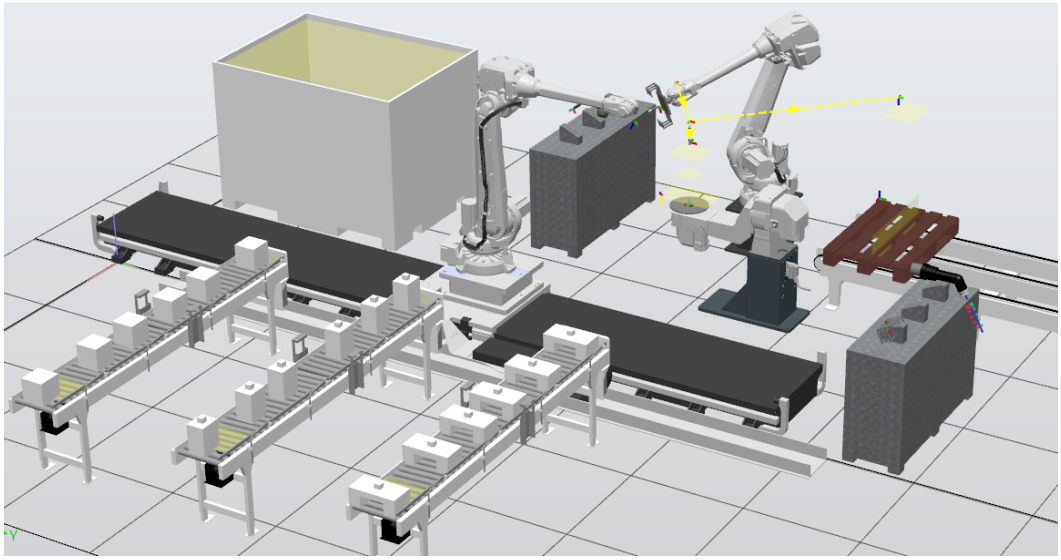
# Universidad Carlos III de Madrid

**Robotics Engineering (250)**

**Systems Engineering and Automation Department**


# INDUSTRIAL ROBOTICS PROJECT



## Design and Implementation of an Automated Assembly and Painting Cell in Robot Studio

## Project Report


Adrián González García -100523018
Claudia Cotobal Gómez – 100496721

# A. Minimum Objectives

## A.1 Manufacturing Process Description

Our manufacturing process focuses on assembling, welding, painting, and palletizing a Minecraft's inspired toy called "Creeper" which consists of three main components: "Patas" (legs), "Cuerpo" (body), and "Cabeza" (head). This process uses 2 robots in a coordinated way:

1. **Robot 1 (ROB1)** handles:
    - Picking components from the conveyors
    - Quality checking
    - Assembling parts on the work positioner
    - Painting the assembled toy
2. **Robot 2 (ROB2)** handles:
    - Welding the parts together
    - Picking the finished assembly
    - Palletizing the completed products

Some key functionalities of our manufacturing process are:

- Quality verification with random testing and rejection of defective parts
- Both robots change between tools (gripper/paint gun and weld gun/gripper)
- A rotary plate that facilitates the robots the welding and painting functions by rotating the assembled pieces
- Automated palletizing in a pattern of 4 positions
- User interaction with FlexPendant for initializing the program and selecting the phase wanted

The system operates as a continuous process with a proper synchronization between both robots using shared variables such as WorkingZone_Busy, soldar, paint and paletizing. To optimize the efficiency of the production, the system allows parallel operations and ensures a continuous flow of parts from the conveyors, also controlled by signals.

## A.2 Robot Justification

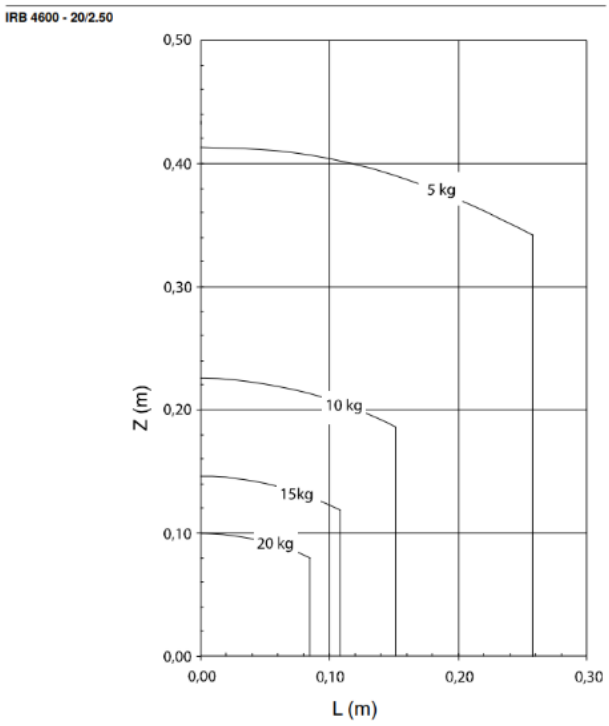
Both robots were selected considering workspace requirements, load capacity, and the number of degrees of freedom needed for the manufacturing process:

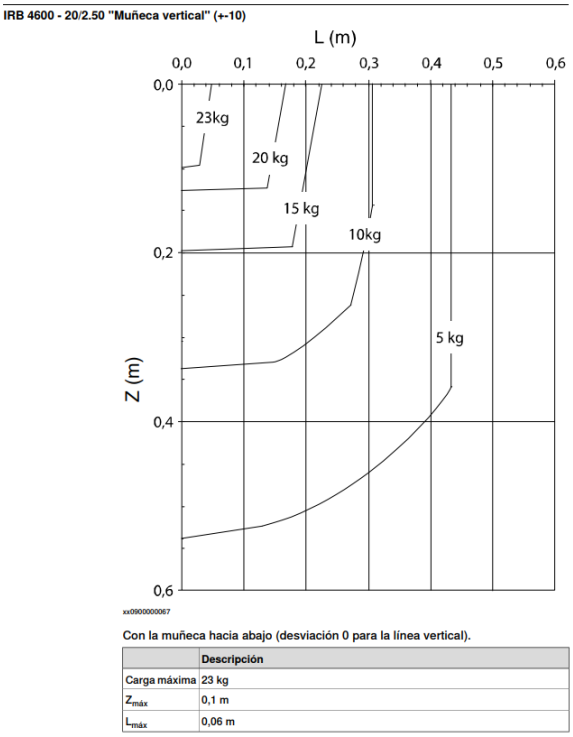
We selected the IRB 4600-20/2.50 model for both robots because of its good balance of reach, payload capacity, and precision. As shown in the load diagrams, this robot can handle our lightweight toy components (approximately 4 kg each) while maintaining high precision for welding and positioning operations.

**ROB1** is mounted on an IRBT4004_STD_250_0_6_04 track system, which allows the robot to move linearly along the cell. This configuration significantly extends the robot's workspace, enabling it to reach all the conveyors, work positioners, and the trash container without repositioning.

**ROB2** remains stationary but works with an RBP_A250_D1000_M2009_REV1_01 rotary positioner that rotates the workpiece to help the robot access it to weld it from different angles and later assists in the process of palletizing.

Neither robot is redundant since all 6 degrees of freedom are necessary to achieve the required positions and orientations throug the process.




## A.3 Use of Multiple TCPs

Each robot in our system uses multiple tool center points (TCPs):

**ROB1 Tools:**

- Gripper_tool: A gripper used for picking and placing components
- PaintGun: Used for painting the assembled product

**ROB2 Tools:**

- Weldgun: Used for welding operations
- Gripper_tool: Used for handling the completed product during palletizing

Both robots have tool change systems with well-defined tool properties. For example, the Gripper_tool is defined with appropriate weight, center of gravity, and orientation parameters:

```
PERS tooldata Gripper_tool:=[TRUE,[[0,0,100],[1,0,0,0]],[5,[0,0,40],[1,0,0,0],0,0,0]];
```

Similarly, the PaintGun tool has its own specific parameters:

```
PERS                                                                    tooldata
PaintGun:=[TRUE,[[290,0,570],[0.866025,0,0.5,0]],[5.5,[30,0,225],[1,0,0,0],0,0,0]];
```

These TCPs allow precise positioning of the appropriate tool for each specific task in the manufacturing process.

