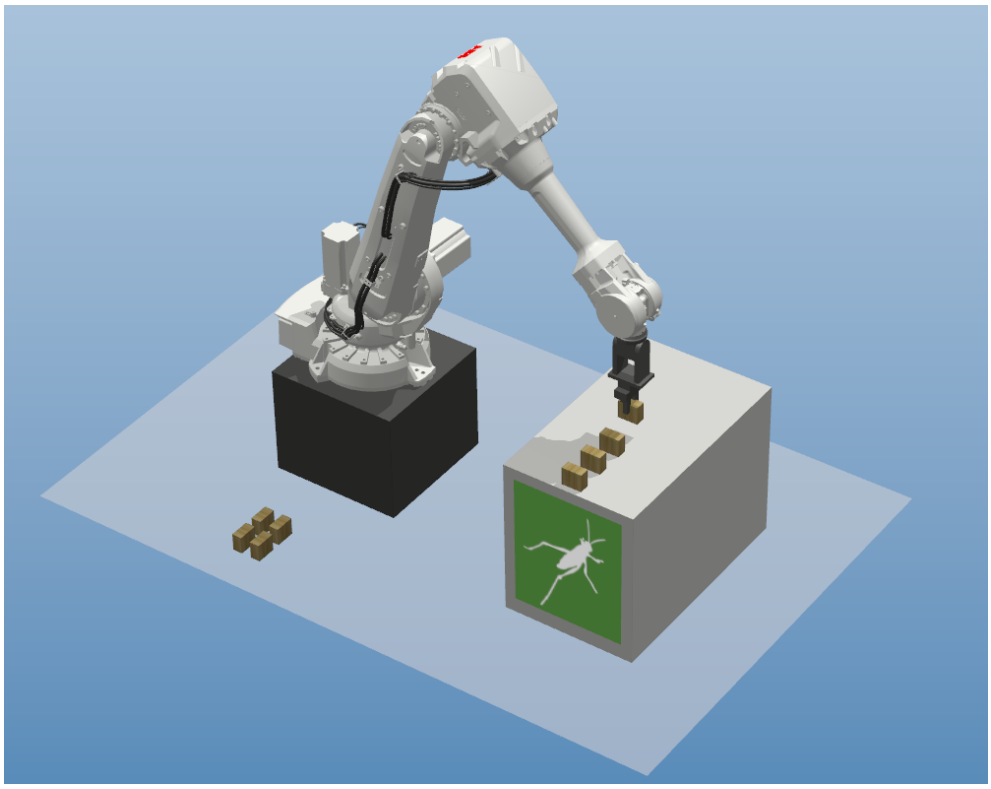
GrassHopper robotic simulation to RobotStudio

Laboratory :
IndexLab

Author :
Thibault Courtois



2024-2025

Guide

Table

Introduction

The aim of this guide is to highlight the advantages of using the GrassHopper plugins for robotic alongside ABB's RobotStudio software to program ABB robots. It also provides the necessary knowledge to work with RobotStudio, enabling users to transfer their simulations from GrassHopper to RobotStudio. This guide is intended for Polimi students who need to use an ABB robotic arm and want to quickly understand the basics of RobotStudio, how to create a digital twin of the real robotic arm they intend to use, the structure of RAPID programming, and how to jog the real robot. This guide is not mean to help you create a simulation by using exclusively RobotStudio. The creation of tool path and target on robot studio for example will not be explained in this text.

Thus, I have decided to use a tiny part of this guide to explain the pros and cons of both the GrassHopper plugins for robotic and RobotStudio to demonstrate the benefits of combining these two software. Then, I will cover the basics of RobotStudio simulation, including downloading and setting up the software and explaining the environment. An important section of this text is dedicated to the manual mode in RobotStudio, which operates identically to the manual mode on the real system. Finally, the last part of this guide is a tutorial that details every step necessary to implement a pick-and-place simulation created in GrassHopper into RobotStudio.

This guide was written after my first month of internship at the IndexLab laboratory. Before that, I had never used Rhino, Grasshopper, or RobotStudio. This guide is based on the limited knowledge I gained from using both software and the methodology I employed. Consequently, it is far from exhaustive and may contain inaccuracies or unoptimized sections. Therefore, I encourage every reader to consult the official ABB guides referenced in the complementary resources section of this guide to enhance their knowledge, double-check whenever they have doubts, and further their mastery of RobotStudio.

1 Why using both GrassHopper and RobotStudio ?

1.1 GrassHopper robot plugins

I have used two plugins in GrassHopper to simulate an ABB IRB 4600-45-205 robotic arm (IndexLab ABB robot). The first one is the *Robot* plugin by Vicose, and the second one is the *Robot Components* plugin, which works similarly to the *Robot* plugin but includes additional tools specific to ABB robots.

The main advantage of using GrassHopper is the ability to quickly create robot targets. By creating an algorithm with GrassHopper blocks, you can simultaneously generate hundreds of targets based on a curve, surface, or brep you create in Rhino or GrassHopper. This is a significant advantage compared to RobotStudio which is less intuitive and fast. Moreover, you can directly integrate Rhino 3D models to create your tools, work objects, and robotic cells for robotic simulation. Consequently, I think that using GrassHopper as a first simulation approach is better compared to a strict utilization of RobotStudio.

However, many tutorials advise using RobotStudio after your GrassHopper simulation to utilize the exact numeric twin of your robot and the ABB inverse kinematic algorithm solver. This solver simulates more singularities and is more accurate because it is the same one used on the real robot. In addition, GrassHopper will generate a RAPID program (the programming language used by ABB robots), which will include one line of code for each target (thus will create RAPID programs that have hundreds of lines). This approach is not optimized, readable, or reusable for other students or people who have not read your GrassHopper algorithm but will, ultimately, use the FlexPendant (the remote controller for your ABB robotic arm) to run the RAPID program. So it is for me essential to learn the basics of RAPID programming to restructure your code and provide a comprehensive version inside the real robot.

Finally, it is difficult to simulate tool animations (such as a gripper closing) and work object displacement (such as visualizing the displacement of a block picked and placed) in GrassHopper. Managing work object data and creating targets relative to different orientation frames (we will see this in the tutorial) is also challenging in GrassHopper, especially with the *Robot* plugins (although it can be done more easily with the *Robot Components* plugins). These tasks are often necessary in RobotStudio.

1.2 RobotStudio

RobotStudio is far slower than GrassHopper to program robot target and path. Indeed, I will not describe this process in this guide, but you will need to create frames manually for every target you want (unless you wants to use advanced features that are similar to GrassHopper but less intuitive). Tool integration is also slower and more detailed. The software globally needs more rigor.

In addition to the advantages developped in the GrassHopper part, I would say that working with RobotStudio particularly the manual mode will fasten your mastery of the real robot. So it won't be a lost of time at the end.

2 RobotStudio setup

2.1 Generalities

BAs a start, let's have a basic explanation of the software RobotStudio and the RobotWare firmware. These two definitions are given in the *Operating manual : Getting started, ICR5 and RobotStudio* you can find here here.

RobotStudio

"RobotStudio is an engineering tool for configuration and programming of ABB robots, both real robots on the shop floor and virtual robots in a PC. To achieve true offline programming, RobotStudio utilizes ABB VirtualRobot™ Technology.

RobotStudio has adopted the Microsoft Office Fluent User Interface. The Office Fluent UI is also used in Microsoft Office. As in Office, the features of RobotStudio are designed in a workflow-oriented way.

With add-ins, RobotStudio can be extended and customized to suit your specific needs. Add-ins are developed using the RobotStudio SDK. With the SDK, it is also possible to develop custom SmartComponents which exceed the functionality provided by RobotStudio's base components."

Concerning RobotStudio SDK and SmartComponents, we won't use them in this guide, but everything is detailed in the ABB official manuals if you want to go further.

RobotWare

"RobotWare is a generic term for all software to be installed in the robot system designed to operate the robot."

The firmware installed on the computer inside the ICR5 armory of the real robot you are using is a version of RobotWare. RobotWare firmware as for 3D printer firmware for example is updated frequently but is physically installed in your ICR5 controller with an installation disk.

Thus, even if the last version of the firmware can be installed on RobotStudio, it is not possible for your real ICR5 controller. Consequently, we will need to set up the correct RobotWare version in RobotStudio (usually a downgrade of the last version) to perfectly twin your real system.

2.2 Installation and setup

2.2.1 Download and Licensing

To download RobotStudio, go to the official website and navigate to this page [1]. If it is the first time you are downloading the software, you can have a 1-month free license by filling the following fields : [2].

Fig. 1 – Downloading RobotStudio

At the end of the download, unzip RobotStudio folder and launch *setup.exe* to begin its installation.
When you open RobotStudio for the first time, two messages will pop up : the first one concerns **licensing,** and the second one makes you **download and install a RobotWare** distribution.

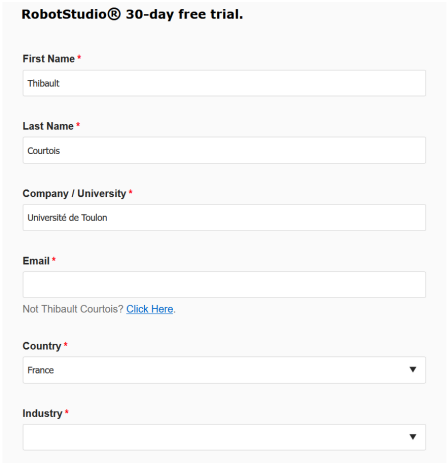


Fig. 2 – Trial License

Let's first discuss about licensing : if you have closed the message and want to register a license, go to *File, option* and then select *General : licensing* in the option window. If it is the first time you downloaded the software, check if you have the 1-month trial license by clicking on *View Licenses*.

If you want to use the **license of IndexLab which is a network license** (a multi-user license linked to a server that allows a certain number of user on the same TCP/IP network) you will have to follow the following steps :

- Quit RobotStudio
- **You need to edit a precise file :** *Slps.Distributor.Services.dll.config* related to licensing and SLP distributor to enable a multi-user license (by default, the feature is disabled). It is located in *C:/Program Files (x86)/ABB/SLP.Distributor.Host/Services* and you need to uncomment the following line : *<add key="Slps.Distributor.Service.EnableUsageLogging" value="true">*.

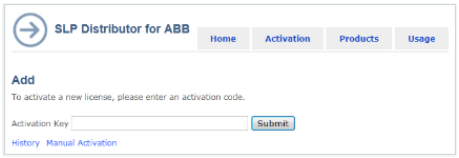


Fig. 3 – SLP distributor server

- Then you need to **install the SLP Distributor server** on your computer. For this, you need to find the unzipped folder of RobotStudio and open the .exe file located in */Utilities/SLP Distributor*.
- Select **TCP ports 2468 and 8731** during the installation of the SLP Distributor.
- You can then go to your local server *http://<server>:2468/web* and **activate the multi-user license key** [3].

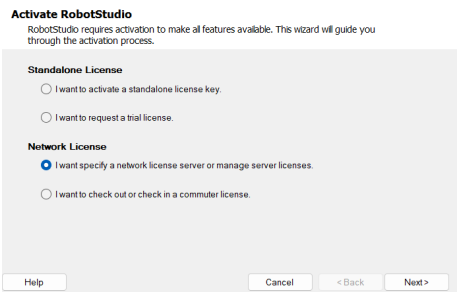


Fig. 4 – Activate RobotStudio License

- You can now **reopen RobotStudio**, go to *File, option* and then select *General : licensing* and click *Activate RobotStudio License*.

- Then select the first option below *Network License*, click *Next* [4]
- Write down the **IP address of your computer** in the text input field and click **finish**.
- RobotStudio will then need to restart, and you will normally have your **license registered** [5]
- **If you have difficulties or need more details, you can click *Help* when you select*Activate RobotStudio License* and go to *Getting Started : How to activate RobotStudio*** [6]

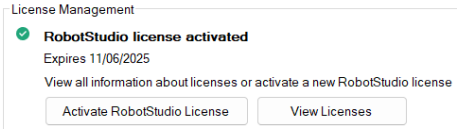


Fig. 5 – License registered

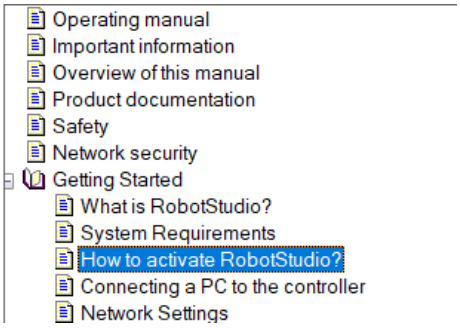


Fig. 6 – Help

2.2.2 Installing the suitable RobotWare

Now we need to install a **RobotWare distribution**. To make the perfect digital twin of the robot you want to simulate, you need to find the suitable RobotWare version. Indeed, RobotWare is frequently updated but your ICR5 controller is not and keep the same software version.

To find it, you need to power up your Robot and use the FlexPendant (remote controller of your robot). Then you need to find the **System Info Menu** and select **System Properties** [7].

If you are using the ABB robot of IndexLab, the RobotWare version is RW6.02_01.00.1029.

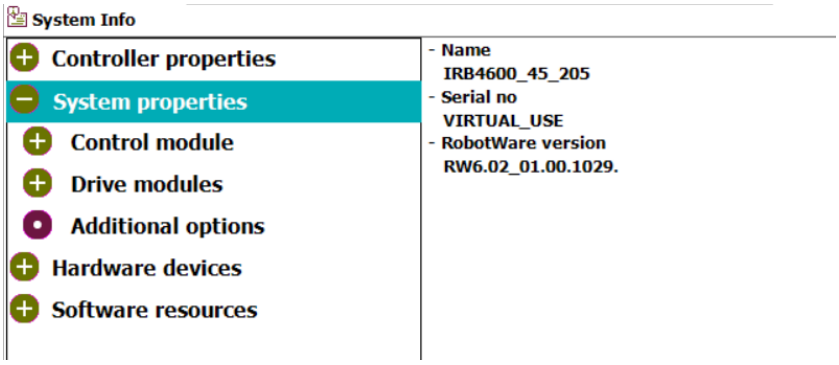


Fig. 7 – RobotWare version

You can then go to the **Add-ins** window of RobotStudio and install the **suitable RobotWare version [8]**, **[9]**. If you already have a RobotWare version installed, you can add another version by clicking on **Add**.

You can now create your first RobotStudio project by going to *File, New* and selecting *Project.* You can click the option **Include a Robot and Virtual Controller** and select the configuration you want **[11]**.

If you have not chosen this option when creating your project, it is not a problem, just go to **Home menu** and select the button **Virtual Controller Build Station [10]**.

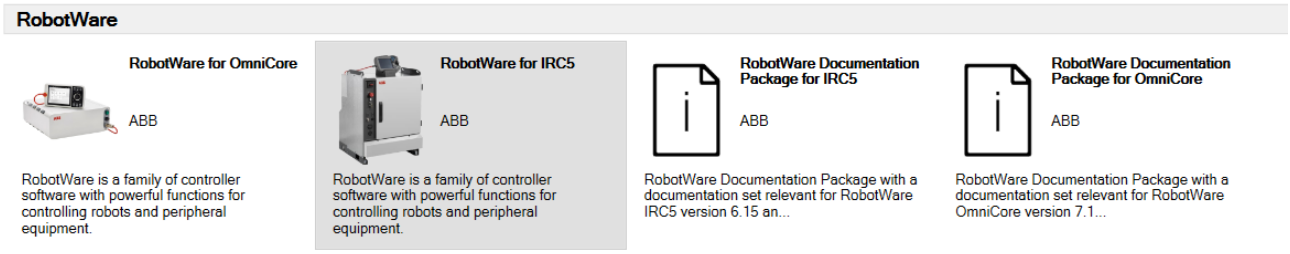


Fig. 8 – Available RobotWare

Fig. 9 – RobotWare Version

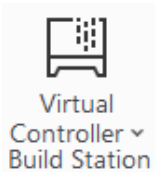


Fig. 11 – Include a Robot and Virtual Controller

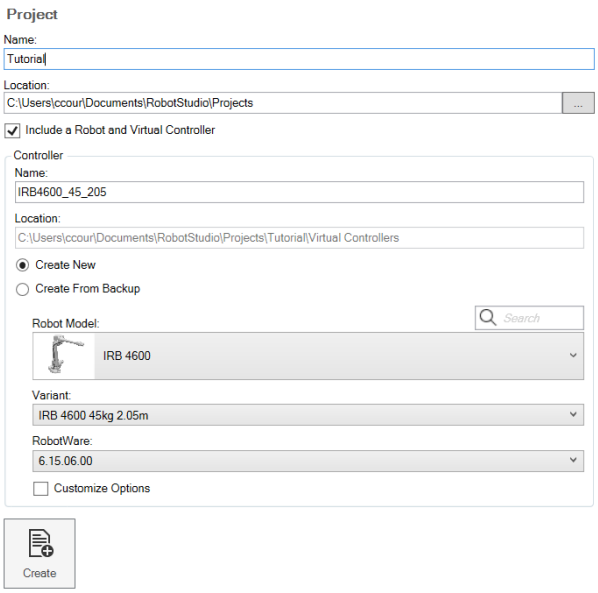


Fig. 10 – Include a Robot and Virtual Controller

You will then have your selected robot in the **graphic window**, at the position (0,0,0) in the world coordinate system **[12]** :

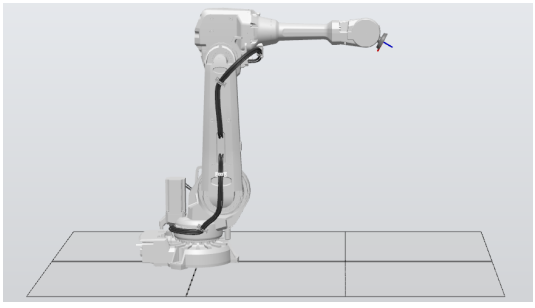


Fig. 12 – Graphic window

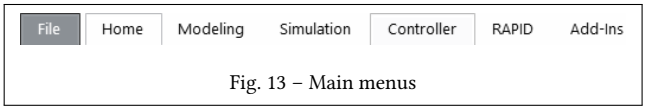


Fig. 13 – Main menus

3 Robot Studio Environment

In this section, I will describe the key features of RobotStudio by first covering each main menu, and then detailing some important features I used during my internship, particularly in the modeling panel, the controller panel, and the RAPID panel.

3.1 Main panels general descriptions

Apart from the file menu, I have explained in the RobtStudio setup section [5.9], there are 6 main menus in RobotStudio [13] :

- **The home panel** [14] is not the most useful from our point of view. Indeed, the main feature of this panel is to create RobotTargets, JointsTargets and Path which is useless for us because we already have a GrassHopper simulation and thus a raw RAPID program that already gather every target we need.
  However, **the ABB library button** as well as the **Import library button** are essential to simulate the robot that you want and bring some specific ABB product you can download via the Add-in menu (tools, conveyor, robotic cells ...) to your simulation. Except the **build station buttons**, the **path programming** and the **Graphics** (used to create other views and customize the layout) groups of buttons, the other features of this window come from other main panels.

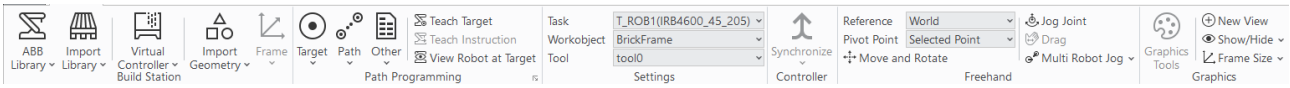


Fig. 14 – Home panel

- **The modeling panel** is essential as it will allow us to import our 3D models, integrate our tools, model directly some basic components inside RobotStudio and move them in our environment. Because this panel has a dedicated part in this section, we won't go further here.

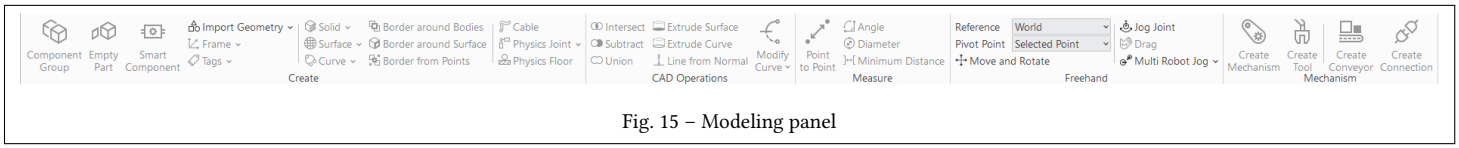


Fig. 15 – Modeling panel

- **The simulation panel** [16] is used to launch your simulation and to record it (pretty intuitively) and thus will be useful at the end of your work to get a professional final result. Its key feature is to **configure events** (with the signal you defined inside your controller) and launch actions based on these events. From this window, we can also **configure WorkObject frames and Tooldata**.

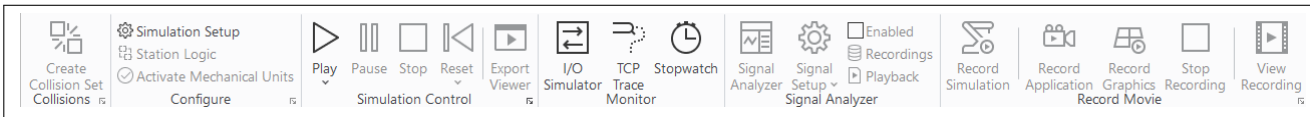


Fig. 16 – Simulation panel

- **The controller panel** is a key menu. From there you will be able to select the operating mode and thus jog the robot (manual mode section). You can also configure your simulated ICR5 system by creating new signals, load configurations and backup your system. As for the modelling window, I dedicated a section for this panel.
- **The RAPID window** enable you to create your own programs and modify them. It is a development environment and thus have the same features as many other IDE. You can debug your code with breakpoints, move your program pointer, run in step by step mode ... In this section, I will explain the RAPID code structure.
- **The Add-in gallery** [17] is used to install libraries for your simulation. Every ABB Robots can be found there, as well as 3D models for tools, documentation, robotic cells, RobotStudio optional features ... If you use some specific equipment you have not model yet, check this gallery, especially the end of the models' gallery where you can find some tools, conveyor and external axis.

