12:00:00.0Z",
                "state": 0,
                "state_text": "Not started"
            }
        ],
        "queued_at": "2000-12-31T12:00:00.0Z",
        "started_at": "2000-12-31T12:00:00.0Z",
        "state": 0,
```


(continues on next page)

(continued from previous page)

```
        "state_text": "Not started",
        "webhook": {
            "context": "Some identifier included in callback to server",
            "uri": "http://myserver.com:8002/status"
        }
    }
]
```

500: General server error

503: No connection to backend

5.4.2.8 POST /queue

Add a mission to the queue

Request body

```
{
    "id": "00000000-0000-0000-0000-000000000000",
    "name": "mission_name",
    "programs": [
        {
            "arguments": [
                {
                    "name": "string",
                    "type": 0,
                    "value": "string"
                }
            ],
            "name": "program_name"
        }
    ]
}
```

Responses

201: Success

```
{
    "name": "mission_name",
    "id": "00000000-0000-0000-0000-000000000000",
    "queued_at": "2025-03-06T13:20:23.141454506Z",
    "started_at": "",
    "completed_at": "",
    "state": 0,
    "state_text": "Not started",
    "message": "",
    "webhook": {
        "uri": "",
```

(continues on next page)

(continued from previous page)


```
        "context": ""
    },
    "location": "",
    "programs": [
        {
            "name": "program_name",
            "id": "9b661832-a6cb-4020-99c9-f10320e05e66",
            "arguments": [],
            "started_at": "",
            "completed_at": "",
            "state": 0,
            "state_text": "Not started",
            "message": "",
            "in_line_program": "",
            "webhook": {
                "uri": "",
                "context": ""
            }
        }
    ]
}
```


400: Bad Request

409: Mission with the given id already exists

5.4.2.9 DELETE /queue

Clear mission queue

Responses

204: Deleted

500: General server error

503: No connection to backend

5.4.2.10 GET /queue/{mission_id}

Get mission from queue

Responses

200: Success

```
{
    "name": "mission_name",
    "id": "00000000-0000-0000-0000-000000000000",
    "queued_at": "2025-03-06T13:20:23.141454506Z",
    "started_at": "",
    "completed_at": "",
    "state": 0,
    "state_text": "Not started",
    "message": "",
    "webhook": {
        "uri": "",
        "context": ""
    },
    "location": "",
    "programs": [
        {
            "name": "program_name",
            "id": "9b661832-a6cb-4020-99c9-f10320e05e66",
            "arguments": [],
            "started_at": "",
            "completed_at": "",
            "state": 0,
            "state_text": "Not started",
            "message": "",
            "in_line_program": "",
            "webhook": {
                "uri": "",
                "context": ""
            }
        }
    ]
}
```


`404: General server error`

5.4.2.11 `PUT /queue/{mission_id}`

Modify a mission in the queue

Path Parameters

`mission_id` (required)

Request body


```
{
    "id": "00000000-0000-0000-0000-000000000000",
    "name": "mission_name",
    "programs": [
        {
```


(continues on next page)

(continued from previous page)


```
            "arguments": [
                {
                    "name": "string",
                    "type": 0,
                    "value": "string"
                }
            ],
            "name": "program_name"
        }
    ]
}
```


Responses

200: Success


```
{
    "name": "mission_name",
    "id": "00000000-0000-0000-0000-000000000000",
    "queued_at": "2025-03-06T13:20:23.141454506Z",
    "started_at": "",
    "completed_at": "",
    "state": 0,
    "state_text": "Not started",
    "message": "",
    "webhook": {
        "uri": "",
        "context": ""
    },
    "location": "",
    "programs": [
        {
            "name": "program_name",
            "id": "9b661832-a6cb-4020-99c9-f10320e05e66",
            "arguments": [],
            "started_at": "",
            "completed_at": "",
            "state": 0,
            "state_text": "Not started",
            "message": "",
            "in_line_program": "",
            "webhook": {
                "uri": "",
                "context": ""
            }
        }
    ]
}
```


400: General user error (or mission not found)

500: General server error

503: No connection to backend

5.4.2.12 DELETE /queue/{mission_id}

Delete a mission in the queue

Path Parameters

`mission_id` (required)

Responses

`204: Deleted`

`500: General server error`

`503: No connection to backend`

5.4.2.13 GET /mission/current

Get the currently executing mission

Responses `200: Success`