## A.4 Work Object Definitions

Our process uses multiple work objects for both robots:

**ROB1 Work Objects:**

- Conveyor_Cabezas: Frame for picking head
- Conveyor_Cuerpos: Frame for picking body
- Conveyor_Patas: Frame for picking leg
- PortaHerramientas: Tool rack frame
- Trash_Container: Frame for discarding defective parts
- Work_Table: Frame for assembly operations
- Safe_Track: Reference frame for the external track

**ROB2 Work Objects:**

- PortaHerramientas_ROB2: Tool rack frame
- Wobj_WeldBody: Frame for welding body to legs
- Wobj_WeldHead: Frame for welding head to body
- Wobj_PickPiece: Frame for picking the completed assembly
- Wobj_Palet: Frame for palletizing operations

These work objects create reference points that make programming easier and help the robots work accurately in different parts of the manufacturing area.

## A.5 Point Capture and Trajectory Transformations

All robot trajectories in our project are built using transformations of captured points. We use both offset operations and direct modification of robtarget components to create complex movements.

For example, we define tool positions by applying transformations to base points:

```
Pos_Griper_10 := T_Pos_PH_10;
Pos_Griper_10.trans.x := T_Pos_PH_10.trans.x + 415;
Pos_Griper_10.trans.y := T_Pos_PH_10.trans.y + 195;
Pos_Griper_10.trans.z := T_Pos_PH_10.trans.z - 41.34;
Pos_Griper_10.rot := [0.2588, -0.9659, 0, 0];
```

We frequently use the Offs() function to create offset positions from base points:

```
MoveL Offs(Target, 0, 0, -300), v300, fine, tool\WObj:=frame;
```

Additionally, we create new points by directly modifying component coordinates, as seen in the definition of painting positions:

```
Paint_Pos1 := Paint_Approach;
Paint_Pos1.trans := [100,100,550];
Paint_Pos2 := Paint_Pos1;
Paint_Pos2.trans.x := Paint_Pos2.trans.x + 50;
Paint_Pos2.trans.z := Paint_Pos2.trans.z - 200;
```

These ways of changing points make our program more flexible and let us adjust robot movements easily without having to teach every single position manually.

## A.6 Use of Approaching Points

Approaching points are consistently used throughout our process to ensure safe tool positioning before engaging in precise operations:

**Tool change approaching points:**

```
MoveJ
T_Pos_PH_10,vmax,z10,tool0\\WObj:=PortaHerramientas;
```

**Part pickup approaching points:**

```
MoveJ Target, vmax, z10, tool\\WObj:=frame;
```

**Welding approaching points:**

```
MoveJ       Offs(PC_30,       -500,       200,       900),       v1000,       z0,
Weldgun\\WObj:=Wobj_WeldBody;
```

**Palletizing approaching points:**

```
MoveJ        Offs(Target,        0,        0,        150),        v500,        z0,
Gripper_tool\\WObj:=Wobj_Palet;
```

These approaching points keep the robot at a safe distance before it does precise work. This helps prevent crashes and makes the robot's movements smoother when it moves from one task to another.

## A.7 Appropriate Movement Instructions

Our program uses appropriate movement instructions for different situations:

**MoveJ:** Used for joint movement during positioning and approaching, where the exact path is not critical:

```
MoveJ
Aproach_PH_ROB_2,vmax,z10,tool0\\WObj:=PortaHerramientas_ROB2;
```

**MoveL:** For linear movement during precision operations where a straight line path is required:

```
MoveL CC_10, v100, fine, Weldgun\\WObj:=Wobj_WeldHead;
```

**MoveAbsJ:** For absolute joint positions, used for safe positions and initialization:

```
MoveAbsJ JT_Safe_Track_10, v500, z200, tool0;
```

Speed and zone parameters are appropriately selected for each movement:

- Speeds range from v100 (slow/precise) to vmax (rapid positioning)
- Zone parameters (z0, z5, z10, z50) control path precision
- The fine setting is used for movements requiring exact positioning (e.g., tool pickup/drop, welding)

These different movement commands help the robot move in the best way for each job it needs to do in the manufacturing process.

## A.8 Non-linear Program Flow: Quality check

Our manufacturing process doesn't follow just one path. Instead, it can take different routes based on part quality.

When the robot picks up a part, it checks if the part is valid or defective with a quality control camera located in tool0. It performs the classification by creating a random number and dividing it by 300. This gives us a number between 0 and about 109, which works like a quality score following a percentage form 0 to 100.

If this quality score is less than 32 (like getting a low grade), the part is defective, and the robot throws it away in the trash bin. The robot then goes back to get another part and checks again. If the value is 32 or higher, the part passes the quality check and the robot places it on the worktable, moving on to the next assembly phase.

This system of checking quality creates different possible paths based on quality, making our process more like a real world manufacturing system, where not every part is perfect.

## A.9 Parameter-passing Routines

We created several parameter-passing functions that can accept different inputs. This helps our code stay organized and lets us use the same functions in different situations without having to write the same code repeatedly.

**pick_part:** Is in charge of picking up the parts with parameters for target position, tool, and work object frame:

```
PROC pick_part(robtarget Target,
PERS tooldata tool, PERS wobjdata frame)
```

**drop_part:** Handles the part placement with similar parameters:

```
PROC                          drop_part(robtarget                          Target,
PERS tooldata tool, PERS wobjdata frame)
```

**change_tool:** Manages the tool changes with parameters for the current and new tool positions:

```
PROC change_tool(robtarget actual, robtarget nueva)
```

**Paint_Side:** This controls the painting operations with offset parameters:

```
PROC Paint_Side(num off, num off_2)
```

These functions make our program more efficient by allowing the same procedures to be used with different parameters throughout the program. For example, the same "pick_part" routine is used for picking up the pieces (head, body, or legs) using different tools and from different workobjects, instead of needing separate functions for each situation.

### A.10 Robotic Implementation Justification

Using robots for our manufacturing process makes sense for several practical reasons.

The assembly, welding, painting, and palletizing tasks require repeating the same precise movements repeatedly and uninterrupted, which is the main reason why making automated system for this process is much more effective and almost necessary.

Secondly, because of the higher flexibility and workspace utilization. The robots can handle different quality parts and adapt to different process phases, making it more flexible than fixed automation. Moreover, the robots can work across a large area, reaching to many different workstations that would otherwise require separate fixed machines.

Third, some tasks like welding operations need high precision and consistency that would be difficult for human workers to maintain throughout a shift. Robots can also handle tool changes seamlessly and work without stopping, increasing productivity.

Finally, the quality control system automatically throws away bad parts, ensuring only valid pieces to continue the process.

## B. Extensions

### B.1 Multi-Robot Cell Implementation

Our project uses two robots (ROB1 and ROB2) working in a coordinated cell. Each robot fulfills all the required specifications for tools, work objects, and trajectory planning.

The robots communicate through shared variables to synchronize their operations and avoid workspace conflicts.

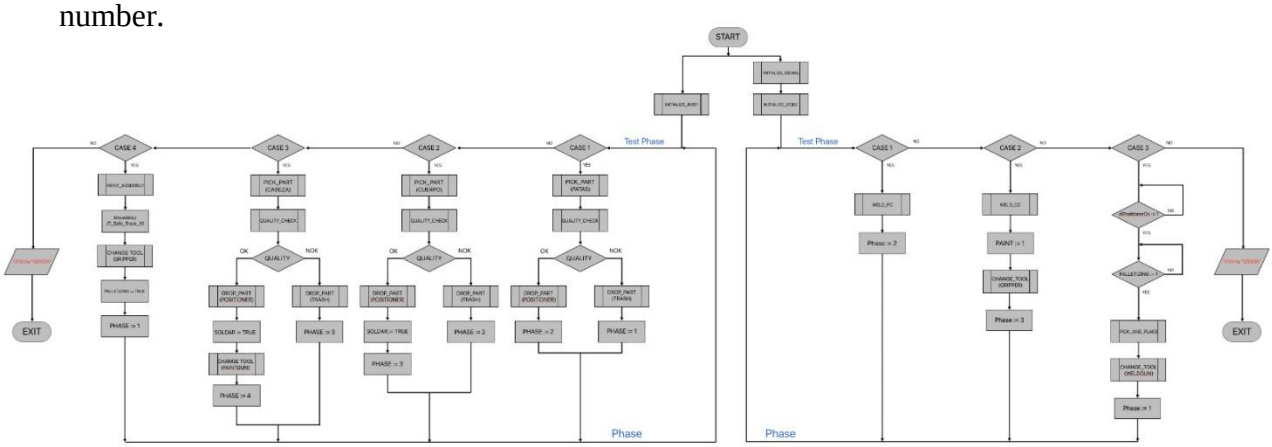
### B.2 Modular Program Structure

Our code is organized into separate modules for better organization:

- Main program modules for each robot
- Function modules containing specialized operations
- Calibration data modules for tool and work object definitions
- Shared variables module for inter-robot communication

We created this flowchart that visually represents the program's execution logic, showing how the robots make decisions and coordinate their actions.