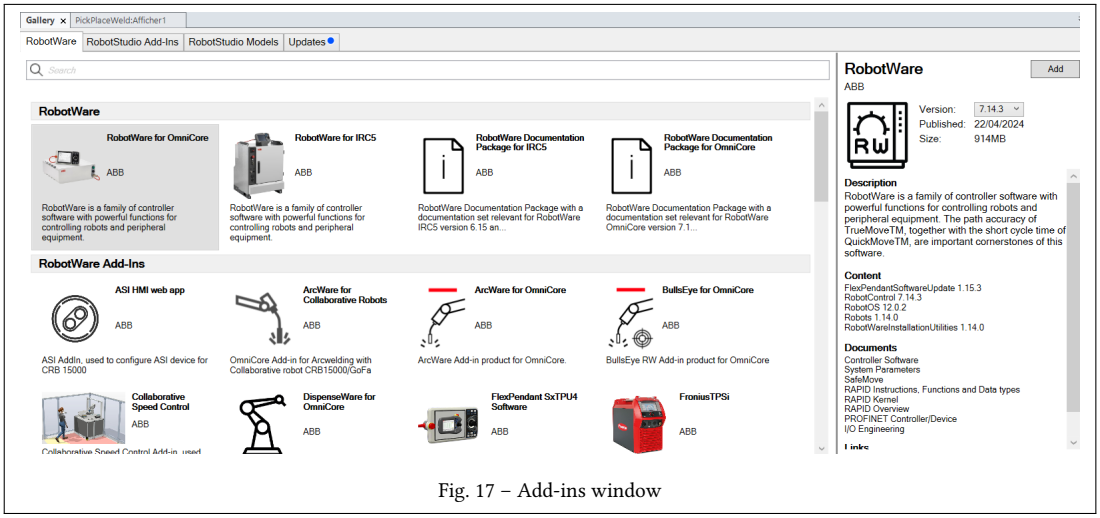


Fig. 17 – Add-ins window

3.2 Modeling Window

First, let's describe the controls to rotate and move inside the central view [18].

- **To rotate**, hold **control + shift + left mouse button**
- **To move**, hold **control + left mouse button**
- **To zoom**, you can use the mouse scroll wheel, you can also press **shift + right mouse button** to create a zooming window inside the view.

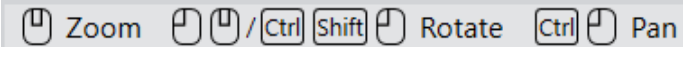


Fig. 18 – Moving commands

For every window, you have a sidebar that shows important information to the user. In the modelling window, we can access **the layout** [19]. The layout, displays every mechanism and components of your simulation (ABB robot is considered as a mechanism). You can select an item by clicking on their name, and selecting several of them by holding shift and left-clicking. Right-clicking on an item shows a lot of different features we are going to explain here.

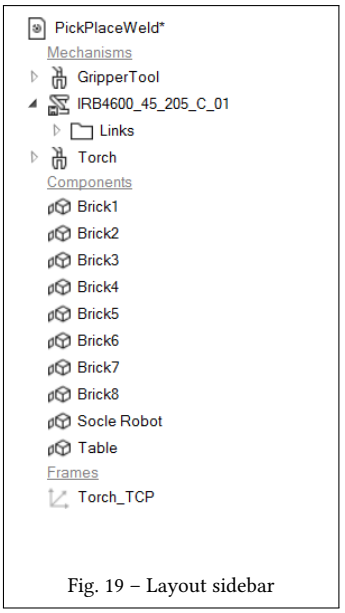


Fig. 19 – Layout sidebar

You can create pieces with the **Create button group** [20]. Using RobotStudio is suitable for simple shapes, but you will always prefer to use a proper CAD software such as Rhino and import your work inside RobotStudio. For example, using the **Create Box option**, you can create a parallelepiped type of shape [20].

When you have created a normal component such as a box, you can then apply several interesting operations by right-clicking on it in the layout sidebar. Here are some useful command :

- You can change their **local origin** (*right click, Modify, Set Local Origin*) [21], [27]. Indeed, RobotStudio and mechanics in general are **all about frames**, and everything is relative to the reference frame we are using. When you create an object, its local origin will be the origin point of the reference frame chosen during its creation (by default, the world reference). Every object has a local frame relative to the world frame and is located based on the position of the local frame within the reference frame coordinate system.

  Thus, when creating any object, any translation will be applied to the local frame of the object, and then the position of the object will be updated depending on the position of its local frame. This has some consequences !

  For example, if you want to attach your object to the flange of the robot and then update its position, the local frame of your object will align with the local frame of the flange. Your object will be translated, but will remain in the same relative position to its local frame (and might be flying around 1 meter away from the flange). When you select an object, its local frame will appear so you can check if everything is set up correctly.

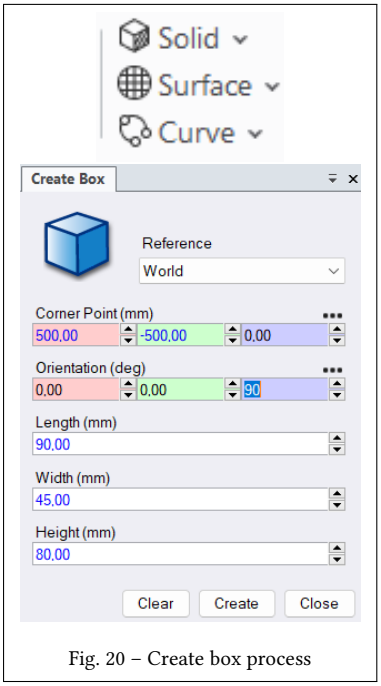
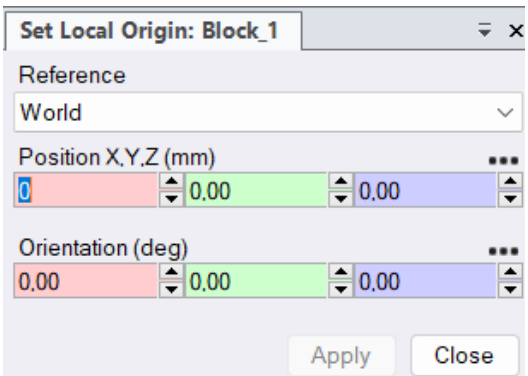
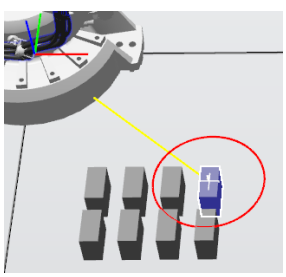


Fig. 20 – Create box process



Set Local Origin



Pre-visualization

Local Origin changed

Fig. 21 – Set Local Origin process

- You can also **duplicate an object** [22] by offsetting it repeatedly about an axis (or more) or rotating it about another. **Keep in mind that duplicating an object will also duplicate its frame properties, including its local frame definition,** so try to set the local origin before duplicating a piece. Before confirming the duplication and any other placement option, you will have a pre-visualization on the graphic window [23]

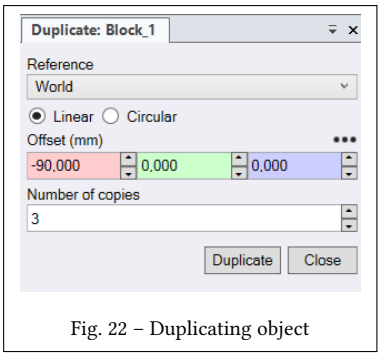
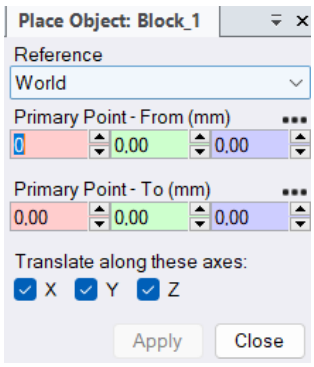


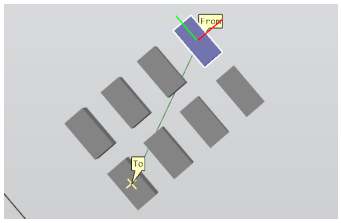
Fig. 22 – Duplicating object

Fig. 23 – Duplication pre-visualization

- I have used a lot **the placed by one point** [24] feature you can find in the placement suboptions. For our basic usage of the modelling panel, you won't need much more to accurately place your objects [26]. As for the duplicate option, you will have a pre-visualization of the composed movement applied on your object.



Set by one point



Pre-visualization

Fig. 24 – Place by one point process

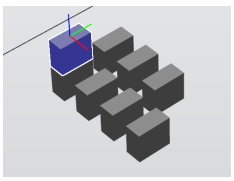


Fig. 25 – Object placed

- Finally, you can **attach** your object position to another component of your simulation like the flange of your robot, or your tool. It is a very interesting feature we will need in the final tutorial. You can choose to update the position (local frame of the object at the same position as the local position of the other component) or keep the same distance and rotation compared to their initial attaching position and orientation.

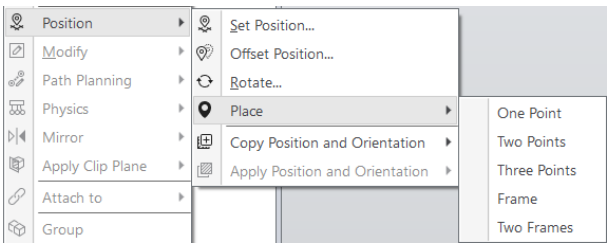


Fig. 26 – Place sub-options

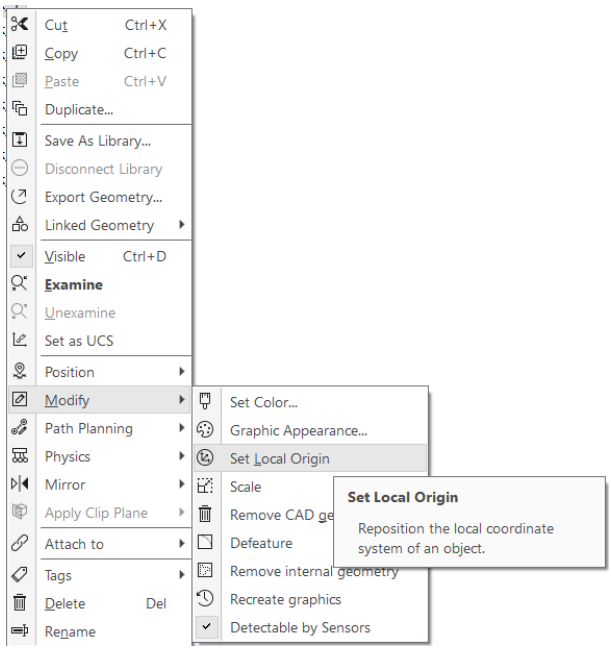


Fig. 27 – Local origin change menu

For these three options and a lot more of them, you will need to **precisely select points on your objects**. For this you need to modify the **snap mode option** [27] where you can choose the object snap option for example. You can set up a point by left-clicking on an input coordinate field (**your mouse pointer will then change into a cross**) and visualizing the appropriate point with the snap pre-visualization [29].

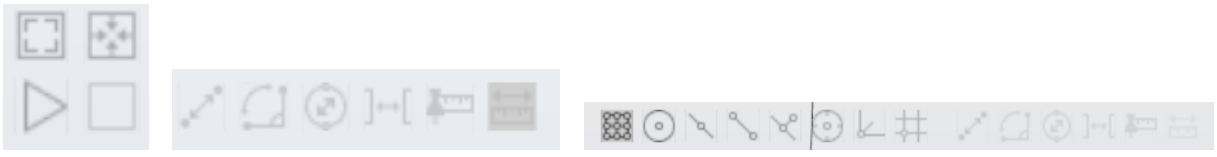


Fig. 28 – Graphic window quick menu

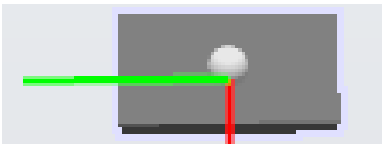


Fig. 29 – Snap point visualization

Here are the other operation you can do on a regular object :

- Cut, copy and past them.
- Save and export their geometry
- Make them visible/invisible, zoom on them with the **examine option** and set them as **UCS** (user coordinate system).
- **Change their appearance** with the **modify button**
- Give them a physic (never used), mirror, group, rename, tag and delete them.

The second type of component you can simulate are **mechanisms**. You can create mechanism from the **mechanism panel** you will then be able to fully integrate tools, conveyor and more (tool integration is fully explaned in the tutorial part). When you right-click on a **mechanism** such as a tool or a robot, you have other specific options such as :

- **Replace it or modify it**. Replacing a Robot for example will not destroy all your work, everything will be adapted to the new robot if you didn't choose the right one in the first place (but you will need to alter your RAPID program as the quaternion orientation are different from a robot to another).
- **Jog joints or jog in linear mode, modify the current robot configuration and place it in specific positions, such as home** [30]. It is really useful when you have tested a code that placed a robot in a bad position, and you want to jog it quickly without using the virtual FlexPendant.
- Align the tool center point, save the TCP position as a Robtarget or the pose as a JointTarget.

Finally, if you need to quickly move an object without precision, there is another feature you can use : **the free-hand panel** [31]. You can select the reference frame and **move and rotate** button. Then, when you will select an object, gimbals will appear to drag and rotate items. You can also use **the jog joints** button, select and drag a joint intuitively.

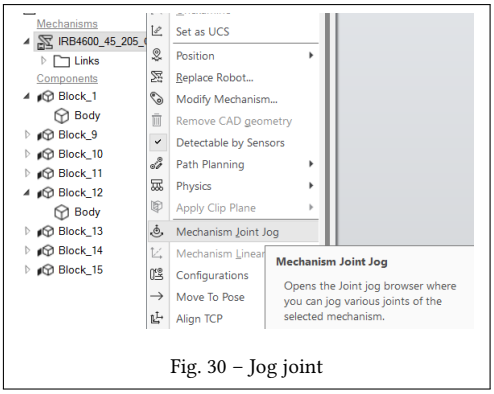


Fig. 30 – Jog joint

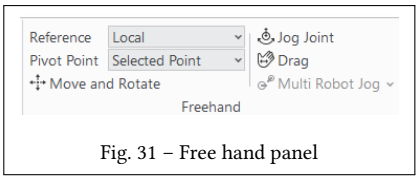


Fig. 31 – Free hand panel

3.3 Controller and simulation window

This window offer various parameters that can be very complex. I have only use the outer part of the iceberg, and I am going to introduce you the basic. For any further information, read the official manual in the complementary resources section [5.9].

First let's see what are the buttons of this window and what can we do with them [32] :

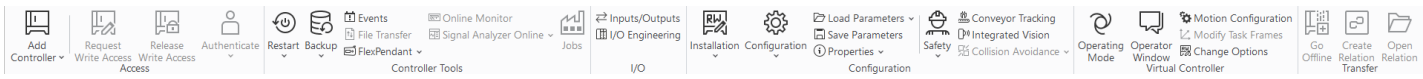


Fig. 32 – Controller panel

- The first group of button is used to connect the real ICR5 controller, authenticate and write directly some modification on it. I have not used these options.
- Then we have the **controller tools** used to simulate the **FlexPendant** (remote controller of the robot) and to restart the system with the **restart button** when you have done some modification that requires it. The FlexPendant utilization is completely described in the next section of this guide. You can also back up your system if you previously have done a safety save, for example.
- The **I/O** panel is used to visualize the signals you have defined for your simulation, you can also change their value. You can also access the I/O engineering for advanced signal input and output configuration (not used in my case).
- The **configuration panel**, is the one that I mostly used. From here, you can install another version of RobotWare with the **installation button** you can also **configure your ICR5**. Indeed, with the **configuration** button and the **I/O system**, the window [33] will pop up, and you will be able to visualize and navigate through every signal installed in your system. With a right click on the table, you can **add a new signal**. The instance editor will then pop up [34] where you can **name your signal, define its access level and most importantly set the type of signal. Often you will use digital output.** After the creation of a new signal, you will need to restart your controller (finish the configuration of every signal and then restart). I have not used any of the safety options.

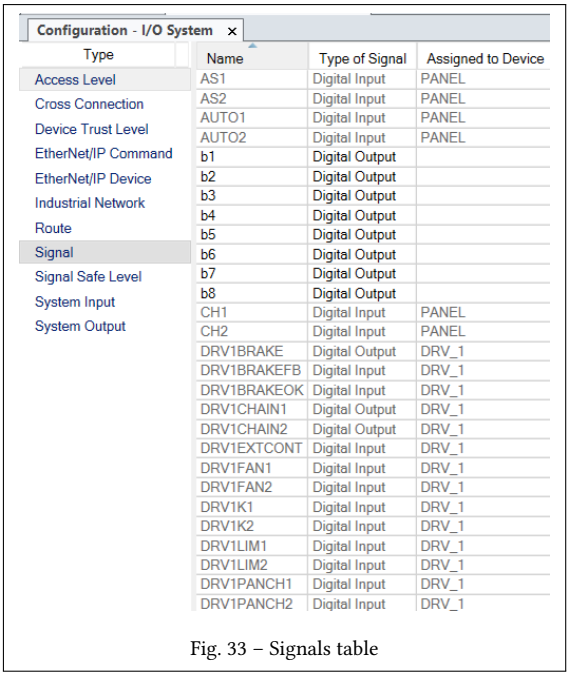


Fig. 33 – Signals table

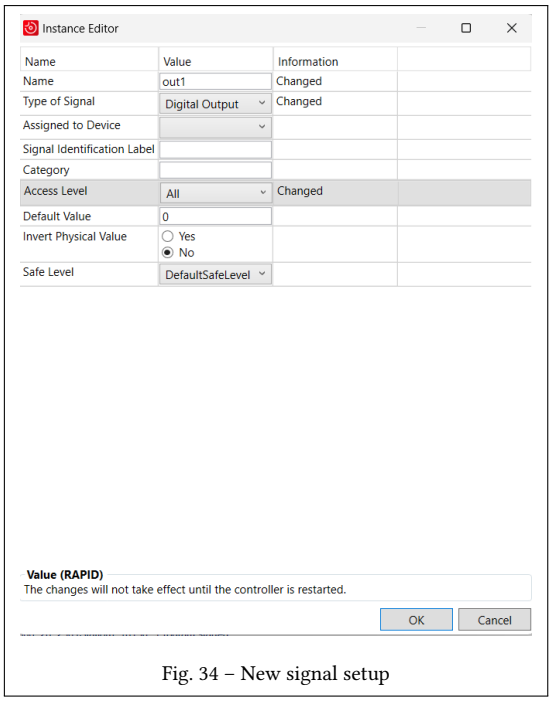


Fig. 34 – New signal setup

The last panel I have used is the **Virtual controller panel** and the **Operating mode** button used to switch between the three mode : automatic, manual and manual 100% (explanation in the manual mode section [**4.1**]).

Finally for the controller window, you can access **the Path&Target** sidebar [**36**] that will enable you to configure or modify Work Object Frames by right-clicking on them or on the **Work object & Targets title** (more explanation about this in the manual mode section [**4.1**] and the tutorial section [**5.1**]).

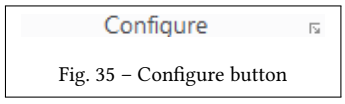


Fig. 35 – Configure button

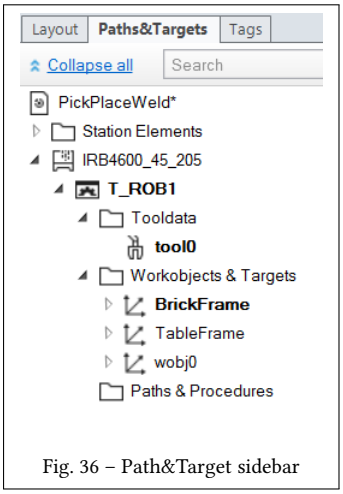


Fig. 36 – Path&Target sidebar

After creating signals and restarting the ICR5, you can now use them with the **Simulation window** and the **configure button** [**35**] to associate them with actions in your simulation (attach object when the signal is active, change a mechanism, move an object ....). A window like [**37**] will pop up. We will see some example of action and signals management in the tutorial section.

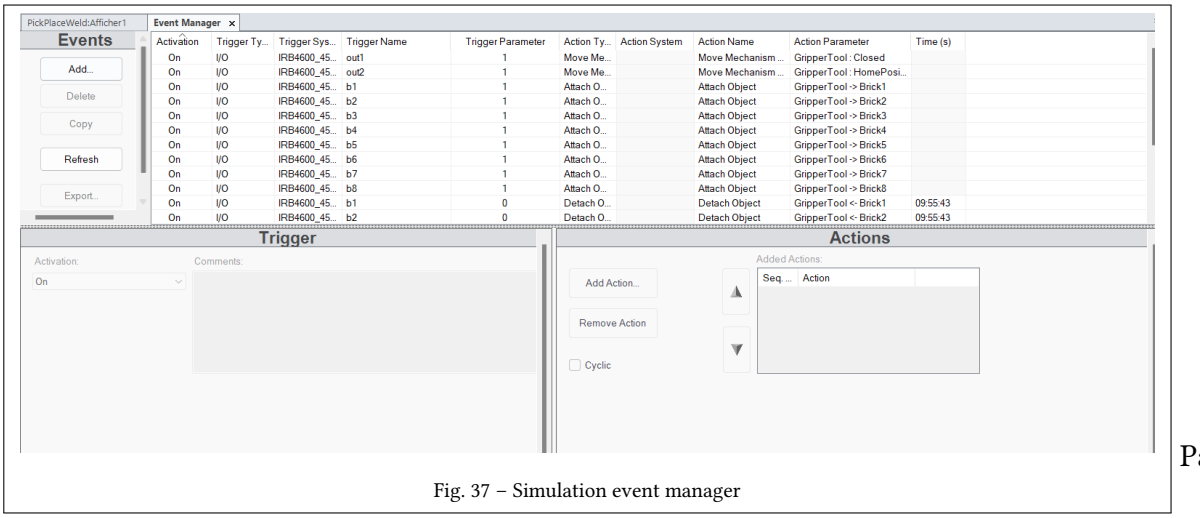


Fig. 37 – Simulation event manager

Fig. 38 – Trace button

Another interesting feature with this window is to **visualize the trace of the TCP**. You can activate this feature from this button [38]. Initially, the default tool is the **flange of your robot**. If you want to change the traced TCP, you will need to **declare a tooldata in the path&target menu** (in the tutorial). You can also associate the appearance of the trace with the value of a signal (for example, torch welding or not) to differentiate pure movement actions and work actions, for example [39].

