The Pennsylvania State University

College of Arts and Architecture

Stuckeman Center for Design Computing

Algorithmic Tectonics

A tutorial on RAPID coding and ABB RobotStudio

Advisor: Professor Daniel Cardoso

Shokofeh Darbari

Spring 2015

Contents

Introduction

This tutorial is an introduction to RAPID programming and also presents the basic steps of using RobotStudio, ABB's simulation and offline programming software. While some of the RAPID programming features are left out due to their complexity, the most essential parts are described so that it can be followed by beginners easily.

Regarding the RobotStudio software, two different approaches will be introduced. In the first part (Section 5), simulation of the robot using RAPID programming will be explained, which provides a professional way to simulate and also move an ABB robotic arm.

The second approach will be discussed in the Section 6, which is the simulation of the robot using visual components, without a need for RAPID programming.

The tutorial follows a step-by-step format, including a snapshot of the software at each step, in order to help to better understand the concepts and actions.

In order to get a better understanding of RAPID programming terms and instructions, we refer the reader to "Technical reference manual: RAPID Instructions, Functions and Data types", provided by ABB.

Section 1: Rapid Basics

1.1 About RAPID

If you want a computer to do something, a program is required. RAPID is a programming language for writing such a program.

The native language of computers consists of only zeros and ones. This is virtually impossible for humans to understand. Therefore computers are taught to understand a language that is relatively easy to understand - a high level programming language. RAPID is a high level programming language, it uses some English words (like IF and FOR) to make it understandable for humans.

The following shows a simple RAPID program example:

```
MODULE MainModule
    VAR num length;
    VAR num width;
    VAR num area;
    PROC main()
        length := 10;
        width := 5;
        area := length * width;
    END PROC
ENDMODUL
```

1.2 RAPID data

1.2.1. Variables

There are many different data types in RAPID. For now, we will focus on the three general data types:

Data Type	Description
Num	Numerical data, can be both integer and decimal number. E.g. 10 or 3.14159.
String	A text string. E.g. "This is a string". Maximum of 80 characters.
Bool	A boolean (logical) variable. Can only have the values TRUE or FALSE.

Declaring a variable is the way of defining a variable name and which data type it should have. A variable is declared using the keyword VAR, according to the syntax:

VAR *datatype identifier*;

Example:

```
VAR num length;
VAR string name;
VAR bool finished;
```

A value is assigned to a variable using the instruction **:=**

```
length := 10;
name := "John"
finished := TRUE;
```

1.2.2. Persistent Variables

A persistent variable is basically the same as an ordinary variable, but with one important difference.

A persistent variable remembers the last value it was assigned, even if the program is stopped and started from the beginning again.

Declaring a persistent variable:

A persistent variable is declared using the keyword PERS. At declaration an initial value must be assigned.

```
PERS num nbr := 1;

PERS string string1 := "Hello";
```

Example

```
PERS num nbr := 1;

PROC main()

nbr := 2;

ENDPROC
```

If this program is executed, the initial value is changed to 2. The next time the program is executed the program code will look like this:

```
PERS num nbr := 2;

PROC main()

nbr := 2;

ENDPROC
```

1.2.3. Constants

A constant contains a value, just like a variable, but the value is always assigned at declaration and after that the value can never be changed. The constant can be used in the program in the same way as a variable, except that it is not allowed to assign a new value to it.

Declaring a constant variable:

The constant is declared using the keyword CONST followed by data type, identifier and assignment of a value.

```
CONST num gravity := 9.81;

CONST string greating := "Hello"
```

1.2.4 Operators

Basically there is three different operators in RAPID:

- **Numerical operators**
- **Relational operators**
- **String operators**

Numerical operators

Operator	Description	Example
+	Addition	reg1 := reg2 + reg3;
-	Subtraction/ unary minus	reg1 := reg2 - reg3; reg1 := -reg2;
*	Multiplication	reg1 := reg2 * reg3;
/	Division	reg1 := reg2 / reg3;

Relational operators

Operator	Description	Example
=	equal to	flag1 := reg1 = reg2; flag1 is TRUE if reg1 equals reg2
<	Less than	flag1 := reg1 < reg2; flag1 is TRUE if reg1 is less than reg2
>	Greater than	flag1 := reg1 > reg2; flag1 is TRUE if reg1 is greater than reg2
=<	less than or equal to	flag1 := reg1 <= reg2; flag1 is TRUE if reg1 is less than or equal to reg2
>=	greater than or equal to	flag1 := reg1 >= reg2; flag1 is TRUE if reg1 is greater than or equal to reg2
<>	not equal to	flag1 := reg1 <> reg2; flag1 is TRUE if reg1 is not equal to reg2

String operators

Operator	Description	Example
+	String concatenation	VAR string firstname := "John"; VAR string lastname := "Smith"; VAR string fullname; fullname := firstname + " " + lastname; The variable fullname will contain the string "John Smith".

1.3. Controlling the program flow

The program examples we have seen so far are executed sequentially, from top to bottom. For more complex programs, we may want to control which code is executed, in which order, and how many times. First we will have a look at how to set up conditions for if a program sequence should be executed or not.

1.3.1. IF THEN ELSE

The IF instruction can be used when a set of statements only should be executed if a specified condition is met.
If the logical condition in the IF statement is true, then the program code between the keywords THEN and ENDIF is executed. If the condition is false, that code is not executed and the execution continues after ENDIF.
An IF statement can also contain program code to be executed if the condition is false. If the logical condition in the IF statement is true, then the program code between the keywords THEN and ELSE is executed. If the condition is false, then the code between the keywords ELSE and ENDIF is executed.

1.3.2. FOR LOOP

Another way of controlling the program flow is to repeat a program code sequence a number of times

1.3.3. WHILE LOOP

The repeating of a code sequence can be combined with the conditional execution of the code sequence. With the WHILE loop the program will continue repeating the code sequence as long as the condition is true.

1.4. Rules and recommendation for RAPID syntax

1.4.1. Semicolon

The general rule is that each statement ends with a semicolon.

Examples

```
Variable declaration:
VAR num length;
Assigning values:
area := length * width;
Most instruction calls:
MoveL p10,v1000,fine,tool0;
```

Exceptions

Instruction Keyword	Terminating Keyword
IF	ENDIF
FOR	ENDFOR
WHILE	ENDWHILE
PROC	ENDPROC

1.4.2. Comments

A line starting with ! will not be interpreted by the robot controller. Use this to write comments about the code.

Examples

```
! Calculate the area of the rectangle
area := length * width;
```

1.4.3. Capitalized keyword

RAPID is not case sensitive, but it is recommended that all reserved words (e.g. VAR, PROC) are written in capital letters.

1.4.4. Indentations

To make the programming code easy to grasp, use indentation. Everything inside a PROC (between PROC and ENDPROC) should be indented. Everything inside an IF-, FOR- or WHILE statement should be further indented

Examples

```
VAR bool repeat;
VAR num times;
PROC main()
    repeat := TRUE;
    times := 3;
    IF repeat THEN
        FOR i FROM 1 TO times DO
            MoveL Topoint, V100,fine, tool0;
        ENDFOR
    ENDIF
END PROC
```

Section 2: RAPID robot functionality

2.1. Move instruction

The advantage with RAPID is that, except for having most functionality found in other high level programming languages, it is specially designed to control robots. Most importantly, there are instructions for making the robot move.

A simple move instruction can look like this:

```
MoveL p10, v1000, fine, tool0;
```

where:

• MoveL is an instruction that moves the robot linearly (in a straight line) from its current position to the specified position.

• p10 specifies the position that the robot shall move to.

• v1000 specifies that the speed of the robot shall be 1000 mm/s.

• fine specifies that the robot shall go exactly to the specified position and not cut any corners on its way to the next position.

• tool0 specifies that it is the mounting flange at the tip of the robot that should move to the specified position.

MoveL syntax

```
MoveL ToPoint Speed Zone Tool;
```

- *ToPoint*

The destination point defined by a constant of data type *robtarget*. When programming with the FlexPendant you can assign a robtarget value by pointing out a position with the robot. When programming offline, it can be complicated to calculate the coordinates for a position.

For now, let us just accept that the position x=600, y=-100, z=800 can be declared and assigned like this:

CONST robtarget p10 := [[600, -100, 800], [1, 0, 0, 0], [0, 0, 0,0], [9E9, 9E9, 9E9, 9E9, 9E9, 9E9]];

MoveL syntax

MoveL *ToPoint Speed Zone Tool*;

- *Speed*

The speed of the movement defined by a constant of data type speeddata. There are plenty of predefined values, such as:

Predifined Speeddata	**Value**
V5	5 mm/s
V100	100 mm/s
v1000	1000 mm/s

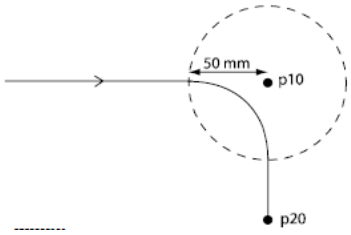
- *Zone*

Specifies a corner zone defined by a constant of data type zonedata. There are many predefined values, such as:

Predifined Speeddata	**Value**
Fine	The robot will go to exactly the specified position
Z10	The robot path can cut corners when it is less than 10 mm from *ToPoint*.
Z50	The robot path can cut corners when it is less than 50 mm from *ToPoint*.

Example

```
MoveL p10, v1000, z50, tool0;
MoveL p20, v1000, fine, tool0;
```



- *Tool*

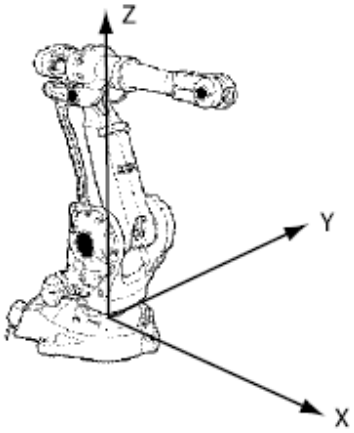
Specifies the tool that the robot is using, defined by a persistent variable of data type *tooldata*.

tool0 is a predefined tool, representing the robot without any tool mounted on it, and should not be declared or assigned. Any other tool should be declared and assigned before being used.

2.1.2 Coordinate system

- Base coordinate system

The position that a move instruction moves to is specified as coordinates in a coordinate system. If no coordinate system is specified, the position is given relative to the robot base coordinate system (also called base frame). The base coordinate system has its origin in the robot base.



- Customized coordinate system

Another coordinate system can be defined and used by move instructions.

Which coordinate system the move instruction shall use is specified with the optional argument *\WObj*.

```
MoveL p10, v1000, z50, tool0\WObj:=wobj1;
```

2.1.3. Wobjdata- Work object data

Wobjdata is used to describe the work object that the robot welds, processes, moves within, etc.

If work objects are defined in a positioning instruction, the position will be based on the coordinates of the work object. The advantages of this are as follows:

• If position data is entered manually, such as in off-line programming, the values can often be taken from a drawing.