Two important labels appear in these diagrams: "Test Phase" shows where the program evaluates the current phase value using a TEST-CASE structure (similar to switch-case), while "Phase" at the bottom refers to the variable that stores the current manufacturing phase number.



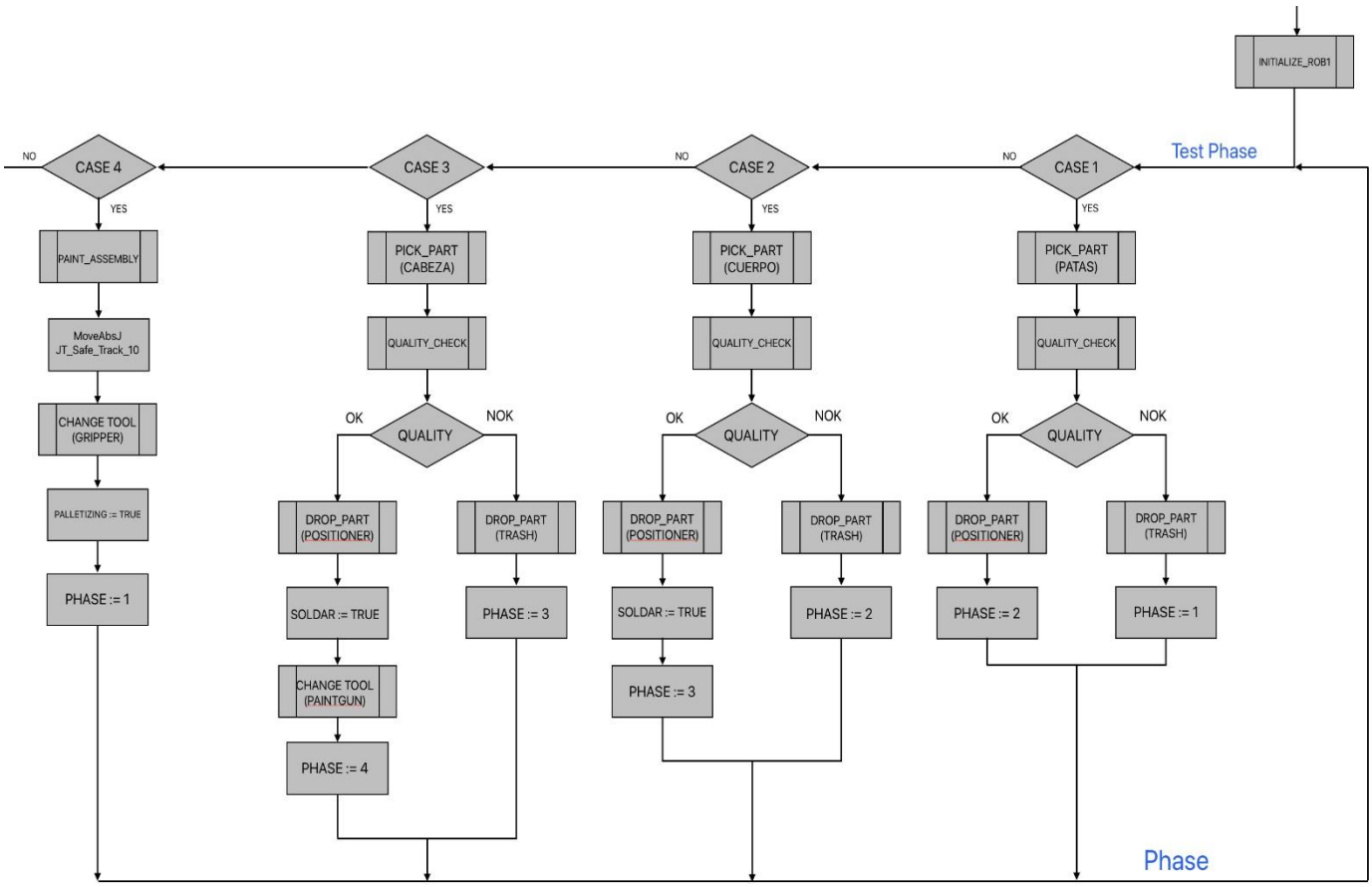
For Robot 1, the flowchart shows the complete process flow from initialization through four distinct phases:

- Phase 1: Assembly of "Patas" (legs)
- Phase 2: Assembly of "Cuerpo" (body)
- Phase 3: Assembly of "Cabeza" (head)
- Phase 4: Painting the assembled toy

The first three phases follow a similar pattern: the robot picks up a specific component from its conveyor, performs a quality check (which returns a Boolean to determine the next step), and then either places the component on the worktable (if it passes quality) or discards it to the trash container (if defective). This creates branches in the flowchart that show how defective parts are sent to the trash container while good parts continue through the assembly process.

The fourth phase is different since it involves changing to the paint gun tool and executing the painting sequence.

Throughout these operations, the "MoveAbsJ" command appears when the robot needs to move to an absolute joint position, particularly when changing tools or moving along the track to reach different workstations safely.



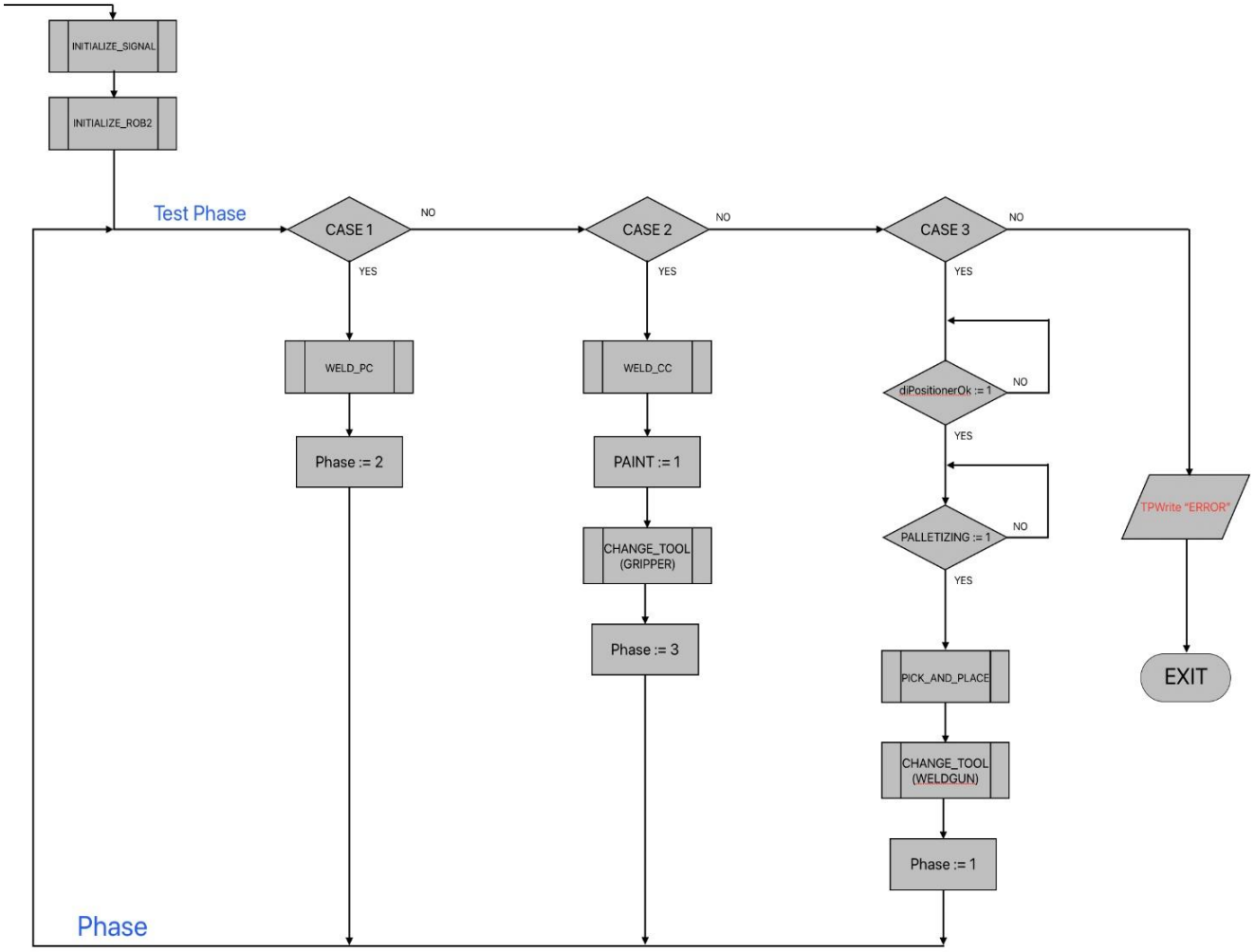
For Robot 2, the flowchart shows the welding, tool changing, and palletizing sequences that operate in coordination with Robot 1.