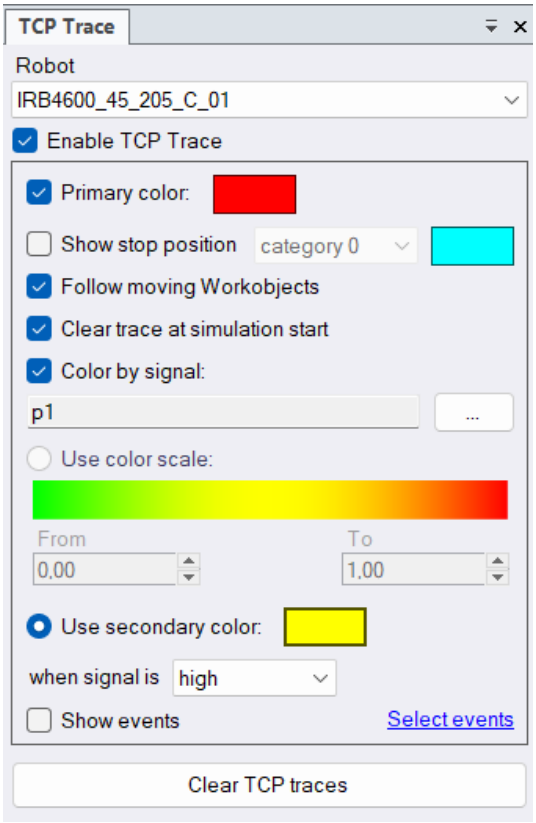


Fig. 39 – TCP trace options

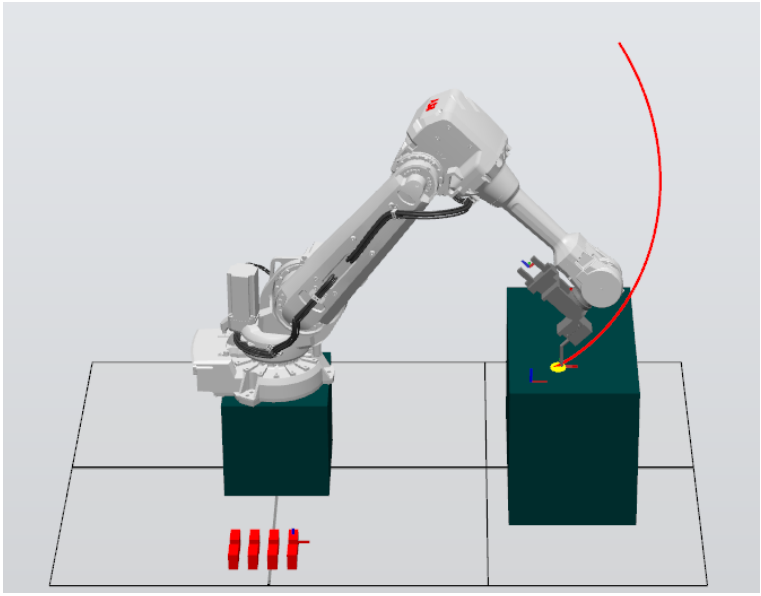


Fig. 40 – TCP trace options

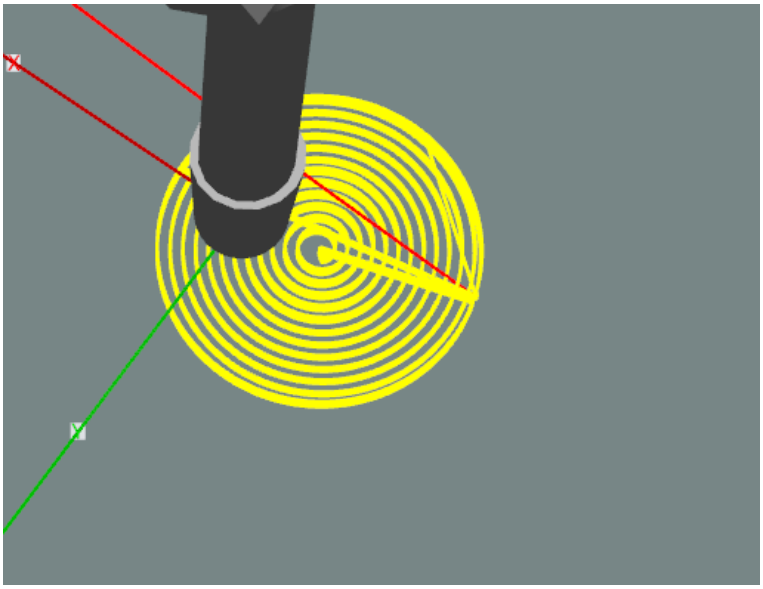


Fig. 41 – TCP trace options

3.4 RAPID structure

RAPID is a high level programming language (far away from computer (assembler) or machine (G code) language) to makes it understandable by humans. However, the raw codes that will give you GrassHopper plugins is not really intuitive for a human.

In RAPID, programs or **TASKS** are sets of module, where **MODULES** are themselves sets of **instructions** that will describe the robot working. Instructions are used to command the robot to do specific actions such as moving, setting an output ... They take arguments (what logically needs the instruction to work properly) for example for a move instruction, you will need a position, an orientation (gathered in a RobotTarget), and a tool.

Instruction are used to command the robot and sometimes needs arguments that are generated by functions. **Functions** are sequences of RAPID code that generates an output of a certain data type that is not an instruction but can be used by them. For our basic tutorial application, we will need really few of them. Instructions and functions are completely different instances of your program. In fact, functions are one of the three types of what is called a **Routine** in RAPID.

There are two other types of routines:

- **Procedures** which are subprograms, like sections in your report, for example. You can create them and call them anywhere in your code. They represent a sequence of instructions that have a meaning together. They will be really useful to create sense in our programs.
- **Trap routines** that are special procedures used when a specific event occurs (for example, a human is detected in the robotic cell). It is an advanced feature I have not used.

Routines have one common point : they are all sequences of RAPID code, that have a sense together and uses arguments. Procedures are sequences of instructions and generic code, trap routines are sequences of instructions and generic code triggered by a flag (generated by a specific event) and functions are sequences of generic code that is not producing an instruction.

When coding, you will have to declare and play with different variables with different **data types**. Like instructions and functions, there are plenty of data types, but they are only 3 kind of data :

- A **variable** (like in every other programming language)
- A **persistent** that can be seen as a global variable in Python.
- A **constant** that cannot be changed after its declaration.

Finally, there are two types of Modules : normal ones and **system modules**. Programs that use different modules needs to include them at the beginning of their code (like python libraries). **Only one module of a program can have a main procedure**. However, **system modules don't need to be included**, that's why they are commonly used to declare tools and objects that are needed by a lot of other modules but are not subject by many changes.

In the tutorial part, you will structure your GrassHopper RAPID program to apply all of this. Your program will have a structure such as [42] with a task, one module with three procedures and two system modules.

Apart from what I will detail in the tutorial part, I won't go further with RAPID program. You are free to read the official manual to cover your needs (you can find them in the complementary resources section [5.9]).

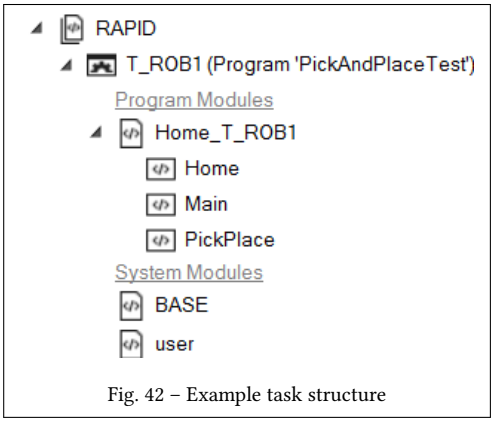
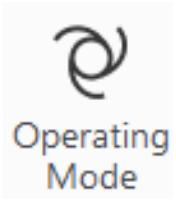


Fig. 42 – Example task structure

4 Manual Mode with RobotStudio

In this part of the guide, we will discuss how the manual mode works in RobotStudio and on the real ABB Robot. ABB has replicated the exact same FlexPendant (the remote controller for ABB robots) in RobotStudio.



4.1 Launching manual mode

In the first place, you need to find the **operating mode button**, which can be found in the **Controller menu** [43].

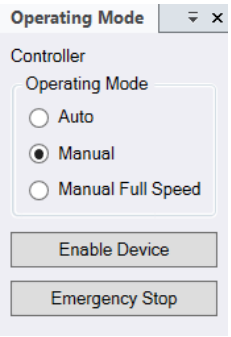


Fig. 43 – Operating mode

There are 3 types of mode :

- **Automatic mode** that enable you to launch programs on your simulated robot from the RAPID menu and to launch programs from your production window without having to press the security of the FlexPendant. Be careful, when launching a program on the real IndexLab robot in automatic mode, speed declaration in RAPID program encapsulate several speeds that can be faster than your GrassHopper one.
- **Manual mode** (the one we are interested in). You will need this time to press the security each time you want to move the robot by jogging or programs you coded. You won't be able to accelerate the speed of the robot above the limits of your program's code.
- **Manual full speed mode** exactly like manual mode, but you will be able to accelerate the speed of the robot above the limits you set in your code with the Quick set options, for example.

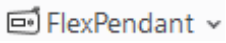


Fig. 44 – FlexPendant

After enabling the device in manual mode, you will then click the **FlexPendant button** [44], which is still in the controller page. A virtual FlexPendant replica will then pop up [45].

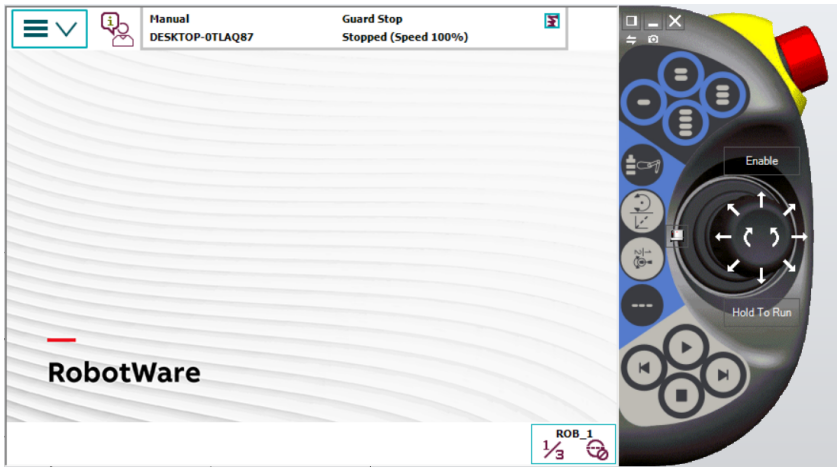


Fig. 45 – Default view

4.2 FlexPendant basics

Let's now pay attention to the first display of the FlexPendant [45].
We can see two separate sides : the screen side where you will be able to navigate through the different menus proposed by ABB and the button side we will discuss later.

On the initial screen side, we can see in the top left corner a button to access the different main menus [46]. In the top, we have access to various information gathered in a state view [47].
It indicates :

- **The operating mode** you are using.
- **If motors are currently activated or not** (depending on whether you are pressing the security of the FlexPendant or not and in this case if you have pressed the **enable button** that simulate the security system).
- The serial number of the robot you are linked to.
- **The percentage of speed you are using**, 100% being 100% of the actual speed you coded in your program at the current instruction
- And finally, a shortcut to **the event log menu** (we will see this feature later).

In the bottom right corner, there is also the **quick set button** [48]. From this button you will access different options related to the mechanical unit you are controlling, the speed, the program execution type, the speed and the current task. The button already displays the **name of the robot** you are controlling (ROB_1 is always the main Robot name linked to the ICR5 controller), **the type of movement** and if you are using **the increment option** or not.

Thus, one can see that from the initial view, without opening any menu of the FlexPendant, you can already control and visualize the key parameters of your robot.

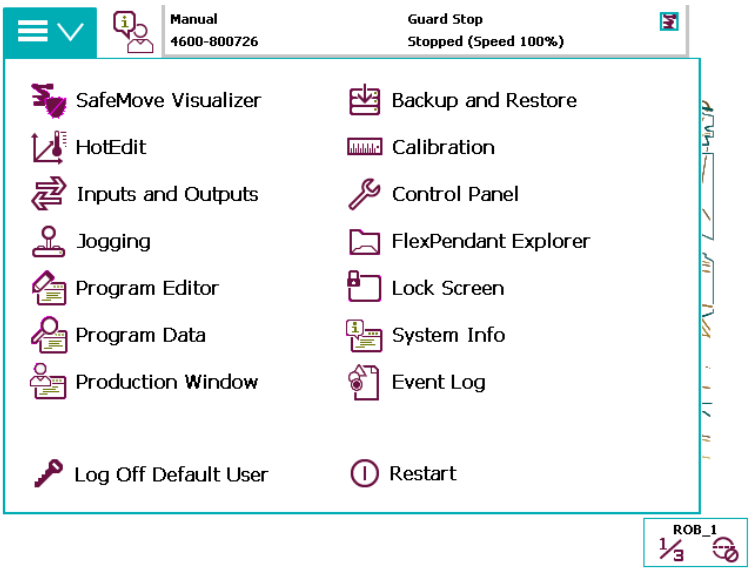


Fig. 46 – Main menu

Fig. 47 – State view

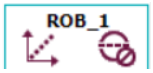


Fig. 48 – Quick set button

Let's now go through the different menu from **[46]** :

- **SafeMove Visualizer**

  This menu is available if your system has the safe move option. I have not used it, so we won't go further with this option.

- **HotEdit**

  HotEdit enables you to offset the Robtargets of a loaded program by offsetting or reorienting their frames, when the program is running. I have never used it, so we won't go further with this option.

- **Inputs and Outputs**

  This menu is here to visualize every input and output signal defined in your system, see their current value and their type. You can select the type of signal you want to see (in our case, we will use only digital outputs). You can also change the current state of this signal and see what it does on your system (for example, switch digital outputs linked to your gripper to see what happen). **[49]**.

- **Jogging**

  Jogging menu enable you to visualize the current orientation of every joint or the position and quaternion orientation of the TCP (tool center point) depending on the movement type you are using (joint movement or linear movement). You can also change various property related to the movements of your robot, I have detailed everything in this **[dedicated part]**.

- **Program Editor**

  With this menu, you can program in RAPID from scratch or load a program you want to change. It is not really convenient, and it is more suitable to program everything on RobotStudio. It can still be useful to make some tiny changes, but if the RobotStudio simulation is perfect, you won't need it. With **debug button**, you can verify your program. Any error in the program will generate a pop-up you will need to acknowledge. To navigate quickly into your code, you can also use button such as **PP to main**, **PP to cursor** (meaning program pointer to ...). The program pointer is the tiny purple arrow at the left of your code. You will also see it on the production window. **[50]**.

- **Program Data**

  Program Data helps you to visualize, access and configure every data declared in your programs or saved by default in the robot. Again, you can edit your data from this menu, but it is for me more convenient to do it from RobotStudio and simulate any changes. Indeed, changing a confdata (configuration of the robot), tooldata or workobject data can drastically change the behavior of your robot, and it could be dangerous. **[51]**.

- **Production Window**

  Production window is used to run the program you have loaded and edited in the program editor. You can also directly load a program from your USB driver. Pay attention to the Program Pointer. The program will start from the position of the Program Pointer as soon as you press the play button. Don't hesitate to press **PP to main button** as it will put the pointer to the beginning of your main routine, which is normally the beginning of your program if you respected the usual RAPID program structure. **[52]**.

- **Backup and Restore**

  I have never used this menu, but I learned that it is an important feature on the FlexPendant. You can save the current state of your system in backup folders (inside the ICR5 memory for the real robot or inside your computer if you are in RobotStudio) by pressing the **Backup button**. Then if you are in a bad position meaning you have set bad parameters everywhere without understanding everything and now your system is not responding correctly, or if you did a wrong calibration, or if you need a specific system configuration that you have backed up, you can use the **Restore system button** and select the convenient configuration. **[53]**.

- **Calibration**

  On the real system after switching on the robot, you will need to calibrate it in this menu. The automatic routine is necessary to calibrate every joint motor and mechanical reducer. Meaning that before switching off the robot at the end of every session, you need to put every joint close to 0° (perfect precision is not needed as you want to jog in the good engine revolution) to have a correct synchronization for your next calibration. I have never used the advanced calibration methods. **[54]**.

- **FlexPendant Explorer**

  This is the file explorer system of ABB. Everything is detailed on figure **[55]** **[56]**. You will use it to load programs from your USB key to the production window.

- **Control panel**

  From this menu, you will be able to configure the global appearance and properties of the FlexPendant. You can also set the language and calibrate the touch screen. What is more interesting is that you can set the 4 programmable keys, with the **Progkeys option**. You can also configure the commonness of your signal (to access it more easily with the Inputs and Outputs menu), visualize, **Add and define** new signals. For this, go to the **Configuration option** and press **Add**. **[57]**.

- **System Info**

  From this menu you will access any details you need on this system, from the version of RobotWare to the number of hours each joint motor has been used since comissioning.

- **Event Log**

  Finally, the event log displays every system message. Basically, every action you are doing will be referenced here so you can understand every step you followed leading to a dysfunction, for example. **If something critical happen, you will receive a pop-up you will need to acknowledge**. After the acknowledgment, it can be seen at the event log menu. Use the **View button** to filter the messages (there are a lot of them !).

In the next subpart we will detail every button of the FlexPendant, the jogging menu, the quick set menu, and some details on calibration processes, data types and mechanical unit's explanation.

If you need further information on other specific properties, read the ABB official manuals available in the complementary resources section [5.9].

