DTA - FH Aachen

2023, Prof. Jörg Wollert (FH Aachen)

Jul 14, 2023

CONTENTS

1 Preamble **3**

2 Digital Twin of a machine plant **5**

3 Siemens NX MCD: Simple Robotic Arm Modelling **7**
3.1 What you need . . . 7
3.2 Preparation . . . 7
3.3 Assignment . . . 7

4 Virtual Commissioning of a Production Line **11**
4.1 Learning Outcome . . . 11
4.2 Task . . . 12
4.3 Preparation: Station IMS3 . . . 12
4.4 Station IMS3: Practical Assignment . . . 15
4.5 Preparation: Station IMS4 . . . 27
4.6 Station IMS4: Practical Assignment . . . 29
4.7 Preparation: Station IMS5 . . . 43
4.8 Station IMS5: Practical Assignment . . . 44
4.9 Preparation: Station IMS7 . . . 55
4.10 Station IMS7: Practical Assignment . . . 56
4.11 Summary . . . 68
4.12 Troubleshooting . . . 68

5 Operator Digital Twin for Assistance Systems **71**
5.1 Short Introduction to Digital Twins . . . 71
5.2 Digital Twins of Operators . . . 72

6 OEE Evaluation with Python **73**
6.1 Learning Outcome . . . 73
6.2 Task . . . 74
6.3 Setting up the work environment . . . 74
6.4 Understanding the work data . . . 79
6.5 Basic program structure . . . 80
6.6 Data preparation - Work data . . . 81
6.7 OEE calculation . . . 82
6.8 Visualization . . . 84
6.9 Data evaluation . . . 85
6.10 Problem Solving . . . 85

7 Digital Twin of a Robot Dog **87**
7.1 Short Introduction to Digital Twins . . . 87

7.2 Creating a Digital Twin for a Robotic Dog . . . 88

8 Bittle 93
8.1 Learning Outcome . . . 93
8.2 Workspace Setup . . . 95
8.3 Hardware Setup . . . 102
8.4 Bittle Body . . . 104
8.5 Electrical and Controller Properties . . . 105
8.6 Calibration . . . 113
8.7 Controlling Bittle . . . 122
8.8 Extensible Modules . . . 128
8.9 Serial Commands Cheat Sheet . . . 137
8.10 Tasks . . . 137
8.11 Teach Bittle New Skills . . . 141
8.12 Controlling Bittle (Advanced) . . . 158
8.13 Controlling Bittle (Codecraft) . . . 163
8.14 Problem Solving . . . 171

9 Der Roboterhund Bittle 179
9.1 Learning Outcome . . . 179
9.2 Einführung . . . 179
9.3 Grundlagen von Robotern und Programmierung . . . 180
9.4 Programmierung . . . 183
9.5 Tasks . . . 191
9.6 Weitere Informationen zu Bittle . . . 196
9.7 Additional Tasks . . . 197

10 Arduino Basics 201
10.1 Introduction . . . 201
10.2 Embedded Systems . . . 202
10.3 Programming in C . . . 203
10.4 Arduino Programming . . . 208
10.5 Getting started with Arduino IDE . . . 210
10.6 Problem Solving . . . 211

11 TIA Portal Basics 213
11.1 Learning Outcome . . . 213
11.2 TIA Portal . . . 213

12 TIA OPC-UA 225
12.1 Configuring OPC-UA server in TIA Portal . . . 225
12.2 Creating OPC-UA Server Interface . . . 227

13 PLCSIM Advanced 229
13.1 Setting up simulation support in TIA Portal . . . 229
13.2 Launching PLCSIM Advanced and creating a PLC instance . . . 229

14 OPC-UA client 233
14.1 Prosys OPC-UA Browser . . . 233

15 PLC-Basics 235
15.1 Information processing . . . 236
15.2 PLC Programming (IEC 61131) . . . 237
15.3 Programming languages (IEC 61131) . . . 239
15.4 Sources . . . 249

16 Siemens NX MCD **251**
16.1 Learning Outcome . . . 251
16.2 Virtual Commissioning . . . 251
16.3 Mechatronics Concept Designer . . . 252
16.4 Summary . . . 268

17 Node Red Basics **269**
17.1 Learning Outcome . . . 269
17.2 General information . . . 270
17.3 Nodes . . . 272
17.4 Flows . . . 273
17.5 Messages . . . 273
17.6 Information and Debug . . . 275
17.7 Palette Manager . . . 275
17.8 Functions . . . 275
17.9 Tasks . . . 278

18 Visual Studio Code Basics **283**
18.1 Learning Outcome . . . 283
18.2 General information . . . 284
18.3 Gitlab Extension . . . 287
18.4 Arduino Extension . . . 292

19 Introduction to Version Control with Git **297**
19.1 Learning Outcome . . . 297
19.2 Introduction . . . 298
19.3 Workflow . . . 298
19.4 Commits . . . 301
19.5 Git Branches . . . 303
19.6 Merging and Merge conflicts . . . 303
19.7 Tags . . . 303
19.8 Rebase . . . 305
19.9 Git Interface . . . 307
19.10 Git Commands . . . 307
19.11 Tasks . . . 308

This page contains the learning activities developed for the Digital Twin Academy by the FH Aachen University of Applied Sciences.

CHAPTER

ONE

PREAMBLE

The following Workbook contains some special formatted parts to guide you through the Sprints and Tasks. **Please read these parts really carefully.**

Note: This is a hint to get better solutions.

Warning: This is a warning.

Danger: Danger! Check carefully, you may damage hardware by doing something wrong.

```
This is source code.
```

Todo: This is a task to check your knowledge.

CHAPTER

TWO

DIGITAL TWIN OF A MACHINE PLANT

Today, digital twin concepts can be applied in a variety of different fields including the manufacturing sector. DTs can for example help design assets or expand existing assets. DTs do that by allowing different applications access to the assets' information. One such application that could be DT-driven is virtual commissioning, which will be covered in this course.

This course serves as a beginner's guide and first step into virtual commissioning. The usage of software tools like Siemens NX, PLCSIM Advanced, and TIA Portal is described step by step. Previous knowledge in automation technology is a plus, but not a requirement for this course.

CHAPTER

THREE

SIEMENS NX MCD: SIMPLE ROBOTIC ARM MODELLING

In this module you will model a simple robotic arm mechanism using Siemens NX.

3.1 What you need

3.1.1 Software

- Siemens NX version 1872 or newer (pre-installed on the lab PC)

3.1.2 Files

CAD files of the examples used in this module can be found here: https://fh-aachen.sciebo.de/s/AFAGrOi2zRFL1y7

For this task, the correct assembly can be found in `simple_robotic_arm`. Download all the parts from the `simple_robotic_arm` file and open `ASSEMBLY_simple_robotic_arm` in Siemens NX.

3.2 Preparation

- *Siemens NX MCD*

3.3 Assignment

In this exercise a running simulation of a simple robotic arm mechanism will be created. The CAD files are available. In Siemens NX MCD, different physical properties will be assigned to parts in order to reflect the function of the simple station.

Note: When you first start NX, you may get an error saying there are no licenses available. Click "OK" and navigate to File->Utilities->Select Bundles. In the window that opens, select and add both available bundles then click `OK`. This solves the error.

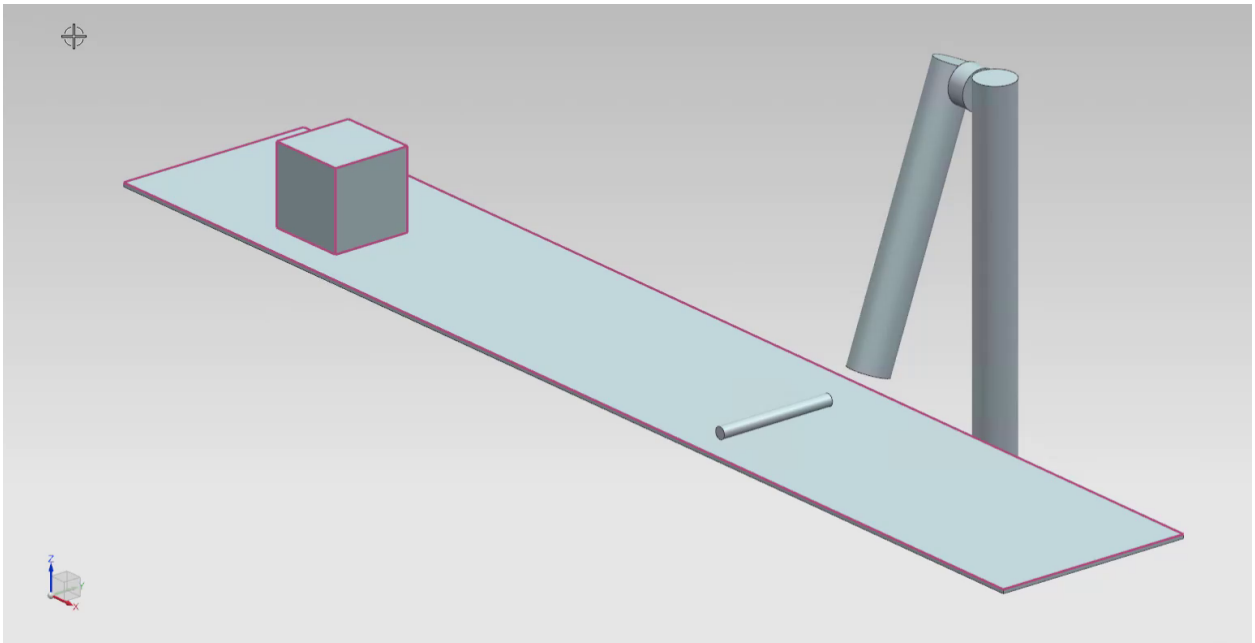


Fig. 3.1: Simple robotic arm functionality

3.3.1 Box and Conveyor Band

In the first steps, rigid and collision bodies will be assigned to parts the will interact with each other.

Todo:

- Assign a rigid body to the box and call it `Box`.
- Assign a collision body to the box and call it `Box`. Hint: to choose the entire box volume, right click on the box and select `Choose from list`. Then select the entire box from the list.
- Assign collision body to the upper surface of the transporter band and call it `Transport_band`.
- Assign a transport surface to the upper surface of the transporter band and call it `Transport_band`. Specify the correct direction vector and set the parallel velocity to `100 mm/s`.
- Run the simulation and check if the box is transported on the transport band.

3.3.2 Simple Robotic Arm

Todo:

- Assign a rigid body to the robotic arm and call it `Robot_arm`.
- Assign a collision body to the robotic arm and call it `Robot_arm`.
- Assign a hinge joint to the robotic arm and call it "Robot_arm_HingeJoint". Specify the direction vector and the anchor point in a way that it rotates around the robotic joint.
- Assign a position control to the hinge joint object. Test different `Destination` and `Speed` settings.
- Finally set the destination to `0` and the speed to `500 °/s`

3.3.3 Collision Sensor

Todo:

- Assign a collision sensor to the small cylinder on the transport band and give it a suitable name.

3.3.4 Conditions - Using Operations as if-Statements

Todo:

- Create an operation in the `Sequence Editor`. Choose the position control object as the `Select Object` choice. Make a check next to `position` and give in a value of `70`. Select the collision sensor object (from the) as the condition object and specify that `triggered == true`. Name the operation `Arm_out`.
- Create a second operation. Choose the position control object as the `Select Object` choice. Make a check next to `position` and give in a value of `0`. Select the collision sensor object (from the `Physics Navigator`) as the condition object and specify that `triggered == false`. Name the operation `Arm_in`.

Run the simulation and check if it achieves the desired behavior.

CHAPTER

FOUR

VIRTUAL COMMISSIONING OF A PRODUCTION LINE

In this module, virtual commissioning is applied on the Lucas Nülle Industrial Mechatronic System (IMS) assembly line.

4.1 Learning Outcome

4.1.1 Introduction

In the context of virtual commissiong, a physics-based model of the Lucas Nülle assembly line should be created and used to verify the PLC control code.

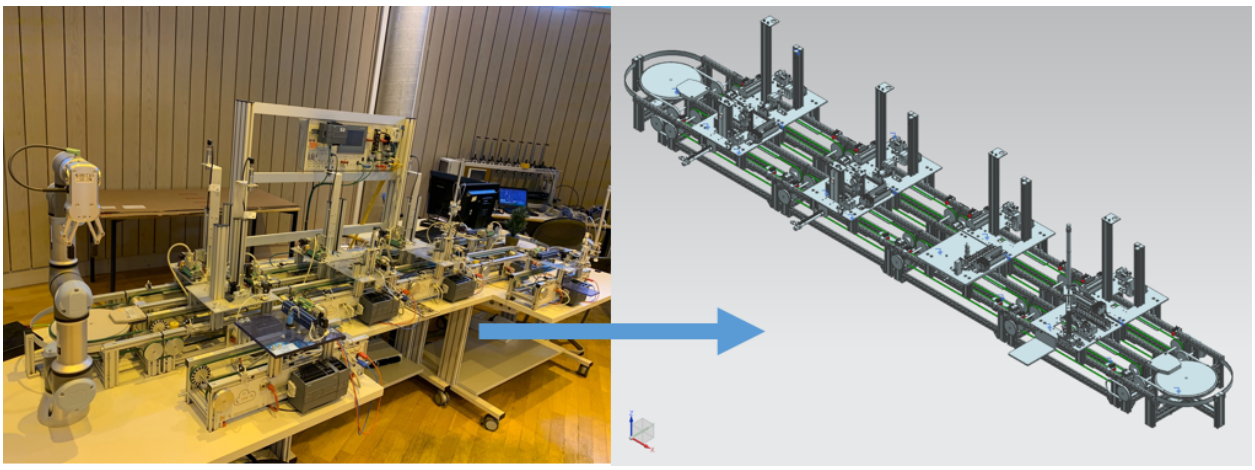


Fig. 4.1: The Lucas Nülle assembly line modelled in Siemens NX

- You will carry out a virtual commissioning application.
- You will get to apply your knowledge of the basics of Siemens NX MCD.
- You will create a physics-based model.
- You will get familiar with the TIA Portal environment.
- You will be able to program Siemens PLCs using FBD, LAD, and SCL programmming languages.
- You will be able to simulate 15xx Siemens PLCs.

4.1.2 Requirements

- *PLC-Basics*
- *TIA Portal Basics*
- *PLCSIM Advanced*
- *TIA OPC-UA*
- *Siemens NX MCD*
- *OPC-UA client*

4.1.3 What you need

Software

- Siemens NX version 1872 or newer
- TIA Portal V15 or newer
- PLCSIM Advanced V2.0 or newer
- CAD files of the respective IMS station

Files

CAD files of the IMS stations of the practical assignment as well as for the examples used can be found here: https://fh-aachen.sciebo.de/s/IEPUWHYIDWIKSZX

4.2 Task

The goal of the IMS assembly line is to assemble a product that consists of three parts shown in the following figure. Different station in the assembly line are responsible for mounting different parts of the product or handling the end product.

The end product consists of a top-part, a bottom-part and a bolt that connects both parts.

The following sections will handle each IMS station. The task is to build a kinematic model in NX MCD and write a control code in TIA Portal for each station. The control code should uploaded to a simulated PLC and an OPC-UA connection should be established between the PLC and the kinematic model.

4.3 Preparation: Station IMS3

4.3.1 Goal

- To get to know the structure and functionality of the IMS3 station

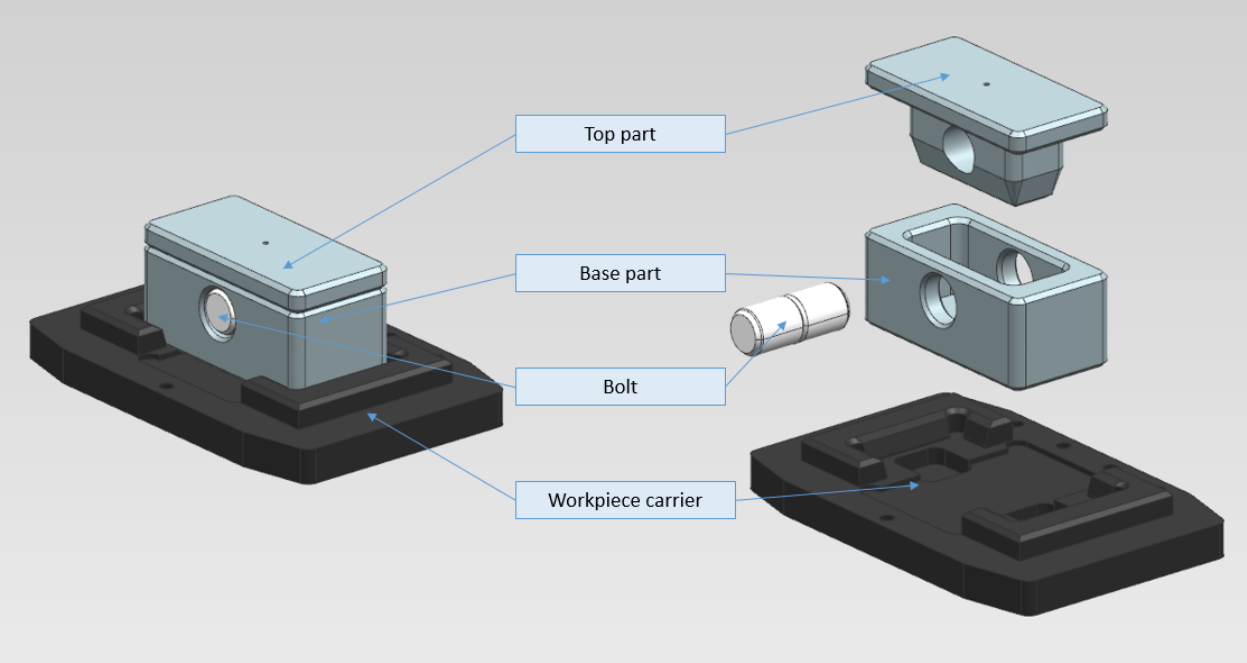


Fig. 4.2: The end product

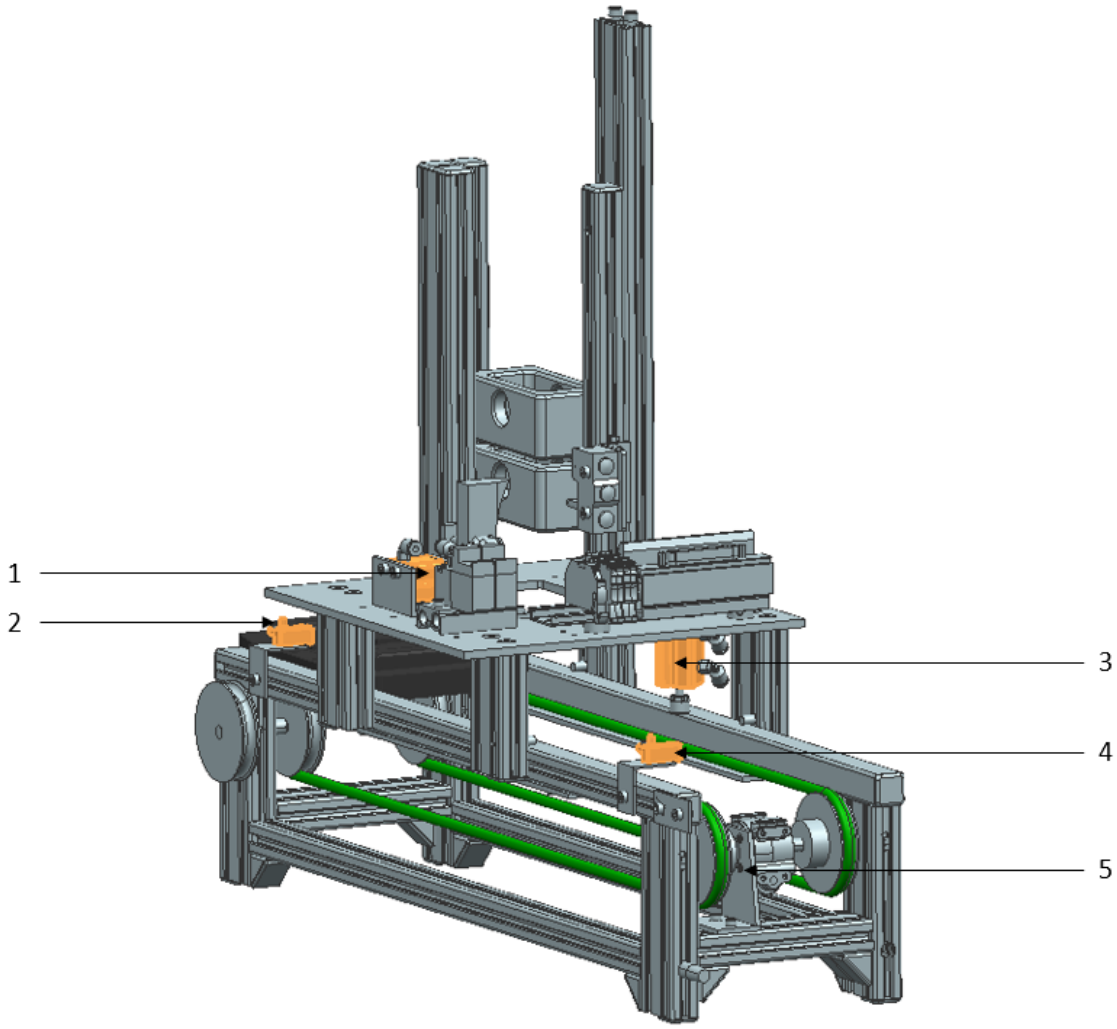
4.3.2 Task

The following questions/task will help you get familiar with the functionality of the station.

- Identify each of the components highlighted in the following figure and their respective function. Are they sensors or actuators? Would they be defined as inputs or as outputs in a PLC program?
- Describe or outline in a sketch the process of the station.
- Sketch a state machine diagram to describe the functionality of the station.

Siemens NX MCD

- To which elements should a rigid body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a collision body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a joint be assigned in Siemens NX MCD? What type of joints are required to model this station's behaviour? Is specifying both attachment object and base object required?
- For which objects should a position control be created? (hint: position controls are created to manipulate the position of joints)
- Which signals should be created inside Siemens NX MCD? What is the function of each signal?
- Which operations should be created? Describe the function of each operation.


1
2
3
4
5

PLC Program

- Which inputs and outputs should be defined in the PLC program?

4.4 Station IMS3: Practical Assignment

4.4.1 Goal

- To create a physics-based model of the IMS3 station
- To implement the state machine of IMS3 in a TIA Portal program
- To control the physics model with the created state machine program via OPC UA

4.4.2 IMS3 Station Functionality

The purpose of the IMS3 station is to mount the base part on the workpiece carrier.

The station has a few sensors and actuators, shown in the following figure.

Note: The guiding magazine in the above figures is blinded out, however, the active collision surfaces assigned to it are shown in pink.

4.4.3 The Virtual Layer - Mechatronics Concept Designer

Physical properties should be assigned to different parts of the IMS3 station. Open the assembly file called `ASSEMBLY_IMS3.prt` in Siemens NX and navigate to the `Mechatronics Concept Designer` application.

Base part and workpiece carrier

The base part (the part being mounted) and the workpiece carrier already have rigid body and collision bodies assigned to them. These objects are assigned on the single part's level. To see the objects, double click on the respective part in the `Assembly Navigator` window and navigate to the `Physics Navigator` window.

Note: Definition of collisions: Two objects collide when they get in contact with and exert forces onto each other. Collisions are part of the station's functionality. In the context of the simulation, collisions do not refer to accidents.

Rigid and collision bodies

Note: When assigning collision bodies to parts, only sides/areas of the part that will undergo collision during the simulation should be chosen. Avoid defining the entire part as a collision body, because it increases the complexity of the body and unnecessarily slows down the simulation.

Note: When assigning collision bodies to parts, try to choose simple geometry (box, cylinder, sphere, area etc.) if complicated geometry (mesh) is not crucial for the physical behaviour of the part.

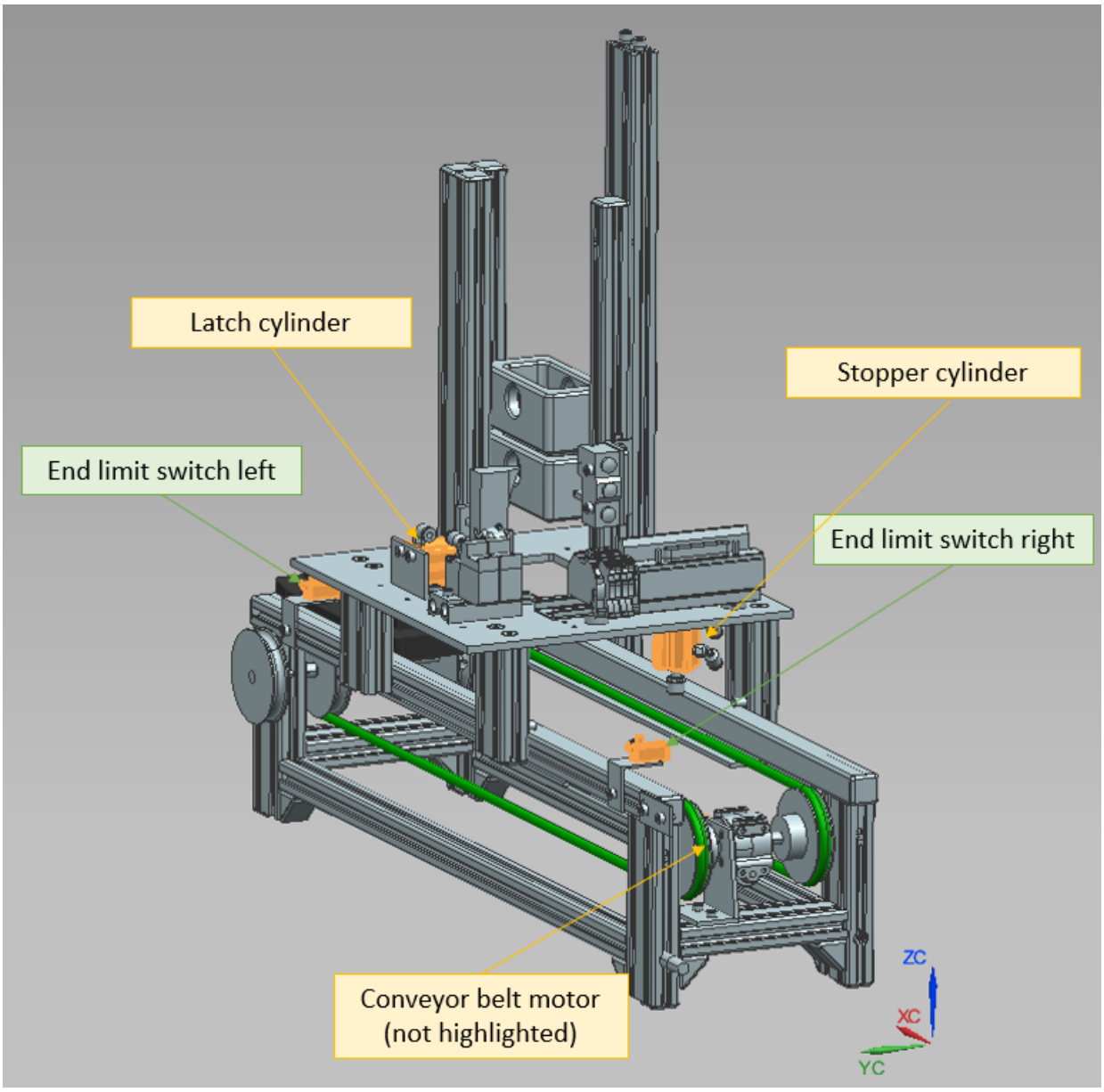


Fig. 4.3: Sensors and actuators on the IMS3 station

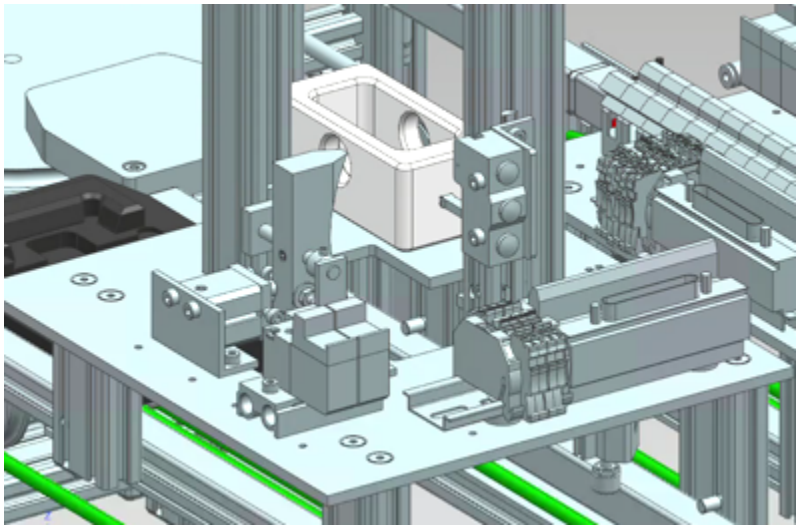


Fig. 4.4: The station's functionality

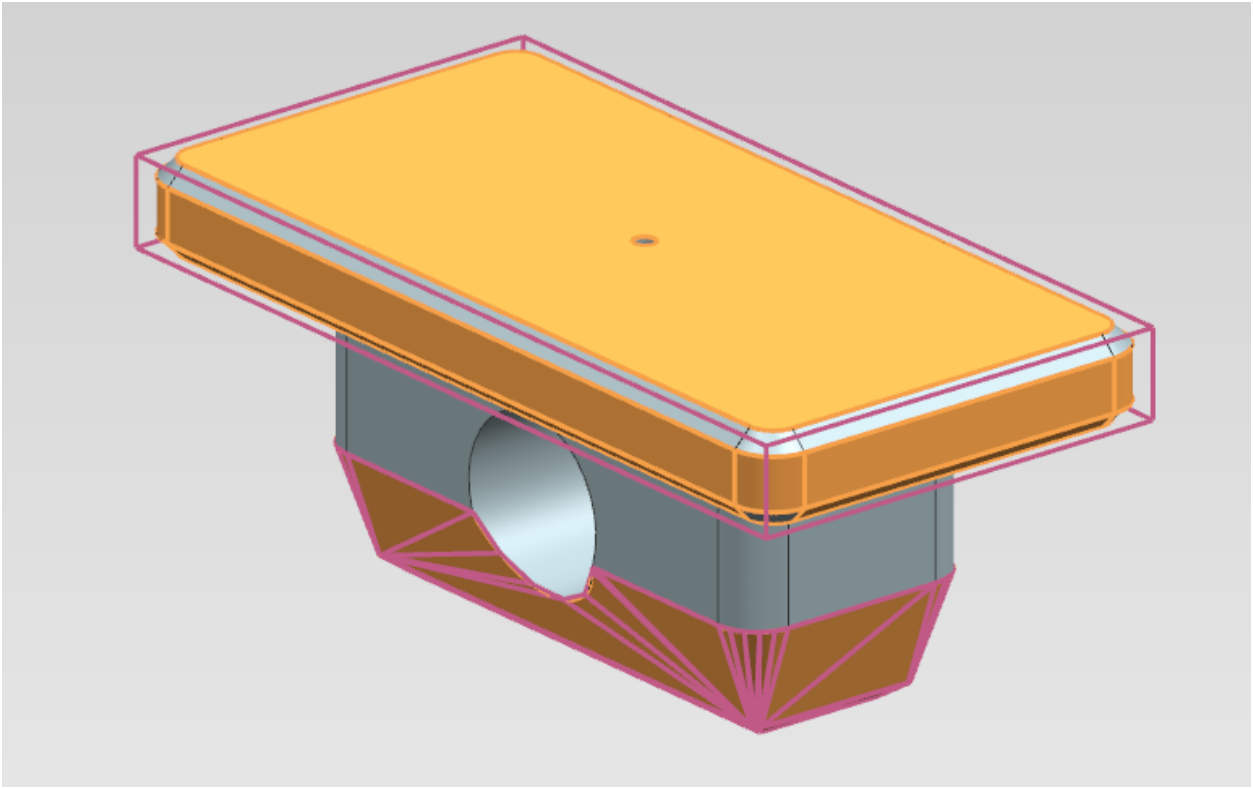


Fig. 4.5: An example of collision body assignment. Collision surfaces are shown in pink. The upper part has a simplified geometry and surfaces on the part that will not undergo collision were not chosen, which reduces the complexity.

Todo: Assign collision bodies to the magazine surfaces that store the parts and along which the parts fall during the separation process. These surfaced keep the parts in place and guide the parts during the separation process. Assign other collision bodies to the left and right side areas for the conveyor belt. These areas help guide the carrier along the conveyor belt.

Conveyor belt

To simulate a conveyor belt, a rectangular surface will be used as a `Transport surface`. The band itself will be blinded out while the simulation is running.

Todo: Assign a `Transport surface` to the part `200213_TransportflaecheDUMMY`. Do that on the `Entire_Transport_Band` level (double-click on `Entire_Transport_Band` from the Assembly Navigator to enter that assembly's level).

Mounting mechanism - latches

Rigid and collision bodies

A latch prevents the current part from falling and separates parts from the magazine when the mechanism is triggered. Assign a rigid and collision body to the latch. Use the option `mesh` for the collision body.

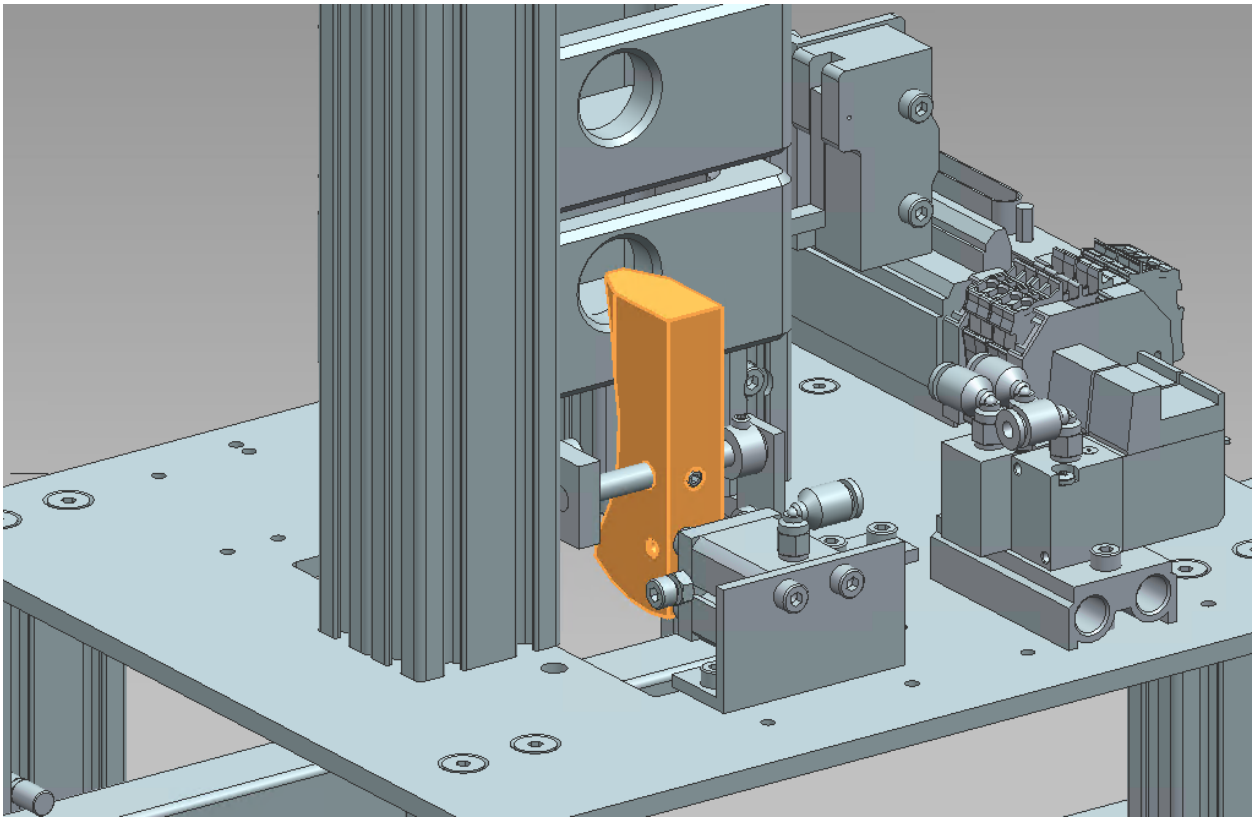


Fig. 4.6: A latch is used for the separation process.

Note: Assign physical properties to the latches on the `LM9680_Vereinzeln_SE` assembly level. Can mesh be avoided? Try using a simplified collision geometry for the parts of the latch that actually collide.

Joint

The latch should be allowed to rotate around the bolt on which it is mounted.

Todo: Assign a `hinge joint` to the latche. Specify the axis vector and the anchor point around which the latche will rotate. Because the base of the joint (i.e., the rest of the assembly) will not be moving, the base object can be left unassigned. Give the joint a clear name.

Angular Spring Joint

The latch 'latches' back to its original position through an angular spring joint (also known as a simple torsion spring). Assign an `Angular Spring Joint` to the latche. Set the spring constant to `20 N.mm/°`, damping to `0.1 N.mm.s/°`, and relaxed position to 30° away from up-straight adjustment. The following figure shows the angular position of the latch. In a relaxed position (if there were no cylinder to collide with the latch), the latch would be at a 30° angle to the vertical axis. With the cylinder touching the latch, the 30° is never reached. The resulting new angle (marked in blue) is not relevant to setting the parameters of the `Angular Spring Joints`.

Note: To check if the given parameters reflect the intented behaviour, one can run the simulation and see how the latches will rest. Always run the simulation on the top most hierarchy level i.e., in the IMS3 assembly and not in a sub-assembly.

Danger: Running the simulation in a sub-assembly might be handy, but it leads to an OPC UA communication bug. Avoid running the simulation unless its in the top most assembly level. If OPC UA communication is not working, see the troubleshooting section on how to fix it.

The angular position of the latche will be physically controlled by the latch cylinder. As the cylinder phyically pushes onto the latche, it induces the separating motion for the bottom parts magazined in the part tunnel. As the cylinder retracts back, the latch latches back to its original position under the effect of the torsion springs.

Mounting Mechanism - Cylinders (Latch Cylinders and Stopper Cylinder)

A cylinder is used to push the latch and to perform the separation mechanism. When the cylinder is retracted, the latch swings back to its original positions through a torsion spring attached to it.

Note: The torsion spring is not shown in the CAD Model.

Rigid and collision bodies

Todo: Assign rigid and collision bodies to the cylinder parts.

Note: When assigning rigid bodies to part, a few parts can be selected together and defined as one colelctive rigid body. This should be done if those parts will always undergo the same motion together. In the case of the stopper

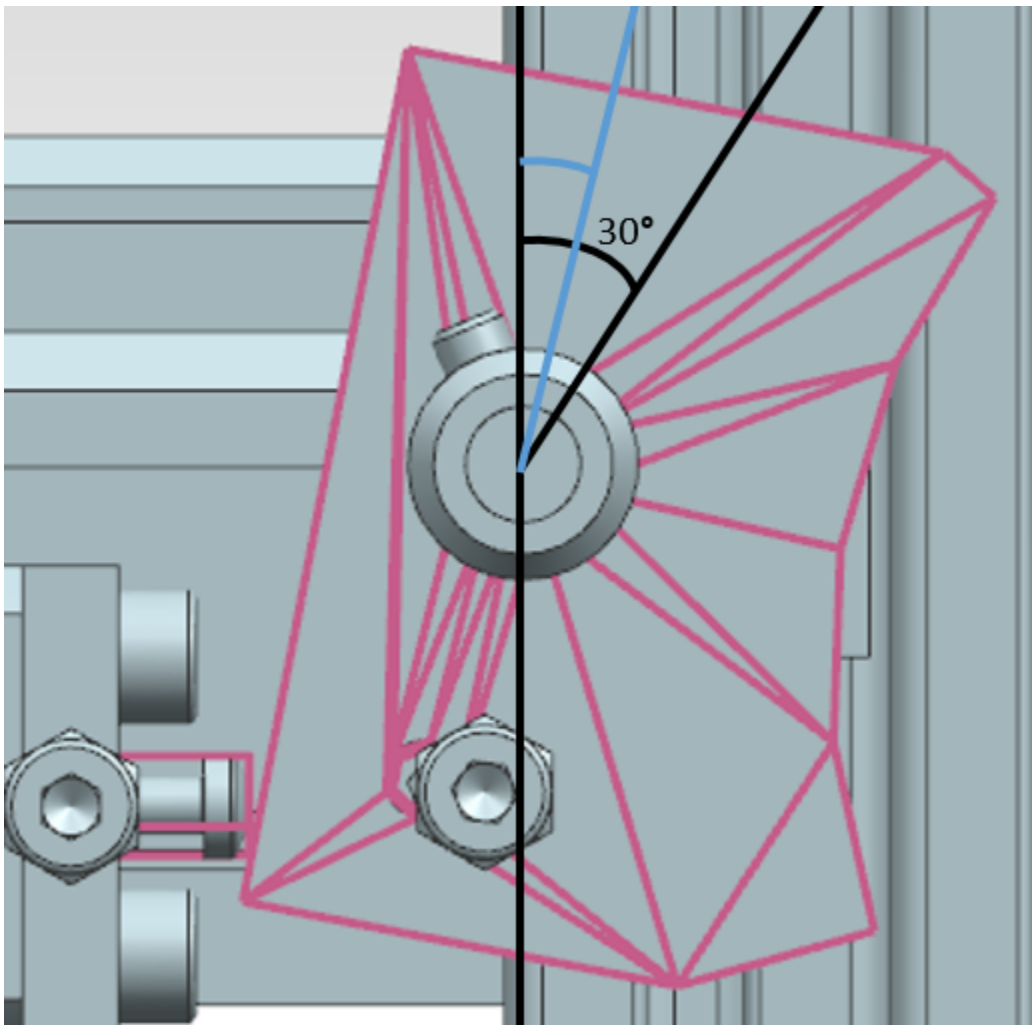


Fig. 4.7: Latch positioning with contact to the cylinder

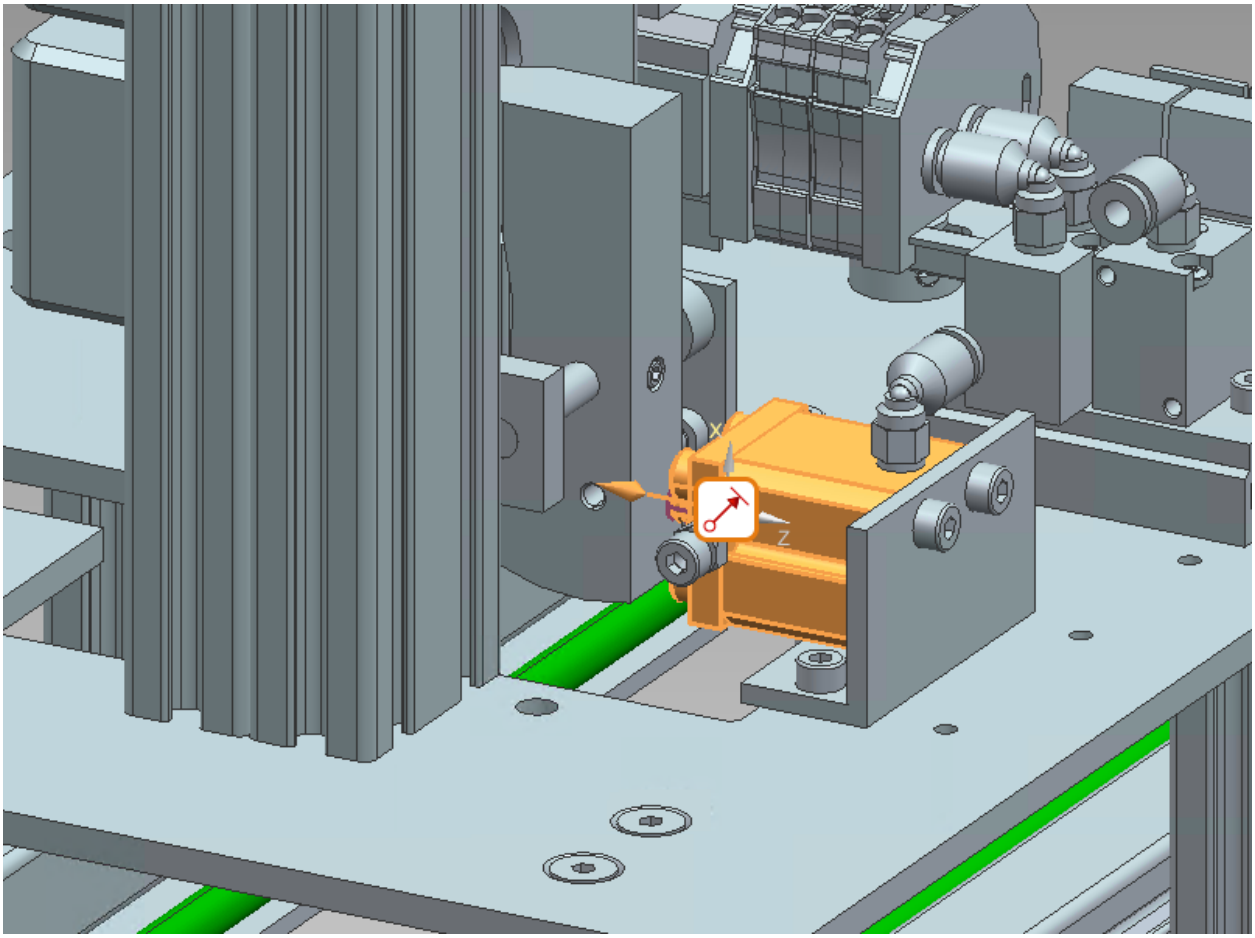


Fig. 4.8: A cylinder is used to push the latch. Another cylinder is used to stop the workpiece carrier.

cylinder, the cylinder itself but also the attached rubber object around it can be selected together.

Joints

Each cylinder should be constrained in all directions but one. Since every rod will only be either driven out or in, a sliding joint should be assigned to each cylinder.

Todo: Assign sliding joints to each cylinder.

Note: As an attachment object, the rods should be chosen. Becasue the housing of the cylinders will not move during the simulation, the base object can be left unselected. Selecting the housing of the cylinders as the base object is in this case optional.

Note: Make sure to name each of the joints appropriately. For example: `Cylinder_SlidingJoint`, and `Stopper_SlidingJoint`

Note: Define a lower limit of 0 and an upper limit of 10. This prevents the physical object of wandering beyond those limits.

Position Control

To control the position of the object on the sliding joint, position controls have to be assigned to the sliding joint objects.

Todo: Create position controls and assign them to each of the sliding objects.

Note: Name the position controls appropriately. For example: `Cylinder_PositionControl`, and `Stopper_PositionControl`

Signals

In order to, in turn, control those position controls through boolean variables from the PLC program later (the goal), one should create signals in MCD and name them appropriately. For example: `bStopper_down` and `bSwings_open`.

Note: Only one signal can be responsible for triggering both latches. Therefore, do not create two signals for the two latches, but create one signal to control both of them.

Signal Name	Signal Function
bStopper_down	sets the stopper's position control position to 5mm
bSwing_open	sets the latch's position control position to 5mm

Todo: Create 2 boolean signals in MCD for the position controls of the cylinders. Be sure to define the signal as an `Inputs`, since they are inputs from MCD's perspective.

Note: Do not link the signals to any runtime variables. These signals will trigger opertations later.

Note: You can create a symbol table for the all of those signals and call it `PLCSIM_Advanced`. Symbol tables are a good way to arrange and organize signals that are related together. Since these signals will communicate with PLCSIM Advanced, such a name for the symbol table provides clarity.

Operations

Operations are like if-statements in the MCD simulation. They can be used to trigger something in the simulation if an external (or internal) condition is met.

Todo: Create an operation to send the stopper cylinder down. Call it `Stopper_down_operation` for clarity.

Note: Operations can be found in the `Sequence Editor` menu on the left.

The selected physics object of the operation is the stopper's position control `Stopper_PositionControl`. The runtime parameter selected is the position, because this is the parameter to be manipulated by the operation. The new value is inserted in the `Value` column. Under `Condition`, the condition object selected is the signal `bStopper_down`. The if-value of the signal should be set to `true`.

The equivalent in C++ would be:

```
if(bStopper_down == true) {stopper_sliding_position = 5;}
```

Todo: Create an additional operation that would set the value of `Stopper_PositionControl` to `0 mm` if `bStopper_down` is false. Call the operation `Stopper_up_operation`.

Todo: Create 2 more operations to control the latch cylinder: Use `bSwings_open` as the boolean condition-variable for these operations.

Todo: Create a boolean signal and use an operation to turn the conveyor belt on and off.

Sensing Mechanism - Entry and Exit Sensors

Collision Sensor

The IMS3 has entry and exit sensors at the start and end of the conveyor belt (see section on IMS3 Station Functionality). These two sensors will be simulated as two small cubes with `collision sensor` properties.

Create a new `block` in `mechanical concept` under the `Home` tab. Give the block the dimensions of 10mm x 10mm x 10mm. Under the `Assemblies` tab, click on `Create New` to create a new model. Name the model something appropriate, like `end_limit_switch_dummy` and click `OK`. To select the block just created, navigate to `Part Navigator` on the left-side menu and choose the block that was just created. Now the block is a model and can be seen in the `Assembly Navigator`. Choose `Move Component` under the `Assemblies` tab and move the new block to where the sensor is supposed to be.

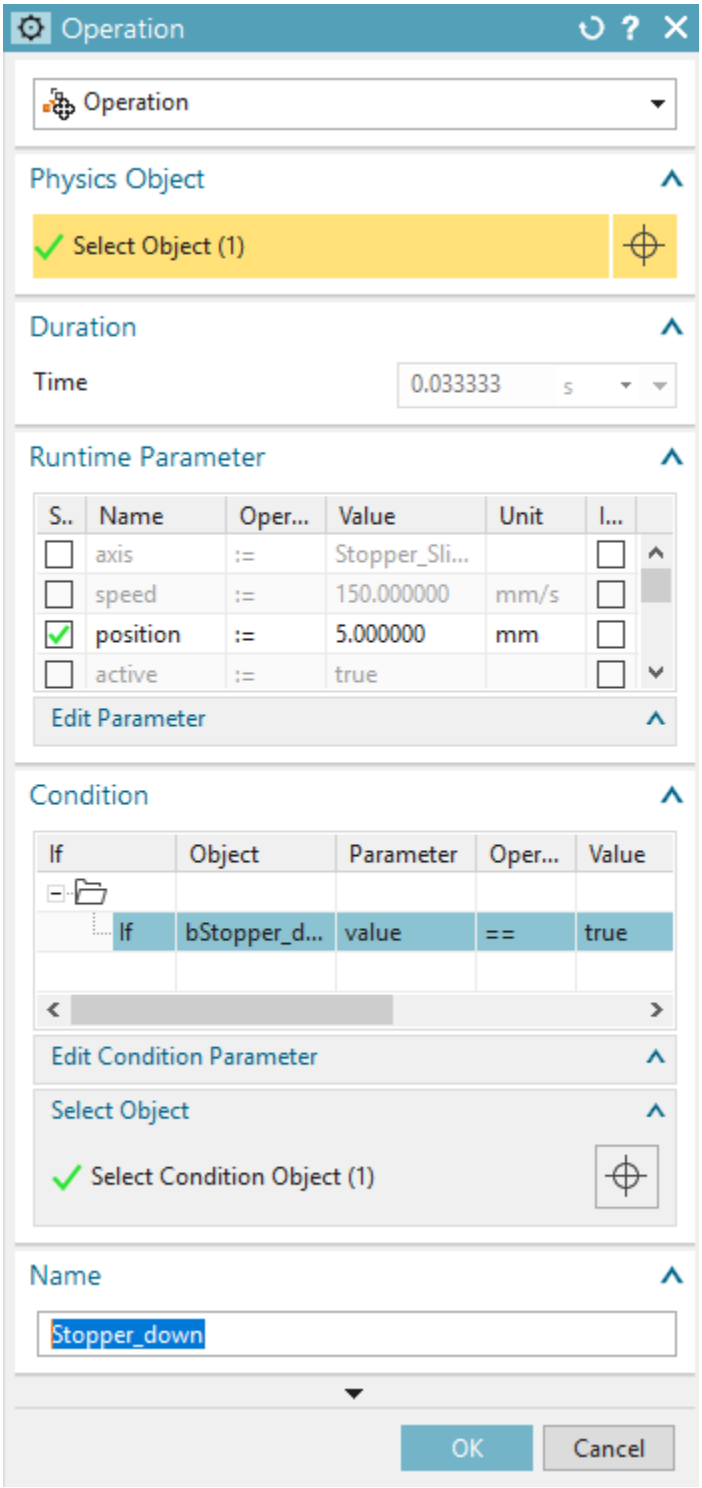


Fig. 4.9: An operation to send the stopper's sliding joint to position `5mm` if the signal `bStopper_down` is true.

Note: When moving an object, clicking on Specify Orientation will enable you to drag on the three axis to move the object.

Now that the dummy-sensor is in place on the conveyor belt. Assign a collision sensor to it. Be sure to call the collision sensor something appropriate, like `End_limit_switch_right`.

Signal

For the dummy sensor to send its reading to the PLC, a signal should be created.

Todo: Create a boolean signal and connect it with a runtime parameter. The physics object selected should be the `End_limit_switch_right` object. Be sure to define the signal as an `Output`, since it is an output from MCD's perspective.

Note: It is important to give the signal a clear name. For example: `bEnd_limit_switch_right`. Where `b` in he beginning stands for boolean.

The entry sensor is done for now. Later, `End_limit_switch_right` will be mapped to a PLC signal with the exact same name.

Todo: Go through the same steps again and create a second sensor to detect the carrier's exit. Call the new sensor signal `bEnd_limit_switch_left`.

4.4.4 The Physical Layer – PLC Program

A program to control the IMS3 station can be found in the downloads. In this module, a PLC will be simulated to control the station.

TIA Portal

The following figure shows a state machine diagram for the IMS3 station. This state machine is implemented in the program.

Todo: Open the program in TIA Portal.

Note: Outputs of the MCD Simulation are inputs of the `IMS3` function block, and vice versa. Use the same exact names that are used in the variables in the `IMS3` function block in the MCD simulation also. This allows for auto-mapping later.

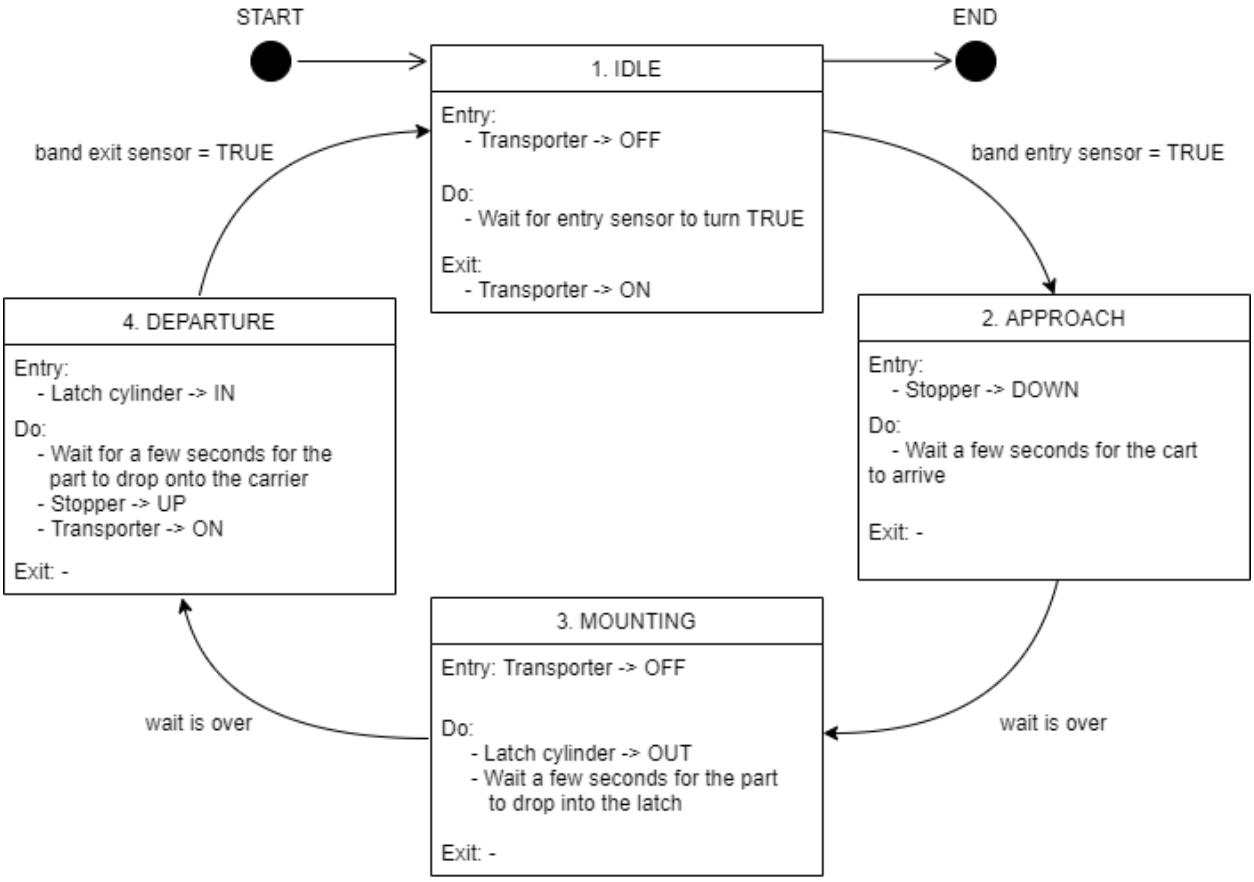


Fig. 4.10: A state machine diagram for the IMS3 station's functionality.

IMS_3

		Name	Data type	Default value	Retain	Accessible from HMI/OPC UA	Writable from HMI/OPC U	Visible ...
1		Input				☐	☐	☐
2		bTransport_band_entry	Bool	false	Non-retain	☑	☑	☑
3		bTransport_band_exit	Bool	false	Non-retain	☑	☑	☑
4		Output				☐	☐	☐
5		bStopper_down	Bool	false	Non-retain	☑	☑	☑
6		bSwing_open	Bool	false	Non-retain	☑	☑	☑
7		bTransporter_on	Bool	false	Non-retain	☑	☑	☑
8		InOut				☐	☐	☐
9		<Add new>				☐	☐	☐
10		Static				☐	☐	☐
11		nState	Int	1	Non-retain	☑	☑	☑
12		bEntry	Bool	false	Non-retain	☑	☑	☑
13		Temp				☐	☐	☐
14		<Add new>				☐	☐	☐
15		Constant				☐	☐	☐
16		<Add new>				☐	☐	☐

Fig. 4.11: Variables of the IMS3 function block

PLCSIM Advanced

Todo: Start PLCSIM Advanced and start a PLC simulation.

Note: Make sure the PLC simulated has the same name as the PLC in the TIA Portal program.

Note: Make sure `PLCSIM Virtual Eth. Adapter` is selected.

In TIA Portal, compile your program and upload it to the simulated PLC.

4.4.5 Controlling the MCD Application using the PLC Program

Todo: Make sure an OPC UA server is configured in the PLC program and connect to it in MCD. When including variables from the PLC program through OPC UA, choose the variables that you need to control the simulated production station.

Todo: In the `Signal Mapping` window in MCD, click the option `Do Auto Mapping`. This will automatically map identically-named signals to each other.

Todo: Run the simulation and watch the PLC's variables in a watch table. The production station should now be controlled through the simulated PLC.

4.5 Preparation: Station IMS4

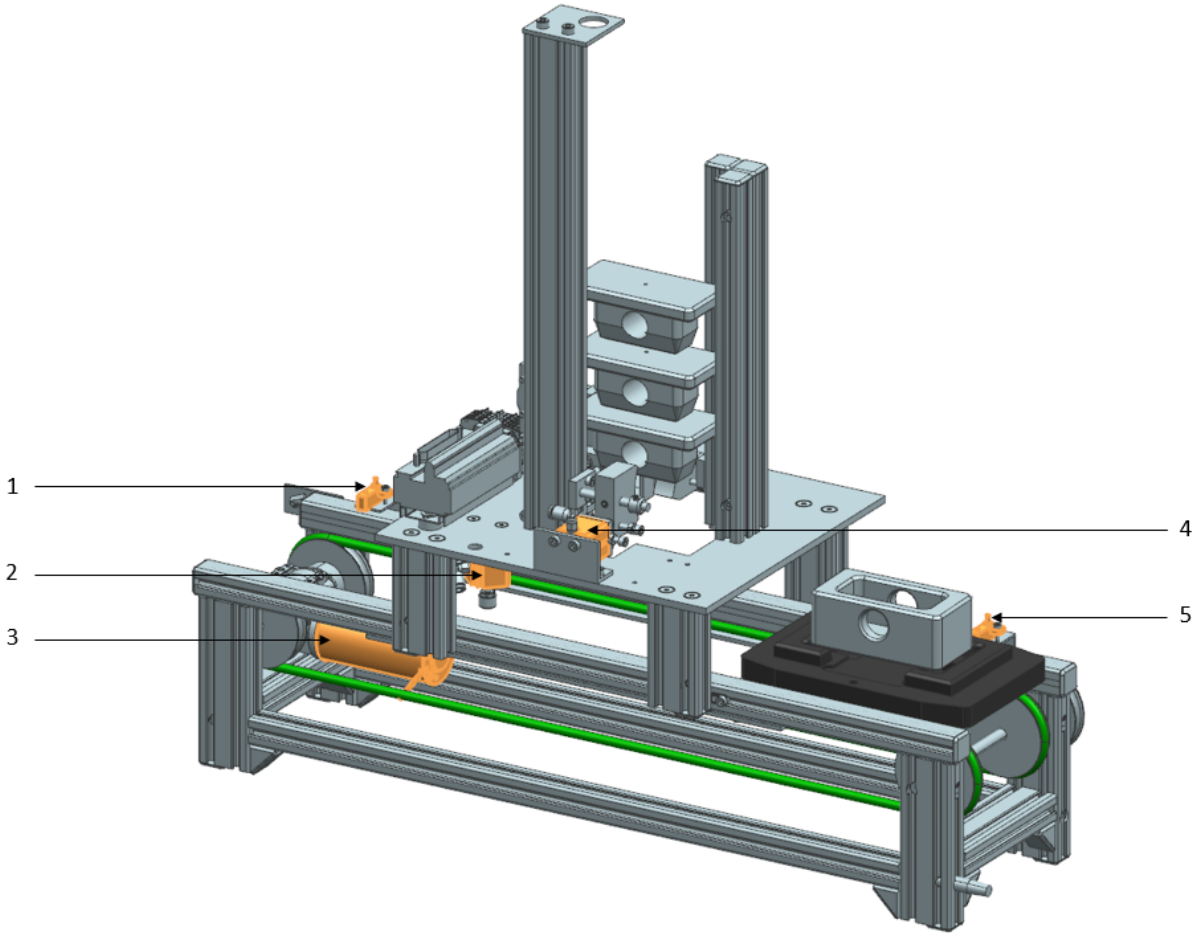
4.5.1 Goal

- To get to know the structure and functionality of the IMS4 station

4.5.2 Task

The following questions/task will help you get familiar with the functionality of the station.

- Identify each of the components highlighted in the following figure and their respective function. Are they sensors or actuators? Would they be defined as inputs or as outputs in a PLC program?
- Describe or outline in a sketch the process of the station.
- Sketch a state machine diagram to describe the functionality of the station.


1
2
3
4
5

Siemens NX MCD

- To which elements should a rigid body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a collision body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a joint be assigned in Siemens NX MCD? What type of joints are required to model this station's behaviour? Is specifying both attachment object and base object required?
- For which objects should a position control be created? (hint: position controls are created to manipulate the position of joints)
- Which signals should be created inside Siemens NX MCD? What is the function of each signal?
- Which operations should be created? Describe the function of each operation.

PLC Program

- Which inputs and outputs should be defined in the PLC program?

4.6 Station IMS4: Practical Assignment

4.6.1 Goal

- To create a physics-based model of the IMS4 station
- To implement the state machine of IMS4 in a TIA Portal program
- To control the physics model with the created state machine program via OPC UA

4.6.2 IMS4 Station Functionality

The purpose of the IMS4 station is to mount the top part to the already-existing base part. The bottom part arrives on a workpiece carrier being transported on a conveyor belt.

To achieve that purpose, the station uses a few sensors and actuators, shown in the following figure.

Note: The IMS4 station has two latch cylinders. One on each side of the top part being separated.

Note: The guiding magazine in the above figures is blinded out, however, the active collision surfaces assigned to it are shown in pink.