The robot follows three main phases controlled by TEST-CASE structures: welding the legs to body, welding the head to body, and palletizing.

Throughout these operations, Robot 2 checks "diPositionerOK" to make sure the work holder is turned to the right position before welding or handling parts.

After Robot 1 finishes painting, it sets the "PALLETIZING=1" flag, which tells Robot 2 to start the "PICK_AND_PLACE" operation. This operation involves picking up the finished toy from the work holder and putting it in the right spot on the pallet.

Both robots share information through common variables to ensure that they never interface into each other's workspace to keep the production running correctly.



The controller signal list shows all the digital inputs (DI) and digital outputs (DO) that enable communication between the robots and peripheral devices in the cell. These signals are essential for synchronization and control, allowing the robots to, for example:

- detect the presence of a part (diCabezaReady, diCuerpoReady, diPatasReady)
- confirm the attachment of a tool (diObjectAttached_Gripper, diToolAttached_Rob20)
- control the work positioner (doPos10, doPos20, doRotatePlate0)
- manage the assembly process (doAttachCabeza, doAttachCuerpo).
- initialize the conveyor system (doInitiallize_Conveyor_P0) to start the flow of parts into the manufacturing cell

The communication between different program components is managed through shared variables like WorkingZone_Busy, soldar, paint, and paletizing.

These variables coordinate the activities of both robots to prevent them from colliding and ensuring a correct sequence of operations.

This way of organizing our program with well defined communication channels makes it easier to understand, maintain, and modify if changes are needed in the future.

| | Name | Type | Value | Min Value | Max Value | Simulated | Network | Device | Device Mapping | Category | Label |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | diCabezaReady | DI | 1 | 0 | 1 | Yes | <none> | <none> | | | |
| 1 | diCuerpoReady | DI | 1 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | diObjectAttached_Gripper | DI | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | diObjectAttached_Rob20 | DI | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | diPaletInPos0 | DI | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 1 | diPatasReady | DI | 1 | 0 | 1 | Yes | <none> | <none> | | | |
| 1 | diPosExecuted_Gripper | DI | 1 | 0 | 1 | Yes | <none> | <none> | | | |
| 1 | diPosExecuted_Gripper_Rob20 | DI | 1 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | diPositionerOK0 | DI | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 1 | diToolAttached_Rob20 | DI | 1 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachCabeza | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachCuerpo | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachObject_Gripper | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachObject_Rob20 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachParent0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachPatas | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachTool_Rob10 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doAttachTool_Rob20 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doCabezaPicked | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doChangeParent0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doCuerpoPicked | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doDettachBody | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doDettachTool_Rob10 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doDettachTool_Rob20 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doInitiallize_Conveyor_P0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPaletIn0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPaletOut0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPatasPicked | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPos10 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPos20 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPos30 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPos40 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPosCabeza_Gripper | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPosCuerpo_Gripper | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPosCuerpo_Rob20 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPosHome0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPosHome_Gripper | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doPosHome_Rob20 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doRotatePlate0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doSetParent0 | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |
| 0 | doSinkTrash | DO | 0 | 0 | 1 | Yes | <none> | <none> | | | |

### B.3 Function Implementation

We've implemented several FUNC routines to help keep our code organized and let the program make choices based on the answers they give back.

- **initialize_ROB1** returns the initial phase:

```
FUNC num initialize_ROB1()
```

- **SeleccionarHerramienta_ROB1** returns the selected tool:

```
FUNC num SeleccionarHerramienta_ROB1()
```

- **Quality_Check** performs the quality check and returns a boolean:

```
FUNC bool Quality_Check()
```

### B.4 Advanced RAPID Instructions

Our program uses several advanced RAPID instructions:

- **WaitDI** for waiting on digital inputs
- **PulseDO** for triggering digital outputs
- **WaitUntil** for synchronization based on shared variables
- **TPReadNum** for user interaction through the Flex Pendant
- **Rand()** for randomized quality check

### B.5 Inter-Robot Communication

We've implemented a communication system using shared variables:

```
PERS bool soldar;
PERS bool WorkingZone_Busy;
PERS bool paint;
PERS bool paletizing;
PERS bool ROB2_Ready;
PERS bool ROB1_Ready;
```

These variables help the robots work together, coordinate activities, signal the end of task performed, and avoid getting into each other's workspace. For example:

- ROB1 sets soldar := TRUE when parts are ready for welding
- ROB2 checks this flag (WaitUntil soldar = TRUE) before starting the welding process
- ROB2 sets soldar := FALSE when welding begins

### B.6 Process Optimization

Our program implements several optimization methods using specialized variable types and advanced RAPID instructions.

We've defined tooldata variables with precise parameters for both robots, including center of gravity, weight, and orientation information:

```
PERS tooldata Gripper_tool:=[TRUE,[[0,0,100],[1,0,0,0]],[5,[0,0,40],[1,0,0,0],0,0,0]];

PERS tooldata PaintGun:=[TRUE,[[290,0,570],[0.866025,0,0.5,0]],[5.5,[30,0,225],[1,0,0,0],0,0,0]];
```

Similarly, we've defined multiple wobjdata variables as reference frames for different operations:

```
TASK          PERS          wobjdata          Conveyor_Cabezas:=[FALSE,TRUE,"",[[1945,-
1060,241],[0,0,0,1]],[[0,0,0],[1,0,0,0]]];

TASK                                    PERS                                    wobjdata
Work_Table:=[FALSE,TRUE,"",[[4279.5,1315,275.5],[1,0,0,0]],[[0,0,0],[1,0,0,0]]];
```

Our code also has advanced RAPID instructions to optimize performance and reliability:

- Configuration control management with ConfJ\Off and ConfL\off for complex movements
- Synchronization with external equipment using WaitDI instructions
- Triggering momentary signals with PulseDO for efficient communication
- Coordinating robot activities through WaitUntil with shared variables
- Implementing quality simulation using the Rand() function

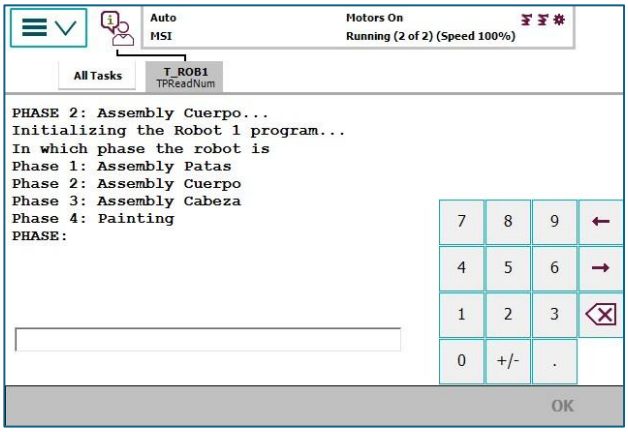
## B.7 User Interface Integration

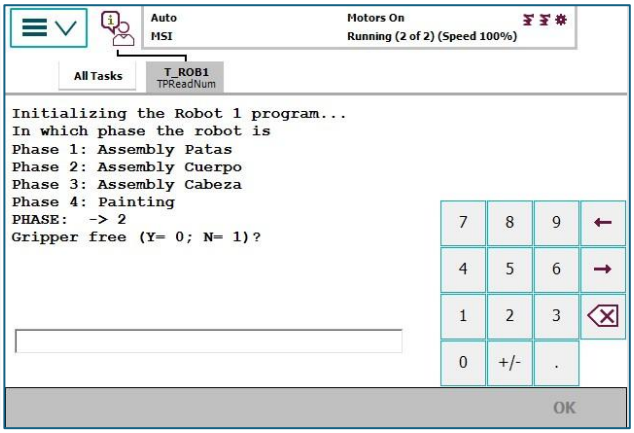
The program includes user interaction through the FlexPendant:

```
TPWrite "In which phase the robot is";
TPWrite "Phase 1: Assembly Patas";
TPWrite "Phase 2: Assembly Cuerpo";
TPWrite "Phase 3: Assembly Cabeza";
TPWrite "Phase 4: Painting";
TPReadNum Phase, "PHASE: ";
```