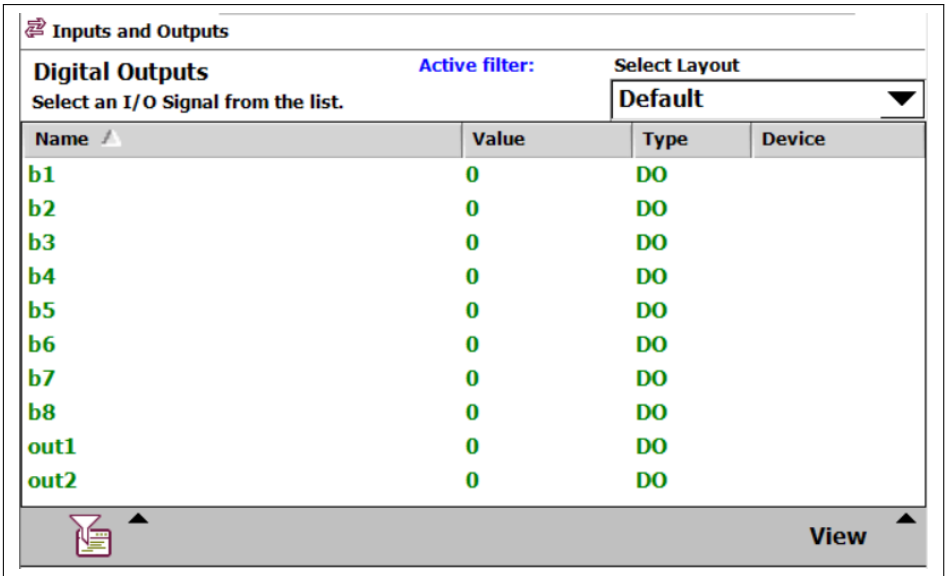


Fig. 49 – Inputs & Outputs

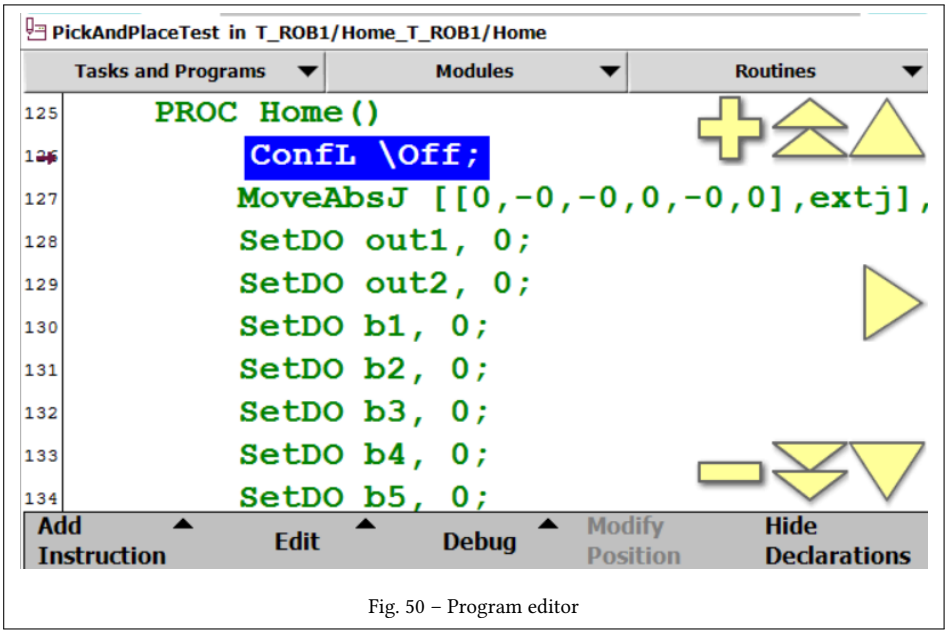


Fig. 50 – Program editor

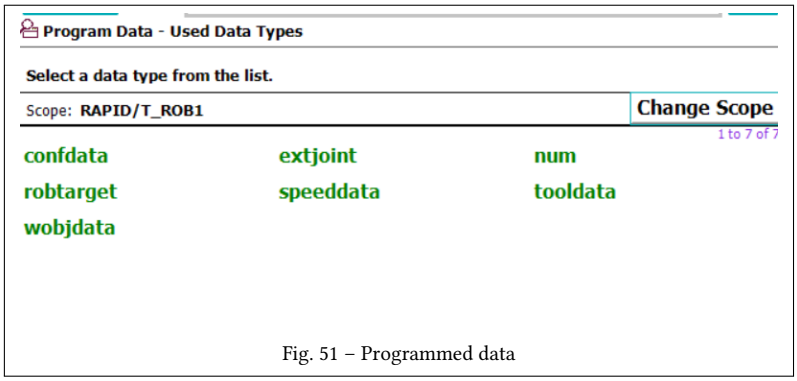


Fig. 51 – Programmed data

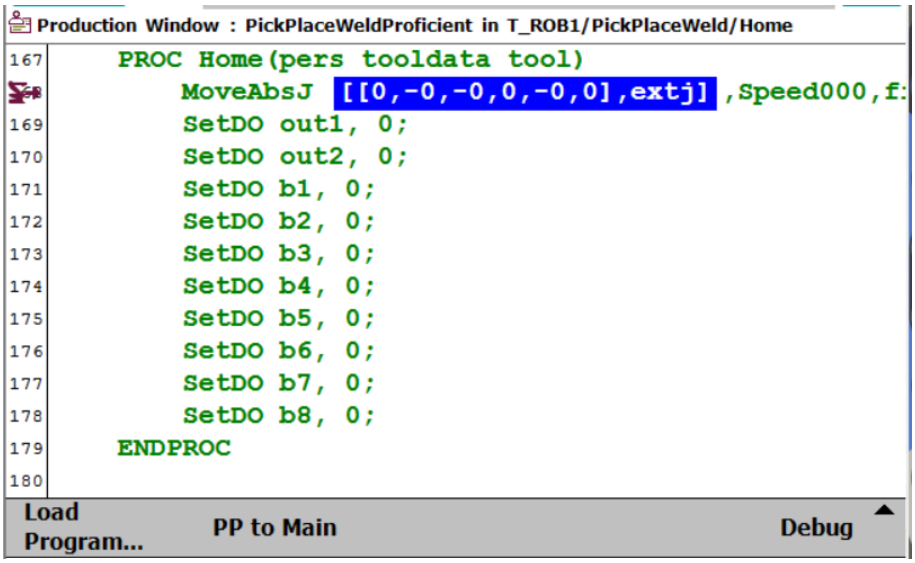


Fig. 52 – Production window

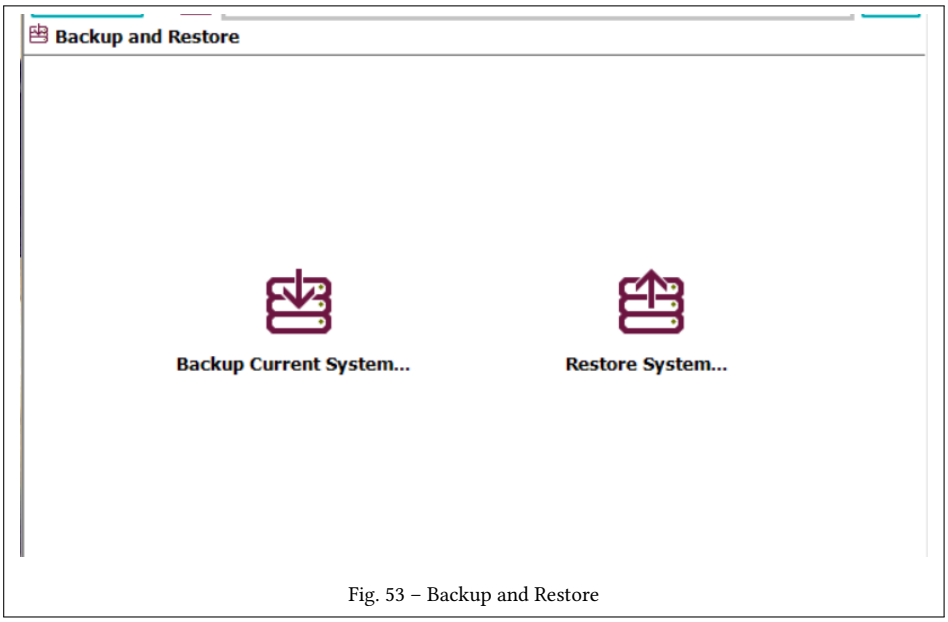


Fig. 53 – Backup and Restore

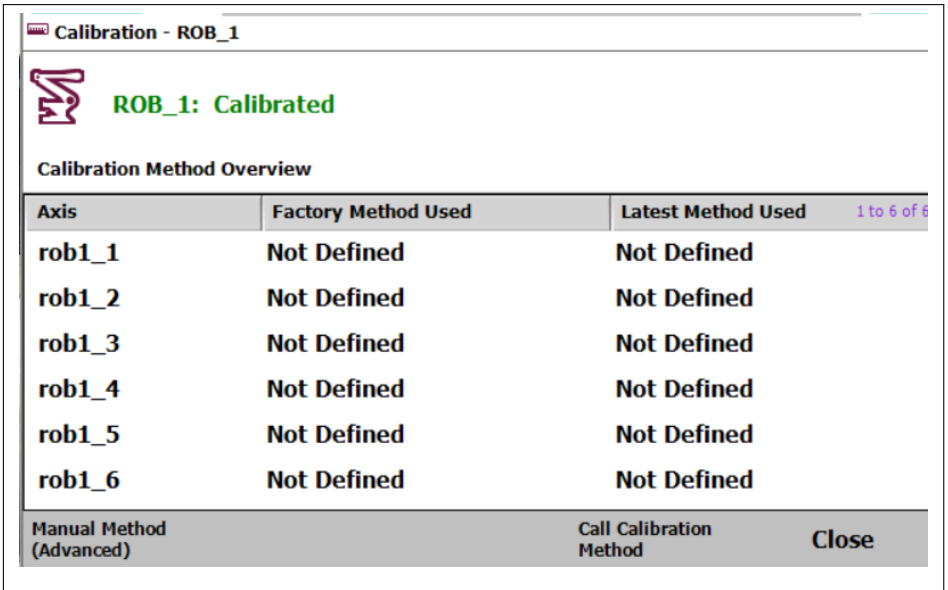


Fig. 54 – Robot Calibration

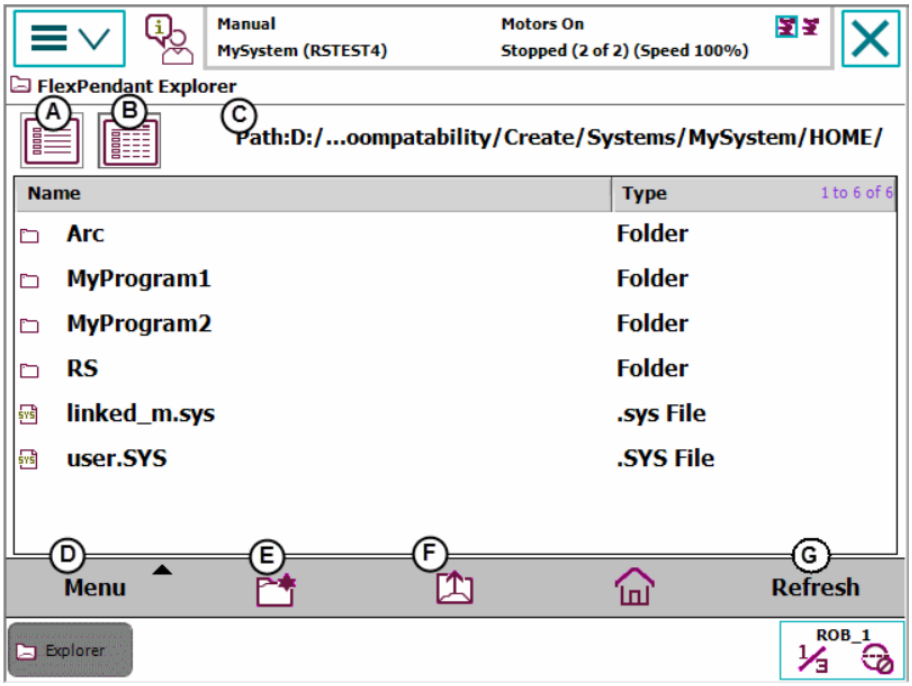


Fig. 55 – FlexPendant explorer

A	Simple view. Tap to hide type in the file window.
B	Detailed view. Tap to show type in the file window.
C	Path. Displays folder paths.
D	Menu. Tap to display functions for file handling.
E	New folder. Tap to create a new folder in current folder.
F	Up one level. Tap to change to parent folder.
G	Refresh. Tap to refresh files and folders.

Fig. 56 – FlexPendant explorer detail

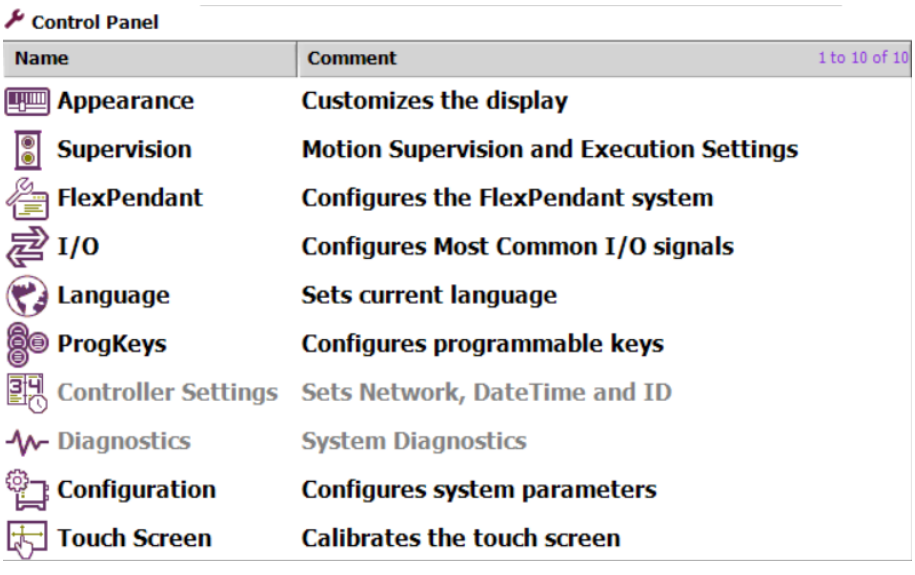


Fig. 57 – Control Panel

4.3 Jogging and Calibration

There are two fundamental types of movements in robotic : linear movements and joint movements.

- **Linear movement** in robotics refers to the controlled motion of the Tool Center Point (TCP) along a straight path in the X, Y, or Z directions, based on input from the joystick in our case. When executing linear movements, the robot's control system coordinates multiple joints simultaneously to ensure the TCP follows the desired trajectory. Unlike individual joint movements, linear movements require precise synchronization of several joints to maintain the straight-line path of the TCP. This type of movement is essential for tasks that demand high tool accuracy, such as precision assembly, machining, and detailed inspections.

  This type of motion is not "natural" for example as a human you will never make a straight line motion with your finger even if you are really focused (it will be close but not accurate). Thus, these types of motion will generate more unnatural position for your robot and may lead to singularities (the robot won't be able to perform the requested movement). Any movements close to singularities will generate an error pop-up you will need to acknowledge when jogging.

- **Joint movement** is more natural for the robot, and singularities will be less common. With this type of movement, each of the joints are commanded to follow a non-linear path. It is interesting to use this type of movement when you don't need tool accuracy (but speed) and when you want to place the robot in a certain position (for example, home position).

Now that we know what are the two types of movement we can select to jog the robot, let's see what are the actions we need to do on the FlexPendant to select the type of movement and move the robot. The left side of the FlexPendant is composed of different buttons [58].

First we have the **enable** button that mimic the security of the physical version. You need to press it to switch on the motors in manual mode every time you want to move the robot, and even when a program is being played. In RobotStudio it is a state button, but physically you will need to maintain the pressure.

Next, we have **the joystick** that has 3 degrees of freedom (horizontal, vertical and one rotation). Consequently, you will be able to control joints 3 by 3 for the joint's movement. You can do some diagonal inputs that will command 2 of the 3 joints you are working with. The more you push the joystick out of his balance position and the quicker the movements are, it takes some time to be used to the joystick control.

On the left of the joystick, there is a tiny button to mimic the physical front interface of the ICR5 controller [59]. This **ICR5 button** is useful if you want to switch from manual mode to manual 100% or automatic mode. Be careful about automatic mode and manual 100% as the speed limits will be different. For security reason I advise you to test your program step by step in manual mode, then to play it still in manual mode, and finally you can try using automatic mode.

Fig. 58 – FlexPendant button side

Still on the left, there are 4 really important buttons [59].

The first one switch **the mechanical unit** you are controlling. Mechanical unit designate every mechanical system linked to your ICR5 controller. Indeed, some ICR5 can control two robots and many external axes (like a bed that can be rotated or a conveyor). The main mechanical unit of your ICR5 is the main robot called ROB_1 by default. Index lab ICR5 is only linked to one robot, so this button is not really useful in our case.

The second button is the **linear motion button**. If you are currently jogging the robot in a joint movement, and you press this button, you will switch to linear mode. This will change the quick set button in the bottom right corner of your screen as well as the jog window display [60]. You will now see the coordinates of the TCP of the tool you are using in the chosen reference frame. You can change the reference frame with the **position format button**.

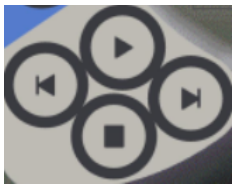


Fig. 59 – Button detailed

You can also visualize the quaternion orientation of your TCP (you can select an Euler angle visualization with the position format button). If you use the joystick, you can now move the TCP, but not change its orientation. If you press again the **linear motion button** you will enter the **Reorient motion mode**, and you will be able to change the orientation of the TCP by rotating about the X, Y or Z axis of the reference plane without moving the TCP (really important feature).

Keep in mind that we are moving and rotating about the axes of the reference plane, thus, shifting from the world default frame to the tool frame can change everything [61] [62]!

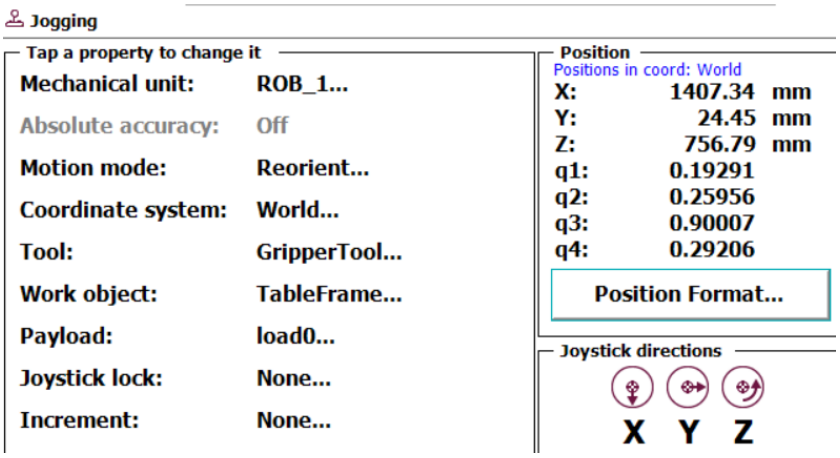


Fig. 60 – Linear mode jogging window

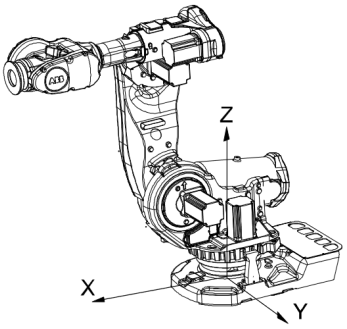


Fig. 61 – World frame

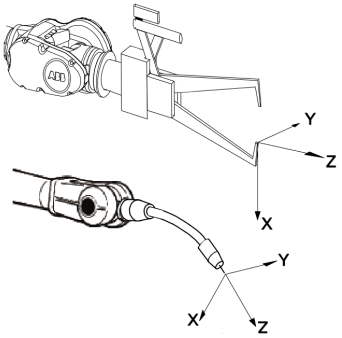


Fig. 62 – Tool Frame

The third button is **the joint movement button**. When this button is pressed, the jog window will change [63] and you will be able to visualize in degrees the orientation of each joints about their pivot axes. The quick set button in the bottom right corner will change to [64] and you will be able to command joints 1, 2 and 3 with the joystick. Another press on the button will switch to the control of joint 4, 5, 6 and will be indicated again on the quick set button display [65]. Here is the detail of the 6 joints for a common 6 axes ABB robot [66]. If the diagonal movement is annoying you or if you want to control only one joint, you can lock the joystick in 1 or 2 directions by selecting the **joystick lock property** on the jogging window.

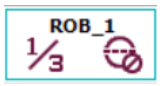


Fig. 64 – Quick set display 1

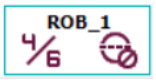


Fig. 65 – Quick set display 2

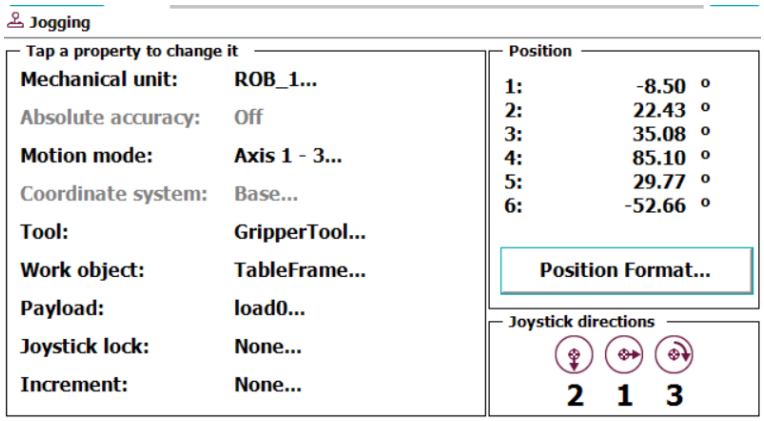


Fig. 63 – Joint jogging window

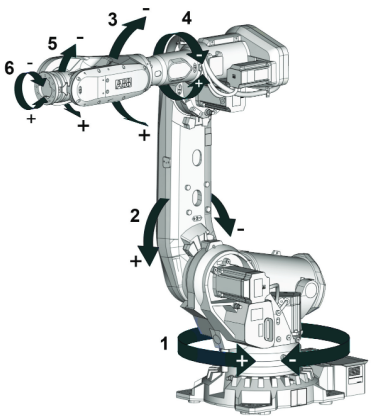


Fig. 66 – General 6 axes ABB robot

Finally, the last button is used to enable **incremental movement**. Consequently, movement won't be continuous but incremental to anticipate more easily the movement you are commanding and to have one more security layer. You can set the increment property by clicking the **increment property** in the jogging window [67]. Additionally, the robot will be way slower and more accurate.

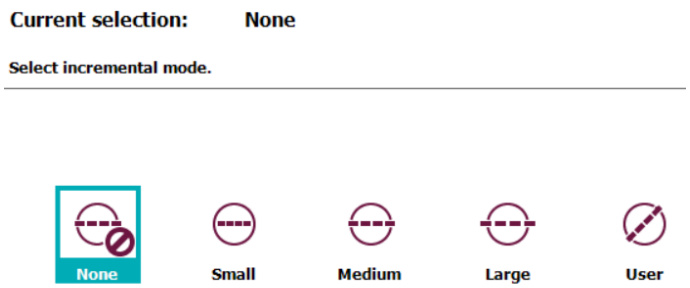


Fig. 67 – Increment settings

The second set of button from [59] is used to run a program on the production window, step by step or continuously. You can stop at any moment with the square button. **It is better to stop with the square button than releasing the security, as a security stop is stressful for the motors/reductors.**

The last set of buttons are **the programmable keys** you can set up from the control panel window as explained earlier.

Now let's focus ourselves to **the Quick set menu** I mentioned in earlier subparts [68]. From it, you can choose the mechanical units you are controlling (first option), select the increment type we want as mentioned earlier (option 2), chose the run mode meaning if you want to repeat the program in cycle or run it only once (option 3), setting up a step increment in your program (one line of code, one code block ...) (option 4), modifying the percentage of speed relative to the programmed speeds (you can only decrease the speed in manual mode, if you want to increase it you need to select manual 100% or automatic mode) (option 5), and finally select the operating mode (option 6).

Fig. 68 – Quick set options

One can observe that with only the quick set menu and the FlexPendant's buttons, you can already completely jog your robot, without opening any window.

Before ending this part, let's speak about crucial data types in ABB architecture and calibration processes. A **tool data type** is used to code a tool in RAPID program [69] and includes :

- A name
- A first boolean that indicated if the robot holds the tool or not at the robot flange
- The coordinates and quaternion orientation of the TCP (tool center point) as a pose data.
- The coordinates of the center of mass, the inertia of the tool and his mass as a load data.

After setting up your tool in RobotStudio (see the tutorial section [5.1]), you can **calibrate the TCP**. You can indeed change the TCP frame coordinates by using a calibration method proposed by ABB that will change the TCP frame coded in the simulated environment RobotStudio to perfectly match the real world.

For this, you need to use **a calibration stake** that you are going to place somewhere in the robot accessible area. You will then approach your tool to the end of the stake that will be your center point [70] (if it's a gripper for example you want the end point of the stake between the two claws, if it is a torch you want it at the tip of the torch). Then in the **jogging menu**, select the **properties of your tool**, select your tool, press **edit** and **define**. A window like [71] will pop up.

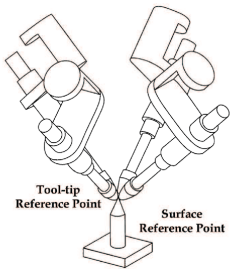


Fig. 70 – TCP calibration

After select **the TCP (default orient.) method**, select the first point and press **modify position** when the end of the stake is where you want your TCP to be. Then orient the tool by using the **reorient movement option** (by pressing twice on linear mode button) and change the tool orientation. Finally select the second point and press **modify position**. Do the same thing for point 3 and 4.

With these four different orientations, the software will determine the TCP position with a precision below the millimeter.

```
tooldata GripperTool:=[TRUE,[[0.069,-0.022,335],[1,0,0,0]],[5.6,[0.069,-0.022,335],[1,0,0,0],0,0,0]];
```

Fig. 69 – Tooldata

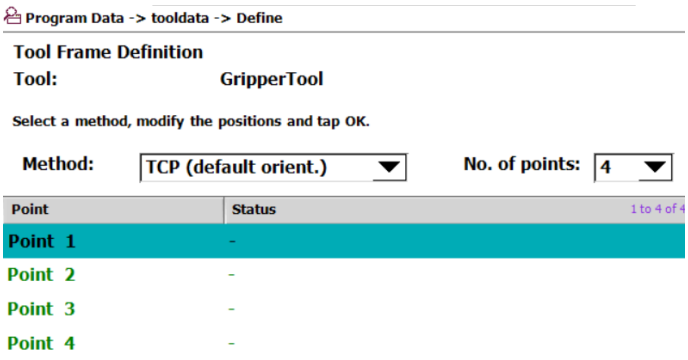


Fig. 71 – Tool calibration window

A wobjdata (work object data) is a frame related to a specific point in your work environment, for example, the corner of a table, the corner of a piece you want to pick and place ... If you want some targets of your program to be relative to these object (we will discuss this in the last part of this guide), you need to create and calibrate wobjdata (the default one being the base frame of your robot).

```
wobjdata BrickFrame := [FALSE, TRUE, "", [[0, 0, 0], [1, 0, 0, 0]], [[100, -700, 40], [1, 0, 0, 0]]];
```

Fig. 72 – Wobjdata

A wobjdata [72] is coded by :

- Two boolean : if the robot holds the work object or not and if the user frame is fixed or not.
- A user frame, which is the frame used to locate the object frame coded by a pose data
- An object frame, which is the work object frame (corner of a table for example) coded by a pose data

You can **calibrate your work object data** by selecting **the work object properties** in the jogging menu, then selecting the desired work object, pressing **edit**, **define** and selecting the 3 points method only for the object (for the user plane I have always kept the default which is the base frame) in the window that will pop up [73].

The first point (X_1) is the origin point of the frame, the second point is another point on the X axis (these two point define the X axis), the last point belongs to the Y axis that will be perpendicular to X axis cut the Y axis at point X_1. You can use stakes and a specific calibration tool (like a stake mounted on the flange) to bring the TCP to the good position and press **modify position** for each point.

Of course, you need to calibrate your TCP before calibrating works object.