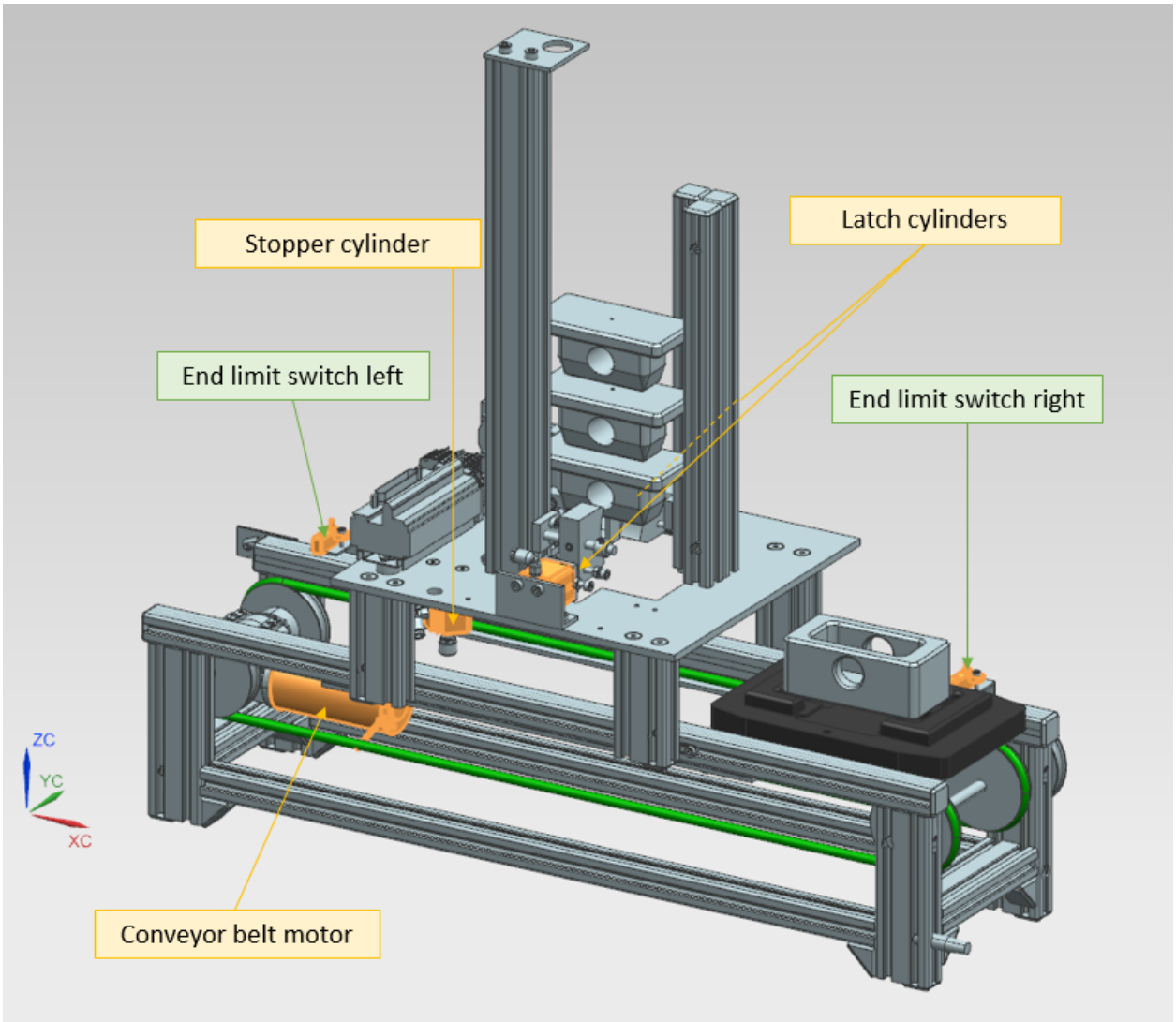


Fig. 4.12: Sensors and actuators on the IMS4 station

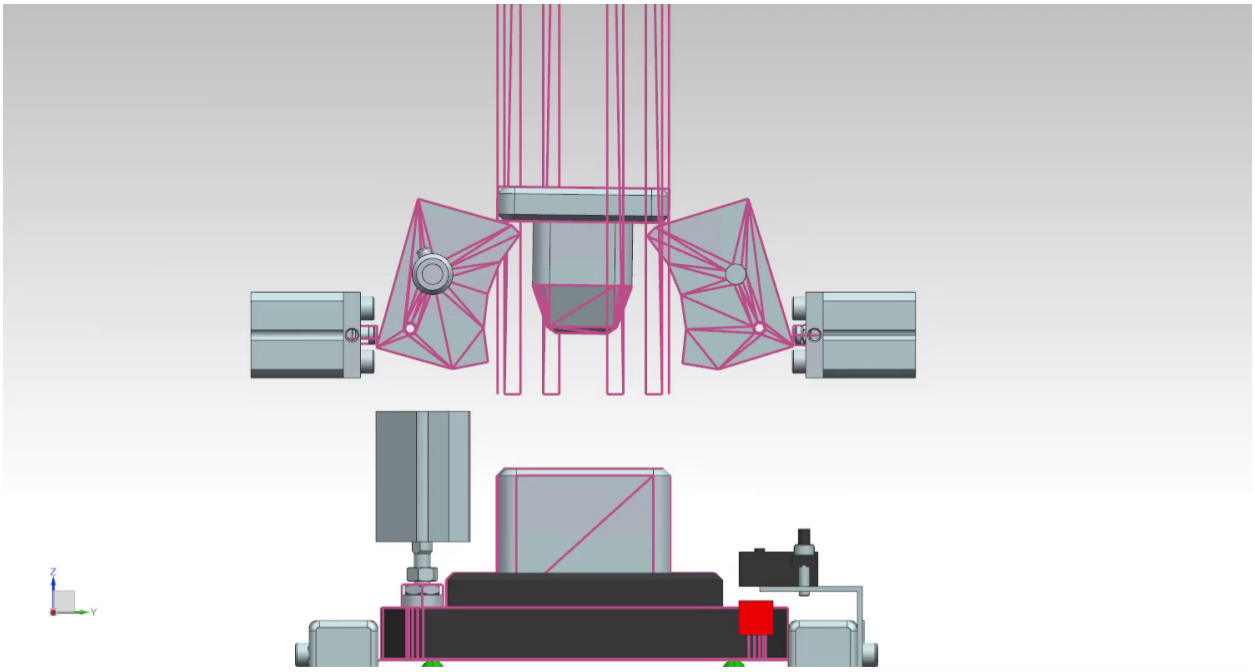


Fig. 4.13: Latch mechanism

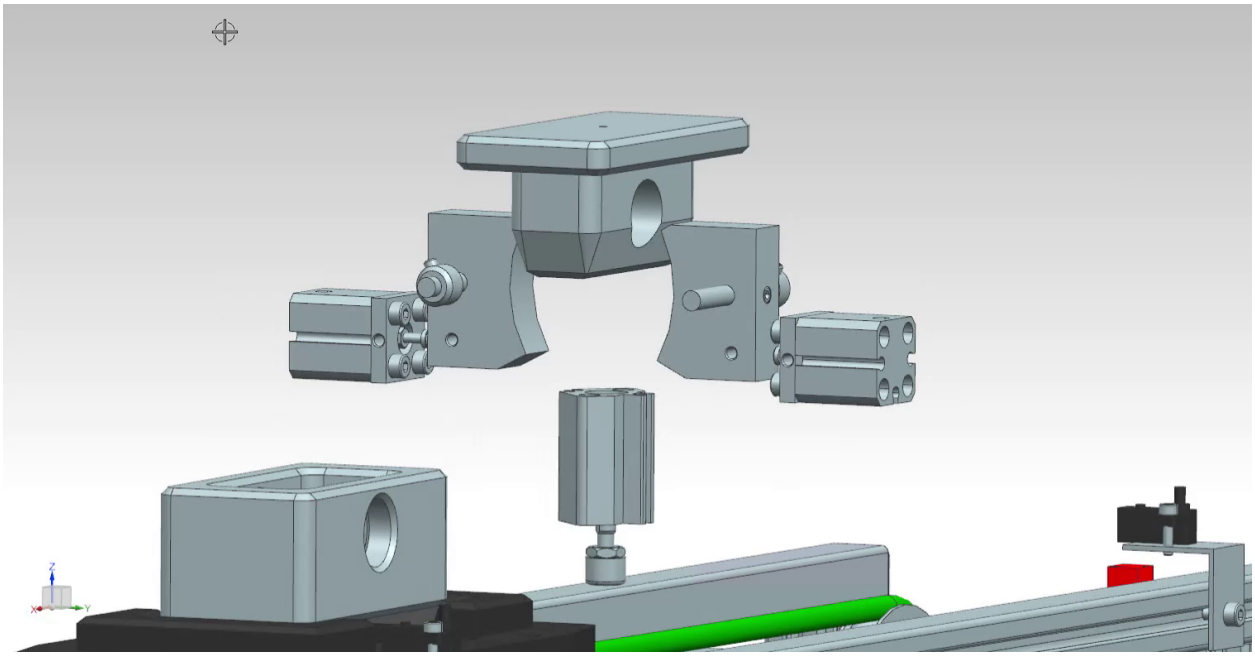


Fig. 4.14: Station's functionality

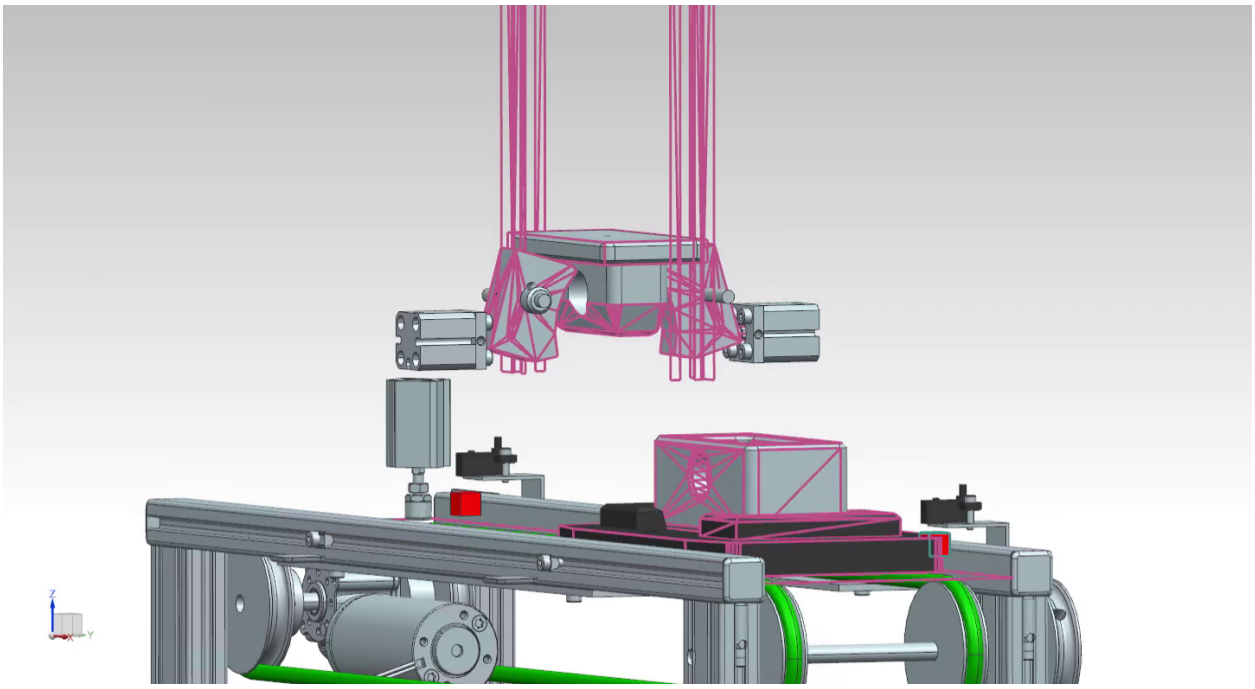


Fig. 4.15: Station's functionality

4.6.3 The Virtual Layer - Mechatronics Concept Designer

Physical properties should be assigned to different parts of the IMS4 station. Open the assembly file called `IMS4.prt` in Siemens NX and navigate to the `Mechatronics Concept Designer` application.

Top part, base part and workpiece carrier

The base part, top part (the part being mounted) and the workpiece carrier already have rigid body and collision bodies assigned to them. These objects are assigned on the single part's level. To see the objects, double click on the respective part in the `Assembly Navigator` window and navigate to the `Physics Navigator` window.

Note: The definition of collisions: Two objects collide when they get in contact with and exert forces onto each other. Collisions are part of the station's functionality. In the context of the simulation, collisions do not refer to accidents.

Rigid and collision bodies

Note: When assigning collision bodies to parts, only sides/areas of the part that will undergo collision during the simulation should be chosen. Avoid defining the entire part as a collision body, because it increases the complexity of the body and unnecessarily slows down the simulation.

Note: When assigning collision bodies to parts, try to choose simple geometry (box, cylinder, sphere, etc.) if complicated geometry (mesh) is not crucial for the physical behaviour of the part.

Todo: Assign collision bodies to the magazine surfaces that store the top parts and along which the top parts fall

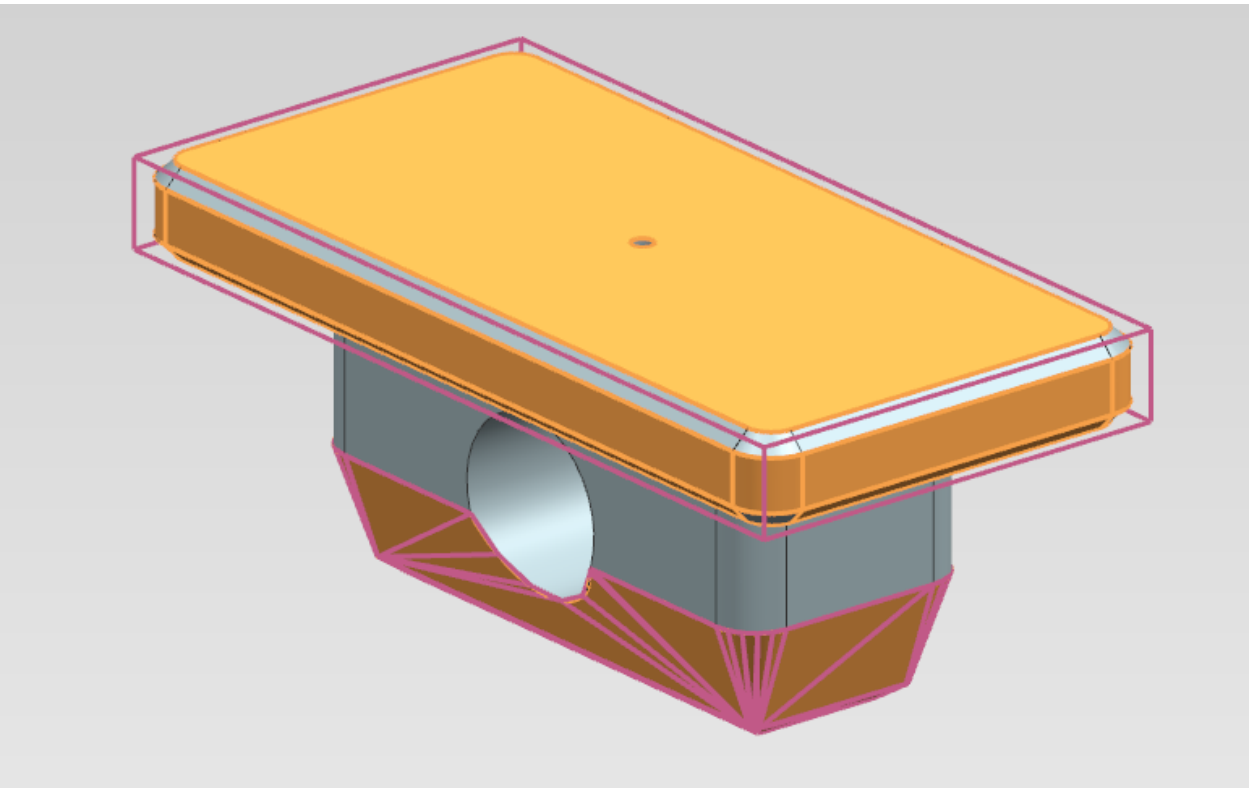


Fig. 4.16: An example of collision body assignment. Collision surfaces are shown in pink. The upper part has a simplified geometry and surfaces on the part that will not undergo collision were not chosen, which reduces the complexity.

during the separation process. These surfaced keep the top parts in place and guide the top parts during the separation process. Assign other collision bodies to the left and right side areas for the conveyor belt. These areas help guide the carrier along the conveyor belt.

Conveyor belt

To simulate a conveyor belt, a rectangular surface will be used as a `Transport surface`. The band itself will be blinded out while the simulation is running.

Todo: Assign a `Transport surface` to the part `200213_TransportflaecheDUMMY`. Do that on the `Entire_Transport_Band` level.

Mounting mechanism - latches

Rigid and collision bodies

The latches on both sides are what hold the parts from falling. They also separate the parts in two steps. Because they will collide with the top parts, rigid and collision bodies should be assigned to them. Because the latches' shape is crucial to their functionality, the option `mesh` should be chosen when assigning collision bodies to both latches.

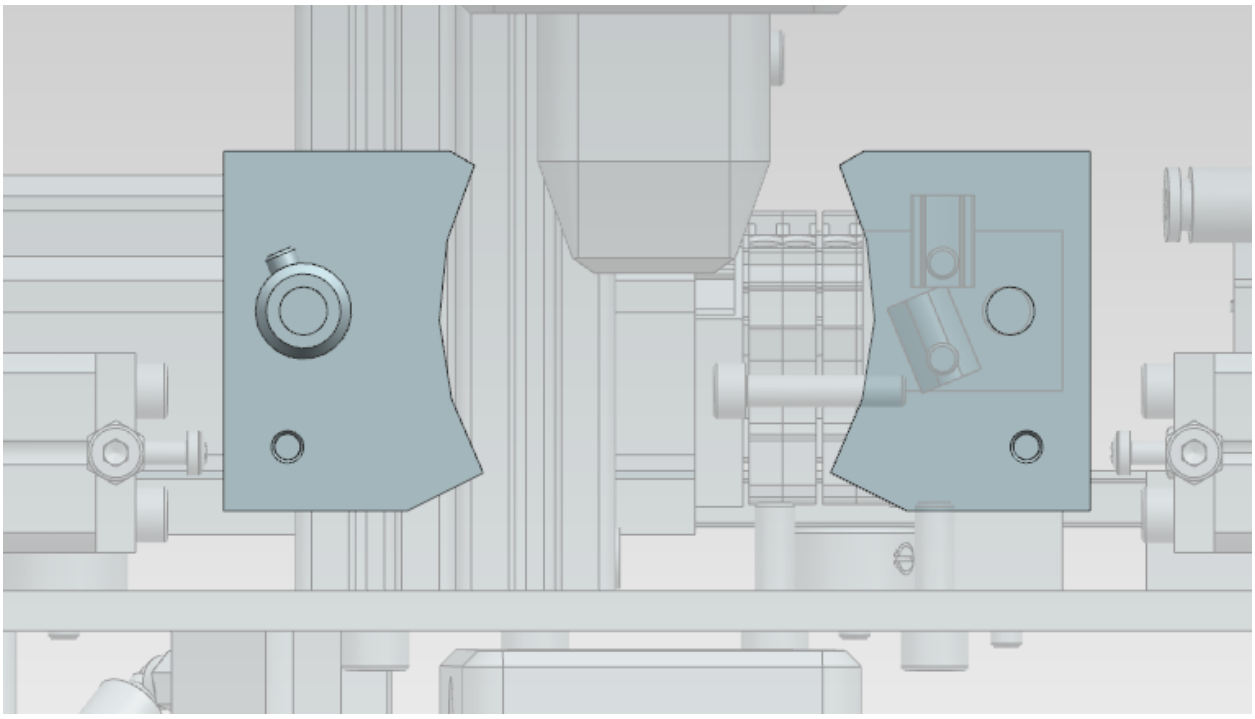


Fig. 4.17: Two latches are used to separate the parts on two steps.

Note: Assign physical properties to the latches on the `LM9681_Montage` assembly level. Can mesh be avoided? Try using a simplified collision geometry for the parts of the latch that actually collide.

Joints

Both latches should be allowed to rotate along the bolts on which they are mounted.

Todo: Assign `hinge joints` to both latches. Specify the axis vector and the anchor point around which the latches will rotate. Because the base of the joint will not be moving, the base object can be left unassigned. Give the joints a clear name.

Angular Spring Joints

Each latch 'latches' back to its original position through an angular spring joint (also known as a simple torsion spring). Assign an `Angular Spring Joint` to each of the latches. Set the spring constant to `20 N.mm/°`, damping to `0.1 N.mm.s/°`, and relaxed position to 30° away from up-straight adjustment. The following figure shows the angular position of one latch. In a relaxed position (if there were no cylinder to collide with the latch), the latch would be at a 30° angle to the vertical axis. With the cylinder touching the latch, the 30° is never reached. The resulting new angle (marked in blue) is not relevant to setting the parameters of the `Angular Spring Joints`.

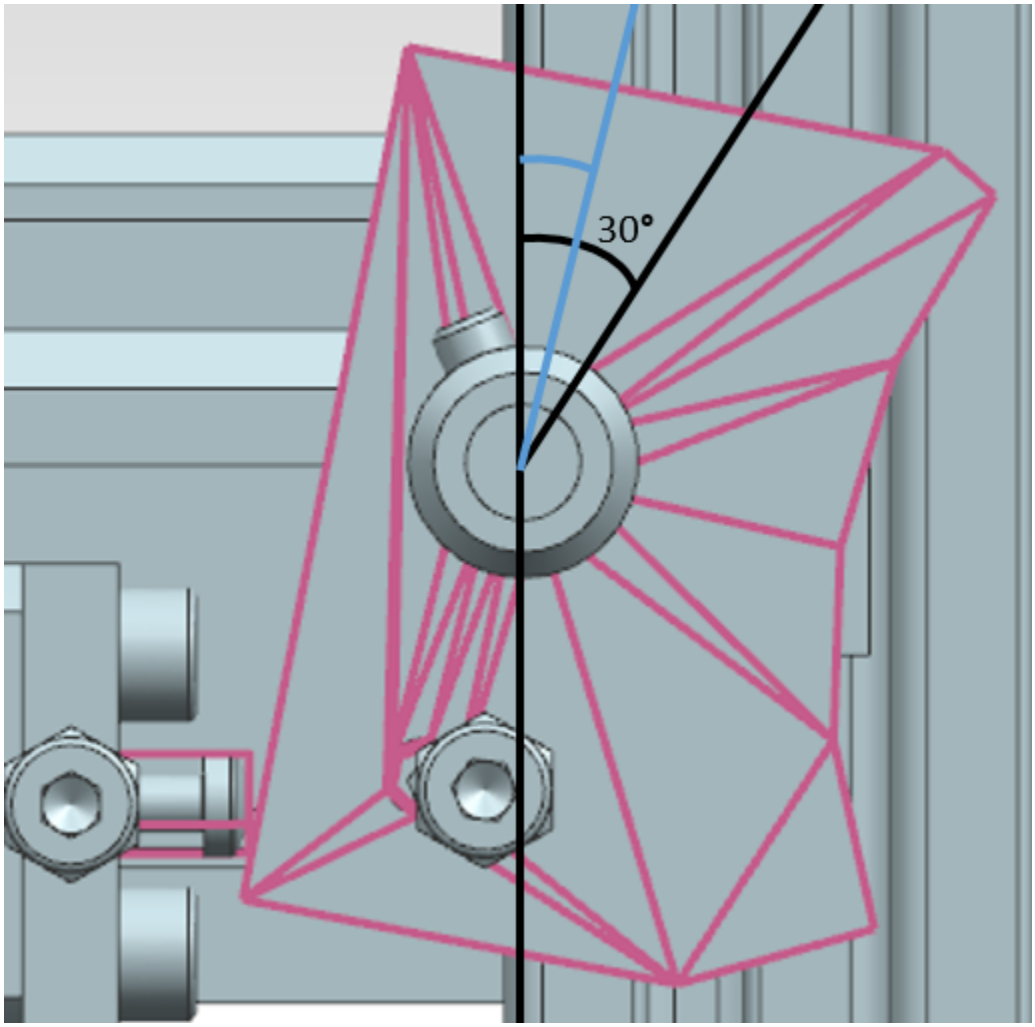


Fig. 4.18: Latch positioning with contact to the cylinder

Note: To check if the given parameters reflect the intented behaviour, one can run the simulation and see how the latches will rest. Always run the simulation on the top most hierarchy level i.e., in the IMS4 assembly and not in a

sub-assembly.

Danger: Running the simulation in a sub-assembly might be handy, but it leads to an OPC UA communication bug. Avoid running the simulation unless its in the top most assembly level. If OPC UA communication is not working, see the troubleshooting section on how to fix it.

The angular position of the latches will be physically controlled by the latch cylinders. As the cylinders phyically push onto the latches, they induce the separating motion for the top parts magazined in the part tunnel. As the cylinders retract back, the latches latch back to their original position under the effect of their torsion springs.

Mounting Mechanism - Cylinders (Latch Cylinders and Stopper Cylinder)

Two cylinders are used to move the latches and create the part-separating motion. When the cylinders are retracted, the latches swing back to their original positions through a torsion spring attached to them (not shown in the CAD model).

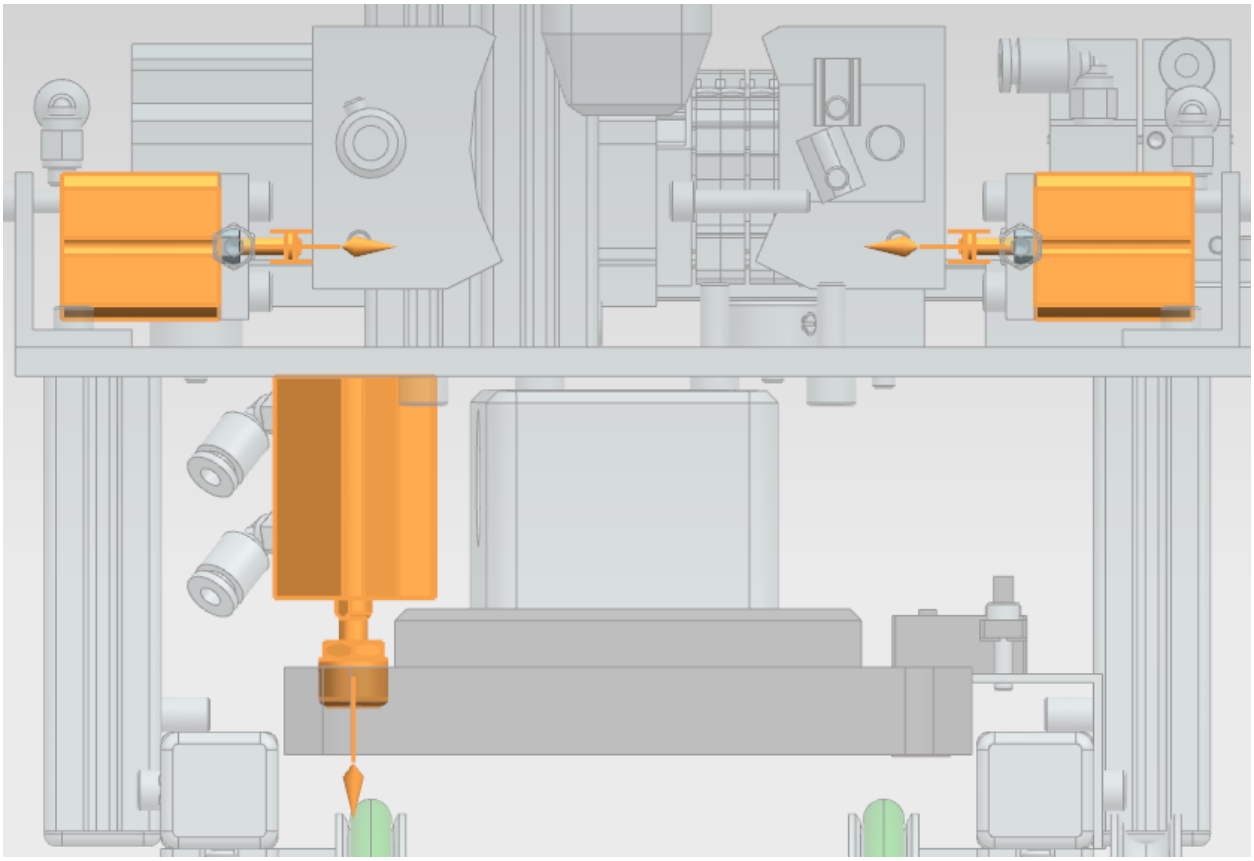


Fig. 4.19: Two cylinders are used to push the two latches. One cylinder is used to stop the workpiece carrier.

Rigid and collision bodies

Todo: Assign rigid and collision bodies to the cylinder parts.

Note: When assigning rigid bodies to part, a few parts can be selected together and defined as one colelctive rigid body. This should be done if those parts will always undergo the same motion together. In the case of the stopper cylinder, the cylinder itself but also the attached gummy object around it can be selected together.

Joints

Each cylinder should be constrained in all directions but one. Since every cylinder will only be either driven out or in, a sliding joint should be assigned to each cylinder.

Todo: Assign sliding joints to each cylinder.

Note: As attachment object, the cylinder itself should be chosen. Becasue the housing of the cylinders will not move during the simulation, the base object can be left unselected. Selecting the housing of the cylinders as the base object is in this case optional.

Note: Make sure to name each of the joints appropriately. For example: `Latch_1_SlidingJoint`, `Latch_2_SlidingJoint`, and `Stopper_SlidingJoint`

Note: Define a lower limit of 0 and an upper limit of 10. This prevents the physical object of wandering beyond those limits.

Position Control

To control the position of the object on the sliding joint, position controls have to be assigned to the sliding joint objects.

Todo: Create position controls and assign them to each of the 3 sliding objects.

Note: Name the position controls appropriately. For example: `Latch_1_PositionControl`, `Latch_2_PositionControl`, and `Stopper_PositionControl`

Signals

In order to, in turn, control those position controls through boolean variables from the PLC program later (the goal), one should create signals in MCD and name them appropriately. For example: `bStopper_down` and `bSwings_open`.

Note: Only one signal can be responsible for triggering both latches. Therefore, do not create two signals for the two latches, but create one signal to control both of them.

Signal Name	Signal Function
bStopper_down	sets the stopper's position control position to 5mm
bSwings_open	sets the latch's position control position to 5mm

Todo: Create 2 boolean signals in MCD for the position controls of the cylinders. Be sure to define the signal as an

Inputs, since they are inputs from MCD's perspective.

Note: Do not link the signals to any runtime variables. These signals will trigger opertations later.

Note: You can create a symbol table for the all of those signals and call it PLCSIM_Advanced. Symbol tables are a good way to arrange and organize signals that are related together. Since these signals will communicate with PLCSIM Advanced, such a name for the symbol table provides clarity.

Operations

Operations are like if-statements in the MCD simulation. They can be used to trigger something in the simulation if an external (or internal) condition is met.

Todo: Create an operation to send the stopper cylinder down. Call it Stopper_down_operation for clarity.

Note: Operations can be found in the Sequence Editor menu on the left.

The selected physics object of the operation is the stopper's position control Stopper_PositionControl. The runtime parameter selected is the position, because this is the parameter to be manipulated by the operation. The new value is inserted in the Value column. Under Condition, the condition object selected is the signal bStopper_down. The if-value of the signal should be set to true.

The equivalent in C++ would be:

```
if(bStopper_down == true) {stopper_sliding_position = 5;}
```

Todo: Create an additional operation that would set the value of Stopper_PositionControl to 0 mm if bStopper_down is false. Call the operation Stopper_up_operation.

Todo: Create 4 more operations to control both latch-cylinders: Two operations to drive the cylinders out and two operations to drive the cylinders in. Use bSwings_open as the boolean condition-variable for these operations.

Todo: Create a boolean signal and use an operation to turn the conveyor belt on and off.

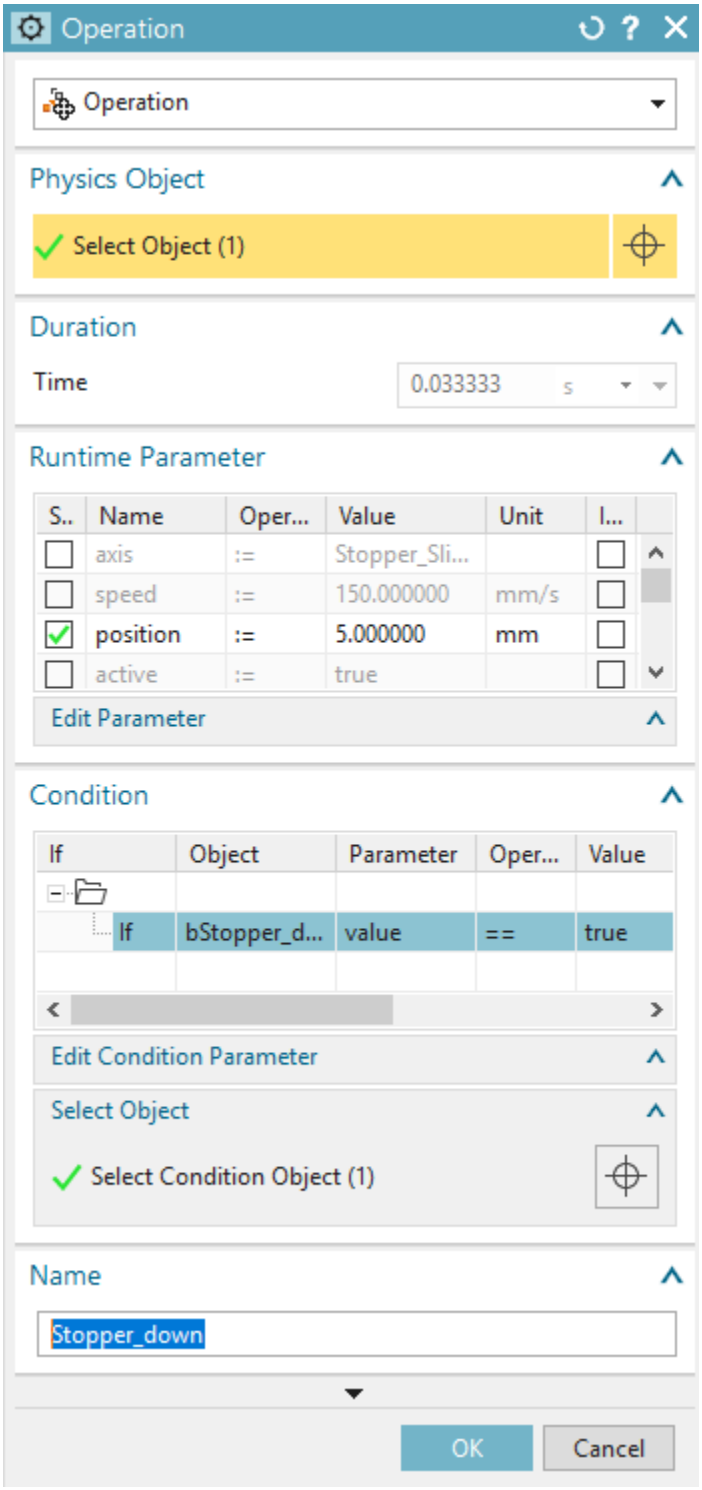


Fig. 4.20: An operation to send the stopper's sliding joint to position `5mm` if the signal `bStopper_down` is true.

Sensing Mechanism - Entry and Exit Sensors

Collision Sensor

The IMS4 has entry and exit sensors at the start and end of the conveyor belt (see section on IMS4 Station Functionality). These two sensors will be simulated as two small cubes with `collision sensor` properties.

Create a new `block` in `mechanical concept` under the `Home` tab. Give the block the dimensions of 10mm x 10mm x 10mm. Under the `Assemblies` tab, click on `Create New` to create a new model. Name the model something appropriate, like `end_limit_switch_dummy` and click `OK`. To select the block just created, navigate to `Part Navigator` on the left-side menu and choose the block that was just created. Now the block is a model and can be seen in the `Assembly Navigator`. Choose `Move Component` under the `Assemblies` tab and move the new block to where the sensor is supposed to be.

Note: When moving an object, clicking on Specify Orientation will enable you to drag on the three axis to move the object.

Now that the dummy-sensor is in place on the conveyor belt. Assign a collision sensor to it. Be sure to call the collision sensor something appropriate, like `End_limit_switch_right`.

Signal

For the dummy sensor to send its reading to the PLC, a signal should be created.

Todo:

> Create a boolean signal and connect it with a runtime parameter. The physics object selected should be the `End_limit_switch_right` object. Be sure to define the signal as an `Output`, since it is an output from MCD's perspective.

Note: It is important to give the signal a clear name. For example: `bEnd_limit_switch_right`. Where `b` in the beginning stands for boolean.

The entry sensor is done for now. Later, `End_limit_switch_right` will be mapped to a PLC signal with the exact same name.

Todo: Go through the same steps again and create a second sensor to detect the carrier's exit. Call the new sensor signal `bEnd_limit_switch_left`.

4.6.4 The Physical Layer – PLC Program

A program to control the IMS4 station can be found in the downloads. In this module, a PLC will be simulated to control the station.

TIA Portal

The following figure shows a state machine diagram for the IMS4 station. This state machine is implemented in the program.

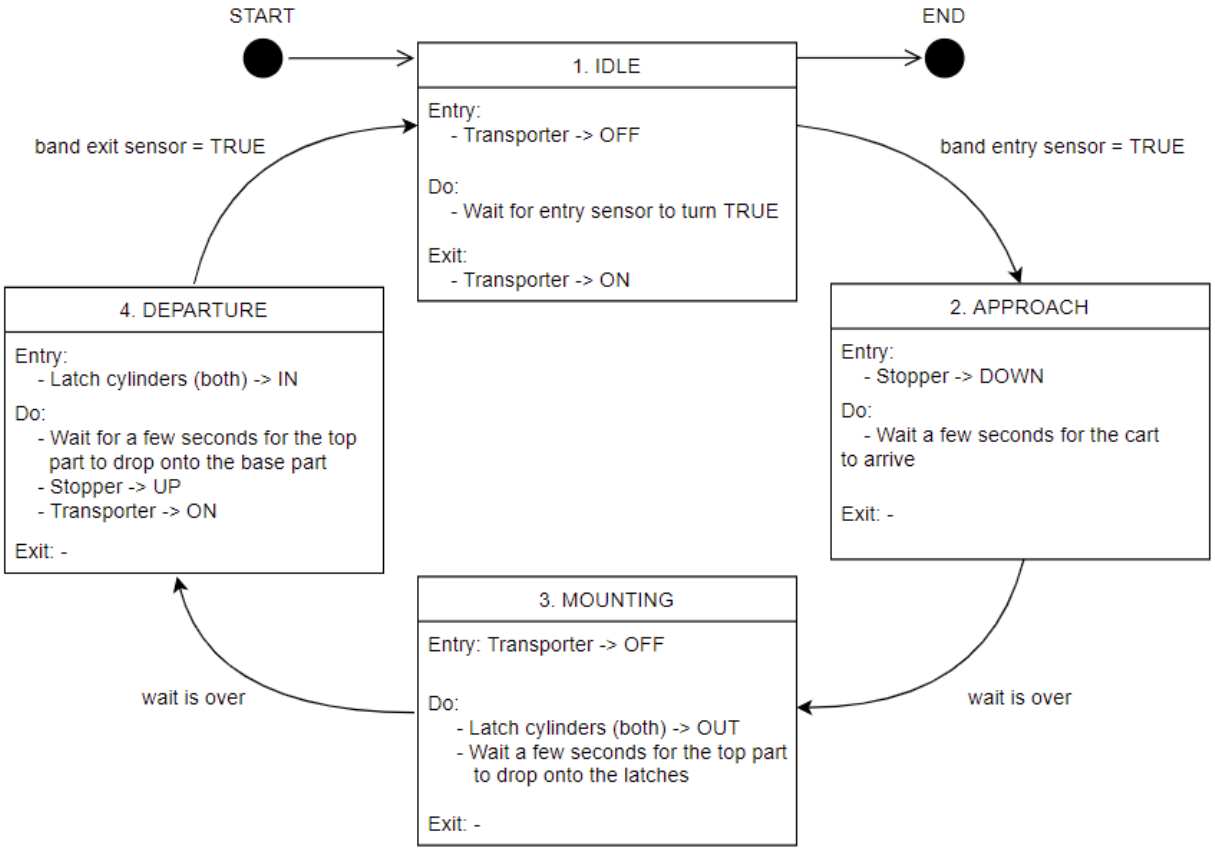


Fig. 4.21: A state machine diagram for the IMS4 station's functionality.

Todo: Open the program in TIA Portal.

Note: Outputs of the MCD Simulation are inputs of the `IMS4` function block, and vice versa. Use the same exact names that are used in the variables in the `IMS4` function block in the MCD simulation also. This allows for auto-mapping later.

IMS_4

	Name	Data type	Default value	Retain	Accessible from HMI/OPC UA	Writable from HMI/OPC UA	Visible..
1	Input				☐	☐	☐
2	bTransport_band_entry	Bool	false	Non-retain	☑	☑	☑
3	bTransport_band_exit	Bool	false	Non-retain	☑	☑	☑
4	Output				☐	☐	☐
5	bStopper_down	Bool	false	Non-retain	☑	☐	☑
6	bSwing_open	Bool	false	Non-retain	☑	☐	☑
7	bTransporter_on	Bool	false	Non-retain	☑	☐	☑
8	InOut				☐	☐	☐
9	<Add new>				☐	☐	☐
10	Static				☐	☐	☐
11	nState	Int	1	Non-retain	☑	☑	☑
12	bEntry	Bool	false	Non-retain	☑	☑	☑
13	Temp				☐	☐	☐
14	<Add new>				☐	☐	☐
15	Constant				☐	☐	☐
16	<Add new>				☐	☐	☐

Fig. 4.22: Variables of the IMS4 function block

PLCSIM Advanced

Todo: Start PLCSIM Advanced and start a PLC simulation.

Note: Make sure the PLC simulated has the same name as the PLC in the TIA Portal program.

Note: Make sure `PLCSIM Virtual Eth. Adapter` is selected.

In TIA Portal, compile your program and upload it to the simulated PLC.

4.6.5 Controlling the MCD Application using the PLC Program

Todo: Make sure an OPC UA server is configured in the PLC program and connect to it in MCD. When including variables from the PLC program through OPC UA, choose the variables that you need to control the simulated production station.

Todo: In the `Signal Mapping` window in MCD, click the option `Do Auto Mapping`. This will automatically map identically-named signals to each other.

Todo: Run the simulation and watch the PLC's variables in a watch table. The production station should now be controlled through the simulated PLC.

4.7 Preparation: Station IMS5

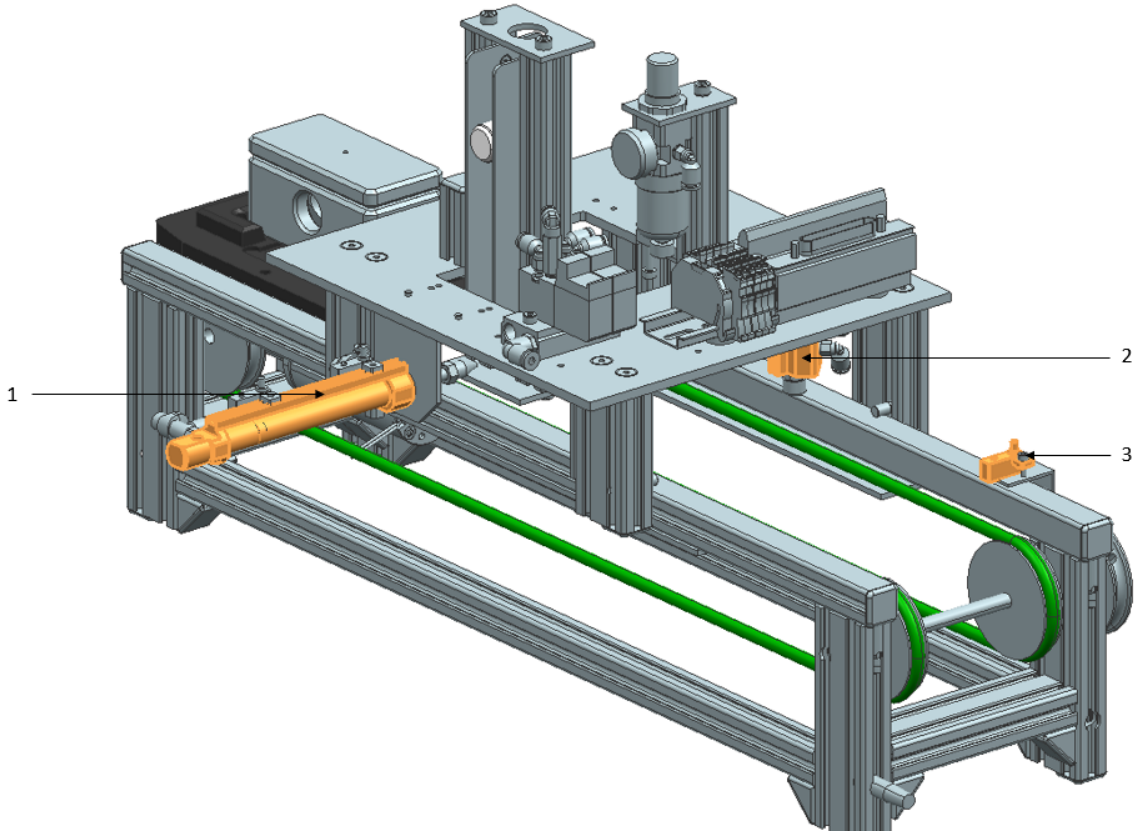
4.7.1 Goal

- To get to know the structure and functionality of the IMS5 station

4.7.2 Task

The following questions/task will help you get familiar with the functionality of the station.

- Identify each of the components highlighted in the following figure and their respective function. Are they sensors or actuators? Would they be defined as inputs or as outputs in a PLC program?



- Describe or outline in a sketch the process of the station.
- Sketch a state machine diagram to describe the functionality of the station.

Siemens NX MCD

- To which elements should a rigid body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a collision body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a joint be assigned in Siemens NX MCD? What type of joints are required to model this station's behaviour? Is specifying both attachment object and base object required?
- For which objects should a position control be created? (hint: position controls are created to manipulate the position of joints)
- Which signals should be created inside Siemens NX MCD? What is the function of each signal?
- Which operations should be created? Describe the function of each operation.

PLC Program

- Which inputs and outputs should be defined in the PLC program?

4.8 Station IMS5: Practical Assignment

4.8.1 Goal

- To create a physics-based model of the IMS5 station
- To implement the state machine of IMS5 in a TIA Portal program
- To control the physics model with the created state machine program via OPC UA

4.8.2 IMS5 Station Functionality

The purpose of the IMS5 station is to insert a bolt that connects the top part and the bottom part together. Both parts arrive on the workpiece carrier that is transported on a conveyor belt.

The station uses a few sensors and actuators shown in the following figure.

Note: A left end limit switch exists on the left end of the conveyor belt

---

Note: The guiding magazine in the above animation is blinded out, however, the active collision surfaces assigned to it is shown in pink.

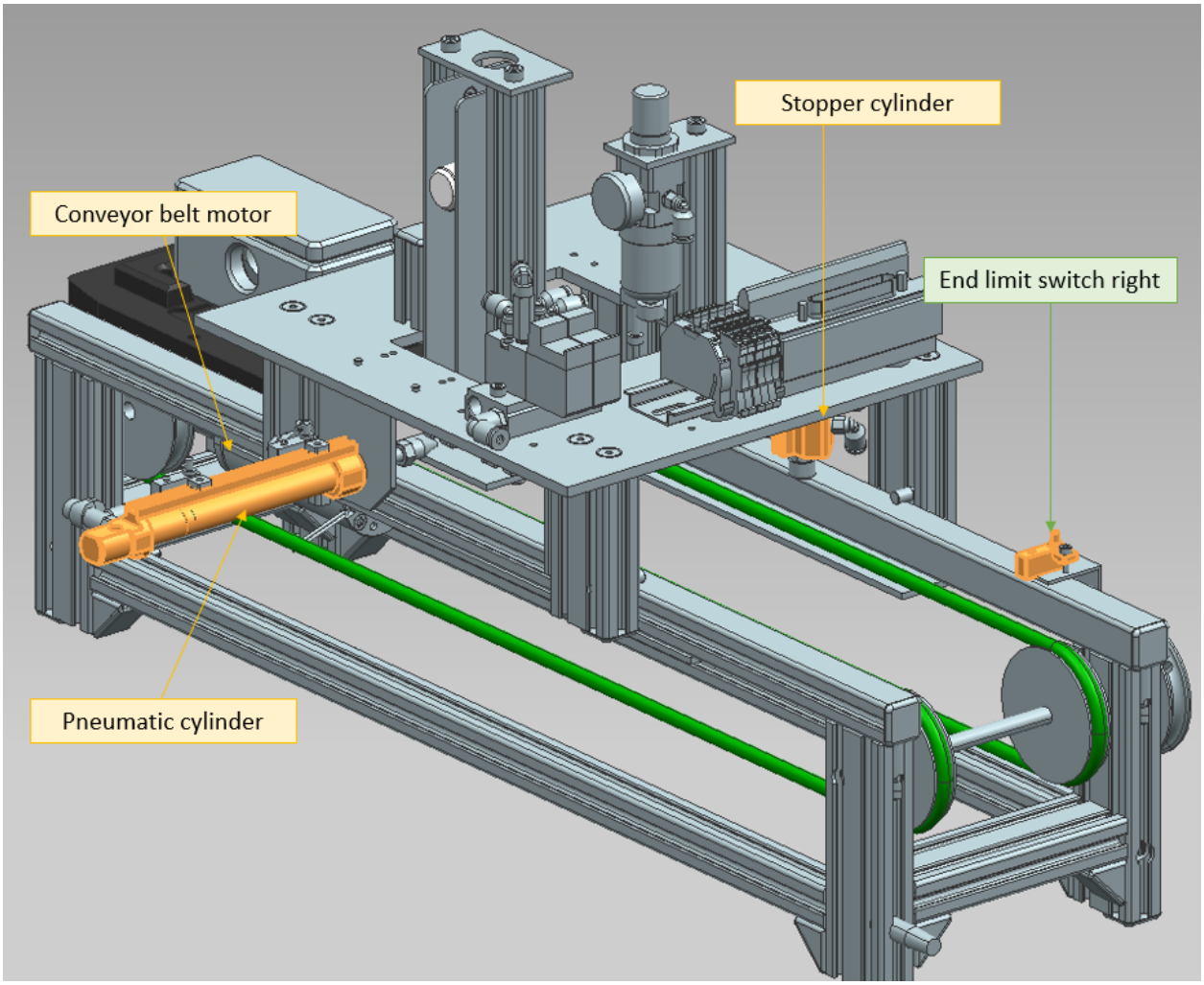


Fig. 4.23: Sensors and actuators on the IMS5 station

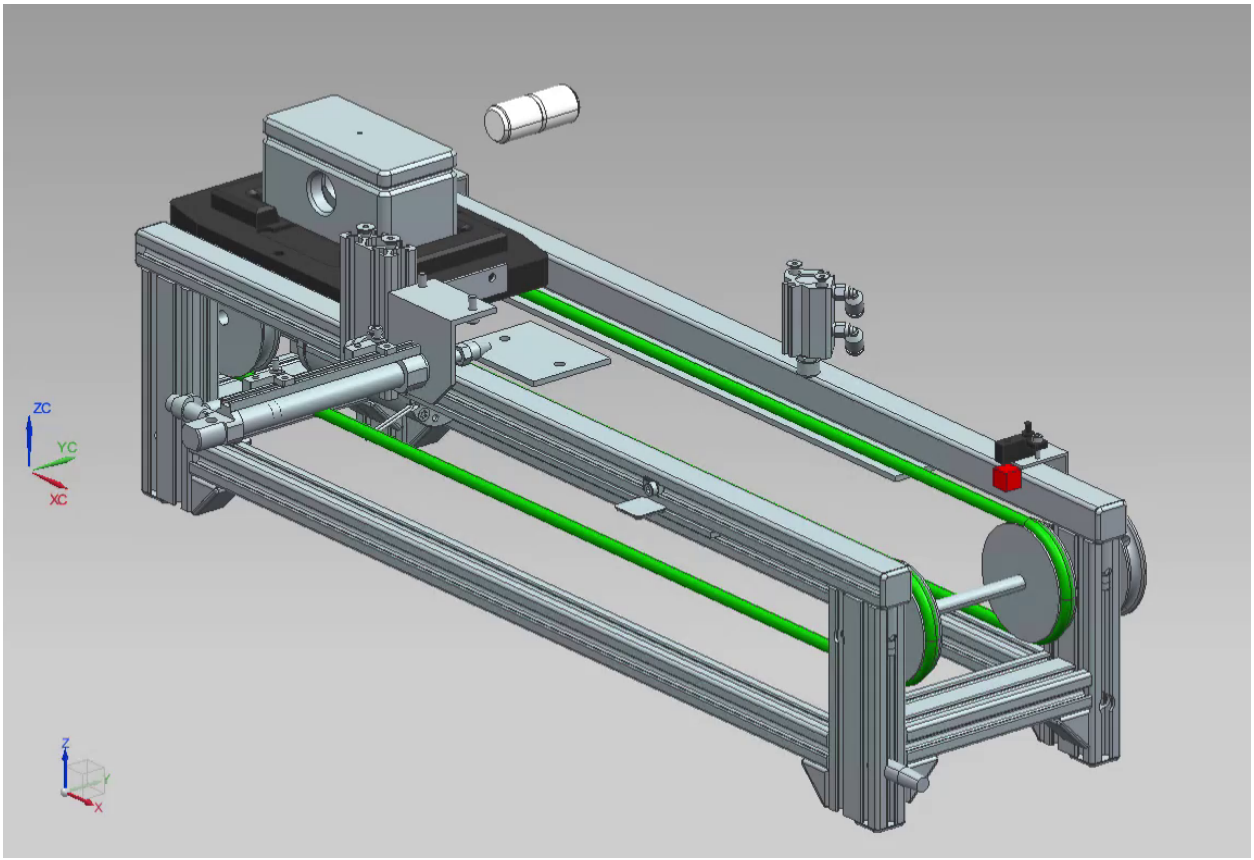


Fig. 4.24: IMS5 mechanism

4.8.3 The Virtual Layer - Mechatronics Concept Designer

Physical properties should be assigned to different parts of the IMS5 station. Open the assembly file called `ASSEMBLY_IMS5.prt` in Siemens NX and navigate to the `Mechatronics Concept Designer` application.

Top part, base part, bolt, and workpiece carrier

The base part, top part, bolt (the part being mounted) and the workpiece carrier already have rigid body and collision bodies assigned to them. These objects are assigned on the single part's level. To see the objects, double click on the respective part in the `Assembly Navigator` window and navigate to the `Physics Navigator` window.

Note: The definition of collisions: Two objects collide when they get in contact with and exert forces onto each other. Collisions are part of the station's functionality. In the context of the simulation, collisions do not refer to accidents.

Rigid and collision bodies

Note: When assigning collision bodies to parts, only sides/areas of the part that will undergo collision during the simulation should be chosen. Avoid defining the entire part as a collision body, because it increases the complexity of the body and unnecessarily slows down the simulation.

Note: When assigning collision bodies to parts, try to choose simple geometry (box, cylinder, sphere, etc.) if complicated geometry (mesh) is not crucial for the physical behaviour of the part.

Todo: Assign collision bodies to the magazine surfaces that store the bolts and along which the bolts fall during the separation process. These surfaced keep the bolts in place and guide the bolts during the separation process. Assign other collision bodies to the left and right side areas for the conveyor belt. These areas help guide the carrier along the conveyor belt.

Conveyor belt

To simulate a conveyor belt, a rectangular surface will be used as a `transport surface`. The band itself will be blinded out while the simulation is running.

Todo: Assign a `transport surface` to the part `200213_TransportflaecheDUMMY`. Do that on the `Entire_Transport_Band` level.

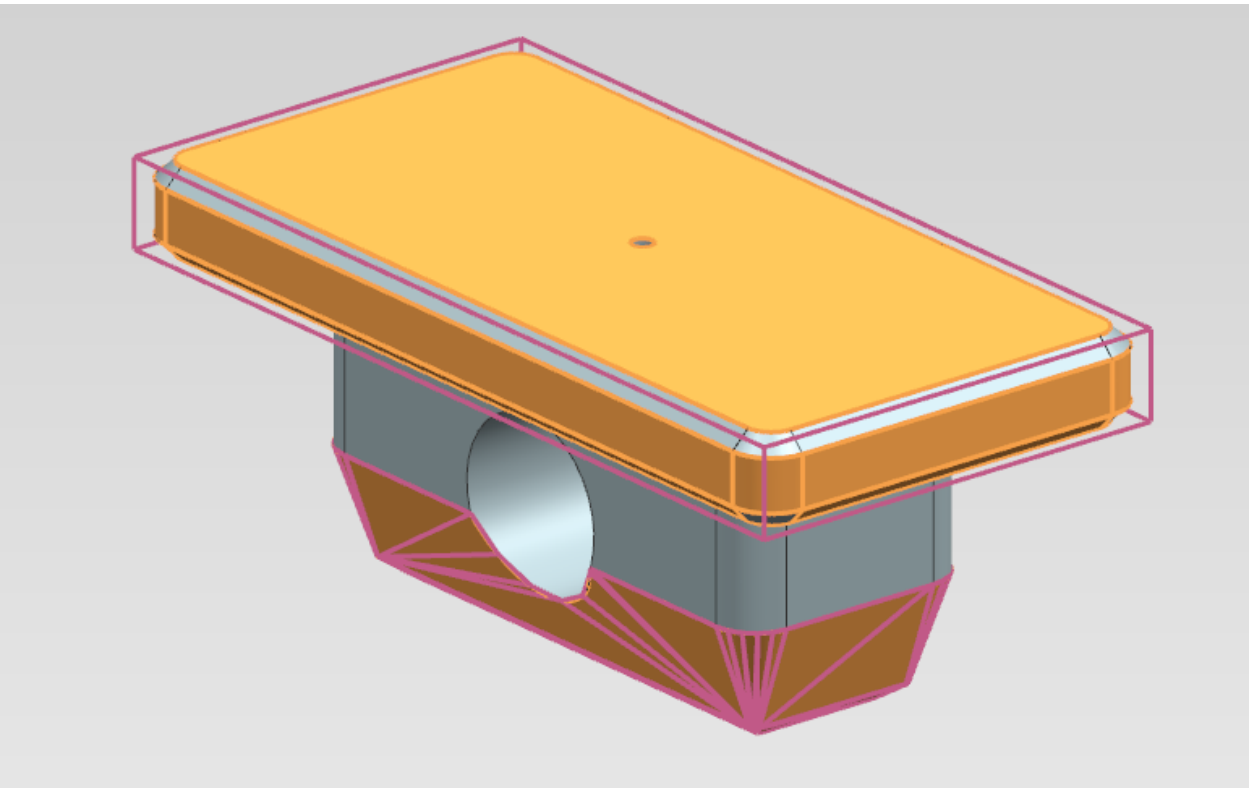


Fig. 4.25: An example of collision body assignment. Collision surfaces are shown in pink. The upper part has a simplified geometry and surfaces on the part that will not undergo collision were not chosen, which reduces the complexity.

Mounting Mechanism - Cylinders (Mounting Cylinder and Stopper Cylinder)

A pneumatic cylinder is used to push the bolt into the other pieces. Another cylinder is used as a stopper for the workpiece carrier.

Rigid and collision bodies

Todo: Assign rigid and collision bodies to the appropriate cylinder parts.

Note: When assigning rigid bodies to part, a few parts can be selected together and defined as one colelctive rigid body. This should be done if those parts will always undergo the same motion together. In the case of the stopper cylinder, the cylinder itself but also the attached gummy object around it can be selected together.

Joints

Each cylinder rod should be constrained in all directions but one. Since every cylinder will only be either driven out or in, a sliding joint should be assigned to each cylinder rod.

Todo: Assign sliding joints to each cylinder.

Note: As attachment object, the cylinder's rod should be chosen. Becasue the housing of the cylinders will not move during the simulation, the base object can be left unselected. Selecting the housing of the cylinders as the base object is in this case optional.

Note: Make sure to name each of the joints appropriately. For example: `Cylinder_SlidingJoint`, and `Stopper_SlidingJoint`

Note: Define a lower limit of 0 and an upper limit of 10. This prevents the physical object of wandering beyond those limits.

Position Control

To control the position of the object on the sliding joint, position controls have to be assigned to the sliding joint objects.

Todo: Create position controls and assign them to each of the sliding objects.

Note: Name the position controls appropriately. For example: `Cylinder_PositionControl` and `Stopper_PositionControl`

Signals

In order to, in turn, control those position controls through boolean variables from the PLC program later (the goal), one should create signals in MCD and name them appropriately. For example: `bStopper_down` and `bCylinder_out`.

Signal Name	Signal Function
bStopper_down	sets the stopper's position control position to 5mm
bCylinder_out	sets the mounting cylinder's position control position to 7mm

Todo: Create 2 boolean signals in MCD for the position controls of the cylinders. e sure to define the signals as `Inputs`, since they are inputs from MCD's perspective.

Note: Do not link the signals to any runtime variables. These signals will trigger opertations later.

Note: You can create a symbol table for the all of those signals and call it `PLCSIM_Advanced`. Symbol tables are a good way to arrange and organize signals that are related together. Since these signals will communicate with PLCSIM Advanced, such a name for the symbol table provides clarity.

Operations

Operations are like if-statements in the MCD simulation. They can be used to trigger something in the simulation if an external (or internal) condition is met.

Todo: Create an operation to send the stopper cylinder down. Call it `Stopper_down_operation` for clarity.

Note: Operations can be found in the `Sequence Editor` menu on the left.

The selected physics object of the operation is the stopper's position control `Stopper_PositionControl`. The runtime parameter selected is the position; this is the parameter to be manipulated by the operation. The new value is inserted in the `Value` column. Under `Condition`, the condition object selected is the signal `bStopper_down`. The if-value of the signal should be set to `true`.

The equivalent in C++ would be:

```
if(bStopper_down == true) {stopper_sliding_position = 5;}
```

Todo: Create an additional operation that would set the value of `Stopper_PositionControl` to `0 mm` if `bStopper_down` is false. Call the operation `Stopper_up_operation`.

Todo: Create two more operations to control the mounting cylinder: One operations to drive the cylinder out and one operation to drive the cylinder in. Use `bCylinder_out` as the boolean condition-variable for these operations.

Todo: Create a boolean signal and use an operation to turn the conveyor belt on and off.

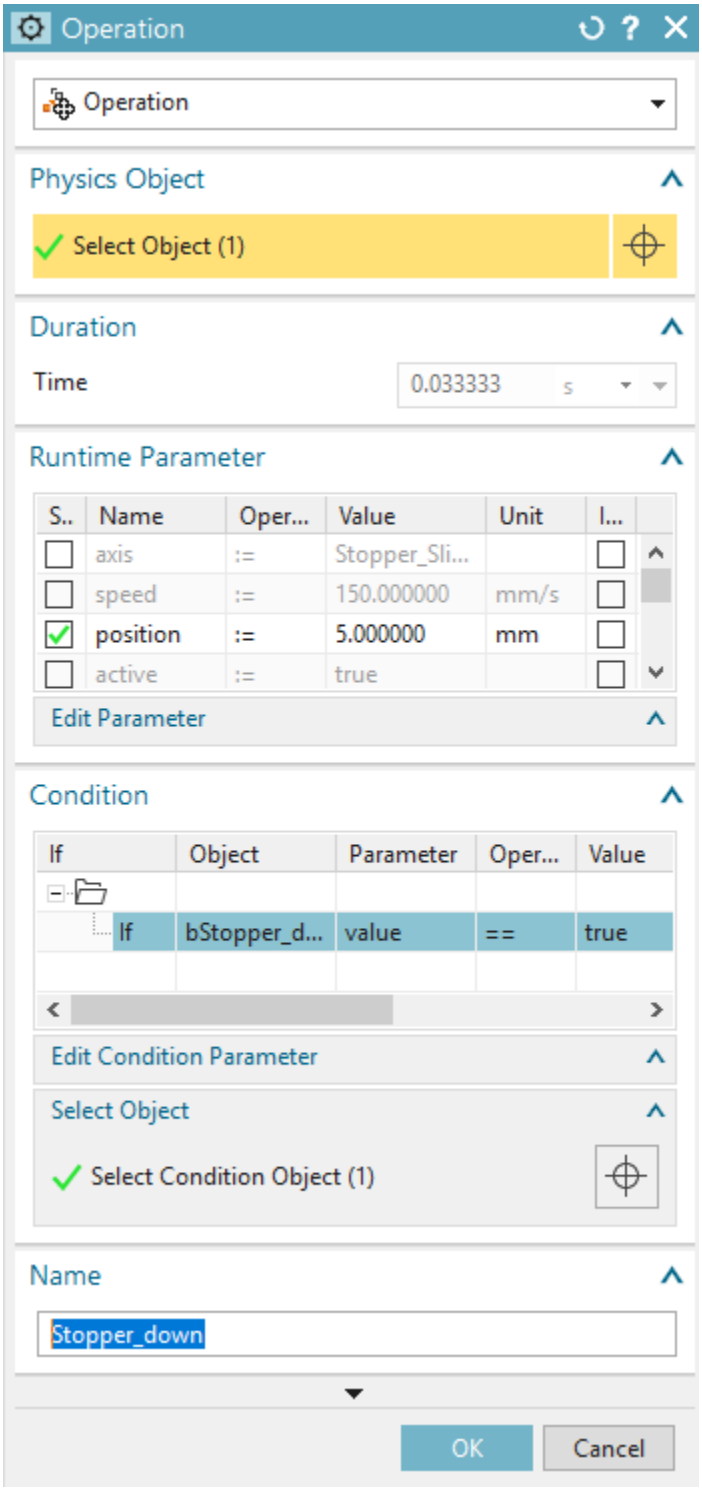


Fig. 4.26: An operation to send the stopper's sliding joint to position `5mm` if the signal `bStopper_down` is true.

Sensing Mechanism - Entry and Exit Sensors

Collision Sensor

The IMS5 has entry and exit sensors at the start and end of the conveyor belt (see section on IMS5 Station Functionality). These two sensors will be simulated as two small cubes with `collision sensor` properties.

Create a new `block` in `mechanical concept` under the `Home` tab. Give the block the dimensions of 10mm x 10mm x 10mm. Under the `Assemblies` tab, click on `Create New` to create a new model. Name the model something appropriate, like `end_limit_switch_dummy` and click `OK`. To select the block just created, navigate to `Part Navigator` on the left-side menu and choose the block that was just created. Now the block is a model and can be seen in the `Assembly Navigator`. Choose `Move Component` under the `Assemblies` tab and move the new block to where the sensor is supposed to be.

Note: When moving an object, clicking on Specify Orientation will enable you to drag on the three axis to move the object.

Now that the dummy-sensor is in place on the conveyor belt. Assign a collision sensor to it. Be sure to call the collision sensor something appropriate, like `End_limit_switch_right`.

Signal

For the dummy sensor to send its reading to the PLC, a signal should be created.

Todo: Create a boolean signal and connect it with a runtime parameter. The physics object selected should be the `End_limit_switch_right` object. Be sure to define the signal as an `Output`, since it is an output from MCD's perspective.

Note: It is important to give the signal a clear name. For example: `bEnd_limit_switch_right`. Where `b` in the beginning stands for boolean.

The entry sensor is done for now. Later, `End_limit_switch_right` will be mapped to a PLC signal with the exact same name.

Todo: Go through the same steps again and create a second sensor to detect the carrier's exit. Call the new sensor signal `bEnd_limit_switch_left`.

4.8.4 The Physical Layer – PLC Program

A program to control the IMS5 station can be found in the downloads. In this module, a PLC will be simulated to control the station.

TIA Portal

The following figure shows a state machine diagram for the IMS5 station. This state machine is implemented in the program.

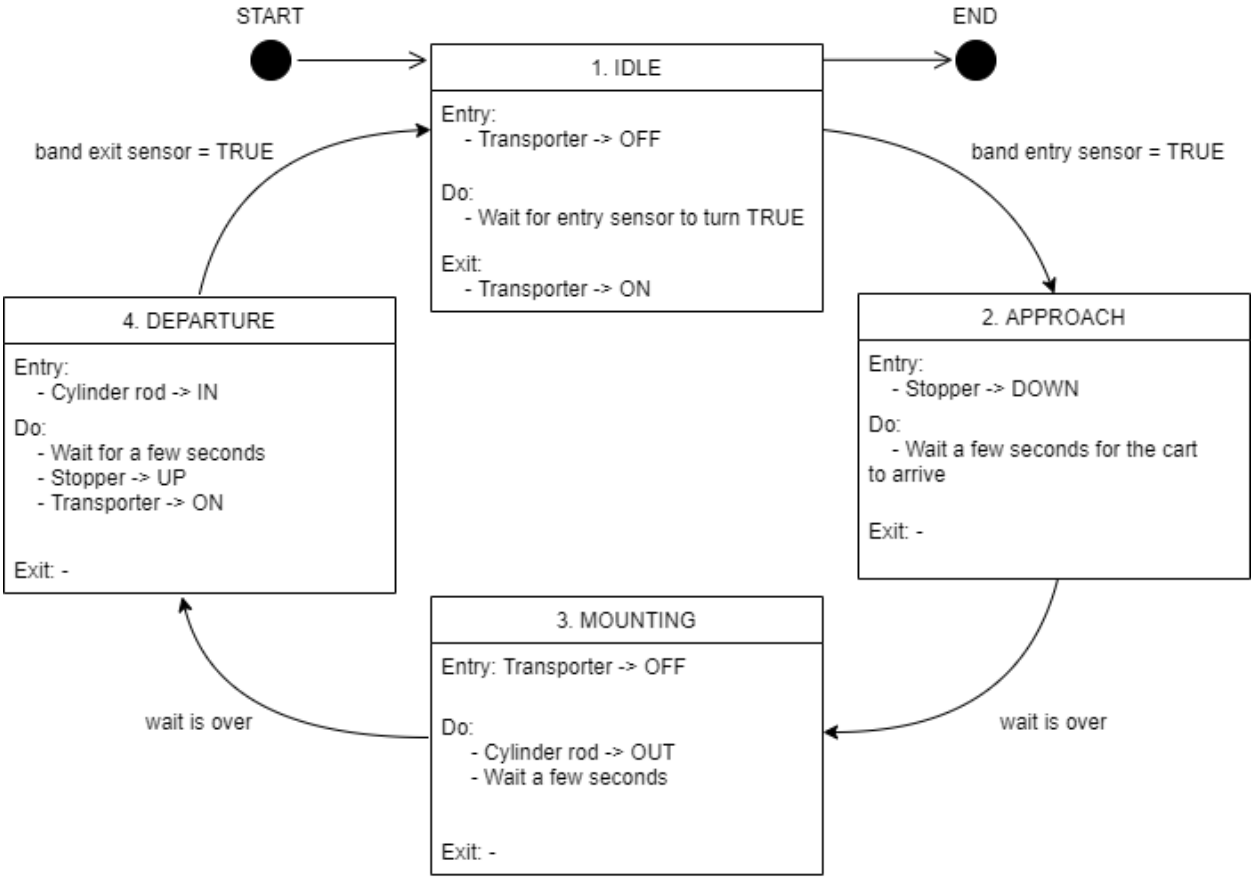


Fig. 4.27: A state machine diagram for the IMS5 station's functionality.