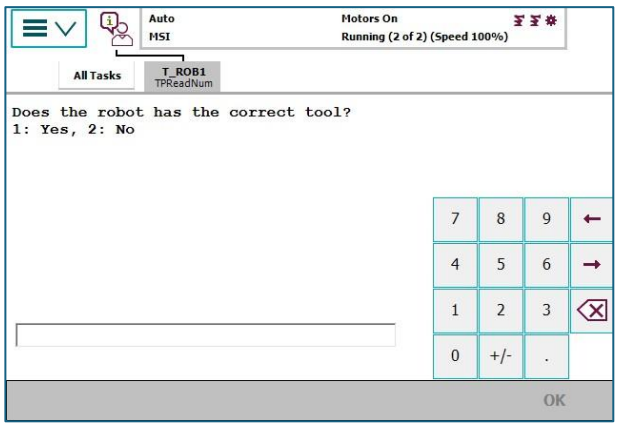
This allows operators to initialize the system and select the starting phase.

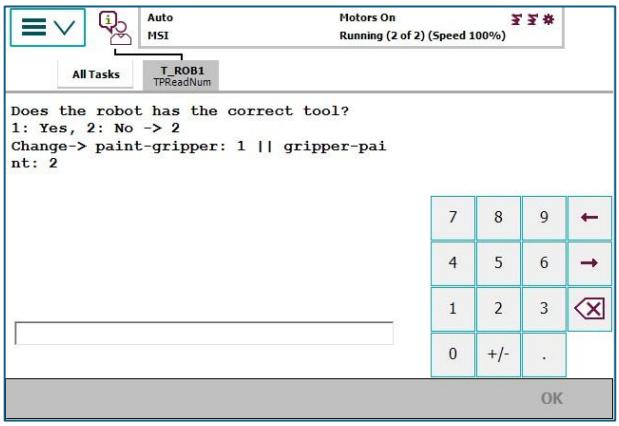
Selecting which phase to begin with and confirming whether the gripper is free or holding a part



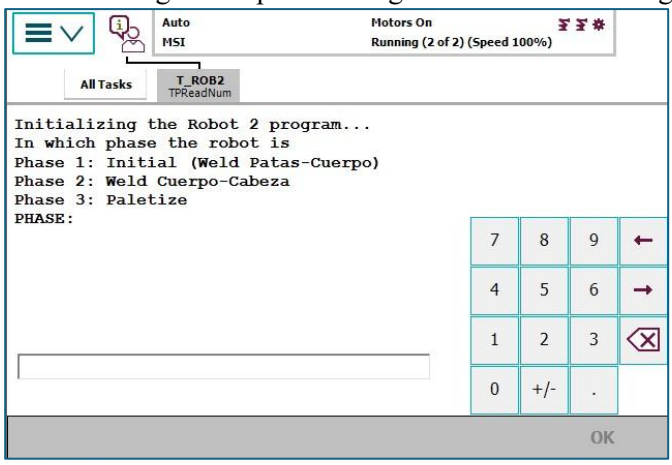

Checking if Robot 1 has the correct tool attached, offering the option to change tools if necessary

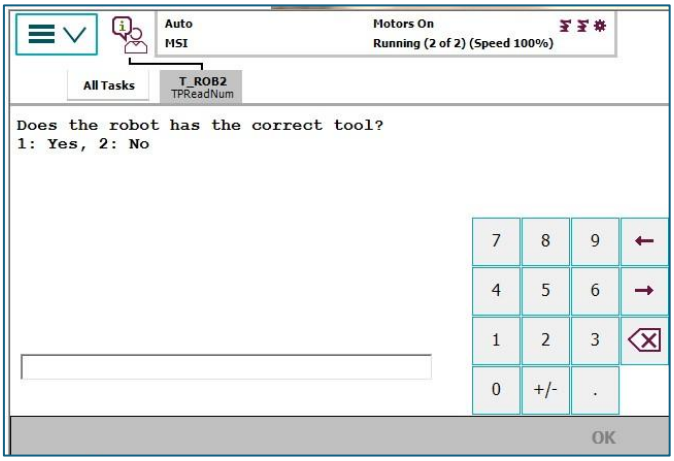


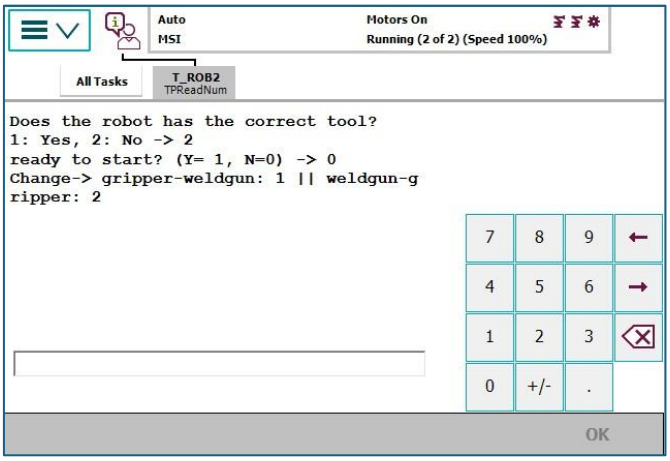
Similar sequence for robot 2:

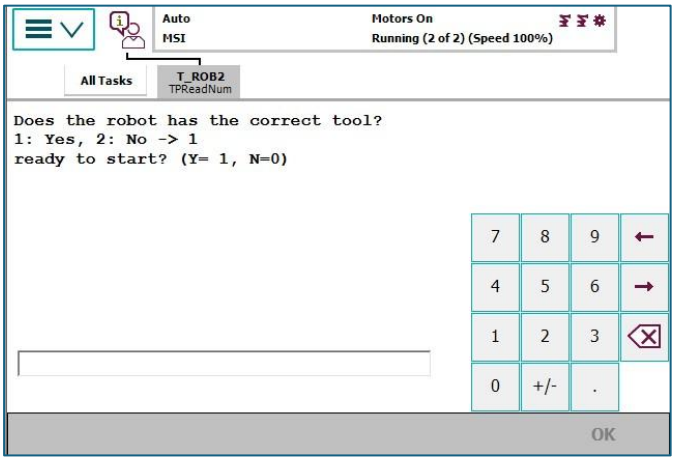
Selecting which phase to begin with and confirming whether the gripper is free or holding a par



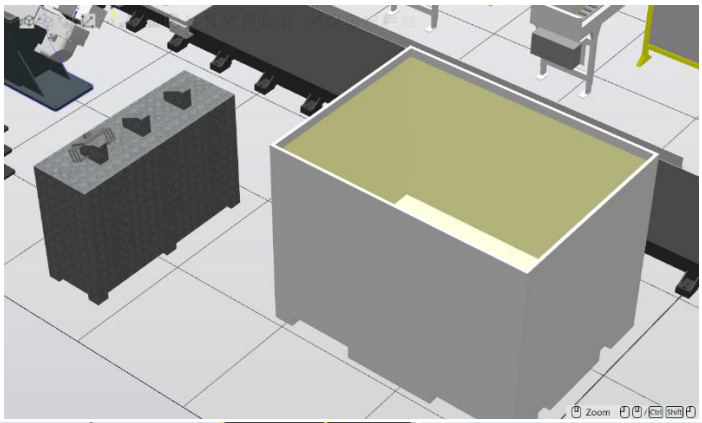

Checking if Robot 1 has the correct tool attached, offering the option to change tools if necessary