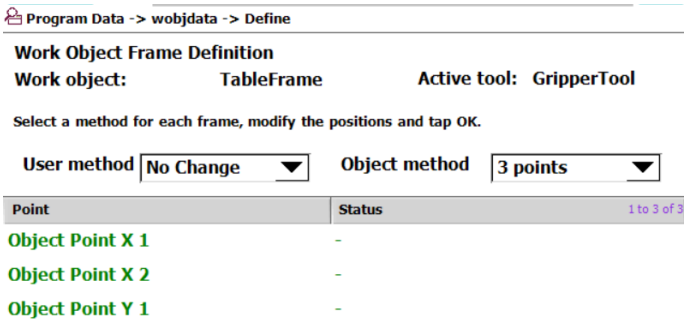


Fig. 73 – Work object calibration

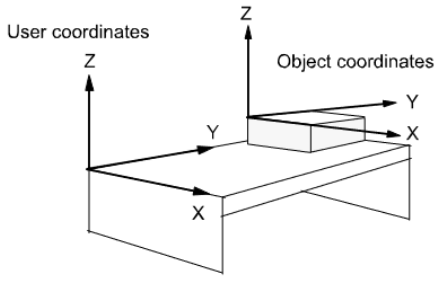


Fig. 74 – Work object explanation

Finally, let's talk about **ABB Robot axis configuration**. When you define a target, for the majority of them, your robot will have **more than one joint orientation sequence to reach it [75]**. These joint orientation sequences are called **axis configuration**. The default configuration of a robot is [0000] coded in rotation quadrants of axis 1, 4 and 6 plus an extra virtual axis used for specifying the wrist center in relation to other axes.

ABB manual definition of quadrants : *The quadrants are numbered from zero for positive (counterclockwise) rotation and from -1 for negative (clockwise) rotation.*
For a linear axis, the integer specifies the range (in meters) from the neutral position in which the axis is located.

Sometimes, some tricky targets are not reachable with the default robot configuration (or the configuration you registered for this specific target) and this will result in a singularity and the stopping of your program if you don't disable **the configuration control algorithm of the ABB controller**. Thus, you can turn off configuration control to enable the controller to calculate and change the robot axis configuration depending on the target (this may generate some unwanted result if you are working in a closed environment but will enhance the target reachability of your robot).

As explained before, MoveL (linear movement) movements are often generating singularities compared to MoveJ (joint movement) movements. Thus, ABB, gives you the opportunity to disable the configuration control algorithm for linear movements or joints movement with the following instructions : **ConfL Off and ConfJ Off** to separate both movements types. That's why my main procedure in **[50]** is beginning with **ConfL Off**.

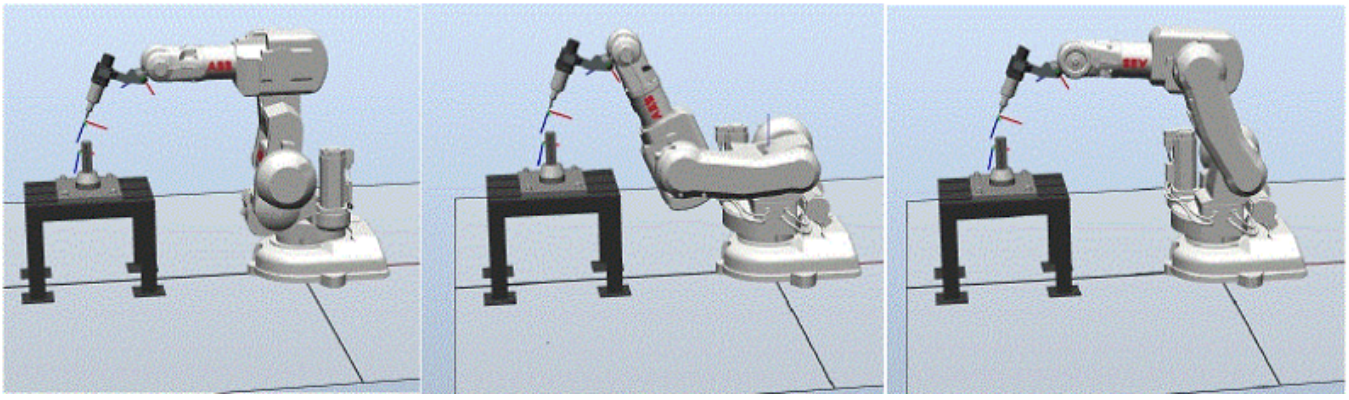


Fig. 75 – ABB RobotStudio Manual example for different configuration possibilities

5 Pick and Place tutorial

5.1 GrassHopper start

You can **download the starter GrassHopper simulation** from my GitHub Page, in the complementary resources section [5.9] or follow this tutorial with your own. There are **two simulations** on my GitHub, the first one is a simulation made with the *Robot plugin* where I have not found a way to create a work object plane (thus every target is relative to the base of the robot and not to other specific frames) and the other one is made with the *Robot component plugin* where I have used some component to create work objects.

Thus, the **second simulation is more advanced** and will work if we change the configuration of the robotic environment. For example, is we move the table or the blocks we want to pick and place, we will only need to recalibrate our Robot and the program will work because targets position are relative to the work object plane and not the world plane. **With the first simulation, you will need to place the blocks and table exactly at the same position (you are calibrating the robotic environment, but not the robot in itself).**

Both simulations are the same : the IndexLab ABB robot (ABB IRB 45-205) picks and places 8 blocks on a table with MoveL and MoveJ movements using a pneumatic gripper tool from IndexLab. The robot is programmed to change its speed before gripping and releasing blocks, signals are changed to activate and deactivate air valves to open and close the gripper and some pauses are introduced to give the robot some times to grip and release blocks (the operation is not instantaneous) **[77]**. They are also using the same **simplified version of the gripper tool** that we will integrate in RobotStudio **[76]**.

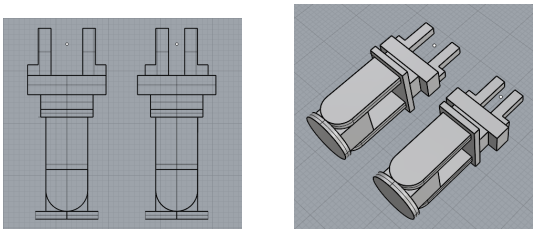
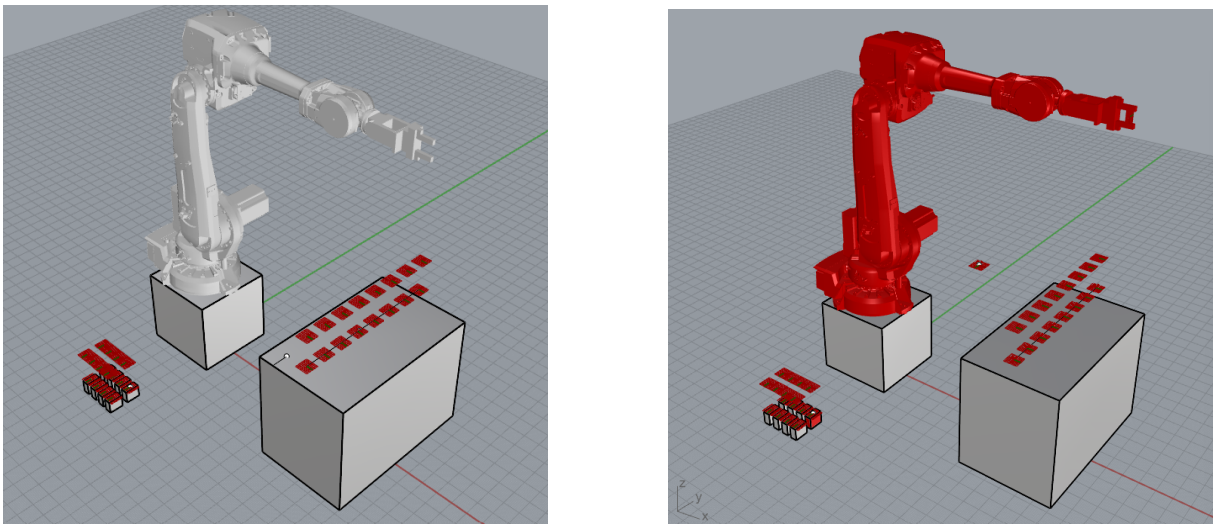


Fig. 76 – Simplified gripper model

Grasshopper simulation algorithms are subdivided in different algorithm for :

- Tool center point parametrization and tool integration
- Home target definition
- Bricks and table planes creation
- Targets creation and scheduling
- Program generation and simulation
- Collision checking (only for the first simulation)
- Double tools (gripper closed and unclosed) only for the first simulation
- Work object data creation (only for second simulation)



Robot plugin simulation

Robot Component Plugin

Fig. 77 – GrassHopper robot component plugin simulation

The output of these GrassHopper simulation is a raw RAPID programm [78].

Code

```
                                                                {0;0;0;0;0;0;0;0;0;0;0;0}
 0 MODULE Home_T_ROB1
 1 VAR extjoint extj := [9E9,9E9,9E9,9E9,9E9,9E9];
 2 VAR confdata conf := [0,0,0,0];
 3 PERS tooldata GripperUnclose:=[TRUE,[[0.069,-0.022,335],[1,0,0,0]],[5.6,[0.069,-0.022,335],[1,0,0,0],0,0,0]];
 4 PERS tooldata GripperClose:=[TRUE,[[0.069,-0.022,335],[1,0,0,0]],[5.6,[0.069,-0.022,335],[1,0,0,0],0,0,0]];
 5 TASK PERS wobjdata DefaultFrame:=[FALSE,TRUE,"",[[0,0,0],[1,0,0,0]],[[0,0,0],[1,0,0,0]]];
 6 TASK PERS speeddata Speed000:=[100,180,5000,1080];
 7 TASK PERS speeddata Speed001:=[200,180,5000,1080];
 8 TASK PERS speeddata Speed002:=[10,180,5000,1080];
 9 PROC Main()
10 ConfL \Off;
                                                                {0;0;0;0;0;0;0;0;0;0;0;1}
 0 MoveAbsJ [[0,-0,-0,0,-0,0],extj],Speed000,fine,GripperUnclose;
 1 SetDO out1, 0;
   SetDO out2, 0;
 2 MoveJ [[172.5,-655,262.5],[0,1,0,0],[-2,0,1,1],extj],Speed001,fine,GripperUnclose \WObj:=DefaultFrame;
 3 MoveL [[172.5,-655,62.5],[0,1,0,0],conf,extj],Speed002,fine,GripperUnclose \WObj:=DefaultFrame;
   SetDO out1, 1;
 4 WaitTime 1;
   SetDO out1, 0;
 5 MoveL [[172.5,-655,262.5],[0,1,0,0],conf,extj],Speed002,fine,GripperClose \WObj:=DefaultFrame;
 6 MoveAbsJ [[0,-0,-0,0,-0,0],extj],Speed000,fine,GripperClose;
 7 MoveJ [[1303,-415,1010],[0,0.70711,0.70711,0],[-2,0,0,0],extj],Speed000,fine,GripperClose \WObj:=DefaultFrame;
 8 MoveL [[1303,-415,810],[0,0.70711,0.70711,0],conf,extj],Speed002,fine,GripperClose \WObj:=DefaultFrame;
   SetDO out2, 1;
 9 WaitTime 1;
   SetDO out2, 0;
10 MoveAbsJ [[0,-0,-0,0,-0,0],extj],Speed000,fine,GripperUnclose;
11 MoveAbsJ [[0,-0,-0,0,-0,0],extj],Speed000,fine,GripperUnclose;
12 SetDO out1, 0;
   SetDO out2, 0;
13 MoveJ [[172.5,-780,262.5],[0,1,0,0],[-2,0,1,1],extj],Speed001,fine,GripperUnclose \WObj:=DefaultFrame;
14 MoveL [[172.5,-780,62.5],[0,1,0,0],conf,extj],Speed002,fine,GripperUnclose \WObj:=DefaultFrame;
   SetDO out1, 1;
15 WaitTime 1;
   SetDO out1, 0;
```

Fig. 78 – Raw output code

I advise you to read the RobotStudio environment section [3.2], especially the modeling panel explanation, before starting this tutorial as we will model our robotic environment with RobotStudio features.

5.2 Robotic cell creation

In a first place, let's model the foundation, bricks and the table of the GrassHopper simulation in RobotStudio with the modeling window.

Follow the RobotStudio setup section **[2.1]** to download and install the suitable RobotWare version, and create a pick and place project with the correct Robot (in our case it is an ABB IRB 4600-45-205). You should have this graphic window : **[79]**.

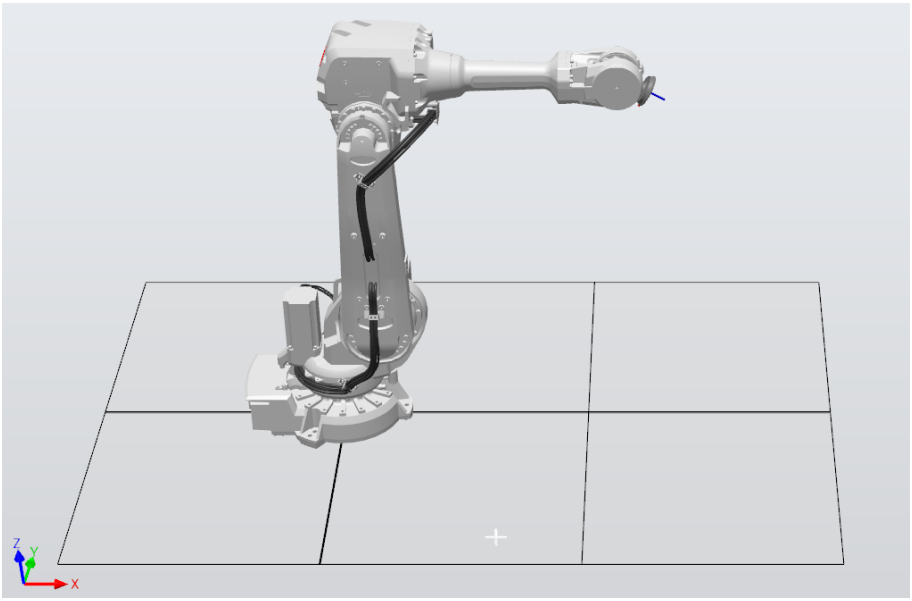
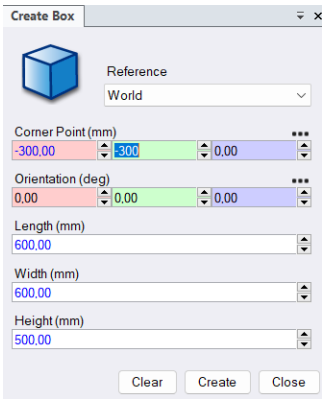


Fig. 79 – Initial graphic window

Let's create a cubic block to model the foundation below the IndexLab ABB robot. For that, we use the modeling window **Create panel**, **Solid button** and select the **box feature**. Height dimension should be 500 mm and the block should be centered in the Robotic cell so (0X, 0Y) coordinate in the world reference coordinate system **[80]**.

Then, we want to move our Robot to put it on top of this foundation. For this, select your Robot in the layout side bare, **right-click, place and select set one point**. The first point should be (0, 0, 0) and the second one (0, 0, 500) **[81]**. A first pop-up should appear to ask for task frame update, select yes. A second one will ask you to restart the controller, select yes **[82]**.



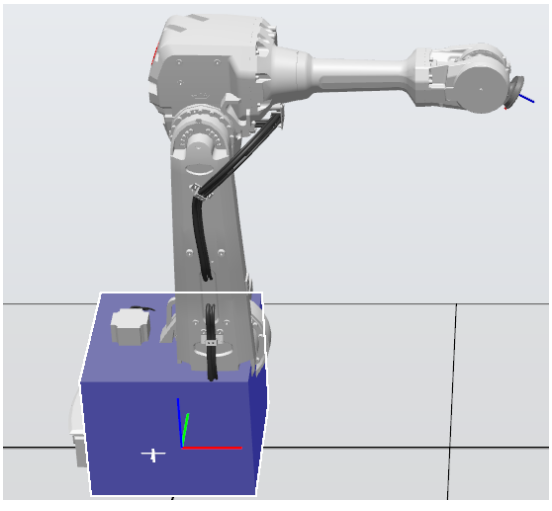


Fig. 80 – Foundation creation

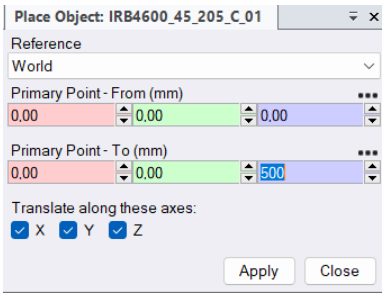


Fig. 81 – Robot placement

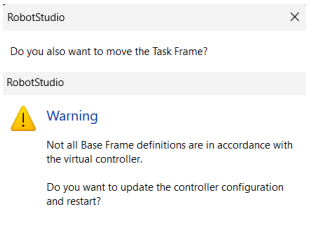


Fig. 82 – Warning pop-up

Now let's create the table. **If you are using the robot plugin simulation, the table needs to be exactly at the same distance from the robot than in GrassHopper**. For the other plugin, we just need it to be reachable by the robotic arm, because we will create a reference frame for the table. As for the foundation, we use the **create box option**, and we move it with the **set one point option**. Table center XY coordinates should be (0, 1300), and its dimension are (600, 1000, 720) mm **[83]**.

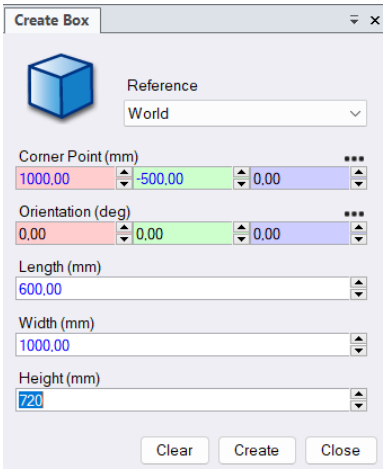


Fig. 83 – Table creation

Finally, we need to create the 8 blocks. As for the table, the first block should be at the same position as the GrassHopper simulation if you are using the robot plugin one. For the robot component plugin, this has no importance. Blocks dimensions are (45, 90, 80) mm. There is a 90 mm X axis offset between their center and a 125 mm Y axis offset between the two rows. The first block XY coordinates are (150, -700) mm. We use the same features as for the table : **[84]**.

Right-click on the brick and **rename it**, use the **modify, change local coordinate option** and set the local coordinate to the center of the upper face of the block **[85]** (use the snap object feature). Then we can use the **duplicate** feature and duplicate the first block three times with a -90 mm X offset from the center of its upper face. Duplicate once with a -125 mm Y axis offset, and duplicate the first block of the second row with -90 mm offset **[86]**.

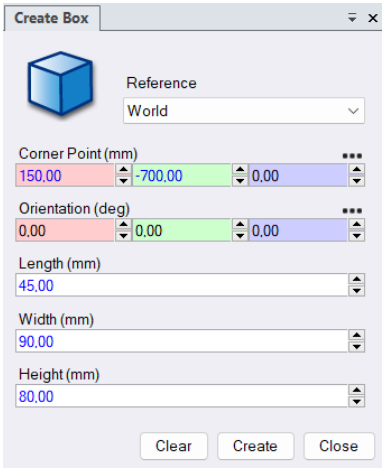


Fig. 84 – First block creation

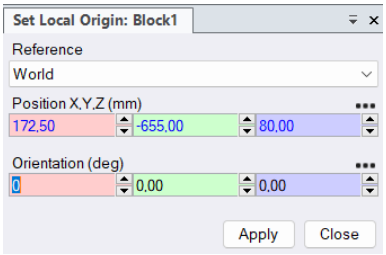


Fig. 85 – First block local coordinate

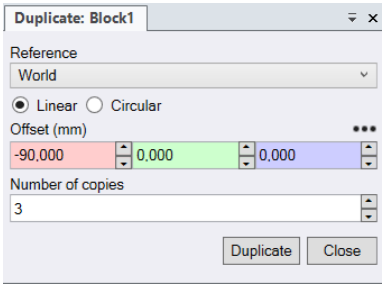


Fig. 86 – Duplication of block 1

Fig. 87 – Duplication pre-visualization

You should now have a graphic window like these : [88]. You can then rename everything as you want, set the appearance of your blocks to create a nice visual.

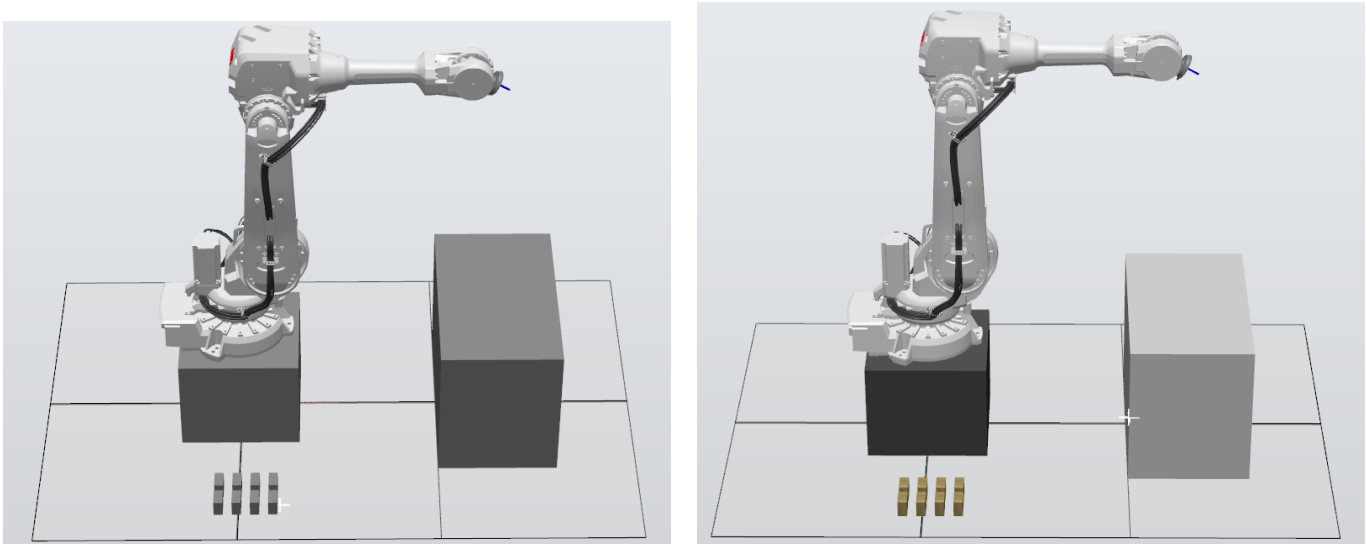


Fig. 88 – Robotic cell elements

5.3 Tool integration

There are two ways to integrate tools in RobotStudio. The first one is to create the tool with the **create tool option** and the second one is to create a **tool mechanism** [89]. The one for the rest of the tutorial will be the tool created with the create mechanism option.

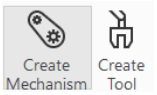


Fig. 89

5.3.1 Create tool

Let's create a conic tool (fictive and really simple tool) to understand how to create a tool in RobotStudio.