Todo: Open the program in TIA Portal.

Note: Outputs of the MCD Simulation are inputs of the `IMS5` function block, and vice versa. Use the same exact names that are used in the variables in the `IMS5` function block in the MCD simulation also. This allows for auto-mapping later.

IMS_5

	Name	Data type	Default value	Retain	Accessible from HMI/OPC UA	Writable from HMI/OPC U	Visible in ...
1	Input				☐	☐	☐
2	bTransport_band_entry	Bool	false	Non-retain	☑	☑	☑
3	bTransport_band_exit	Bool	false	Non-retain	☑	☑	☑
4	Output				☐	☐	☐
5	bStopper_down	Bool	false	Non-retain	☑	☑	☑
6	bSwing_open	Bool	false	Non-retain	☑	☑	☑
7	bTransporter_on	Bool	false	Non-retain	☑	☑	☑
8	InOut				☐	☐	☐
9	<Add new>				☐	☐	☐
10	Static				☐	☐	☐
11	nState	Int	1	Non-retain	☑	☑	☑
12	bEntry	Bool	false	Non-retain	☑	☑	☑
13	Temp				☐	☐	☐
14	<Add new>				☐	☐	☐
15	Constant				☐	☐	☐
16	<Add new>				☐	☐	☐

Fig. 4.28: Variables of the IMS5 function block

PLCSIM Advanced

Todo: Start PLCSIM Advanced and start a PLC simulation.

Note: Make sure the PLC simulated has the same name as the PLC in the TIA Portal program.

Note: Make sure `PLCSIM Virtual Eth. Adapter` is selected.

In TIA Portal, compile your program and upload it to the simulated PLC.

4.8.5 Controlling the MCD Application using the PLC Program

Todo: Make sure an OPC UA server is configured in the PLC program and connect to it in MCD. When including variables from the PLC program through OPC UA, choose the variables that you need to control the simulated production station.

Todo: In the `Signal Mapping` window in MCD, click the option `Do Auto Mapping`. This will automatically map identically-named signals to each other.

Todo: Run the simulation and watch the PLC's variables in a watch table. The production station should now be controlled through the simulated PLC.

4.9 Preparation: Station IMS7

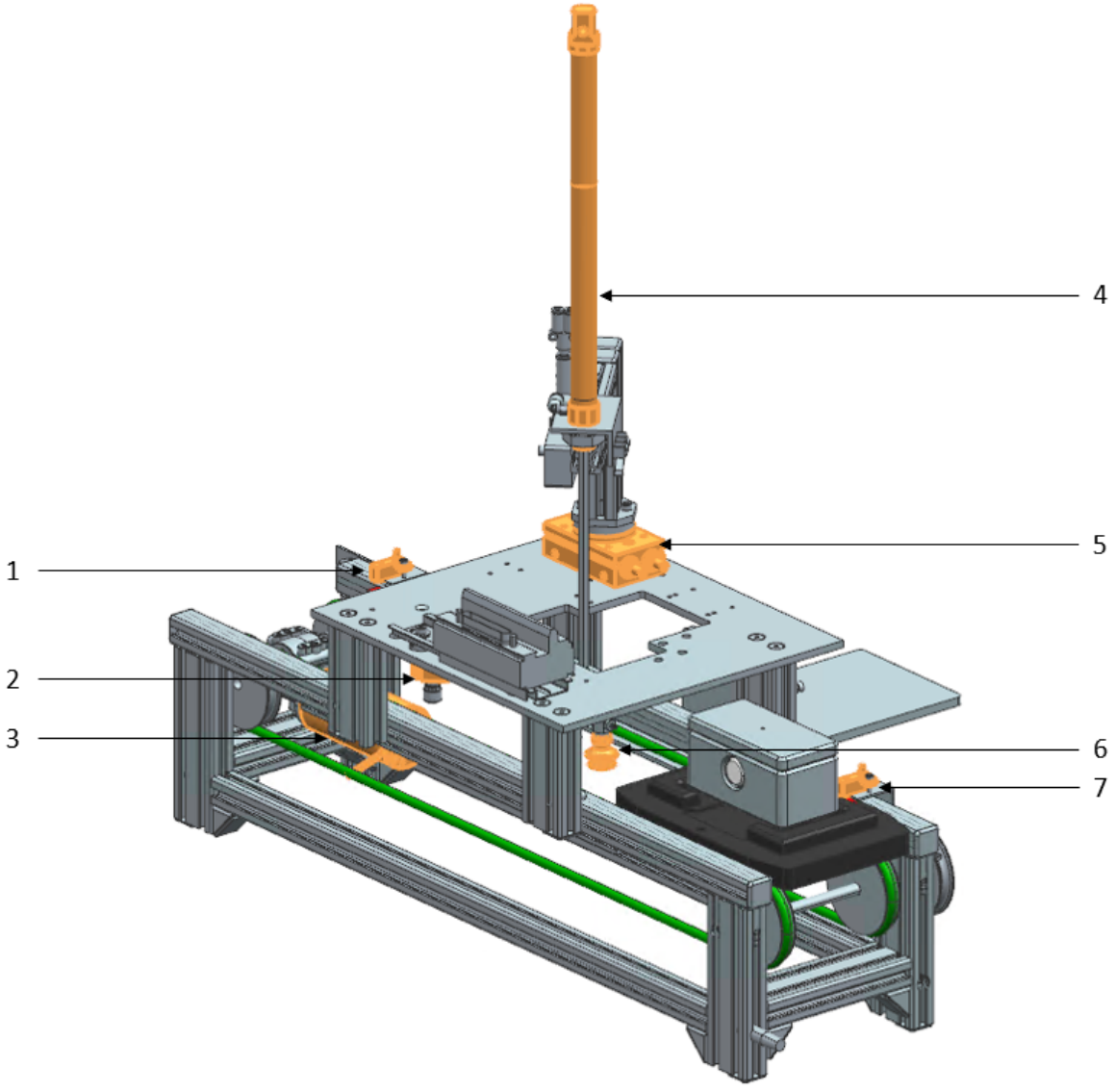
4.9.1 Goal

- To get to know the structure and functionality of the IMS7 station

4.9.2 Task

The following questions/task will help you get familiar with the functionality of the station.

- Identify each of the components highlighted in the following figure and their respective function. Are they sensors or actuators? Would they be defined as inputs or as outputs in a PLC program?



- Describe or outline in a sketch the process of the station.
- Sketch a state machine diagram to describe the functionality of the station.

Siemens NX MCD

- To which elements should a rigid body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a collision body be assigned in Siemens NX MCD? (some elements are not highlighted in the figure above)
- To which elements should a joint be assigned in Siemens NX MCD? What type of joints are required to model this station's behaviour? Is specifying both attachment object and base object required?
- For which objects should a position control be created? (hint: position controls are created to manipulate the position of joints)
- Which signals should be created inside Siemens NX MCD? What is the function of each signal?
- Which operations should be created? Describe the function of each operation.

PLC Program

- Which inputs and outputs should be defined in the PLC program?

4.10 Station IMS7: Practical Assignment

4.10.1 Goal

- To create a physics-based model of the IMS7 station
- To implement the state machine of IMS7 in a TIA Portal program
- To control the physics model with the created state machine program via OPC UA

4.10.2 IMS7 Station Functionality

The purpose of the IMS7 station is to pick-and-place the end-product from the assembly line onto an outside platform. It does so by using a vacuum gripper and a rotating arm to lift the end-product and re-place it on the platform.

The station has a few sensors and actuators, shown in the following figure.

4.10.3 The Virtual Layer - Mechatronics Concept Designer

Physical properties should be assigned to different parts of the IMS7 station. Open the assembly file called `ASSEMBLY_IMS7.prt` in Siemens NX and navigate to the `Mechatronics Concept Designer` application.

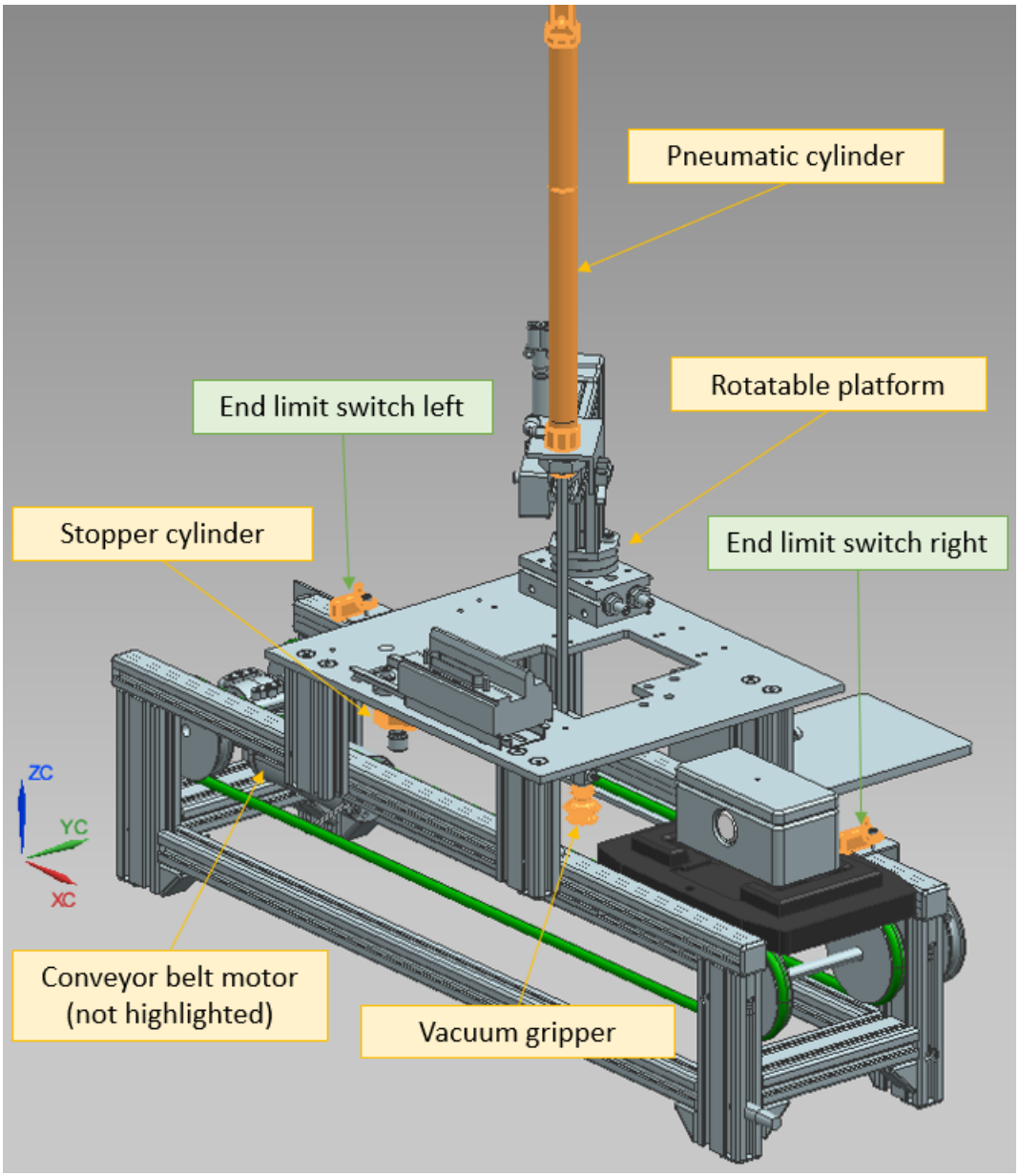


Fig. 4.29: Sensors and actuators on the IMS7 station

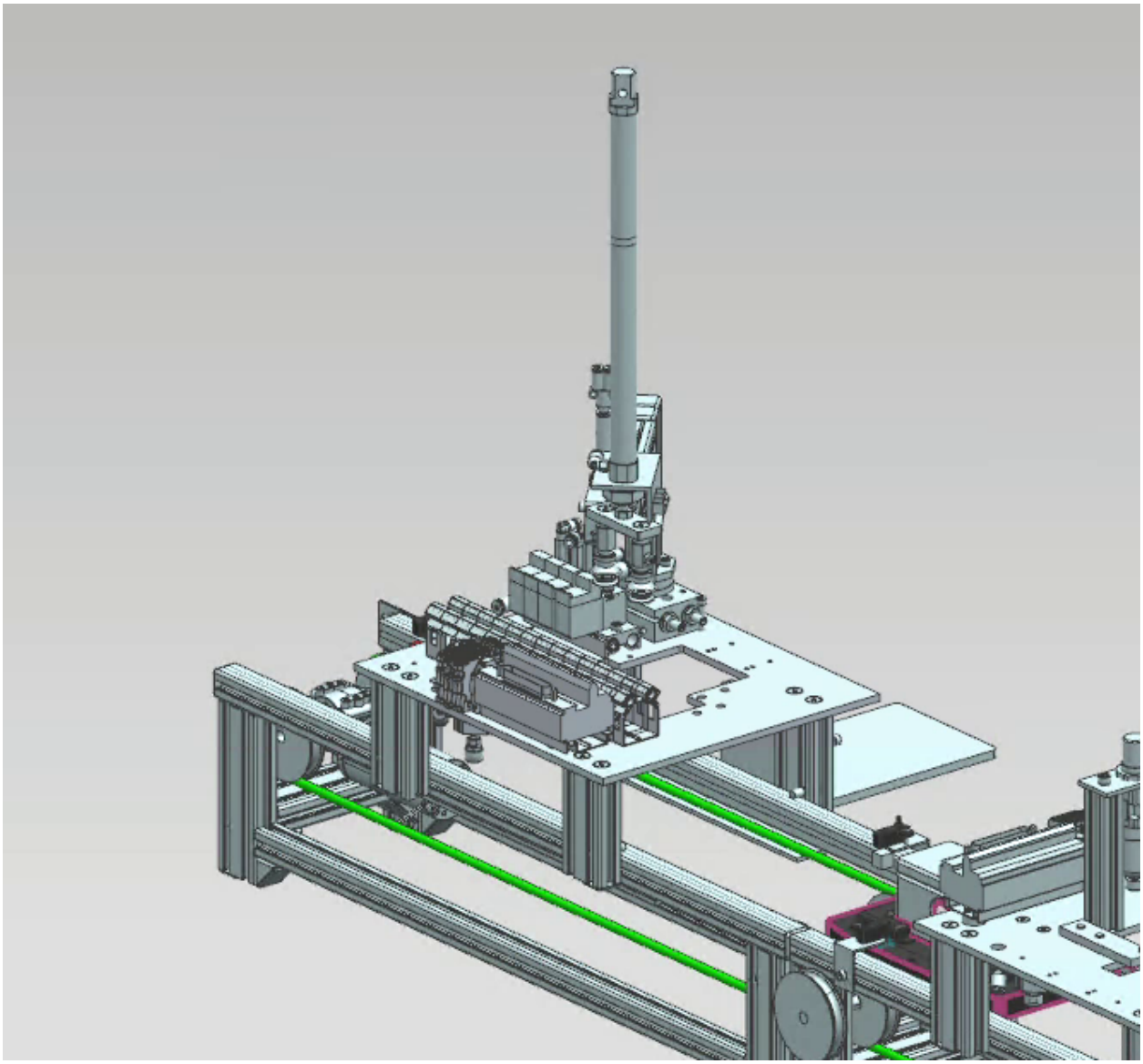


Fig. 4.30: The station's functionality

Top part, base part, bolt, and workpiece carrier

Assign rigid bodies to all parts that are going to move/collide during the simulation of the station. All parts that will not move/collide during the simulation should be ignored.

Rigid and collision bodies

Todo: Assign rigid and collision bodies to the top part, the base part, bolt, and the carrier.

Note: When assigning rigid bodies to part, a few parts can be selected together and defined as one colelctive rigid body. This should be done if those parts will always undergo the same motion together.

Note: When assigning collision bodies to parts, only sides/areas of the part that will undergo collision during the simulation should be chosen. Avoid defining the entire part as a collision body, because it increases the complexity of the body and unnecessarily slows down the simulation.

Note: When assigning collision bodies to parts, try to choose simple geometry (box, cylinder, sphere, etc.) if complicated geometry (mesh) is not crucial for the physical behaviour of the part.

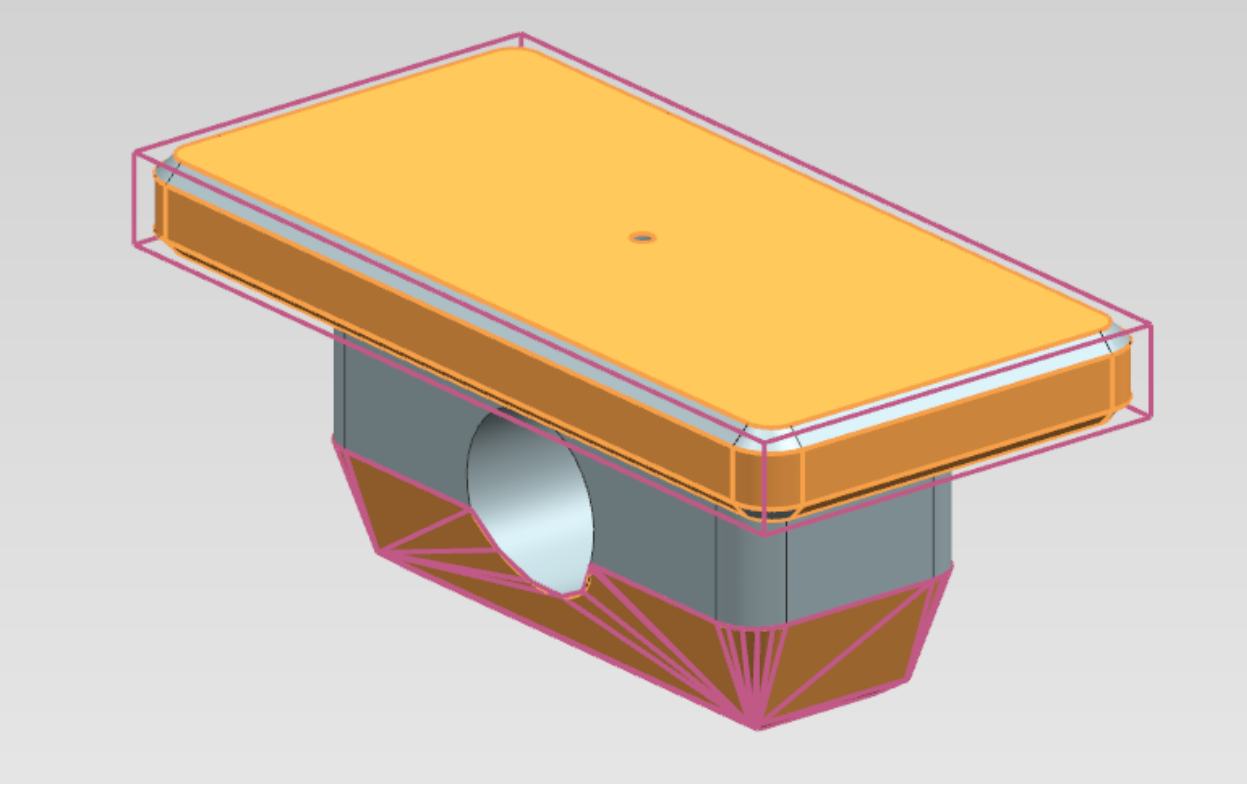


Fig. 4.31: An example of collision body assignment. Collision surfaces are shown in pink. The upper part has a simplified geometry and surfaces on the part that will not undergo collision were not chosen, which reduces the complexity.

Todo: Assign a collision body to the platform that receives the end-product.

Conveyor belt

To simulate a conveyor belt, a rectangular surface will be used as a `Transport surface`. The band itself will be blinded out while the simulation is running.

Todo: Assign a `Transport surface` to the part `200213_TransportflaecheDUMMY`. Do that on the `Entire_Transport_Band` level.

Pick and place mechanism

Rigid and collision bodies

The entire arm rotates to bring the end-product to the specified platform. Because all components of the arm undergo the same motion together at all times during the simulation, they should be defined as a unified rigid body.

Note: Because the cylinder's rod will undergo a different motion than the rest of the arm, it should not be included in the same rigid body definition. The cylinder's rod should be defined as its own rigid body.

Todo: Assign a rigid body and a collision body to the cylinder's rod and to the stopper's rod.

Joints

Todo: Assign a sliding joint to both the cylinder's rod and the stopper's rod and enter appropriate limits.

Note: Does the base object have to be specified in both cylinder's cases? Why or why not?

Todo: Assign a hinge joint at the rotating part of the arm.

Special joint case: vacuum gripper

A fixed joint should be created between the gripper and the top part being picked up. Because those two parts belong to different sub-assemblies, only the base of the fixed joint (the gripper) should be given when creating the fixed joint. The attachment of the fixed joint (the product) should be assigned to the fixed joint later during the simulation using an operation.

The top part is connected to the bottom part through a magnetic contact in addition to the bolt. The magnetic contact will also be simulated using a fixed joint.

Todo: Create a fixed joint between the gripper and the top part. Create another fixed joint between the top part and the base part. At first, give in only the base objects (the gripper for the first fixed joint and the top part for the second

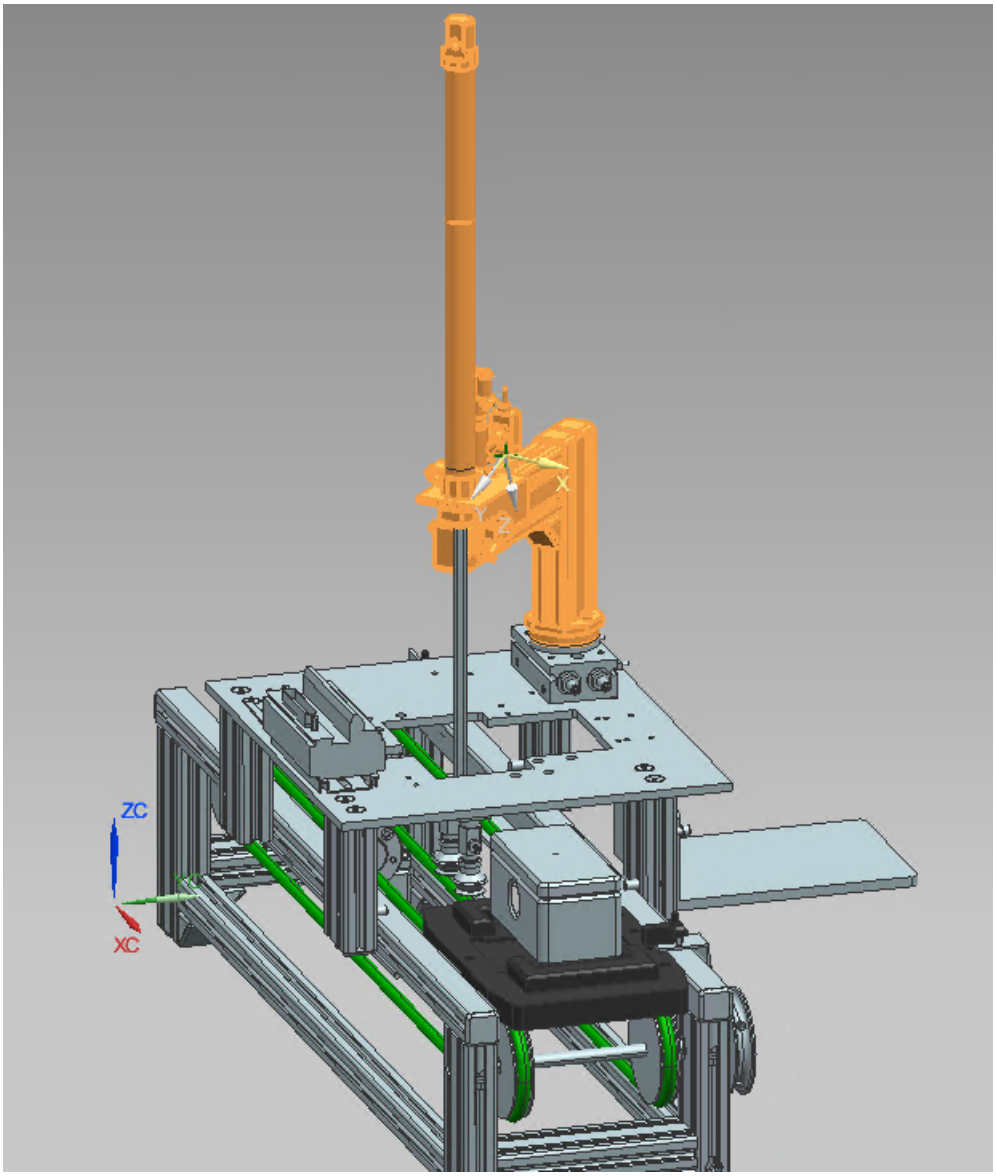


Fig. 4.32: The entire arm is defined as one rigid body

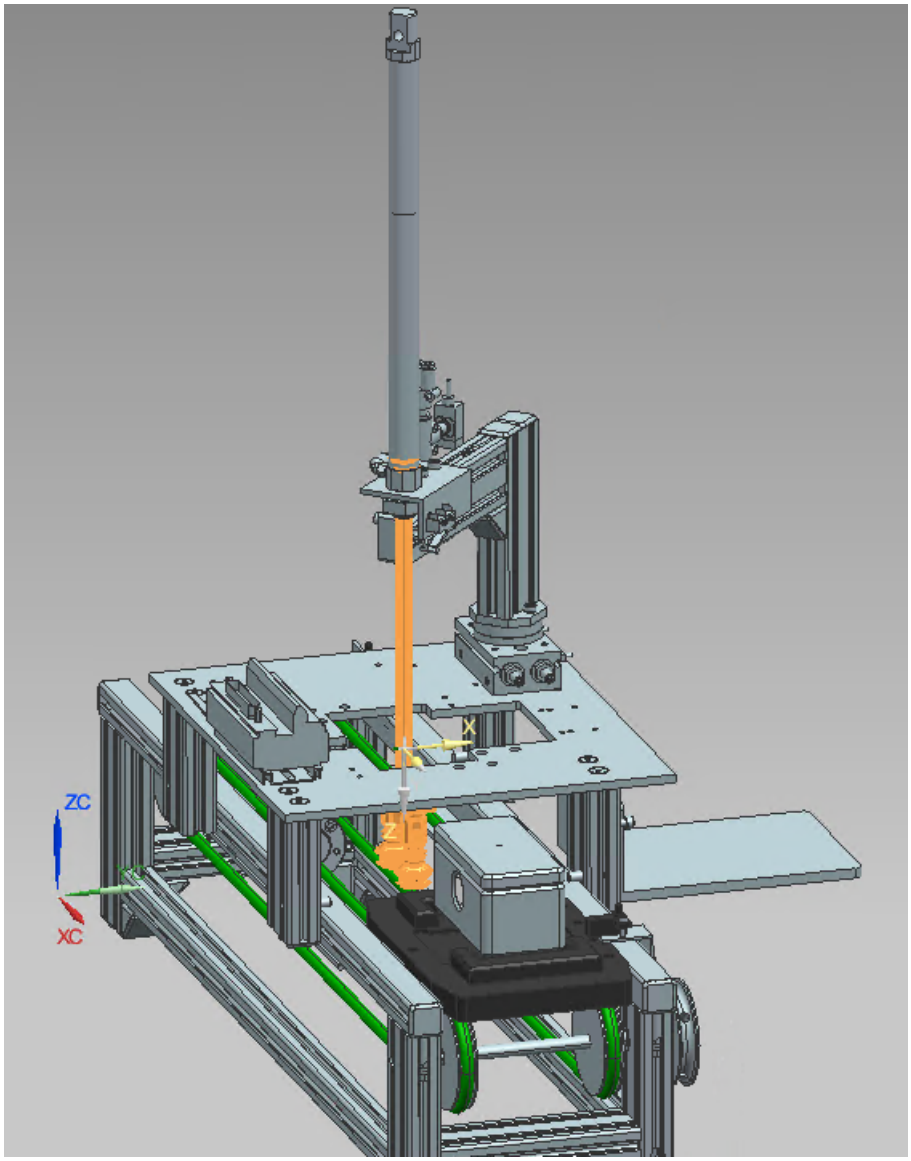


Fig. 4.33: The cylinder's rod is its own rigid body

fixed joint respectively). Manipulate those fixed joints by adding their attachment objects later in the simulation when the gripper is supposed to lift the product using an operation (see section on operations).

Todo: Optional task: is there a way to let the product fall (switch off the fixed joint) if the product's weight exceeds a certain limit? Describe the approach briefly.

Position controls

Todo: Assign position control objects to each of the joints.

Signals

Todo: Assign signals to each of the position control objects.

Operations

Operations are like if-statements in the MCD simulation. They can be used to trigger something in the simulation if an external (or internal) condition is met.

Note: Operations can be found in the `Sequence Editor` menu on the left.

Todo: Create an operation to send the stopper cylinder down. Call it `Stopper_down_operation` for clarity.

The selected physics object of the operation is the stopper's position control. The runtime parameter selected is the position, because this is the parameter to be manipulated by the operation. The new value is inserted in the `Value` column. Under `Condition`, the condition object selected is the signal that was created to control the stopper.

The equivalent in C++ would be:

```
if(bStopper_down == true) {stopper_sliding_position = 5;}
```

Todo: Create operations to rotate the arm, drive the pneumatic cylinder down, and turn on the gripper (turn on the `Stick when Collision` property of the gripper's collision body).

Todo: Determine the position control values by measuring the length of the rod and taking the start position into consideration.

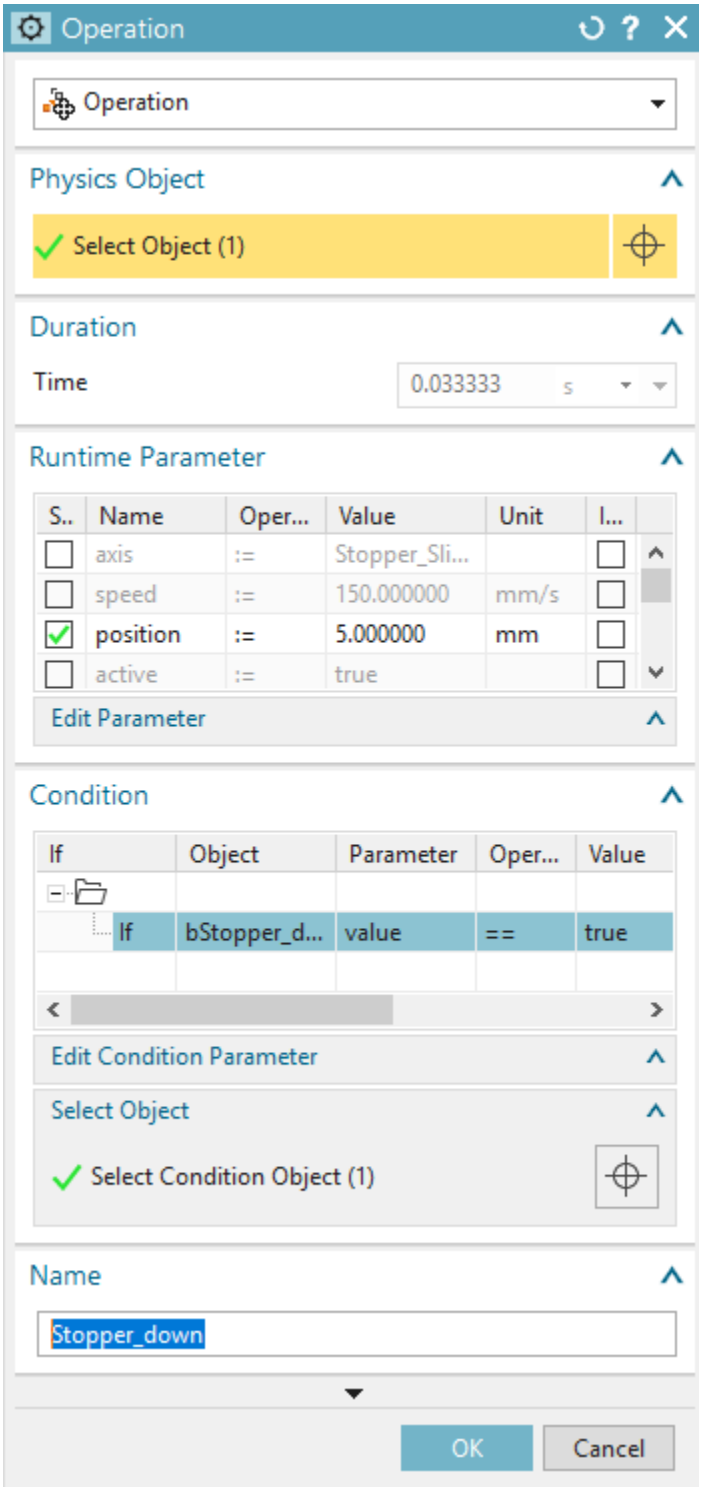


Fig. 4.34: An operation to send the stopper's sliding joint to position `5mm` if the signal `bStopper_down` is true.

Sensing Mechanism - Entry and Exit Sensors

Collision Sensor

The IMS7 has entry and exit sensors at the start and end of the conveyor belt (see section on IMS7 Station Functionality). These two sensors will be simulated as two small cubes with `collision sensor` properties.

Create a new `block` in `mechanical concept` under the `Home` tab. Give the block the dimensions of 10mm x 10mm x 10mm. Under the `Assemblies` tab, click on `Create New` to create a new model. Name the model something appropriate, like `end_limit_switch_dummy` and click `OK`. To select the block just created, navigate to `Part Navigator` on the left-side menu and choose the block that was just created. Now the block is a model and can be seen in the `Assembly Navigator`. Choose `Move Component` under the `Assemblies` tab and move the new block to where the sensor is supposed to be.

Note: When moving an object, clicking on Specify Orientation will enable you to drag on the three axis to move the object.

Now that the dummy-sensor is in place on the conveyor belt. Assign a collision sensor to it. Be sure to call the collision sensor something appropriate, like `End_limit_switch_right`.

Signal

For the dummy sensor to send its reading to the PLC, a signal should be created.

Todo: Create a boolean signal and connect it with a runtime parameter. The physics object selected should be the `End_limit_switch_right` object. Be sure to define the signal as an `Output`, since it is an output from MCD's perspective.

Note: It is important to give the signal a clear name. For example: `bEnd_limit_switch_right`. Where `b` in the beginning stands for boolean.

The entry sensor is done for now. Later, `End_limit_switch_right` will be mapped to a PLC signal with the exact same name.

Todo: Go through the same steps again and create a second sensor to detect the carrier's exit. Call the new sensor signal `bEnd_limit_switch_left`.

4.10.4 The Physical Layer – PLC Program

A program to control the IMS7 station can be found in the downloads. In this module, a PLC will be simulated to control the station.

TIA Portal

The following figure shows a state machine diagram for the IMS7 station. This state machine is implemented in the program.

Todo: Open the program in TIA Portal.

Note: Outputs of the MCD Simulation are inputs of the `IMS7` function block, and vice versa. Use the same exact names that are used in the variables in the `IMS7` function block in the MCD simulation also. This allows for auto-mapping later.

PLCSIM Advanced

Todo: Start PLCSIM Advanced and start a PLC simulation.

Note: Make sure the PLC simulated has the same name as the PLC in the TIA Portal program.

Note: Make sure `PLCSIM Virtual Eth. Adapter` is selected.

In TIA Portal, compile your program and upload it to the simulated PLC.

4.10.5 Controlling the MCD Application using the PLC Program

Todo: Make sure an OPC UA server is configured in the PLC program and connect to it in MCD. When including variables from the PLC program through OPC UA, choose the variables that you need to control the simulated production station.

Todo: In the `Signal Mapping` window in MCD, click the option `Do Auto Mapping`. This will automatically map identically-named signals to each other.

Todo: Run the simulation and watch the PLC's variables in a watch table. The production station should now be controlled through the simulated PLC.

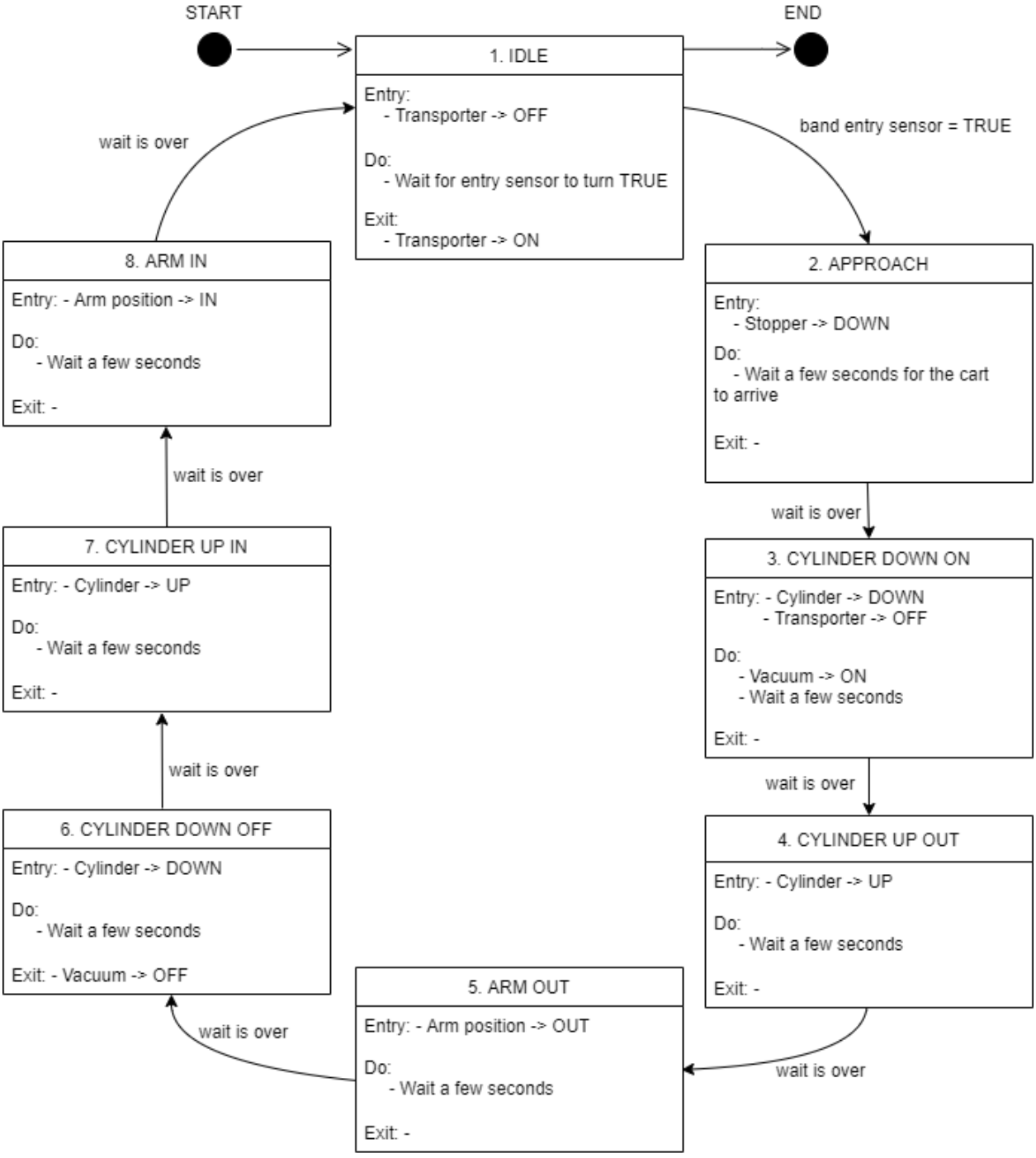


Fig. 4.35: A state machine diagram for the IMS7 station's functionality.

IMS_7

	Name	Data type	Default value	Retain	Accessible from HMI/OPC UA	Writable from HMI/OPC U	Visible in ...
1	Input				☐	☐	☐
2	bTransport_band_entry	Bool	false	Non-retain	☑	☑	☑
3	Output				☐	☐	☐
4	bStopper_down	Bool	false	Non-retain	☑	☑	☑
5	bTransporter_on	Bool	false	Non-retain	☑	☑	☑
6	bCylinder_up	Bool	false	Non-retain	☑	☑	☑
7	bVacuum_gripper	Bool	false	Non-retain	☑	☑	☑
8	bArm_out	Bool	false	Non-retain	☑	☑	☑
9	InOut				☐	☐	☐
10	<Add new>				☐	☐	☐
11	Static				☐	☐	☐
12	nState	Int	1	Non-retain	☑	☑	☑
13	bEntry	Bool	false	Non-retain	☑	☑	☑
14	Temp				☐	☐	☐
15	<Add new>				☐	☐	☐
16	Constant				☐	☐	☐
17	<Add new>				☐	☐	☐

Fig. 4.36: Variables of the IMS7 function block

4.11 Summary

The virtual commissioning of an entire production line is a process that takes effort to achieve but reduces the overall project time through early verification of the control code.

After simulating each station and testing the individual code, the next step would be to combine all stations in one whole simulation of the entire production line. Likewise, the individual codes should also be combine and the entire setup should be tested for further bugs. Bringing us a step closer to a more confident real commissioning with the real hardware.

4.12 Troubleshooting

4.12.1 OPC UA Communication Troubleshooting

Runtime error: Cannot connect to this OPC Server

If upon starting the simulation the above error shows up, navigate to `External Signal Configuration` in the `Automation` tab. The connection status of the OPC Server has to be 'connceted'. If the status is 'connected', click OK and try running the simulation again. If the status is 'unknown', click on `Refresh Server Status`. This may change the status back to 'connected' and the simulation will then run.

Green tick is missing; cannot choose external variables in External Signal Configuration

If this is the case, close all open assemblies and restart Siemens NX.

OPC UA communication connected but variables do not get updated during the simulation #1

If the OPC UA communication is succesful and mapping is done correctly, but the variables do not get updated through OPC UA in either direction (i.e., both inputs and outputs do not respond), then a certain part or sub-assembly is causing this problem. The solution is to start rebuilding the current main assembly in a new file and adding each part one by one, testing the OPC UA connection after adding each part. This way, you will be able to spot the part and/or assembly that is causing the communication issue.

If the cause of the issue is a sub-assembly, create a new sub-assembly with a different name and import all parts of the old sub-assembly in. Use the new assembly instead of the old one. This should allow the communication to function.

If the cause of the problem is a part, reconstruct the part in a new file and use the new part (copying the old part in a new file does not resolve the problem).

OPC UA communication connected but variables do not get updated during the simulation #2

If the OPC UA communication was running perfectly but all of a sudden the variables stop responding, this may be caused by running a simulation only on a sub-assembly or part level. Undoing the past 3 steps and reselecting the main assembly then running the simulation will undo this problem.

Siemens NX MCD offers the possibility to run simulations on the sub-assembly or part level. This is helpful to see how only one part will be simulated without simulating the entire assembly. However, it is recommended to run the simulation only on main assembly level because of this OPC UA communication bug.

Only some variables are updated through OPC UA

This issue occurs when communicating with a PLC program over OPC UA.

In case the OPC UA communication seems to function with only some variables and with other variables it seems to be down, make sure that the variables being transfered over OPC UA are declared in a `function block` in the PLC program and not mapped directly to a PLC I/O module.

4.12.2 PLCSIM Advanced V2.0 Troubleshooting

PLC Instance cannot be started: licence could not be found

If the simulation instance of a PLC does not run, open PLCSIM Advanced as an administrator and try again.

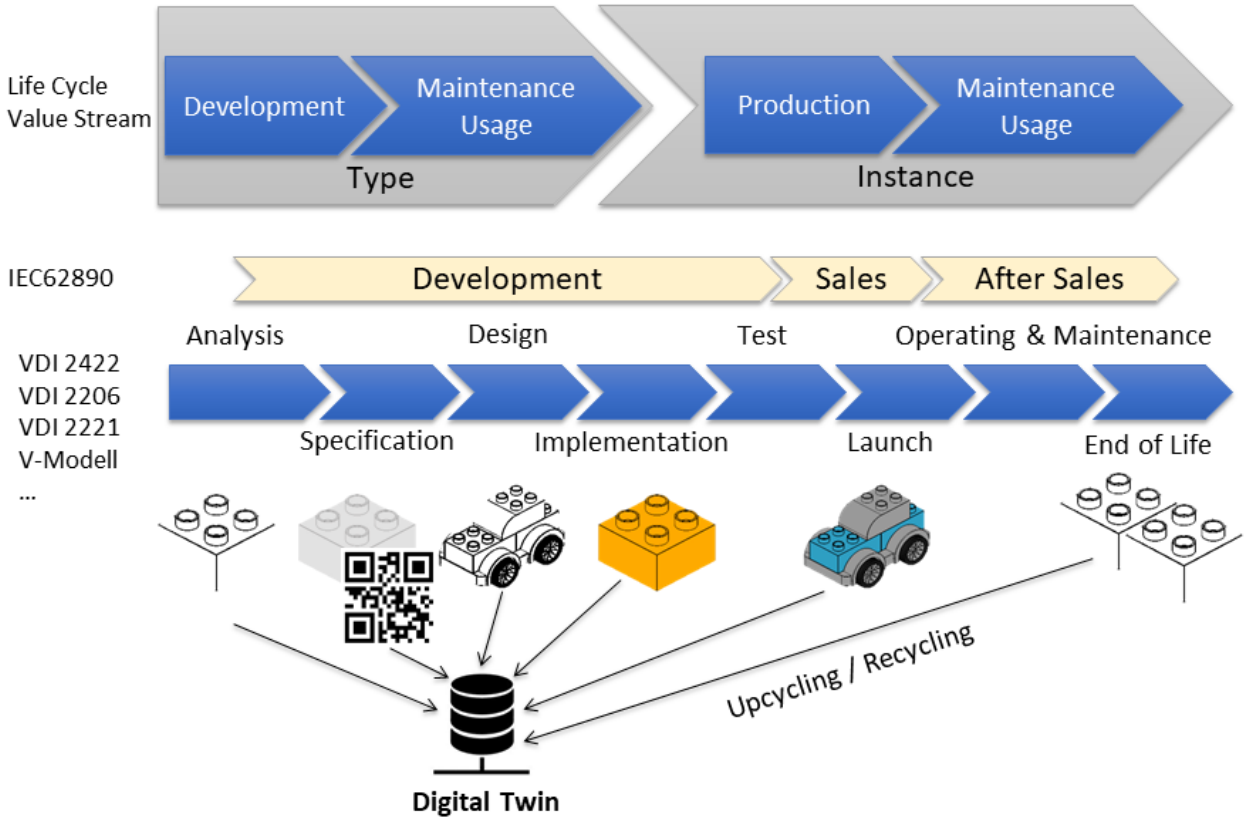
CHAPTER

FIVE

OPERATOR DIGITAL TWIN FOR ASSISTANCE SYSTEMS

5.1 Short Introduction to Digital Twins

Digital twins are often defined as "Digital representation, sufficient to meet the requirements of a set of use cases" (Plattform Industrie 4.0). Integrating data about the asset (object which has a value to an organization and is therefore managed individually) from different life cycle phases, a digital machine twin can provide valuable structuring of data and insights; from asset development and configuration all the way to the asset's end of life.



In the figure below, we can differentiate a type and an instance. All planning data creates the type of the asset. The **type** is used to implement the physical (real-world) instance of the asset. The planning activities are inclusive of all domains namely mechanical, electrical and software. A type becomes an **instance** when development and prototype production is completed, and the actual product is manufactured.

For example, the type of car includes CAD models, wiring diagrams, accessories which can be varied according to a desired use case. Thus, the type specifies which different car configuration are available in form of a 150% model.

When the customer selects their individual configuration, for example in an online shop, a virtual 100% model is created. The 150% model as well as the 100% model are still part of the product type. As soon as the 100% model is manufactured, an individual instance is created. Thus, the customer gets a real-world touchable car with a unique serial number. Customers will use their car instances and create specific and individual lifecycles with respect to their usage.

When the product is delivered to the customer, this marks the beginning of a product's lifetime or service time. During this, the product provides the service for which it was planned and manufactured. During service life, prescribed maintenance is suggested to avoid sudden breakdown of the product. Once the product's service time ends or the product is no longer required, the product is decommissioned, and this marks as end of life. During the end of life, the product can be recycled, up cycled, or decomposed according to requirements.

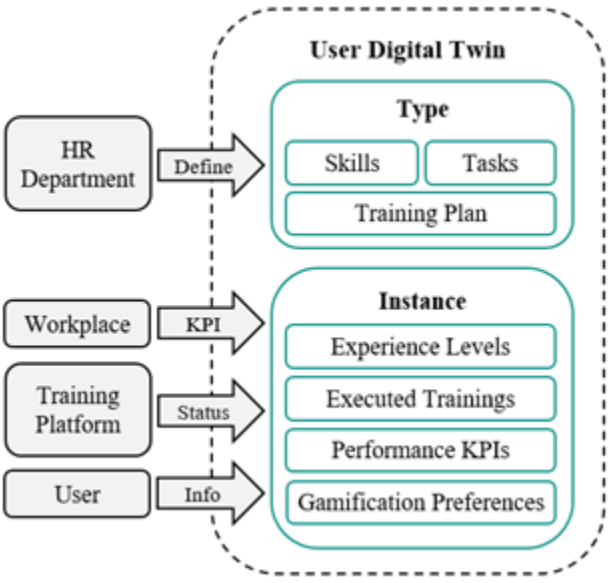
The **digital twin** includes data from the type (analysis, specification, and design phases) as well as data from the instance (implementation, system test, operation, and maintenance phases). Also, end of life phases must be included. The collected data can be analyzed and a predictive model can be created. This way, objects which are, for example, subject to wear can be replaced at the right time.

5.2 Digital Twins of Operators

Assistance systems are important for humans in production systems to handle the increasing complexity. They can display relevant indicators, provide information about work routines, or notify their user in case of problems and errors. Combined with gamification approaches, they allow personalized feedback and targeted support during work. Using gamified level systems, habituation effects, and dependencies upon the assistance systems can be decreased.

However, the development and usage of gamified assistance systems require access to different data sources, which often remains a challenge.

In this course, we are accessing automatically generated work data from a manual work station and transfer it to useful indicators for user digital twins. This step is important as the raw data is not suitable for user digital twin applications.



CHAPTER

SIX

OEE EVALUATION WITH PYTHON

In this course, work data from a manual working station will be analyzed according to different KPI, such as the Overall Equipment Effectiveness (OEE).

6.1 Learning Outcome

In this course, you will learn how to use existing sensor data to calulate work indicators, such as the OEE. You will also create visualizations for further data analysis and interpretation.

6.1.1 Requirements

- Basic programming principles
- Basic knowledge of Python including Pandas

Hint: If you're unfamiliar with python or need a refresher, please prepare the following micro courses:

- https://www.kaggle.com/learn/intro-to-programming
- https://www.kaggle.com/learn/python

Hint: If you have not worked with Pandas before, please prepare the following micro course:

- https://www.kaggle.com/learn/pandas

Hint: You can download a Pandas cheat sheet here: https://pandas.pydata.org/docs/user_guide/index.html

---

6.1.2 What you need

Hardware

- Laptop or PC

Software

- PyCharm Community version (https://www.jetbrains.com/pycharm/)
- Python 3.8 (or larger)

6.1.3 Sources and Resources

- https://pandas.pydata.org/docs/getting_started/intro_tutorials/index.html
- https://pandas.pydata.org/
- https://matplotlib.org/

6.2 Task

The Industry 4.0 model factory at FH Aachen includes a manual working station (MWS). The MWS includes different sensoric devices for user and workspace tracking as well as a projector to display work instructions. Currently, the MWS is used for a variety of assembly tasks, such as the mounting of an industrial handgrip or a dog-shaped structure.

https://youtu.be/xtaxXZJaG_Y

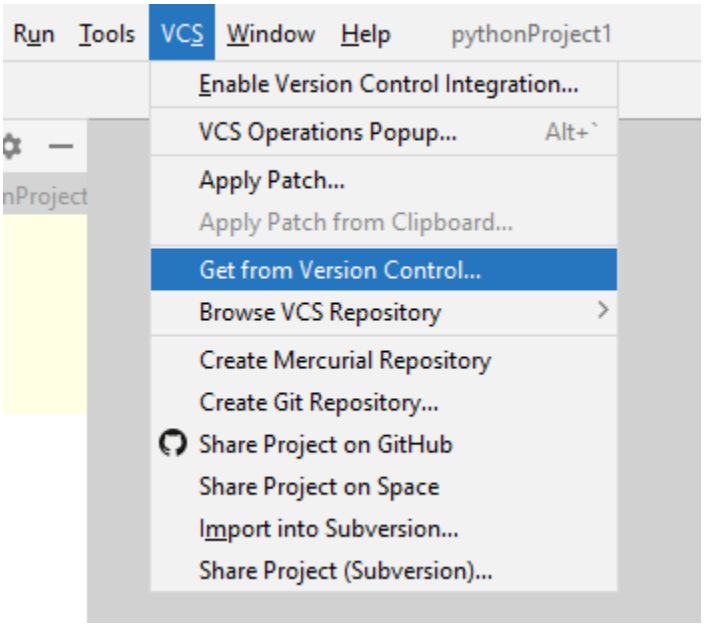
The data from the MWS will be used for the indicator calculations. In order to see and understand the work data, you need to set up the work environment first.

6.3 Setting up the work environment

All required data and the prepared python modules are available in a gitlab repository: https://git.fh-aachen.de/lectures-material-wollert/oee_python_course

---

Todo: Fork and clone this repository (using HTTPS).

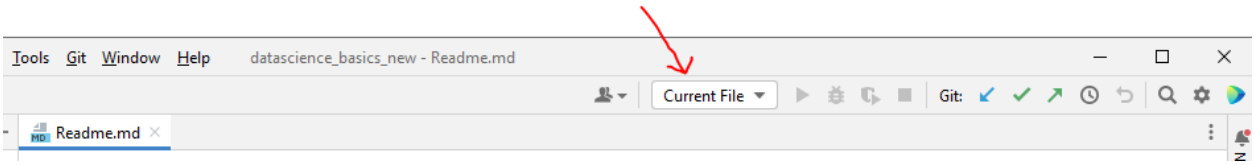


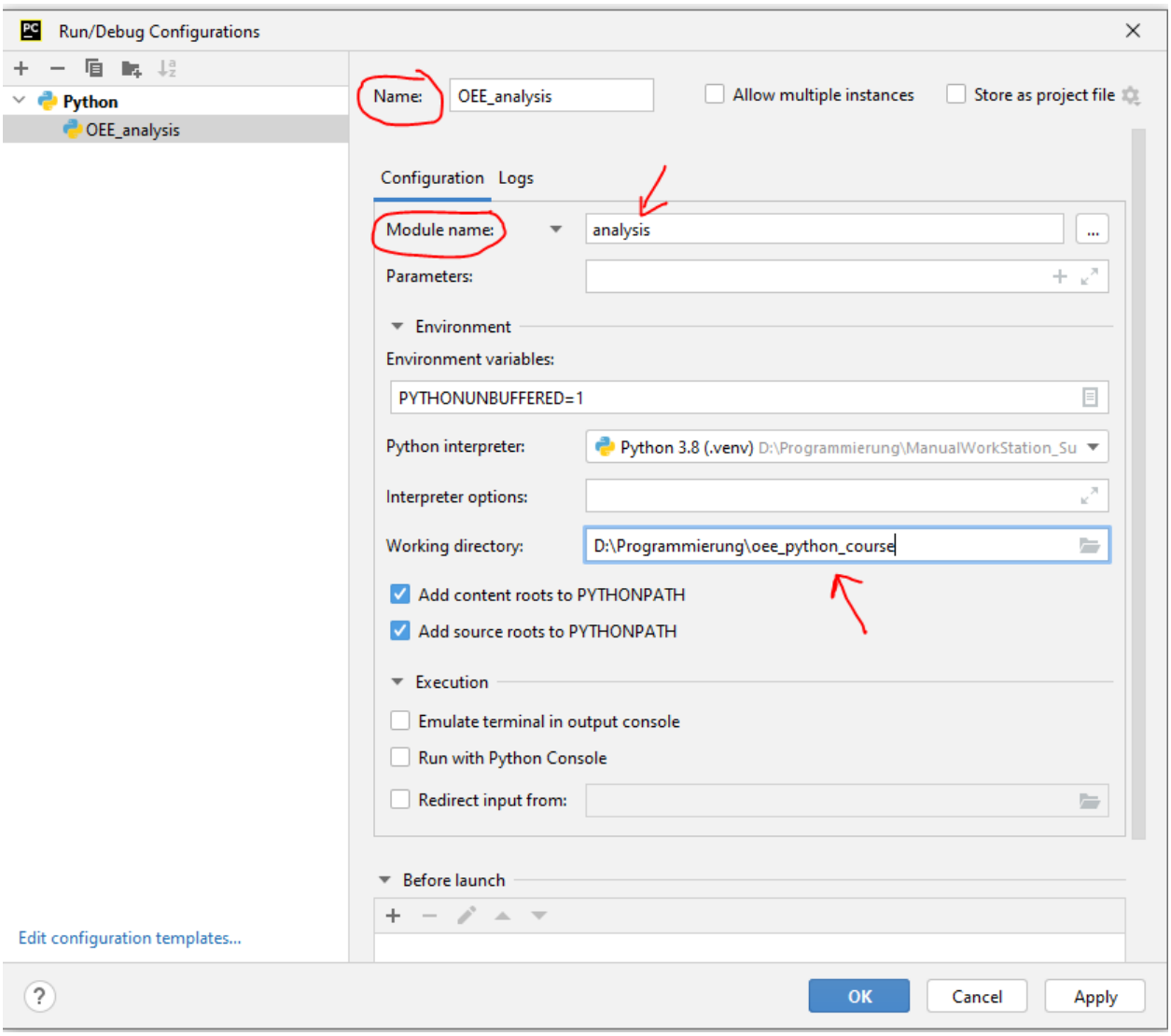
Todo: Check if all required libraries (see `requirements.txt`) have been installed automatically.

Therefore, go to `Files` -> `Settings` -> `Project` -> `Python Interpreter`.

If libraries are missing, you can manually install them using the + symbol.

Todo: Add a new Python run configuration to start the program logic in PyCharm. Add your module (here: *analysis.py*) to the configuration. Set the working directory to your project folder.

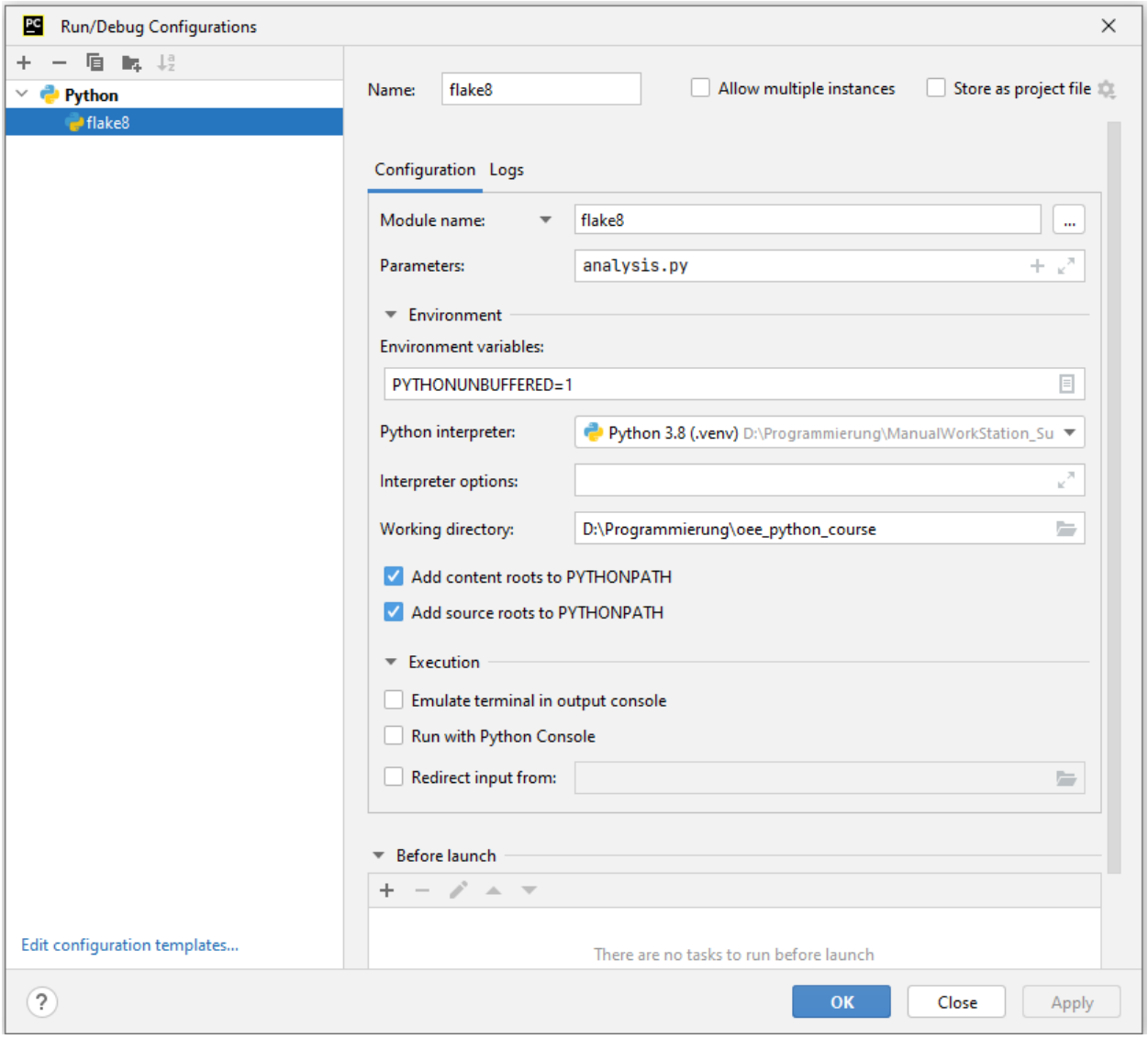




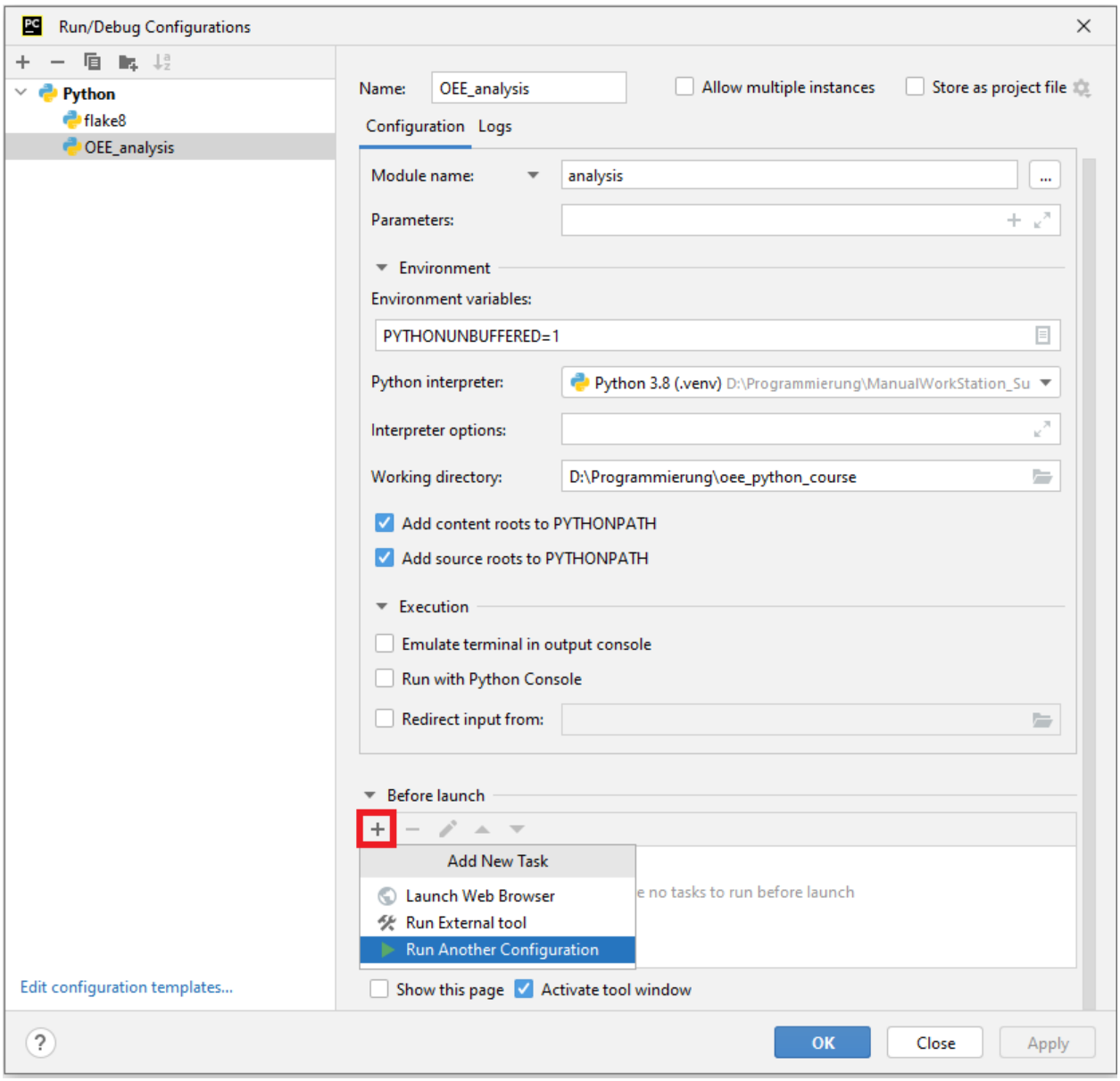
Todo: Check if the main function is executed when you start the configuration (the logger should show an *AssertionError: The function 'get_dataframe_from_directory' has no return value*).

(The assertion error is thrown as the function `get_dataframe_from_directory` is not yet implemented.)

Todo: Include style checking using flake8 to make your program more readable. Add the module which should be checked in `Parameters`. Set the working directory to your project folder.



Todo: You can automatically check your programming style before executing the main program logic by adding the flake8-configuration to your main configuration.