• Programs can be reused quickly following changes in the robot installation. If, for example, the fixture is moved, only the user coordinate system has to be redefined.

Work Obj Syntax

```
PERS wobjdata wobj2 :=[ rob hold, ufprog, "ufmec", uframe, oframe];
```

- *robhold*

robot hold

Data type: bool

Defines whether or not the robot in the actual program task is holding the work object:

• TRUE: The robot is holding the work object, i.e. using a stationary tool.

• FALSE: The robot is not holding the work object, i.e. the robot is holding the tool.

- *ufprog*

user frame programmed

Data type: bool

Defines whether or not a fixed user coordinate system is used:

• TRUE: Fixed user coordinate system.

• FALSE: Movable user coordinate system, i.e. coordinated external axes are used. Also to be used in a MultiMove system in semicoordinated or synchronized coordinated mode.

- *ufmec*

user frame mechanical unit

Data type: string

The mechanical unit with which the robot movements are coordinated. Only specified in the case of movable user coordinate systems (ufprog is FALSE).

Specify the mechanical unit name defined in system parameters, e.g. orbit_a.

- *Uframe*

user frame

Data type: pose

The user coordinate system, i.e. the position of the current work surface or fixture:

• The position of the origin of the coordinate system (x, y and z) in mm.

• The rotation of the coordinate system, expressed as a quaternion (q1, q2, q3, q4).

If the robot is holding the tool, the user coordinate system is defined in the world coordinate system (in the wrist coordinate system if a stationary tool is used).

For movable user frame (ufprog is FALSE), the user frame is continuously defined by the system.

- *oframe*

object frame

Data type: pose

The object coordinate system, i.e. the position of the current work object:

• The position of the origin of the coordinate system (x, y and z) in mm.

• The rotation of the coordinate system, expressed as a quaternion (q1, q2, q3, q4).

Example

PERS wobjdata wobj2 :=[FALSE, TRUE, "", [[300, 600, 200], [1, 0, 0 ,0]], [[0, 200, 30], [1, 0, 0 ,0]]];

The work object is described using the following values:

• The robot is not holding the work object.

• The fixed user coordinate system is used.

• The user coordinate system is not rotated and the coordinates of its origin are x= 300, y = 600 and z = 200 mm in the world coordinate system.

• The object coordinate system is not rotated and the coordinates of its origin are x= 0, y= 200 and z= 30 mm in the user coordinate system.

The work object data **wobj0** is defined in such a way that the object coordinate system coincides with the world coordinate system. The robot does not hold the work object. Wobj0 can always be accessed from the program, but can never be changed (it is stored in system module BASE).

PERS wobjdata wobj0 := [FALSE, TRUE, "", [[0, 0, 0], [1, 0, 0 ,0]], [[0, 0, 0], [1, 0, 0 ,0]]];

2.1.4. Other move instruction

There are a number of move instructions in RAPID. The most common are MoveL, MoveJ and MoveC.

MoveJ

MoveJ is used to move the robot quickly from one point to another when that movement does not have to be in a straight line.

Use MoveJ to move the robot to a point in the air close to where the robot will work. A MoveL instruction does not work if, for example, the robot base is between the current position and the programmed position, or if the tool reorientation is too large. MoveJ can always be used in these cases. The syntax of MoveJ is analog with MoveL.

Example

```
MoveJ p10, v1000, fine, tool0;
```

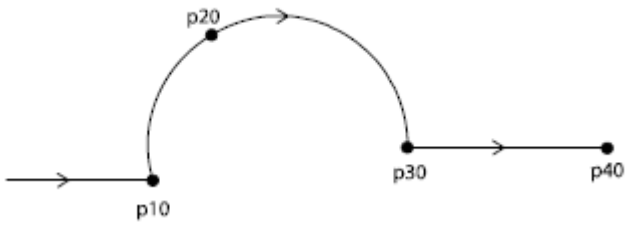
There are a number of move instructions in RAPID. The most common are MoveL, MoveJ and MoveC.

MoveC

MoveC is used to move the robot circularly in an arc.

Example

```
MoveL p10, v500, fine, tool0;
MoveC p20, p30, v500, fine, tool0;
MoveL p40, v500, fine, tool0;
```



Section 3: Structure of coding in RAPID

3.1. Procedure

The RAPID code examples we have looked at have only executed code in the procedure main. The execution automatically starts in the procedure named main, but there can be several procedures. A procedure must be declared with the keyword PROC followed by the procedure name, the procedure arguments and the program code that the procedure should execute. A procedure is called from another procedure (except main, which is automatically called when the program starts).

```
PERS tooldata tool0...
CONST robtarget p10:=
PROC main()
    ! Call the procedure draw_square
        draw_square 100;
        draw_square 200;
        draw_square 300;
        draw_square 400;
ENDPROC
PROC draw_square(num side_size)
        VAR robtarget p20;
        VAR robtarget p30;
        VAR robtarget p40;
        ! p20 is set to p10 with an offset on the y value
        p20 := Offs(p10, 0, side_size, 0);
        p30 := Offs(p10, side_size, side_size, 0);
        p40 := Offs(p10, side_size, 0, 0);
        MoveL p10, v200, fine, tPen;
        MoveL p20, v200, fine, tPen;
        MoveL p30, v200, fine, tPen;
        MoveL p40, v200, fine, tPen;
```

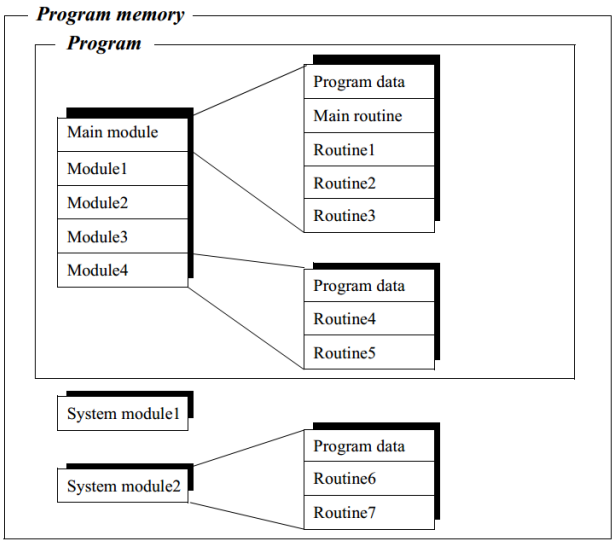
```
        MoveL p10, v200, fine, tPen;
ENDPROC
```

3.1.Modules

A RAPID program can consist of one or several modules. Each module can contain one or several procedures.

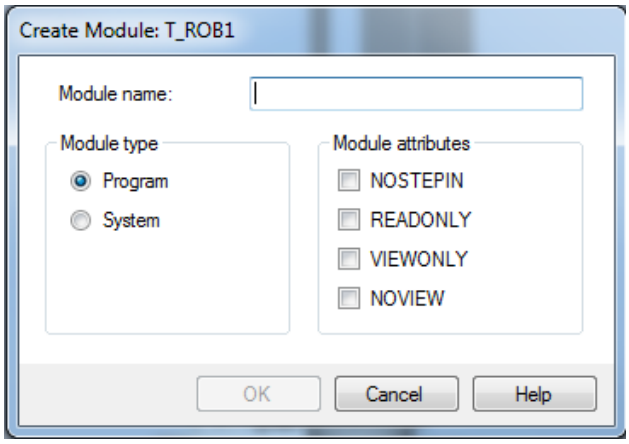
The small and simple programs that are shown here use only one module. In a more complex programming environment, some standard procedures, used by many different programs, can be placed in a separate module.

```
MODULE MainModule
...
draw_square;
...
ENDMODULE
MODULE figures_module
PROC draw_square()
...
ENDPROC
PROC draw_triangle()
...
ENDPROC
PROC draw_circle()
...
ENDPROC
ENDMODULE
```



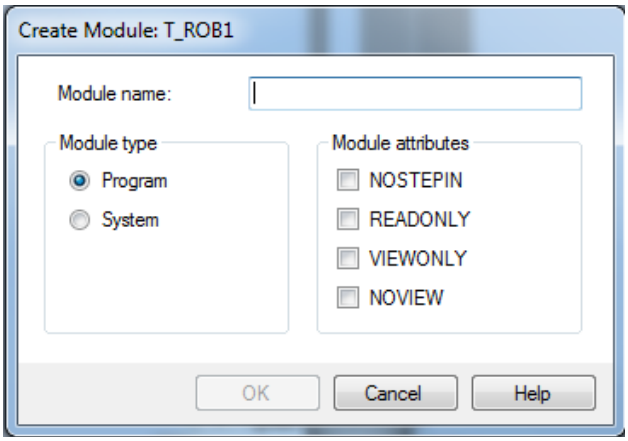
3.2.1. Program Modules

A program module is saved with the file ending .mod, e.g. *figures_module.mod*. There can only be one program active on the robot controller, i.e. only one of the modules can contain a procedure named main.



3.2.2. System Modules

A system module is saved with the file ending .sys, e.g. system_data_module.sys. Data and procedures that should be kept in the system even if the program is changed should be placed in a system module. For example, if a persistent variable of type tooldata is declared in a system module, a recalibration of the tool is preserved even if a new program is loaded


Create Module: T_ROB1
Module name:
Module type
Program
System
Module attributes
NOSTEPIN
READONLY
VIEWONLY
NOVIEW
OK
Cancel
Help

Section 4: Data with multiple value

4.1. Arrays

An array is a variable that contains more than one value. An index is used to indicate one of the values. The declaration of an array looks like any other variable, except that the length of the array is specified inside { }.

```
VAR num my_array{3};
```

An array can be assigned all its values at once. When assigning the whole array the values are surrounded by [ ] and separated by commas.

```
my_array := [5, 10, 7];
```

It is also possible to assign a value to one of the elements in an array. Which element to assign a value to is specified inside { }.

```
my_array{3} := 8;
```

4.2. Composite data types

A composite data type is a data type that contains more than one value. It is declared as a normal variable but contains a predefined number of values.

- *pos*

A simple example of a composite data type is the data type pos. It contains three numerical values (x, y and z).