The first step is to **create the geometry of the tool** in the modeling panel and to put it in the right position (attached to the flange of the robot). In the modeling panel, right-click on your robot in the layout sidebar and select the **jog-joint option** [90]. We want our robot to be in a neutral position, so **jog the fifth axis to 0°** [91]. Use the create solid option and select the **cone shape**. Then, with the snap object option, select the origin of the flange as the center of our cone base. **Create a cone with a diameter of 125 mm and a height of 200 mm and a 90° orientation on the Y axis** (world frame reference) [93]. Finally, **attach the cone geometry to the six axis without updating the position** (otherwise your tool won't follow the movement of the robot). Rename your component to ConeTool. You should obtain the following model : [92].

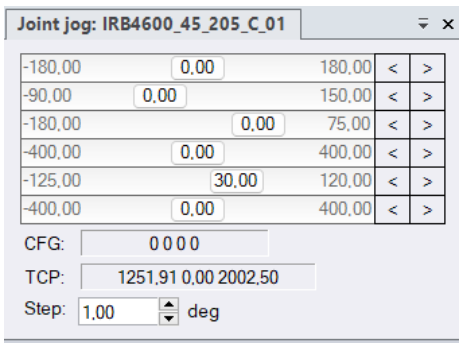


Fig. 90 – Jog joint feature

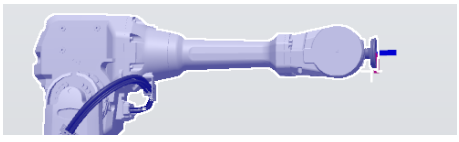


Fig. 91 – Neutral postion

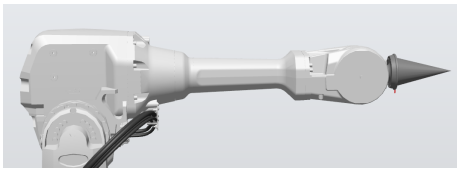


Fig. 92 – Cone created

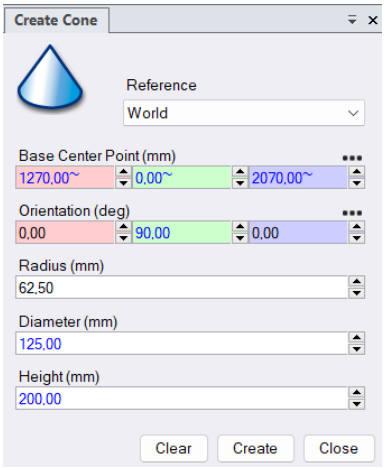


Fig. 93 – Cone creation

Now, let's use **the create tool option**. Name it ConeTool, select the **existing geometry** option and chose your ConeTool. For a better precision when you will transfer your RobotStudio simulation to the real Robot, you can put the right mass, center of gravity and inertia of your tool [94]. Next, we need to **declare the TCP** of this tool. This TCP declaration use the former tool data reference frame coordinate system (the flange tool data). Use the snap object option and select the end of the cone for the TCP position. The orientation of the TCP is the same as the flange frame, so no rotation is needed [95]. When the creation if finished, your **mechanism should appear in the layout** and the new TCP frame should appear in the graphic window [96].

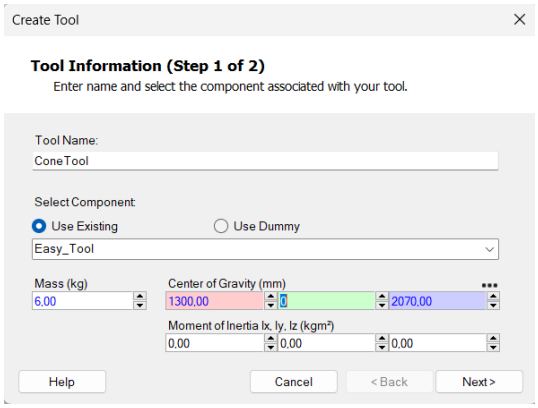


Fig. 94 – ConeTool creation

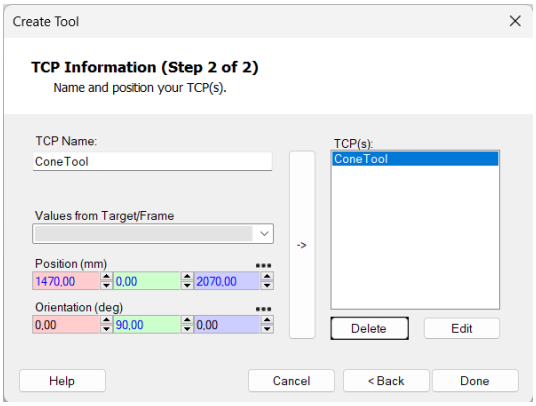


Fig. 95 – ConeTool creation step 2

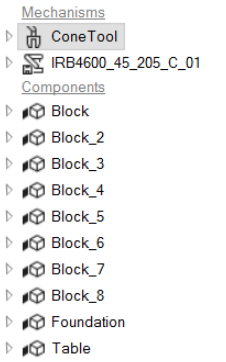


Fig. 96 – ConeTool created

Finally, we want to **test our new tool in the simulation panel**. For this, go to the **path&target sidebar** and **create a new tool data** with ConeTool name. Use snap object to modify the TCP position and don't change the orientation [99]. In the **sync properties**, select **the PERS storage type and the user module**. Indeed, we want to declare our tool in the user module, which is a system module (variable declared inside this module can be used without other declaration in any other module).

Now, go to the **RAPID window**, double-click the user module and **declare your tool with the following line** : [97] (I have explained the tool declaration in the manual mode section [4.1]).

```
PERS tooldata ConeTool := [TRUE, [[0, 0, 200], [1, 0, 0, 0]], [6, [0, 0, 50], [1, 0, 0, 0], 0, 0, 0]];
```

Fig. 97 – Cone tool declaration

Finally, **write down the following test program in the Module1**. We are using the default wobj0 work frame data, which is the world frame [100]. The default data are declared in the base module, which is also a system module [98].

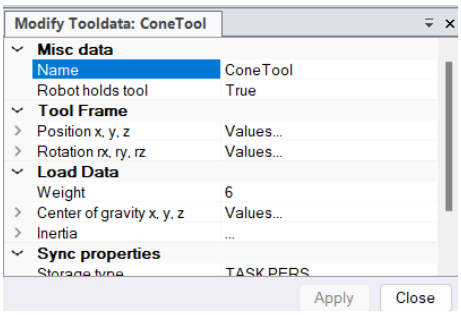


Fig. 99 – Tool data declaration

```
1  MODULE BASE (SYSMODULE, NOSTEPIN, VIEWONLY)
2
3   ! System module with basic predefined system data
4   !*********************************************
5
6   ! System data tool0, wobj0 and load0
7   ! Do not translate or delete tool0, wobj0, load0
8   PERS tooldata tool0 := [TRUE, [[0, 0, 0], [1, 0, 0, 0]],
9                           [0.001, [0, 0, 0.001],[1, 0, 0, 0], 0, 0, 0]];
10
11  PERS wobjdata wobj0 := [FALSE, TRUE, "", [[0, 0, 0],[1, 0, 0, 0]],
12                          [[0, 0, 0],[1, 0, 0, 0]]];
13
14  PERS loaddata load0 := [0.001, [0, 0, 0.001],[1, 0, 0, 0], 0, 0, 0];
15
16  ENDMODULE
17
```

Fig. 98 – Base system module

```
MODULE Module1
    PROC main()
        MoveAbsJ [[0,-0,-0,0,-0,0],[9E9, 9E9, 9E9, 9E9, 9E9, 9E9]],v200,fine,ConeTool \WObj:=wobj0;
        MoveJ [[172.5,-655,262.5],[0,1,0,0],[-2,0,1,1], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]],v200,fine,ConeTool \WObj:=wobj0;
    ENDPROC
ENDMODULE
```

Fig. 100 – Simple module 1 program

Verify your program in the RAPID window and **apply the modification** (apply button). A green bar should appear on the left side of your RAPID program. **Go to the simulation panel** and activate the **TCP trace feature** (change the default trace color) **[101]**. You can finally launch the simulation with the **start button** ! You should obtain the following result : **[102]**.

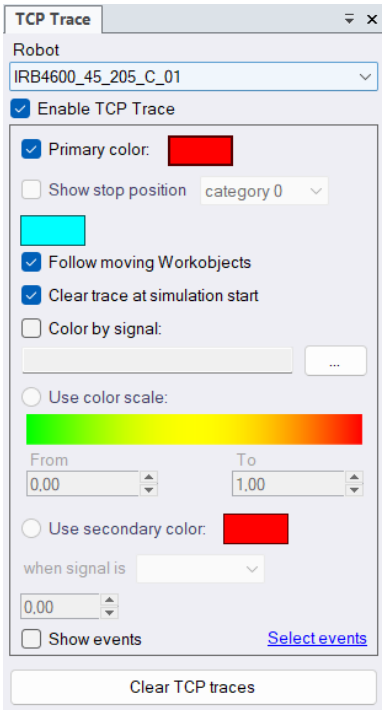


Fig. 101 – TCP trace activation

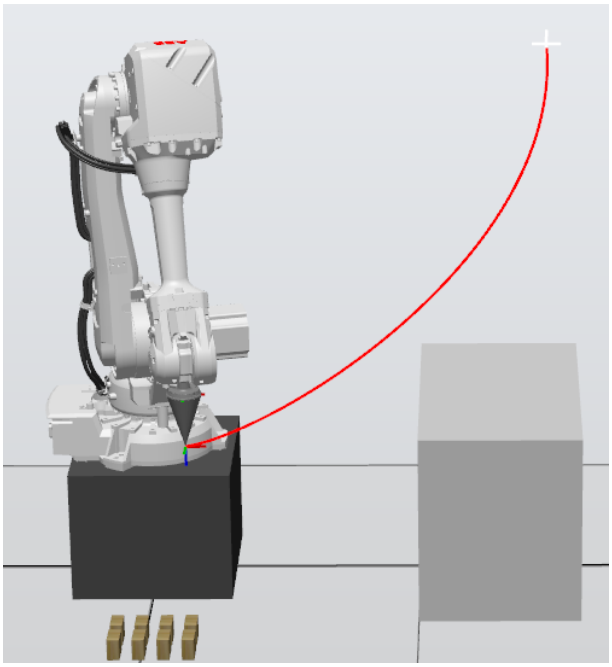


Fig. 102 – TCP trace activation

5.3.2 Create mechanism tool

In this subpart, we are going to **integrate a gripper tool designed from Rhino**. The gripper is made of 3 pieces: the tool mount and the two claws of the gripper. We only use **two .sat files** as the claws are exactly the same. Of course, the .sat files are available on my GitHub page.

From the modeling panel, use the **import geometry feature to import the .sat geometry files (if you did the ConeTool tutorial, you can make it disappear as well as the ConeTool TCP frame, we won't use them anymore)**. The pieces will be imported at the origin of the graphic window, inside the foundation **[103]**. Change their position with the **set position option** to be able to see them, and to select them with the snap object option.

Next, we want to place the **tool mount on the flange with the set one point** feature (chose the center of the tool mount flange lower surface as the primary point -from and the origin of the flange frame as the primary point -to) **[104]**. You will then need to **rotate the tool mount** with a 90° rotation about the Y world reference frame axis (don't forget to choose the correct rotation origin with the snap object option) **[105]**.

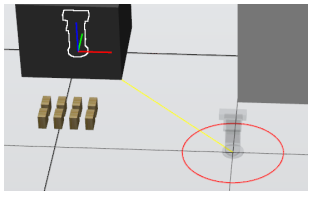
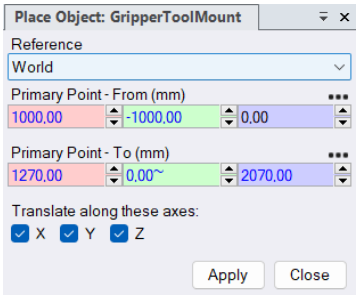


Fig. 103 – Tool mount imported



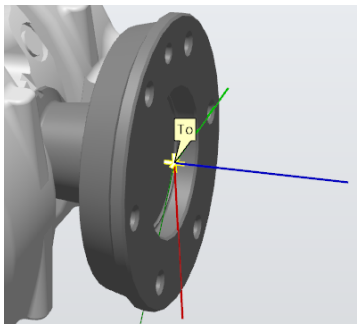


Fig. 104 – Tool mount place process

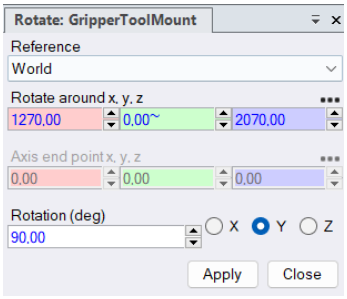


Fig. 105 – Tool mount rotation

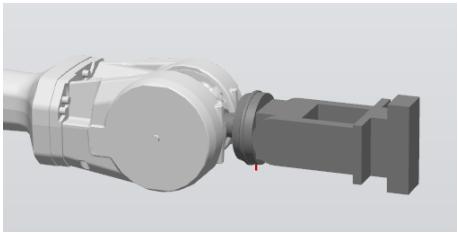


Fig. 106 – Tool mount placed

For the **first gripper claw**, we will use the same features and select the following points : **[107]**. A **90° rotation is also needed** about the Y axis (choose the correct rotation origin). You will have this result : **[109]**.

Finally, we need to **duplicate the first gripper claw with a Z offset of 150 mm [110]** and to **rotate** it with a 180° rotation about the X world reference frame axis (again, watch out for the correct rotation origin). **Attach the tool mount component to the flange** (don't update the position) and **attach the two claws to the tool mount component** (again, don't update). You will now have the final geometry of the gripper tool **[112]**! Feel free to modify the appearance of the gripper as you wish.

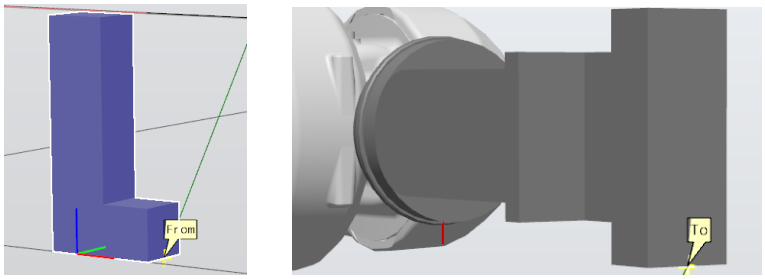


Fig. 107 – First claw place process

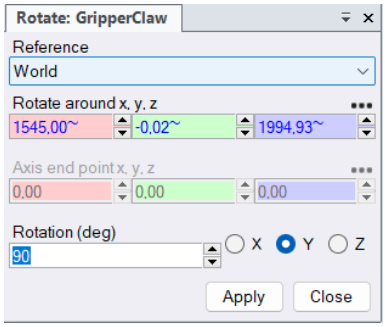


Fig. 108 – Claw rotation

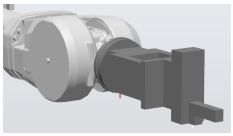


Fig. 109 – First claw placed

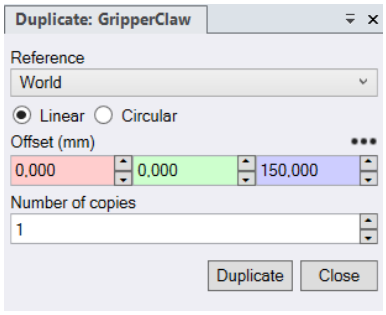


Fig. 110 – Claw duplication

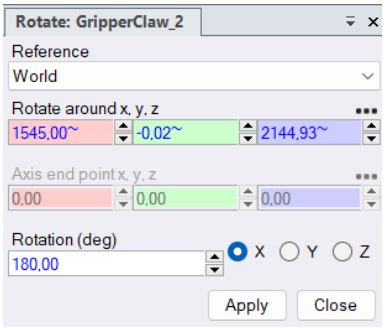


Fig. 111 – Second claw rotation

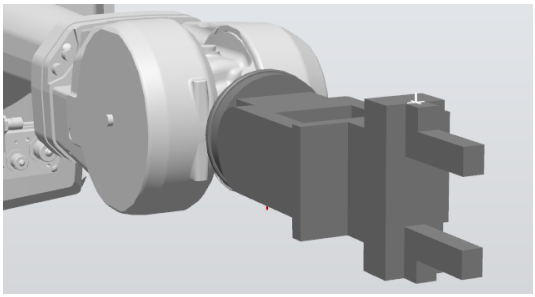


Fig. 112 – Final gripper tool geometry

We can now create the **gripper tool mechanism the with the create mechanism feature** of the modeling panel. Name it GripperTool and select the tool mechanism type **[113]**.You will then need to follow the following steps (double-click on them to enter the parameters):

- **Links :** The gripper tool has **3 links** : one base link made of the tool mount (L1) and two links between the claws and the tool mount (L2 and L3) **[114]**.
- **Joints :** The gripper tool has **two prismatic joints** between the claws and the tool mount responsible for the gripping action. You need to parametrize them as such : **[115]**, meaning that the first claw can move from 15mm about the Z axis (world reference frame) and that the joint is a prismatic joint between the tool mount and the first claw. The second joint is a symmetry of the first one (-15mm about the Z axis) **[116]**. Don't forget to jog the joints to verify if your parametrization is correct.

- **Tooldata :** Here, you need to **specify the TCP of the tool** as well as the mass and inertia (for a better precision on the real system). Tooldata should belong to the base link, the TCP must be placed slightly below the middle of the segment crossing the middle of both gripper claws **[117]** (use the snap object and then modify the coordinate as you need). The mass of the real tool is 5.6 kg, the center of mass position and the inertia of the tool are unknown **[118]**.

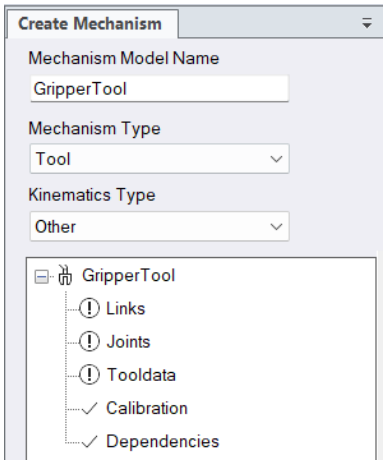


Fig. 113 – Mechanism creation

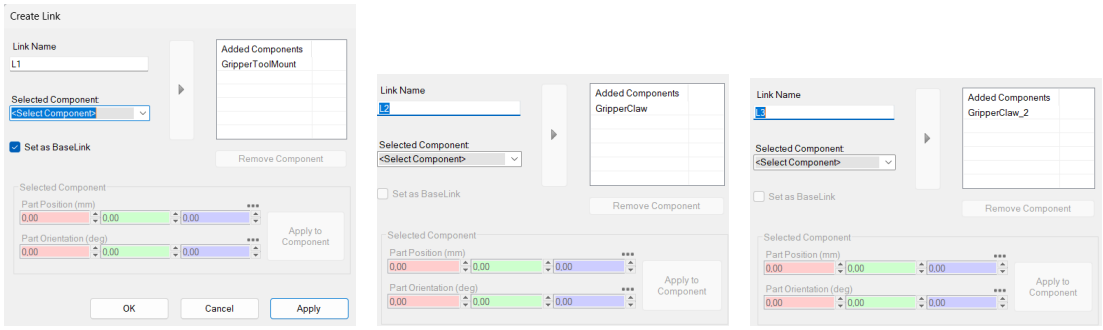


Fig. 114 – Links creation

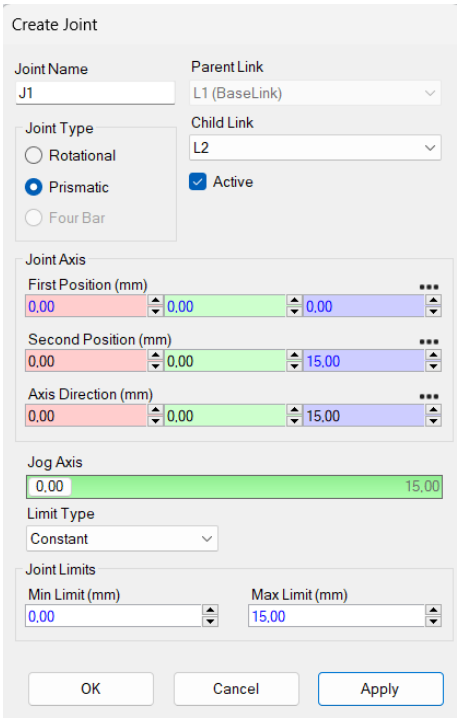


Fig. 115 – J1

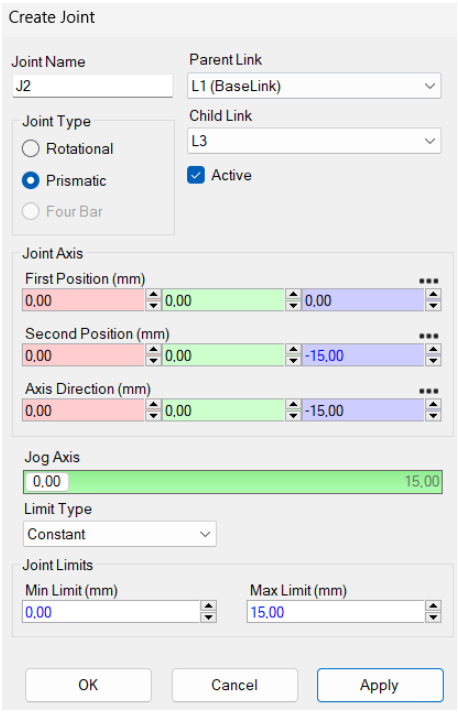


Fig. 116 – J2

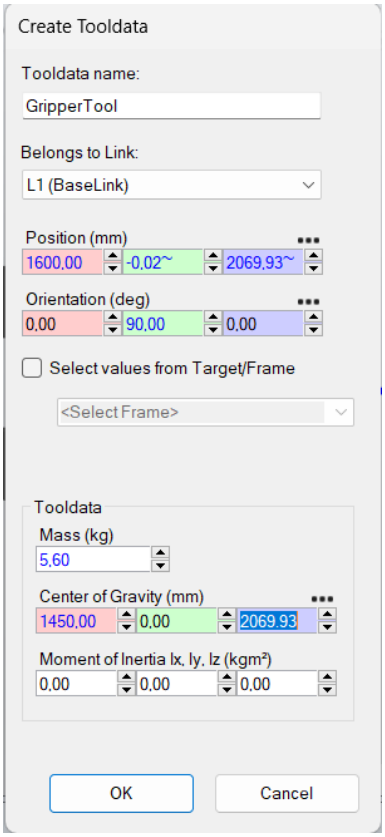


Fig. 118 – Tooldata declaration

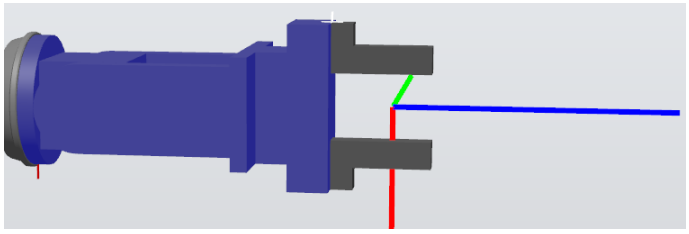


Fig. 117 – TCP placement

Now we can **compile the mechanism** every mechanism step is ticked **[119]**, your gripper tool should then appear in the **layout**. We can then **add a gripper position** relative to the position of the two prismatic joint. Two position are needed : an **unclosed gripper position [120]** and a **closed gripper position [121]** (if the display of the gripper is buggy, it's normal).