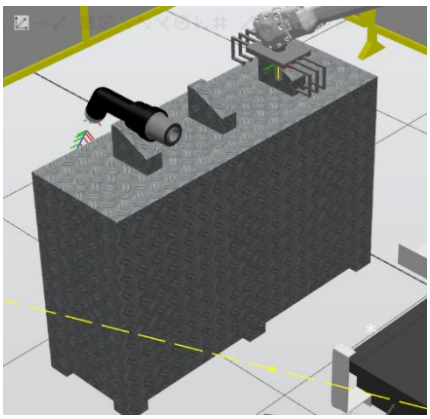


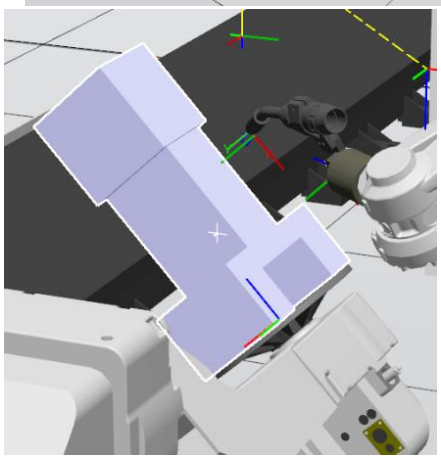



## B.8 Custom Components Created with Solid Edge

We designed several custom components using Solid Edge for our manufacturing cell, such as the toy parts (heads, bodies, legs), a trash container for rejected parts, and tool holders for both robots.

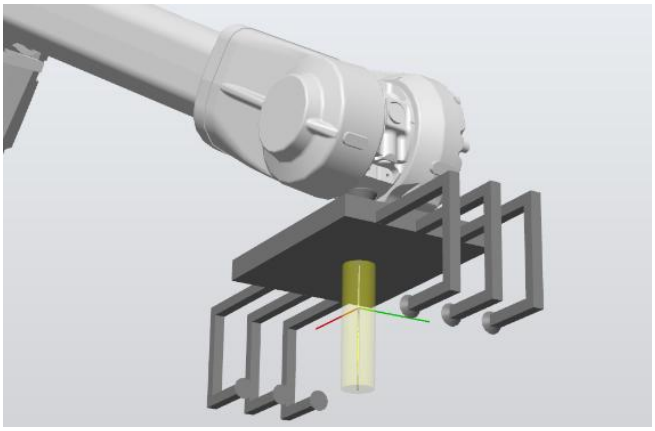






## B.9 Smart Component Gripper

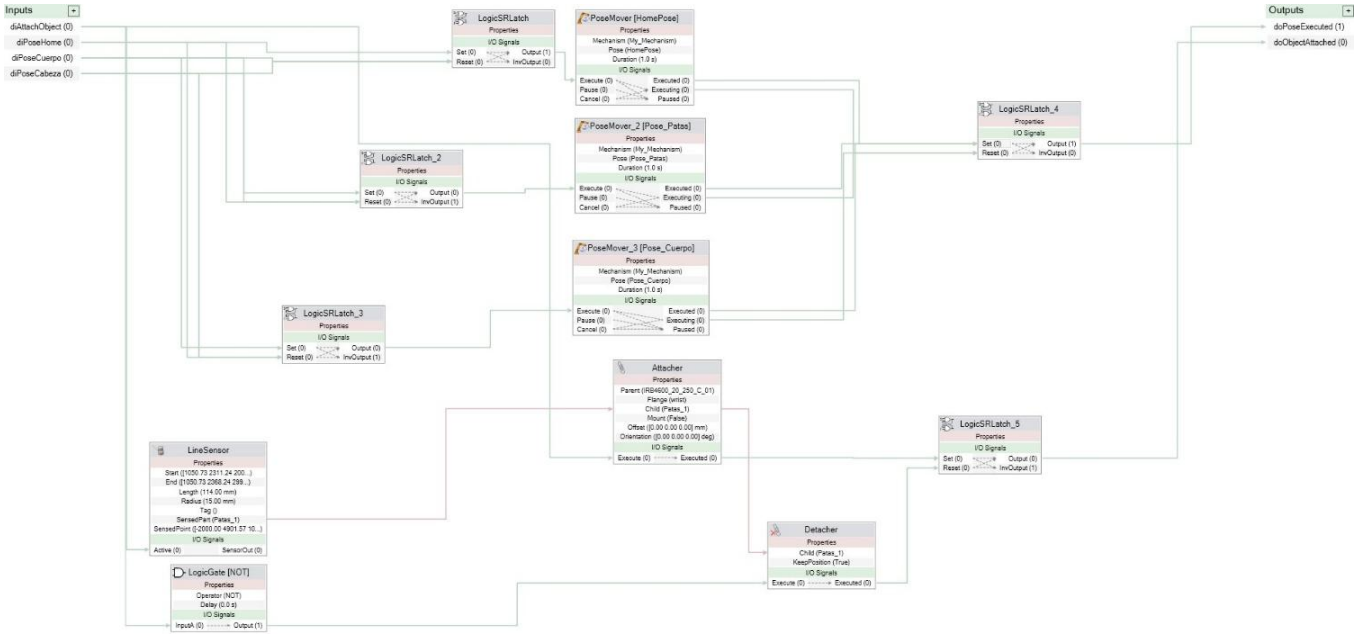
Our grippers were designed using Solid Edge and implemented as Smart Components in RobotStudio.



These grippers include collision detection, pneumatic behavior simulation, and programmable open/close positions through digital signals.

When a gripper approaches a part, the "ClosestObject" component detects proximity, and the "Attacher" component creates a parent-child relationship between the gripper and the part when activated.

This makes the gripper behave realistically in the simulation, allowing it to detect when it has successfully taken a part and provide feedback through signals like diObjectAttached_Gripper.

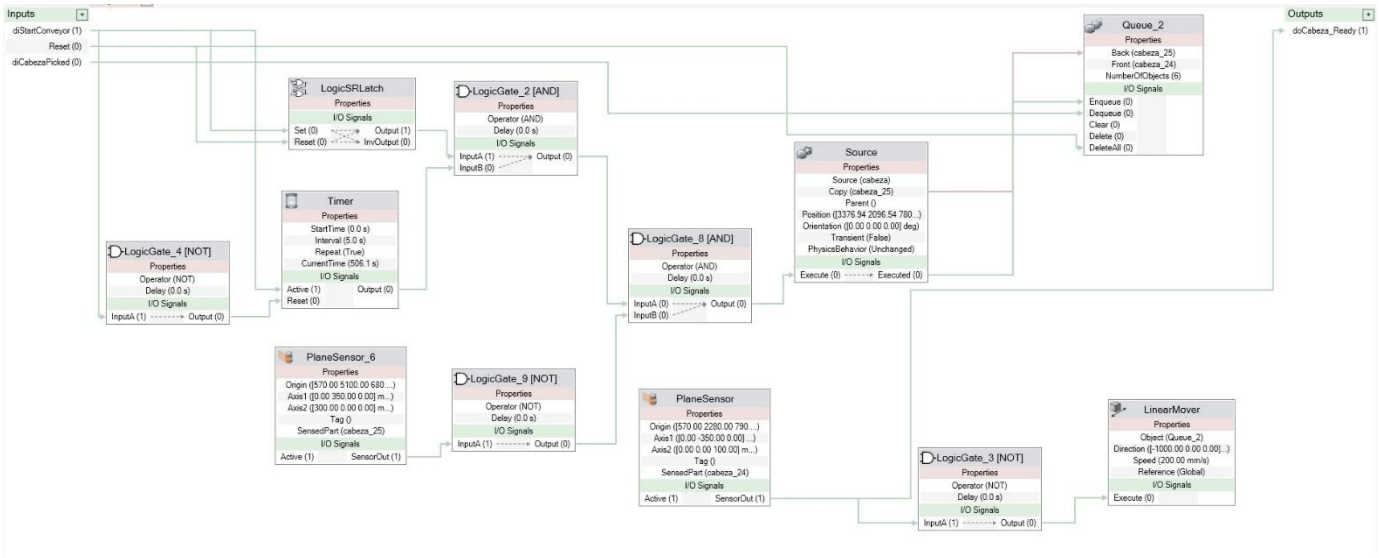


## B.10 Smart Component Integration

Apart from the gripper, our manufacturing cell incorporates several smart components that improve significantly the simulation's realism and functionality:

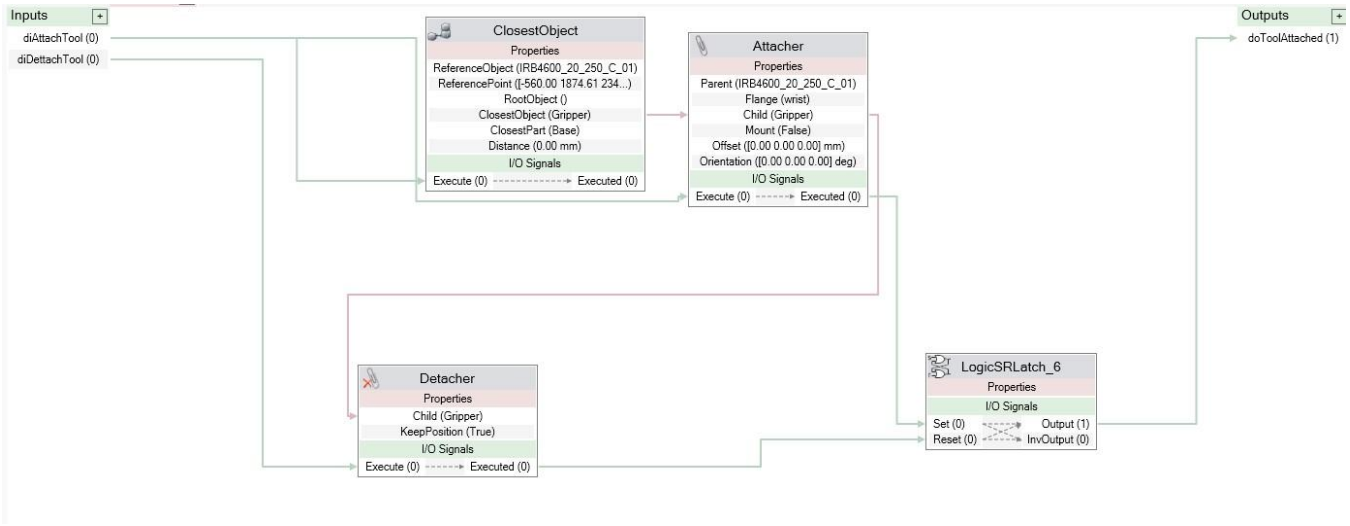
1. **Conveyor Systems (SC_C_1, SC_C_2, SC_C_3)**

   These smart components simulate the three conveyor belts that transport the different parts (heads, bodies, and legs) to the pickup positions. Each conveyor includes sensors that detect the parts and read signals through digital inputs (diCabezaReady, diCuerpoReady, diPatasReady).

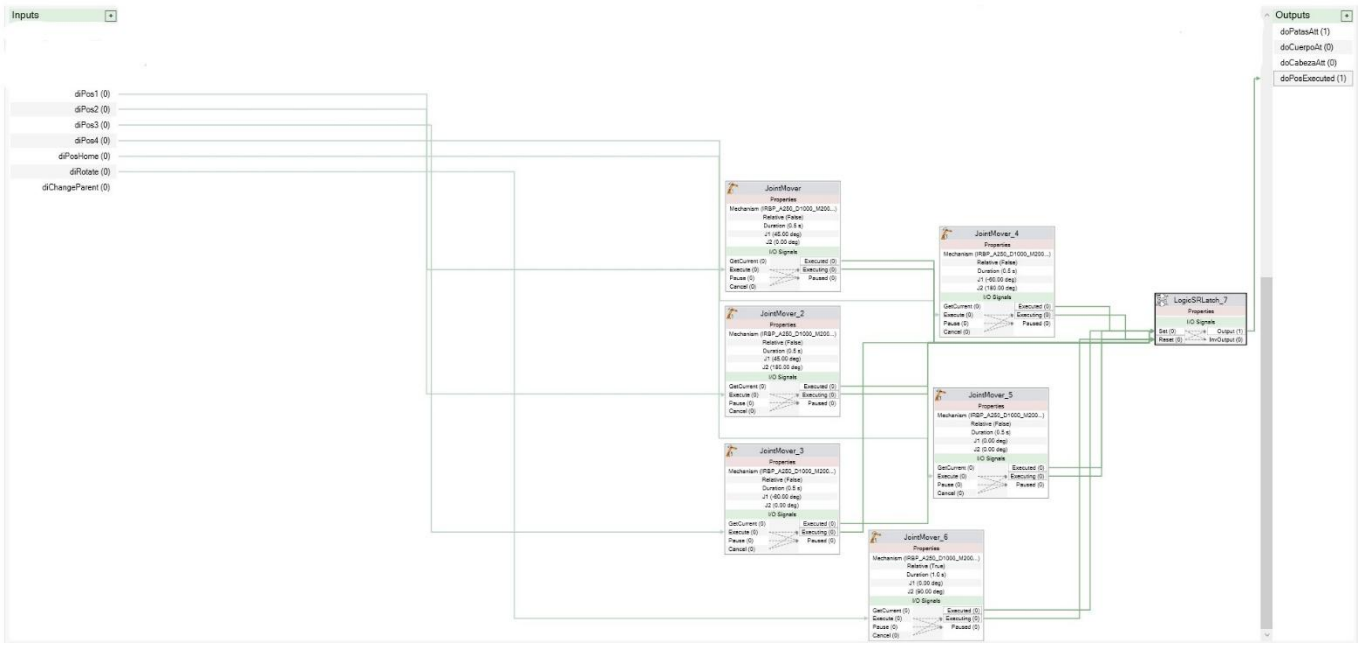



2. **Tool Changers (Tool_Attacher_ROB_1, Tool_Attacher_ROB_2)**

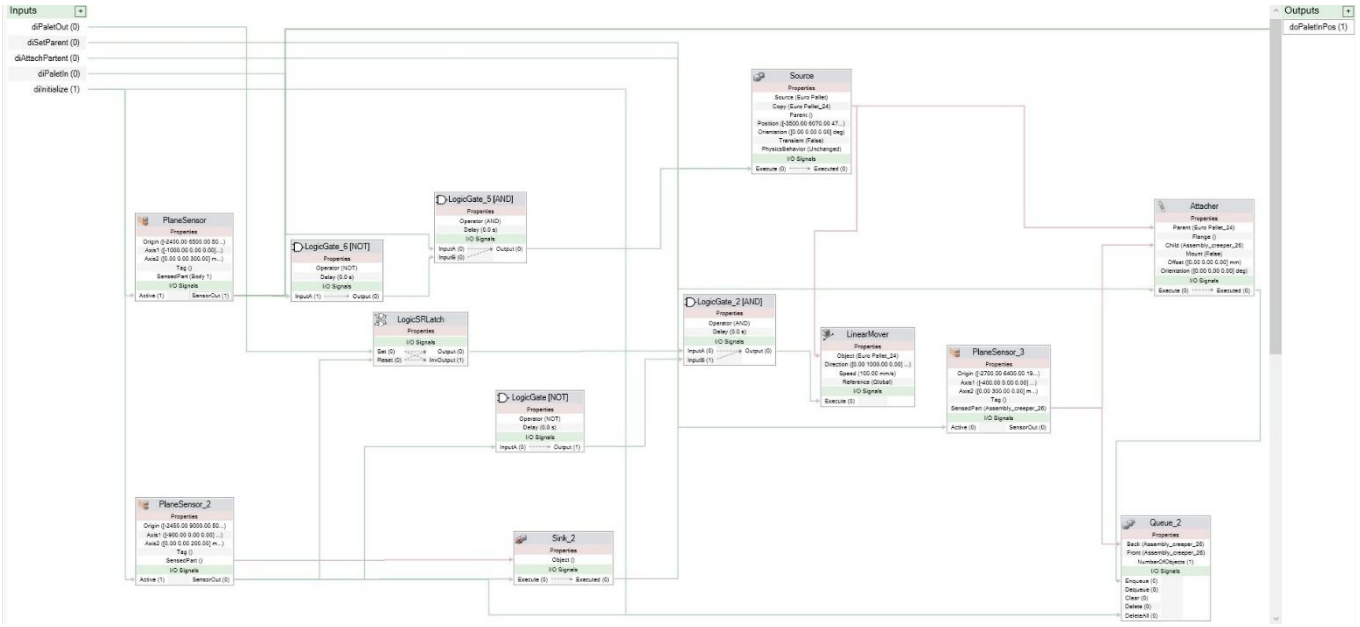
   These components manage the tool change process. They detect the proximity to the tool holders and the handling attachment and detachment signals (diAttachTool, diDetachTool) to realistically simulate changes.