The declaration looks like a simple variable:

```
VAR pos pos1;
```

Assigning all values is done like with an array:

```
pos1 := [600, 100, 800];
```

The different components have names instead of numbers. The components in pos are named x, y and z. The value in one component is identified with the variable name, a point and the component name:

```
pos1.z := 850;
```

- *orient*

The data type orient specifies the orientation of the tool. The orientation is specified by four numerical values, named q1, q2, q3 and q4.

```
VAR orient orient1 := [1, 0, 0, 0];
```

- *pose*

A data type can be composed of other composite data types. An example of this is the data type pose, which consists of one *pos* named trans and one orient named rot.

```
VAR pose pose1 := [[600, 100, 800], [1, 0, 0, 0]];

VAR pos pos1 := [650, 100, 850];

VAR orient orient1;

pose1.pos := pos1;

orient1 := pose1.rot;

pose1.pos.z := 875;
```

robtarget

robtarget is too complex a data type to explain in detail here, so we will settle for a brief explanation. *robtarget* consists of four parts:

Data Type	Name	Description
Pos	trans	X,y and z coordination
Orient	Rot	Orinetation
Confdata	Robconf	Specifies robot axes angels

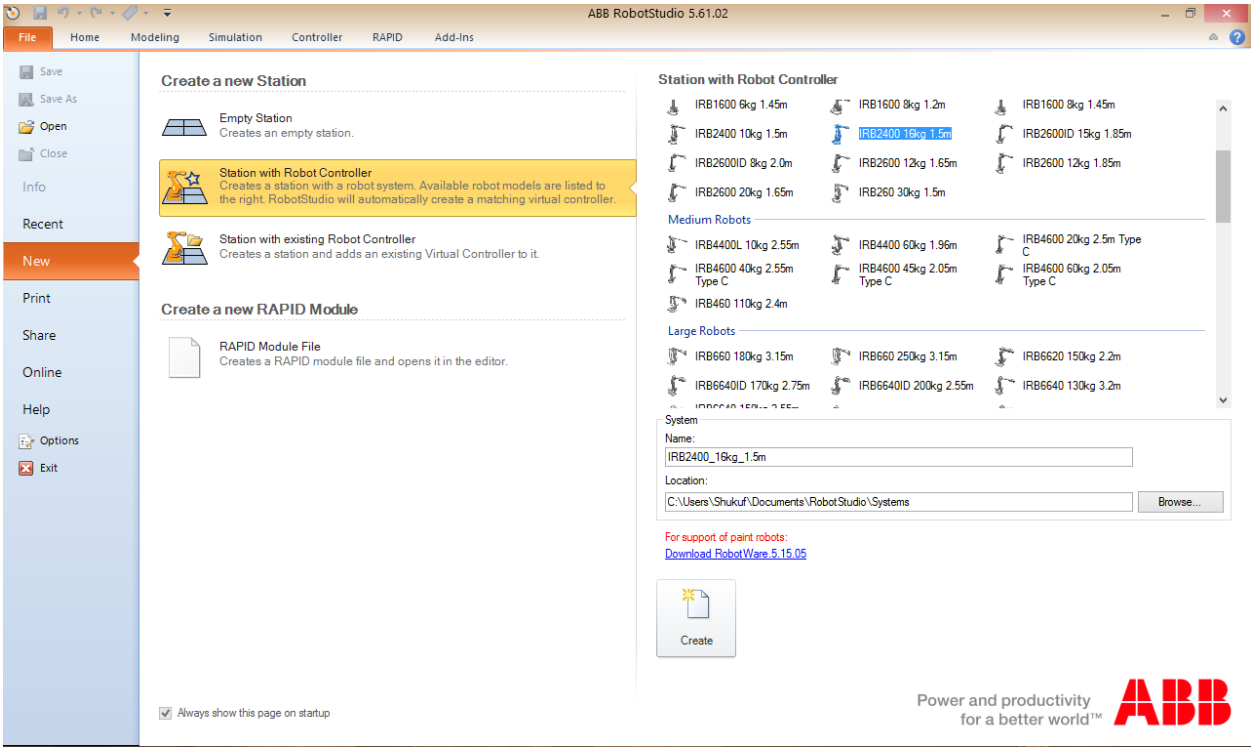
extjoint	Extax	Specifies positions for up to 6 additional axes. The value is set to 9E9 where no additional axis is used.

```
VAR robtarget p10 := [ [600, -100, 800], [0.707170, 0, 0.707170,0], [0, 0, 0, 0], [ 9E9, 9E9, 9E9, 9E9, 9E9, 9E9] ];

        ! Increase the x coordinate with 50

        p10.trans.x := p10.trans.x + 50;
```

Section 5: Simulating the robot using Rapid programming

In this section, the basic steps to start a simulation from a RAPID code will be discussed.

Step 1: Under the File menu, create a New Station with Robot Controller

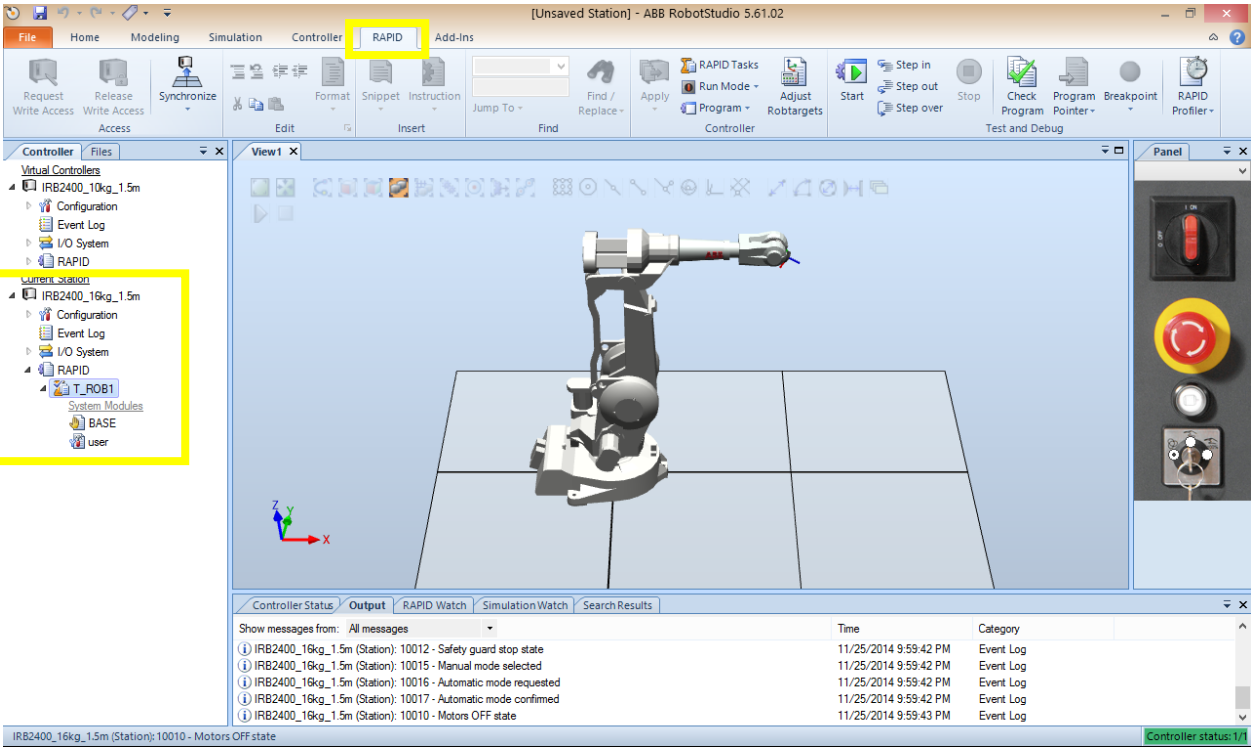


Step 2: Select the robot model (IRB2400 16kg 1.5m in our example) and click on Create

Wait for virtual controller status to connect (the status bar on the bottom right of the screen shows controller the status and changes from red to green)

Note: some messages in the output window will appear regarding the status of controller and the robot.

Step 3: Select the RAPID tab, then expand the RAPID and also T_ROB1 in the Controller window.



There are two system modules called *BASE* and *user* which have been created automatically and should not be changed.

It is important to mention that RAPID language is a module-based language. For programming the Robot we have to add modules to define procedures and functions. Here we start defining our system data including *work object* and *tool* in a module named *DataCalib*. Different targets and paths will be defined in another module called *Module 1*.

Step 4**:** Right click on *T_ROB1* and select *New Modules….* A new window will be opened to enter the name and type of the module to be created. Enter an optional name (e.g. *DataCalib*) and select the *Program* type. There is no need to give it any attribute.

CalibData module will be added to the *Controller* window under T_ROB1 as a *Program Module*. Double click on CalibData module and open it to view the contents, which looks like the following:

```
MODULE CalibData
ENDMODULE
```

All the system data including *world object* should be defined in the body of this module.

```
MODULE DataCalib
```

```
TASK PERS wobjdata Workobject_1:=[FALSE,TRUE,"",[[0,0,0],[1,0,0,0]],[[0,0,0],[1,0,0,0]]];

ENDMODULE
```

Now it is time to create *Module_1* in order to define Targets and Paths.

Step 5**:** Create another Program module (e.g. *Module_1*) by right clicking again on *T_ROB1* in the *Controller* window and selecting *New Modules….*

Open the module by double clicking on it and define targets, paths and procedures for the program.