```
{
    "name": "mission_name",
    "id": "00000000-0000-0000-0000-000000000000",
    "queued_at": "2025-03-06T13:20:23.141454506Z",
    "started_at": "2025-03-06T13:22:21.111653556Z",
    "completed_at": "",
    "state": 1,
    "state_text": "Executing",
    "message": "",
    "webhook": {
        "uri": "",
        "context": ""
    },
    "location": "",
    "programs": [
        {
            "name": "program_name",
            "id": "9b661832-a6cb-4020-99c9-f10320e05e66",
            "arguments": [],
            "started_at": "2025-03-06T13:22:21.111653556Z",
            "completed_at": "",
            "state": 1,
            "state_text": "Executing",
            "message": "",
            "in_line_program": "",
            "webhook": {
                "uri": "",
                "context": ""
```


(continues on next page)

(continued from previous page)

```
            }
        }
    ]
}
```

204: No mission is currently executing

500: General server error

503: No connection to backend

5.4.2.14 GET /log

Get the mission log on the robot

Responses

200: Success

```
[
    {
        "name": "mission_name",
        "id": "00000000-0000-0000-0000-000000000000",
        "queued_at": "2025-03-06T13:18:59.558354826Z",
        "started_at": "2025-03-06T13:19:02.995345305Z",
        "completed_at": "2025-03-06T13:56:42.083492668Z",
        "state": 2,
        "state_text": "Completed",
        "message": "Execution finished",
        "webhook": {
            "uri": "",
            "context": ""
        },
        "location": "",
        "programs": [
            {
                "name": "program_name",
                "id": "00000000-0000-0000-0000-000000000000",
                "arguments": [],
                "started_at": "2025-03-06T13:19:02.998248805Z",
                "completed_at": "2025-03-06T13:56:42.083456888Z",
                "state": 2,
                "state_text": "Completed",
                "message": "Execution finished",
                "in_line_program": "",
                "webhook": {
                    "uri": "",
                    "context": ""
                }
            }
```

(continues on next page)

(continued from previous page)

```
        ]
    }
]
```

500: General server error

503: No connection to backend

5.4.2.15 GET /log/{mission_id}

Get a mission from the mission log

Path Parameters

mission_id (required)

Responses

200: Success


```
{
    "name": "mission_name",
    "id": "00000000-0000-0000-0000-000000000000",
    "queued_at": "2025-03-06T13:18:59.558354826Z",
    "started_at": "2025-03-06T13:19:02.995345305Z",
    "completed_at": "2025-03-06T13:56:42.083492668Z",
    "state": 2,
    "state_text": "Completed",
    "message": "Execution finished",
    "webhook": {
        "uri": "",
        "context": ""
    },
    "location": "",
    "programs": [
        {
            "name": "program_name",
            "id": "00000000-0000-0000-0000-000000000000",
            "arguments": [],
            "started_at": "2025-03-06T13:19:02.998248805Z",
            "completed_at": "2025-03-06T13:56:42.083456888Z",
            "state": 2,
            "state_text": "Completed",
            "message": "Execution finished",
            "in_line_program": "",
            "webhook": {
                "uri": "",
                "context": ""
            }
        }
```


(continues on next page)

(continued from previous page)