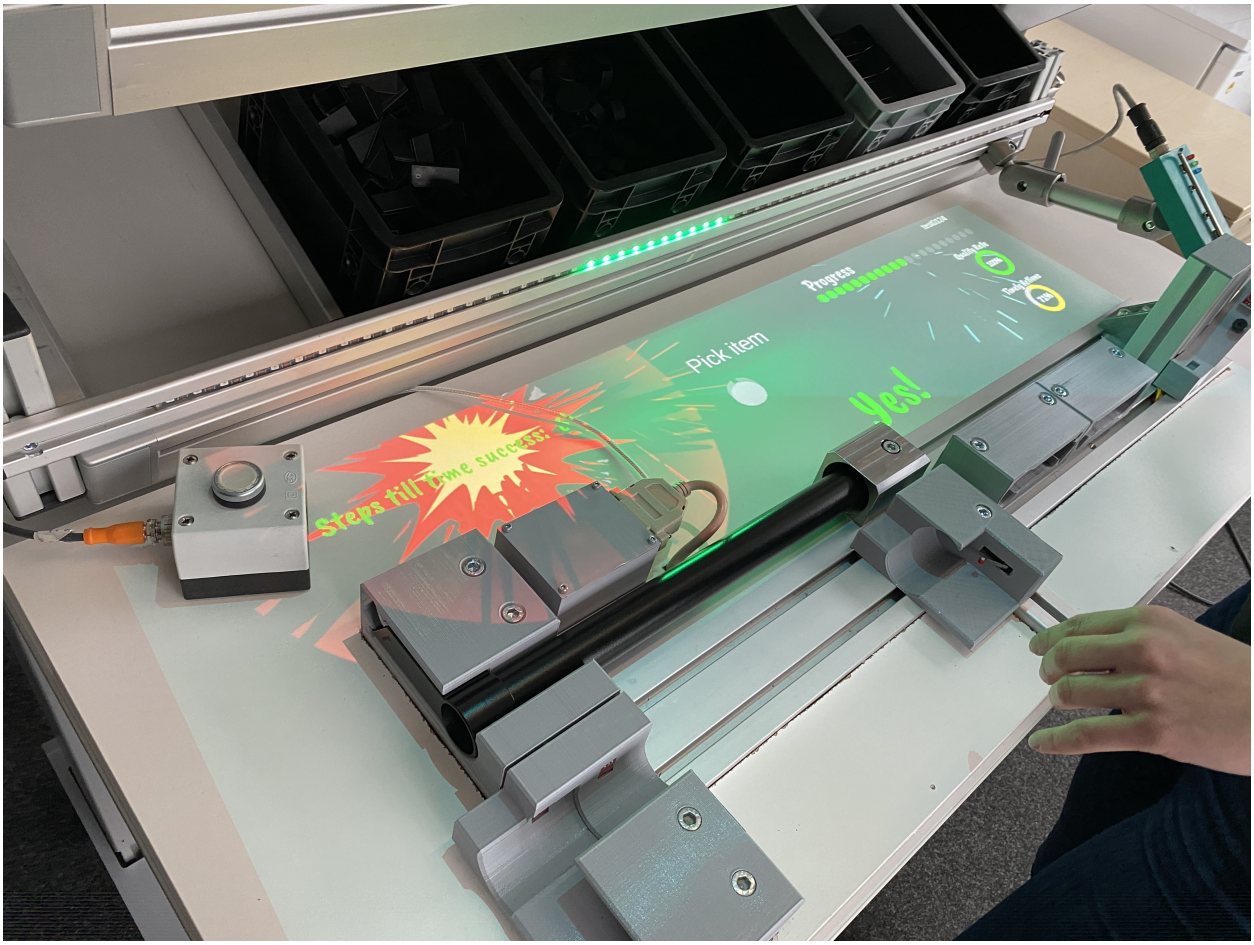


Todo: Check if the work data is in your project repository.

6.4 Understanding the work data

The MWS of the Industry 4.0 model factory displays situation-dependent work instructions and detects user interactions. It consists of

- A height-adjustable working desk on which instructions can be projected,
- A storage area with a pick-by-light system,
- A tool holder and two acknowledgment buttons, and
- A Poka-Yoke workpiece holder for assembly operations.



The work sequence consists of 20 work steps in which parts must be mounted or clipped to form an industrial handgrip of the company Item Industrietechnik GmbH. One tool, a screwdriver, is required for the assembly execution. First, the side parts of the handgrip are premounted twice and placed in a storage box. Afterward, these side parts are attached to the main profile, and the covers are placed over the screws.

In order to reduce the assembly complexity and simplify the small parts handling, a workpiece holder is used. The workpiece holder consists of 3D-printed parts, which limit the mounting options. This way, unintended errors are prevented (Poka-Yoke principle).

The manual workstation is capable of detecting the majority of user interactions automatically:

- A LiDAR sensor is used to capture grabbing actions in the storage area.
- The tool holder identifies if a tool is available or not.

- The workpiece holder checks if the workpieces are placed in the correct positions.
- Also, two check positions are integrated, ensuring all covers are placed correctly, and the side areas are mounted tightly.

As the cover clipping cannot be identified automatically, these work steps must be acknowledged manually by pressing a hand or foot button. In order to evaluate the user interactions, the system stores all interactions on a granular basis, including user id, timestamps, durations, run number, and correctness.

For a user study, over 60 test persons were asked to assemble the handgrip five times at the MWS. The participants had different backgrounds, from mechatronics to psychology.

Todo: Check the provided work data in your project repository (folder `workdata`).

Each number (501-5xx and 601-6xx) represents a test user who was requested to manufacture the handgrip five times.

The node specifies the name of a work step according to a given work sequence. For instance, the work sequence starts with a work step (node) called "Place_Nut_1" followed by "Place_Holder_1". Each work step can include several user interactions, such as picking a nut and placing it afterwards.

The "DefaultTime" specifies an initial approximation of working times according to the performance of ten initial users.

In this project, the goal is to analyze the obtained work data according to productivity and quality; the main focus is on the Overall Equipment Effectiveness (OEE). In this application example, the different working days should be analyzed and compared.

As the work data was obtained during a user study, no traditional working shifts were used but time slots (60min) during which the participants had to assemble the handgrip five times but also answer some questionnaires. The starting time of the planned working slots were noted in a .csv file

---

Todo: Check out the planned work slots `planned_working_times.csv`

6.5 Basic program structure

For this course, a basic program structure is provided which predefines all required functions to solve the data analysis task.

The `main()` function includes predefined function calls related to the different sections of this course. It also includes tests to automatically validate the function implementations.

Todo: Open the `analysis.py` file and check the `main()` function. It contains the outline of this course and is structured accordingly. It looks complicated, however most of it are associated tests. Look for instance at `assert df is not None`. This line raises an `AssertionError` if the variable `df` is None. The testing provides you with a direct feedback and lets you know if your implemented functions work as expected.

Warning: Please only change the `main()` function when instructed during this course (indicated by the term *FIXME*). All new program code should be added to the other predefined functions.

Hint: The predefined function contain documentation explaining how the function should work. Additionally, hints are provided to help you implement the functions. For every hint usually just a few lines of code need to be written.

You should use the hints to comment your code. Feel free to find your own solution if you don't want to follow the hints.

6.6 Data preparation - Work data

In a first step, the obtained work data files should be merged into one dataframe. Therefore, the function `get_dataframe_from_directory` should be implemented.

Todo: Expand the `get_dataframe_from_directory` function to load all work data files and put them in one dataframe.

Ensure that the timestamp is in the Pandas `datetime` format and use the timestamp as index and sort your data according to it using `sort_index()`.

Remove all columns containing "Unnamed".

Todo: Check if the created dataframe gets validated successfully and if the dataframe has the correct shape.

In a second step, sensor-based picking errors should be removed from the user data analysis.

Todo: Implement the `mask_dataframe` function to remove:

- all rows that contain "RemoveItem" in column "ActionName"
- all columns that contain "pickItem" in row "ActionType" only if row "Error" is True.

Todo: Check if the adjusted dataframe gets validated successfully and if the dataframe has the correct shape.

6.6.1 Base value calculation

Based on the obtained work data, the following information should be calculated for further analysis:

- Cycle times per user and run
- Mean times per user action (e.g. mean times for action "pickItem" for object "Nut") as setpoints for production planning

Cycle times

The cycle times (run times) allow an analysis regarding the individual user performance. Also, it can be analyzed if the user performance increases with the run numbers.

Todo: Implement the `calculate_cycle_times` function.

Create a new dataframe which includes all single cycle times per user and run number.

As the timestamps still include the removed actions, please use the `DeltaTime` values for your cycle time calculation.

Todo: To create a better overview for the data analysis, calculate the following information and put them in the predefined dictionary:

- minimum cycle time (seconds) for all runs of all users
- average cycle time (seconds) for all runs of all users
- maximum cycle time (seconds) for the last run of each user
- average cycle time (seconds) for the last run of each

As the timestamps still include the removed actions, please use the `DeltaTime` values for your cycle time calculation.

Setpoints

The obtained data should be used to set new default times (setpoints) for future handgrip assembly actions at the MWS. For instance, the default times are required to provide individual feedback to each user regarding his performance and improvements.

As the mounting actions vary in effort and dexterity, individual default times for each action are required instead of general default times per action type.

Todo: Implement the `calculate_setpoints` function to calculate the average working time per user activity and store them in a csv file for further usage. The average working times should be specified according to "ActionType" and "Node".

Also, return the setpoints as a multiindex pandas series.

6.7 OEE calculation

The overall equipment effectiveness is one of the main process indicators for production environments. It is calculated for production resources (e.g., a manual work station) for specific time intervals (e.g., days). It is not used to compare the effectiveness of different production resources but to analyze the effectiveness of **one specific** resource.

The typical OEE calculation is based on the three OEE Factors: Availability (A), Performance (P), and Quality (Q).

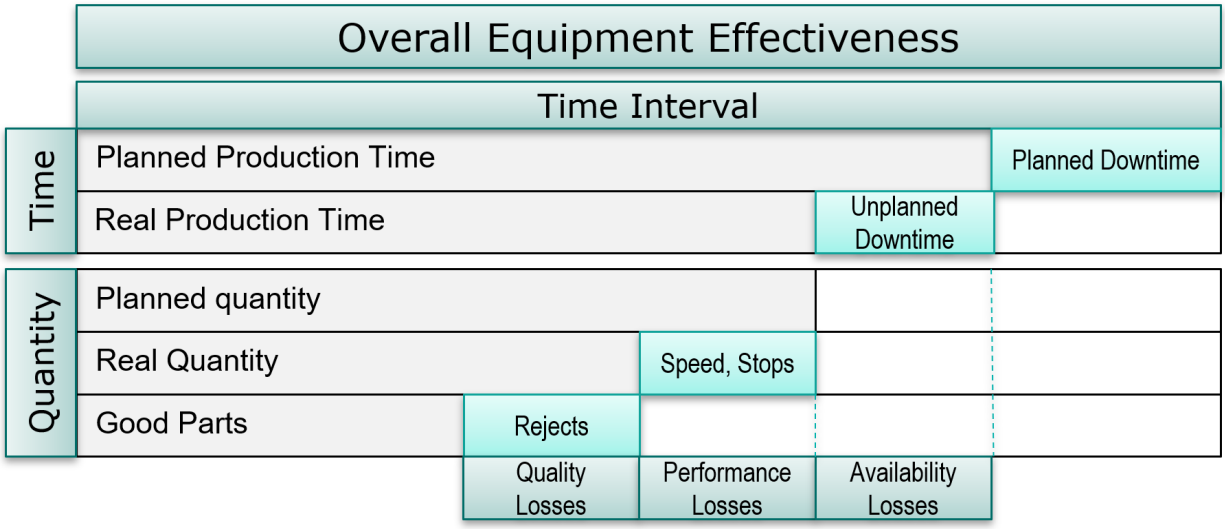
$$OEE = A \cdot P \cdot Q$$

In general, there exist different possibilities on how to calculate the OEE parameters. Typical examples are shown below.

For the availability (A), the real production time (t_{real}) is compared to the planned production time (t_{plan}). For instance, a workstation was planned to be operated 6 hours during an 8 hour shift; however, it was only running for 3 hours due to unplanned circumstances. Thus, the availability of this time interval is 50%.

$$A = \frac{t_{real}}{t_{plan}}$$

For the performance, the ideal production time (t_{ideal}) is calculated using the planned cycle time (t_{cyc}) and the number of produced parts ($n_{produced}$). Next, the ideal production time (t_{ideal}) is compared to the real production time (t_{real}) for the produced parts. For instance, 100 parts were produced during the three hours (180 minutes) working time.



According to the production planning, each part requires a planned cycle time of 1 minute. Thus, ideal production time was 100 minutes and the performance of this time interval 55.5%.

$$t_{ideal} = \frac{t_{cyc}}{n_{produced}}$$

$$P = \frac{t_{ideal}}{t_{real}}$$

The quality is usually calculated by dividing the number of good parts (n_{good}) by the number of total manufactured parts (n_{total}). For instance, 90 parts were good parts and 10 parts got rejected. Thus, the quality was 90%.

$$Q = \frac{n_{good}}{n_{total}}$$

Note: In this example of a handgrip assembly, the quality of the assembly routine is calculated instead of focusing on finished assembly parts. Therefore, the number of correct production actions ($n_{goodactions}$) are compared to the total number of production actions ($n_{totalactions}$).

$$Q = \frac{n_{goodactions}}{n_{totalactions}}$$

In order to calculate the OEE indicators, the existing dataframe will be expanded.

6.7.1 Work information per day

The availability calculation for the obtained data set is a bit tricky as the work data was obtained in a user study with a fixed number of runs and not in a traditional shift system. Therefore, the planned production time per day is accessed. The planned starting times of each user slot can be found in the file `planned_working_times.csv`.

Todo: Adjust the function `calculate_work_data()` to create a new dataframe `data` with work information according to working days.

For the OEE indicator calculation and analysis, the following information is required **per working day**:

- Number of timeslots (`TimeSlots`)
- Number of users (`Users`)
- Number of user work actions (`Actions`)
- Number of production steps with errors (`Errors`)
- Number of produced parts (`ProducedParts`)
- Real working time (`RealWorkingTime`)
- Unused working time due to no-show participants (`UnusedWorkingTimes`)

6.7.2 OEE parameters per day

As all required information have been obtained in the previous steps, the OEE parameters can be calculated per working day.

Todo: Adjust the `calculate_oee()` function to add the OEE parameters per day to the existing dataframe `data`.

6.8 Visualization

Visualizations help us to understand and interpret data. Typically, *matplotlib* is used for this purpose. However, the *pandas* library also provides basic functionalities to create plots which can be further adjusted using matplotlib.

Hint: If you want to create a new visualization or if you are looking for some specific elements, such as titles or colorbars, it is always a good option to use examples: https://matplotlib.org/stable/gallery/index

6.8.1 OEE graph

Todo: Implement the function `draw_oee()` to create an OEE bar chart displaying the OEE parameters per day. Include a heading, axis labels, and a legend. Show the parameters in % and adjust the figure size according to your data.

Save the figure as png-file in the folder *figures*.

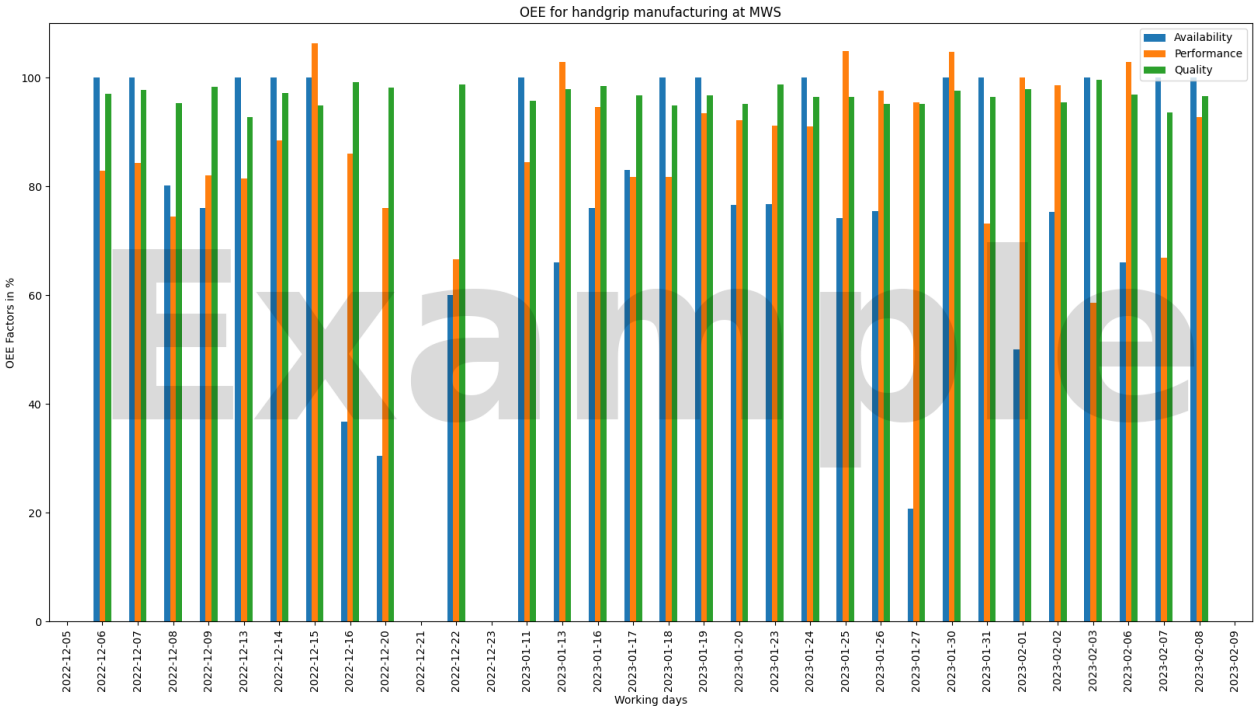
6.8.2 Scatter plot - cycle times

Todo: Implement the function `draw_scatter` to create one scatter plot for all cycle times (each point should represent one run) and one scatter plot for the cycle times of the last run. Add a histogram to both scatter plots.

Also, add a line plot for the average cycle times to both scatter plots.

Save the figure as png-file.

Do not forget to label your axes and include a heading.



6.9 Data evaluation

Now you have a good basis to understand your data and start improvement processes.

Todo: What would you do to improve the OEE?

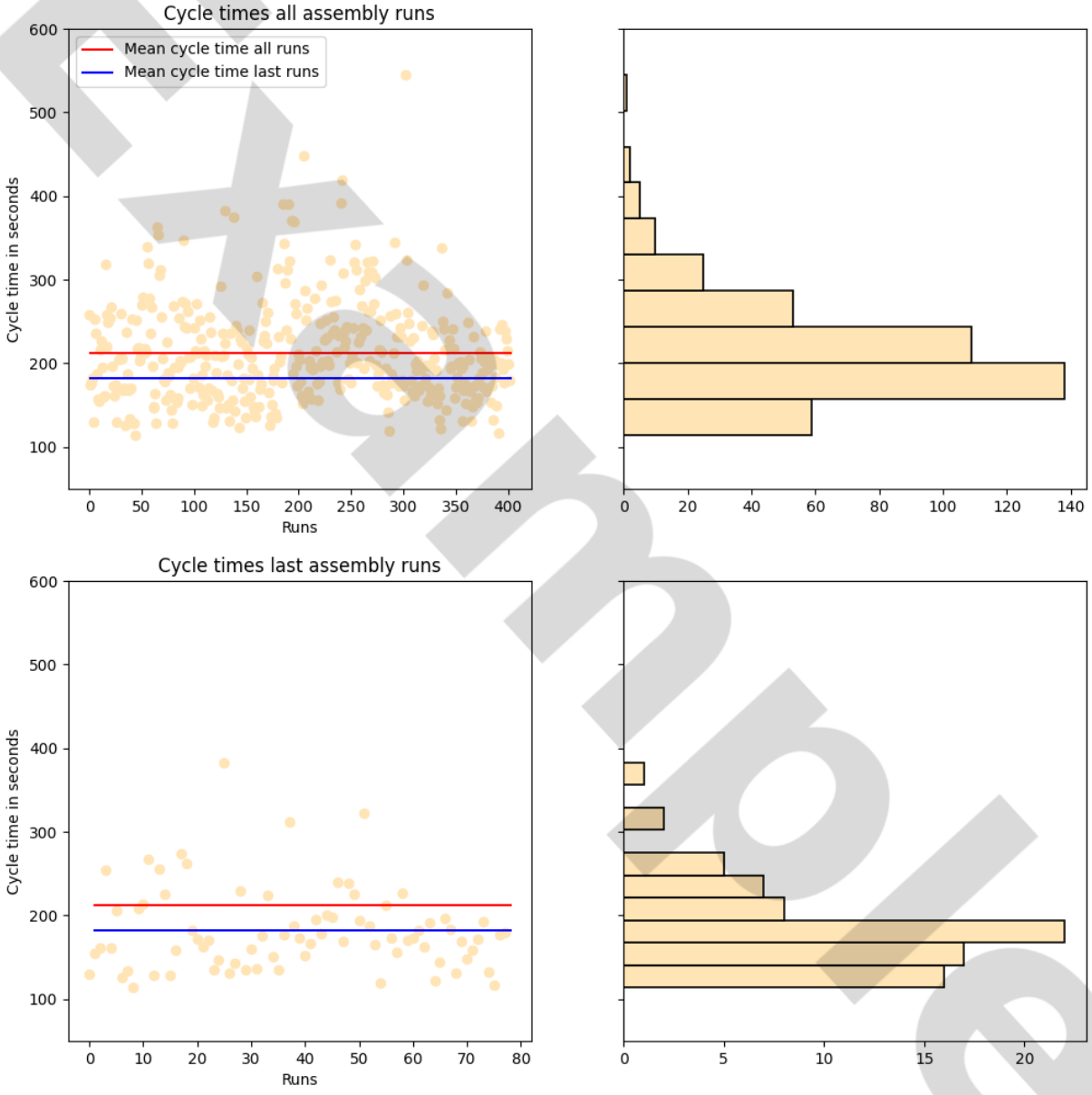
Todo: What do you see as main factors which negatively influence the OEE?

6.10 Problem Solving

6.10.1 Plotting

The render backend does not work

Usually, the render backend from `matplotlib` should open an interactive window. If this is not the case, there is some error with the installation. As a quick fix you can avoid the interactive window, by using the `plt.savefig` to store the graph on your filesystem. If you want to draw a new graph, you need to remove the old graph from your memory. Usually, this is done automatically, by closing the interactive window. Call `plt.close` to to this task manually.


Cycle times all assembly runs
Mean cycle time all runs
Mean cycle time last runs
Cycle time in seconds
Runs
Cycle times last assembly runs
Cycle time in seconds
Runs

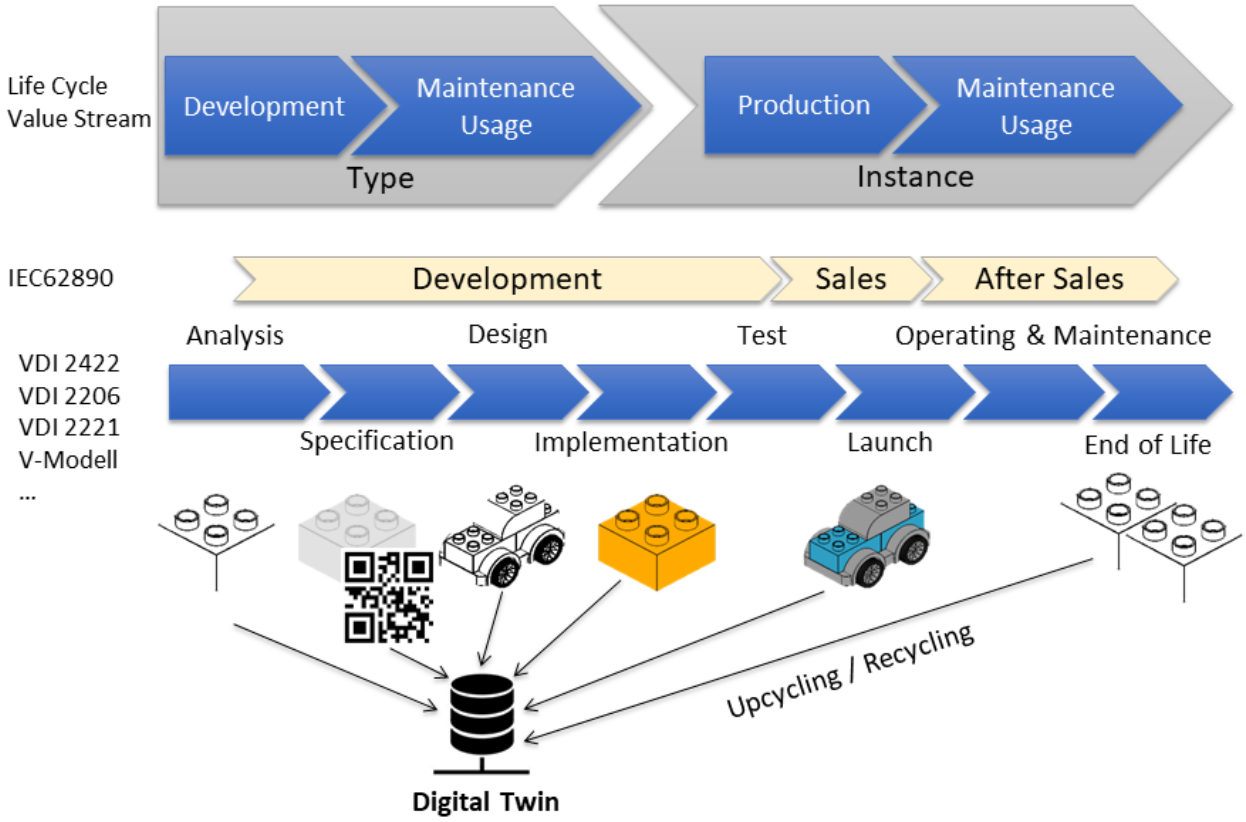
CHAPTER

SEVEN

DIGITAL TWIN OF A ROBOT DOG

7.1 Short Introduction to Digital Twins

Digital twins are often defined as "Digital representation, sufficient to meet the requirements of a set of use cases" (Plattform Industrie 4.0). Integrating data about the asset (object which has a value to an organization and is therefore managed individually) from different life cycle phases, a digital machine twin can provide valuable structuring of data and insights; from asset development and configuration all the way to the asset's end of life.



In the figure below, we can differentiate a type and an instance. All planning data creates the type of the asset. The **type** is used to implement the physical (real-world) instance of the asset. The planning activities are inclusive of all domains namely mechanical, electrical and software. A type becomes an **instance** when development and prototype production is completed, and the actual product is manufactured.

For example, the type of car includes CAD models, wiring diagrams, accessories which can be varied according to a desired use case. Thus, the type specifies which different car configuration are available in form of a 150% model.

When the customer selects their individual configuration, for example in an online shop, a virtual 100% model is created. The 150% model as well as the 100% model are still part of the product type. As soon as the 100% model is manufactured, an individual instance is created. Thus, the customer gets a real-world touchable car with a unique serial number. Customers will use their car instances and create specific and individual lifecycles with respect to their usage.

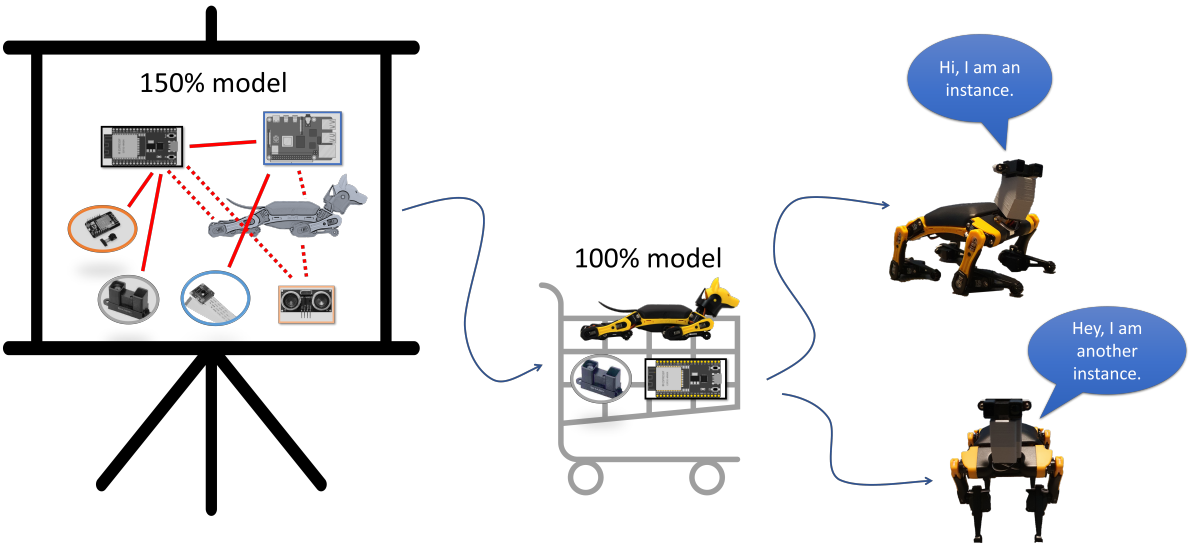
When the product is delivered to the customer, this marks the beginning of a product's lifetime or service time. During this, the product provides the service for which it was planned and manufactured. During service life, prescribed maintenance is suggested to avoid sudden breakdown of the product. Once the product's service time ends or the product is no longer required, the product is decommissioned, and this marks as end of life. During the end of life, the product can be recycled, up cycled, or decomposed according to requirements.

The **digital twin** includes data from the type (analysis, specification, and design phases) as well as data from the instance (implementation, system test, operation, and maintenance phases). Also, end of life phases must be included. The collected data can be analyzed and a predictive model can be created. This way, objects which are, for example, subject to wear can be replaced at the right time.

7.2 Creating a Digital Twin for a Robotic Dog

In this course, you will learn to use a robot dog "Bittle" to develop your individual robodog instance. You will collect data from your instance of the robot dog and use it to build a digital twin. This course is designed to explain the concept of digital twin and its application with interactive ready to buy modules with plug and play interface. Bittle is a robot dog that can do simple tasks, such as walking or waving. However, it cannot perform complex tasks, such as object avoidance or line following, out of the box. This must be implemented by expanding the existing robodog hardware and by creating additinal program code.

An overview of the type and instance of robot dog with additional modules can be visualized in the image below.



When looking at the robot type, you will see that different robot configurations are possible, depending on your hardware selection. The hardware selection possibilities can be organized according to different hardware modules:

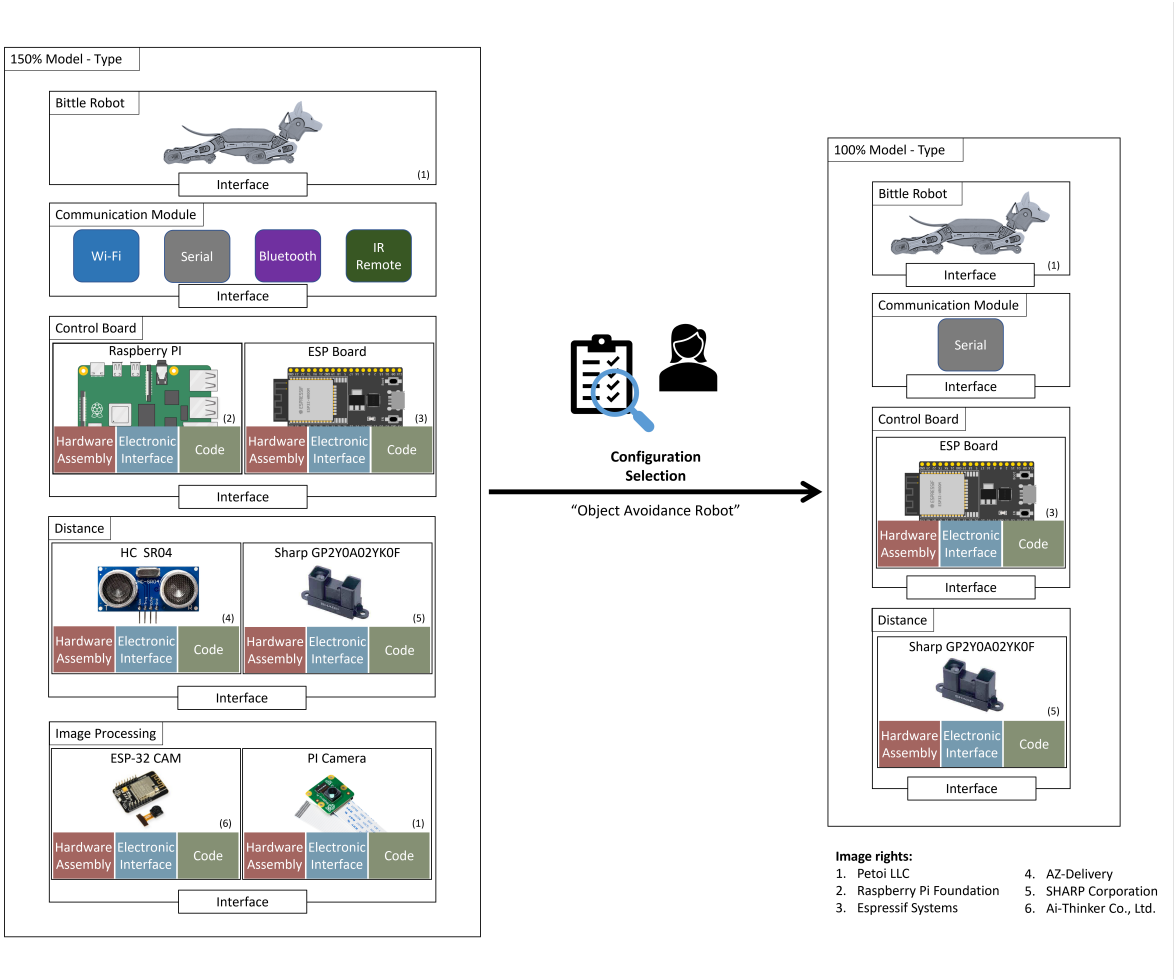
- Control board: This is the main brain of the robot.

- Locomotion hardware: Parts that help robot move in real world.
- Peripheral devices: Sensors and Actuators used to interact with the real world.
- Communication interface: Protocols used for sending and receiving information between control board and peripheral devices or locomotion hardware.

All hardware options as well as the base structure make up the **150% model** of the robot. Thus, the 150% model describes all possible configuration variants of the robot dog creating the DT type.

Depending on your use case, you can create a specific robot configuration (100% model). As an example, the following configuration is selected:

- Control Board -> ESP32
- Locomotion hardware -> Bittle dog
- Communication interface between control board and locomotion hardware -> UART
- Peripheral devices -> Sharp GP2Y0A02YK0F (Distance measurement)



When the product described by the 100% model is manufactured, assembled, and given a unique article number, it becomes an instance. Different instances of the same robot type will perform differently due various factors, such as battery charge, position of the sensor, the code snippet used etc.

Once the instance is ready, the data can be collected and stored. This data will help in analyzing the performance of that specific instance and can be used to compare with other instances or to improve the 150% model type. Thus forming

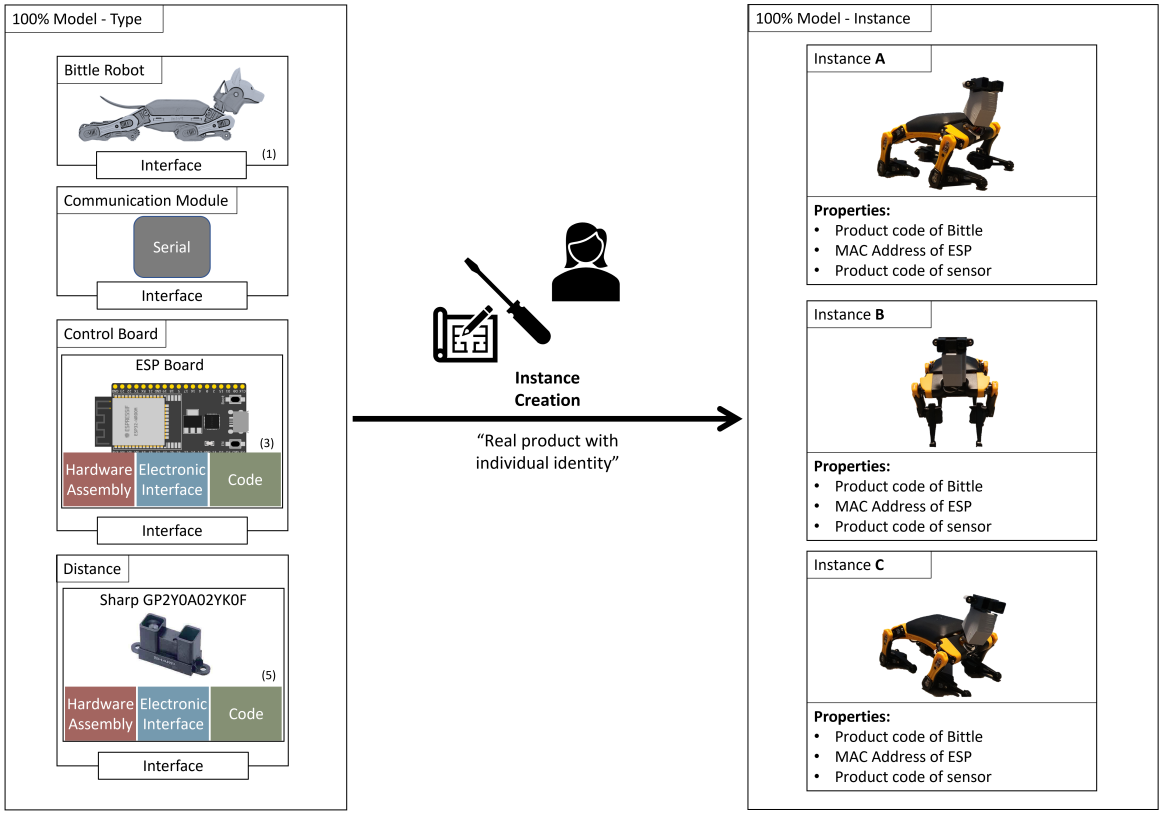


Image rights:

1. Petoi LLC
2. Raspberry Pi Foundation
3. Espressif Systems
4. AZ-Delivery
5. SHARP Corporation
6. Ai-Thinker Co., Ltd.

a DT.

CHAPTER

EIGHT

BITTLE

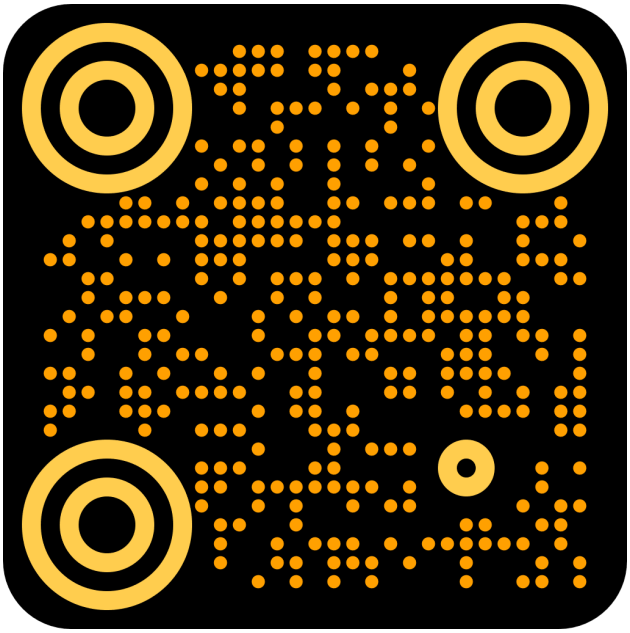


Fig. 8.1: QR Code to course

8.1 Learning Outcome

8.1.1 Introduction

8.1.2 Requirements

- *Arduino Basics*
- *Visual Studio Code Basics*
- *Introduction to Version Control with Git*

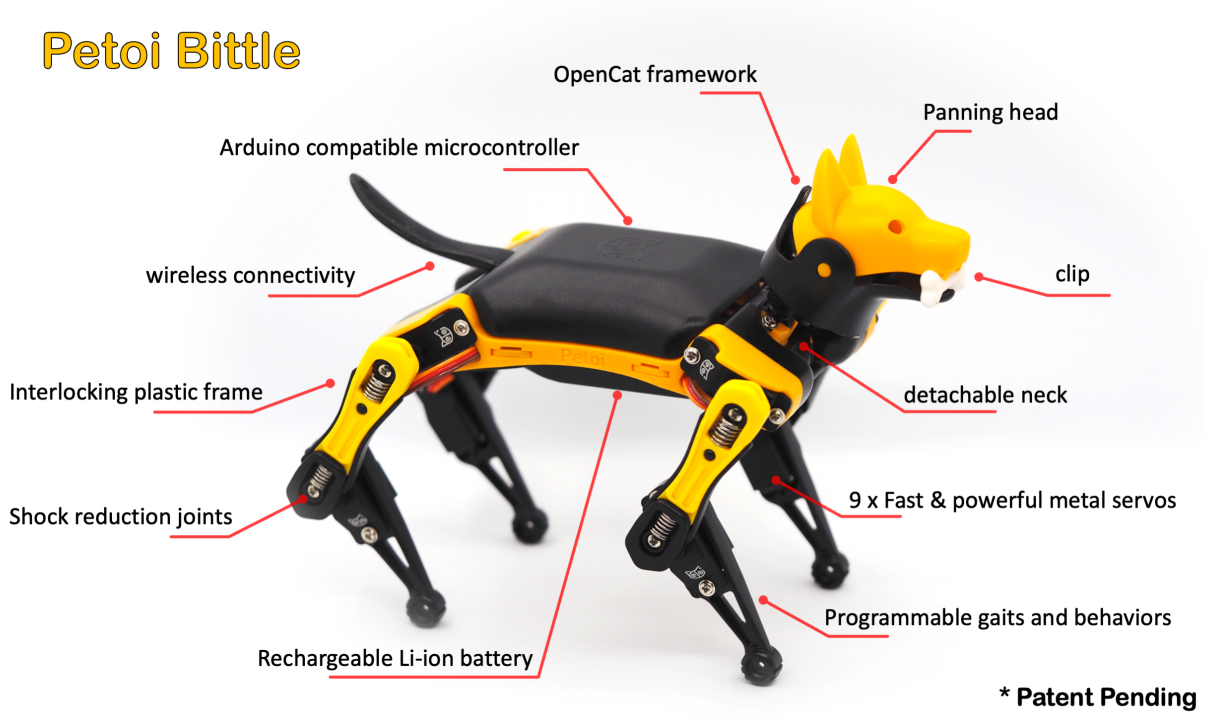


Fig. 8.2: Introduction

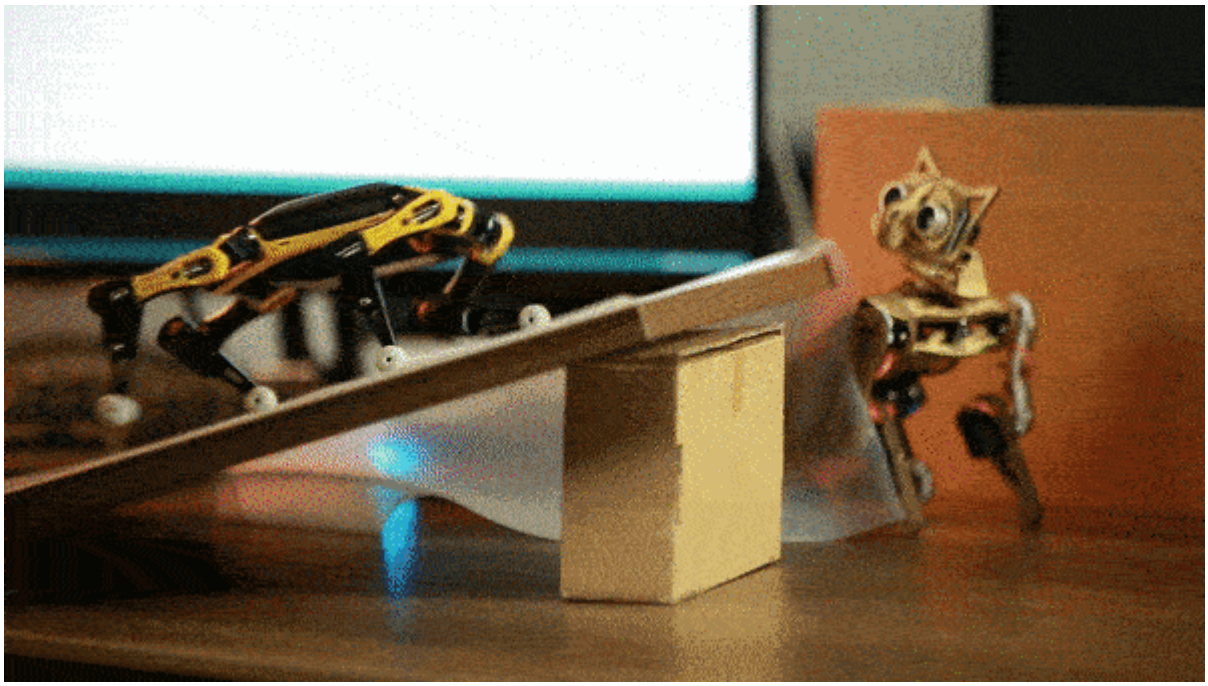


Fig. 8.3: Bittle demo

8.1.3 What you need

Hardware

- Bittle robot
- Arduino UNO
- Electronics Kit (including sensors)

Software

- Visual Studio Code
- Git Version Control
- Arduino IDE 1.8.X
- Node-RED

8.1.4 Sources and Resources

1. VS Code https://code.visualstudio.com/
2. GitLab https://git.fh-aachen.de/
3. GitLab Extension for VS Code https://marketplace.visualstudio.com/items?itemName=GitLab.gitlab-workflow
4. PlatformIO https://platformio.org/
5. Arduino IDE https://www.arduino.cc/en/software
6. Git https://git-scm.com/
7. Professional Git by Brent Laster (available online in University Library)
8. Petoi Bittle User Manuals https://bittle.petoi.com/
9. Bittle Online Help https://www.yuque.com/tinkergen-help-en/bittle
10. Node-RED https://nodered.org/docs/getting-started/
11. Git Book V2 https://git-scm.com/book/en/v2
12. Real-time C++ Fourth edition. ISBN 3-662-62996-8 (University Library)

8.2 Workspace Setup

In this section, all the required software is installed. Visual Studio Code is used for the code development.

Todo: Please download the presented software with the links provided in their respective sub-sections.

8.2.1 Visual Studio Code

Visual Studio Code is a source code editor. It supports various programming languages and can be combined with compilers to work as a package integrated development environment IDE.

VS Code has IntelliSense which allows word based completions. This you can write code syntax without typos.

You also get access to various extensions. These enhance the working of VS Code and also adds new features as per requirements.

Fig. 8.4: VS Code by Microsoft

Todo: Download and install Visual Studio Code.

Download Link: https://code.visualstudio.com/download

Todo: If you have not yet worked with Visual Studio Code before, follow this course on *Visual Studio Code Basics*

8.2.2 Arduino IDE Extension

The introductory programming sessions will be conducted using Arduino UNO board. The board can be programmed using C++. To write the code, the Arduino extension will be required in Visual Studio Code. This code will then be compiled using a GCC compiler and uploaded to the board using a flashing program. For uploading, the board has to be connected to the PC using a USB A (PC side) to USB B (UNO side) cable.

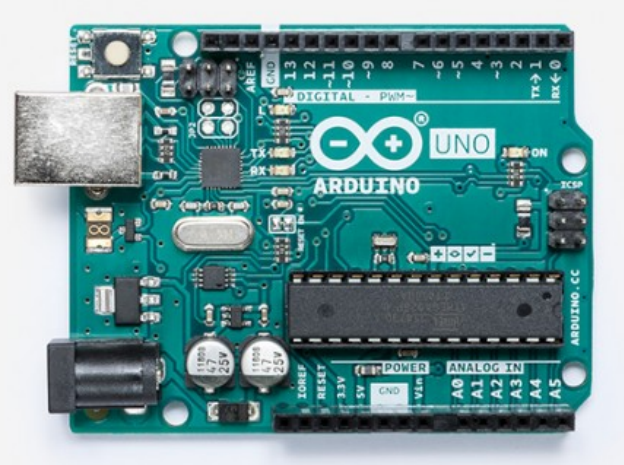


Fig. 8.5: Arduino Uno (ref: Arduino Store)

Todo: Download and install the Arduino IDE to program your Arduino Uno.

Download Link: https://downloads.arduino.cc/arduino-1.8.19-windows.zip

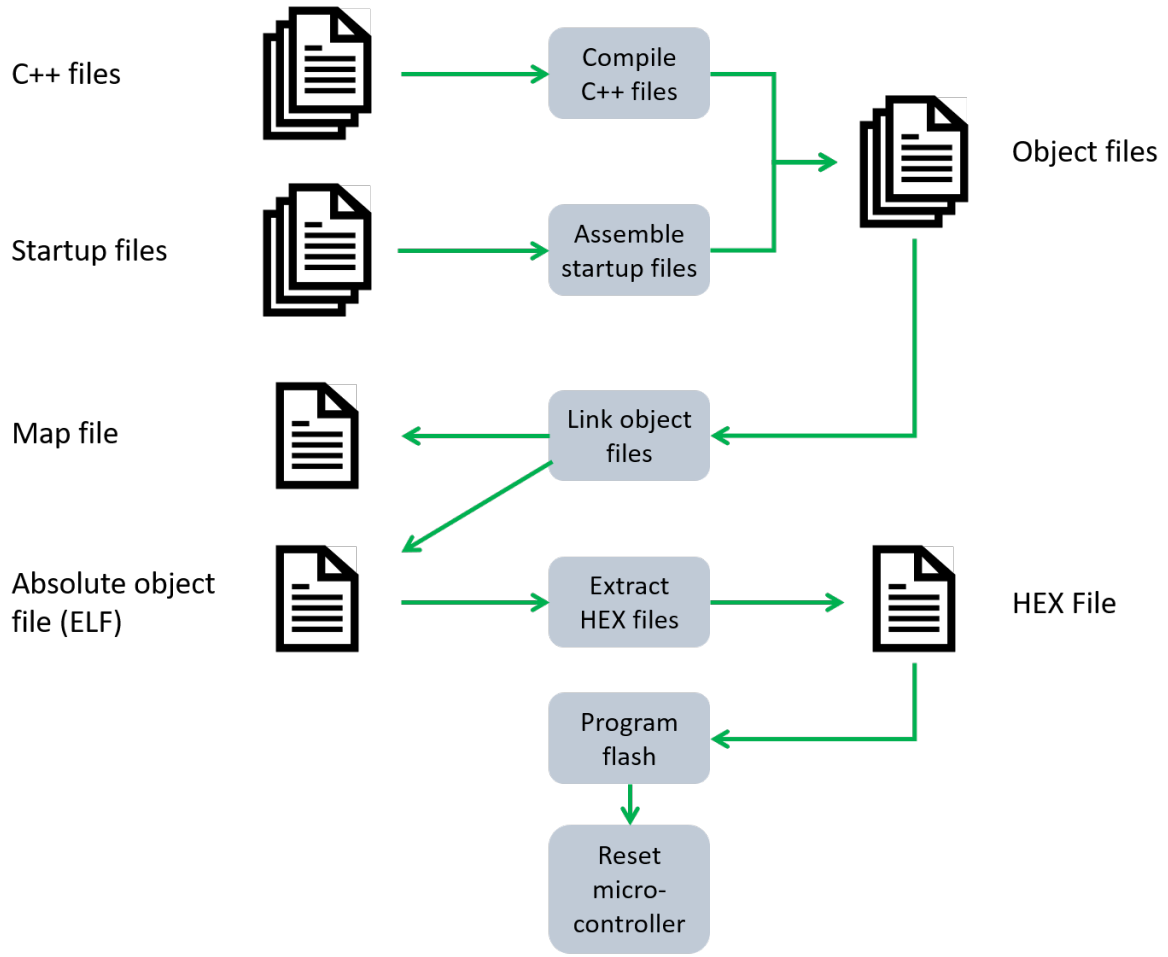


Fig. 8.6: Development Workflow (ref: Real-time C++)

Version: 1.8.X

Todo: Check if VS Code automatically detected the installed Arduino IDE.

If in case it doesn't happen, follow the steps below:

- Open **Extensions** from Activity Bar. Search Arduino extension.
- Click on **small gear** (manage menu) next to it. It will open a new menu.
- Click on **Extension Settings**. New tab will open.
- Scroll down and find **Arduino Path**
- Enter the complete installation address of your Arduino IDE. C:Program Files...

Hint: More information about the extension can be found in the *Visual Studio Code Basics*

Todo: Check out the Arduino examples provided with the libraries on the Side Bar. Open and test the Blink example.

Hint: Open **Explorer** and in the end a section named **Arduino Examples** can be expanded. Expand Built-in Examples -> 01. Basics -> Blink. A new window with the example will open.

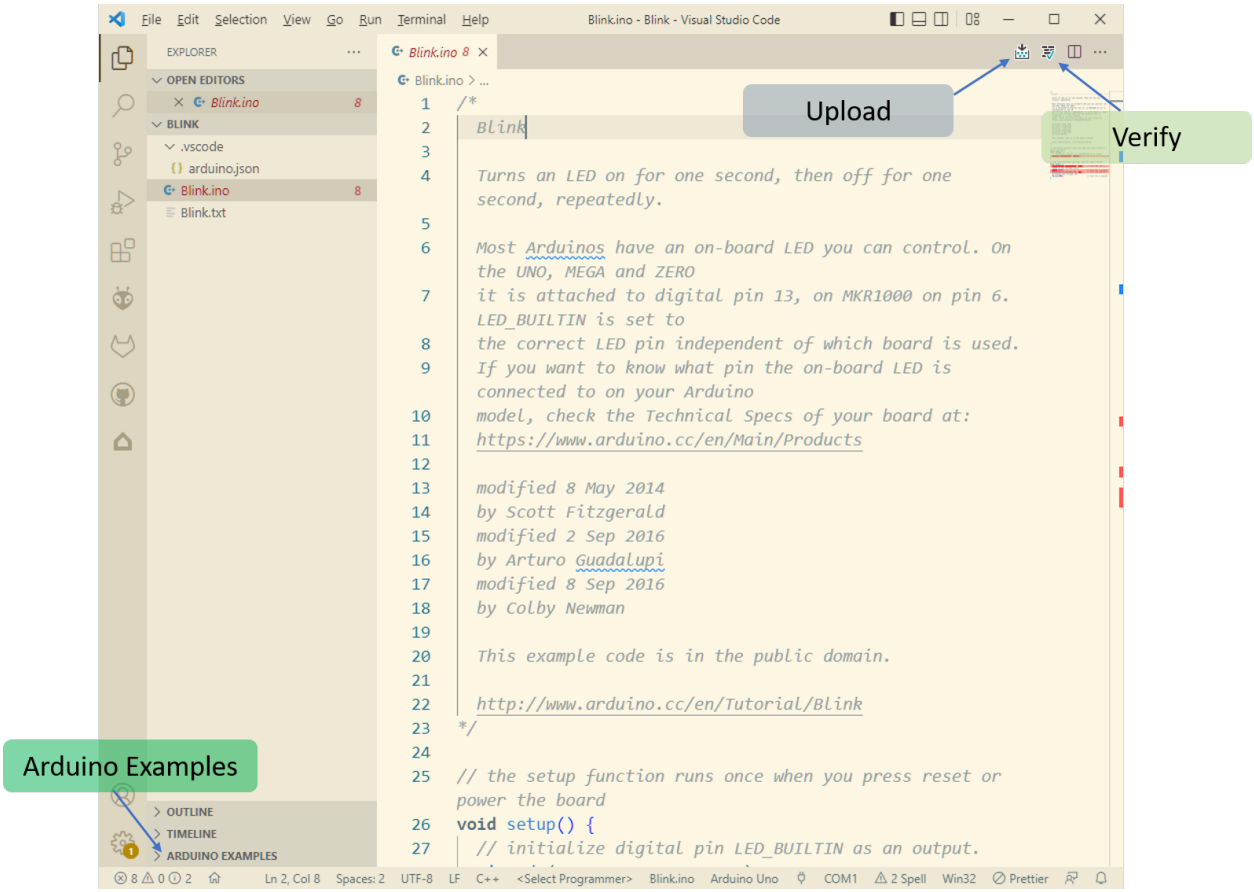


Fig. 8.7: VS Code Arduino UI

Todo: If you have not yet worked with the Arduino IDE before, follow this course on *Arduino Basics*

8.2.3 Version control using Git and GitLab

A version control is a kind of system which allows you to keep track of the changes that have been made to a code over a duration of time. This means that you can, at any given point in time, revert back to the older versions of the code you are working on.

Git is a popular VC software. Git works on Distributed Version Control Systems. This system provides everyone the copy of all files and allows user to edit them locally. The user can then work on the files locally and then upload the files to the server. The advantage of DVCS is that, if a server crashes, a local user can upload the files and make it running as they have a complete copy of the server data.

In this course, Git will be used as part of Visual Studio Code.

Todo: Download and install Git on your device

Download Link: https://git-scm.com/downloads

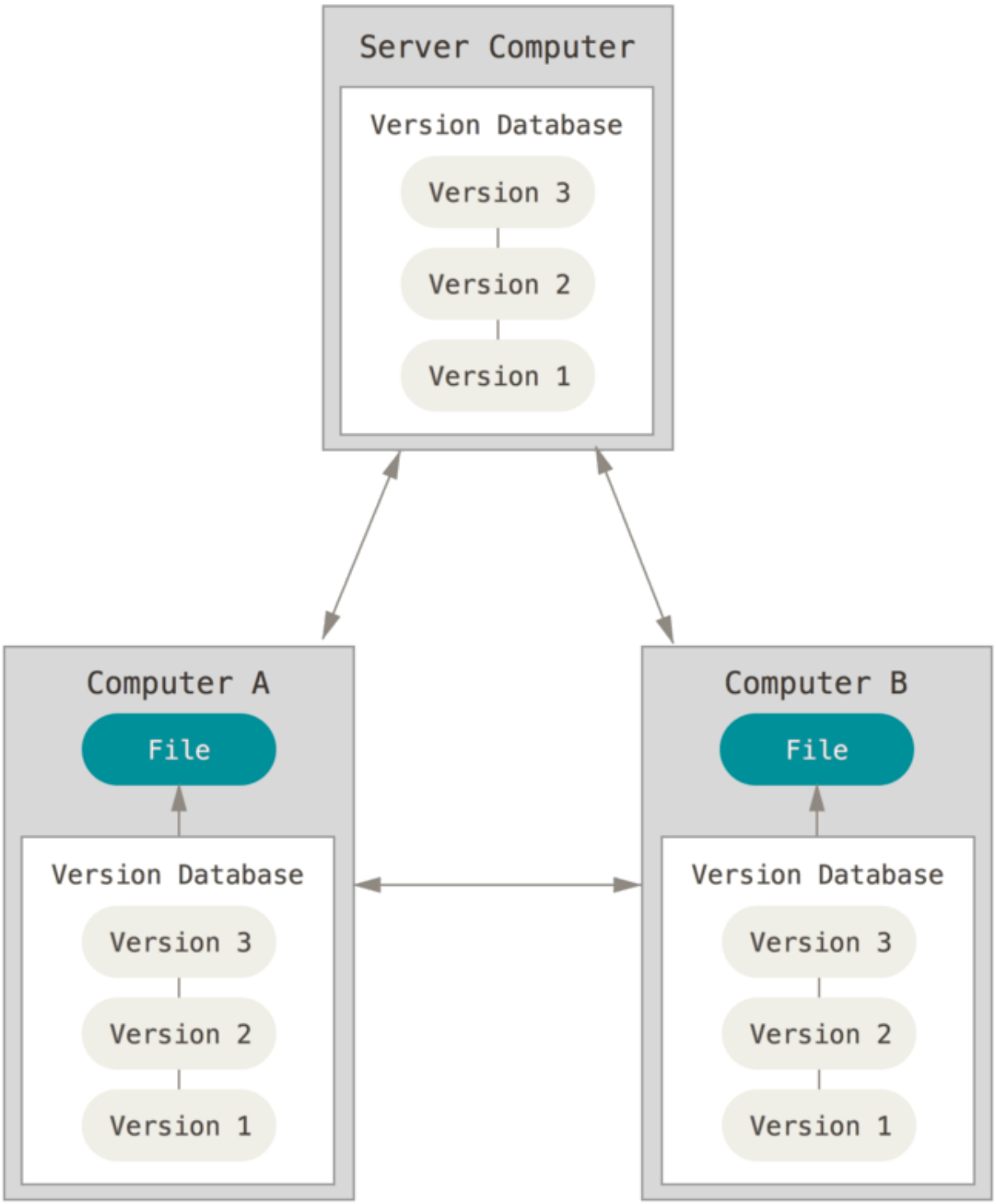


Fig. 8.8: Distributed Version Control System. (ref: Git Book V2)

Todo: If you have not yet worked with Git before, please follow this course on *Introduction to Version Control with Git*

GitLab is a DevOps tool used for hosting Git repositories. It allows collaborative team work to develop softwares. The difference between Git and GitLab or other platforms (Bitbucket, GitHub) is that these platforms allow users to upload their git projects online. This makes collaborative team work easy. As for git, you can even work with git on a local computer without the need of a hosting platform. The local git project will stay for your use only and would not be shared directly with your teammates.

We will be using GitLab for this course as our university hosts a server of it, thus making it easy for us to use the platform. To use the University's GitLab server, use the following link: https://git.fh-aachen.de/

You can check the access by logging on the following website using your FH-Kennung (FH identifier) (AB1234S) and password.

Todo: Create a profile for the FH-Aachen GitLab account and confirm that you have access to the GitLab server of your university.

Todo: Add your GitLab account to Visual Studio Code. You can follow the steps outlined in *Visual Studio Code Basics*

Todo: Create a new repository for your project. Share the repository with your teammates with proper access rights.

Todo: Create a new branch apart from Main (Master) branch. Work in the new branch only. **When you are sure about your work , merge the changes from the new branch to Main (Master) branch.**

8.2.4 Node-RED

Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. It provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click. (ref: Node-RED)

Todo: Download node-RED for your device and follow the **Quick Start** introduction: https://nodered.org/docs/getting-started/windows

Palettes is the menu on the left side. It contains all the available nodes that be used for designing a flow. The nodes are sorted with their use and the tree structure can be expanded and collapsed as per requirements.

External libraries can be installed on node-red using Palette Manager. You can open **Palette Manager** from the top right menu in node-red.

Todo: If you have not yet worked with Node-RED before, follow this course on *Node Red Basics*

Todo:

Please install the following palettes:

- node-red-node-serialport (https://flows.nodered.org/node/node-red-node-serialport)
- node-red-dashboard (https://flows.nodered.org/node/node-red-dashboard)

8.2.5 Everything ready?

Todo: Check if you have the following setup running on your PC:

- Visual Studio Code ready.
- Arduino IDE 1.8.X ready.
- Arduino Extension can be used in VS Code.
- Your GitLab account is accessible via VS Code.
- You have a repository created on the GitLab website and it's cloned on your PC using VS Code.
- Node-RED is running on your device and you can access it via your web browser.

8.3 Hardware Setup

8.3.1 Pre-assembled kit

Todo:

If you received the pre-assembled Bittle, you need to:

1. insert the neck into the body.
2. bend the knees to natural angles.
3. drag the curly wire from the knee side to the shoulder side to avoid squeezing when the knee joints rotate.
4. put the joints into the following posture before turning on the power.
5. long press the battery's button for 2~3 seconds to power on/off.

Warning: The pre-assembled Bittle is only coarse-tuned. You still need to calibrate Bittle's joint servos and final assembly to fine-tune its joints for the best performance.

If a small calibration is required, it can be done using App/Code. This will be done in the upcoming sections.

If the deviation is more than that possible with App/Code, you need to remove the corresponding part of the servo and re-install. Please refer this for details.

Fig. 8.9: Unassembled Bittle Kit and Pre-assembled Bittle.

Fig. 8.10: Rest position of Bittle before powering.

8.3.2 Base kit

Todo: If you received the a base kit with the single parts, you need to follow the tutorial provided here: https://bittle.petoi.com/4-Assemble-The-Frame

and here:

https://bittle.petoi.com/5-connect-wires

8.4 Bittle Body

Hint: The walk pattern of animals is explained nicely presented here: (https://www.animatornotebook.com/learn/quadrupeds-gaits)

Hint: Various other parts can be found here -> https://github.com/PetoiCamp/NonCodeFiles/tree/master/stl

Fig. 8.11: Standing dog

Danger: The robot is built using custom body parts. PLEASE be careful in removing and installing parts.

To access the main board, you need to remove the back cover of Bittle. The back cover has Click-Lock mechanism.

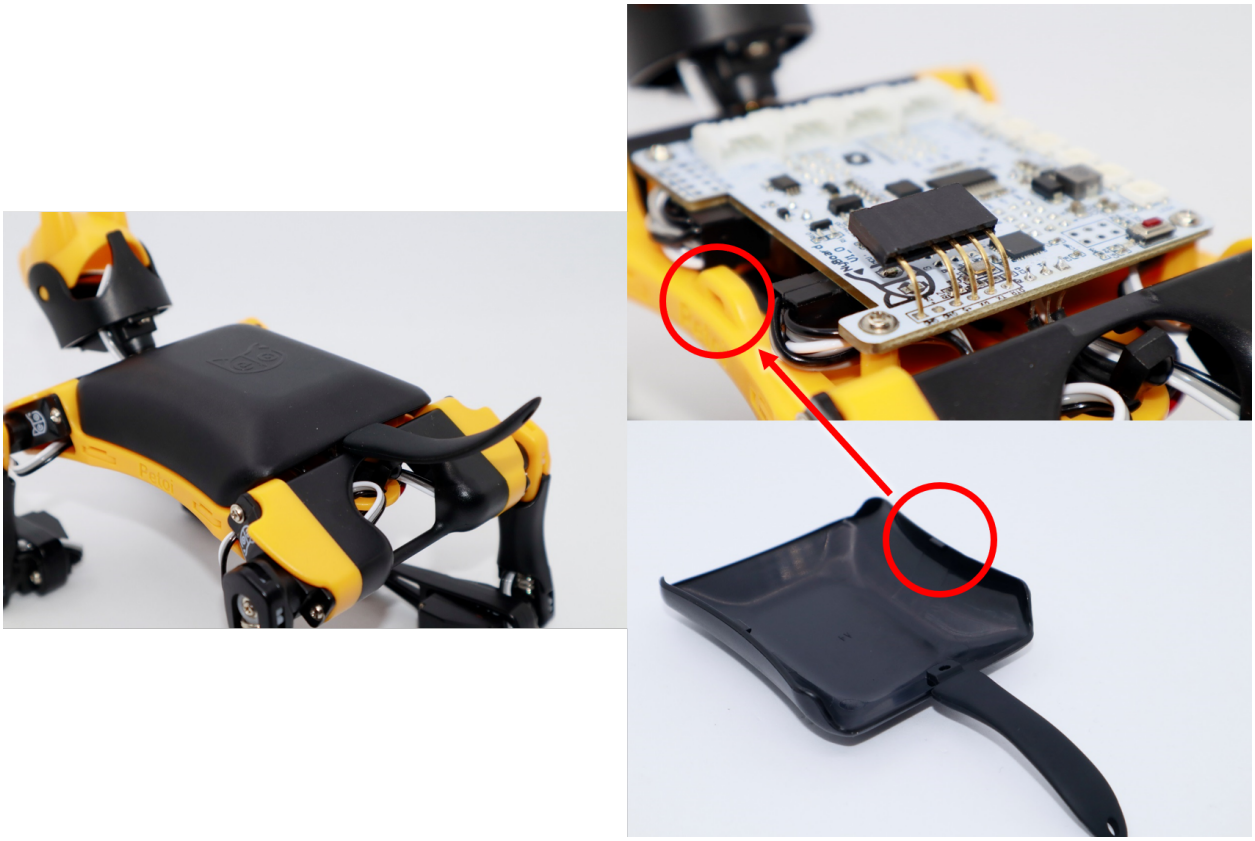


Fig. 8.12: Bittle with top cover

8.5 Electrical and Controller Properties

The components inside Bittle are:

- the main board on the top which is covered with a plastic shell.
- the servos which are installed at every joint. (The motor shaft depicts joints and the motor body is connected to the limb)
- the battery which is installed at the bottom.