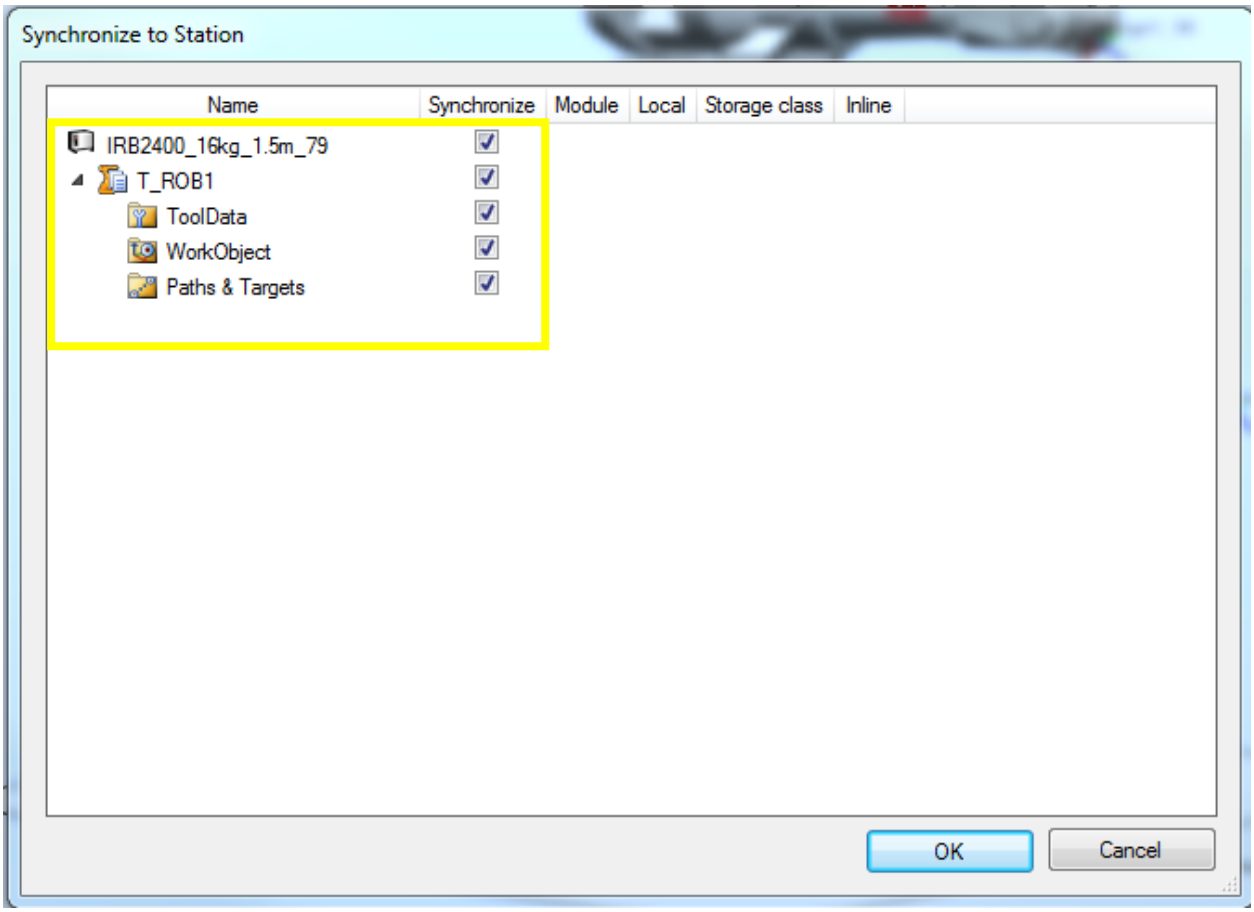
We are about to write a sample program with only two targets and a simple linear path between them. Every program has to contain at least one procedure called *main*.

```
MODULE Module_1

    CONST robtarget
    Target_10:=[[928.61214857,0,1412.499981377],[0.49999989,0,0.866025467,0],[0,0,0,0],[9E9,9
    E9,9E9,9E9,9E9,9E9]];
    CONST robtarget
    Target_20:=[[928.61214857,200,1412.499981377],[0,0,1,0],[0,0,0,0],[9E9,9E9,9E9,9E9,9E9,9E
    9]];
        PROC Path_10()
            MoveL Target_10,v1000,z100,tool0\WObj:=wobj0;
            MoveL Target_20,v1000,fine,mytool\WObj:=Workobject_1;
        ENDPROC
        PROC main()
            Path_10;
        ENDPROC

ENDMODULE
```

In the above program, we have defined two targets named *Target_10* and *Target_20* and a linear path between them named *Path_10*. Please note that paths are defined as procedures. The program also contains a *main* procedure, which calls *Path_10* to be executed.

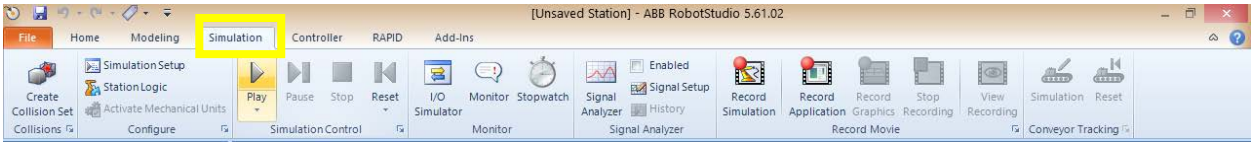
Step 6**:** The new modules have to be applied to the controller. Select the Apply icon in the top ribbon of the RobotStudio software and press Apply All. After applying the changes, the stars next to the name of the modules will disappear.

Step 7**:** Now it is time to synchronize the code to the Station. Press the Synchronize button in the top ribbon and select *Synchronize to station…* option. Check all boxes in front of the objects and press OK.

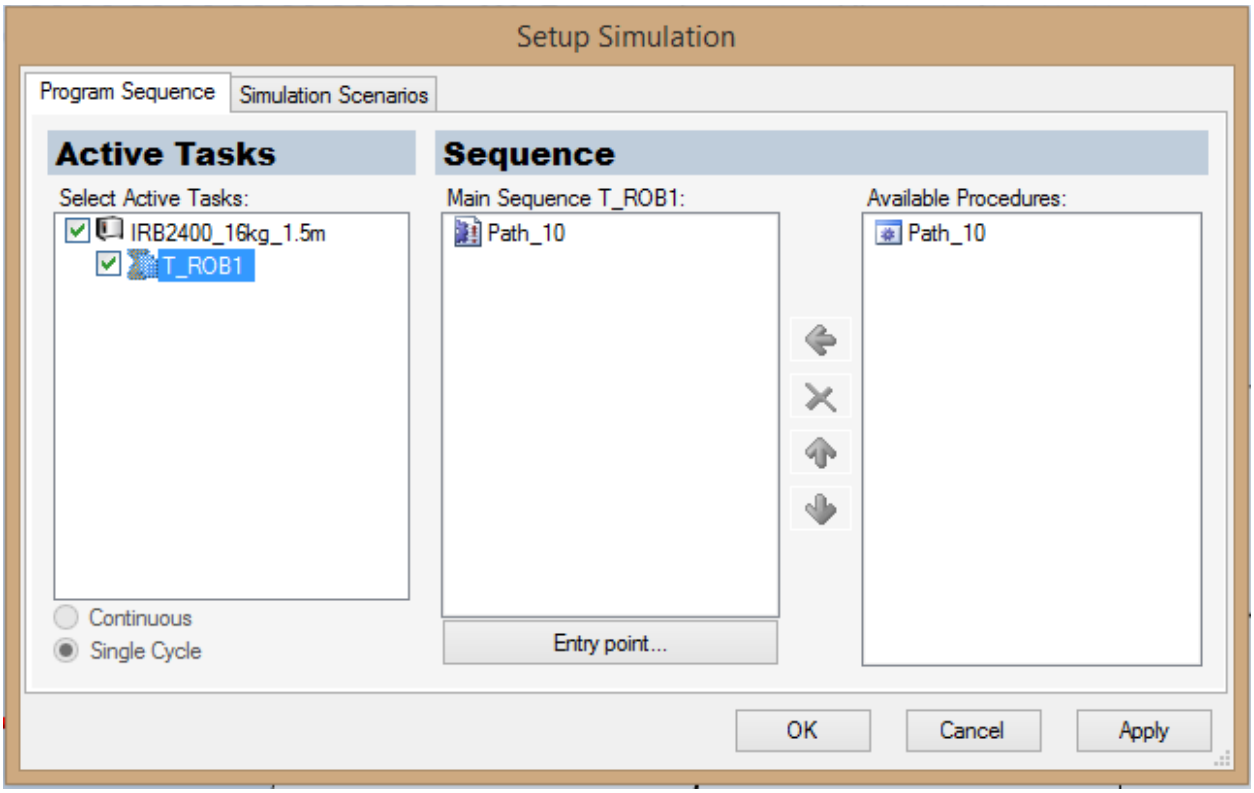


After synchronizing the station with the virtual controller, a *Synchronization to station completed* message will appear in the output window.

Step 8**:** In order to simulate our code on the virtual controller, go to the *Simulation* menu and select the *Simulation Setup* from the ribbon.



In the Simulation Setup window opened, click on T_ROB1 active task (note that all the procedures except *main* should be listed in the sequence column. Press OK to apply the changes.

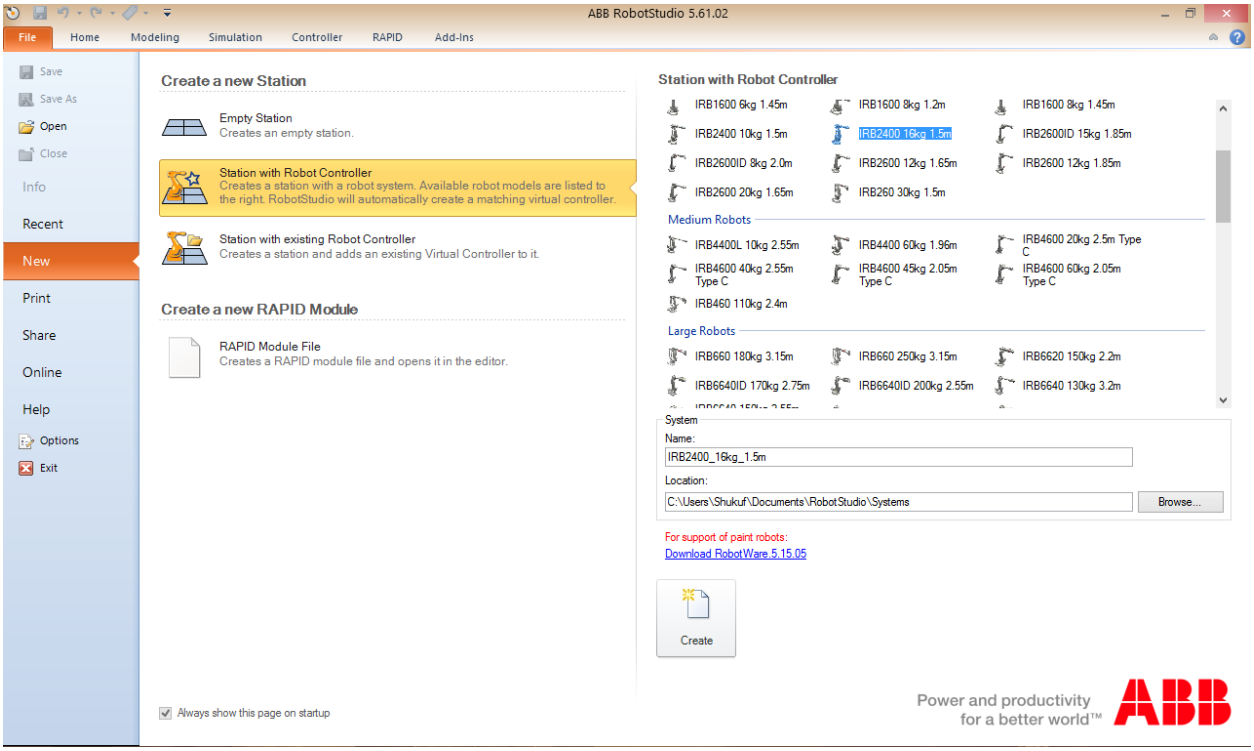


Step 9**:** In order to start the simulation, select the *Play* option from the *Simulation Control* of the top ribbon. The code will be executed and the robot moves from Target_10 to Target_20 on a linear path, Path_10.

Section 6: Simulating the robot using visual components

In this section, the basic steps of starting a simulation from 3D environment will be discussed.

Step 1: Under the *File* menu, create a new *Station with Robot Controller.*



Step 2: Select the robot model (IRB2400 16kg 1.5m in our example) and click on *Create* button.