```
    ]
}
```

404: Mission not found

CHAPTER

SIX

MODULES

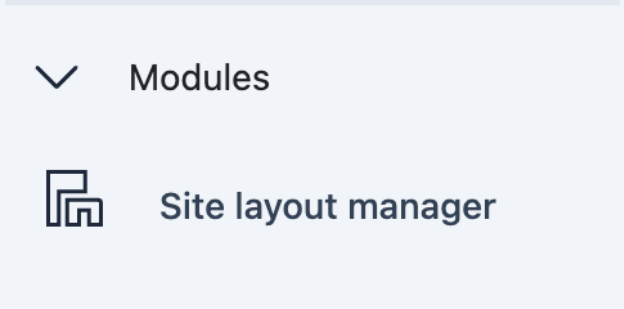
This section describes how to get started with built-in modules.

6.1 Site manager

The Site Manager module includes functionality to add sites, locations, and shelves. This data can be used in the Programming page.

6.1.1 Getting Started

To get started with the Site Manager module, first navigate to the Module Manager page and activate the Site Manager module. After activation, a new dropdown called **Modules** will appear at the bottom of the left-hand side navigation. Under this dropdown, you will find the Site Manager page.

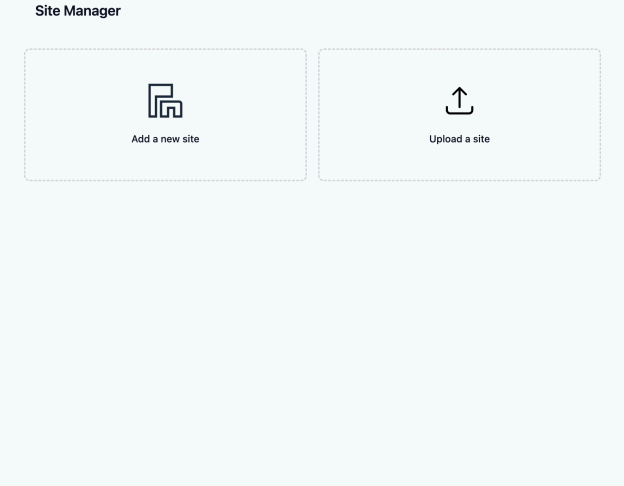


6.1.2 Sites dashboard

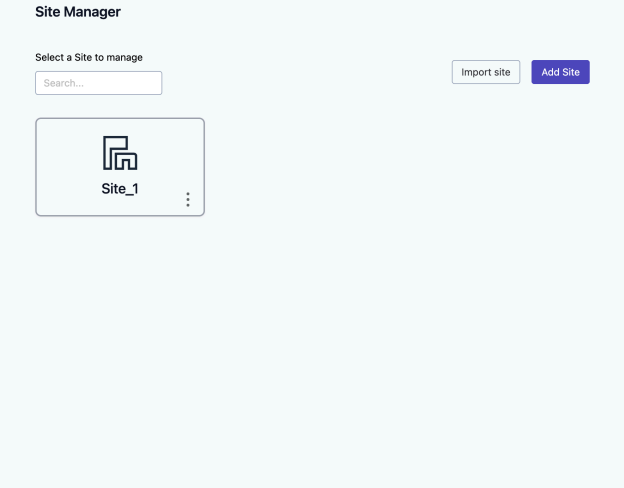
To be able to use any of the functionality, you need to claim the robot.

When you enter the page, you will be presented with a dashboard with the sites that you have created. On the top right, there are Import and Add buttons for importing and adding a site, respectively. On the top left, there is a search bar to search your current sites and a colored box indicating the currently loaded site.

If you have no sites yet, you will see two large buttons to either add or upload a site.



Each site will have an icon on the bottom right, allowing you to delete or export the site.

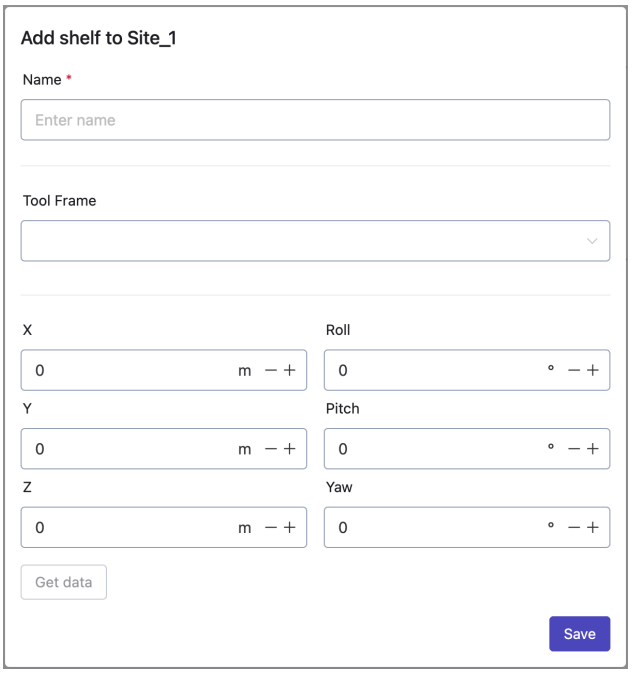


Clicking on any site will take you to a new page where you can view, add, edit, or delete shelves for your chosen site.

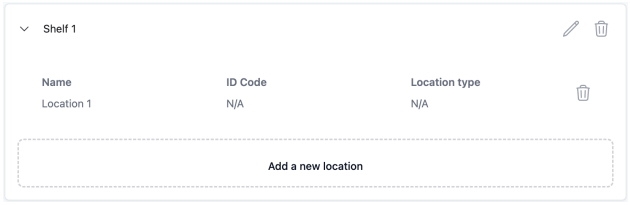
Shelf

A machine or storage shelf, positioned somewhere on the map of the mobile platform. A shelf has a collection of Locations.

To add your first shelf, click on the 'Add a new shelf' button. This will open a modal, where you can enter informati

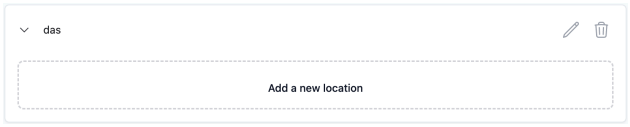


To add shelves after this point, you will need to click on the 'Add Shelf' button in the top right. You can edit or delete shelves using the respective buttons in the shelf dropdown.



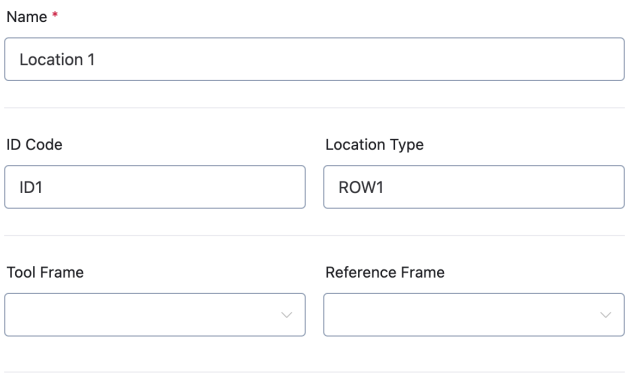
Location A location on a shelf can be a storage position for boxes for intra-logistics or marker positions for calibration.

A location is an entity within a shelf. To view a location, click on the Shelf dropdown. If you have no locations yet, you will see a button to add a location