Todo: Get familiar with the provided hardware.

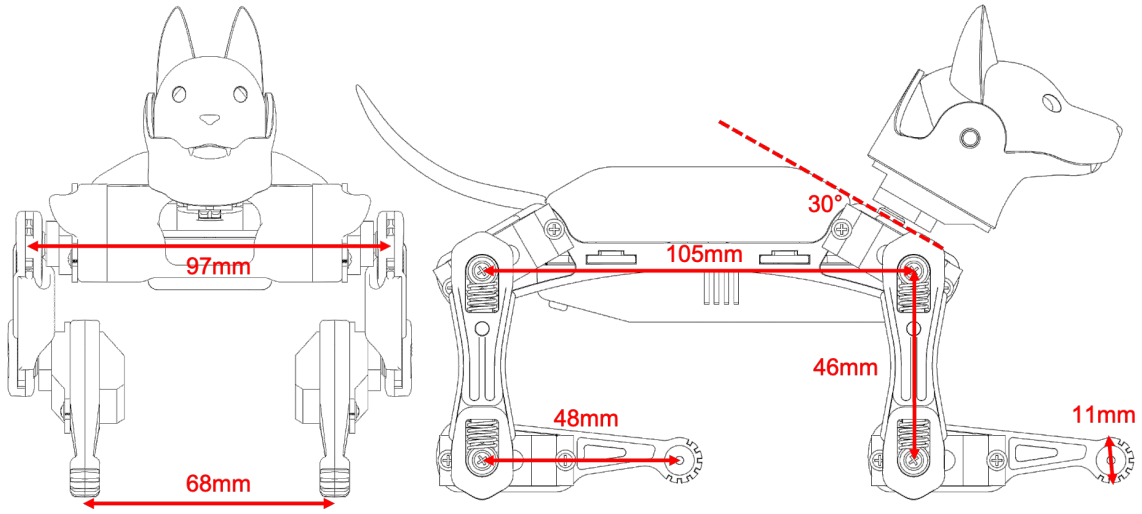


Fig. 8.13: Body Dimensions

8.5.1 Main Board

Specifications

Controller Specifications:

Mainboard Name	NyBoard V1_1
Microcontroller	ATmega328P
Operating Voltage	5V
Chip frequency / Clock speed	16 MHz
DC Current per I/O Pin	20mA

Memory usage:

Flash	32KB
SRAM	2KB
EEPROM	1KB
EEPROM (I2C)	8KB

External Interface:

- UART
- I2C (TWI)
- SPI

Additional on board components:

Fig. 8.14: Standing dog

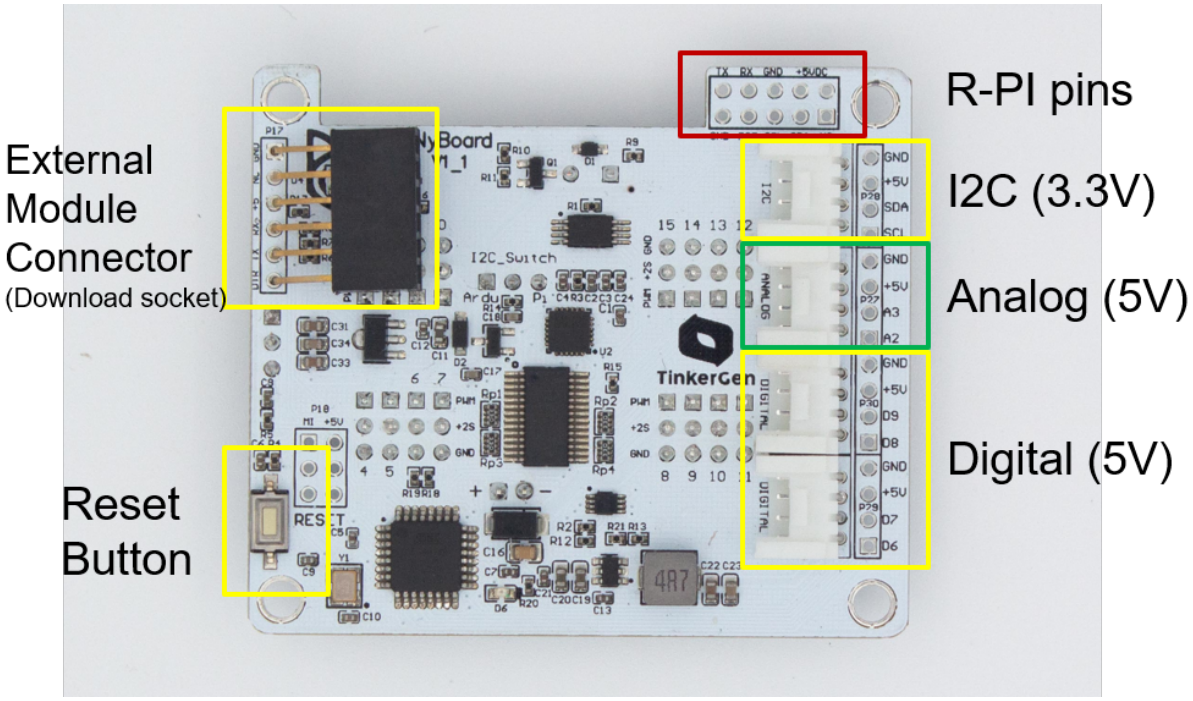


Fig. 8.15: Top view of main board

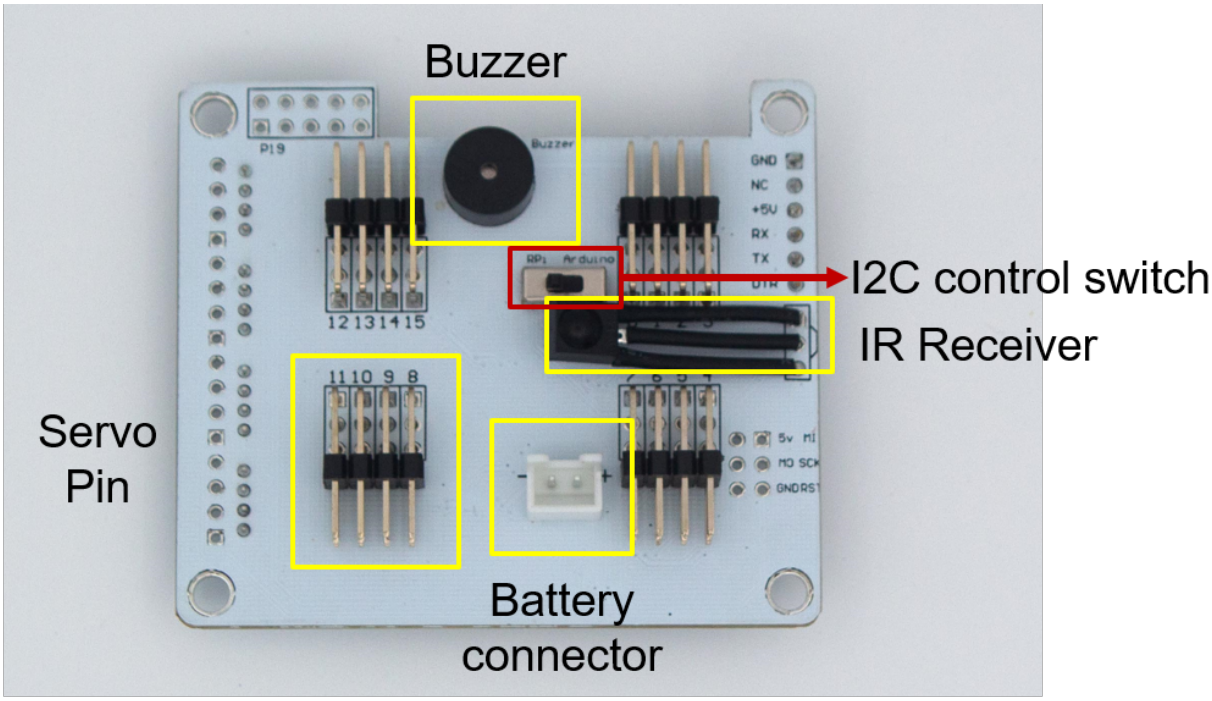


Fig. 8.16: Bottom view of main board

Component	Description	Protocol	Address / Pin
MPU6050	6 axis IMU	I2C	At 0x68
		Interrupt pin	D2
PCA9685	16 channel PWM controller	I2C	At 0x40
AT24C64	EEPROM 8KB	I2C	At 0x54
VS1838B	IR Receiver		D4
Buzzer			D5
Voltage divider			A7
G1	External I2C	I2C	
G2	External analog pins	Analog	A2, A3
G3	External digital pins	Digital	D8, D9
G4	External digital pins	Digital	D6, D7
LED	Single Green LED	Digital	D10

Board interface

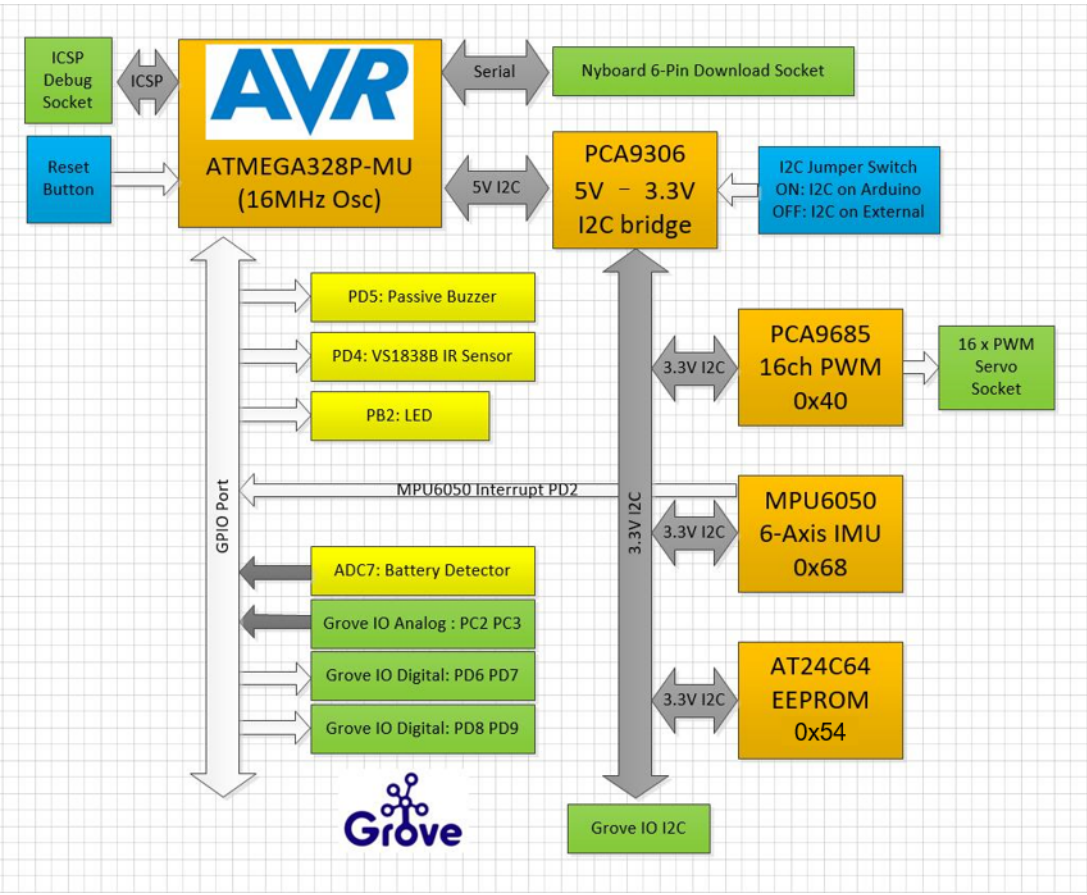


Fig. 8.17: Interface diagram

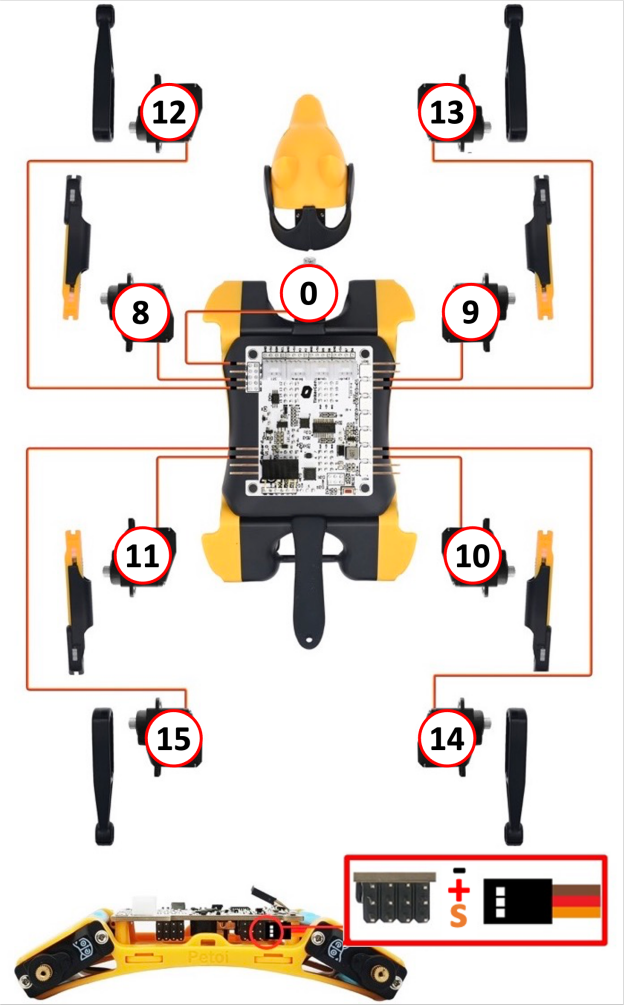


Fig. 8.18: Motor connection layout

Board power supply

The actual voltage is approximately 2x of the reading. A safe range of battery voltage is below 10V. You should charge the battery in time when the battery is lower than 7.4V.

$$Voltage_{real} = \frac{ADC_{reading}}{1024} \times 5.0 \times 2$$

The main chips are powered by a Low-dropout (LDO) linear regulator for noise removal and better stability. LM1117-5V and XC6206P-3.3V are used to power 5V and 3.3V chips. The 3.3V LDO is connected in serial after the 5V LDO for better efficiency.

There's a diode between the battery and LM1117-5V to prevent damage by the wrong connection. There's a self-recover fuse (6V 500mA) on the USB uploader to limit the current and protect the USB port.

The Raspberry Pi consumes much more power, so we choose TPS565201 DC-DC to provide a 5V 3A output. The peak output can be 5A and with high-temperature/current/voltage protection. It will cut off the power when the chip keeps outputting >4A and over 100 Celcius degrees until the temperature drops to normal.

The servos are powered by 2S Li-ion batteries directly. Pay attention not to short connect the power or any pins on the NyBoard.

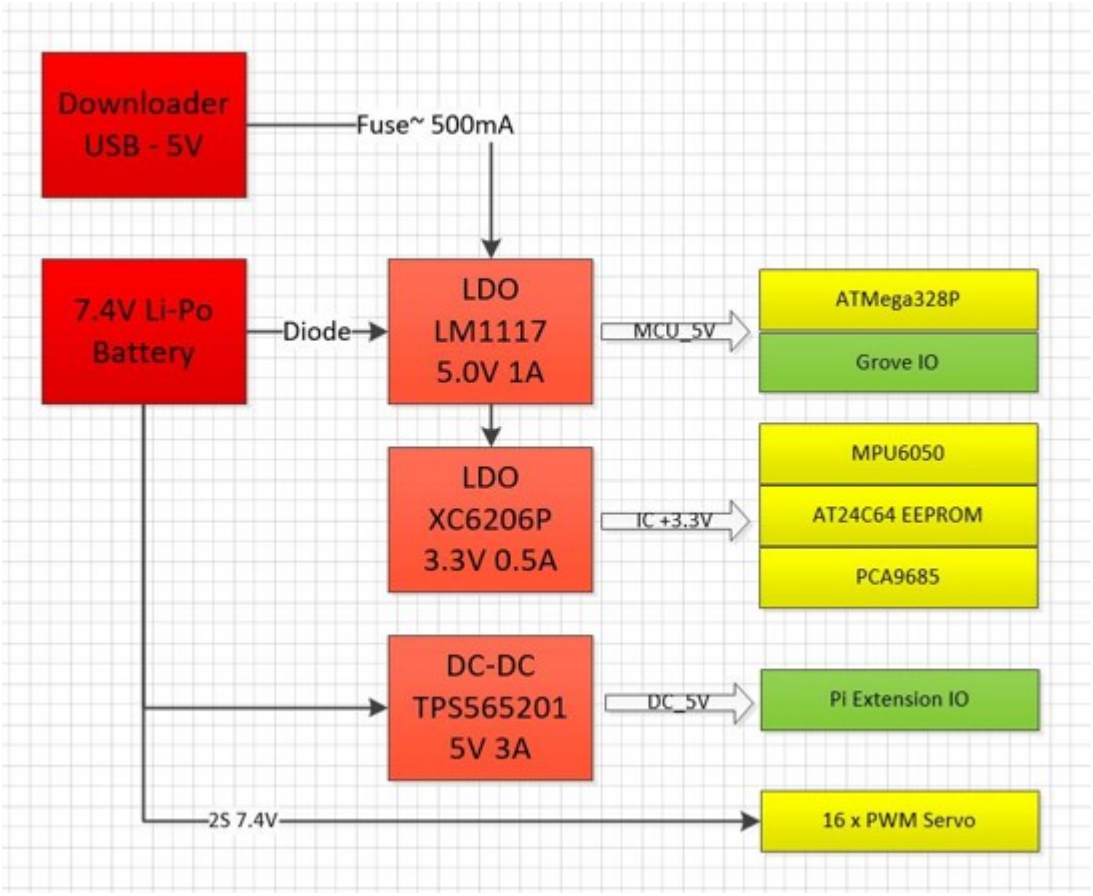


Fig. 8.19: Power supply diagram

8.5.2 Servos

The servos are used to execute the robot movements. These can be turned to the required angle as they have a built-in feedback circuit. Inside a servo motor, there is an electric motor which is in connected to a potentiometer using gear train. The potentiometer is used to measure the angle of rotation of the motor shaft. The wiring of almost all servos include VCC, GND and PWM input. PWM or Pulse width modulation is used to control the motor to the specific angle.

The controller controlling the servo sends pulse signal with duty cycle which nominally ranges between 1ms to 2 ms. For example, if at 1ms the motor turns to -90° and at 2ms the motor turns at +90°, then at 1.5 ms the motor will turn at 0°.

Fig. 8.20: Servo motor

8.5.3 Battery

Specifications of battery:

Battery type	Lithium Polymer
Number of cells	2 of 1000mAh
Total capacity	7.4Wh
Max discharge current	5A
Max output voltage	7.4V
Max recharge current	1A
Max recharge voltage	5V
Charging cable type	micro-USB

Fig. 8.21: Battery location and status LED.

Battery Status

Battery status LED color when Bittle is running or idle.

Blue	Charged
Red	Discharged

Charging status:

Battery status LED color when Bittle is charging.

Green	Charged
Red	Charging

8.6 Calibration

Calibration is done to remove misalignment of servo motors. If not done properly, the robot will have issues in walking due to imbalance and jerky movements.

To calibrate Bittle, there are five methods,

- Use Mobile App (**First choice**)
- Use Desktop App.
- Use Serial Monitor inside Arduino IDE / PlatformIO.

- long-press the battery and boot up the robot with one side up. It will enter the calibration state automatically.
- Press (calibrate) button on IR remote.

Note: The following steps in image are same for all methods.

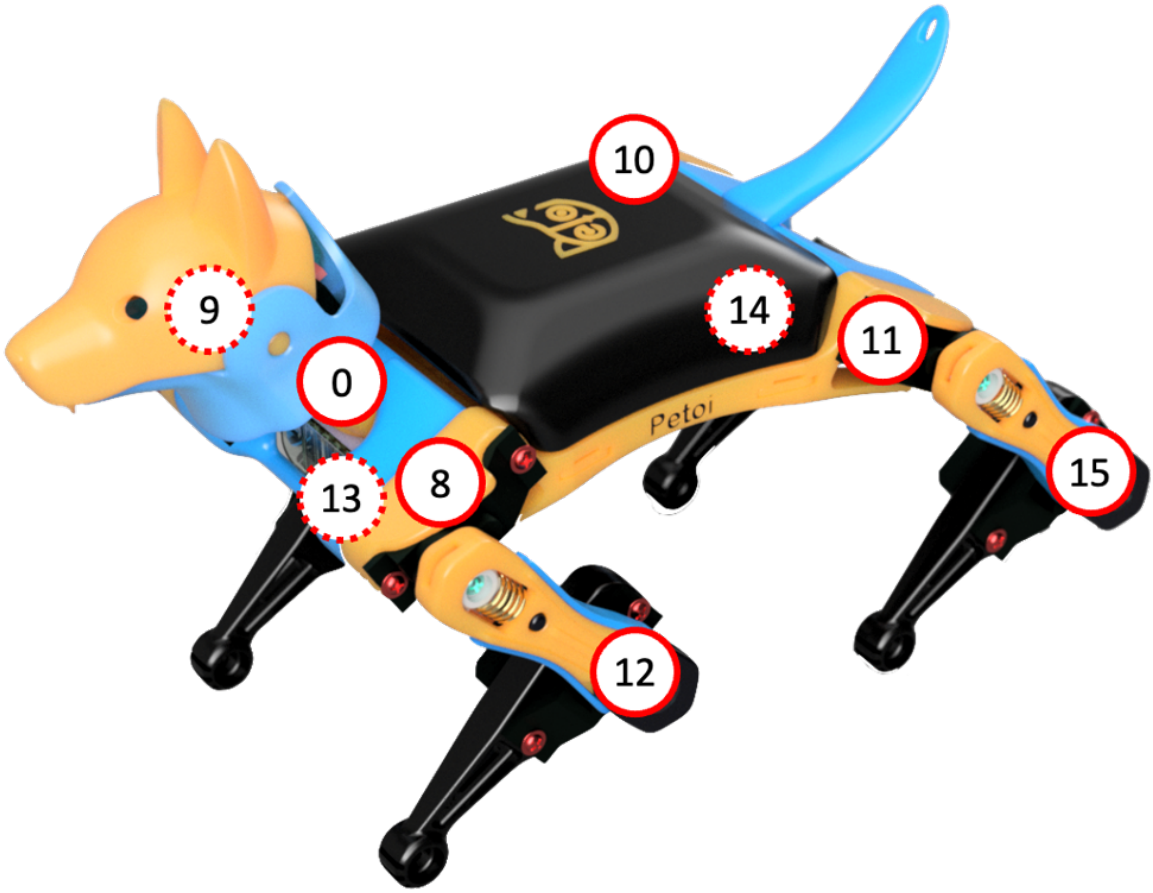


Fig. 8.22: Motor index for calibration.

Calibration Tool (L tool):

It's especially important that you keep a parallel perspective when calibrating Bittle. Use the 'L'-shaped joint tuner as a parallel reference to avoid reading errors.

Align the tips on the tuner with the center of the screws in the shoulder and knee joints, and the little hole on the tip of the foot. Look along the co-axis of the centers.

For each leg, calibrate the shoulder servos (index 8~11) first, then the knee servos(index 12~15). When calibrating the knee, use the matching triangle windows on both the tuner and shank to ensure parallel alignment.

Servo Gear:

The servo gear in the image below divides 360 degrees into 25 sectors (1 gear tooth = 1 section), each taking 14.4 degrees(offset of -7.2~7.2 degrees).

Center of mass:

Fig. 8.23: Calibration Tool placement (L tool).

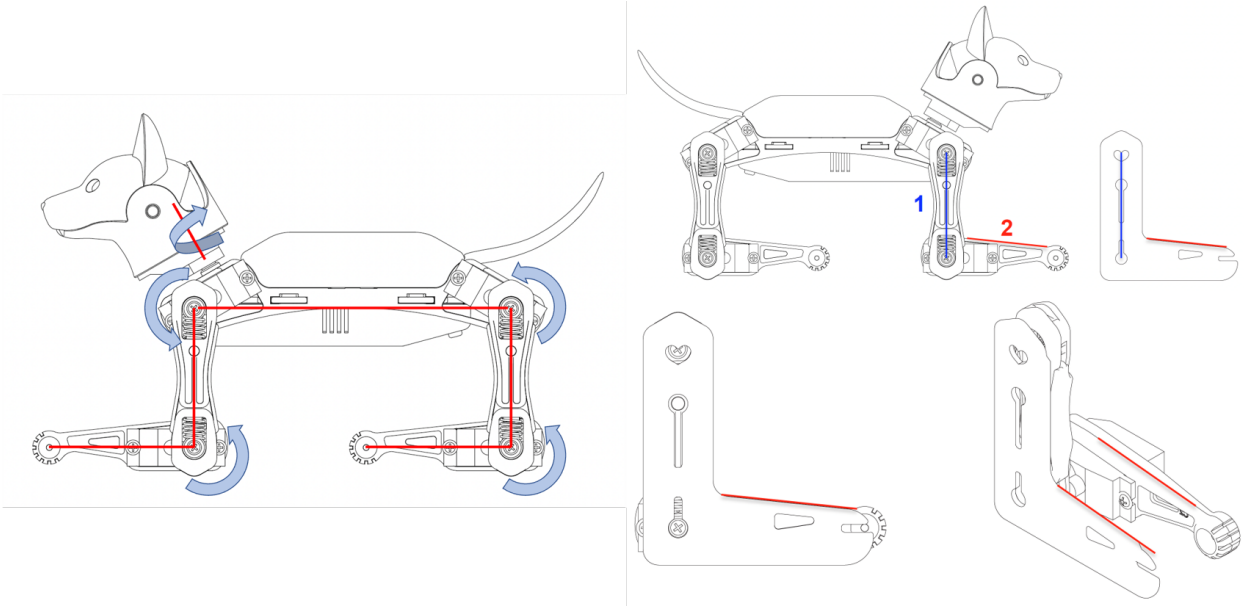


Fig. 8.24: Visualization of precise calibration (on left) & Usage of calibration tool (on right).

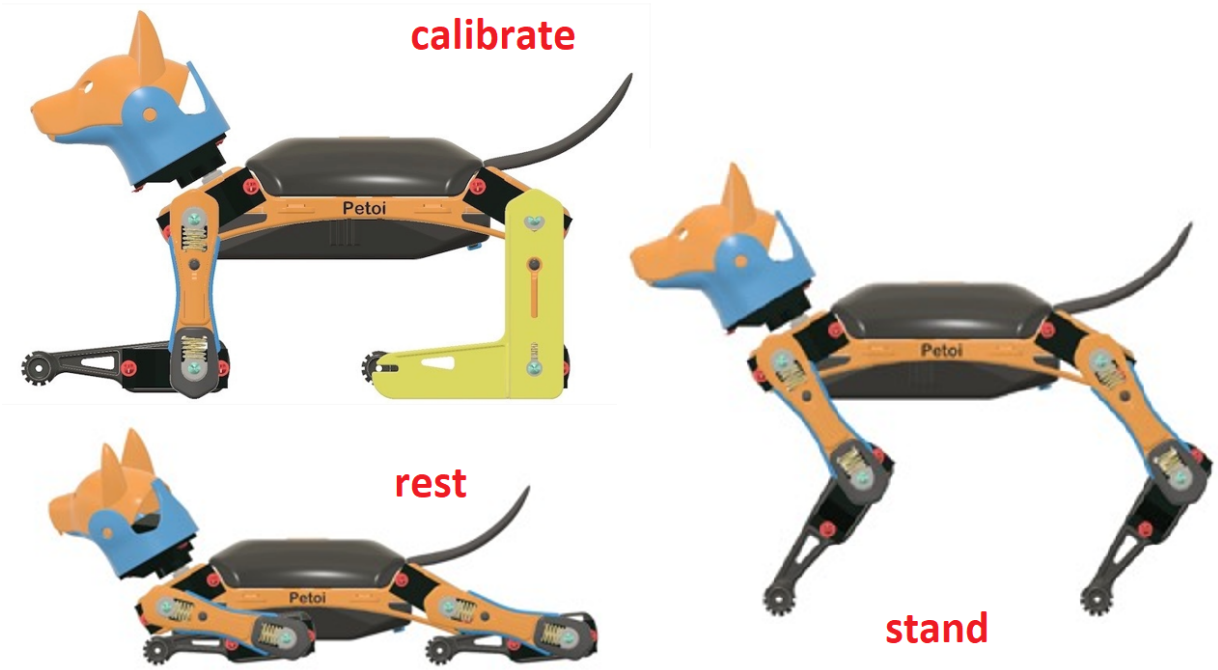


Fig. 8.25: Various postures of Bittle used for calibration.

Fig. 8.26: Servo gear.

Try to understand how the robot keeps balance even during walking. If you are adding new components to the robot, try your best to distribute its weight symmetrically about the spine. You may also need to slide the battery holder back and forth to find the best spot for balancing. Because the battery is heavier in the front, you can also insert it in a reversed direction to shift the center of mass more towards the back.

Danger:

Before calibrating, PLEASE check the following:

- All connections are tight and no loose wires are hanging from the robot.
- Battery is completely charged
- Additional adaptor used is connected properly and the pins are in exact order.

Danger: Please do not force the robot to add heavy objects, which may cause the servos to sweep or get stuck.

Note: If something is not working, please check *Problem Solving section*.

8.6.1 Using Mobile App

To use Mobile App for calibration, you need to install Bluetooth Adaptor to Bittle.

Warning: Pay close attention to the Bluetooth Adaptor's pin order in the image below.

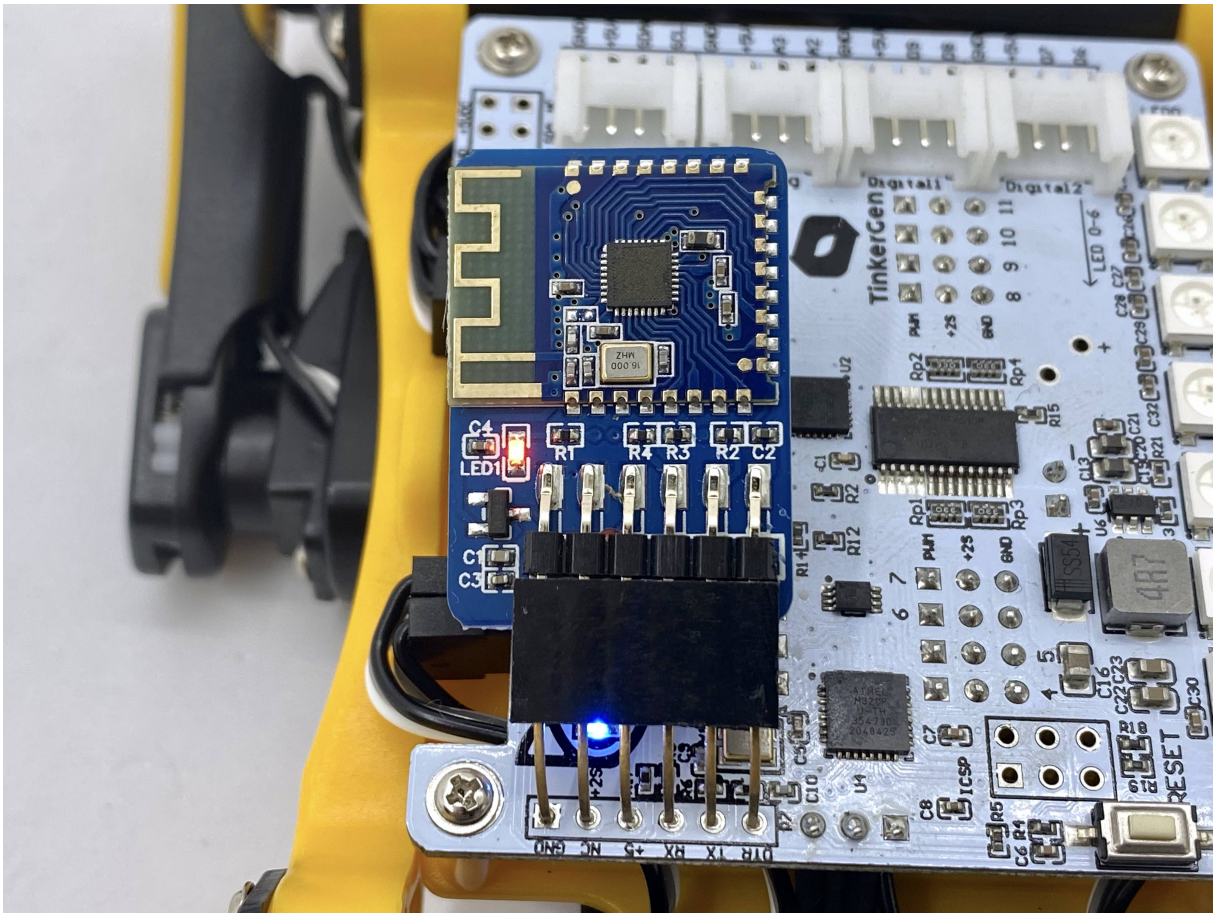


Fig. 8.27: Bluetooth Adaptor connection to Bittle

After connecting the adaptor and powering the Bittle, the LED on the adaptor should blink to indicate that it's waiting for connection. To pair it with app follow the steps below:

- Download the mobile app. Android ` and iOS <https://apps.apple.com/us/app/petoi/id1581548095>`_
- Open the app and scan available bluetooth devices inside the app. (**DO NOT** connect Bittle from Phone's system settings.)
- Grant permissions to the App to use Bluetooth services.
- Bittle may show with the following names in the App: Bittle, Petoi, or OpenCat.
- Select Bittle in Robot Selection (if asked by the app).
- Once done, you will be redirected to control dashboard.

Calibrate Bittle:

Tap the triple dot menu from TOP RIGHT corner. Select **Calibrate**. Follow the on-screen instructions to fine tune your robot.

After calibration, remember to tap the **Save** button to save the calibration offset. Otherwise, tap **<** (back) in the TOP LEFT corner to cancel the calibration values and set them to previous values.

Warning: If the offset is more than +/- 9 degrees, you need to remove the corresponding part of the servo and re-install. Please refer this for details.

8.6.2 Using Desktop App

To use Desktop App for calibration, you need to install USB Adaptor to Bittle.

Warning: Pay close attention to the USB Adaptor's pin order in the image below.

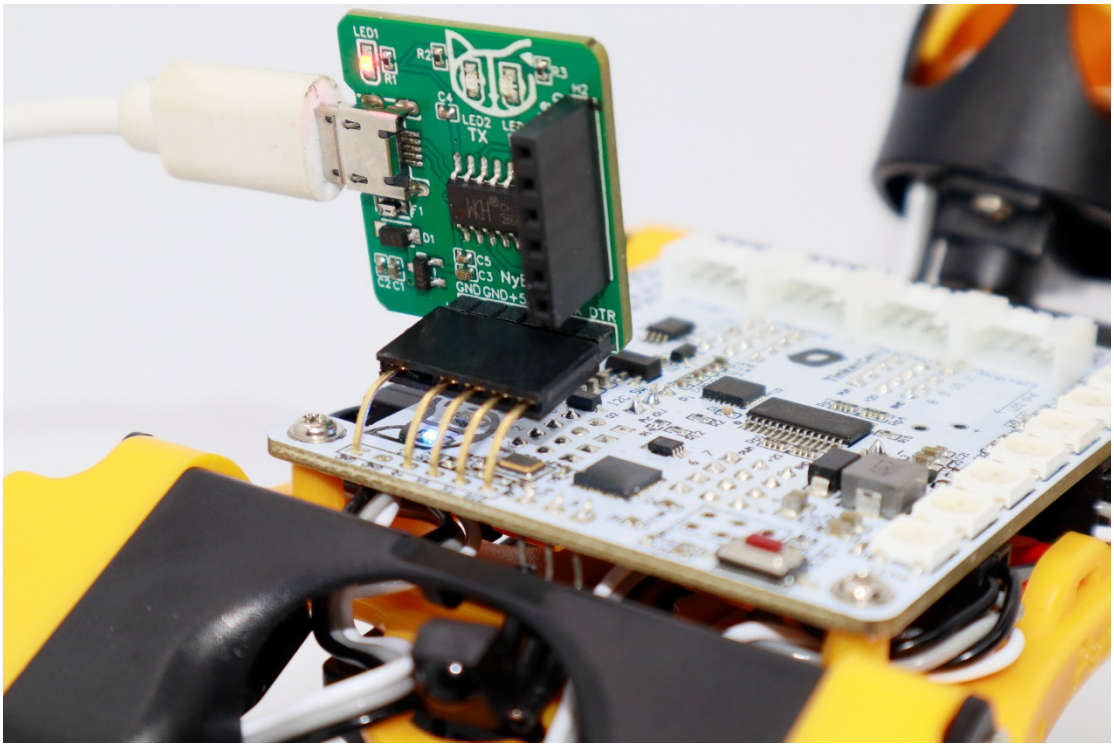


Fig. 8.28: USB Adaptor connection to Bittle

After connecting the adaptor and powering the Bittle, follow the steps below:

- Download the desktop app. Windows & Mac
- Open the app and select the **robot model** as **Bittle**. Select the interface language (is asked).
- Click on **Joint Calibrator**. You will be redirected to calibration interface.
- Click on **Calibration/Calibrate** button from interface to set all servos in calibration position.
- Select the Slider thumb and set it to required value to calibrate the motor angle using L tool.
- You can change the posture of the robot from stand to rest to test the motor calibration.
- After calibration, remember to click the **Save** button to save the calibration offset. Otherwise, click **Abort** (back) in the TOP LEFT corner to cancel the calibration values and set them to previous values.

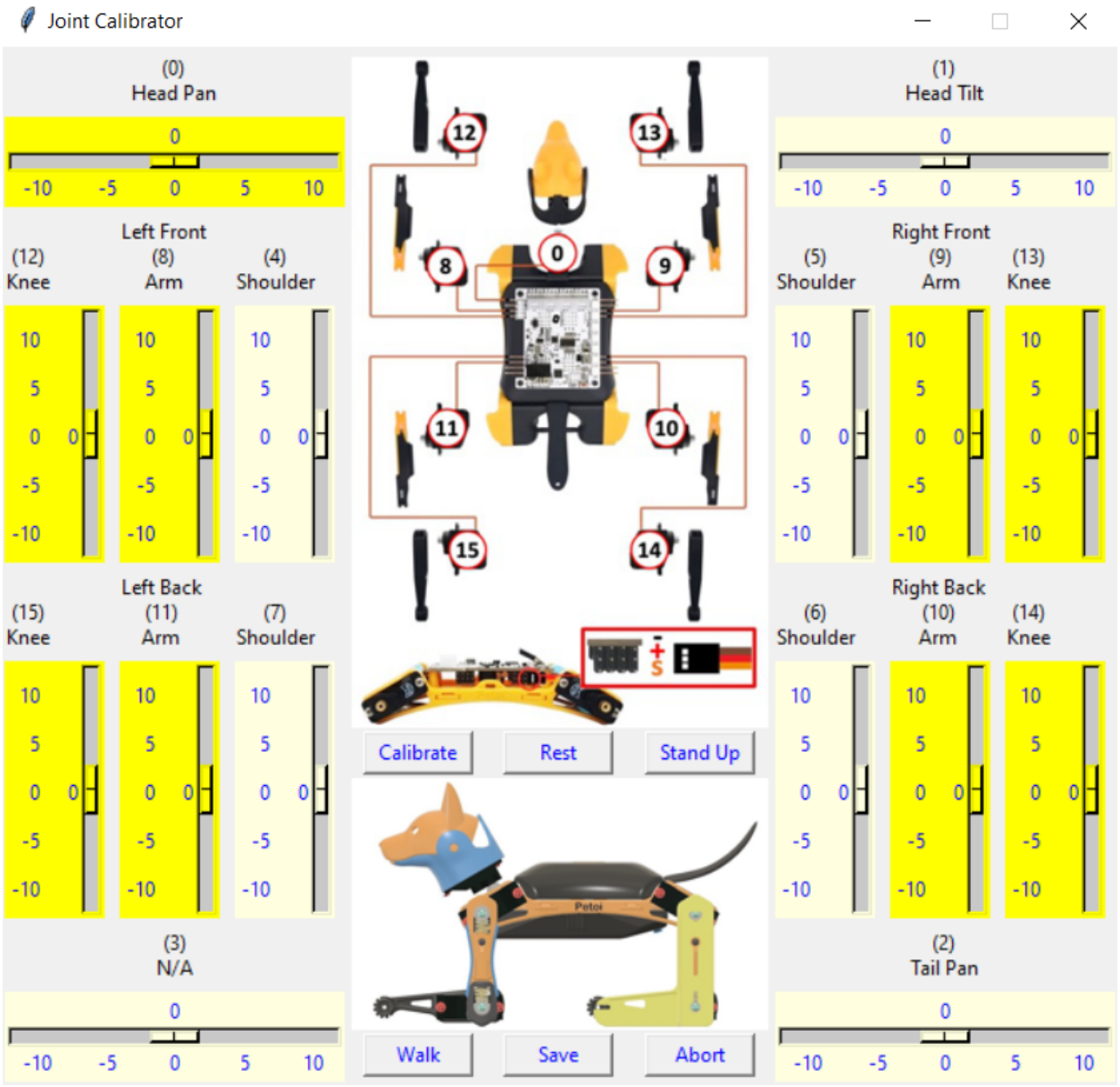


Fig. 8.29: Desktop calibration interface

Warning: If the offset is more than +/- 9 degrees, you need to remove the corresponding part of the servo and re-install. Please refer this for details.

8.6.3 Using Serial Monitor

To use Serial Monitor for calibration, you need to install USB Adaptor to Bittle.

Warning: Pay close attention to the USB Adaptor's pin order in the image below.

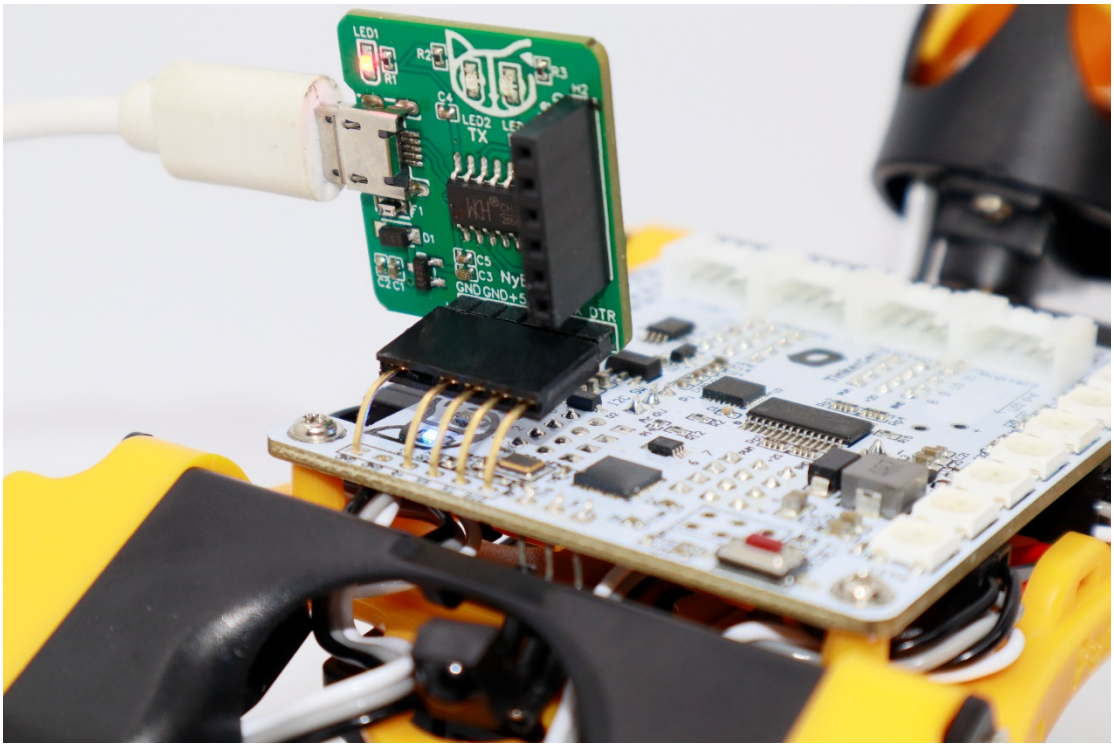


Fig. 8.30: USB Adaptor connection to Bittle

After connecting the adaptor and powering the Bittle, follow the steps below: (Image attached below for reference)

- Download the Arduino IDE / PlatformIO
- Open IDE, select COM Port and open serial monitor. Select Baud rate as **115200** and line ending as **no line ending**.
- Enter **c** in serial monitor to enter calibration mode.
- You will receive 2 rows of numbers. The first row signifies the **index of the motor**, the second row signifies the current calibration offset value.
- To set a custom offset value, use the following -> c<servo index> <offset value> example **c8 6** means giving the 8th servo an offset of 6 degrees.
- Use the L tool to set the angles to required value to calibrate the motor.

- Once done, type **s** to save the calibration values. (You can even save every time after you're done with one servo.)
- Wait until you see **Ready** on serial monitor.
- After calibration, type **d** or **kbalance** to validate the calibration.

Warning: The resolution of the correction amount is 1 degree, do not use decimals.

```
c

0,      1,      2,      3,      4,      5,      6,      7,      8,      9,      10,     11,     12,     13,     14,     15,
-1      -1      -1      -1      -1      -1      -1      -1      -1      -1      -1      -1      -1      -1      -1      -1
c0      0
```

Fig. 8.31: Sample serial control

Warning: If the offset is more than +/- 9 degrees, you need to remove the corresponding part of the servo and re-install. Please refer this for details.

8.7 Controlling Bittle

To control Bittle, there are five methods,

- Use IR Remote (**First choice**)
- Use Mobile App
- Use Arduino IDE / PlatformIO
- Use Desktop App
- Use Python Scripts

Note: To ZOOM the image, **Right Click** the image and click **Open in New Tab**.

8.7.1 Using IR Remote

The IR command map is defined inside **OpenCat/src/infrared.h**. In the file, the commands are described as **#define KXX command** . For example, K00 for the 1st row and 1st column and K32 for the 4th row and 3rd column.

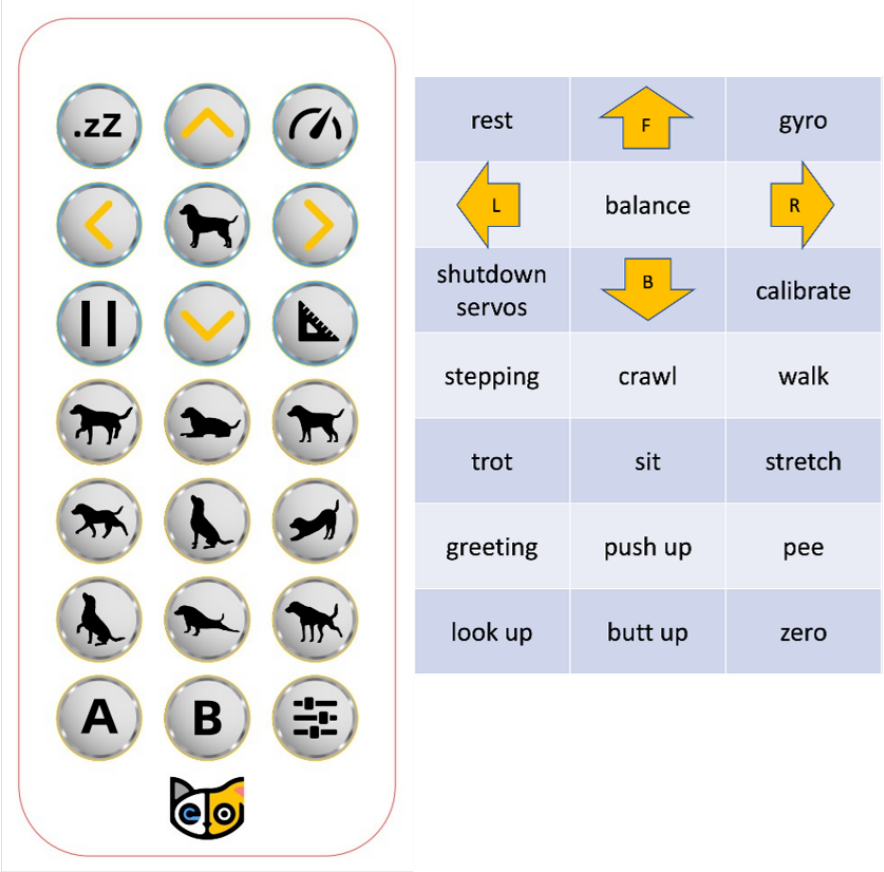


Fig. 8.32: IR controls

- **Rest** puts the robot down and shuts down the servos.
- **Balance** is the neutral standing posture.
- Pressing **F/L/R** will make the robot move forward/left/right
- **B** will make the robot move backward
- **Calibrate** puts the robot into calibration posture and turns off the gyro
- **Stepping** lets the robot step at the original spot
- **Crawl/walk/trot** are the gaits that can be switched and combined with the direction buttons
- Buttons after trot are preset postures or other skills
- Gyro will turn on/off the gyro for self-balancing. Turning off the gyro can accelerate and stabilize the slower gaits. But it's NOT recommended for faster gaits such as trot. Self-righting will be disabled because the robot no longer knows it's flipped.
- Different surfaces have different friction and will affect walking performance. The carpet will be too bushy for the robot's short legs. It can only crawl (command kcr) over this kind of tough terrain.
- You can pull the battery pack down and slide along the longer direction of the belly. That will tune the center of mass, which is very important for walking performance.
- When the robot is walking, you can let it climb up/down a small slope (<10 degrees)

8.7.2 Using Mobile App

The mobile app has the following dashboard.

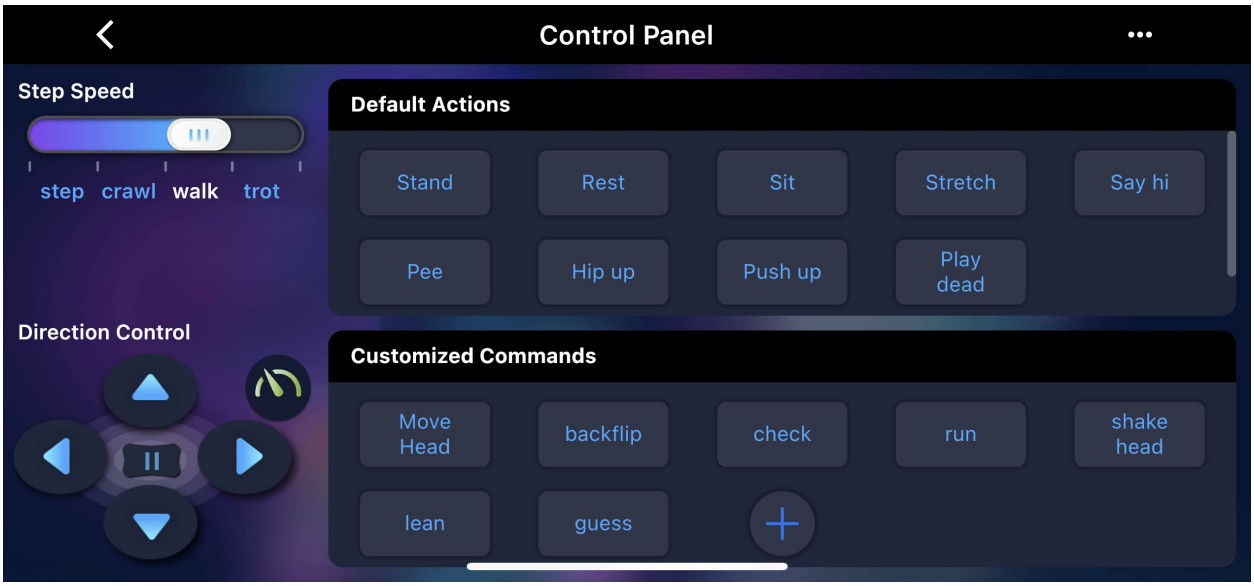


Fig. 8.33: App dashboard

The app allows the following functions and controls:

Gaits:

The left panel sets both the robot's gaits and directions and send combined command, such as "walk left" and "trot forward". The robot will only move if an initial gait and direction are selected. The "step" has no direction, and "backward" has left and right directions.

The **Pause** button "||" will pause the robot's motion and turn off the servos, so that you can rotate the joints to any angle.

The **Turbo** button (green dial icon) turns on/off the gyro which is used to detect the robot's body orientation. Turning it **on** will make the robot keep adjusting to body angles, and will know when it's upside down. Turning it **off** will reduce calculation and make it walk faster and more stable.

Default Actions:

The built-in postures and behaviors can be triggered by pressing the buttons.

> **Warning:** Don't press the button too frequently and repeatedly. Allow some time for the robot to finish its current tasks.

Customized Commands

You can also define customized commands by pressing the "+" button. Long-press a custom command button to edit it. There's a lite serial console to test the command and even configure the robot.

These commands are created using the Serial Commands defined in "Using IDE" section. For example, **i 8 -30 12 40 0 35** means that motors at 8, 12, 0 will move to angles -30, 40 and 35 respectively.

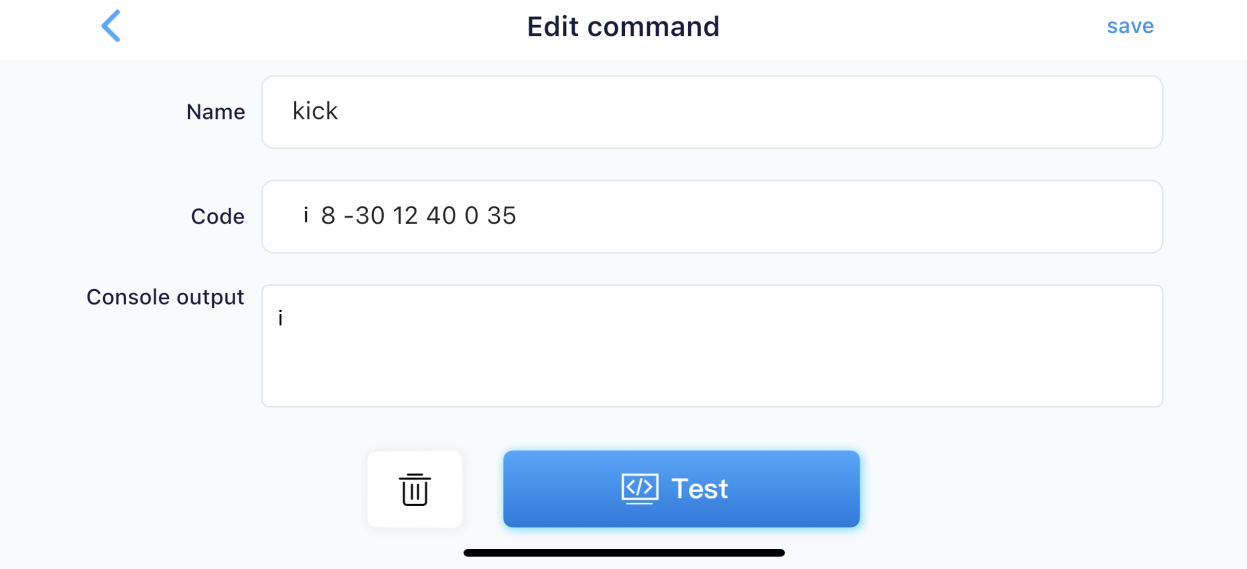


Fig. 8.34: Custom action command

8.7.3 Using IDE

This section mainly focuses on controlling the robot using serial communication. There are pre-defined protocol which uses ASCII encoding. The commands are case sensitive.

Few example commands are:

- **ksit**
- **m0 30**
- **m5 -15**
- **kbalance**

The complete command list is provided in the cheat sheet section.

Note: Some commands, like the c and m commands can be combined.

For example:

Successive “m8 40”, “m8 -35”, “m 0 50” can be written as “m8 40 8 -35 0 50”. You can combine up to four commands of the same type.

To be exact, the length of the string should be smaller than **30 characters**.

You can change the limit in the code but there might be a systematic constraint for the serial buffer.

8.7.4 Using Desktop App

The desktop app has the following dashboard.

Fig. 8.35: App dashboard

The app allows the following functions and controls:

Joint Controller:

The center of this area is a schematic diagram marked with the index number of the joint servo. Control slider is used for changing the degree of servo. The number above the slider indicates the angle value of the servo rotation in real-time. The number below the track indicates the angle range-180 ~ 180of the servo rotation, though the boundaries may not be reachable by the real robot. Press and drag the slider with the mouse to adjust the rotation angle of the servo, and the robot's joints will move in real time.

There are 2 linkage buttons corresponding to each servo (forward linkage symbol "+", reverse linkage symbol "-"). After clicking the linkage buttons of multiple servos, press and drag the control slider of one of the servos with the mouse:

- The rotation direction of the joint servo corresponding to the forward linkage button **+** is the same as the rotation direction of the servo controlled by the mouse.
- The rotation direction of the joint servo corresponding to the reverse linkage symbol **-** is opposite to the rotation direction of the servo controlled by the mouse.

"Unbind All" button: It is used to cancel the linkage of all joint servos at once.

Control the robot body pose sliders as follows:

Global Orientation and Translation	Effect
Pitch	Adjust the pitch angle
Roll	Adjust the roll angle
Spinal	Move in the spinal direction
Height	Raise or lower the robot's body

State Dials:

The state dials are used to connect the device either using USB connecting or via Bluetooth. You can select the appropriate COM port of your robot from the drop down list.

By default the software will be in **Listening** mode and will keep looking for devices to be connected. Once connected, the status changes to **Connected**. You can disconnect and disable auto-lookup by clicking on the same status. (Its a button as well.)

Note: Please select the required PORT from the list. Do not leave it to **ALL** option.

When a device is connected, three Button are available for user -> **Servo**, **Gyro** and **Random**. The button description is mentioned below:

Dial	On State	Off State
Servo	All servos enter the locked state, and	All servos are unloaded, and the joints
	do not manually adjust the robot	can be rotated at will, which is convenient
	posture at this time	for users to manually adjust the robot's posture
Gyro	The robot continuously calculates its	The balancing calculation is skipped to make the
	body orientation and try to keep balance.	loops faster. The gaits will be accelerated.
	It will self-right after falling over.	The robot won't self-right when you flip it.
Random	The robot will randomly move every few	The robot won't move on its own.
	seconds if you have uploaded the firmware	
	with RANDOM mode.	

Preset Postures:

The Preset Postures are shortcuts for some preset poses of each robot. After clicking these posture buttons, both the robot and the sliders will update to the new joint angles.

Skill Editor:

Danger: CONTENT UNDER REVIEW.

8.7.5 Using Python Scripts

This section allows you to send commands using python interface. The commands can be the pre-defined skills like *kbalance* or your custom movement commands which are developed using the base serial commands like *m 0 -30 0 30*

To begin, open Terminal / Command Prompt CMD and navigate to **OpenCat/serialMaster**. (Please refer **Download Base Firmware** for download information.)

To run commands, there two methods:

- Using ardSerial.py
- Using custom scheduler

Using ardSerial.py:

Use the following syntax to send commands:

- With skill as parameter:

```
..\serialMaster>python3 ardSerial.py kbalance
```

- Without parameter:

```
..\serialMaster>python3 ardSerial.py
```

To exit the script use **quit** or **q**.

Using custom scheduler:

Using custom script, you can control individual motor to perform a specific task as defined in the script and at specific timing.

An example file is provided which can be executed using the following command:

```
..\serialMaster>python3 example.py
```

Additional information about this is provided in a separate section to keep this short. You can read about it here *Controlling Bittle (Advanced)*.

8.8 Extensible Modules

8.8.1 Sensors and Actuators

The head of Bittle is designed to be a clip to hold extensible modules. Mentioned below are some popular modules. You can also wire other add-ons thanks to the rich contents of the Arduino and Raspberry Pi community.

You can find the demo codes of these modules in our GitHub repository. The codes can be found under **ModuleTests** folder inside the source code.

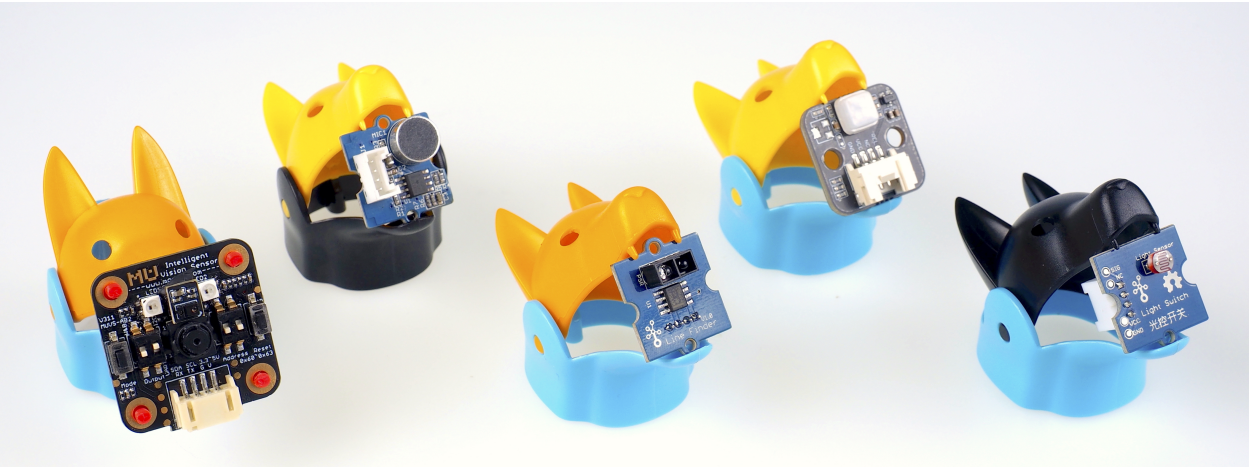


Fig. 8.36: Extensible Modules

The modules are:

- LEDs

- Sound Sensor / Noise Detector
- Light Sensor / Phototransistor
- Touch Sensor
- Infrared Reflective Sensor
- PIR Sensor
- OLED Display
- Ultrasonic Sensor / Distance Sensor

8.8.2 Communication Modules

The Nyboard V1 used by robot uses the Atmel ATMEGA328P controller, which only supports only one serial port.The default serial baud rate is 115200bps. Pin definitions are shown in the table below:

Pin No.	Name	Usage
1	DTR	Modem signal DTR, reset NyBoard after serial download finished.
2	RX	ATMEGA328P RX (receive)
3	TX	ATMEGA328P TX (send)
4	5V	5V power for MCU and chips
5	GND	Ground
6	GND	Ground

USB Adaptor

The module uses a CH340C USB bridge. The uploader has three LEDs: power, Tx, and Rx. Right after the connection, the Tx and Rx should blink for one second indicating initial communication, then dim. Only the power indicator LED should keep lighting up.

NyBoard download interface: used to connect to NyBoard, download program firmware to the robot, and communicate with the computer via serial port.

Communication module debugging interface: used to connect the Bluetooth or WiFi module, update the module program and debug the parameters. In order to avoid the cumbersome operation when connecting with Dupont wires, the pin ordering is slightly different from the NyBoard download interface - the TX/RX interface is reversed, and a GND pin becomes an RTS pin. For details on how to use the debugging interface of the communication module, see the following chapters.

Danger: Do not plug the NyBoard and the other module(WiFi or bluetooth) at the same time!

Warning: If Tx and Rx keep lighting up, there's something wrong with the USB communication. Please check the Adaptor and USB cable.

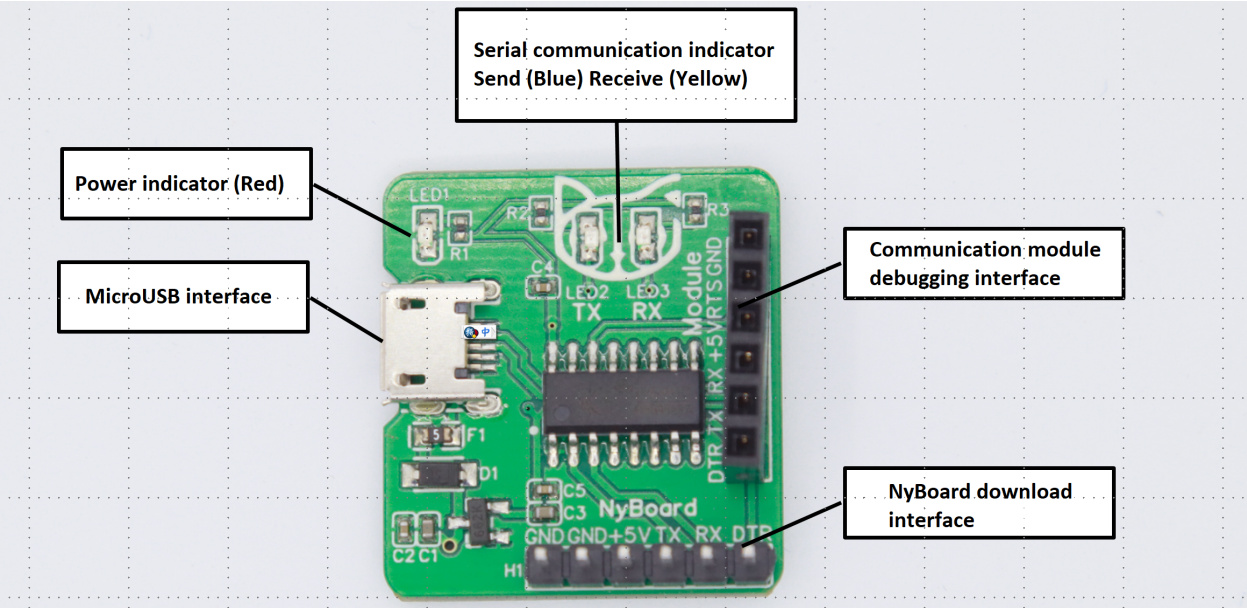


Fig. 8.37: USB Adapter Description

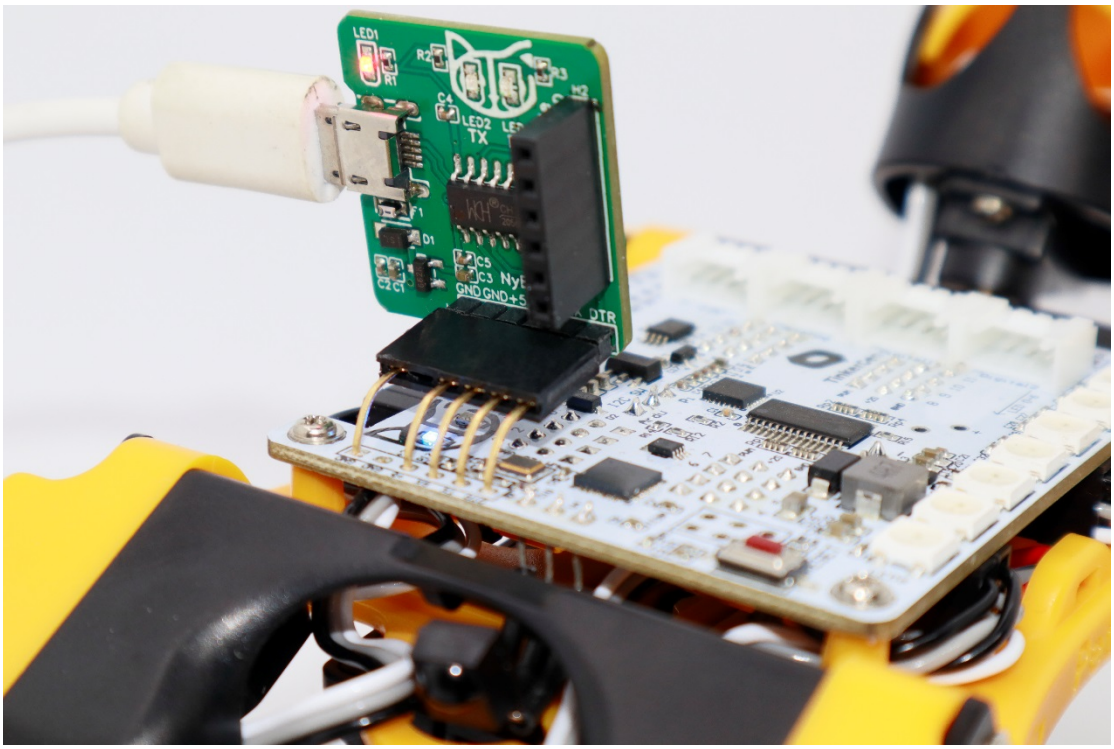


Fig. 8.38: USB Adapter Connection

Bluetooth Adaptor

The module uses JDY-23 Bluetooth 5.0 BLE module. It acts as a bridge between Bluetooth devices and microcontrollers. You can wirelessly upload firmware or control the motion of the robot through a Bluetooth connection. This module is also required to connect with the PETOI mobile app. A blinking LED on the Bluetooth module indicates waiting for a connection. If required, the default PIN for pairing is "0000" or "1234". After the pairing is successful, the system will assign a serial port name.