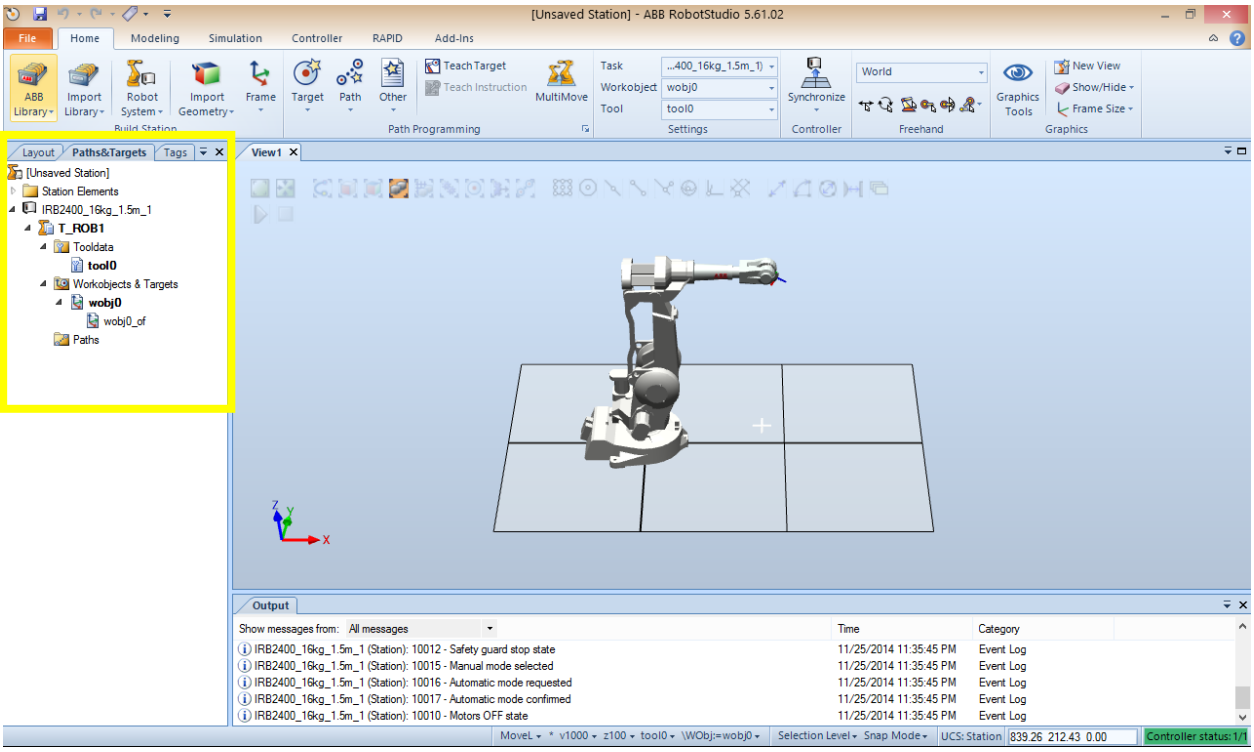
Wait for the virtual controller status, which is shown on the bottom right of the screen, to be *connected* (changes from red to green).

Note that some messages will appear in the output window regarding the status of the controller and the robot.

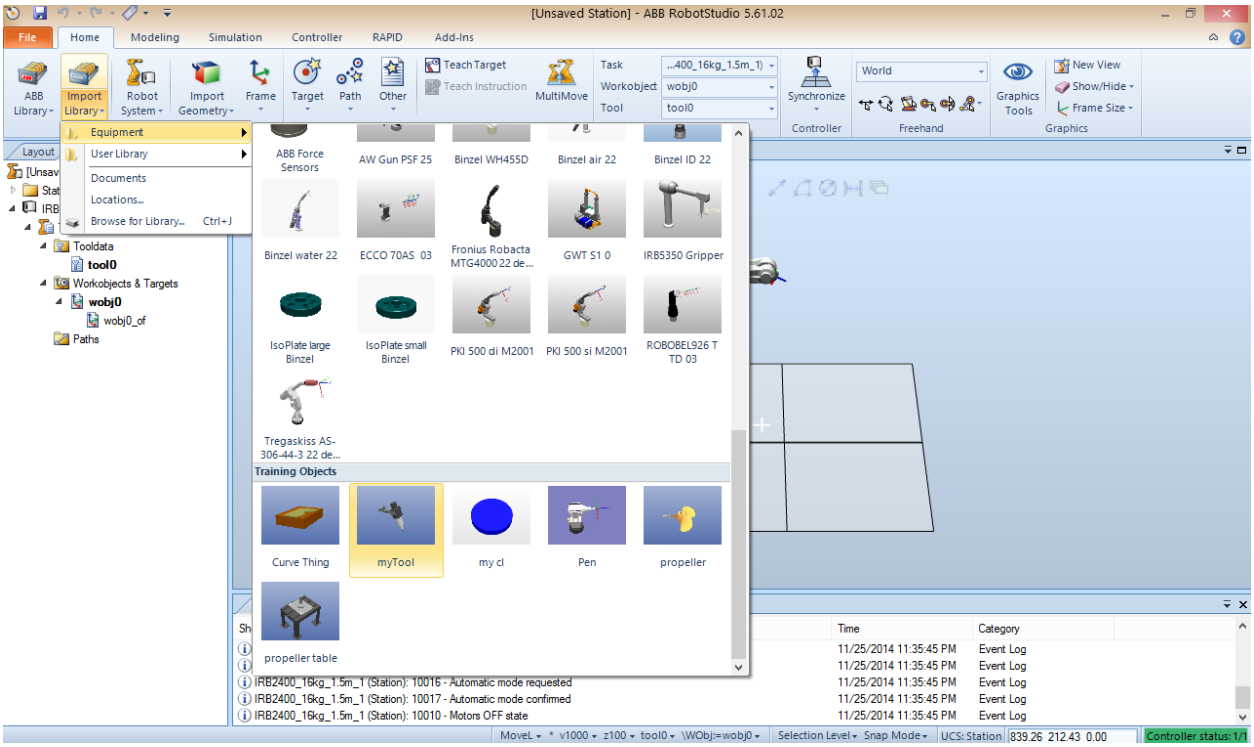
Step 3**:** After the robot and its predefined *work Object (Wobj0)* are appeared in the main window of the RobotStudio software, you can navigate through the work object by holding the *Ctrl* button

on the keyboard and *left click* on the mouse. In order to rotate around the station, you should hold the *Alt* and *Ctrl* button at the same time as *left click*.

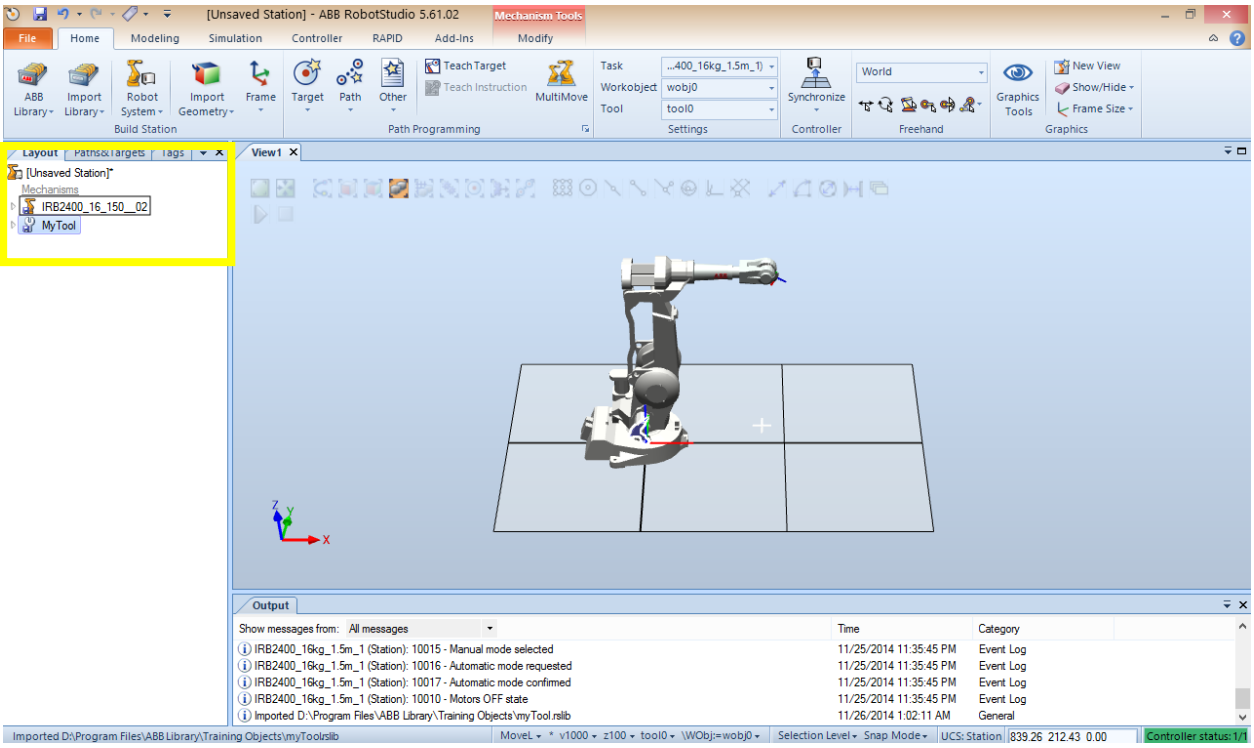
Step 4: Expand all the components under the robot *IRB2400_16Kg_15m_1* located in the P*aths&Target*s part of the right side bar of the screen to review the pre-defined system data such as *Wobj0* and *tool0*.



Step 5**:** It is possible to attach a tool to the end-effector of the robot. The tool could be imported from the *library*, for example a pre-defined tool named *myTool* in the *Training Objects*.