We can **set the transition time between these two position [122]**. You can now test the joints of your gripper tool by right-clicking on it in the layout and select the jog joint feature **[123]**.

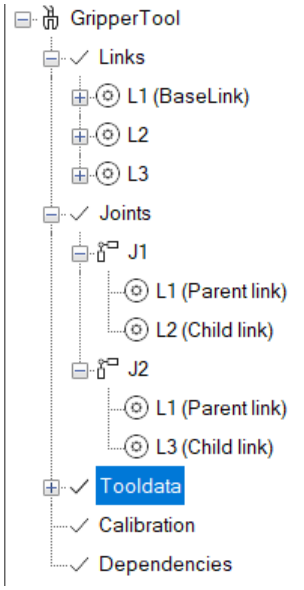


Fig. 119 – Mechanism compiled

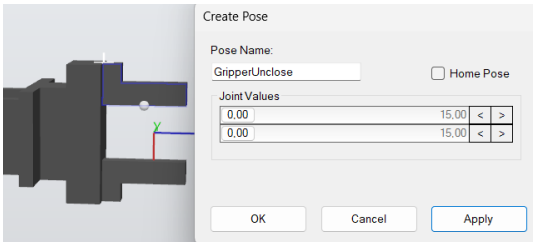


Fig. 120 – Unclosed Gripper

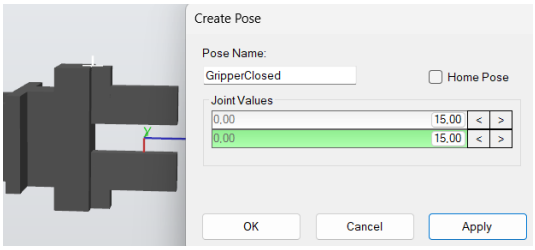


Fig. 121 – Closed Gripper

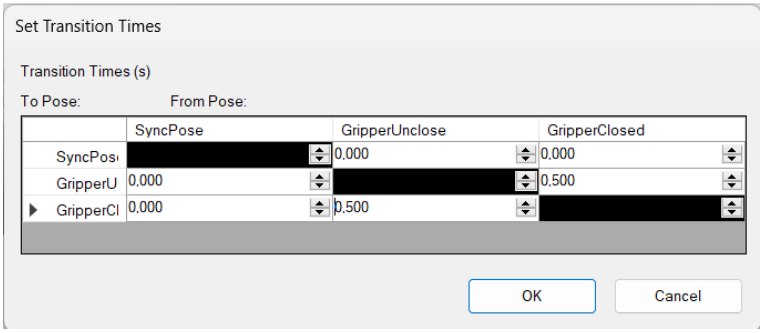


Fig. 122 – Set transition time

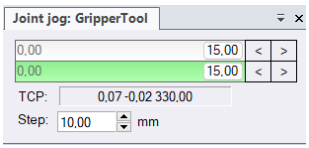


Fig. 123 – Jog gripper joints

Finally, we can **create the tool data in the simulation window, path&target sidebar [??]**. As for the ConeTool, use the snap object and **snap frame parameter** to choose the correct TCP position (relative to the flange reference frame, so no rotation is needed). Update the name, the mass, the center of gravity and select the user module for data synchronization. **You also need to declare the Gripper tool in the user module as for the Cone tool but set the center of gravity to (0, 0, 300)**.

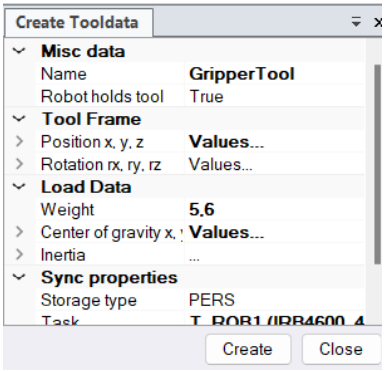


Fig. 124 – Tool data creation

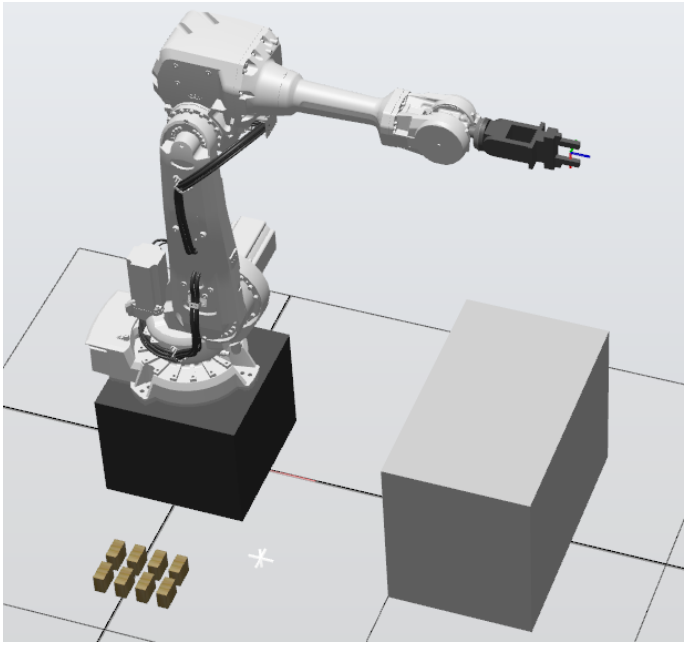


Fig. 125 – Gripper tool integrated

5.4 Work object planes

This part is not mandatory if you work with the Robot plugin simulation, as I haven't found a way to create a work object reference frame with this plugin.

For the other simulation, I have used the work object reference frame : one is used for the **targets relative to the bricks** we want to pick and place, and the other one is used for the **targets relative to the table**. Thus, we are going to **create two work object plane data inside RobotStudio**. I have explained what is a work object reference plane at the end of the manual mode section.

First, we need to **create two frames** that will help use creating the two work object data. Use the **create frame button** of the modeling window and use the snap object button to place the origin in the **center of the first brick upper surface** [126]. Then create another frame by selecting the **bottom left corner of the table** as the origin of the table frame and offsetting by 100 mm the X and Y coordinates [127]. **Rename both frames**, to make your simulation understandable.

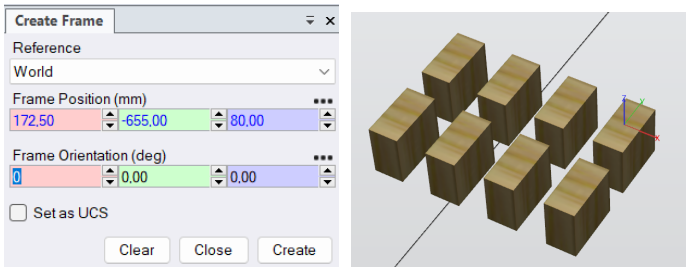


Fig. 126 – Brick object frame creation

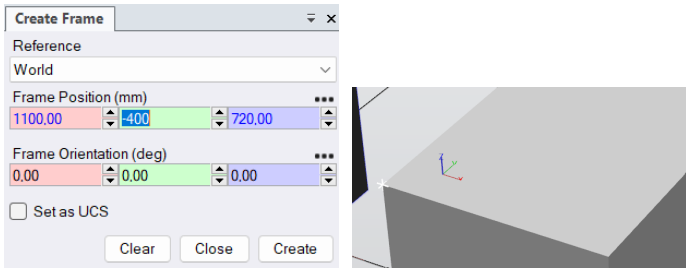


Fig. 127 – Brick object frame creation

Next, go to the simulation window and **create new work object data in the path&target sidebar** [128]. Work object data is using a **user reference frame** and an **object reference frame**. We only need to modify the object reference frame origin coordinate by selecting the origin of the frames we previously created with the snap frame feature [129] [130]. **Modify the sync properties** to PERS data storage and user module name [131]. You need to do the same operation for the brick work object data and the table work object data. After this declaration, work object frame should appear bigger than normal frames [132].

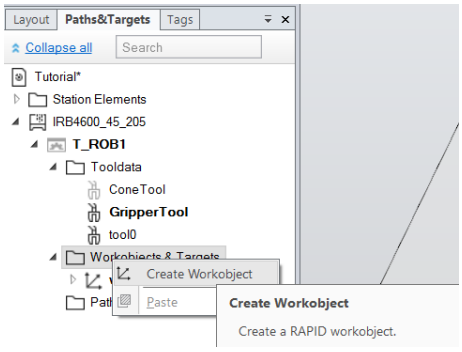


Fig. 128 – Work object in Path&Target

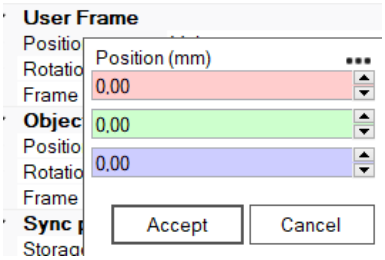


Fig. 129 – User frame not edited

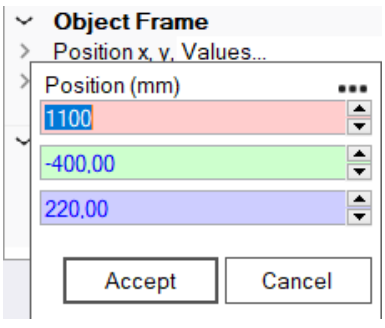


Fig. 130 – Table object frame declaration

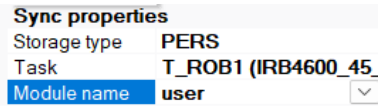


Fig. 131 – Sync properties

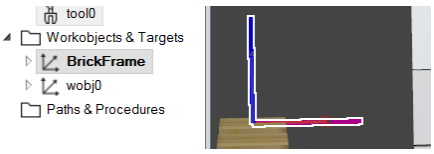


Fig. 132 – Brick work object declared

Finally, we need to declare these data in the **user module**. For that, go to the RAPID window, double-click the user module in the sidebar and write the following lines [133]. It is just the **coding translation of the previous steps**. I have explained this declaration in the manual mode section.

```
PERS wobjdata BrickFrame := [FALSE, TRUE, "", [[0, 0, 0], [1, 0, 0, 0]], [[172.5, -655, 80], [1, 0, 0, 0]]];
PERS wobjdata TableFrame := [FALSE, TRUE, "", [[0, 0, 0], [1, 0, 0, 0]], [[1100, -400, 720], [1, 0, 0, 0]]];
```

Fig. 133 – Work objects data RAPID declaration

5.5 Rapid code formal edition

Now that we have every component, mechanism and reference frame we need, we can start the **editing of our GrassHopper raw code**. The aim of this subsection is to transform our raw code into a **structured RAPID program, easily understandable and readable** for someone that hasn't created the GrassHopper simulation.
For this, we will create a ***Pick_Place* procedure and a main procedure into a *Pick_Place*** module.

First go to the **RAPID window** and **create a new module** for the T_ROB1 task (meaning task of Rob1 robot, Rob1 being the name of the main robot of the ICR5 controller you are using) [??]. Then **copy and past the raw code** of your GrassHopper simulation in the *Pick_Place_Module* RAPID.

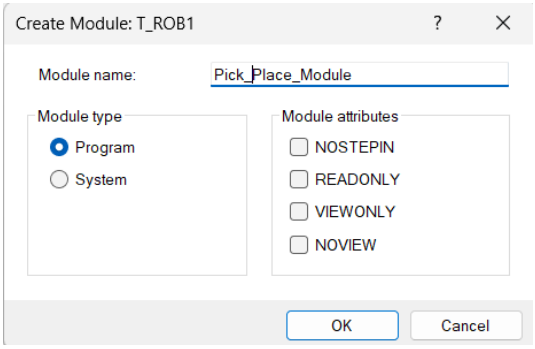


Fig. 134 – Create new module

If you have tested the GrassHopper simulation, you know that the robot **repeats 8 times the same sequence of actions** that uses two targets for the bricks (one above the upper surface of the brick and one on the surface), two targets for the table (one slightly above the table and one above the table at the block height) and the home position. This sequence will be our *Pick_Place* procedure. GrassHopper gives you instructions for each sequence. **We will compress this raw code in only two procedure (main and Pick_Place procedures) !**

For both simulation, you should do the following step :

- **Delete the two lines of the Home_T_ROB1 module and the tool declaration.**
- **Edit the extj and conf declaration, data type should be CONST.**
- Rename Speed000 to "Fast_speed" and Speed001 to "Approach_Speed"

- **Declare jointtargets and robtargets for each movement instruction**, give suitable names to your targets, and use commentary to make the declaration easily understandable. A jointtarget declaration needs a robjoint data that gather the orientation of each joint in a list and an extjoint data for the orientation of each external joint you are using (if no external joint is needed, the extjoint data value is [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]) **[135]**. A robtarget declaration needs a pos data (X, Y, Z) an orientation data (q1, q2, q3, q4) a configuration data (robotic configuration is explained in the manual mode section **[4.1]**) and an extjoint data **[136]**.

```
CONST jointtarget Home := [[0,-0,-0,0,-0,0],extj];
```

Fig. 135 – Jointtarget

```
CONST robtarget Brick1_Upper := [[0, 0, 200], [0, 1, 0, 0], [0, 0, 0, 0], extj];
```

Fig. 136 – Robtarget

In our case, we need to **declare targets for the home position, the upper targets and lower targets plane of each brick and table position.** Remind that movement instruction such as MoveL and MoveJ needs robtargets in their declaration. Thus, you need to **copy and past the beginning of MoveL/MoveJ declaration [137]**.

```
MoveJ [[-226, 0, 200], [0, 1, 0, 0], [0, 0, 0, 0], extj], v200, fine, GripperUnclose\WObj:=BrickFrame;
```

Fig. 137 – Target inside move instruction

If you are using the *Robot plugin* simulation, copy and past every instruction. If you are using the ***Robot component plugin*** simulation, we know that the bricks targets position are relative to the brick work object data frame and that bricks are offsetted by 90 mm about the X axis and 125 mm about the Y axis. For table targets, copy and past the beginning of the movement declaration.

You should have the following targets declaration at the end, for the ***Robot plugin*** : **[143]**, for the ***Robot component plugin*** : **[144]**.

- Next, we need to create the Pick_Place procedure. A procedure is close to a function in other languages. Indeed, a **procedure takes argument** and can be called in other part of our RAPID program (in the main procedure, for example).

  A pick place sequence is composed of the following actions :

  - Fast MoveJ to the brick upper plane
  - Slow MoveL to the brick lower plane and grip
  - Slow MoveL to the brick upper plane
  - Fast MoveAbsJ to home position
  - Fast MoveJ to the table upper plane
  - Slow MoveL to the table lower plane and release of the brick
  - Slow MoveL to the table upper plane

– Fast MoveAbsJ to home

Thus, our procedure will take only **four arguments** : the two bricks plane and the two table plane **[138]**. The core of the procedure is the direct translation of the above list. **To trigger the gripping and release actions of the gripper, we are updating boolean signals** linked to valves inside the pneumatic gripper circuit. **OUT1 signal will close the gripper when equal to 1, OUT2 signal will open it when equal to 1**. Signal must not be equal to one at the same time. That's why we are turning them to 0 just after being to 1.

```
PROC Pick_Place(robtarget Brick_Upper, robtarget   Brick_Lower, robtarget Table_Upper, robtarget Table_Lower)
```

Fig. 138 – Pick_Place procedure declaration

You should have this Pick_Place procedure **if you are using the *Robot plugin*** : **[139]** note the default frame utilization in every instruction, meaning that every movement and target is relative to the default frame (world reference frame).

If you are using the *Robot component* simulation, **we need to put the right reference frame in our instructions**. Movements to upper and lower bricks targets need to be relative to the brick reference frame. Movements to upper and lower table targets need to be relative to the table reference. Movement to home position need to be relative to the world reference frame. You will then have this procedure **[140]**.

```
PROC Pick_Place(robtarget Brick_Upper, robtarget   Brick_Lower, robtarget Table_Upper, robtarget Table1_Lower)
    SetDO out1, 0;
    SetDO out2, 0; ! signal initialisation
    MoveJ Brick_Upper, Fast_Speed, fine, GripperTool \WObj:= DefaultFrame;
    MoveL Brick_Lower, Approach_Speed, fine, GripperTool \WObj:= DefaultFrame;
    SetDO out1, 1; ! gripping the piece
    WaitTime 1; ! waiting for the mechanism to grip correctly
    SetDO out1, 0;
    MoveL Brick_Upper, Approach_Speed, fine, GripperTool \WObj:= DefaultFrame;
    MoveAbsJ Home, Fast_Speed, fine, GripperTool \WObj:= DefaultFrame;
    MoveJ Table_Upper, Fast_Speed, fine, GripperTool \WObj:= DefaultFrame;
    MoveL Table_Lower, Approach_Speed, fine, GripperTool \WObj:= DefaultFrame;
    SetDO out2, 1; ! releasing the piece
    WaitTime 1;
    SetDO out2, 0; ! waiting the mechanism to release correctly
    MoveL Table_Upper, Approach_Speed, fine, GripperTool \WObj:=DefaultFrame;
    MoveAbsJ Home, Fast_Speed, fine, GripperTool \WObj:= DefaultFrame;
ENDPROC
```

Fig. 139 – Pick_Place procedure robot plugin

```
PROC Pick_Place(robtarget Brick_Upper, robtarget   Brick_Lower, robtarget Table_Upper, robtarget Table1_Lower)
    SetDO out1, 0;
    SetDO out2, 0; ! signal initialisation
    MoveJ Brick_Upper, Fast_Speed, fine, GripperTool \WObj:= BrickFrame;
    MoveL Brick_Lower, Approach_Speed, fine, GripperTool \WObj:= BrickFrame;
    SetDO out1, 1; ! gripping the piece
    WaitTime 1; ! waiting for the mechanism to grip correctly
    SetDO out1, 0;
    MoveL Brick_Upper, Approach_Speed, fine, GripperTool \WObj:= BrickFrame;
    MoveAbsJ Home, Fast_Speed, fine, GripperTool \WObj:= DefaultFrame;
    MoveJ Table_Upper, Fast_Speed, fine, GripperTool \WObj:= TableFrame;
    MoveL Table_Lower, Approach_Speed, fine, GripperTool \WObj:= TableFrame;
    SetDO out2, 1; ! releasing the piece
    WaitTime 1;
    SetDO out2, 0; ! waiting the mechanism to release correctly
    MoveL Table_Upper, Approach_Speed, fine, GripperTool \WObj:= TableFrame;
    MoveAbsJ Home, Fast_Speed, fine, GripperTool \WObj:= DefaultFrame;
ENDPROC
```

Fig. 140 – Pick_Place procedure robot component plugin

- We can now **delete every line of code in the raw main procedure except the ConfL instruction** (explanation in the manual mode section) and **call eight time our Pick_Place procedure with different targets argument**. Add at the beginning of the main procedure a MoveAbsJ to home instruction to ensure the start position of the robot. The main procedure should look like this **[141]**. The variable declaration part of your RAPID program should be like this **[142]**.

You have now a structured, functional and readable program !

```
    PROC Main()
        ConfL \Off;
        MoveAbsJ Home, Fast_Speed, fine, GripperTool \WObj:= DefaultFrame;
        Pick_Place Brick1_Upper, Brick1_Lower, Table1_Upper, Table1_Lower;
        Pick_Place Brick2_Upper, Brick2_Lower, Table2_Upper, Table2_Lower;
        Pick_Place Brick3_Upper, Brick3_Lower, Table3_Upper, Table3_Lower;
        Pick_Place Brick4_Upper, Brick4_Lower, Table4_Upper, Table4_Lower;
        Pick_Place Brick5_Upper, Brick5_Lower, Table5_Upper, Table5_Lower;
        Pick_Place Brick6_Upper, Brick6_Lower, Table6_Upper, Table6_Lower;
        Pick_Place Brick7_Upper, Brick7_Lower, Table7_Upper, Table7_Lower;
        Pick_Place Brick8_Upper, Brick8_Lower, Table8_Upper, Table8_Lower;
    ENDPROC
ENDMODULE
```

Fig. 141 – Main procedure

```
CONST extjoint extj := [9E9,9E9,9E9,9E9,9E9,9E9];
CONST confdata conf := [0,0,0,0];
PERS wobjdata DefaultFrame := [FALSE, TRUE, "", [[0, 0, 0], [1, 0, 0, 0]], [[0, 0, 0], [1, 0, 0, 0]]];
TASK PERS speeddata Fast_Speed := [2000,2000,2000,2000];
TASK PERS speeddata Approach_Speed := [100,100,100,100];
```

Fig. 142 – Variable declaration part

```
! Targets declaration
!Home
CONST jointtarget Home :=  [[0,-0,-0,0,-0,0],extj];

!Brick 1

CONST robtarget Brick1_Upper := [[172.5,-655,262.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick1_Lower := [[172.5,-655,62.5],[0,1,0,0],conf,extj];
CONST robtarget Table1_Upper := [[1303,-415,1010],[0,0.70711,0.70711,0],[-2,0,0,0],extj];
CONST robtarget Table1_Lower := [[1303,-415,810],[0,0.70711,0.70711,0],conf,extj];

!Brick 2

CONST robtarget Brick2_Upper := [[172.5,-780,262.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick2_Lower := [[172.5,-780,62.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table2_Upper := [[1303,-285.571,1010],[0,0.70711,0.70711,0],[-2,0,0,0],extj];
CONST robtarget Table2_Lower := [[1303,-285.571,810],[0,0.70711,0.70711,0],[-2,0,0,0],extj];

!Brick 3

CONST robtarget Brick3_Upper := [[100.5,-655,262.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick3_Lower := [[100.5,-655,62.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table3_Upper := [[1303,-156.143,1010],[0,0.70711,0.70711,0],[-2,0,0,0],extj];
CONST robtarget Table3_Lower := [[1303,-156.143,810],[0,0.70711,0.70711,0],[-2,0,0,0],extj];

!Brick 4

CONST robtarget Brick4_Upper := [[100.5,-779,262.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick4_Lower := [[100.5,-779,62.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table4_Upper := [[1303,-26.714,1010],[0,0.70711,0.70711,0],[-2,0,0,0],extj];
CONST robtarget Table4_Lower := [[1303,-26.714,810],[0,0.70711,0.70711,0],[-2,0,0,0],extj];
!Brick 5

CONST robtarget Brick5_Upper := [[23.5,-655,262.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick5_Lower := [[23.5,-655,62.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table5_Upper := [[1303,102.714,1010],[0,0.70711,0.70711,0],[0,0,1,0],extj];
CONST robtarget Table5_Lower := [[1303,102.714,810],[0,0.70711,0.70711,0],[0,0,1,0],extj];

!Brick 6

CONST robtarget Brick6_Upper := [[23.5,-777,262.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick6_Lower := [[23.5,-777,62.5],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table6_Upper := [[1303,232.143,1010],[0,0.70711,0.70711,0],[0,0,1,0],extj];
CONST robtarget Table6_Lower := [[1303,232.143,810],[0,0.70711,0.70711,0],[0,0,1,0],extj];

!Brick 7

CONST robtarget Brick7_Upper := [[-53.5,-655,262.5],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Brick7_Lower := [[-53.5,-655,62.5],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Table7_Upper := [[1303,361.571,1010],[0,0.70711,0.70711,0],[0,0,1,0],extj];
CONST robtarget Table7_Lower := [[1303,361.571,810],[0,0.70711,0.70711,0],[0,0,1,0],extj];

!Brick 8

CONST robtarget Brick8_Upper := [[-53.5,-778,262.5],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Brick8_Lower := [[-53.5,-778,62.5],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Table8_Upper := [[1303,491,1010],[0,0.70711,0.70711,0],[0,0,1,0],extj];
CONST robtarget Table8_Lower := [[1303,491,810],[0,0.70711,0.70711,0],[0,0,1,0],extj];
```