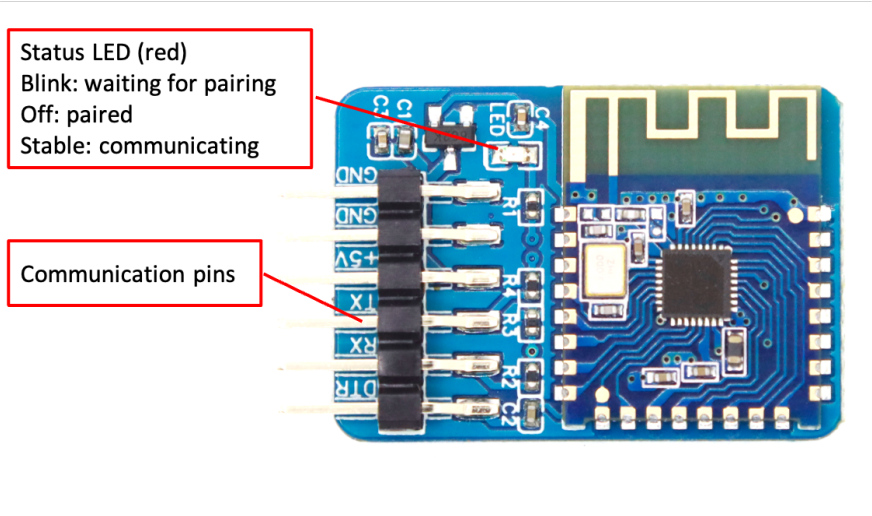


Fig. 8.39: Bluetooth Adapter Description

To externally configure bluetooth adaptor, you can connect the Bluetooth Adaptor with USB Adaptor. To configure the module, use serial monitor with line ending as **NL and CR** and baud rate as **115200**. The commonly used AT commands are given below, for complete list please refer JDY-23's specification sheet.

Usage	**Command**
Check BT module version	AT+VER
Check BT broadcast name	AT+NAME
Change BT broadcast name	AT+NAME<Device Name>
Check serial baud rate	AT+BAUD
Change serial baud rate	AT+BAUD<Baud Rate Identifier>

WiFi Adaptor

The module uses ESP8266EX's official model ESP-WROOM-02D, 4MB QSPI Flash. The module includes an automatic download circuit and a communication module. The automatic download circuit refers to the official recommendation to use 2 S8050 transistors to receive the RTS and DTR signals from the CH340C downloader and trigger the download sequence.

To setup the environment for WiFi adaptor, follow the steps below:

- In Arduino IDE, open **File -> Preferences**.
- Paste http://arduino.esp8266.com/stable/package_esp8266com_index.json in **Additional Boards Manager URL's** section. Click OK and close Preferences Pop-up.
- Open Board Manager from **Tools -> Board -> Board Manager**

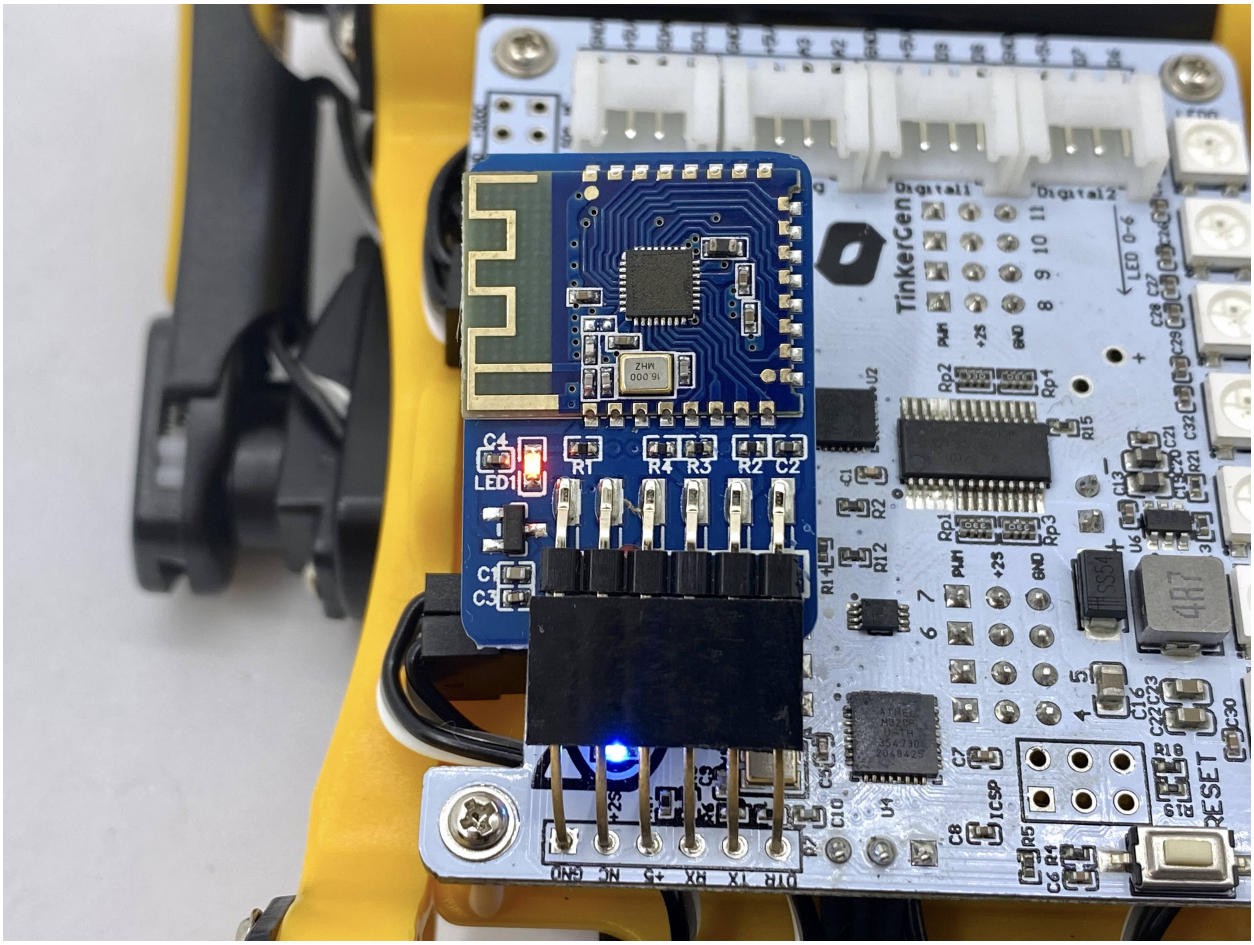


Fig. 8.40: Bluetooth Adapter Connection

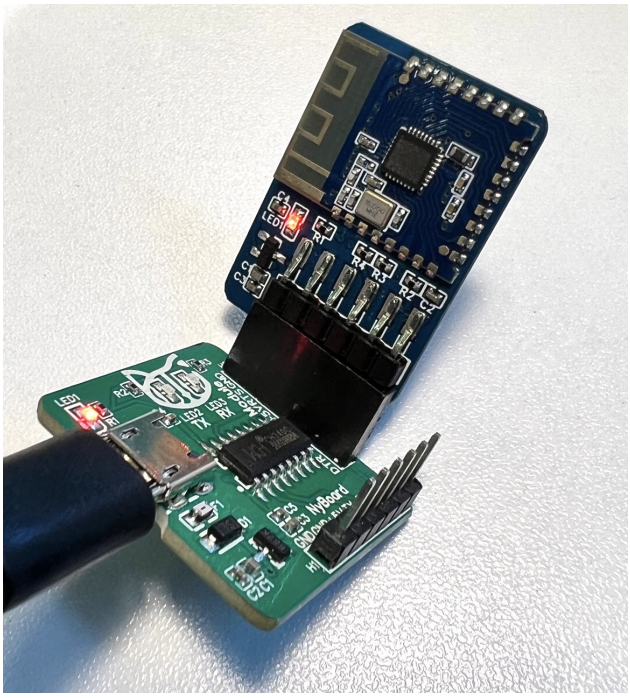


Fig. 8.41: Bluetooth - USB Configure

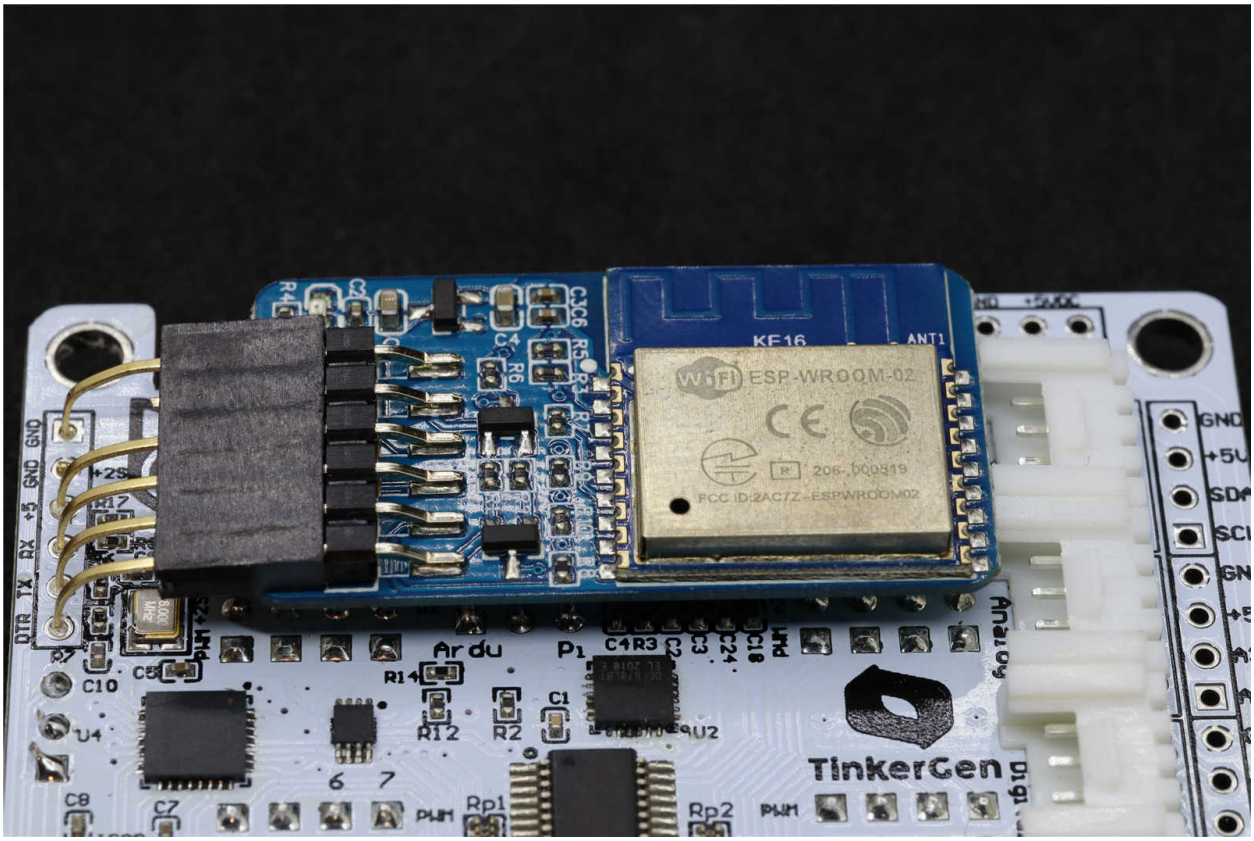


Fig. 8.42: Bluetooth Adapter Connection

- Enter **ESP8266** in the search bar in Board Manager. You will find a search result developed by **ESP8266 Community**.
- Click on **Install**. Wait until it finishes.
- Once done, select **Generic ESP8266 Module** from **Tools -> Board -> ESP8266 Boards**

Select the following settings in **Tools** menu.

Parameters	Settings
Builtin Led	2
Upload Speed	921600
CPU Frequency	80 MHz
Crystal Frequency	26 Mhz
Flash Frequency	40 Mhz
Flash Size	4MB (FS:2MB / OTA:1019KB
Reset Method	DTR reset
lwIP variant	V2 Lower memory
Erase Flash	Only sketch

Fig. 8.43: ESP8266 - USB Configure

To use the module, a sample code is available under OpenCat/ModuleTests/ESP8266WiFiController. The code folder consists of 3 files:

- ESP8266WiFiController.ino: Arduino sketch with server core code.
- mainpage.h: welcome page (html) in a char array.
- actionpage.h: action controller page (html) in a char array.

Steps to use sample code:

- Open code with IDE. Set the parameters for ESP9266 as given above. Connect the ESP8266 to USB Module.
- Upload the code. Open Serial Monitor and set line ending to **NL and CR** and baud rate as **115200**.

- The module will create a Access Point. Using a WiFi enabled device (Mobile, Tablet, PC etc.), connect to the access point named **Bittle-AP**.
- Once WiFi is connected, open a web browser and enter **192.168.4.1** in address bar and go to it.
- Here, configure your WiFi so that this ESP8266 can connect to it and thus to Internet.
- After the WiFi connection between your router and ESP8266 is successful, the AP mode will switch to client mode and now your ESP8266 can connect to Internet.
- Enter the IP address shown on Serial Monitor in address bar of your WiFi device. A webpage with various options will open. From here you can control your robot over WiFi.

Additional information about this is provided in a separate section to keep this short. You can read about it here *Controlling Bittle (Advanced)*.

8.8.3 Imaging Module

In addition to above mentioned modules, you can use a **Camera** module with Bittle to see what your robot dog is seeing.

For this, an ESP32 Camera module is suitable.

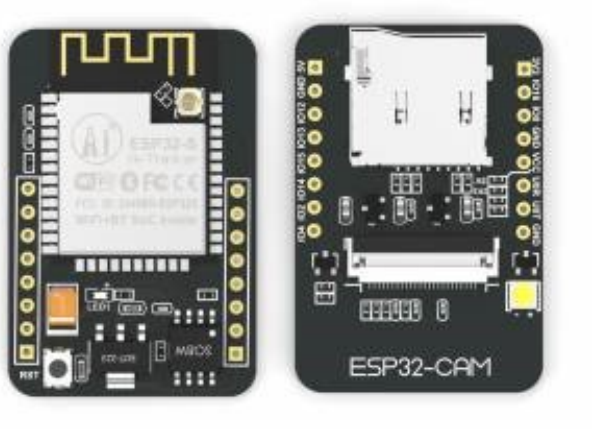


Fig. 8.44: ESP32 Cam Board layout (ref: https://docs.ai-thinker.com/en/esp32-cam)

To setup the environment for ESP32 Camera, follow the steps below:

- In Arduino IDE, open **File -> Preferences**.
- Paste https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json in **Additional Boards Manager URL's** section. Click OK and close Preferences Pop-up.
- Open Board Manager from **Tools -> Board -> Board Manager**
- Enter **ESP32** in the search bar in Board Manager. You will find a search result developed by **esp32**.
- Click on **Install**. Wait until it finishes.
- Once done, select **AI Thinker ESP32-CAM** from **Tools -> Board -> ESP32 Arduino**

Select the following settings in **Tools** menu.

Parameters	Settings
Partition Scheme	Huge APP

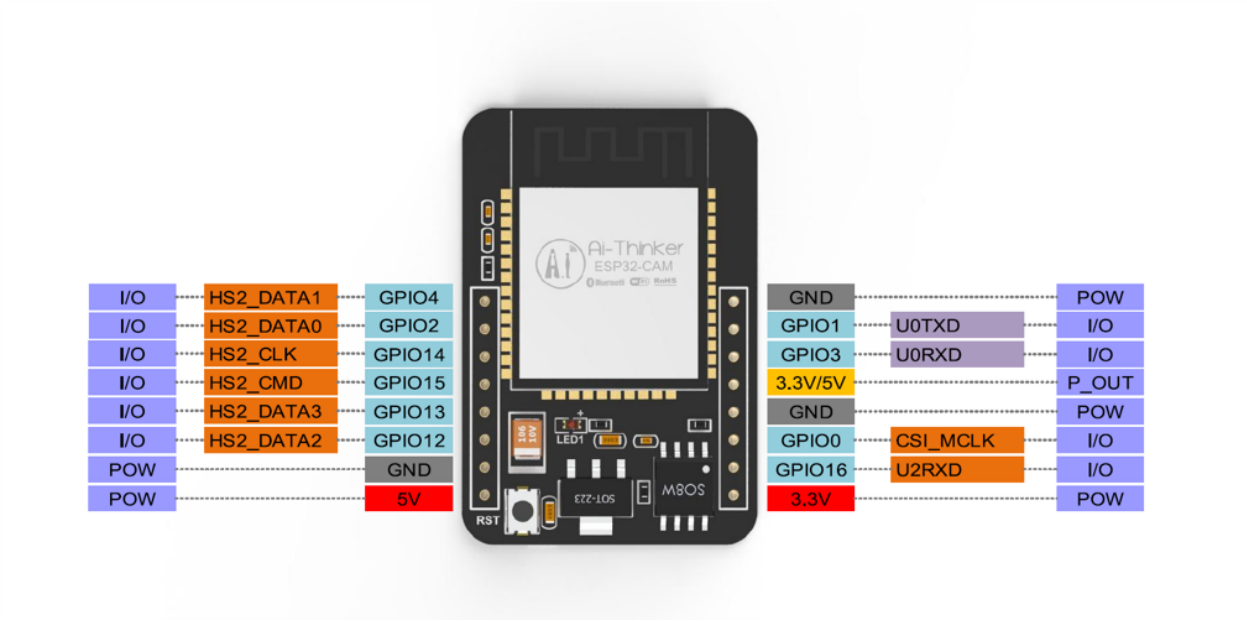


Fig. 8.45: ESP32 Cam Pinouts (ref: https://docs.ai-thinker.com/en/esp32-cam)

Pin Connection with USB Adapter

ESP 32 Cam	USB Adaptor
5V	5V
GND	GND
UnR	TX
UnT	RX
IO0	GND (when uploading)

Steps to use sample code:

- Open code with IDE. Open Example file named **CameraWebServer** from **File** -> **Examples** -> **ESP32** -> **Camera** -> **CameraWebServer**
- Set the credentials for your WiFi, (SSID and Password)
- Please confirm that **CAMERA_MODEL_AI_THINKER** is selected in the code as this is our board.
- Connect the ESP32_Cam module with the FTDI programmer if not already done. Connect the complete setup to PC using USB cable.
- Select correct COM port and Upload the code. While uploading, keep in mind that **GPIO 0 has to connect to GND** and module has to reset to go in download mode before you upload code to it.
- After successful upload, disconnect **GPIO 0 from GND**.
- Open Serial Monitor. Set the Baud Rate to 115200.
- Note the IP-Address and open it in a browser.
- Use onscreen options to control the camera and adjust its parameters.

Note: To use Flash / On board LED, please refer the in lecture presentation for code snippets.

Note: If the upload fails, check in serial monitor if your esp32cam is in download mode (baud rate: 115200).

If it is in download mode, upload again and when you see dots (. . .) in output window, press **reset** button on esp32cam.

If you face problems for more than 30 minutes, contact your instructor.

8.9 Serial Commands Cheat Sheet

OpenCat
Communication Protocol and Parsing

Interface		Token	Encoding	Parameters	Format	Bytes	Function
RasPi Serial Port	Arduino Serial Monitor	'h'	Ascii		char	1	print help information
		'c'		idx*,angle** '\n'	string	strlen + 2	calibrate $servo_{idx}$ by angle
		'm'		idx*,angle** '\n'	string	strlen + 2	move $servo_{idx}$ to angle
		'j'			char	1	show all 16 joint angles
		'd'			char	1	shut down servos
		'p'			char	1	pause motion
		'a'			char	1	abandon calibration
		's'			char	1	save calibration
		'k'		abbreviation '\n'	string	strlen + 2	load skill
		'w'		command '\n'	string	strlen + 2	some future command words
		'r'			char	1	reset board
		'i'	Binary	idx_1 a_1 ... idx_N a_N '~'	string	strlen + 2	list of indexed rotation angles
		'I'		a_1 a_2 ... a_{DoF} '~'	string	DoF + 2	list of all DoF rotation angles

* index range: 0 ~ (DoF - 1)

** angle range: -90 ~ 90. fits in the range of signed char (-128 ~ 127). Also depends on the servos' parameters

Fig. 8.46: Serial Commands List 1

8.10 Tasks

Before doing the tasks, please make sure that you can control the robot using IR Remote. You should also know how to use Serial commands using Arduino Serial Monitor.

You might need to 3D print some parts to mount your sensors.

Note: These tasks will use Arduino UNO are Higher level controller and the Bittle board as low level controller. You must upload all your codes only to Arduino UNO. All sensors and actuators must be connected to Arduino UNO and not directly to Bittle board.

	Type	Command	Note	备注
Control	Token	d	rest and shutdown all servos	休息并关闭所有舵机
		g	turn on/off gyro to boost speed	开关陀螺仪加速运动
		p	pause motion and shut off all servos	暂停动作并关闭所有舵机
		c	enter calibration mode	进入校准模式
		m	move a joint to certain angle	把某关节转动到某角度
Skill	Gait	bk	back	后退
		bkL	backLeft	后退 左
		bkR	backRight	后退 右
		vt	stepping on the same spot	原地踏步
		crF	crawl	爬 低重心对角步态
		crL	crawl left	爬 左
		crR	craw right	爬 右
		wkF	walk	走 三脚着地的步态
		wkL	walk left	走 左
		wkR	walk right	走 右
		trF	trot	小跑 对角步态
		trL	trot left	小跑 左
		trR	trot right	小跑 右
		bdF	bound (not recommanded)	兔子跳（不推荐）
	Posture	balance		正常站立演示自平衡性
		buttUp	buttom up	撅屁股
		calib	calibration	校准姿态
		rest		休息
		sit		坐
		sleep		睡
		str	stretch	伸懒腰
		zero		零姿势 给用户自己设计动作的模板
	Behavior	ck	check around	左右看
		hi	hi sequence	打招呼
		pee	pee sequence	撒尿
		pu	push up sequence	俯卧撑
		rc	recovering sequence	四脚朝天后恢复站立(自动激活)
		pd	play dead	主动翻身倒地装死
		bf	back flip	后空翻（隐藏）

You need to add a 'k' before skills. For example, kbk, kbalance, kck...

The Gaits can be the result of combined command: gait+direction

		Gait	Direction	Serial Command
		cr	F	kcrF
		wk	L	kwkL
		tr	R	ktrR

Fig. 8.47: Serial Commands List 2

Note: Use Serial baud rate as 15200 for communicating with Bittle Board.

Hint: You can use SoftwareSerial for the following tasks. Look about this before using.

Warning: Use **develop** branch for all initial codes. Create other branches according to the requirements.

When everything is ready before submission, merge the code to **main** branch.

WHEN MERGING **develop** TO **main** DE-SELECT "Delete source branch when merge request is accepted." OPTION.

8.10.1 Connections

Connect your Bittle with Arduino UNO / ESP32 CAM using TX and RX pins. The connection should be as below:

Arduino UNO	Bittle Board
TX	Rx
GND	GND
VIN	5V

Warning: When uploading code, you might have to remove the TX and RX pins from Arduino UNO board as they are connected to the USB programmer.

Hint: You can also use Software Serial to virtually create TX and RX on any available digital pin on UNO board.

8.10.2 Task 1

Make Bittle move straight for some distance, perform a skill, turn around and return back to the start point. The command must start from Node-RED.

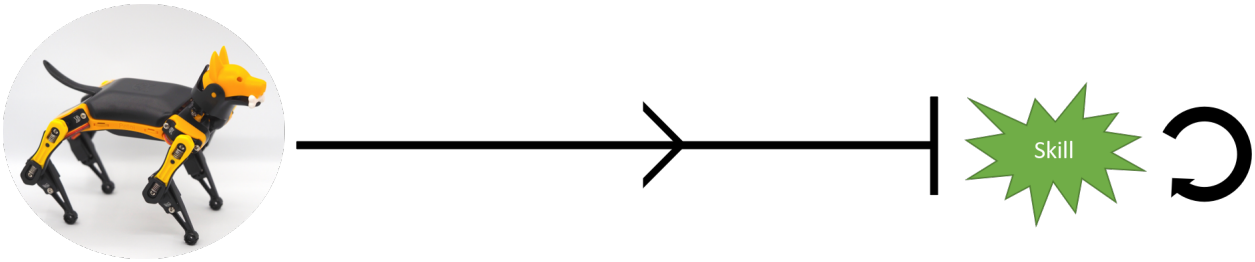


Fig. 8.48: Task: Follow straight line

8.10.3 Task 2

Bittle follows a line. The line will have curves, and will not be always straight.

Using ESP32CAM , you must make bittle follow a line. The line will be White line on a Black (dark) floor.

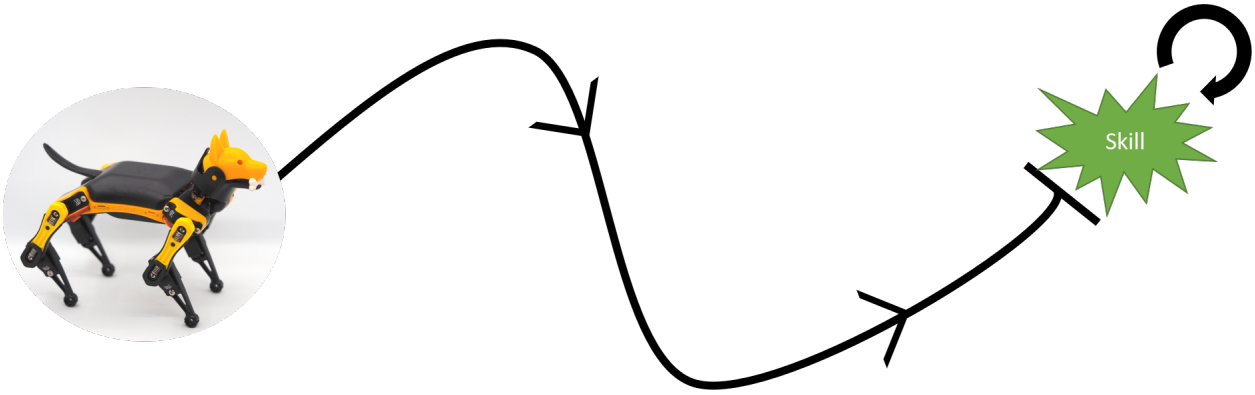


Fig. 8.49: Task: Follow path

8.10.4 Final Task

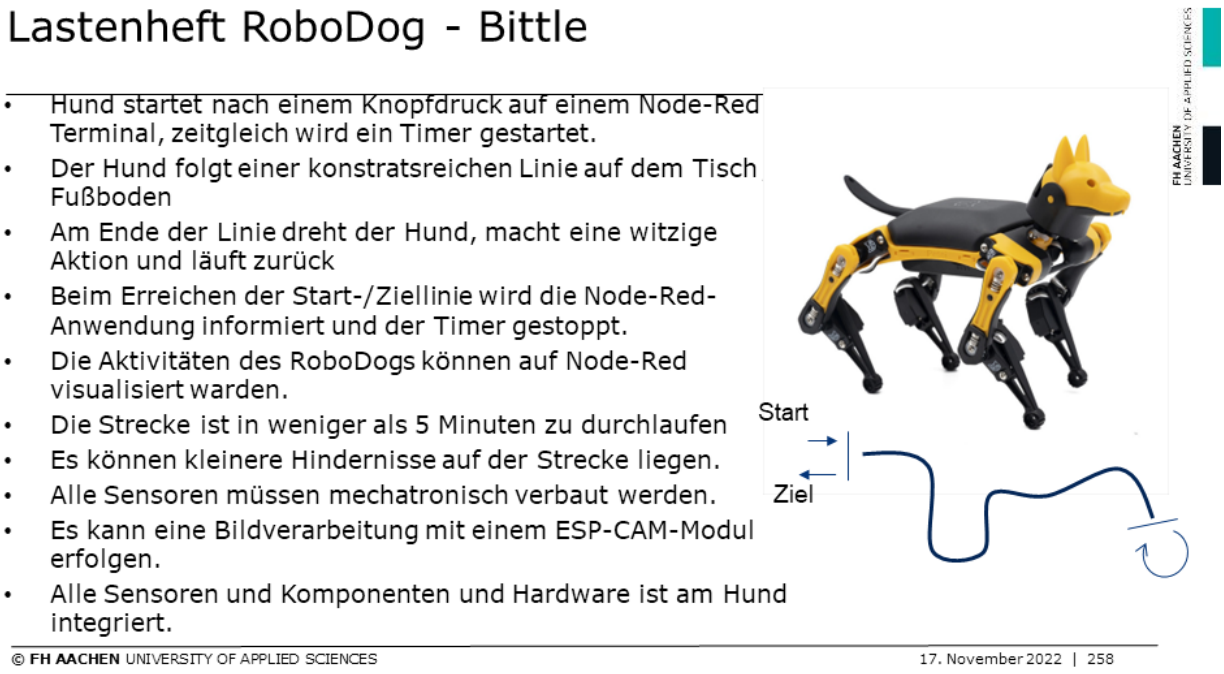


Fig. 8.50: Task: Final Task Lastenheft

Additionally:

1. Doxygen based code document
2. Working Video

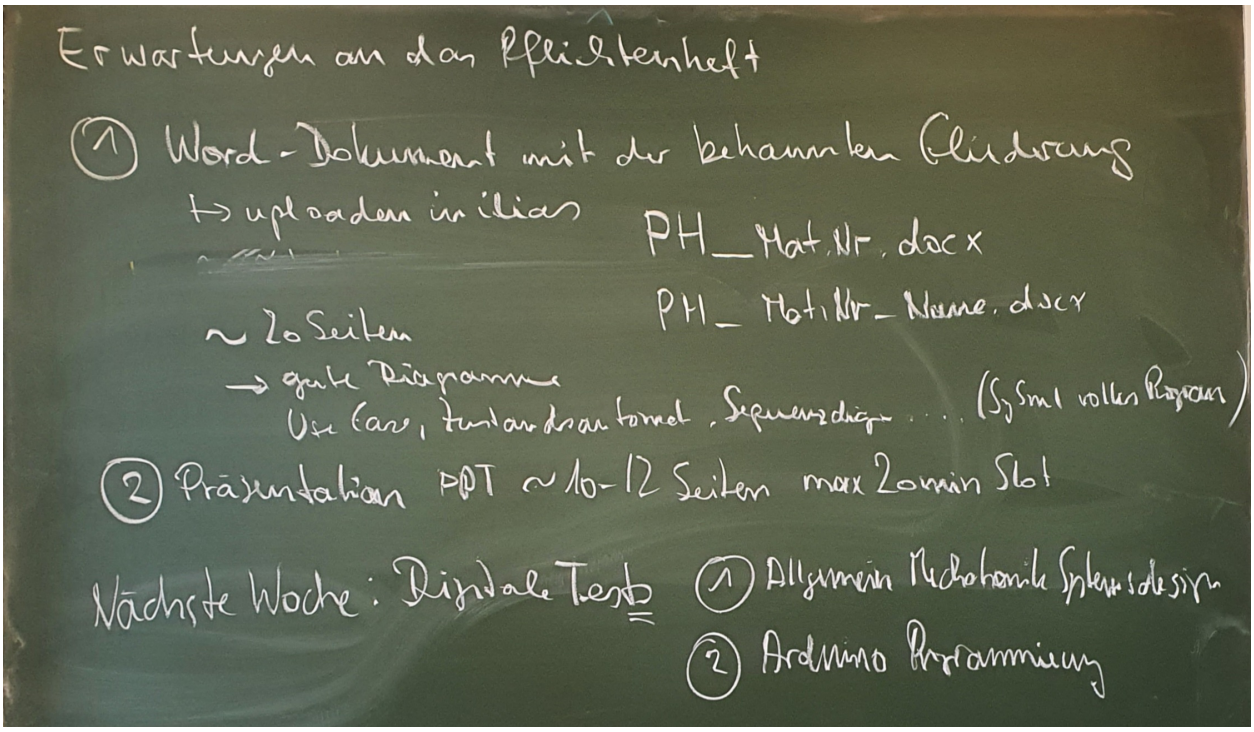


Fig. 8.51: Task: Final Task Instructions

3. Gitlab repository of codes shared with instructor

8.11 Teach Bittle New Skills

8.11.1 Understand skills in InstinctBittle.h.

EEPROM has limited (1,000,000) write cycles. So I want to minimize the write operations on it.

There are two kinds of skills: **Instincts** and **Newbility**. The addresses of both are written to the onboard EEPROM(1KB) as a lookup table, but the actual data is stored at different memory locations:

- I2C EEPROM (8KB) stores **Instincts**.

The Instincts are already fine-tuned/fixed skills. You can compare them to "muscle memory". Multiple Instincts are linearly written to the I2C EEPROM only once with WriteInstinct.ino. Their addresses are generated and saved to the lookup table in onboard EEPROM during the runtime of WriteInstinct.ino.

- PROGMEM (sharing the 32KB flash with the sketch) stores **Newbility**.

A Newbility is any new experimental skill that requires a lot of tests. It's not written to the I2C nor onboard EEPROM, but the flash memory in the format of PROGMEM. It has to be uploaded as one part of the Arduino sketch. Its address is also assigned during the runtime of the code, though the value rarely changes if the total number of skills (including all Instincts and Newbilities) is unchanged.

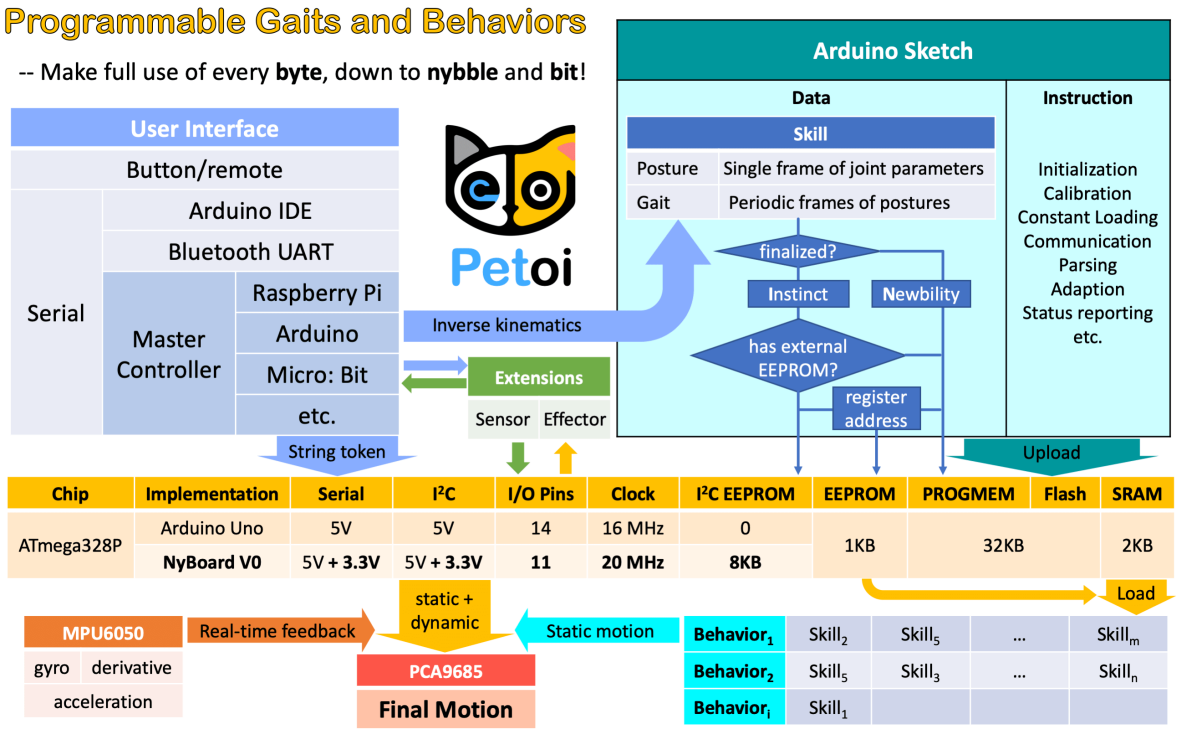


Fig. 8.52: Bittle Instinct Summary

8.11.2 Example InstinctBittle.h

```
//a short version of InstinctBittle.h as example

#define BITTLE
#define NUM_SKILLS 4
#define I2C_EEPROM

const char rest[] PROGMEM = {
1, 0, 0, 1,
-30, -80, -45,   0,  -3,  -3,   3,   3,  75,  75,  75,  75, -55, -55, -55, -55,};
const char zero[] PROGMEM = {
1, 0, 0, 1,
    0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,};

const char crF[] PROGMEM = {
36, 0, -3, 1,
61,  68,  54,  61, -26, -39, -13, -26,
66,  61,  58,  55, -26, -39, -13, -26,
...
51,  81,  45,  72, -25, -37, -12, -25,
55,  76,  49,  68, -26, -38, -13, -26,
60,  70,  53,  62, -26, -39, -13, -26,
};
```

(continues on next page)

(continued from previous page)

```
const char pu[] PROGMEM = {
-8, 0, -15, 1,
6, 7, 3,
0,   0,   0,   0,   0,   0,   0,   0,  30,  30,  30,  30,  30,  30,  30,  30,        5,␣
↪0,
15,  0,   0,   0,   0,   0,   0,   0,  30,  35,  40,  29,  50,  15,  15,  15,        5,␣
↪0,
30,  0,   0,   0,   0,   0,   0,   0,  27,  35,  40,  60,  50,  15,  20,  45,        5,␣
↪0,
15,  0,   0,   0,   0,   0,   0,   0,  45,  35,  40,  60,  25,  20,  20,  60,        5,␣
↪0,
0,   0,   0,   0,   0,   0,   0,   0,  50,  35,  75,  60,  20,  30,  20,  60,        6,␣
↪0,
-15, 0,   0,   0,   0,   0,   0,   0,  60,  60,  70,  70,  15,  15,  60,  60,        6,␣
↪0,
0,   0,   0,   0,   0,   0,   0,   0,  30,  30,  95,  95,  60,  60,  60,  60,        6,␣
↪1,
30,  0,   0,   0,   0,   0,   0,   0,  75,  70,  80,  80, -50, -50,  60,  60,        8,␣
↪0,
};

#if !defined(MAIN_SKETCH) || !defined(I2C_EEPROM)
const char* skillNameWithType[] =
{"crI", "puI", "restI", "zeroN",};
const char* progmemPointer[] =
{cr, pu, rest, zero, };
#else
const char* progmemPointer[] = {zero};
#endif
```

Defined constants

define WalkingDOF 8

defines the number of DoF (Degree of Freedom) for walking is 8 on Bittle.

define NUM_SKILLS 4

defines the total number of skills is 4. It should be the same as the number of items in the list const char* skillNameWithType[].

define I2C_EEPROM

Means there's an I2C EEPROM on NyBoard to save Instincts.

> **Warning:** If you are building your own circuit board that doesn't have it, comment out this line. Then both kinds of skills will be saved to the flash as PROGMEM. Obviously, it will reduce the available flash space for functional codes. If there were too many skills, it may even exceed the size limit for uploading the sketch.

Data structure of skill array

One frame of joint angles defines a static posture, while a series of frames defines sequential postures, such as a gait or a behavior. Observe the following two examples:

```
const char rest[] PROGMEM = {
1, 0, 0, 1,
-30, -80, -45,   0,  -3,  -3,   3,   3,  75,  75,  75,  75, -55, -55, -55, -55,};

const char crF[] PROGMEM = {
36, 0, -3, 1,
61,  68,  54,  61, -26, -39, -13, -26,
66,  61,  58,  55, -26, -39, -13, -26,
...
51,  81,  45,  72, -25, -37, -12, -25,
55,  76,  49,  68, -26, -38, -13, -26,
60,  70,  53,  62, -26, -39, -13, -26,
};
```

They are formatted as:

	Total # of Frames	Expected Body Orientation		Angle Ratio	Indexed Joint Angles															
		Roll	Pitch		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
rest	1	0	0	1	-30	-80	-45	0	-3	-3	3	3	60	60	-60	-60	-45	-45	45	45
crF	36	0	-3	1									61	68	54	61	-26	-39	-13	-26
													...	...	...	...	...	...	...	...
													60	70	53	62	-26	-39	-13	-26

Fig. 8.53: Skill Definition

rest is a static posture, it has only one frame of 16 joint angles. crF is the abbreviation for "crawl forward". It has 36 frames of 8 (or 12, depending on the number of walking DOF) joint angles that form a repetitive gait. Expected body orientation defines the body angle when the robot is conducting the skill. If the body is tilted from the expected angles, the balancing algorithm will calculate some adjustments. Angle ratio is used when you want to store angles larger than the range of -128 to 127. Change the ratio to 2 so that you can save those large angles by dividing 2.

Warning: A posture has only one frame, and a gait has more than one frames and will be looped over.

The following example is a behavior:

```
const char pu[] PROGMEM = {
-8, 0, -15, 1,
6, 7, 3,
0,    0,   0,   0,   0,   0,   0,   0,  30,  30,  30,  30,  30,  30,  30,  30,       5,␣
↪0, 0, 0,
15,   0,   0,   0,   0,   0,   0,   0,  30,  35,  40,  29,  50,  15,  15,  15,       5,␣
↪0, 0, 0,
30,   0,   0,   0,   0,   0,   0,   0,  27,  35,  40,  60,  50,  15,  20,  45,       5,␣
↪0, 0, 0,
```

(continues on next page)

(continued from previous page)

```
15,   0,   0,   0,   0,   0,   0,   0,  45,  35,  40,  60,  25,  20,  20,  60,       5,␣
↪0, 0, 0,
0,    0,   0,   0,   0,   0,   0,   0,  50,  35,  75,  60,  20,  30,  20,  60,       6,␣
↪0, 0, 0,
-15,  0,   0,   0,   0,   0,   0,   0,  60,  60,  70,  70,  15,  15,  60,  60,       6,␣
↪0, 0, 0,
0,    0,   0,   0,   0,   0,   0,   0,  30,  30,  95,  95,  60,  60,  60,  60,       6,␣
↪1, 0, 0,
30,   0,   0,   0,   0,   0,   0,   0,  75,  70,  80,  80, -50, -50,  60,  60,       8,␣
↪0, 0, 0,
};
```

pu is short for "push up", and is a "behavior". Its data structure contains more information than posture and gait.

The first four elements are defined the same as before, except that the number of frames is saved as a negative value to indicate that it's a behavior. The next three elements define the repeating frames in the sequence: starting frame, ending frame, looping cycles. So the **6, 7, 3** in the example means the behavior should loop from the 7th to the 8th frame for 3 times (the index starts from 0). The whole behavior array will be executed only once, rather than looping over like the gait.

Each frame contains 16 joint angles, and the last 4 elements define the speed of the transition, and the delay condition after each transition:

1. The default speed factor is 4, it can be changed to an integer from 1 (slow) to 127 (fast). The unit is **degree per step**. If it's set to 0, the servo will rotate to the target angle by its maximal speed (about 0.07sec/60 degrees). It's not recommended to use a value larger than 10 unless you understand the risks.
2. The default delay is 0. It can be set from 0 to 127, the unit is **50 ms**.
3. The 3rd number is the trigger axis. If it's not 0, the previous delay time will be ignored. The trigger of the next frame will depend on the body angle on the corresponding axis. 1 for the pitch axis, and 2 for the roll axis. The sign of the number defines the direction of the threshold, i.e. if the current angle is smaller or larger than the trigger angle.
4. The 4th number is the trigger angle. It can be -128 to 127 degrees.

Suffix for indicating Instinct and Newbility

You must upload WriteInstinct.ino to have the skills written to EEPROM for the first time. The following information will be used:

```
const char* skillNameWithType[] =
{"crI", "puI", "restI", "zeroN",};
const char* progmemPointer[] =
{cr, pu, rest, zero, };
```

Warning: Notice the suffix **I** or **N** in the skill name strings. They tell the program where to store skill data and when to assign their addresses.

Later, if the uploaded sketch is the main sketch **OpenCat.ino**, and if you are using NyBoard that has an I2C EEPROM, the program will only need the pointer to the Newbility list

const char* progmemPointer[] = {zero};

to extract the full knowledge of pre-defined skills.

8.11.3 Define new skills and behaviors

Modify the existing skill template

There's already a skill called "zeroN" in InstinctBittle.h. It's a posture at the zero state waiting for your new definition. You can first use the command mIndex Offset to move an individual joint to your target position, then replace the joint angles (bold fonts) in array at once:

```
const char zero[] PROGMEM = {
1, 0, 0, 1,
0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,};
```

Because it's declared as a Newbility and doesn't require writing to I2C EEPROM, you can simply upload **OpenCat.ino** every time you change the array (without uploading **WriteInstinct.ino**). You can trigger the new posture by pressing **7>** on the IR remote, or type kzero in the serial monitor.

You can rename this skill, but remember to update the keymap of the IR remote.

Add more skills in InstinctBittle.h

You can add more skills in InstinctBittle.h. Remember to increase the skill number at the beginning of the file and add the corresponding skill name and pointer in the skill list array.

A skill can be called from the serial monitor with the token 'k' command. For example, ksit will move Bittle to posture "sit".

You can also tune new skills by sending posture frames through the serial port, using the **m, i, l** tokens without uploading a new sketch. After fine-tuning the skill, you can save it in the instinctBittle.h and upload it to the board as a newbility or instinct.

> **Warning:** Always check the actual code for the available skill names. We may alter the skill set as we iterate the software.

This git repo (https://github.com/ger01d/kinematic-model-opencat) is a good starting point if you want to develop a customized gait. If you are going to do some inverse kinematics calculation, you may use the following key dimensions to build your model.

Automation

So far Bittle is controlled by the infrared remote. You make decisions for Bittle's behavior.

You can connect Bittle with your computer or smartphone, and let them send out instructions automatically. Bittle will try its best to follow those instructions.

By adding some sensors (like a touch sensor), or some communication modules (like a voice control module), you can bring new perception and decision abilities to Bittle. You can accumulate those automatic behaviors and eventually make Bittle live by itself!

Another direction is to set up a simulation for Bittle then test the model in reality. You may use this Unified Robot Description Format (URDF) (https://github.com/AIWintermuteAI/Bittle_URDF) file for Bittle to set up an NVIDIA Omniverse simulation.

8.11.4 Understand OpenCat.h

The controlling framework behind Bittle is OpenCat, which I've been developing for a while. You can learn more from posts available here (https://www.hackster.io/petoi/opencat-845129)

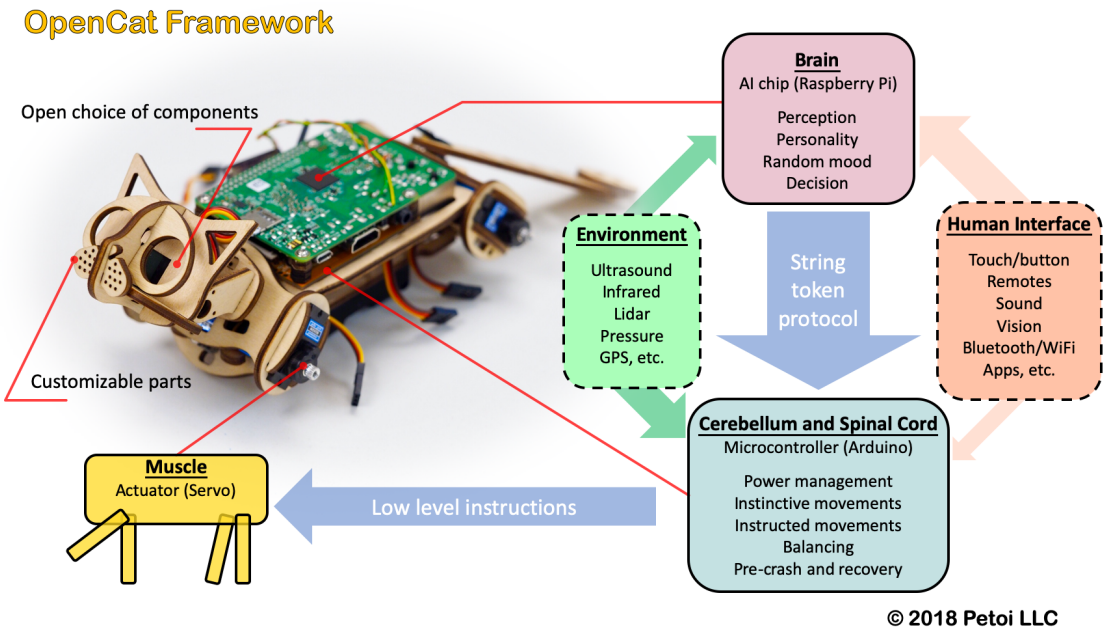


Fig. 8.54: Opencat Firmware Summary

Further reading abut code review: (https://www.petoi.camp/forum/software/flavien-opencat-code-reviews)

8.11.5 Tutorial on Creating New Skills

Preparation

Get familiar with the standard process of assembly, uploading the standard program **Opencat.ino**, and calibration.

Validate that the following functions work as expected.

- Press the button which is in the 2nd row, the 2nd column on the IR remote. Later we will use (2, 2) as the index. You can also enter **kbalance** via serial port. Bittle should stand up and keep balance;
- Press the button which is in the 7th row, the 3rd column on the IR remote. Later we will use (7, 3) as the index. You can also enter **kzero** via serial port. Bittle should enter a posture similar to the calibration state, which is the "zero" skill in the program (as shown in the figure below).

Open the folder **src/**, create a backup file of instinctBittle.h as instinctBittle.hBAK.

The coordinate system of Bittle is shown in the figure below.

Yaw: Rotate around the Z-axis.

Pitch: Rotate around the Y-axis (nose up/down).

Roll: Rotate around the X-axis (tilt left/right).

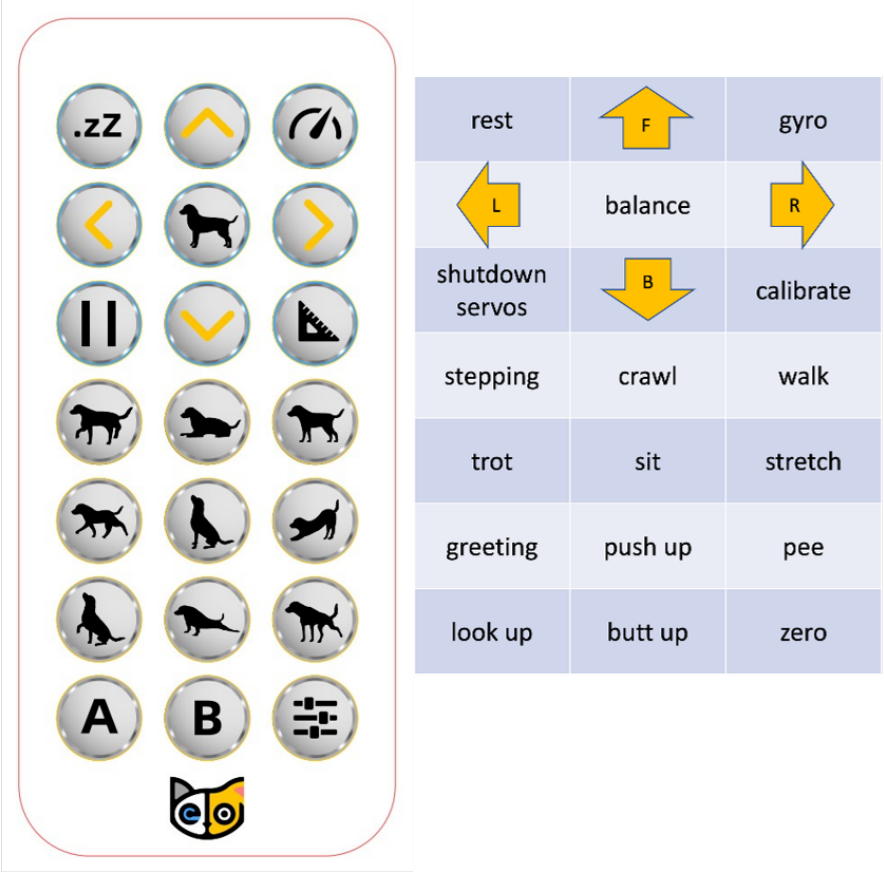


Fig. 8.55: IR Controls

- **Rest** puts the robot down and shuts down the servos.
- **Balance** is the neutral standing posture.
- Pressing **F/L/R** will make the robot move forward/left/right
- **B** will make the robot move backward
- **Calibrate** puts the robot into calibration posture and turns off the gyro
- **Stepping** lets the robot step at the original spot
- **Crawl/walk/trot** are the gaits that can be switched and combined with the direction buttons
- Buttons after trot are preset postures or other skills
- Gyro will turn on/off the gyro for self-balancing. Turning off the gyro can accelerate and stabilize the slower gaits. But it's NOT recommended for faster gaits such as trot. Self-righting will be disabled because the robot no longer knows it's flipped.
- Different surfaces have different friction and will affect walking performance. The carpet will be too bushy for the robot's short legs. It can only crawl (command kcr) over this kind of tough terrain.
- You can pull the battery pack down and slide along the longer direction of the belly. That will tune the center of mass, which is very important for walking performance.
- When the robot is walking, you can let it climb up/down a small slope (<10 degrees)

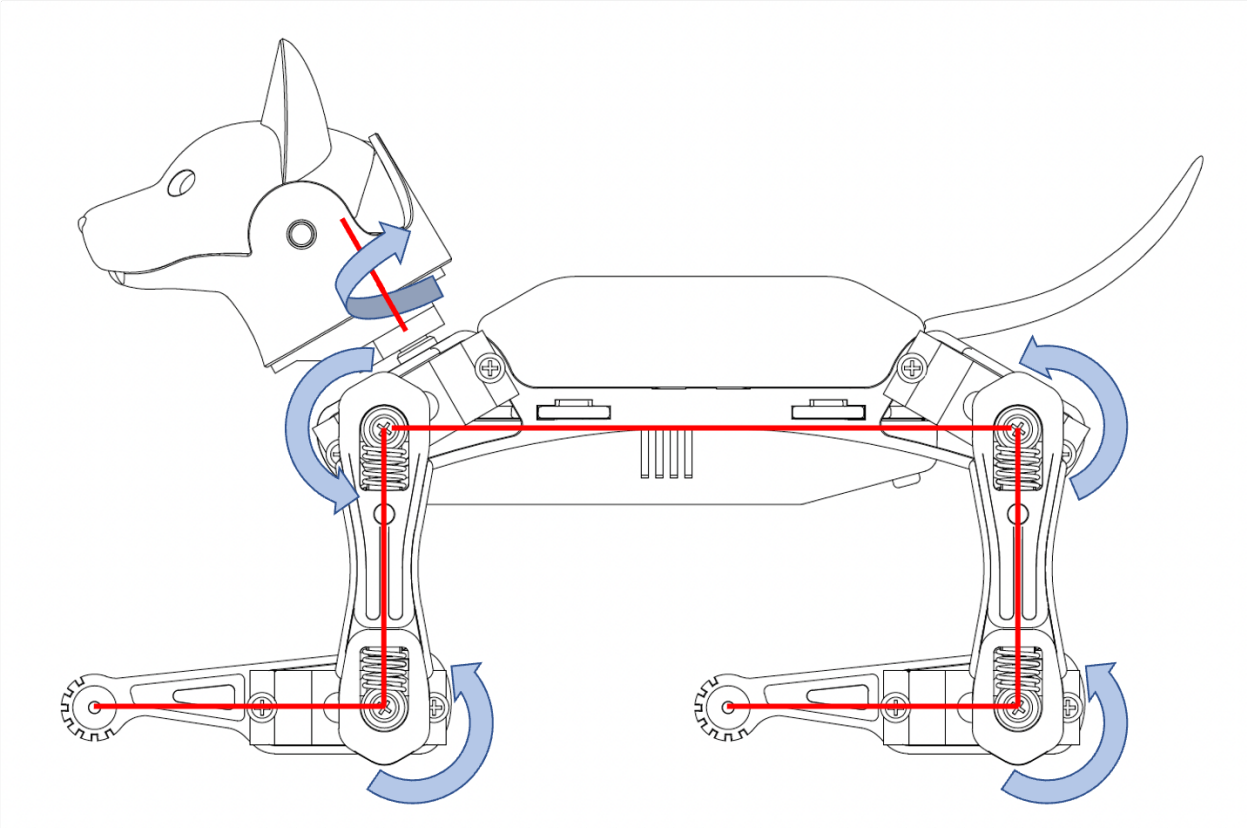


Fig. 8.56: Zero state of Bittle

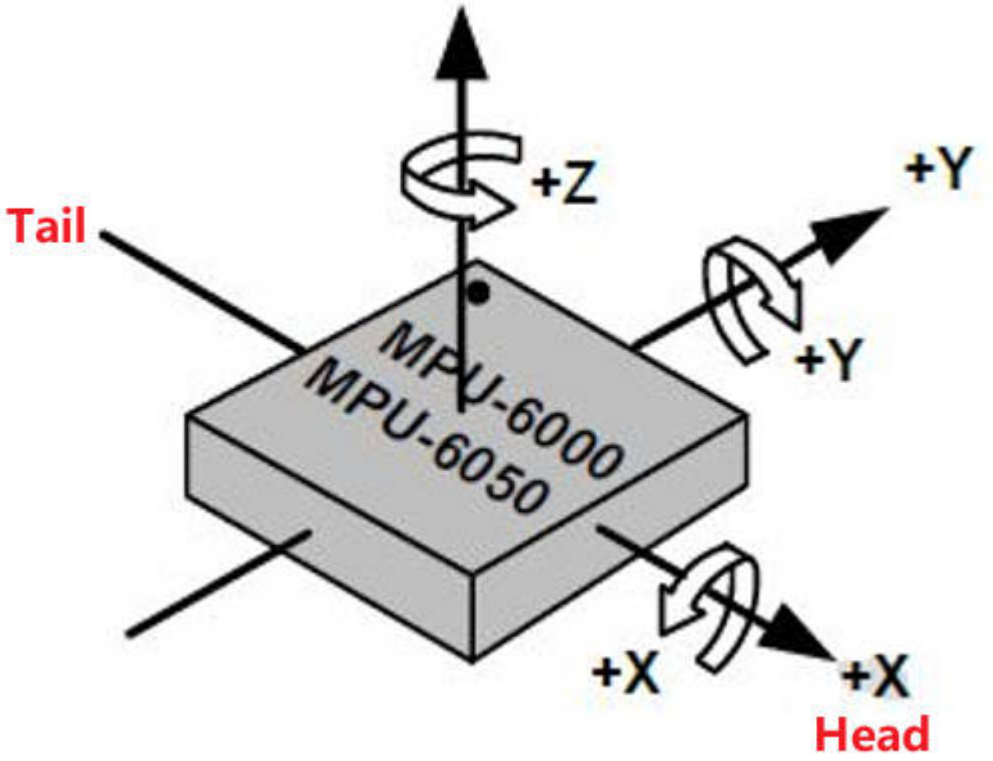


Fig. 8.57: IMU Direction

For the legs, on the left side, counterclockwise is positive, clockwise is negative. On the right side, the rotation directions are mirrored. The origin position and rotation direction of a single leg (upper/lower leg) around the joint are shown in the 2nd figure.

The indexing order of all the joints is shown in the figure below:

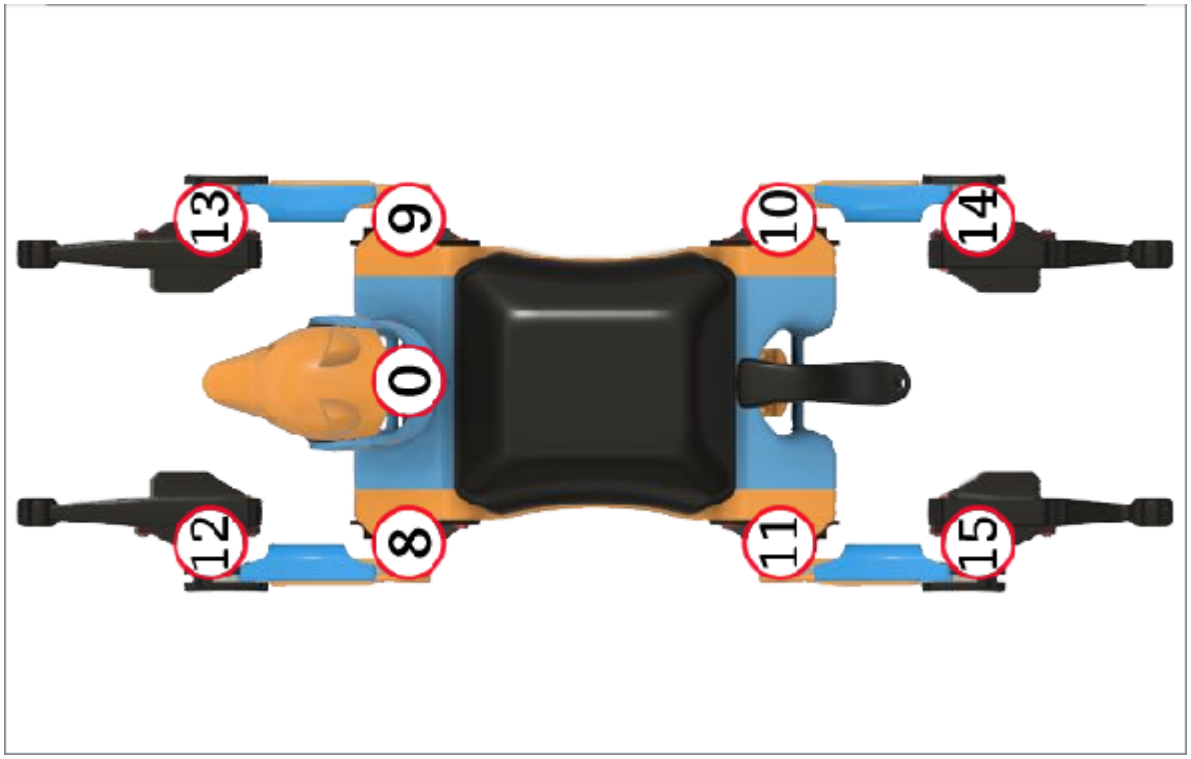


Fig. 8.58: Bittle top view with motor index

The skill arrays are defined in WriteInstinct/instinctBittle.h, formatted as the figure below. **Note** the index starts from 0.

	Total # of Frames	Expected Body Orientation		Angle Ratio	Indexed Joint Angles															
		Roll	Pitch		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
rest	1	0	0	1	-30	-80	-45	0	-3	-3	3	3	60	60	-60	-60	-45	-45	45	45
crF	36	0	-3	1									61	68	54	61	-26	-39	-13	-26
													...	...	...	...	...	...	...	...
													60	70	53	62	-26	-39	-13	-26

Fig. 8.59: Skill Definition

Total # of Frames defines the number of frames of a certain skill.

For example, **rest** is a static posture, it has only one frame of 16 joint angles.

crF is the abbreviation for "crawl forward". It has 36 frames of 8 (or 12, depending on the number of walking DOF) joint angles that form a repetitive gait.

Expected body orientation defines the body angle when the robot is conducting the skill. When all joints move to their

corresponding joint angle values in the current frame, the body inclination should naturally reach the defined expected angle. If the body inclination deviates from the expected angle, the balance algorithm will calculate some adjustment values for each leg servo , in order to keep the body tilt as close to the expected angle as possible.

Angle ratio is used when you want to store angles larger than the range of -128 to 127. Change the ratio to 2 so that you can save those large angles by dividing 2.

Understand the format of a posture

Zero is a static posture, Find the **zero** array in instinctBittle.h:

```
const char zero[] PROGMEM = {
1, 0, 0, 1,
0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,};
```

Change some of these values to:

```
const char zero[] PROGMEM = {
1, 0, 0, 1,
70, 0, 0, 0, 0, 0, 0, 0, -60, 0, 0, 0, 60, 0, 0, 0,};
```

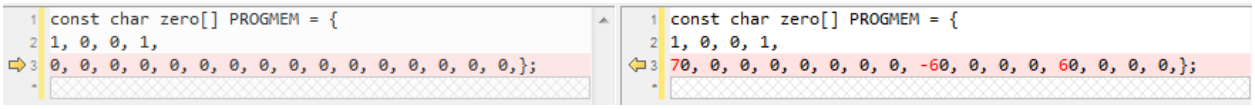


Fig. 8.60: Skill development Code Compare

Save the change, and upload the program **OpenCat.ino** to Bittle (**Note**: there is no need to upload writeinsict.ino or opencat.h), after upload, press the button (9) which is in the 7th row, the 3rd column on the IR remote to trigger the modified **zero** skill. You can see the posture is changed.

The new zero pose looks like this:

The first element (1) represents the total number of frames of the skill, 1 means it is a static posture.

The 4th element (1) represents the Angle ratio. It means all the following indexed joint angles are actual angles (because each of them multiply by 1).

From the 5th to the 20th elements represent 16 indexed joint angles.

For Bittle, the 5th element (joint index 0) means the servo on Bittle's neck rotates counterclockwise 70 (the unit is in degrees). Bittle's head turn to it's left side.

The 13th element (joint index 8) means Bittle's left upper leg rotates clockwise 60 (the unit is in degrees) around the joint.

The 17th element (joint index 12) means Bittle's left lower leg rotates counterclockwise 60 (the unit is in degrees) around the joint.

All the other indexed joint angles remain 0 (the unit is in degrees).

You can define a new posture and upload it to the robot. Call the new posture with the IR remote or the serial monitor.