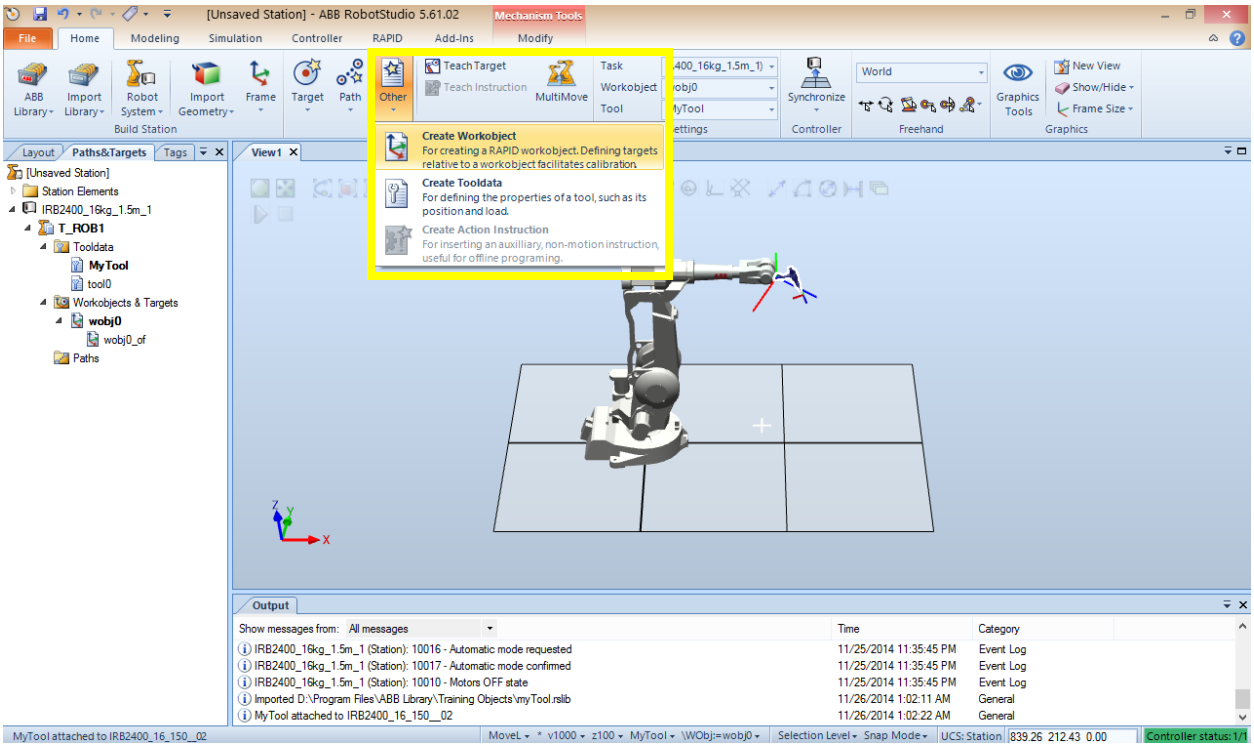


After choosing a tool, we have to attach it to the robot end-effector. To do so, under the *Layout* tab in the right side browser, drag the tool (*Mytool* in this example) on to the robot name (*IRB2400_16_150_02*).



Step 6**:** Define a new *Work Object* as a reference system, so the robot targets could be defined in relation to this reference system. In practice, this is a very important step for the calibration process between the virtual environment and the real environment with a real robot. Usually, the *origin* of a *workobject* is a point that can be easily defined, for example the corner of a room or table.

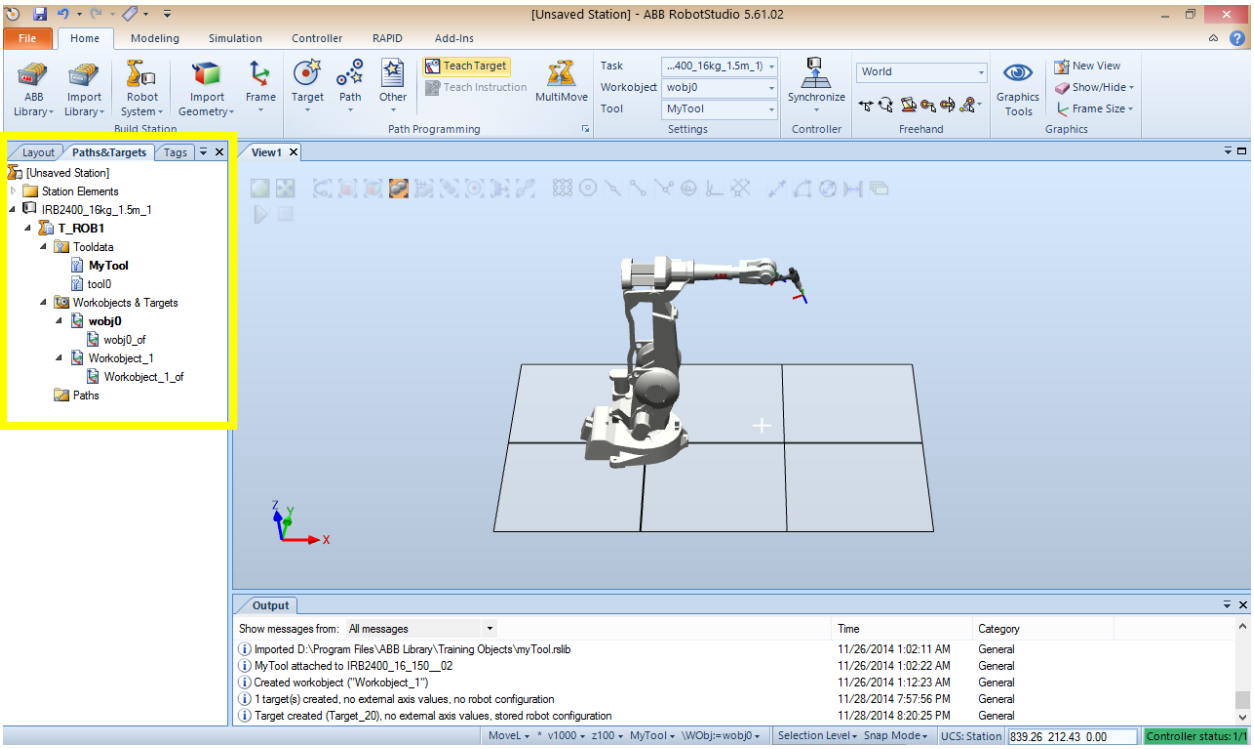
To create a *Work Object*, click on the *"other"* icon in the *Path Programming* section of the ribbon bar and select *Create Workobject.*



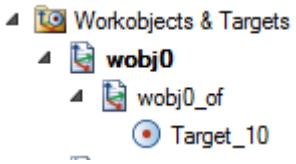
Give a new *Workobject* a name (*Workobject_1* in this example) and press *create*.

Step 7: It is now time to define *Target* points which will be the base for the robot *Paths*. The first *Target* that we have to define is the point where the robot starts its path from. It could be considered as a home position for the robot which has to return to this point as a first step of each iteration. In most of robot applications/programs we usually define a home position for the robot in relation to the base of the robot, in this case, *wobj0*. Make sure that the current *Workobj* is w*obj0*

under the *Settings* part of the ribbon. Then select *Teach Target* in the *Path Programming* part of the ribbon to create a target according to the current position of the robot.



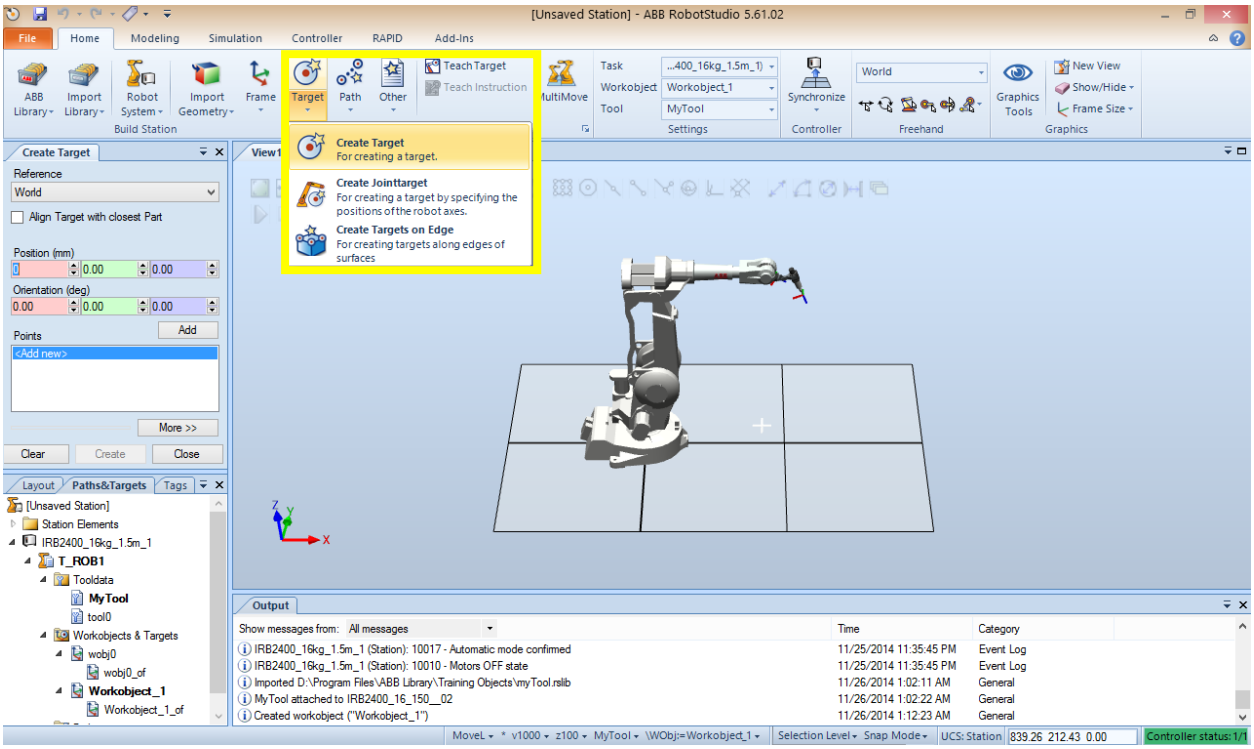
By expanding *wobj0* (in the *Paths&Targets* browser), we can see that a new target called *Target_10* has been added to the work object.



By selecting *Target_ 10* in the browser, its location will be *highlighted* in the view1 tab.

Step 8**:** At this point we want to define a set of targets which are related to *wobj1* in order to shape the rest of the *Path*. Defining targets in a different work object other than wobj0 (which is the robot work object) makes later modifications easier. To create other targets, first change the current *workobject in the Settings* part of the ribbon to *wobj1*, then create a new target by

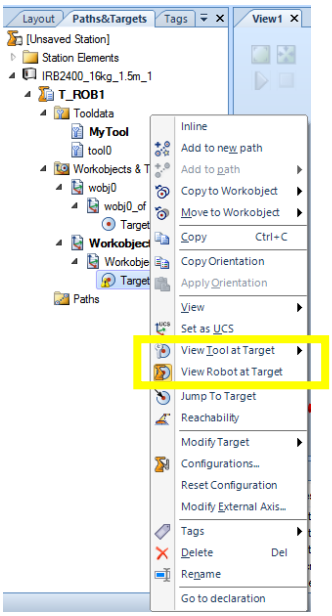
selecting *Target* icon from *Path Programming* part of the ribbon.