After adding a location, you can edit it by clicking on the location row itself and delete it by pressing the delete icon next to it.

Usage in the Programming page After creating a location, you can use these locations in the Programming page. Before doing so, specify an ID code or a Location type to identify the location.



When you navigate to the Programming page, you will see a category called 'Site Manager' in the programming workspace.

Here you will find a few blocks: Using the example ID code from above (ID1) here are some examples of using these blocks:

Locations

This block returns a list of locations

Get Location by from ID code

This block returns a location UID

Example usage:

Pose for Location

This block returns a pose of the location

Example usage:

Type for Location

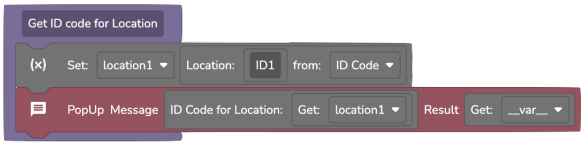
This block returns a location type which you have specified, and can specify when editing a location

Example usage:

ID Code for Location

This block returns an ID code

Example usage:



Mobile Position for Location

This block returns a mobile position Example usage:

6.2 Dashcam

The Dashcam feature adds a dashcam-like functionality to the robot.

6.2.1 Usage

To use the Dashcam:

1. Navigate to the Module Manager page
2. Enable the Dashcam module.

Once enabled, the Dashcam buffers the last 2 minutes of footage from the camera in memory. If the robot enters an error state, the buffered footage is automatically written to disk.

6.2.2 Managing Recordings

Dashcam recordings can be managed on the **Export Log page** (*Export log*). Here, you can:

- Download recordings
- Delete recordings.

Each Dashcam recording is stored as a `.bag` file and can be visualized using compatible software such as RViz or Foxglove Studio.

6.2.3 Limitations

- Recordings are only stored locally on the robot
- Each Dashcam recording can be up to 500 MB in size
- A maximum of 2 GB of recordings are retained; older recordings will be deleted to comply with this limit

---

Note: Before enabling the Dashcam module, consider the privacy implications. Ensure compliance with company policy and local regulations, which may require posting signs indicating video recording is in progress.