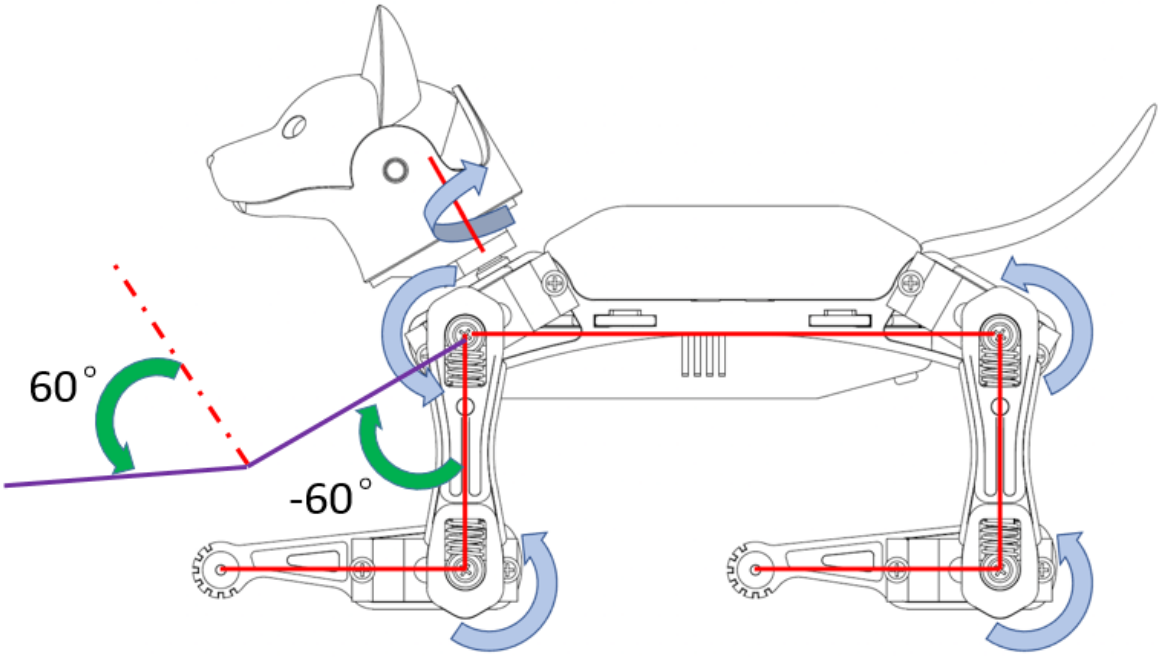


Fig. 8.61: Zero State with Angle

Explain expected body orientations

Look at the example of:

```
const char balance[] PROGMEM = {
1, 0, 0, 1,
0,   0,   0,   0,   0,   0,   0,   0,  30,  30,  30,  30,  30,  30,  30,  30,};
```

and

```
const char sit[] PROGMEM = {
1, 0, -30, 1,
0,   0, -45,   0,  -5,  -5,  20,  20,  45,  45, 105, 105,  45,  45, -45, -45,};
```

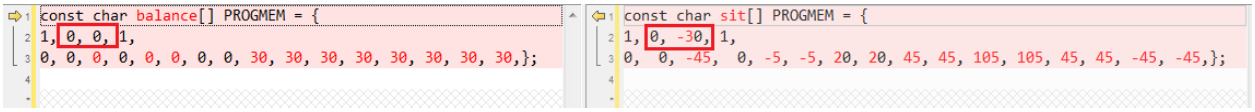


Fig. 8.62: Skill development Code Compare

The 2nd and 3rd elements represent Expected body orientation, corresponding to the roll angle and the pitch angle.

The unit is in degrees.

The sign of the number follows the right-handed spiral rule, look along the direction pointed by the axis arrow, clockwise is positive, counterclockwise is negative.

With the gyro activated, rotate Bittle, when the body is tilted from the expected angles, the balancing algorithm will calculate some adjustments to keep it in this posture.

Explain angle ratio

Look at the example of

```
const char rc[] PROGMEM = {
-3, 0, 0, 2,
0, 0, 0,
0,    0,   0,   0,   0,   0,   0,   0, -88, -43,  67,  87,  42, -35,  42,  42,      15,␣
↪0, 0, 0,
0,    0,   0,   0,   0,   0,   0,   0, -83, -88,  87,  42,  42,  42,  42, -40,      15,␣
↪0, 0, 0,
-8, -20, -11,   0,  -1,  -1,   0,   0,  18,  18,  18,  18, -14, -14, -14, -14,      10,␣
↪0, 0, 0,
};
```

Angle ratio is designed for large angles out of the range -128~127.

The 4th element (2) represents the angle ratio. It means all the following all indexed joint angles real values is equal to each of them multiply by the value of this angle ratio.

Understand the format of a gait

A series of frames defines sequential postures, such as a gait. Find the **bk** array in instinctBittle.h:

bk is the abbreviation for "back".

The first four elements are defined the same as before, The first element (35) means it has 35 frames. Next are 35 frames of 8 indexed joint angles that form a repetitive gait.

Understand the format of a behavior

Modify the zero skill as:

```
const char zero[] PROGMEM = {
-1, 0, 0, 1,
0, 0, 0,
70, 0, 0, 0, 0, 0, 0, 0, -60, 0, 0, 0, 60, 0, 0, 0,4,0,0,0};
```

Fig. 8.63: Zero position

Upload the new zero skill and see the effect. It should be the same. (Later explanation a posture can be considered as a behavior or gait with one frame.)

Copy the content of **hi** array to zero:

Change a few values:

Save and upload OpenCat.ino. Call the new zero skill to see the effect.

For example, the modified **hi** behavior:

The first four elements are defined the same as before, except that the number of frames is saved as a negative value (-3) to indicate that it's a behavior.

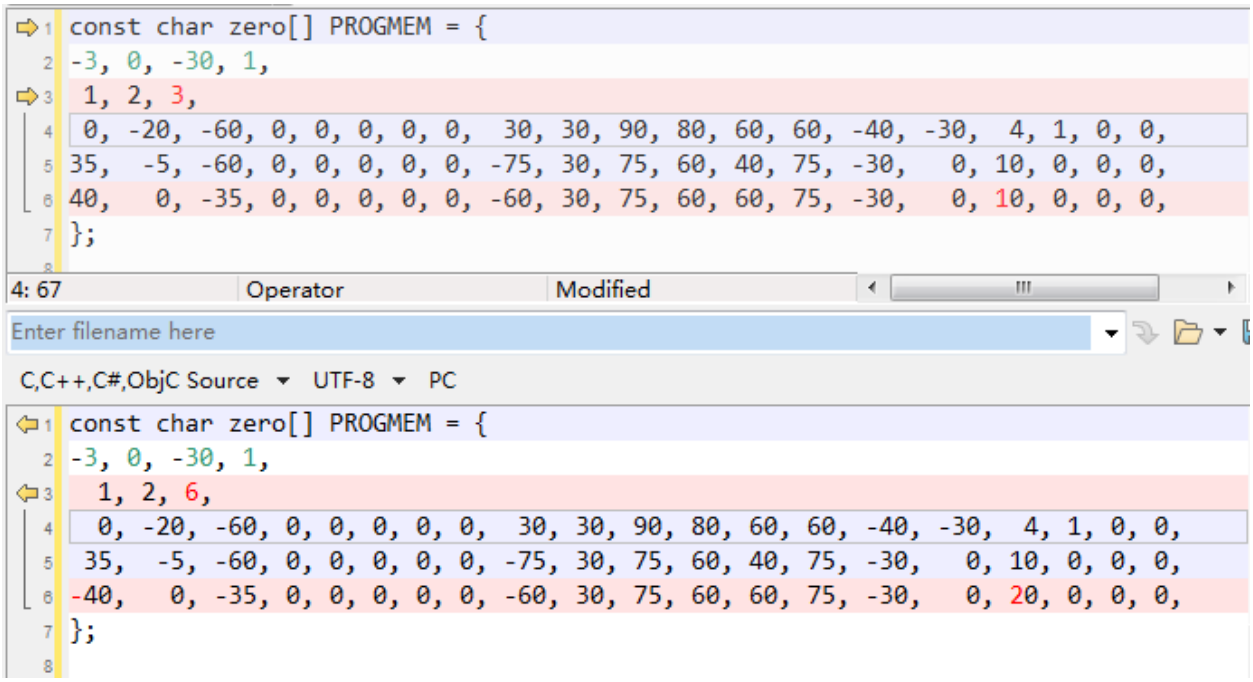


Fig. 8.64: Skill development Code Compare

The 2nd element (0) means the Roll rotation body angle is 0 (The unit is in degrees). The 3rd element (-30) means the Pitch rotation body angle is -30 (The unit is in degrees). If the body is tilted from the expected angles, the balancing algorithm will calculate some adjustments. The 4th element (1) means all the following all indexed joint angles are real values.

The next three elements define the repeating frames in the sequence: starting frame (1), ending frame (2), looping cycles (6). So the **1, 2, 6** in the example means the behavior should loop from the 2nd to the 3rd frame for 6 times (the index starts from 0). The whole behavior array will be executed only once, rather than looping over like the gait.

For behavior, each frame contains 16 indexed joint angles, and the last 4 elements define the speed of the transition, and the delay condition after each transition:

1. The first number represents speed factor. The default speed factor is 4, it can be changed to an integer from 1 (slow) to 127 (fast). The unit is in **degrees per step**. If it's set to 0, the servo will rotate to the target angle by its maximal speed (about 0.07sec/60 degrees). It's not recommended to use a value larger than 10 unless you understand the risks. Here for this example, in the first frame, it is default value (4).

2. The 2nd number represents delay time. The default delay is 0. It can be set from 0 to 127, the unit is 50 ms. Here for this example, in the first frame, it is 1.

3. The 3rd number represents the trigger axis. If it's not 0, the previous delay time will be ignored. The trigger of the next frame will depend on the body angle on the corresponding axis. 1 for the pitch axis, and 2 for the roll axis. The sign of the number defines the direction in which the threshold is exceeded. In the first frame of this example, it is 0, which means this trigger condition is not enabled.

4. The 4th number represents the trigger angle. It can be -128 to 127 degrees. The use of this value needs to be combined with the third number. Only when the robot body rotates in the specified direction exceeds this trigger angle, the action of the next frame can be triggered. In the first frame of this example, it is 0. If the third number is 0, it means that the trigger condition is not enabled, and this value is invalid.

Understand the memory structure

Explanation the locations:

There are two kinds of skills: **Instincts** and **Newbility**. The addresses of both are written to the onboard EEPROM(1KB) as a lookup table, but the actual data is stored at different memory locations:

- I2C EEPROM (8KB) stores **Instincts**.

The Instincts are already fine-tuned/fixed skills. You can compare them to "muscle memory". After uploading the "parameters" firmware, multiple instinctive Instincts write linearly to the I2C EEPROM. Their addresses are generated during the upload of the "parameters" firmware and saved to a lookup table in the onboard EEPROM.

- Flash (sharing the 32KB flash with the program) stores **Newbility**.

A Newbility is any new experimental skill that requires a lot of tests. It's not written to the I2C nor onboard EEPROM, but the flash memory in the format of PROGMEM. It has to be uploaded as one part of the Main function firmware. Its address is also assigned during the runtime of the code, though the value rarely changes if the total number of skills (including all Instincts and Newbilities) is unchanged.

The implementation code:

The first section is active when uploading "parameters" firmware. It contains all the skills' data and pointers. The skill names contain a suffix, "N" or "I" to indicate whether it's a Newbility or Instinct. The Instinct will be saved to the external I2C EEPROM while the Newbility will be saved to the flash. The address of all the skills will be saved to the onboard EEPROM.

The second section is active when uploading Main function firmware. Because the Instincts are already saved in the external EEPROM, their data is omitted to save space. The Newbility will be saved to the flash with the upload so you can update them in the MAIN_SKETCH section . It's very useful if you are still tuning a new skill.

In the example code, only zero is defined as a Newbility so you don't need to re-upload "parameters" firmware for experiments.

Create a new behavior

What if we want to add more skills with customized names and button assignments?

If you want to add more skills, you can refer to the implementation of serial commands (the figure above) and keymap (the 1st figure) in the program code.

Add an example of a Newbility test and assign it to a button on the IR remote. Upload the skill and call it with both IR remote and serial monitor.

Modify a few fields of **test** to make it an **Instinct**. Call the behavior with the IR remote or serial monitor.

> **Warning:** Note: The left side of the above picture is to add a new skill (**Newbility**); the right side is to add a new instinct (**Instinct**).

Remember to add 1 to the number of skills at the beginning of the header file.

Then you can call this test by entering ktest in the serial monitor. You can also call it from the IR remote if you replace a button definition in OpenCat.h.

	Type	Command	Note	备注
Control	Token	d	rest and shutdown all servos	休息并关闭所有舵机
		g	turn on/off gyro to boost speed	开关陀螺仪加速运动
		p	pause motion and shut off all servos	暂停动作并关闭所有舵机
		c	enter calibration mode	进入校准模式
		m	move a joint to certain angle	把某关节转动到某角度
Skill	Gait	bk	back	后退
		bkL	backLeft	后退 左
		bkR	backRight	后退 右
		vt	stepping on the same spot	原地踏步
		crF	crawl	爬 低重心对角步态
		crL	crawl left	爬 左
		crR	craw right	爬 右
		wkF	walk	走 三脚着地的步态
		wkL	walk left	走 左
		wkR	walk right	走 右
		trF	trot	小跑 对角步态
		trL	trot left	小跑 左
		trR	trot right	小跑 右
		bdF	bound (not recommanded)	兔子跳（不推荐）
	Posture	balance		正常站立演示自平衡性
		buttUp	buttom up	撅屁股
		calib	calibration	校准姿态
		rest		休息
		sit		坐
		sleep		睡
		str	stretch	伸懒腰
		zero		零姿势 给用户自己设计动作的模板
	Behavior	ck	check around	左右看
		hi	hi sequence	打招呼
		pee	pee sequence	撒尿
		pu	push up sequence	俯卧撑
		rc	recovering sequence	四脚朝天后恢复站立(自动激活)
		pd	play dead	主动翻身倒地装死
		bf	back flip	后空翻（隐藏）
You need to add a 'k' before skills. For example, kbk, kbalance, kck...				
The Gaits can be the result of combined command: gait+direction				
		Gait	Direction	Serial Command
		cr	F	kcrF
		wk	L	kwkL
		tr	R	ktrR

Fig. 8.65: Serial Commands

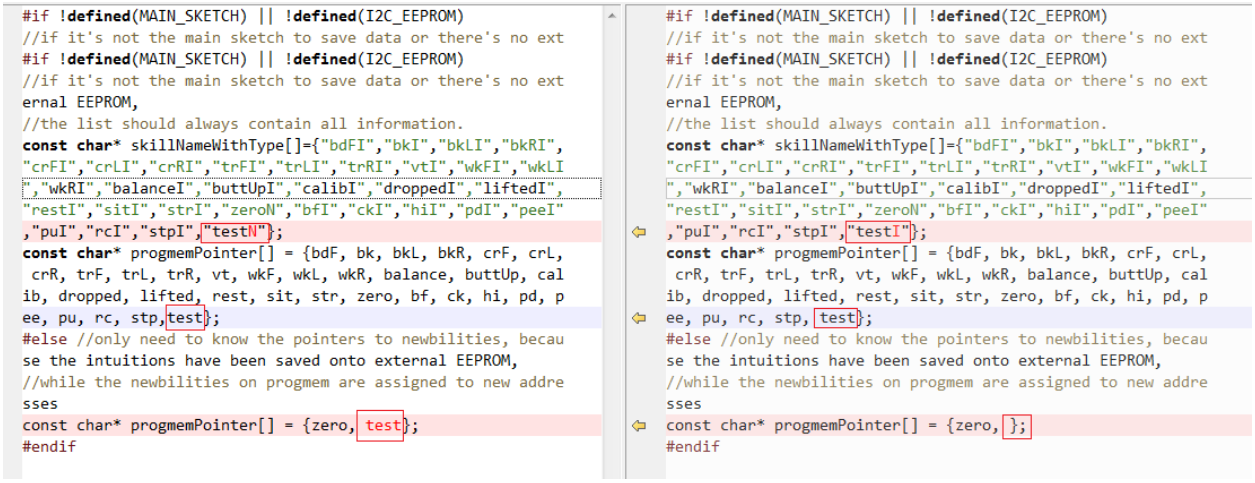


Fig. 8.66: Skill development Code Compare

Tune skills in realtime

You need to understand the above structure to store a skill on the robot. However, when tuning the skills, things can be easier. To do this, you can connect your computer with the robot through the USB or Bluetooth connector. Then you can send string commands that define the joint angles. The program on the NyBoard will listen and perform the instructions in real-time. In the folder **SerialMaster** , you can find the **ardSerial.py** to play the role of Arduino IDE's serial monitor, and you can also write scripts to control Bittle to do more complex movements. Some use cases are listed in **example.py**.

Please refer to **Controlling with Python** section for more details.

8.12 Controlling Bittle (Advanced)

Danger: CONTENT UNDER REVIEW. Please ask the instructor before continuing with this section.

This section will described advanced methods to control your robot.

8.12.1 Using WiFi Adaptor

About the Sample Code: The sample code is a simple web server example, including 2 HTML pages. The two pages are stored in two header files in the form of string constants. The advantage is to avoid calling the client.print function constantly.

Set Up the WiFi Networks: Before we start our web server, we should configure the WiFi to connect to your local area network(LAN). We used to enter the WiFi SSID and password in the program, but it is very inconvenient while we need to change the network environment. We use the WiFi manager library to configure the WiFi information through web.

```
// WiFiManager
WiFiManager wifiManager;
```

(continues on next page)

(continued from previous page)

```
// Start WiFi manager, default gateway IP is 192.168.4.1
wifiManager.autoConnect("Bittle-AP");
```

Web server: Create a new web server and configure port 80 (commonly used HTTP server port)

```
ESP8266WebServer server(80);
```

Configure 3 HTTP service handler: The HTTP response function is to handle the incoming HTTP requests.

```
void handleMainPage() {
//Serial.println("GET /");
server.send(200, "text/html", mainpage);
}
void handleActionPage() {
//Serial.println("GET /actionpage");
server.send(200, "text/html", actionpage);
}
```

The handleMainPage and handleActionPage response 200 (OK) and corresponding web HTML code for your web browser (client).

```
void handleAction(){
String argname = server.arg("name");

if(argname == "gyro"){              // gyro switch
    Serial.print("g");
}
```

The HandleAction function is slightly different. This is an HTTP request processing function with parameter passing. When the parameter is "gyro", the serial port of the WiFi module sends out the command ("g", switch IMU), so that our Bittle will execute the command. So how is this "gyro" parameter generated and passed? Because we sent such an HTTP request with a value to the server:

```
http://IP address or DomainName/action?name=gyro
```

The server parses the action parameter by the function and resolves that the name is "gyro". We can directly enter this URL in the browser and execute it with the keyboard. The more common method is to add a link to the "Walk" button on the ActionPage web page. When the gyro button is pressed, the above URL will be sent to the host. The complete walk button configuration is as follows:

```
<button style="width: 25%" onclick="location.href='/action?name=gyro'">GyroOn/Off</
↪button>
```

After parsing the "name" parameter, we send the actionpage again.

```
server.send(200, "text/html", actionpage);
```

We bond the handler method with the corresponding URLs.

```
server.on("/", handleMainPage);
server.on("/actionpage", handleActionPage);
server.on("/action", handleAction);
```

Start the Web Server:

```
server.begin();
Serial.println("HTTP server started");
```

Handle Client Requests:

```
void loop(void){
server.handleClient();
}
```

8.12.2 Using Python Scripts

The list testSchedule in example.py is used to test various serial port commands. Run the following script code to see the execution effect of each serial port command in the list:

```
for task in testSchedule:
wrapper(task)
```

You can also refer to the content of the stepUpSchedule list (in ..\serialMaster\demos\stepup.py), write a list of behaviors according to your actual needs, and realize your creativity.

Note: When running the scripts under the path of serialMasterdemos, you must first use the "cd demos" command to enter the path where the scripts are located (\serialMaster\demos), and then use the python3 command to run the script (e.g. "python3 stepup.py")

Explanation of the serial port commands in the list testSchedule:

['kbalance', 2]

- 'kbalance' indicates the command to control Bittle to stand normally
- 2 indicates the postponed time after finishing the command, in seconds

['d', 2]

- d indicates the command to put the robot down and shut down the servos
- 2 indicates the postponed time after finishing the command, in seconds

['c', 2]

- c indicates the command to enter calibration mode
- 2 indicates the postponed time after finishing the command, in seconds. After these motion commands are completed, the next command will be executed after a 2-second delay.

['c', [0, -9], 2]

- c indicates the command to enter calibration mode
- 0 indicates the index number of joint servo
- -9 indicates the rotation angle, the unit is degree
- 2 indicates the postponed time after finishing the command, in seconds

Using this format, you can enter the calibration mode to calibrate the angle of a certain joint servo. Note: If you want the correction value in this command to take effect, you need to enter the "s" command after executing this command.

The meaning of this example: the joint servo with serial number 0 rotates -9 degrees. After these motion commands are completed, the next command will be executed after a 2-second delay.

['m', [0, -20], 1.5]

- m indicates the command to control the rotation of the joint servo
- 0 indicates the index number of joint servo
- -20 indicates the rotation angle (this angle refers to the origin, rather than additive) the unit is degree
- 1.5 indicates the postponed time after finishing the command, in seconds. It can be a float number.

['m', [0, 45, 0, -45, 0, 45, 0, -45], 2]

Using this format, multiple joint servo rotation commands can be issued at one time, and these joint servo rotation commands are executed SEQUENTIALLY, not at the same time. The joint angles are treated as ASCII characters, so they can be entered directly by humans.

The meaning of this example is: the joint servo with index number 0 is first rotated to the 45 degree position, and then rotated to the -45 degree position, and so on. After these motion commands are completed, the next command will be executed after a 2-second delay.

['i', [8, -15, 9, -20], 2]

Using this format, multiple joint servo rotation commands can be issued at one time, and these joint servo rotation commands are executed AT THE SAME TIME. The joint angles are treated as ASCII characters, so they can be entered directly by humans.

The meaning of this example is the joint servos with index numbers 8, 9 are rotated to the -15, -20 degree positions at the same time. After these motion commands are completed, the next command will be executed after a 2-second delay.

['M', [8, 50, 9, 50, 10, 50, 11, 50, 0, 0], 3]

- M indicates the command to rotate multiple joint servos SEQUENTIALLY. The angles are encoded as BINARY numbers for efficiency.
- 8, 9, 10, 11, 0 indicate the index numbers of joint servos
- 50, 50, 50, 50, 0 indicate the rotation angle (this angle refers to the origin, rather than additive), the unit is degree
- 3 indicates the postponed time after finishing the command, in seconds

['I', [20, 0, 0, 0, 0, 0, 0, 0, 45, 45, 45, 45, 36, 36, 36, 36], 5]

- I indicates the command to control all joint servos to rotate AT THE SAME TIME (currently the command supports 16 degrees of freedom, that is, 16 servos) . The angles are encoded as BINARY numbers for efficiency.
- 20,0,0,0,0,0,0,0,45,45,45,45,36,36,36,36 indicate the rotation angle of each joint servo corresponding to 0-15 (this angle refers to the origin, rather than additive), the unit is degree
- 5 indicates the postponed time after finishing the command, in seconds

['b', [10,2], 2]

- b indicates the command to control the buzzer to beep
- 10 indicates the music tone
- 2 indicates the lengths of duration, corresponding to 1/duration second
- 2 indicates the postponed time after completing the tone, in seconds

['b',[0, 1, 14, 8, 14, 8, 21, 8, 21, 8, 23, 8, 23, 8, 21, 4, 19, 8, 19, 8, 18, 8, 18, 8, 16, 8, 16, 8, 14, 4],3]

- b indicates the command to control the buzzer to beep

- 0, 14, 14, 21… indicate the music tones
- 1, 8, 8, 8 indicates the lengths of duration, corresponding to 1/duration second
- The last 3 indicates the postponed time after the music melody is played, in seconds

Using this format, multiple tone pronunciation commands can be issued at once, and a simple melody can be played.

The meaning of this example is: play a simple melody, and delay 3 seconds after the music melody is played.

```
ck = [
    -3, 0, 5, 1,
    0, 1, 2,
    45, 0, 0, 0, 0, 0, 0, 0, 45, 35, 38, 50, -30, -10, 0, -20, 6, 1, 0, 0,
    -45, 0, 0, 0, 0, 0, 0, 0, 35, 45, 50, 38, -10, -30, -20, 0, 6, 1, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 30, 30, 30, 30, 30, 30, 30, 30, 5, 0, 0, 0,
    ]
```

['K', ck, 1]

- 'K' indicates the skill data to send to Bittle in realtime
- The skill array is sent to the robot on the go and executed locally on the robot
- You may insert the skills in the skill library or InstinctX.h in this format

8.12.3 Creating New Skills

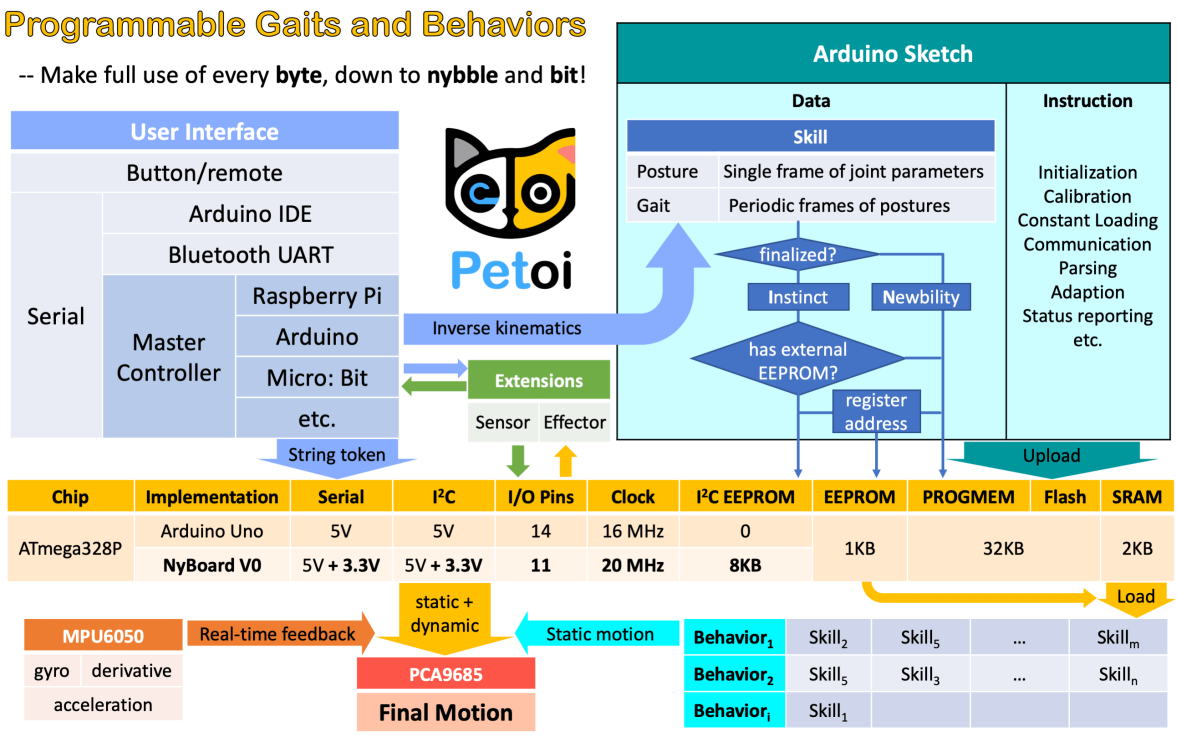


Fig. 8.67: InstinctBittle.h file Summary

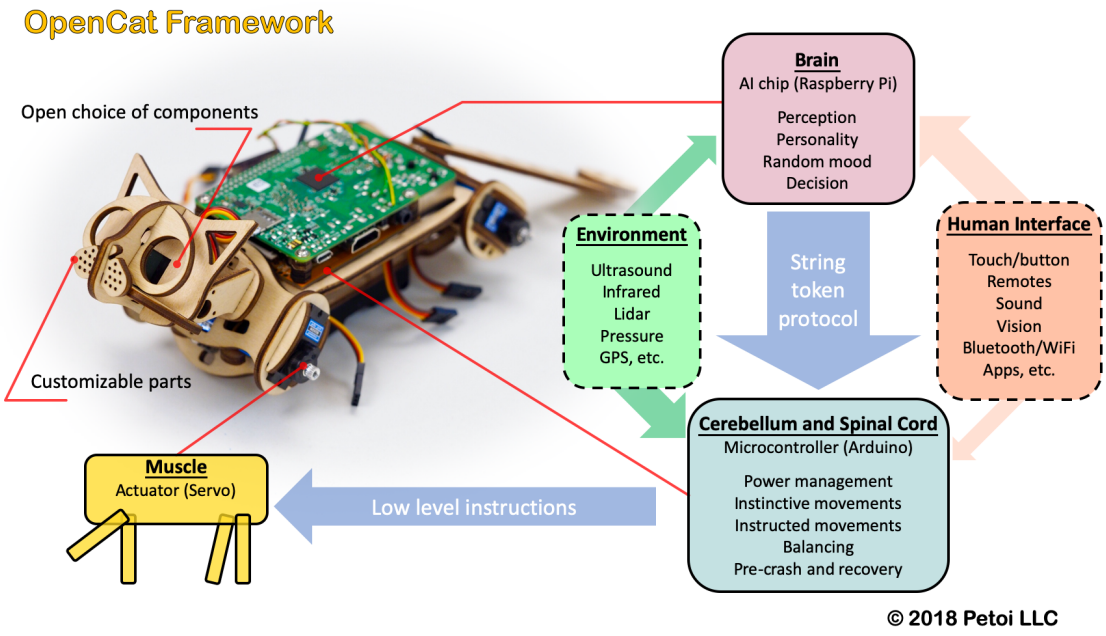


Fig. 8.68: OpenCat Framework Summary

8.13 Controlling Bittle (Codecraft)

8.13.1 Introduction

Codecraft is a graphical programming software. It is based on the Scratch 3.0 language and enables programming by simply "drag & drop" of predefined blocks. It is available as a web and downloadable application.

The web version is available at (https://ide.tinkergen.com/). Here you can make codes but to upload to your hardware, you would need a client application for your PC. You can find more information here: (https://www.yuque.com/tinkergen-help-en/codecraft/assistant?language=en-us)

To download complete software, from the following link: (https://ide.tinkergen.com/download/en/). This includes IDE and uploader for your hardware.

When you open the application, you have to select the hardware you will use for programming. In our case, it's **Bittle**. After that you will be redirected to the programming UI.

For detailed information about UI elements, visit (https://www.yuque.com/tinkergen-help-en/codecraft/tour_overview)

Warning: Note that Codecraft doesn't support OpenCat 2.0. You'd need to use OpenCat 1.0 instead. You can find the details here: https://github.com/PetoiCamp/OpenCat/tree/1.0

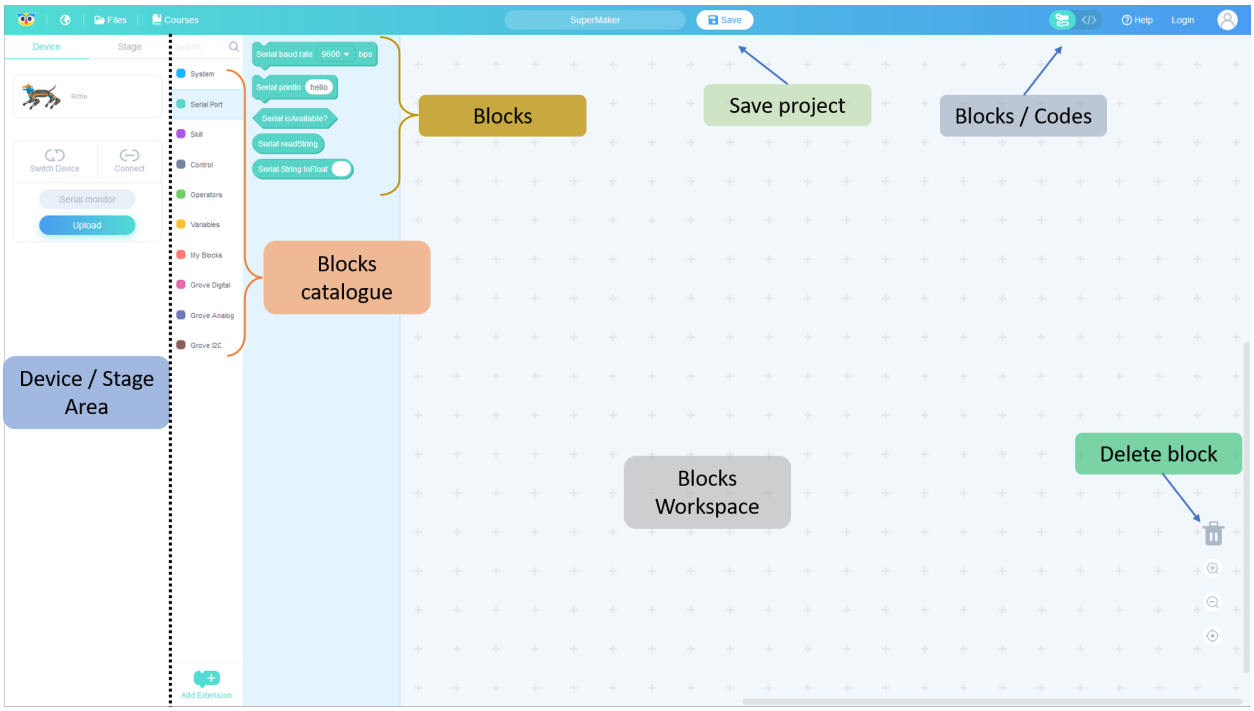


Fig. 8.69: Codecraft UI Guide

Upload Code

To upload the code from Codecraft, you would need to connect the USB-Adaptor (**First choice**). You can also use a Bluetooth Adaptor.

From the **Device / Stage** section, use the **Connect** button to connect to your device. Then use the **Upload** button to upload code.

Monitor Serial Values

From the **Device / Stage** section, use the **Connect** button to connect to your device. Then use the **Serial monitor** button to upload code.

> **Warning:** If the walking movements in the following tutorials are not working as intended. Try re-calibrating the motors. Visit the **Calibration** section for more information.

8.13.2 Programming with Blocks

Block Shapes

There are five different block shapes in Codecraft:

- Hat Block
- Stack Block
- Boolean Block

- Reporter Block
- C Block

1. Hat Block Hat blocks are the blocks that start every script. The rest of the blocks should be attached to top hat block. Please note that different hat blocks start each program code in different ways.

2. Stack Block Stack blocks are the blocks that perform the main commands. They are shaped with a notch at the top and a bump on the bottom — this is so blocks can be placed above and below them.

3. Boolean Block A Boolean block can be placed in corresponding hexagon slots and could not be used separately. Boolean is a condition (true or false).

4. Reporter Block Reporter blocks can be placed anywhere that needs data and could not be used separately. As long as there is a slot corresponding to a reporter block, it can be placed as needed. Reporter blocks can hold numbers and strings.

5. C Block C blocks are blocks that take the shape of "C's". Also known as "Wrap blocks", these blocks loop the blocks within the Cs or check if a condition is true.

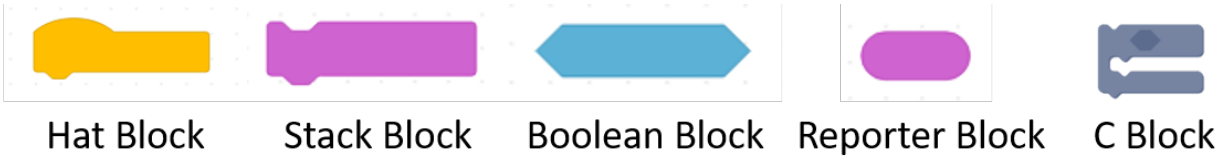


Fig. 8.70: Codecraft Block Guide

Basics

To begin programming, you would need a Setup & Loop block. You can grab one from System block catalogue. This resembles the two sections of an Arduino code.

- The **Setup** block must include code line which are required to be executed only once such as initializing new hardware or defining pin behaviors (whether input pin or output pin).
- The **Loop** block includes code lines which need to be called repeatedly. The sequence of execution if from **top to bottom**. This can be used for turning LED on and off after every 1 second.

The simplest code for movement includes at least three blocks:

- setup-loop block
- set movement block
- execute movement block

The movement blocks have many lines of codes. Each line represents a set of joint angles which is sent from the Bittle's memory to the motors. The block has the required angle list to complete **1** cycle of walk.

Thus to have a continuous walking movement, you need to repeat the block for some time. In terms of Bittle, this execution differs from other wheel based surface robots. In wheel based surface robots, you only need to set the motors on and off to control the movement. But in case of Bittle or other Legged robots, you need to repeat the movement action for certain amount. An example of moving forward and backward is shown below. In this example, the movements are repeated for 1000 times before changing the action.

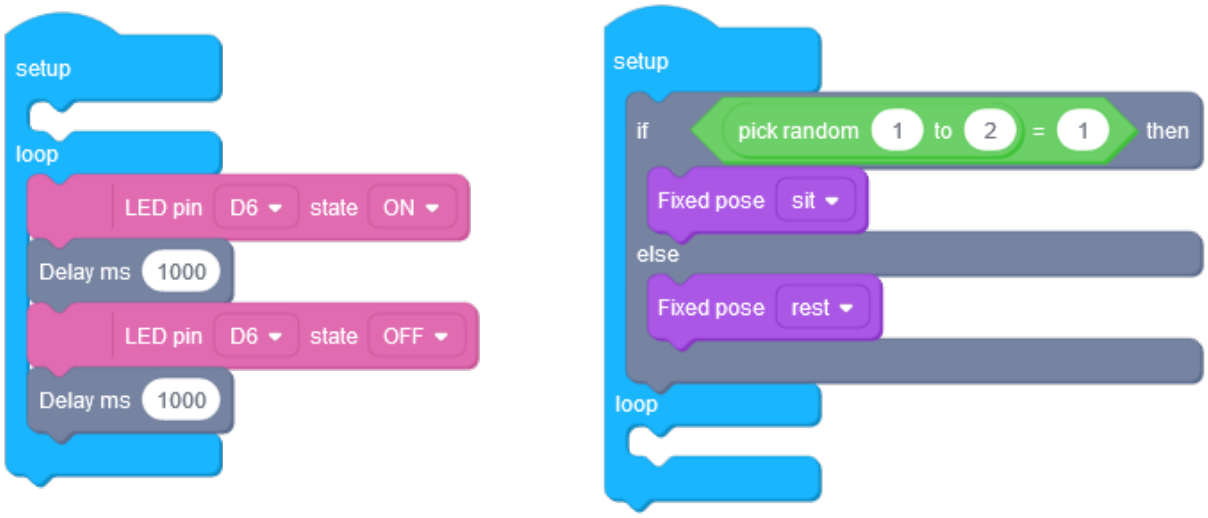


Fig. 8.71: Example: Blink & Initial posture of Bittle

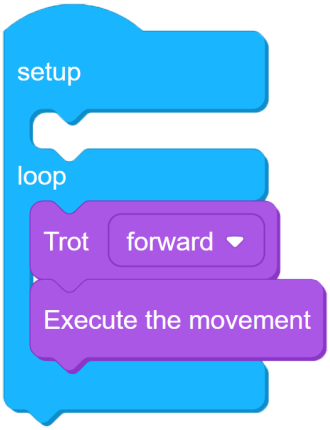


Fig. 8.72: Example: Basic Movements

```
const char tr[] PROGMEM = {
24, 0, 0,
 49, 61, 53, 66, 10, -9, 12, -6,
 52, 49, 56, 55, 11,-12, 14,-10,
 55, 34, 58, 41, 13, -8, 17, -9,
 55, 21, 59, 28, 18,  2, 21, -1,
 57, 15, 60, 21, 21, 17, 26, 14,
 58, 21, 60, 27, 26, 14, 31, 12,
 58, 26, 60, 32, 31, 11, 38, 10,
 58, 31, 58, 36, 38,  9, 46,  9,
 63, 35, 60, 41, 36,  8, 51,  9,
 74, 40, 73, 45, 16,  7, 25,  9,
 73, 43, 74, 48,  3,  9, 10,  9,
 67, 47, 70, 51, -6,  9, -2, 11,
 57, 50, 62, 54,-12, 10, -9, 13,
 43, 53, 49, 57,-10, 12,-10, 15,
 27, 54, 35, 58, -4, 16, -6, 19,
 17, 56, 24, 59,  9, 19,  5, 23,
 18, 58, 24, 60, 16, 23, 14, 28,
 23, 58, 29, 60, 13, 28, 11, 33,
 28, 59, 34, 59, 10, 33, 10, 41,
 33, 58, 38, 54,  9, 40,  9, 56,
 37, 68, 42, 67,  8, 29,  9, 39,
 41, 75, 46, 75,  9, 11,  9, 20,
 45, 71, 49, 74,  9, -1, 10,  5,
 48, 63, 52, 68,  9, -8, 11, -5,
};
```

Fig. 8.73: Servo angles for forward movement

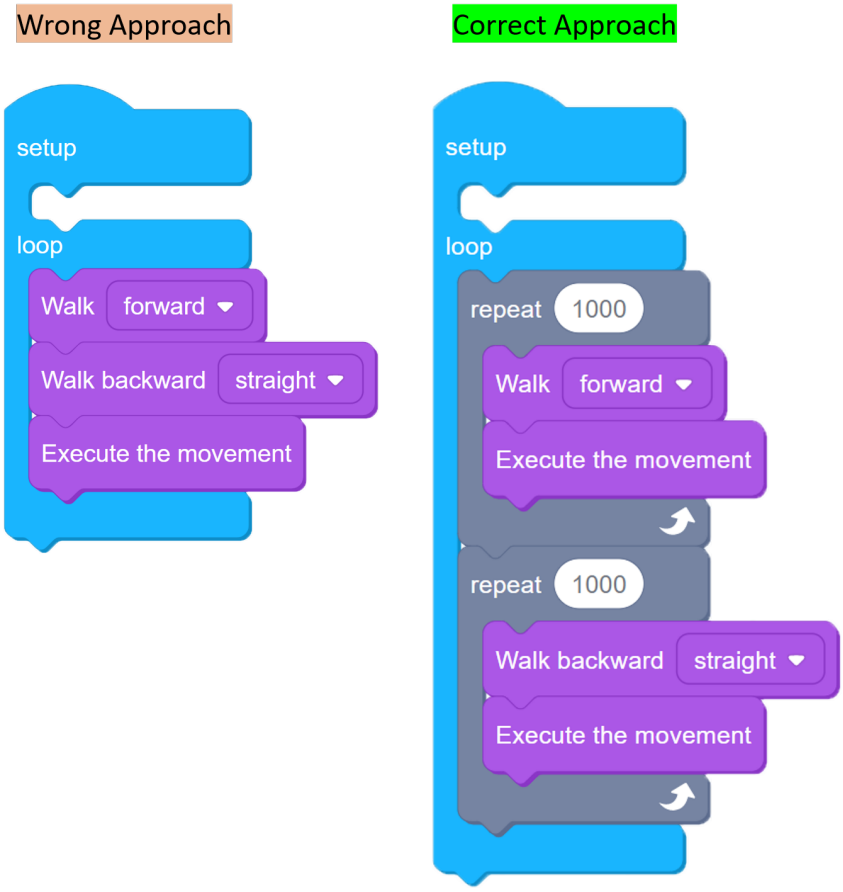


Fig. 8.74: Example: Multiple Actions

Variables

To define custom variables, you can add new variables using **Variables** section in block catalogue. The current version only supports variables of type float.

My Blocks

To create custom function block, use the **My Blocks** section in block catalogue. You can provide 2 type of inputs for the function block; a float value and a boolean value.

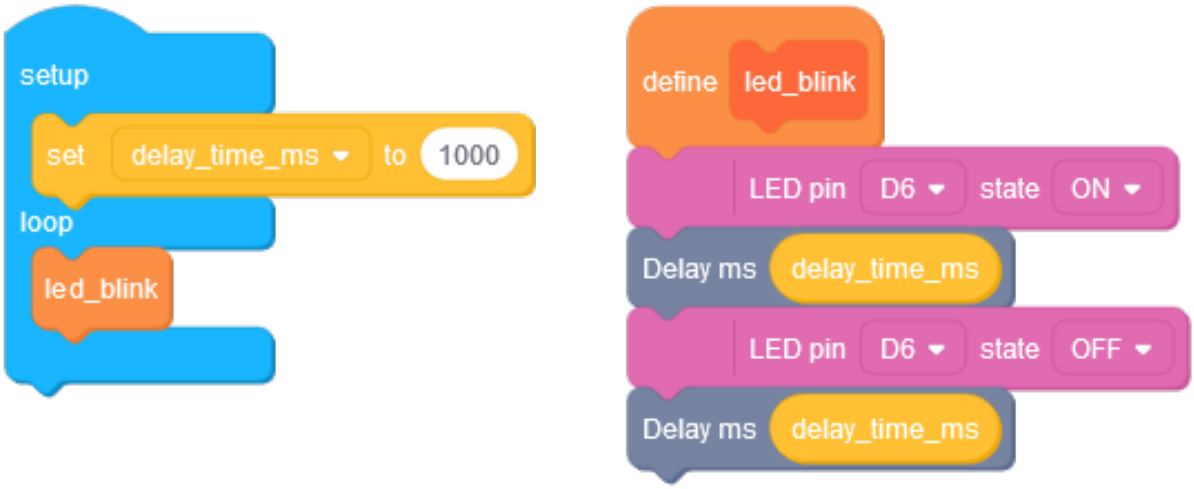


Fig. 8.75: Example: Variables & Blocks

IR-Remote Control

IR Remote can be used in combination to the movement commands to control the robot. The key mapping in reference to Codecraft is provided below:

To include IR library in your code, you will need to add a **Infrared Receiver started** block in your code. This adds the required library and enables the commands related to that library. An example to stop Bittle using IR Remote is shown below. The robot will walk in forward direction until **Play/Pause** button is pressed on the remote.

You can even use serial monitor to print the raw IR codes of buttons pressed on remote.

Advanced Control

Custom Skills

You can create custom skills for your Bittle. To do so, from the Codecraft UI click on **Add Extension**. Select **Create Skill** from the **Extension Library** pop-up. A new catalogue will be added named as **Create Skill**.

Inside **Create Skill**, you can set the Joint angles for all the motors on your robot and can create a sequence of different positions to form a complete action just like the walking forward example shown earlier.

CH-	Switch off servos and rest	CH	Forward	CH+	Switch gyro pose correction on/off
⏮	Turn left	⏭	Stand upright	⏯	Turn right
−	Switch off servos	+	Backward	EQ	Calibration pose
0	Step in place	100+	Crawl	200+	Walk
1	Trot	2	Run	3	Jump
4	Greeting sequence	5	Push-ups	6	Stand on three legs
7	Stretch	8	Sit	9	Custom pose

Fig. 8.76: Example: IR Remote Codecraft

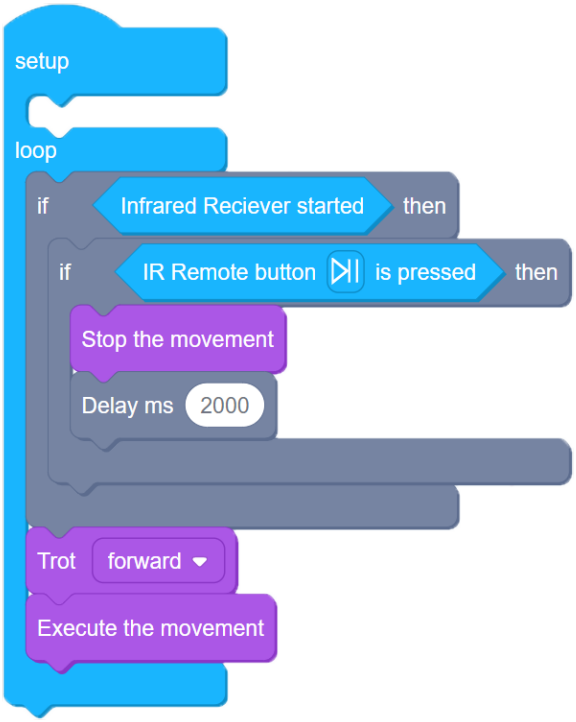


Fig. 8.77: Example: IR Remote Demo

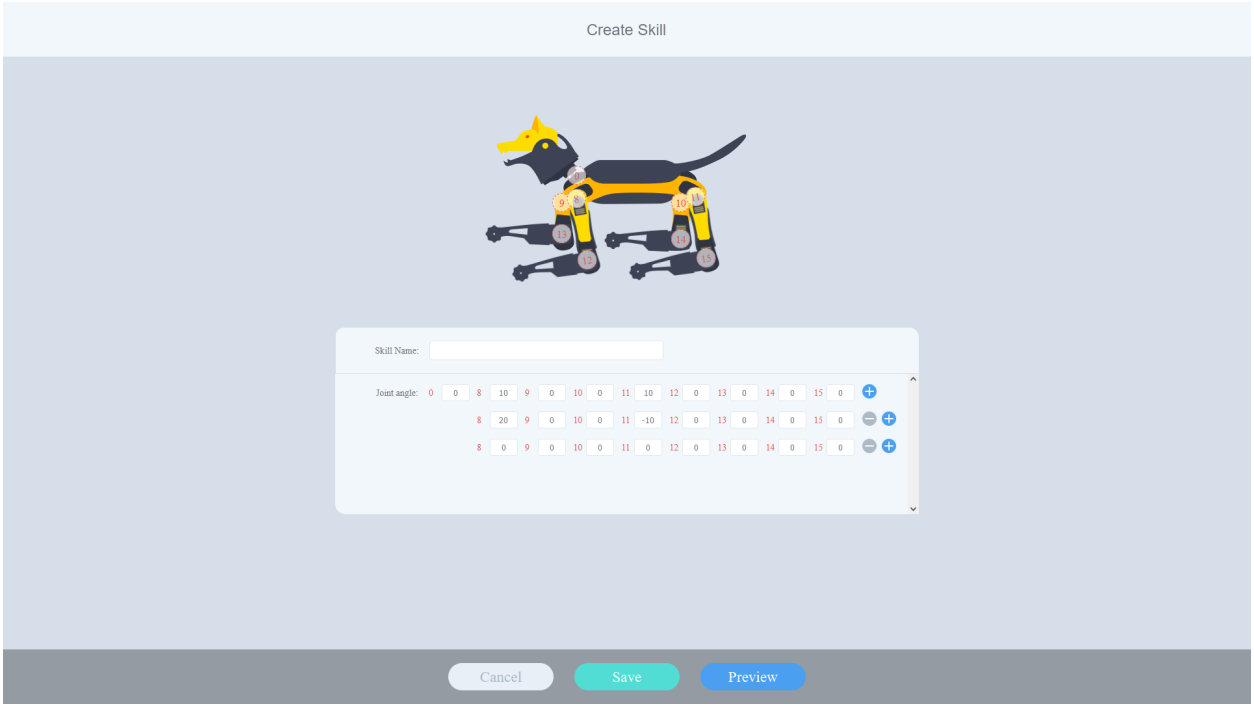


Fig. 8.78: Example: IR Remote Codecraft

Example Block Codes

References

1. Codecraft website https://ide.tinkergen.com/
2. Codecraft tutorials https://www.yuque.com/tinkergen-help-en/codecraft/overview
3. Codecraft Bittle tutorials https://www.yuque.com/tinkergen-help-en/bittle_course/preface

8.14 Problem Solving

8.14.1 Buzzer Sounds

The robot will keep pausing its movements with **beeping sounds** when the battery is low. The battery needs to be charged using a a 5V micro-USB cable. Considering safety, the battery won't supply power during charging.

Buzzer sound meaning:

Sound type	Occasion	Explanation
Short melody	Power on or reboot	The program starts successfully
Short beep	During use	The program receives a command
Repetitive melody	During use and pausing the movements	The battery is low or unconnected

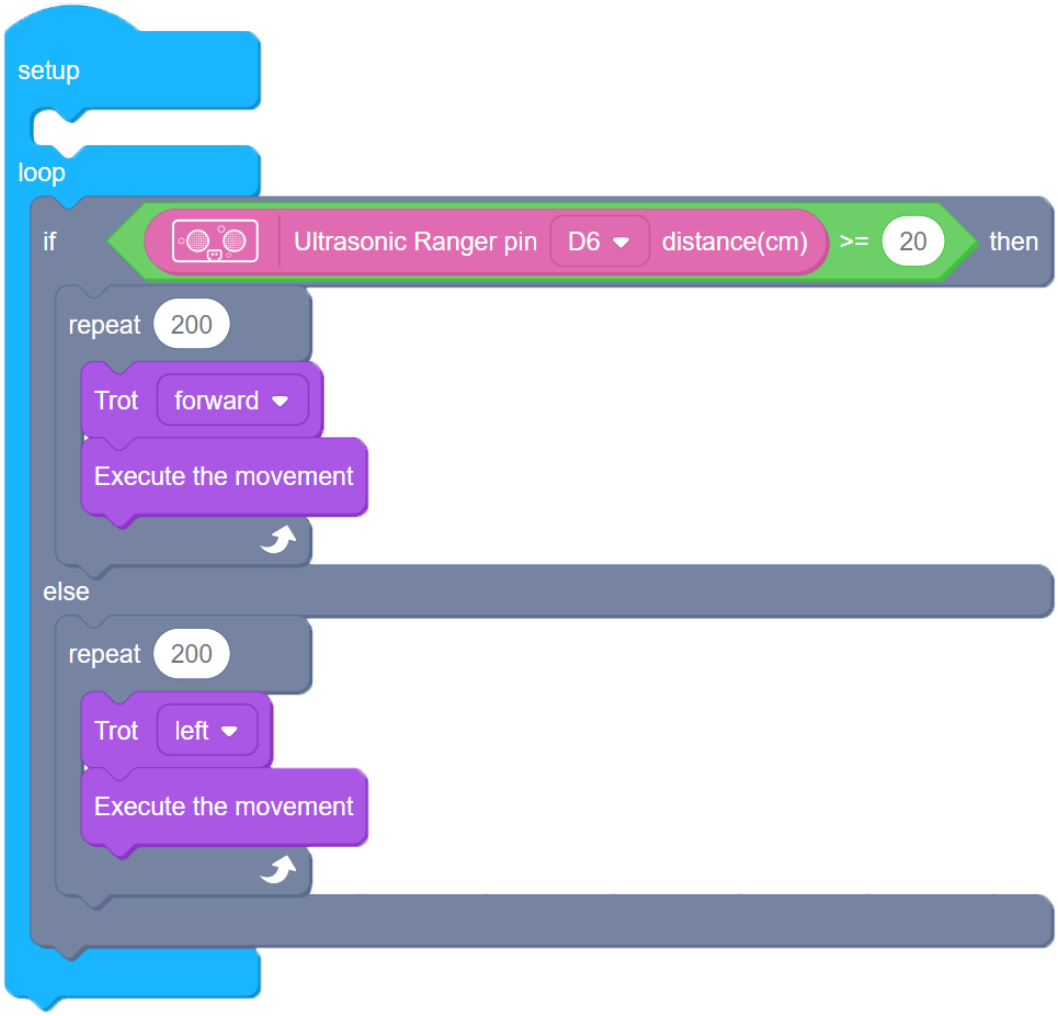


Fig. 8.79: Example: Ultrasonic Sensor Codecraft

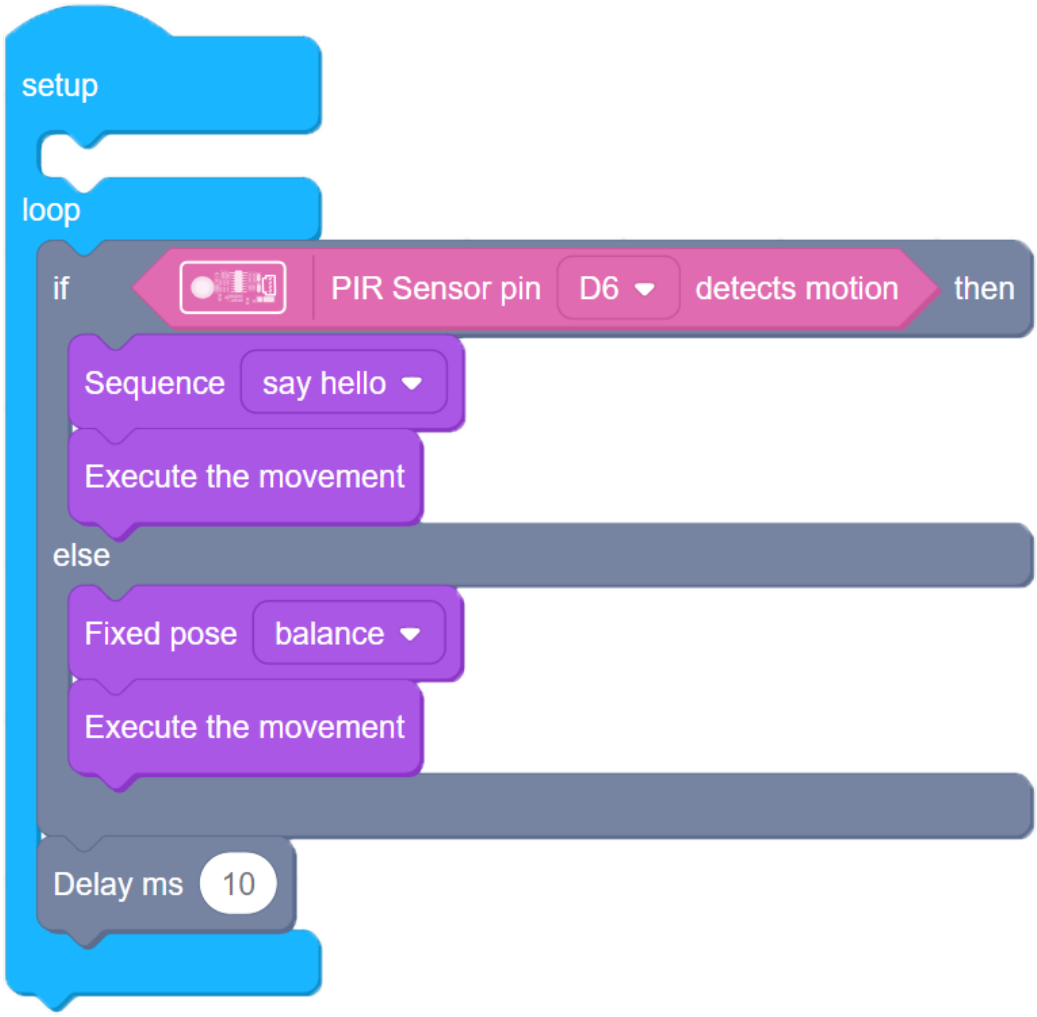


Fig. 8.80: Example: Passive infrared (PIR) Sensor Codecraft

8.14.2 Download Base Firmware

The OpenCat repository consists of various codes from module test to base firmware.

> **Warning:** After downloading Please **DO NOT** modify any file until specifically mentioned in the tutorial.

To begin with download,

1. Copy the link and open it in a browser, the download should start automatically -> https://github.com/PetoiCamp/OpenCat/archive/61fb3efe670c947791739dba86d87118f6021bb8.zip
2. Extract the files and rename the folder to OpenCat. Please check that the files are in following structure **/OpenCat/OpenCat.ino**

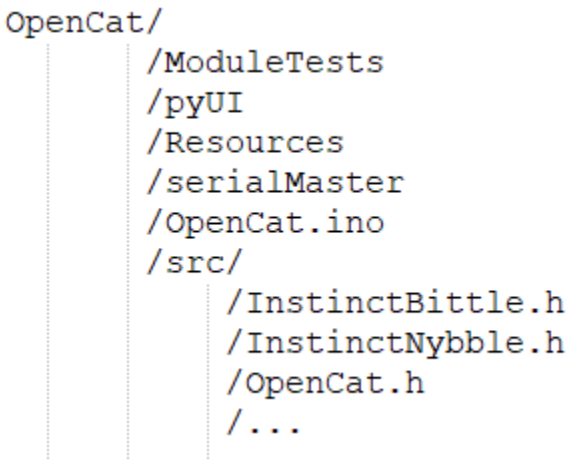


Fig. 8.81: Required file structure

8.14.3 Install Base Firmware

If in case you are unable to configure your device or its not functioning as intended, try installing the firmware again.

> **Warning:** Before proceeding, please confirm that all wire connections are OK and your code is error free.
>
> If the problem is in your code, the base firmware won't be able to fix it.

Follow the steps below to install base firmware:

1. Open Arduino IDE / Platform IO.
2. Open **OpenCat.ino** inside your IDE.
3. In the code, select your robot and board version. Uncomment the required lines by removing **//** and keep the rest of the code as it is.
4. Comment out #define MAIN_SKETCH (if you want to configure your board) by adding **//** in front of the line.
5. Set the Board configurations in IDE (Tools-> Board) to Arduino UNO and select the correct COM port.
6. Upload the code and open serial monitor.

7. Wait until you see **Ready** on serial monitor.

If your calibration is successful, Uncomment #define MAIN_SKETCH to make it active. Upload the code again after that.

8.14.4 Problem Loading Base Firmware

There can be multiple reasons for this, but the most common is incorrect file structure. This file corresponds to the source code downloaded from the link provided. Please consider checking the file structure and comparing it with the structure provided in **Download Base Firmware**.

8.14.5 Problem Uploading Program

Please check if the I2C switch on the main board is in correct mode. The I2C switch changes the master of I2C devices (gyro/accelerometer, servo driver, external EEPROM). On default "Arduino", NyBoard uses the onboard ATmega328P as the master chip. On "RPi", NyBoard uses external chips connected through the I2C ports (SDA, SCL) as the master chip.

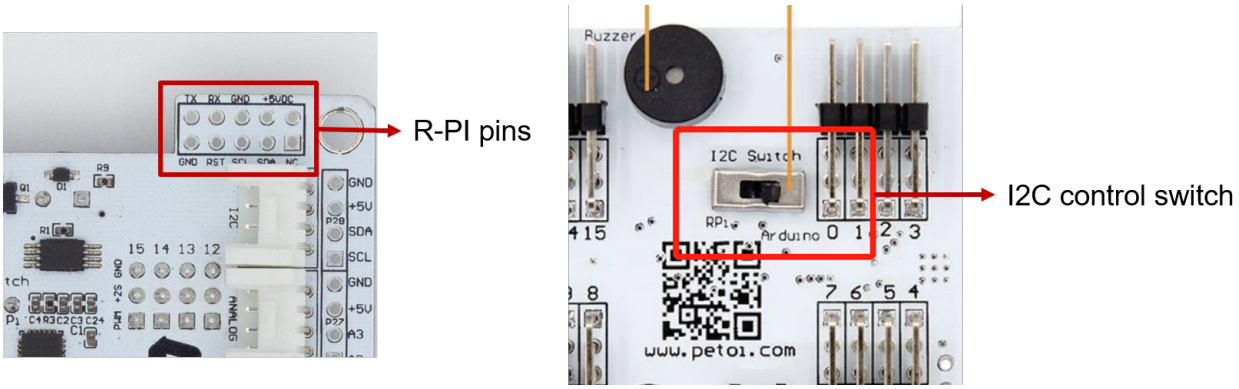


Fig. 8.82: I2C switch

8.14.6 Problem with Device COM Port

If you cannot find the port for your device on IDE, check the **device manager** if your device is connected properly and you have the required device driver installed.

The driver required is CH340. You can download it from here -> http://www.wch-ic.com/downloads/CH341SER_ZIP.html

8.14.7 Problem with Serial Port Permission

If you are using LINUX system and have issues uploading or connecting to COM port, please follow the instructions mentioned here -> https://playground.arduino.cc/Linux/All/#Permission

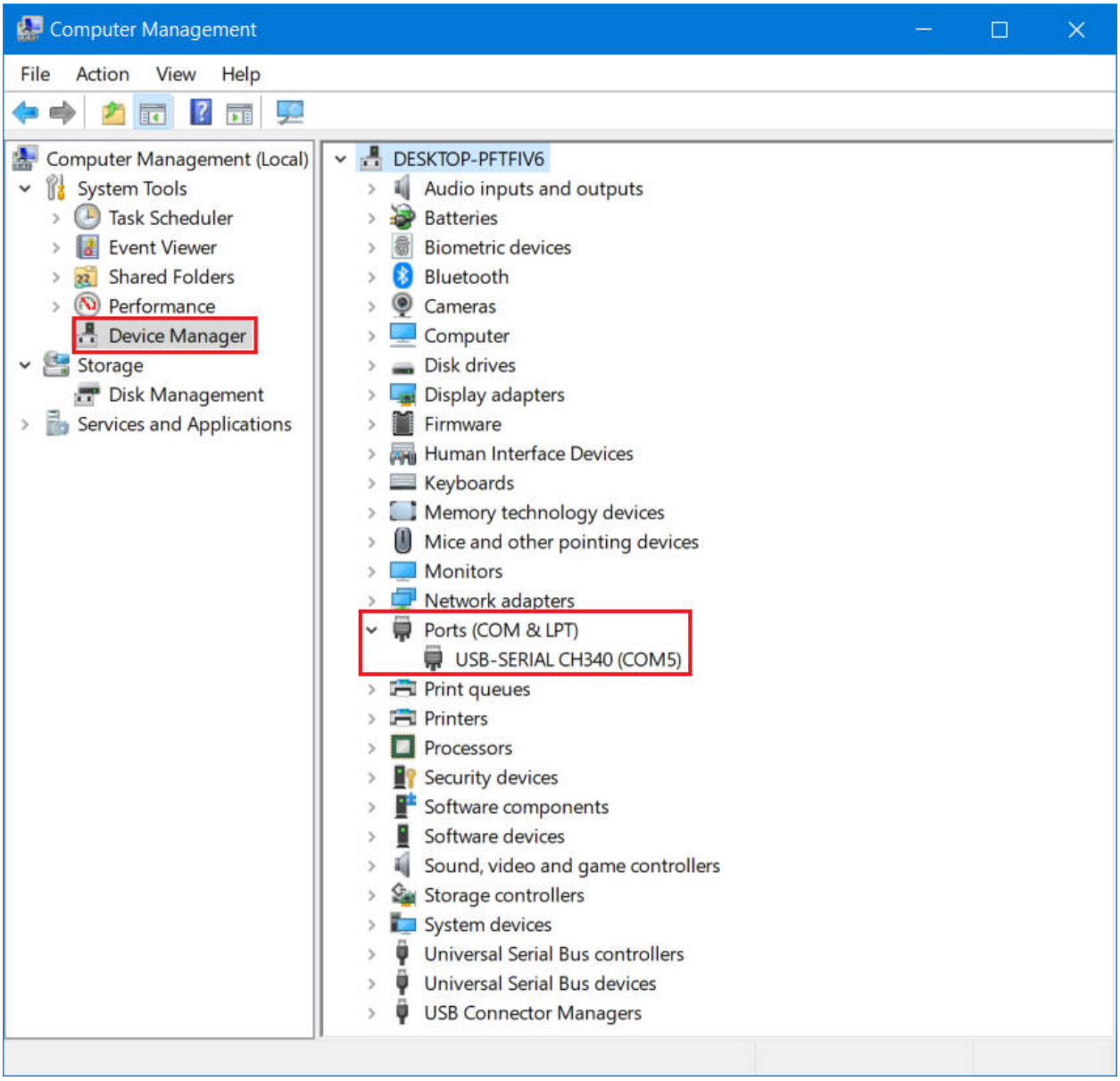


Fig. 8.83: Device manager

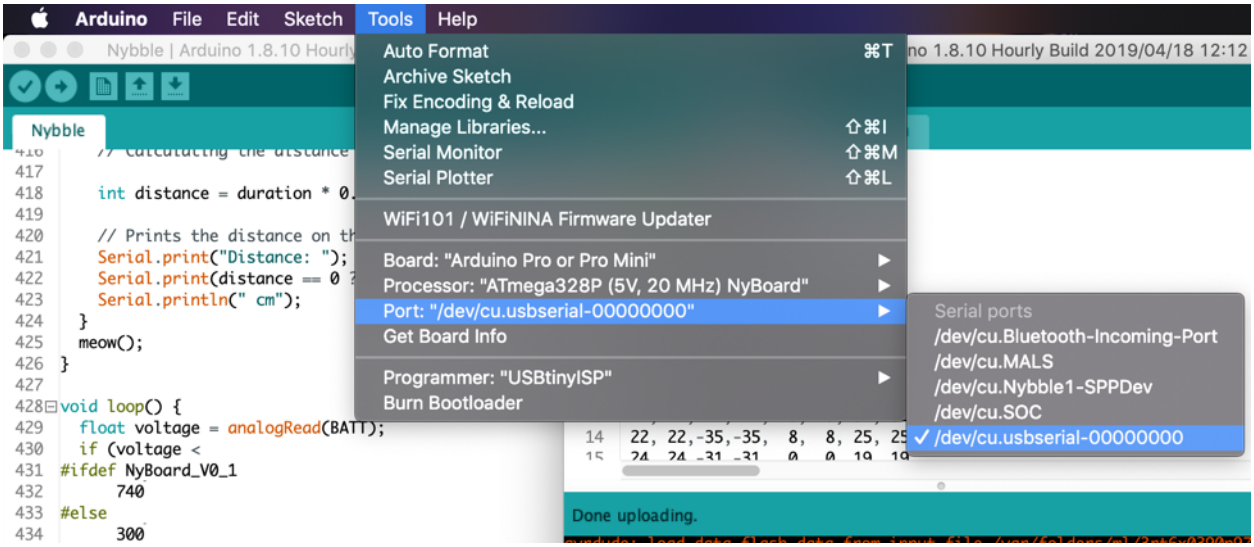


Fig. 8.84: Port selection in IDE

8.14.8 Individual Module Test

You can test individual modules of the robot by uploading the codes available under the folder **ModuleTests**. To do do, open the .ino file from the required test module folder and upload the code by keeping the board setting mentioned earlier.