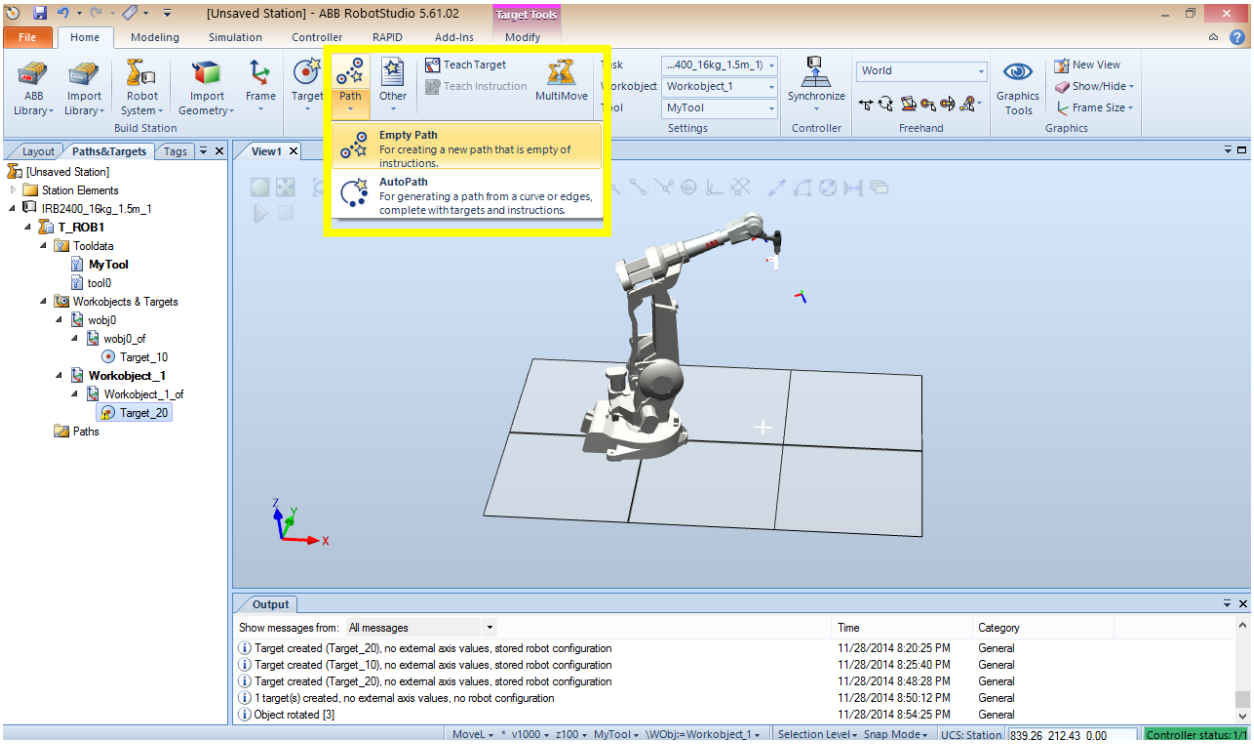
The *Create Target Panel* will show up on the right side of the screen, where you can choose the *Reference* for the coordination system (World in this example) and enter the *X, Y and Z* values as *Position* of the target and three angles for its *Orientation*. By hitting *Create* button, a new target (*Target*_20 in this example*)* can be seen under the *wobj1* component located in *Paths&Targets* browser.

Step 9: Now we are going to check if the robot could reach the defined target points (distance wise and orientation wise). In order to check the orientation of targets, right click on the target name (in *Paths&Targets* browser) and select *View Tool at Target*. If the orientation is not desired, you can change it by going to *Modify Target* in the same list.

For checking that the points are reachable by the robot, select *View Robot at Target*. If the target is out of reach, a warning would be shown in the *Output* window. As the orientation, the position of the target can be changed by using *Modify Target* in the same list



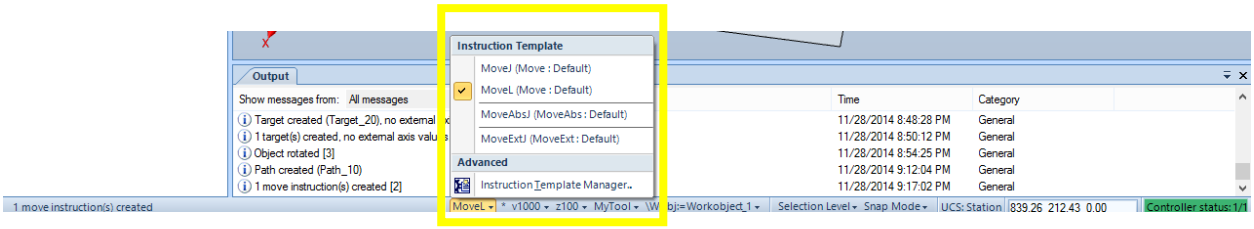
Step 10**:** In order to define the *Path* between targets and make them connected, select the *Path* icon in the *Path and Programming* part of the top ribbon and choose *Empty Path*.



A new Path (*Path_10* in this example) will be added to the *Paths* list, which can be expanded in *Paths&Targets* of the right side browser. We have to drag the target points in a desired order and

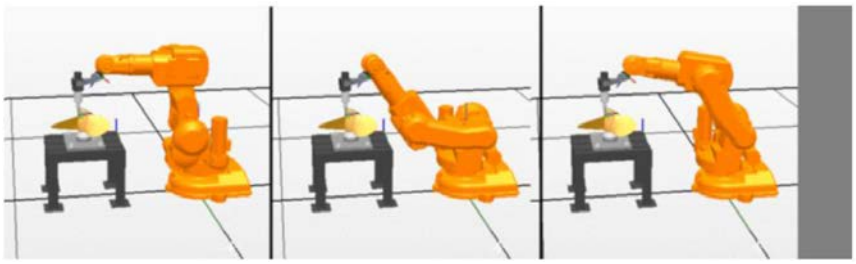
define the form of the movement between them, such as *MoveL* or *MoveJ*. In our example, the first point to drag into the path is *Target_10* (the home position of the robot).

Usually the default form of motion in the path is *Linear* movement, MoveL, and it can be changed by selecting the MoveL located on the bottom status bar.

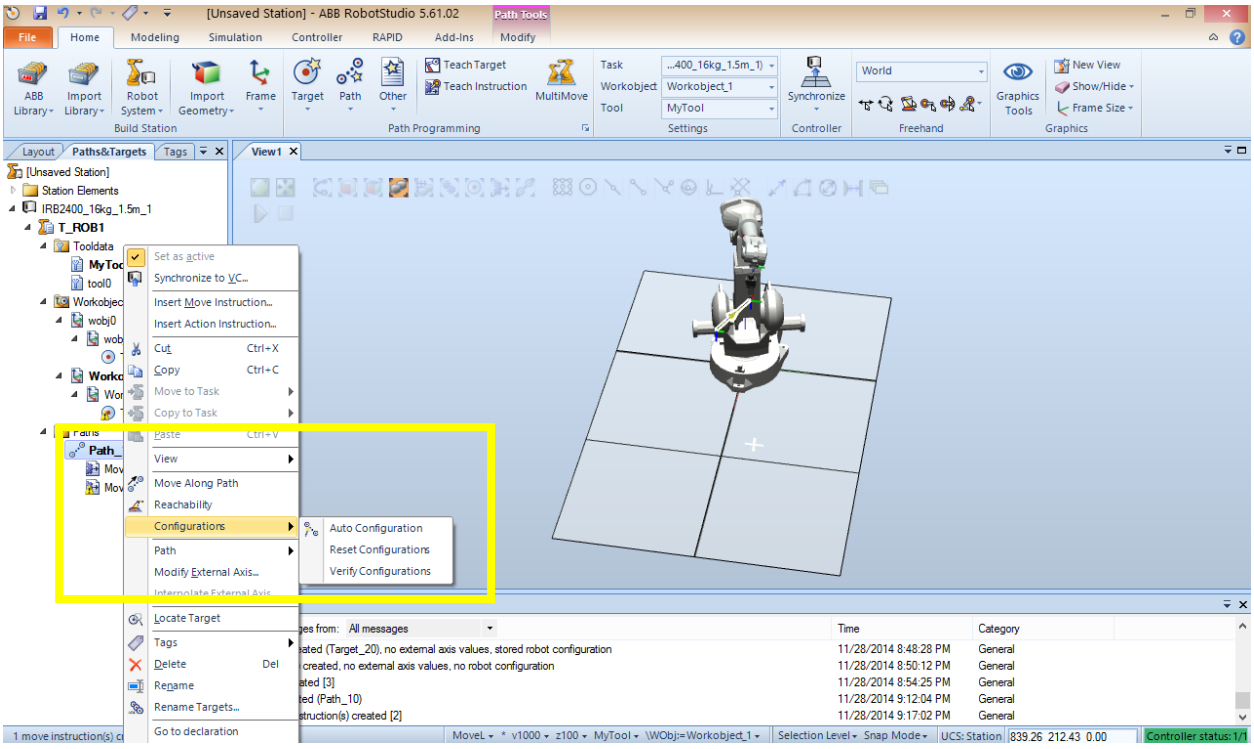


The targets have been connected, however, to eliminate the warnings we have to define the robot configuration for each one. The reason is, as you can see in the figure below, there are different ways (joints configuration) to put the robot end-effector (tool) in the same position and orientation.

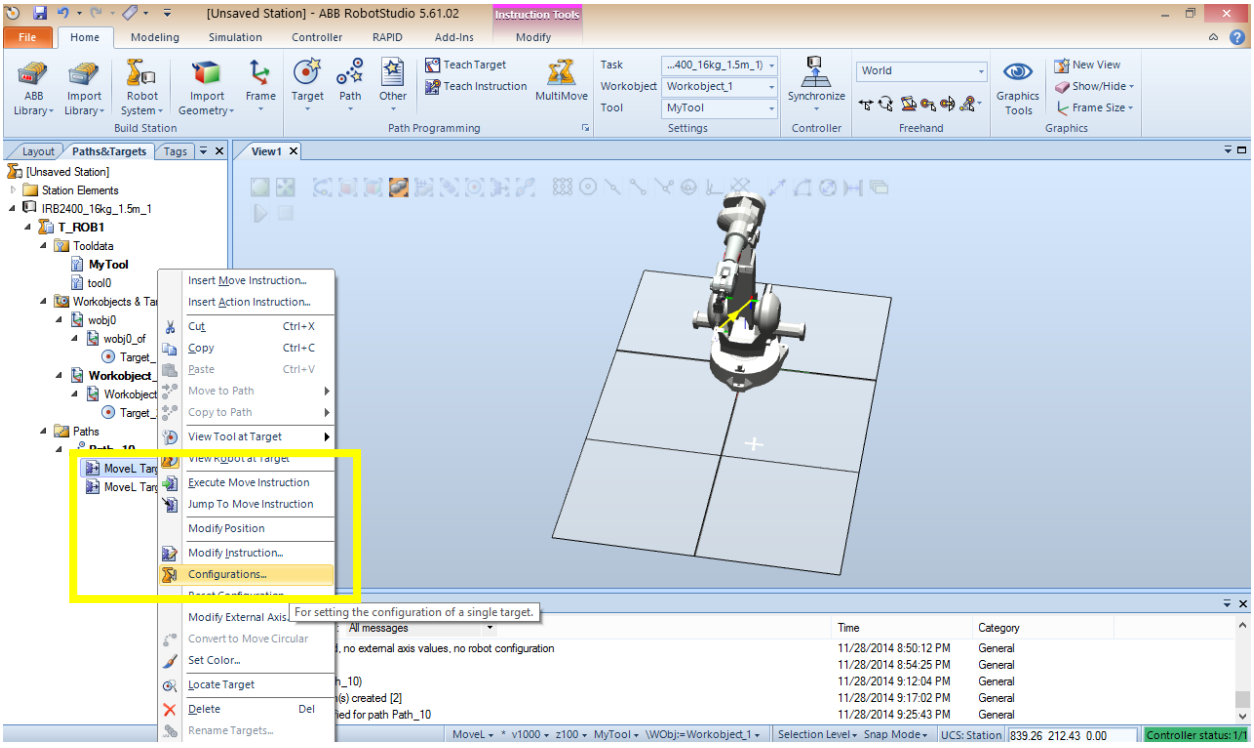
Targets are defined and stored as coordinates in a WorkObject coordinate system. When the controller calculates the position of the robot axes for reaching the target, it will often find more than one possible solution to configuring the robot axes.