Fig. 143 – Robot plugin targets declaration

```
! Targets declaration
!Home
CONST jointtarget Home := [[0,-0,-0,0,-0,0],extj];

!Brick 1
CONST robtarget Brick1_Upper := [[0, 0, 200], [0, 1, 0, 0], [0, 0, 0, 0], extj];
CONST robtarget Brick1_Lower := [[0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], extj];
CONST robtarget Table1_Upper := [[200, -51, 280], [0, 0.707107, 0.707107, 0], [0, 0, 0, 0], extj];
CONST robtarget Table1_Lower := [[200, -51, 80], [0, 0.707107, 0.707107, 0], [0, 0, 0, 0], extj];

!Brick 2
CONST robtarget Brick2_Upper := [[0 ,-125 ,200],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick2_Lower := [[0 ,-125 ,0],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table2_Upper := [[200, 78.43, 280], [0, 0.707107, 0.707107, 0],[-2,0,0,0],extj];
CONST robtarget Table2_Lower := [[200, 78.43, 80], [0, 0.707107, 0.707107, 0],[-2,0,0,0],extj];

!Brick 3
CONST robtarget Brick3_Upper := [[-90 ,0 ,200],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick3_Lower := [[-90 ,0 ,0],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table3_Upper := [[200, 207.86, 280], [0, 0.707107, 0.707107, 0] ,[-2,0,0,0],extj];
CONST robtarget Table3_Lower := [[200, 207.86, 80], [0, 0.707107, 0.707107, 0] ,[-2,0,0,0],extj];

!Brick 4
CONST robtarget Brick4_Upper := [[-90 ,-125 ,200],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick4_Lower := [[-90 ,-125 ,0],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table4_Upper := [[200, 337.29, 280], [0, 0.707107, 0.707107, 0],[-2,0,0,0],extj];
CONST robtarget Table4_Lower := [[200, 337.29, 80], [0, 0.707107, 0.707107, 0],[-2,0,0,0],extj];

!Brick 5
CONST robtarget Brick5_Upper := [[-180 ,0 ,200],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick5_Lower := [[-180 ,0 ,0],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table5_Upper := [[200, 466.71, 280], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];
CONST robtarget Table5_Lower := [[200, 466.71, 80], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];

!Brick 6
CONST robtarget Brick6_Upper := [[-180 ,-125 ,200],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Brick6_Lower := [[-180 ,-125 ,0],[0,1,0,0],[-2,0,1,1],extj];
CONST robtarget Table6_Upper := [[200, 596.14, 280], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];
CONST robtarget Table6_Lower := [[200, 596.14, 80], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];

!Brick 7
CONST robtarget Brick7_Upper := [[-270 ,0 ,200],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Brick7_Lower := [[-270,0 ,0],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Table7_Upper := [[200, 725.57, 280], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];
CONST robtarget Table7_Lower := [[200, 725.57, 80], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];

!Brick 8
CONST robtarget Brick8_Upper := [[-270 ,-125 ,200],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Brick8_Lower := [[-270 ,-125 ,0],[0,1,0,0],[-3,0,0,1],extj];
CONST robtarget Table8_Upper := [[200, 855, 280], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];
CONST robtarget Table8_Lower := [[200, 855, 80], [0, 0.707107, 0.707107, 0],[0,0,1,0],extj];
```

Fig. 144 – Robot component plugin targets declaration

5.6 Signals creation

If you apply the changes in the RAPID window, you will normally receive an error in the output of your code because **we haven't yet created the two signals OUT1 and OUT2 that are generating the gripping and releasing actions of our gripper tool**.

To create new signals, we need to go to the **controller window, click the configuration button and select I/O system**. In the **type column** of the window that has just popped up, **select signal [145]**. You should have the following table **[146]** that gather every signal currently installed in the simulated ICR5 armory controller.

Fig. 145 – Type column

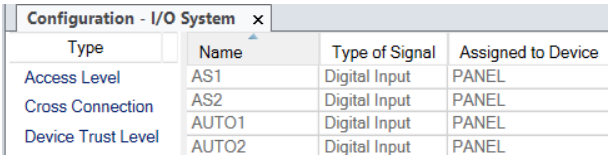


Fig. 146 – Beginning of signal table

To **create a new signal, right-click on the table and select new signal**. With our basic signal usage, we only need to enter the **name, the type and the access level of our new signal**. The following parameters should be chosen for the OUT1 signal **[147]**. When you click ok, your signal is created and installed in the ICR5. To fulfill the installation, you need to restart your ICR5 controller, but we won't do it until we have created every signal we need (so just click ok on the warning from the instance editor **[148]**). OUT1 should now appear in the table **[149]**. **Right-click on the OUT1 row, select copy signal and change the name to "out2"**. This will create the second gripper signal.

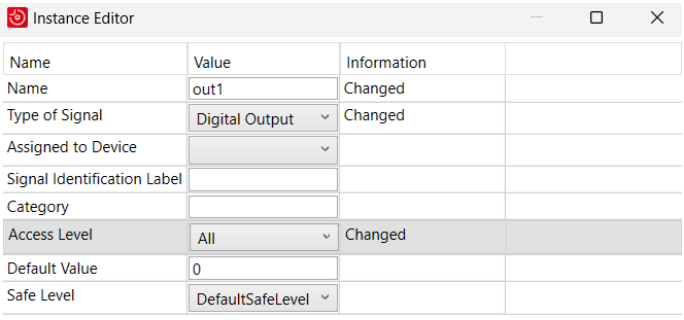


Fig. 147 – out1 creation

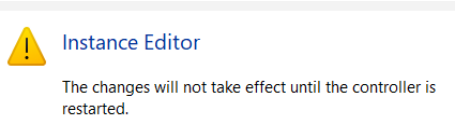


Fig. 148 – Instance editor warning

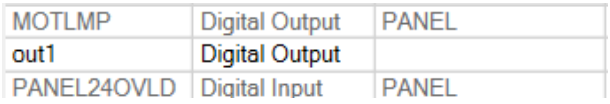


Fig. 149 – out 1 created

For our simulation, we also want the block to move and stay still when picked and placed. For this, we will **create eight other signals (b1 to b8 boolean) that will work as a gripped state flags in our program**. As for out1 and 2, b signals are **digital output** with an "all" access level. As for out1, right-click on the table and create the signal b1. You can then copy and change the name of b1 signal 8 times. You should have these following rows **[150]** when you have finished the b signals creation.

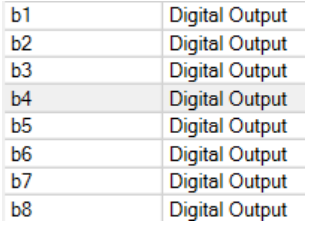


Fig. 150 – b signals created

You can finally **restart your controller (restart button)** to finalize the installation of the signals. We now need to associate actions to each of the created signal.

5.7 Simulation

To **associate signals with actions**, we need to go to the simulation window and click the little arrow next to configure button panel **[151]**. This will **launch the event manager**.

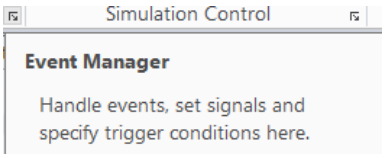


Fig. 151 – Launch event manager

From the event manager window, we need to **add event [152]** (one for each signal). When you create an event, you need to enter :

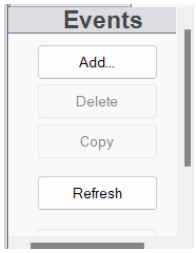


Fig. 152 – Add events

- **The type of trigger**, in our case it will always be a I/O signal changed type of trigger **[153]**.

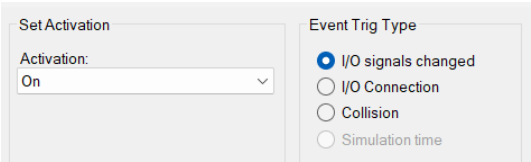


Fig. 153 – Add events

- **The signal from the I/O signal table list and the trigger condition** (value of the signal) **[154]**.
- **The action type**. For out1 and out2, we will **change the gripper tool mechanism to specific joint positions** (closed and unclosed gripper) **[156]**. For b signals, we will **attach and detach the object to the gripper tool mechanism [157]. Attention, you need to select the keep position option** Attention, we will create two events for each b signal : the first one will attach the block, to the gripper tool when the value is 1, the other one will detach the block when the value is 0 **[155]**.

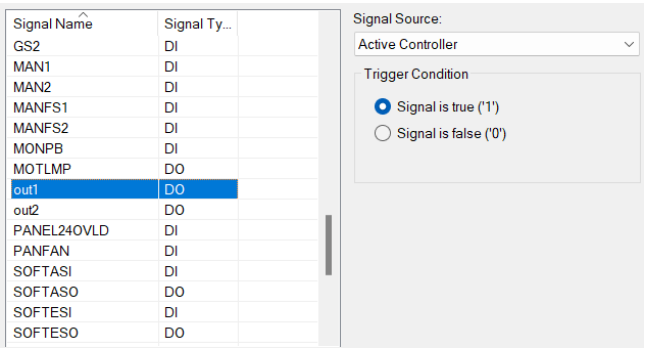


Fig. 154 – Trigger type

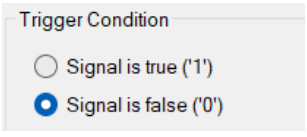


Fig. 155 – Change trigger value

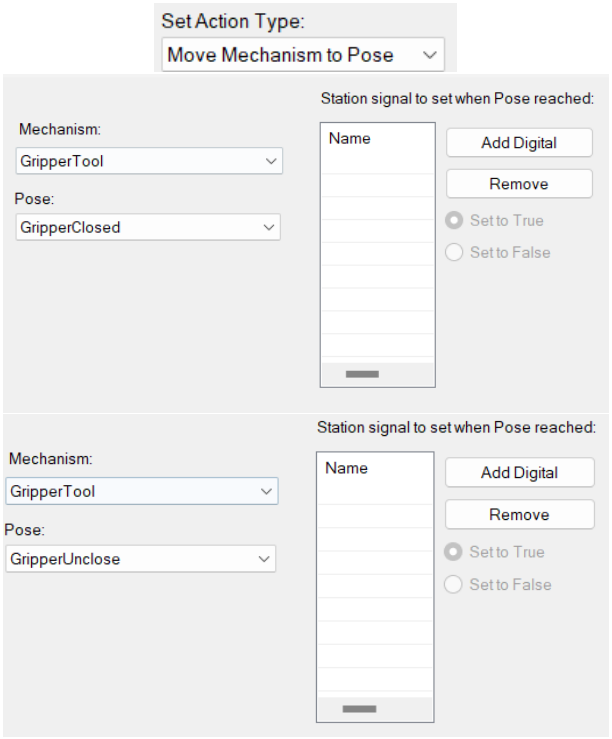


Fig. 156 – Out 1 and 2 events

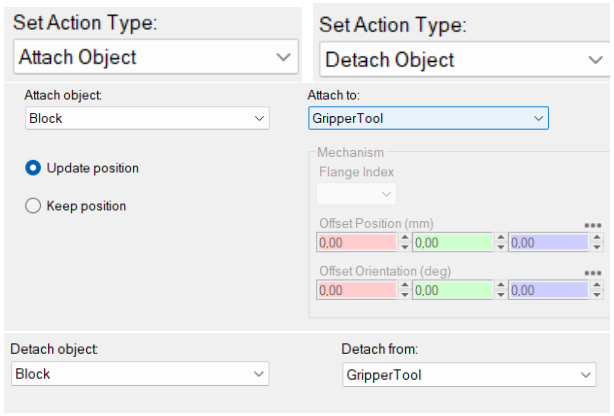


Fig. 157 – b signals event

When you have created the two types of signal for b1, **copy them eight times with the copy button**, select the new signals one by one and modify them with the action and trigger menu below, it will be faster. When every event is created, you should have the following event : **[158]**.

On	I/O	IRB4600_45...	out1	1	Move Me...		Move Mechanism ...	GripperTool : GripperCl...	
On	I/O	IRB4600_45...	out2	1	Move Me...		Move Mechanism ...	GripperTool : GripperUn...	
On	I/O	IRB4600_45...	b1	1	Attach O...		Attach Object	GripperTool -> Block	
On	I/O	IRB4600_45...	b2	1	Attach O...		Attach Object	GripperTool -> Block_2	
On	I/O	IRB4600_45...	b3	1	Attach O...		Attach Object	GripperTool -> Block_3	
On	I/O	IRB4600_45...	b4	1	Attach O...		Attach Object	GripperTool -> Block_4	
On	I/O	IRB4600_45...	b5	1	Attach O...		Attach Object	GripperTool -> Block_5	
On	I/O	IRB4600_45...	b6	1	Attach O...		Attach Object	GripperTool -> Block_6	
On	I/O	IRB4600_45...	b7	1	Attach O...		Attach Object	GripperTool -> Block_7	
On	I/O	IRB4600_45...	b8	1	Attach O...		Attach Object	GripperTool -> Block_8	
On	I/O	IRB4600_45...	b1	0	Detach O...		Detach Object	GripperTool <- Block	
On	I/O	IRB4600_45...	b2	0	Detach O...		Detach Object	GripperTool <- Block_2	11:18:43
On	I/O	IRB4600_45...	b3	0	Detach O...		Detach Object	GripperTool <- Block_3	11:18:54
On	I/O	IRB4600_45...	b4	0	Detach O...		Detach Object	GripperTool <- Block_4	11:19:03
On	I/O	IRB4600_45...	b5	0	Detach O...		Detach Object	GripperTool <- Block_5	11:19:14
On	I/O	IRB4600_45...	b6	0	Detach O...		Detach Object	GripperTool <- Block_6	11:19:20
On	I/O	IRB4600_45...	b7	0	Detach O...		Detach Object	GripperTool <- Block_7	11:19:26
On	I/O	IRB4600_45...	b8	0	Detach O...		Detach Object	GripperTool <- Block_8	11:19:35

Fig. 158 – Every event we need

To make b signals picked flags work, we need to implement some simple instructions in our RAPID program, in the Pick_Place procedure. We will **use a variable named brickCounter in our procedure**, to know which signal to use depending on the brick number.

Thus, we need to **declare 8 variables (brickCounter1 to brickCounter8) in the variable declaration part** of our program that will be put in argument to our Pick_Place procedure (here is the new Pick_Place procedure declaration **[160]** and a call example **[161]**). These variable are **constant, and their data type is num [159]** (brickCounter1 equal 1, brickCounter2 equal 2 ...).

```
!Brick 1
CONST num brickCounter1 := 1;
CONST robtarget Brick1_Upper := [[0, 0, 200], [0, 1, 0, 0], [0, 0, 0, 0], extj];
CONST robtarget Brick1_Lower := [[0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0], extj];
CONST robtarget Table1_Upper := [[200, -51, 280], [0, 0.707107, 0.707107, 0], [0, 0, 0, 0], extj];
CONST robtarget Table1_Lower := [[200, -51, 80], [0, 0.707107, 0.707107, 0], [0, 0, 0, 0], extj];
```

Fig. 159 – BrickCounter declaration

```
PROC Pick_Place(robtarget Brick_Upper, robtarget   Brick_Lower, robtarget Table_Upper, robtarget Table_Lower, num brickCounter)
```

Fig. 160 – New Pick_Place procedure declaration

```
Pick_Place Brick3_Upper, Brick3_Lower, Table3_Upper, Table3_Lower, brickCounter3;
```

Fig. 161 – New Pick_Place procedure call

Then, with a **condition structure close to the switch case structure** in C language, we ensure that the good signal is updated with the suitable value in the procedure **[162]** **at the right moment** (after the wait times for the gripping and releasing action).

```
TEST brickCounter
    CASE 1:
        SetDO b1, 1;
    CASE 2:
        SetDO b2, 1;
    CASE 3:
        SetDO b3, 1;
    CASE 4:
        SetDO b4, 1;
    CASE 5:
        SetDO b5, 1;
    CASE 6:
        SetDO b6, 1;
    CASE 7:
        SetDO b7, 1;
    CASE 8:
        SetDO b8, 1;
ENDTEST
```

```
TEST brickCounter
    CASE 1:
        SetDO b1, 0;
    CASE 2:
        SetDO b2, 0;
    CASE 3:
        SetDO b3, 0;
    CASE 4:
        SetDO b4, 0;
    CASE 5:
        SetDO b5, 0;
    CASE 6:
        SetDO b6, 0;
    CASE 7:
        SetDO b7, 0;
    CASE 8:
        SetDO b8, 0;
ENDTEST
```

Fig. 162 – Test case RAPID structure (gripping and releasing)

Everything is now finalized, you can try to start your simulation and enjoy the final result. Don't forget to reset the simulation at the end of it. Change the speed declaration to make the robot move faster (GrassHopper simulation are really slow).

5.8 Help for debug

If you face some problems, here is a quick list of problem and solution that might help you :

- **Code error :** *Argument WObj specifies a mechanical unit name, which is not activated or is unknown in the system.* You need to put the boolean wobjdata.ufprog to TRUE in work object data declaration.
- **Code error :** *The currently defined TCP load for robot ROB_1 exceeds the maximum allowed load for the robot model.* Change the center of mass coordinates in the tool declaration (reduce them slightly).
- **Robot is doing every action but offsetted about Z axis**. Place the robot at (0,0,0) in the modeling panel and update task frame. Restart the controller, place it back to (0, 0, 500), update the task frame reference and restart as second time the controller.
- **Blocks are not gripped correctly (not inside the gripper), or teleported when gripping.** Verify your events in the simulation event manager. Verify the order of your blocks (blocks 1 3 5 7 first row, blocks 2 4 6 8 second row).
- If you had an error during your simulation, and you decided to reset it before the end of the program, **some signals values might be set to 1 and thus generating errors.** You must set them back manually by using clicking on Inputs/Outputs button of the controller window.
- **Pay attention to your program pointer** when you launch a simulation. It must be at the beginning of your main procedure. Set the program pointer to the main procedure in all tasks, if not.

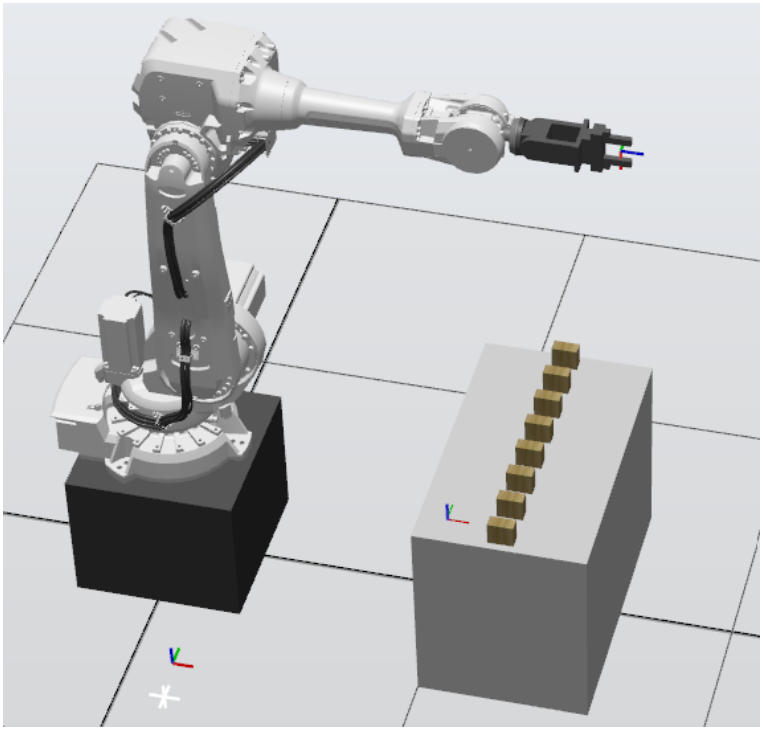


Fig. 163 – Pick&Place done

Conclusion

I hope this guide has been helpful to you. You should now be able to transfer your GrassHopper robotic simulations into RobotStudio, from tool integration to event creation, jog the real robot in manual mode, and have a basic understanding of RAPID code programming, calibration, and robotic arm systems. This guide only scratches the surface of what you can achieve with the combination of RobotStudio and GrassHopper. I encourage you to expand your knowledge with the official ABB manual, create more ambitious simulations beyond this pick-and-place tutorial, and continue advancing in your robotics journey.

Feel free to share your feedback with me (contact information can be found on my website *thibault-courtois.com*). If you've discovered other interesting features in RobotStudio or additional synergies between RobotStudio and GrassHopper, I encourage you to contribute by adding complementary sections and tutorials to this guide. I learned a lot while creating this guide during my IndexLab internship, and I like to think I'm now an advanced ABB robot user (though perhaps not entirely). I'm proud of this work, and thank you for the attention you put in my work.

Finally, I would like to express my gratitude to the wonderful IndexLab team, with whom I had the pleasure of working while creating this guide. The team comprises passionate architects, researchers, and engineers dedicated to innovative robotic manufacturing applications. I would also like to thank Mr. Ruttico, head of IndexLab, for allowing me to use the IndexLab ABB robot and for providing me with a RobotStudio license, which was essential for completing this guide.

Complementary resources

5.9 ABB official manual

Here are some links for official ABB manuals concerning RAPID coding and RobotStudio :

Operating manual : Getting started, IRC5 and RobotStudio

Operating manual : RobotStudio

Operating manual IRC5 with FlexPendant

Operating manual : Introduction to RAPID

Technical reference manual : Rapid overview

Technical reference manual : RAPID Instructions, Functions and Data types

5.10 GitHub link

Here is the link for my GitHub page related to this guide :

GitHub