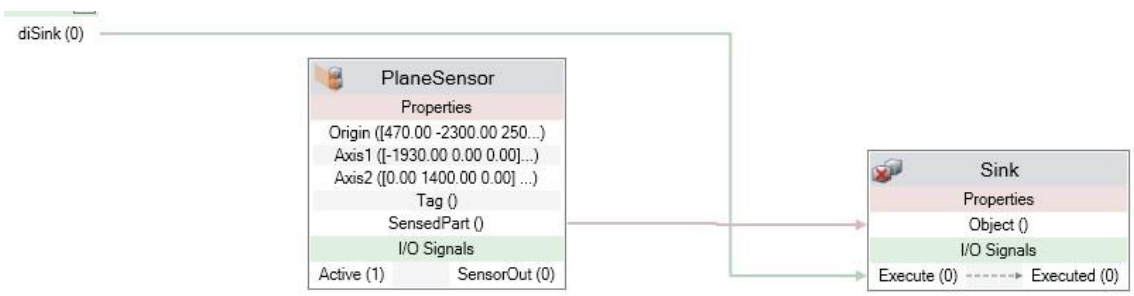
3. **Positioner Mechanism**: The rotary work positioner uses smart components to control its movement and part attachment. It handles the different positions (0°, 90°, 180°, 270°) through JointMover components and coordinates the transfer of parts between robots.



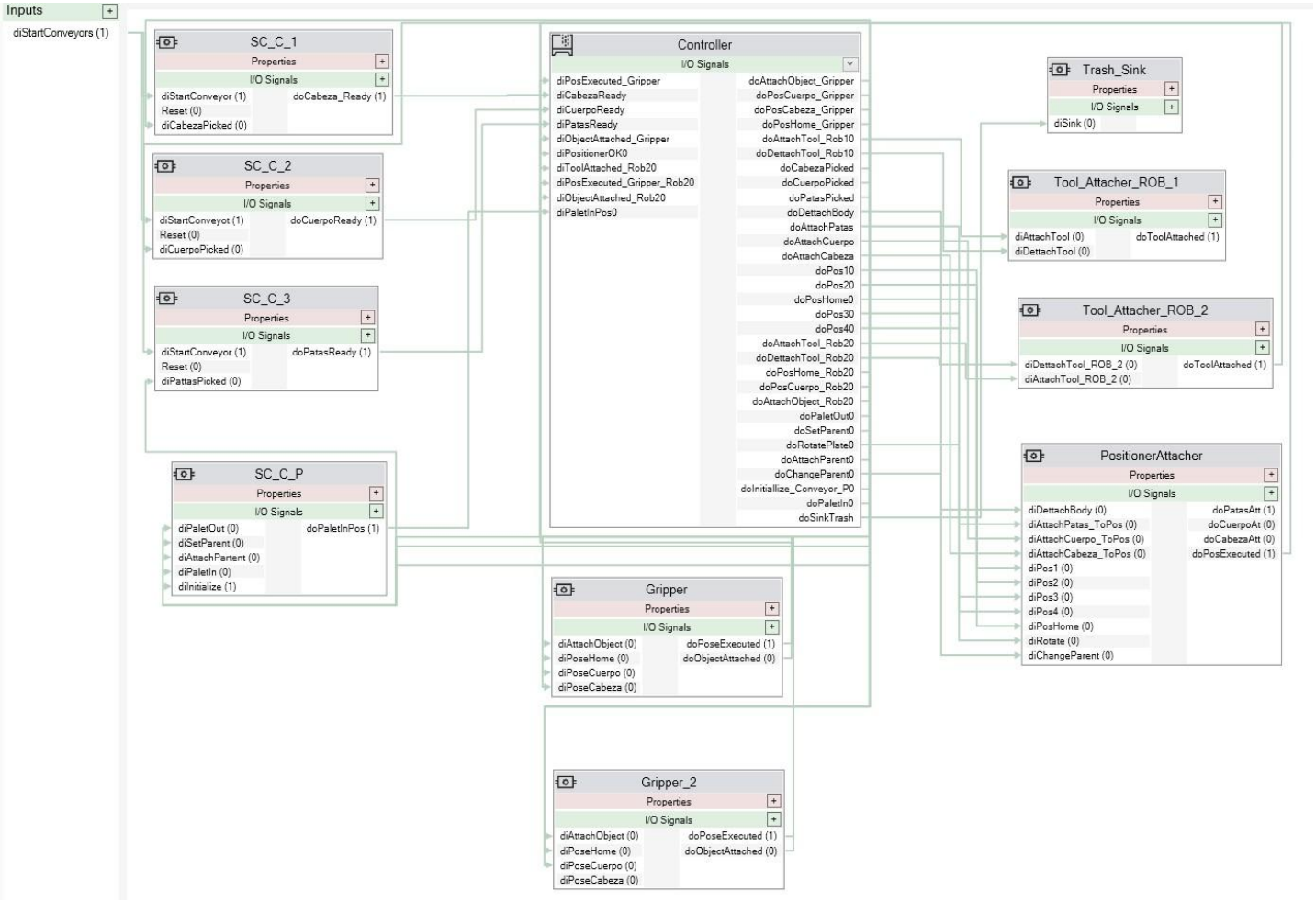

4. **Pallet System (SC_C_P)**: This smart component manages the pallet conveyor, handling the initialization, positioning, and movement of pallets as they're filled and replaced.




5. **Trash Container**: A simple but effective smart component that detects when defective parts are dropped and removes them from the simulation.



Finally, this diagram shows the Station Logic of the whole system:




## B. 11 Single Controller Configuration

We optimized our cell by using a single controller to manage both robots. This approach simplifies how the robots communicate with each other, reduces the amount of hardware needed, and allows them to share information directly.

The controller manages all I/O signals and coordinates the actions of both robots throughout the manufacturing process.

### B.12 Absolute Joint Movement Implementation

Our program utilizes the MoveAbsJ instruction for the most critical positioning operations:

```
MoveAbsJ JT_Safe_Track_10, v500, z200, tool0;
```

This movement instruction moves the robot directly to a specific joint configuration without depending on its current position.

Unlike relative movements, MoveAbsJ guarantees that the robot reaches the same position every time, making it more consistent for critical operations.

### B.13 TEST-CASE Non-Sequential Program Flow

We implemented non-sequential program flow using TEST-CASE structures that allow the robot to execute different operations based on the current phase:

```
TEST Phase
    CASE 1:
        TPWrite "PHASE 1: Assembly Patas...";
        ...
    CASE 2:
        TPWrite "PHASE 2: Assembly Cuerpo...";
        ...
ENDTEST
```

This creates a more flexible program structure compared to sequential execution. This way the system executes only the specific code required for each phase without unnecessary operations.

### B.14 Persistent Variable Implementation

We used PERS type variables to maintain values between simulation sessions:

```
PERS bool soldar;
PERS bool WorkingZone_Busy;
PERS num i;
```

These persistent variables ensure that important information (like pallet position counter, busy zones, etc.) is saved even when the controller is turned off, so production can start again correctly.

### B.15 Three-Parameter Procedures

We implemented advanced procedures with three parameters:

```
PROC pick_part(robtarget Target, PERS tooldata tool, PERS wobjdata frame)
```

This lets us use the same procedure for different targets, tools, and work objects, avoiding repeating code and making our program easier to update and fix.

## C. Conclusions

This project allowed us to simulate a complete automated manufacturing cell with two industrial robots working in coordination. The most challenging aspects were synchronizing both robots without collisions, implementing reliable tool changes, and structuring a flexible program flow that adapts to different phases and part quality.

Despite the complexity, we managed to create a robust and realistic simulation using Smart Components, shared variables, and modular RAPID code. The project has given us valuable experience in robot programming, process automation, and system integration — all essential skills for real-world industrial robotics applications.