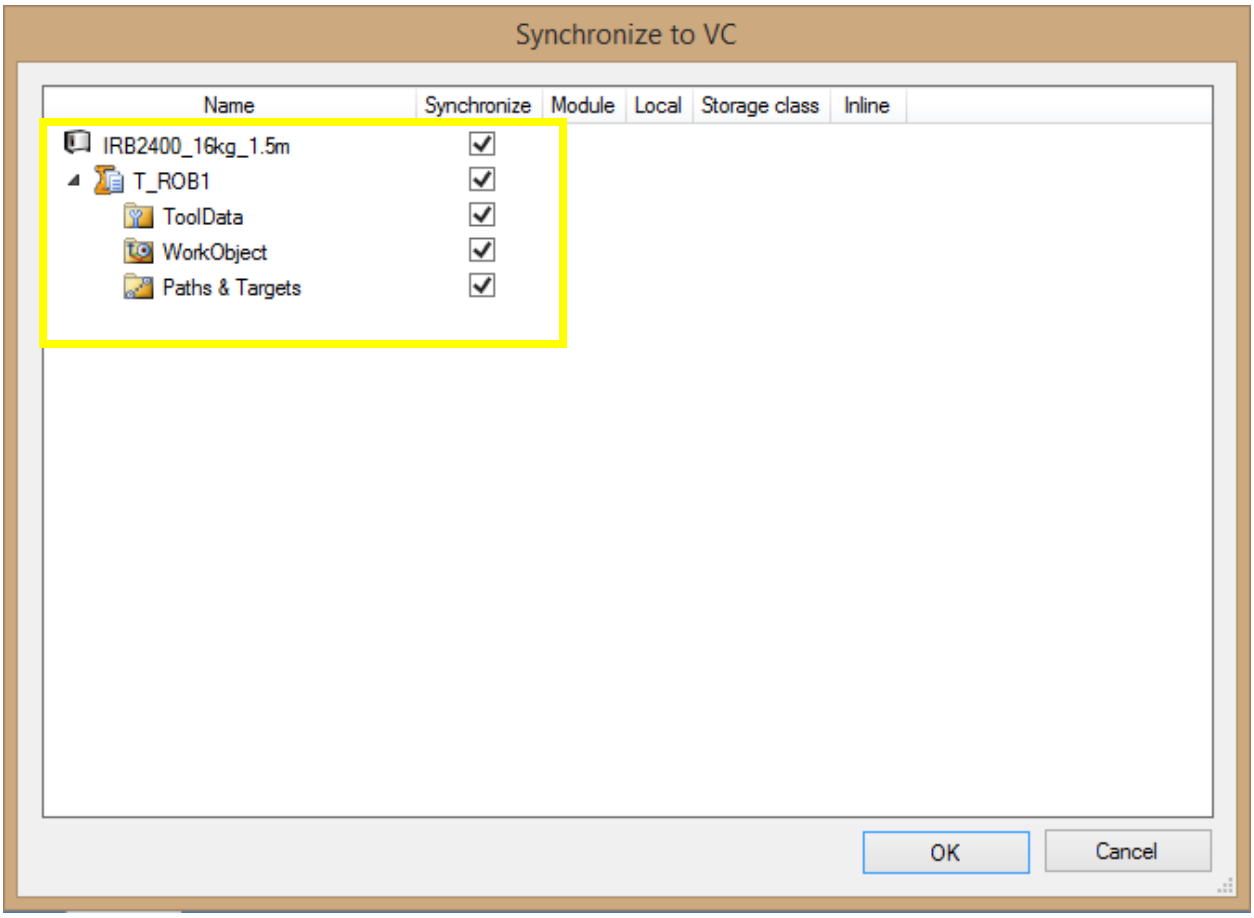
RobotStudio software is able to auto-configure the above mentioned joint configuration. To do so right click on the *Path* name in the right side browser and select *Auto Configuration* from configuration menu.



Another option is to validate and define the join configuration for each target. Just right click on each movement name under the path name in the right side browser and select *Configuration…*

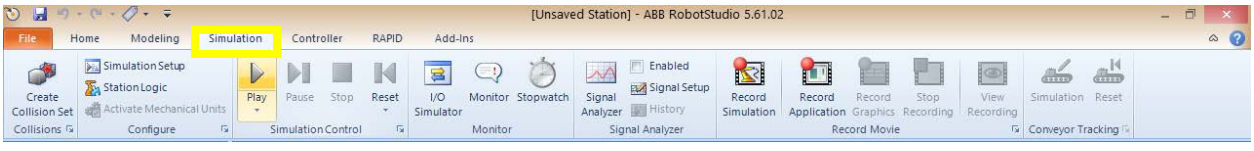


Step 11: It is time to simulate the robot program we just designed. First, synchronize the configuration with the robot controller (it is like to send all data to the robot controller). Press the *Synchronize* button in the top ribbon and choose the *Synchronize to VC…* option. Check all boxes in front of the objects and press OK.

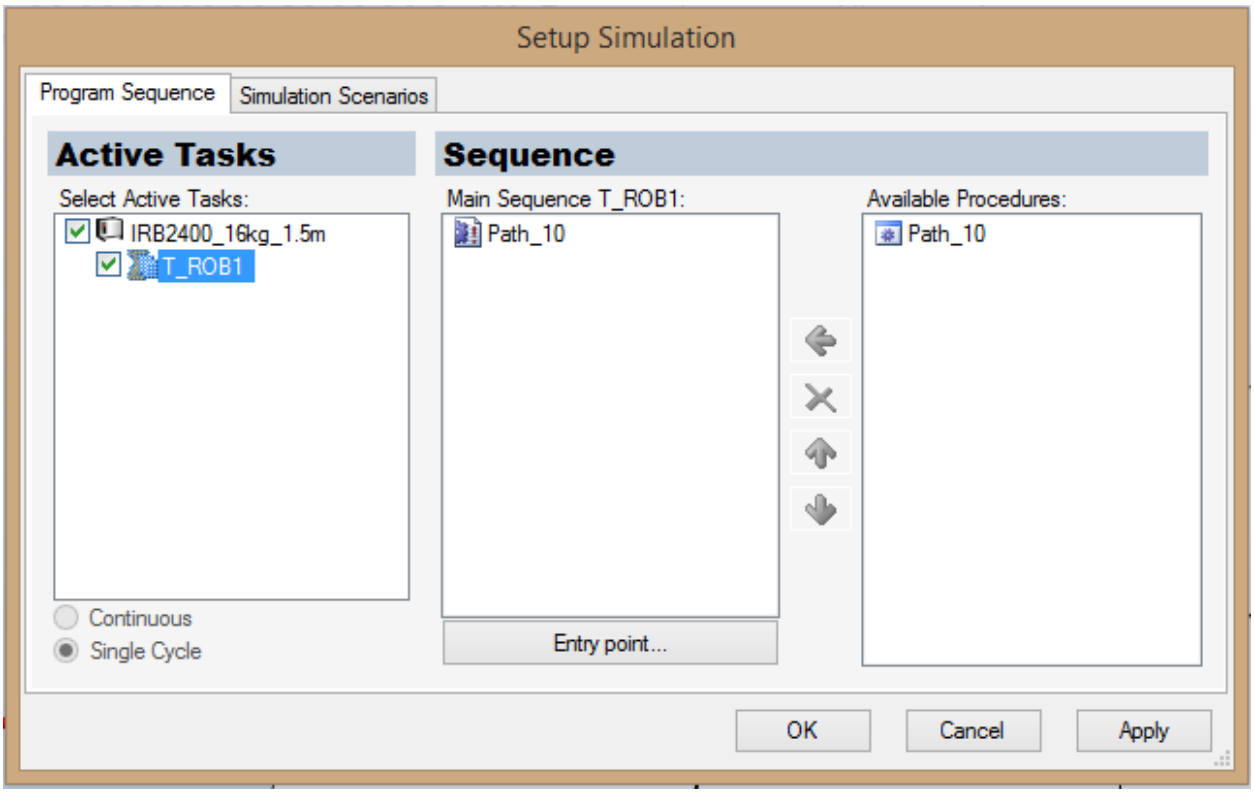


After synchronizing the station with the virtual controller, a *Synchronization to VC completed* message will appear in the output window.

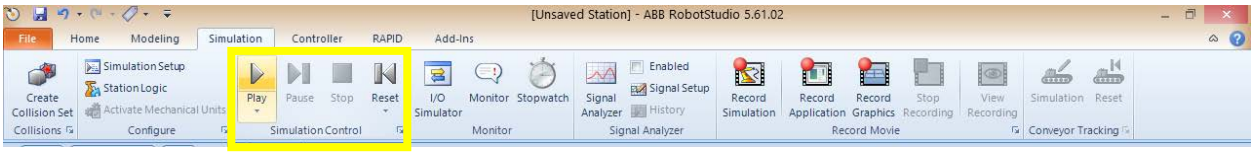
Step 12: In order to simulate our program on the virtual controller, go to the *Simulation* menu and select the *Simulation Setup* from the ribbon.



In the *Simulation Setup* window, click on *T_ROB1* active task (note that all the procedures except *main* should be listed in the sequence column). Press OK to apply the changes.



Step 13**:** In order to start the simulation, select the *Play* option from the *Simulation Control* of the top ribbon. The program will be executed and the robot moves from *Target_10* to *Target_20* on a linear path, *Path_10*.


[Unsaved Station] - ABB RobotStudio 5.61.02
File
Home
Modeling
Simulation
Controller
RAPID
Add-Ins
Create Collision Set
Collisions
Simulation Setup
Station Logic
Activate Mechanical Units
Configure
Play
Pause
Stop
Reset
Simulation Control
I/O Simulator
Monitor
Stopwatch
Monitor
Signal Analyzer
Enabled
Signal Setup
History
Signal Analyzer
Record Simulation
Record Application
Record Graphics
Stop Recording
View Recording
Record Movie
Simulation
Reset
Conveyor Tracking