It is suggested to begin the module test with **testBuzzer.ino** code.

8.14.9 Problem with Python Packages

The Terminal is a built-in interface on Mac or Linux machines. The equivalent environment on Windows machines is called the Command-Line Tool (CMD). It's recommended that you install **Anaconda** to manage your Python environment. It can also provide the Powershell as a Terminal for older Windows machines.

The python scripts work on Python V3. Install pip if not already installed -> https://pip.pypa.io/en/stable/installation/

Install the following packages using pip:

- pyserial
- pillow

```
pip3 install pyserial pillow
```

To run the code:

- In the Terminal, use the cd command to navigate to the OpenCat/pyUI/ folder. You can use the Tab key to auto-complete the path name.
- After entering the pyUI/ folder, enter ls and ensure you can see the UI.py and other python source codes listed.
- Enter python3 UI.py.

8.14.10 Problem Uploading on ESP32 Camera

There are few things to check:

- Correct hardware board selected in **Tools** menu
- GPIO 0 is set to GND when uploading. Reset once after doing that to enable download mode.
- Hardware is 5V compatible. Try switching to 5V input from FTDI programmer.
- Brownout detector was triggered? Include these header files:

```
#include "soc/soc.h" //disable brownout problems
#include "soc/rtc_cntl_reg.h"  //disable brownout problems
```

Include this inside **setup()**

```
WRITE_PERI_REG(RTC_CNTL_BROWN_OUT_REG, 0); //disable brownout detector
```

If you have other issues, check the following link (https://randomnerdtutorials.com/esp32-cam-troubleshooting-guide/)

You can also use Arduino UNO to upload code to your ESp32 CAM module.

Connect the following pins:

ESP32 CAM	Arduino UNO
5V	5V
GND	GND
UnT	Tx (PIN 1)
UnR	Rx (PIN 0)
IO0	GND (when uploading)

Steps to use sample code:

- Upload BareMinimun code to your UNO.
- Select the following settings in **Tools** menu.

Parameters	**Settings**
Board	ESP32 Wrover Module
Upload Speed	115200
Flash Frequency	40 Mhz
Partition Scheme	Huge APP

- Now you can upload your code to your esp32cam.

CHAPTER

NINE

DER ROBOTERHUND BITTLE

In diesem Kurs geht es darum, einen Roboterhund mit individuellen Fähigkeiten zu entwickeln.

9.1 Learning Outcome

9.1.1 Einführung

Menschen haben Roboter erfunden, damit sie sehr langweilige oder schwere Arbeit nicht mehr selbst erledigen müssen. Ein Roboter ist in der Lage komplexe Aktionen automatisch udn sehr genau auszuführen. Roboter können von einem externen Steuergerät wie z.B. einem Joystick () geführt werden oder die Steuerung befindet sich innerhalb des Roboters wie das Gehirn bei einem Menschen().

In diesem Kurs geht es unsere Roboterhunde “Bittle” und ihre Erziehung. Mit deiner Programmierung können sie Aktionen ausführen oder Kunststücke machen.

9.1.2 Was du dafür brauchst

Hardware

- Bittle Roboterhund

Software

- Codecraft (https://ide.tinkergen.com/)

9.2 Einführung

Schaut euch zunächst die verschiedenen Arten von Robotern an:

https://youtu.be/XN_nPC4mD8k

Wenn der Link nicht funktioniert, bitte hier klicken: https://youtu.be/XN_nPC4mD8k

Coole Roboter, oder? Auch die Programmierung ist nicht so schwierig. Man benötigt nur etwas Übung und Geduld .

Todo: Diskutiere die folgenden Fragen mit deinem Nachbarn: 1) Wo werden Roboter eingesetzt? 2) Wie bewegen sich die Roboter im dem dargestellten Video? 3) Hast du schonmal einen 4-beinigen Roboter wie eine Katze oder einen Hund gesehen? 4) Wofür könnte man einen 4-beinigen Roboter nutzen?

Jetzt machen wir eine kurze Videopause . Das Video wurde an der DARPA Robotics Challenge (DRC) gemacht. Bei der Challenge sollten Roboter Rettungsaktionen ausführen. Ohne die richtige Programmierung ist für einen Roboter nichts selbstverständlich. Der Roboter kann nicht einmal von selbst das Gleichgewicht halten oder etwas greifen. Aber überzeuge dich selbst..

https://youtu.be/g0TaYhjpOfo

Wenn der obige Link nicht funktioniert, bitte hier klicken: https://youtu.be/g0TaYhjpOfo

9.3 Grundlagen von Robotern und Programmierung

Jetzt kommen wir zu Bittle, unserem Roboterhund. In diesem Teil erhälst du Informationen über den Hund und lernst, wie ein Programmierer zu denken. Bittle sieht aus und funktioniert wie ein echter Hund.

Fig. 9.1: Hi, ich bin Bittle.

Bittle ist zu vielen Dingen fähig, sieh dir die kurze Demo an.

https://youtu.be/jJhdiQQT1o41

Wenn der obige Link nicht funktioniert, klicken Sie hier: https://youtu.be/jJhdiQQT1o4

Um solche Roboter herzustellen, benötigt man einige wichtige Komponenten:

- **Control Board** -> Dies ist das Gehirn des Roboters. Du gibst dem Board mit deiner Programmierung Anweisungen und es wird diesen Anweisungen folgen.

- **Motoren** -> Mit den Motoren kann sich Bittle bewegen. Sie wandeln elektrische Energie (Strom) in mechanische Energie (Drehmoment) um. Sie sind in verschiedenen Größen erhältlich.
- **Batterie** -> Dies ist die Energiequelle für Roboter. Dadurch kann sich Bittle frei bewegen, ohne dass er an eine Steckdose angeschlossen werden muss.

Der digitale Zwilling des Roboterhundes spielt auch bei der Herstellung des Hundes eine Rolle. Der digitale Zwilling ist die digitale Darstellung des Hundes und beschreibt alle seine Komponenten.

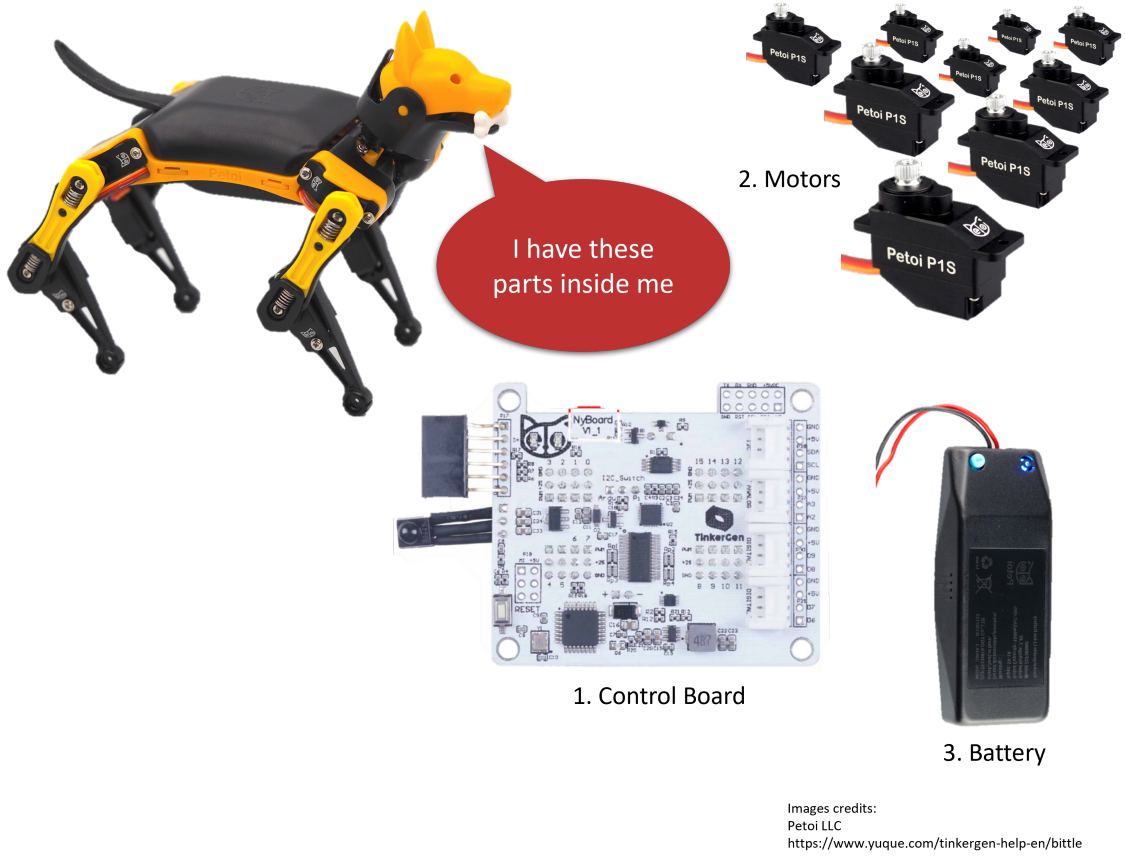


Fig. 9.2: Das sind meine Bestandteile!

Versuche einmal Bittle zu sagen, dass er sich setzen soll. Hat es etwas bewirkt?

Wahrscheinlich nicht. Derzeit versteht Bittle die Anweisungen nicht. Du musst dem Hund beibringen, was bei dem Befehl “Sitz!” machen soll. Wenn du Computern oder Robotern Anweisungen gibst, die sie verstehen und ausführen, wird das Programmieren genannt.

Ein Programm ist eine Liste von Anweisungen, die dann von dem Roboter ausgeführt werden. Oft ist es schwierig die richtige Reihenfolge der Anweisungen zu erkennen. Man verwendet dann Flussdiagramme, um den Prozess zu planen. Ein Flussdiagramm zeigt einen Entscheidungsprozess, der ein bisschen wie ein Code aufgebaut ist. Wenn du zum Beispiel eine Tasse Tee zubereiten möchtest, dann könntest du ein Flussdiagramm wie dieses erstellen.

Die Bedeutung der Symbole ist unten angegeben:

Ein weiteres Beispiel für ein Flussdiagramm kann wie folgt sein: Stell dir vor, du müsstest eine Straße überqueren und es gibt keine Ampel auf der Straße. Du musst also überprüfen, ob die Straße leer ist und ob du sie überqueren kannst. Wenn ein Auto in der Nähe der Kreuzung ist, dann wartest du solange, bis es wieder sicher aussieht. Dies wird Entscheidungsprozess genannt. Versuche, ein Flussdiagramm für die Situation mit der Straßenkreuzung zu zeichnen.

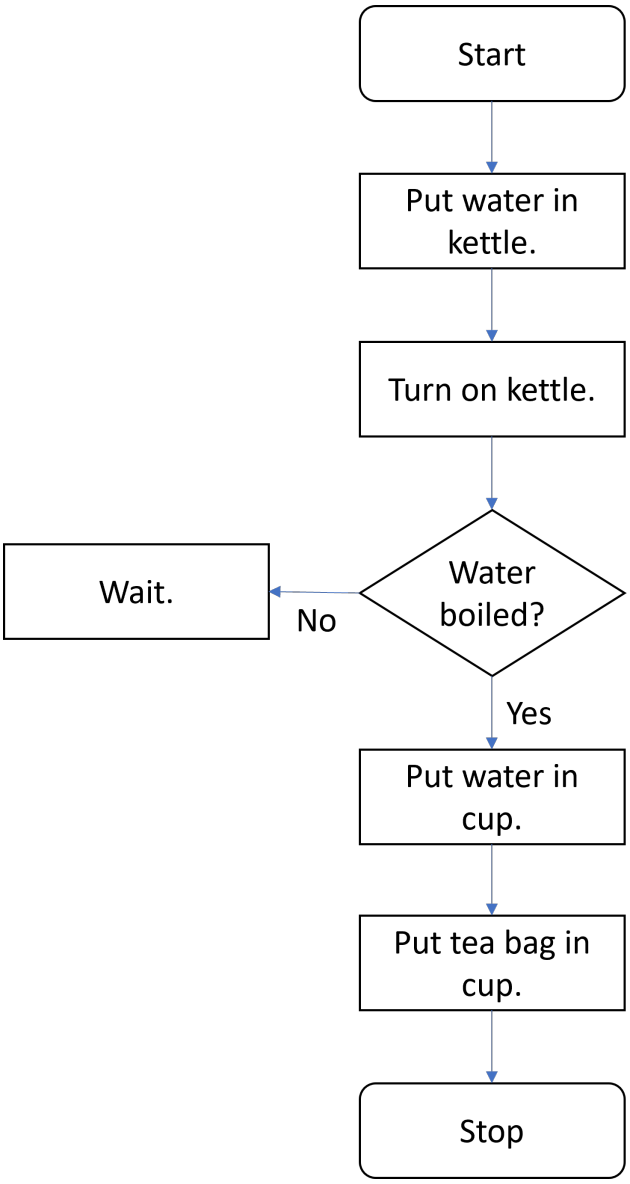


Fig. 9.3: Flow chart

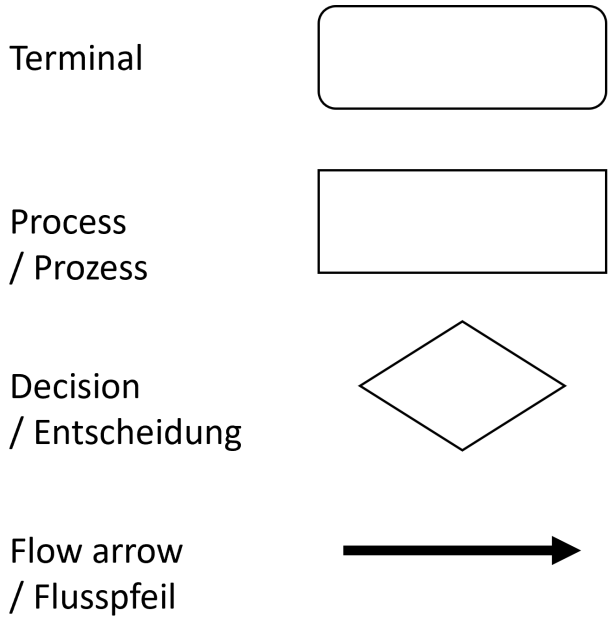


Fig. 9.4: Flow chart Symbole

Der Entscheidungsprozess spielt eine wichtige Rolle bei der Programmierung. Bevor du mit einem Projekt oder einer Programmieraufgabe beginnst, solltest du dir folgende Fragen stellen:

1. Was sind die Ziele?
2. Was weiß ich über das Projekt?
3. Was muss ich lernen, um diese Aufgabe zu erledigen?
4. Kann ich eine große Anweisung in kleine, einfache Anweisungen zerlegen?

9.4 Programmierung

In diesem Teil lernst du Bittle zu programmieren. Aufgeregt?

Um den Roboterhund zu programmieren, verwendest du die Codescraft-Software. Die Schnittstelle sieht so aus,

Zum Programmieren kannst du den Blockkatalog öffnen und einen Block in den Arbeitsbereich ziehen. Gehe die folgende Anleitung durch, damit du mit dem Programmieren beginen kannst.

9.4.1 Roboter mit PC verbinden

Um das Programm auf dem Roboter hochzuladen, musst du Bittle an einen PC anschließen. Du benötigst folgende Dinge:

- Roboterhund
- USB-Kabel
- USB-Anschluss für Roboter (USB-TTL-Konverter)

Folge den unteren Schritten:

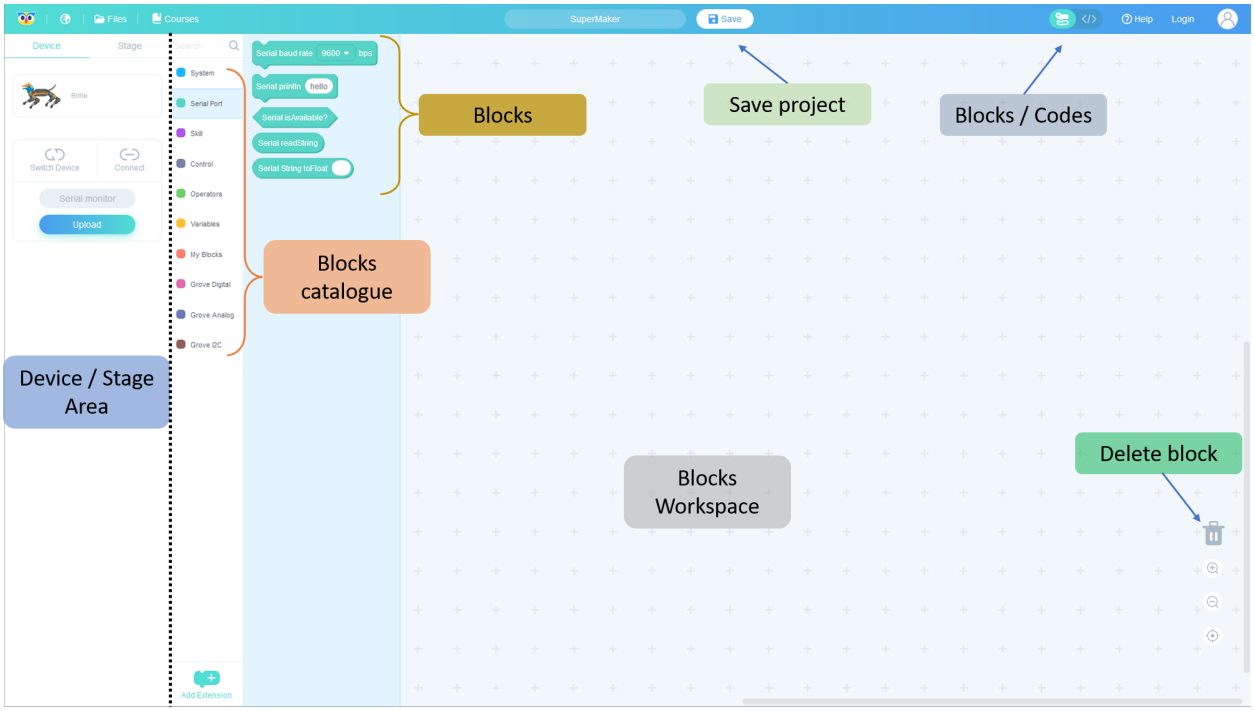


Fig. 9.5: Codecraft UI (Benutzeroberfläche)

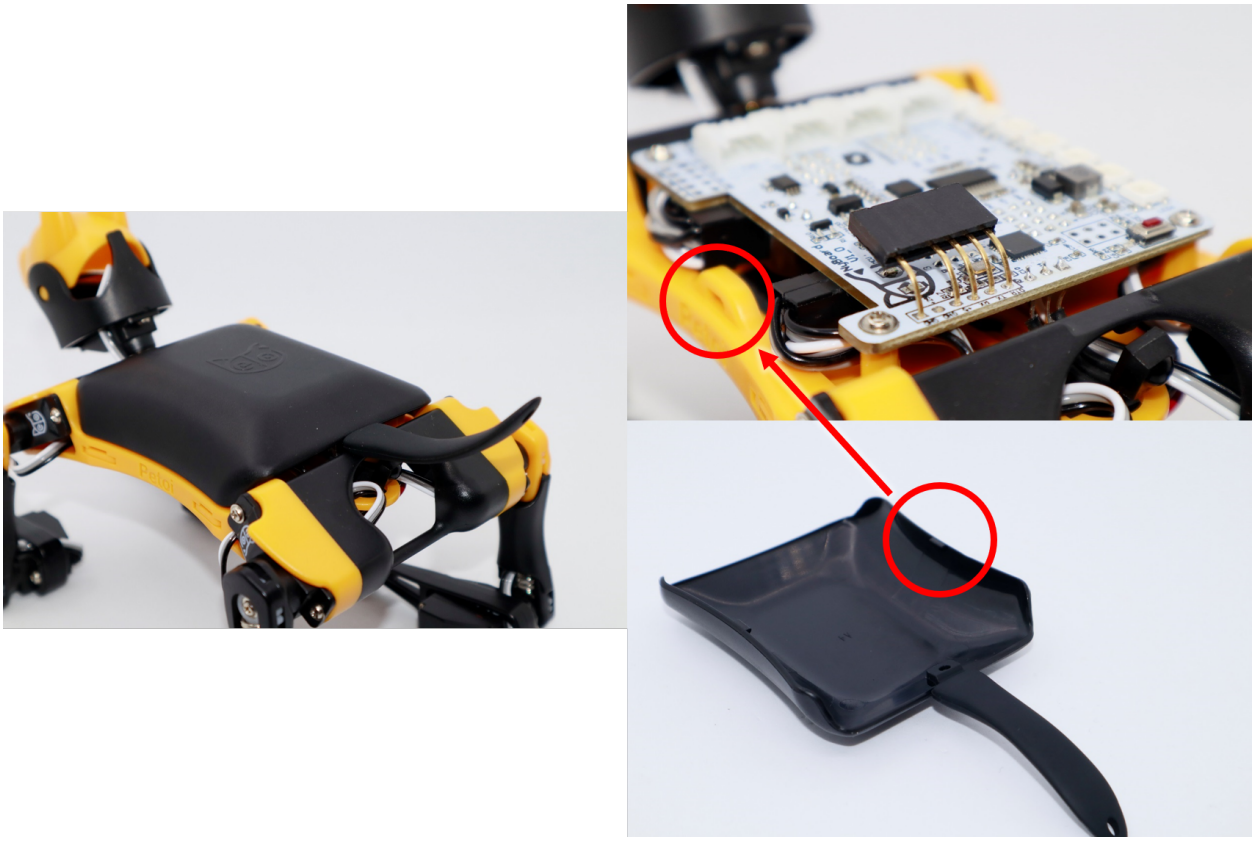


Fig. 9.6: Körper und Deckel.

1. Entferne die Abdeckung des Roboters.
2. Stecke die Micro-USB-Seite des Kabels in den USB-Stecker und verbinde den USB-Stecker mit dem Roboterhund.

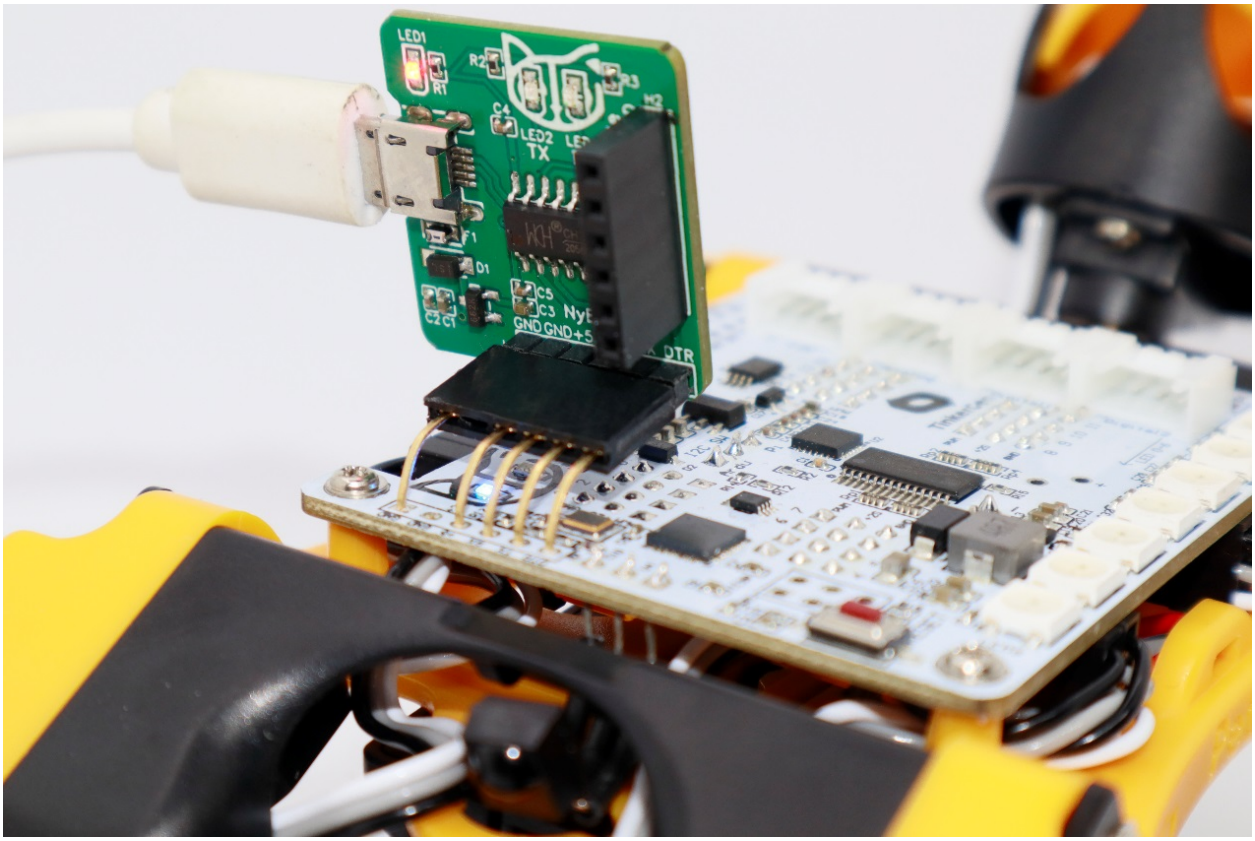


Fig. 9.7: USB Verbindung.

3. Verbinde das andere Ende des USB-Kabels mit dem PC.
4. Unter dem Hund befindet sich ein kleiner Knopf. Halte den Knopf einige Sekunden lang gedrückt, bis ein blaues Licht neben dem Knopf leuchtet. So wird der Roboter eingeschaltet.
5. Öffne die Codecraft-Software über die Taskleiste.
6. Wähle Bittle in der Hardwareauswahl aus.
7. Klicke auf der linken Seite auf die Schaltfläche **Verbinden**. Wähle dein Gerät aus der Liste aus und bestätige mit OK.
8. Du siehst ein Pop-Up-Fenster, das die Verbindung bestätigt. Wenn nicht, dann wende dich an einen Betreuer.

Sehr gut, der Hund ist mit dem Programm verbunden! Lies dir noch kurz eine Einleitung zu den Blöcken durch, dann kannst du mit dem Programmieren beginnen.

Fig. 9.8: Batterietaste ein- und ausschalten

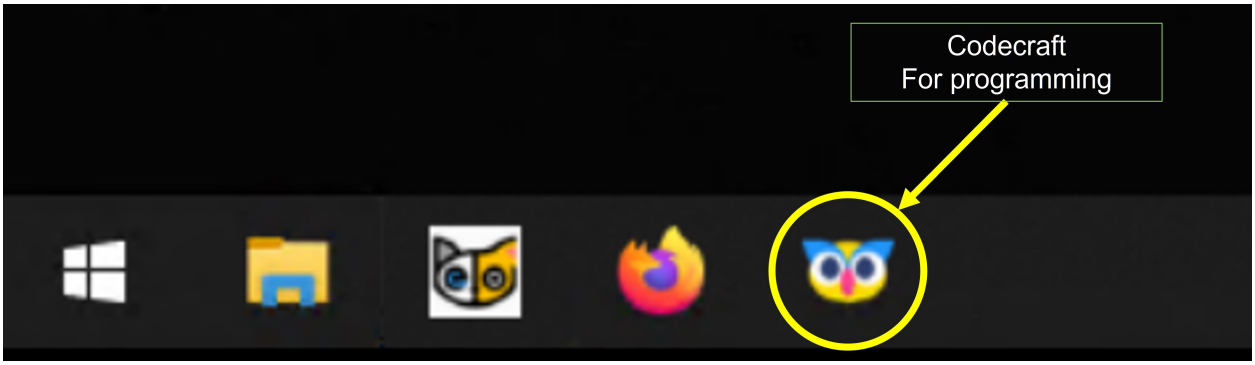


Fig. 9.9: Codecraft icon (Symbol)

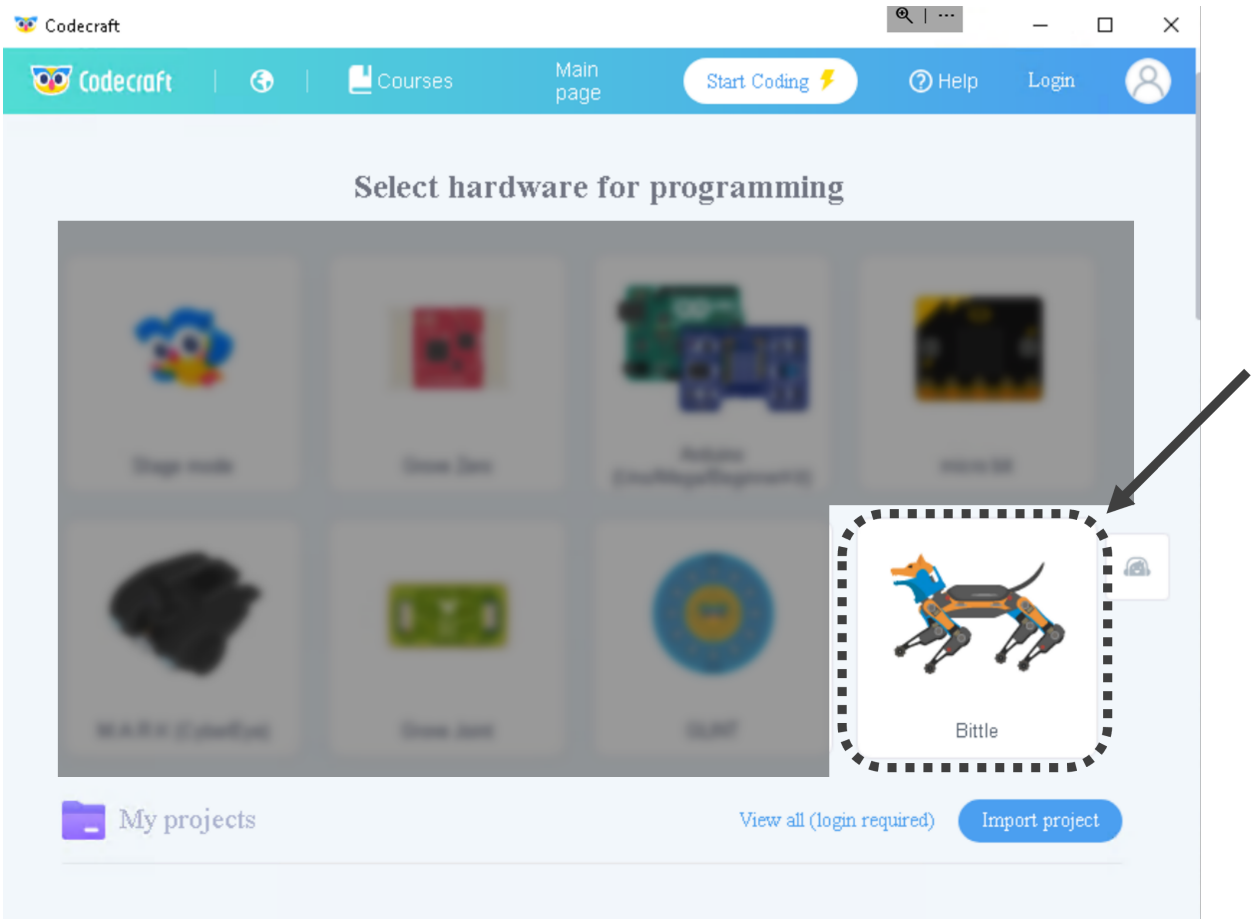


Fig. 9.10: Bittle icon im Codecraft

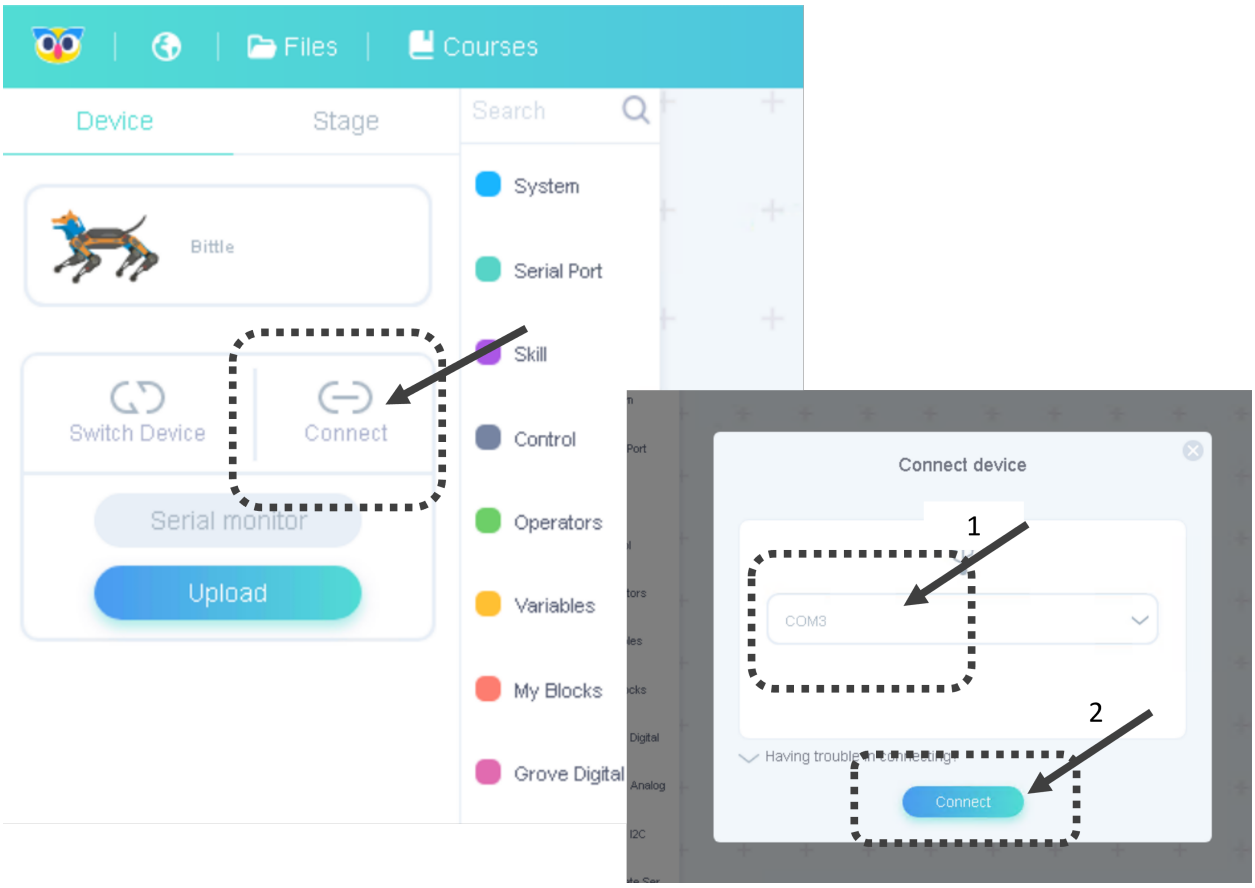


Fig. 9.11: Verbindungstest.

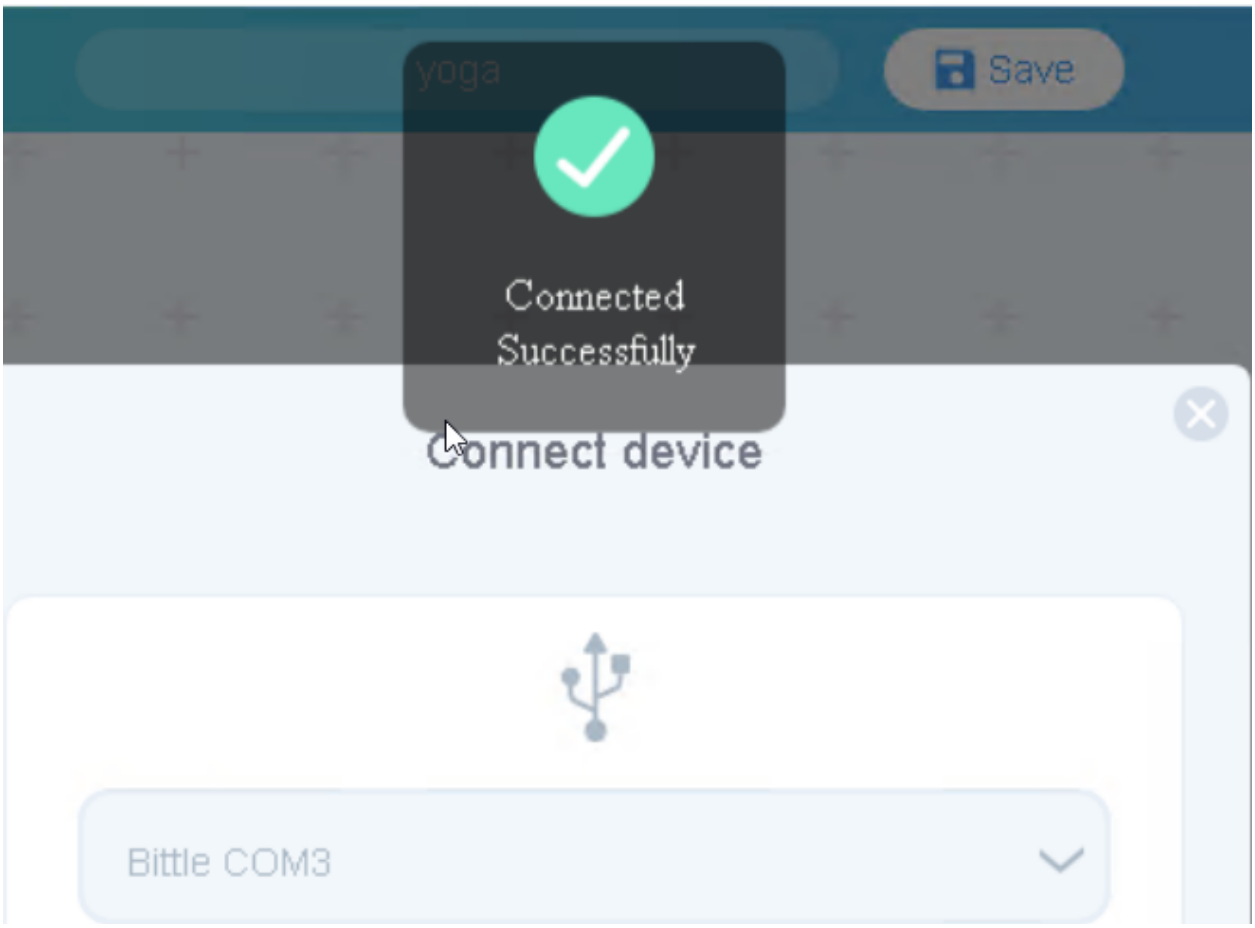


Fig. 9.12: Verbindung erfolgreich.

9.4.2 Programmierung mit Bausteinen:

Um den Roboter zu programmieren werden Blöcke verwendet. Blöcke haben verschiedene Formen und Farben. Hinter den Blöcken sind einzelne Code-Schnipsel. Der PC verwendet ein anderes Programm, um die einzelnen Schnipsel zu einem Programm zu fügen und auf den Roboter hochzuladen. Die Blöcke sind also nur eine Hilfestellung um dir das Programmieren zu erleichtern.

Blockformen

Es gibt fünf verschiedene Blockformen in Codecraft:

- Kopfblock (Hat Block)
- Stapelblock (Stack Block)
- Boolescher Block (Boolean Block)
- Reporter-Block (Reporter Block)
- C-Block

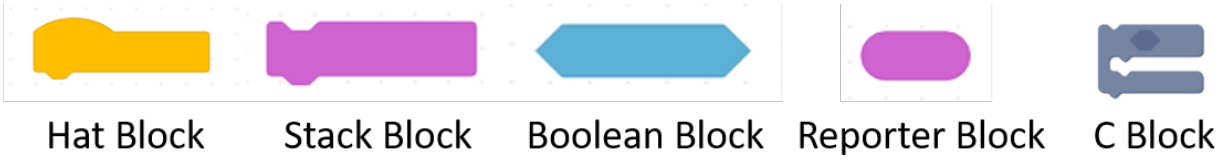


Fig. 9.13: Codecraft-Blockanleitung.

Beschreibung:

1. Kopfblock: Dies sind die Blöcke, die jedes Skript starten. Der Rest der Blöcke sollte am Kopfblock befestigt werden. Bitte beachte, dass verschiedene Kopfblöcke jeden Programmcode auf unterschiedliche Weise starten.
2. Stapelblock: Diese Blöcke führen die Hauptbefehle aus. Sie sind oben mit einer Kerbe und unten mit einer Erhebung geformt, damit viele Stapelblöcke aufeinander gestapelt werden können.
3. Boolescher Block: Sie können in entsprechenden sechseckigen Löchern platziert werden und können nicht ohne einen passenden anderen Block verwendet werden. Mit Boolean ist eine Bedingung (wahr oder falsch) gemeint. Erinnere dich an das Beispiel mit der Straßenkreuzung. Hier wurden auch boolsche Bedingungen verwendet, um eine Entscheidung zu finden.
4. Reporter-Block: Reporter-Blöcke können überall dort platziert werden, wo Daten benötigt werden. Du kannst sie wie die Booleschen Blöcke nicht einzeln verwenden. Wenn es einen Steckplatz gibt, der einem Reporterblock entspricht, kann er nach Bedarf platziert werden. Reporter-Blöcke können Zahlen und Zeichenfolgen enthalten.
5. C-Block: Diese Blöcke haben die Form eines „C“. Diese Blöcke wiederholen die Anweisungen innerhalb des Cs solange, wie ihre Bedingung wahr ist. Die Bedingung kann am Anfang oder an Ende des Blocks stehen und wird von dir festgelegt. Du könntest dem Hund zum Beispiel mit dem C-Block die Anweisung geben: “Laufe solange gerade aus, bis ich eine Taste auf der Fernbedienung drücke.”

Ein letzter Abschnitt, dann kannst du endlich mit Bittle spielen.

9.4.3 Katalog

Es gibt etwa 10 Kataloge, in denen die Blöcke nach ihrer Anwendung unterteilt sind. Die wichtigsten Kataloge sind:

- System: Enthält Blöcke wie Setup & Loop, Buzzer-Blöcke und andere integrierte Hardware-Blöcke. Du brauchst diese Blöcke zum Starten des Programms.
- Skill: Enthält Blöcke mit vordefinierten Fertigkeiten wie Gehen, Sitzen, Liegestütze und vieles mehr.
- Control: Enthält Blöcke, die bei der Entscheidungsfindung helfen. Dies kann ein if-else-Block, ein C-Block oder ein Programmendblock sein.

Du kannst auch gerne die anderen Kategorien erkunden.

Es gibt einen wichtigen Block. Es ist ein spezieller Block, der nur einmal in einem Programm verwendet wird.

Es heißt Setup & Loop Block. Du findest ihn im Systemkatalog. Dieser Block besteht wie im Namen aus 2 Teilen, Setup und Loop.

- Der Setup-Teil führt die darin enthaltenen Blöcke nur einmal aus. Dies kann verwendet werden, um Startmelodien abzuspielen. So kannst du dem Benutzer mitteilen, dass der Roboter gestartet ist.
- Der Loop-Teil führt die darin enthaltenen Blöcke wiederholt aus. Die Reihenfolge der Ausführung ist von oben nach unten.

Wenn du mehrere Setup & Loop Blöcke in deinem Programm hast, dann funktioniert dein Programm nicht wie gewünscht.

Jetzt kannst du mit dem Programmieren beginnen. Hier sind ein paar Aufgaben.

9.5 Tasks

9.5.1 Aufgabe 1 - Erste Tests

Teil 1

1. Füge einen Setup & Loop-Block hinzu.
2. Füge in Setup eine Buzzer-Melodie (Der Block heißt: “Buzzer play melody”) hinzu.
3. Klicke auf der linken Seite auf die hochladen (Upload). Warte bis der Upload abgeschlossen ist.

Teil 2

1. Verschiebe nun die Buzzer-Melodie von Setup auf Loop.
2. Klicke auf hochladen (Upload) und warte, bis der Upload abgeschlossen ist.

Was beobachtest du? Was verändert sich wenn man den Melodie-Block von Setup in Loop verschiebt?

Note: Erklärung:

Wenn ein Block im Setup ist, dann läuft er nur einmal zu Beginn eines Programms ab. Wenn ein Block in einer Schleife (Loop) ist, dann wird er immer und immer wieder ausgeführt (bis der Akku leer ist).

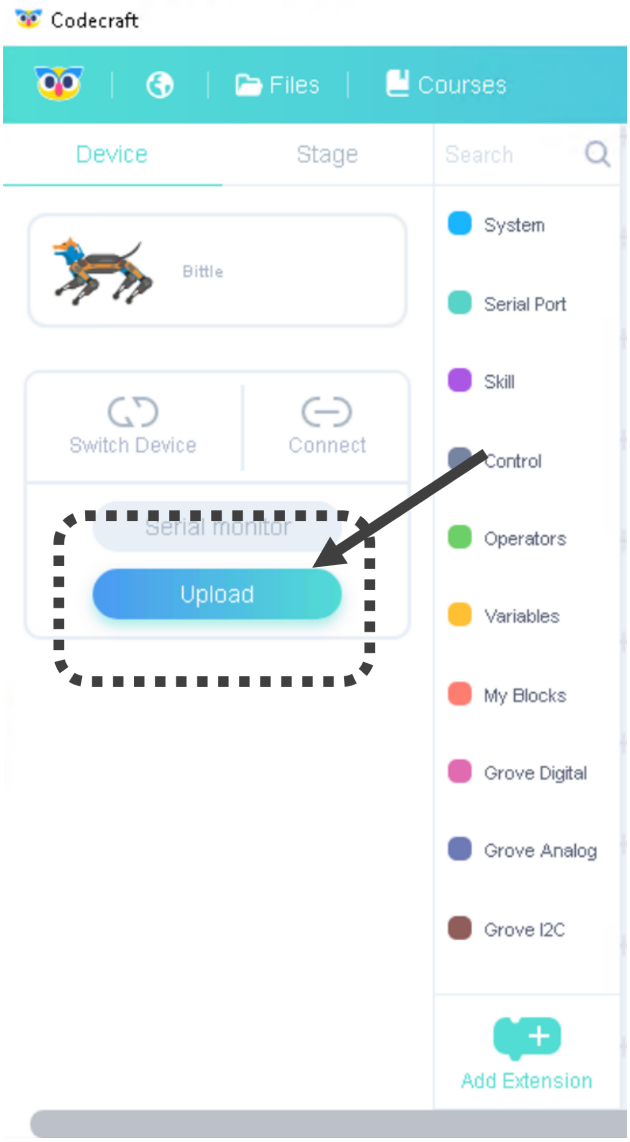


Fig. 9.14: Codecraft Upload.

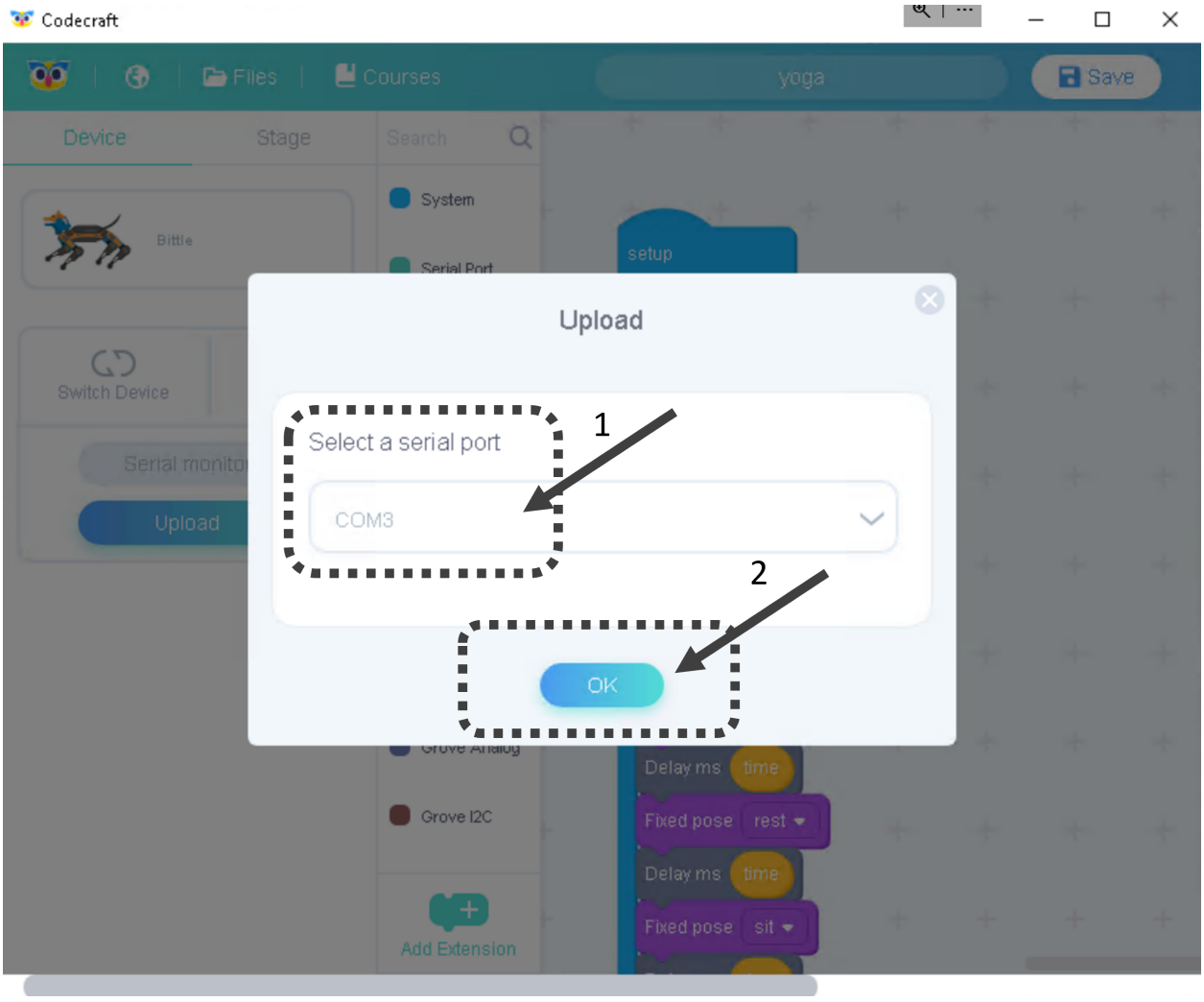


Fig. 9.15: Codecraft Upload continued.

9.5.2 Aufgabe 2 - Lassen Sie den Hund laufen

1. Entferne die Buzzer-Melodie, wenn du sie nicht mehr hören möchtest.
2. Füge den Walk-Block aus dem Skill-Katalog innerhalb der Schleife (Loop) hinzu.
3. Füge den Bewegungsblock (Execute the movement) nach dieser Schleife (Loop) aus. Dieser Block ist wichtig, da mit nur mit ihm eine Bewegung ausgeführt wird.
4. Klicke auf hochladen (Upload) und warte, bis der Upload abgeschlossen ist.

Warning: Bittle sollte jetzt anfangen zu laufen. Seie vorsichtig, da er immer noch mit dem PC verbunden ist. Entferne den USB-Anschluss, um mit ihm spazieren zu gehen.

Note: Erklärung:

In der obigen Aufgabe hast du den Befehl Walk (Gehen) und dann die Bewegung ausführen (Execute the movement) verwendet. Der Befehl Walk wird nur zur Vorbereitung für die Aktion gebraucht. Ausgelöst wird die Bewegung mit dem Bewegung-ausführen-Block (Execute the movement).

9.5.3 Aufgabe 3 - Steuere den Hund mit der Fernbedienung.

1. Füge die Blöcke wie im Bild ein und verbinde sie passend.
2. Klicke auf hochladen (Upload) und warte, bis der Upload abgeschlossen ist.

Zunächst beginnt Bittle zu laufen. Drücke nun die obere linke Taste auf der Fernbedienung.

Was siehst du? Hält der Roboter an?

Note: Erklärung:

In dieser Aufgabe hast du einen if-Block verwendet. Der If-Block ist eine Anweisung mit einer Bedingung, die vorher abgefragt wird. Wenn eine bestimmte Bedingung erfüllt ist, wird dieser Teil des Codes ausgeführt. Der Code wird übersprungen, wenn die Bedingung nicht erfüllt ist. Der If-Block ähnelt den Überlegungen aus deinem täglichen Leben. Zum Beispiel:

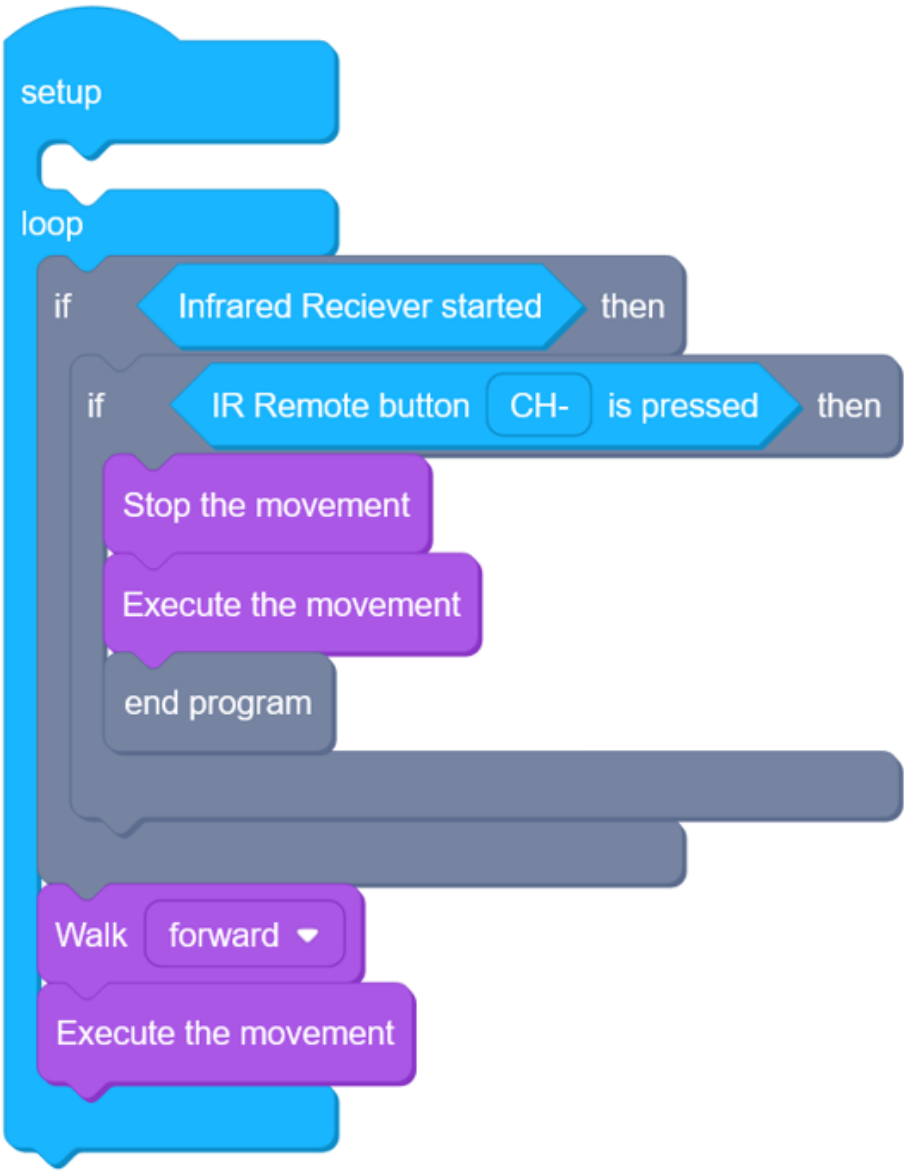


Fig. 9.16: Code für die Fernbedienung.

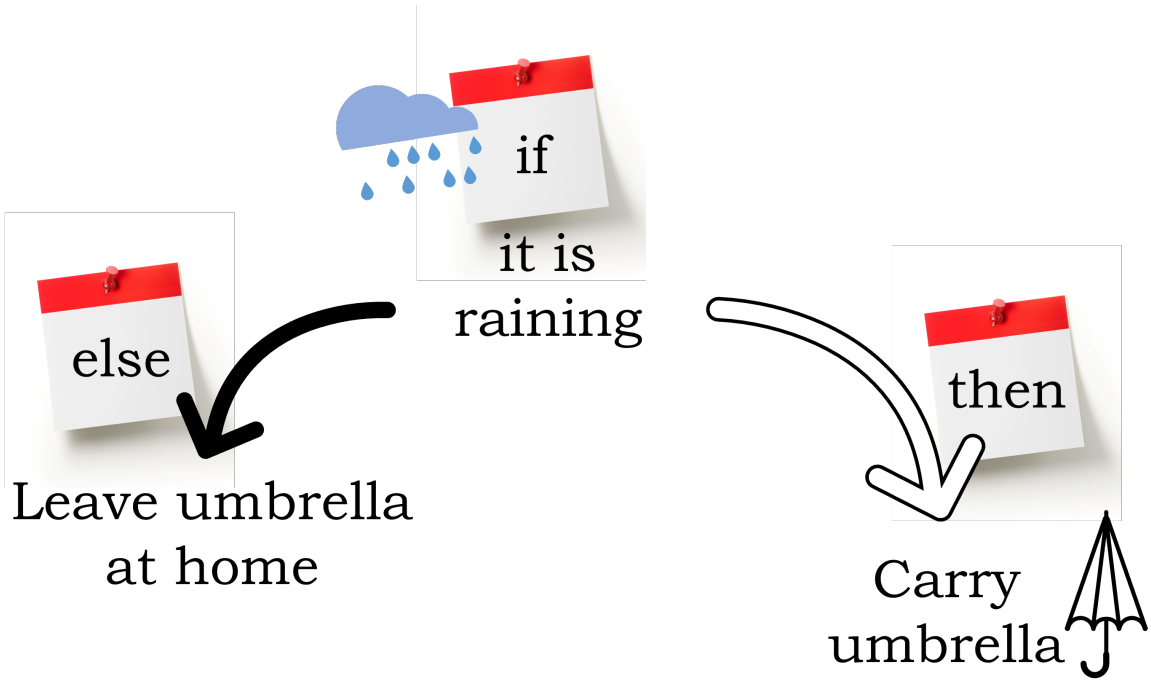


Fig. 9.17: If/Else mit Regen.

Hier hast du den if-Block verwendet, um die Verbindung zwischen Roboter und Fernbedienung zu überprüfen. Wenn Bittle das Signal empfängt, dann ist die Bedingung des ersten If-Blocks korrekt und die Bedingung des zweiten If-Blocks wird überprüft. Jetzt wird geprüft, welche Taste gedrückt wurde. Wenn du die von dir programmierte Taste gedrückt hast, dann ist auch die zweite Bedingung korrekt und die Anweisung des zweiten If-Blocks wird ausgeführt. Wenn du eine andere zufällige Taste gedrückt hast, dann wird der if-Block übersprungen und der Rest des Codes läuft weiter.

Wow, du bist schon fertig und hast das Ende der heutigen Reise erreicht.

Schon fertig und noch Zeit?

Es gibt eine versteckte Seite für Sie. *Klicke hier für die Zusatzaufgaben.*

9.6 Weitere Informationen zu Bittle

Möchtest du mehr über den Roboterhund erfahren? Diese Websites können dir helfen:

Grundlagen / Ich habe gerade erst angefangen:

- https://scratch.mit.edu/
- https://microbit.org/get-started/first-steps/introduction/

Mittelstufe / Ich weiß etwas:

- https://docs.arduino.cc/learn/starting-guide/getting-started-arduino
- https://docs.micropython.org/en/latest/

- https://i40.fh-aachen.de/courses/dta/activities/bittle/index.html

Fortgeschritten / Bring it on

- https://www.hackster.io/
- https://www.instructables.com/circuits/
- https://www.tinkercad.com/projects

Scanne diesen QR-Code und erhalte alle Links direkt aufs Handy.

Fig. 9.18: QR Code.

Wenn du Projektvorschläge oder andere Ideen hast, dann schreibe uns gerne eine Mail!

9.7 Additional Tasks

9.7.1 Aufgabe 4 - Programmiere eine Fernbedienung, um die Bewegung des Hundes zu steuern.

1. Erstelle einen Code für die Fernsteuerung wie im Bild.

   Wenn du also die Vorwärts-Taste drückst, dann muss der Hund vorwärts gehen. Wenn du die Links-Taste drückst, dann muss Bittle nach links gehen. Am besten verwendest du if-Blöcke ähnlich wie in der vorherigen Aufgabe. Hinweis: Du musst den if-Block mit „Infrared Receiver started“ nur einmal hinzufügen.

2. 1. Klicke auf hochladen (Upload) und warte, bis der Upload abgeschlossen ist.

Hat alles geklappt? Gute Leistung!

Versuche dich an der nächsten Aufgabe.

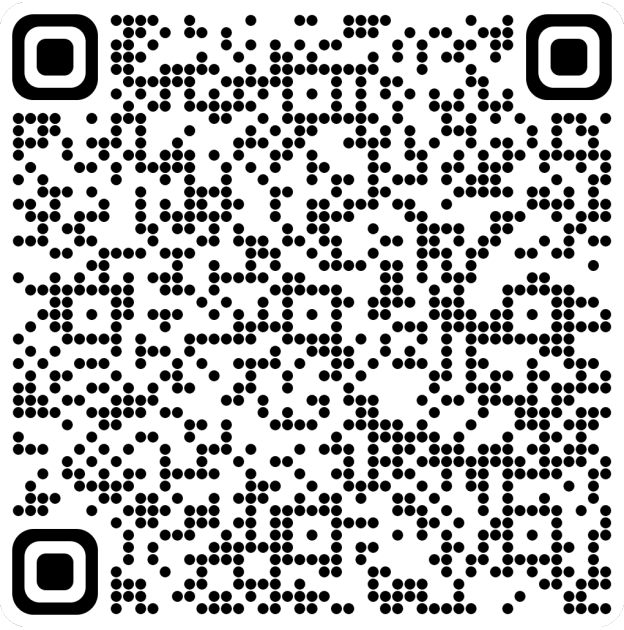


Fig. 9.19: QR Code.

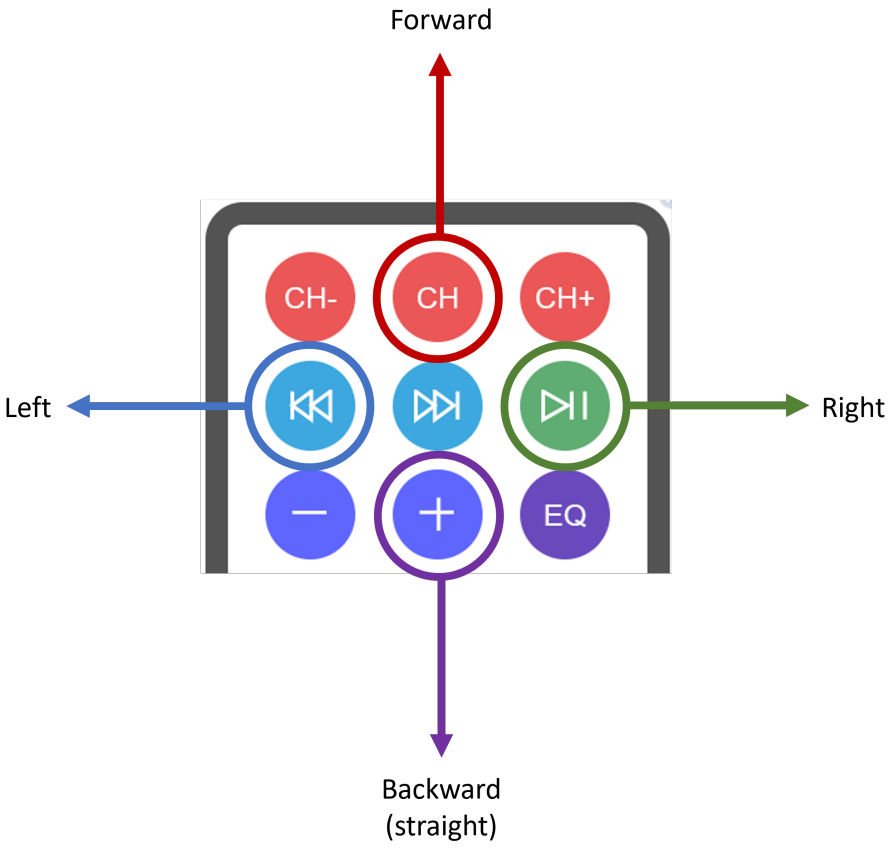


Fig. 9.20: Fernbedienung.

9.7.2 Aufgabe 5 - Roboterübung

Schaue dir das Video unten an und hilf dem Roboter, einige Übungen zu machen.

https://youtu.be/YGm_7YcTz1U

Wenn der obige Link nicht funktioniert, klicke hier: https://youtu.be/YGm_7YcTz1U

Hint: Du kannst verschiedene fixed poses (Programmblöcke für einzelne Kunststücke) verwenden, damit die Hunde die Fitnessübungen machen. An dieser Stelle kannst du auch kreativ werden und dir eigene Fitnessübungen für Bittle überlegen.

9.7.3 Aufgabe 6 - Digitaler Zwilling

Ein digitaler Zwilling (oder "digital twin" in englisch) ist eine digitale Nachbildung (zum Beispiel von einer Maschine). Dabei werden alle Daten über die Maschine wie die Betriebsanleitung, Sensordaten oder Programme gesammelt und verwaltet. Die Daten sind nicht nur aus der Anwendung des Bittle. Es können auch Daten aus der Konstruktion sein. Und sogar die Daten, die beim Recycling eines Bittle generiert werden, können im digitalen Zwilling gespeichert werden.

Was für ein Zweck hat ein digitaler Zwilling? Überlegt zusammen nach möglichen Vorteilen für ein Unternehmen, wenn es von einer Maschine einen digitalen Zwilling anfertigt.

Das ist alles, was wir für heute haben.

Kommen wir zum letzten Abschnitt. *Klicke hier.*

CHAPTER

TEN

ARDUINO BASICS

10.1 Introduction

Arduino is an open-source electronics platform based on easy-to-use hardware and software. Arduino boards are able to read inputs (e.g. a finger on a button or a Twitter message) and can turn them into an output (e.g. activating a motor, or publishing something online). You can tell your board what to do by sending a set of instructions to the microcontroller on the board.

All Arduino boards are completely open-source, empowering users to build them independently and eventually adapt them to their particular needs. The software, too, is open-source, and it is growing through the contributions of users worldwide.

Thanks to its simple and accessible user experience, Arduino has been used in thousands of different projects and applications. The Arduino software is easy-to-use for beginners, yet flexible enough for advanced users. It runs on Mac, Windows, and Linux. *(Source: https://www.arduino.cc/en/Guide/Introduction)*

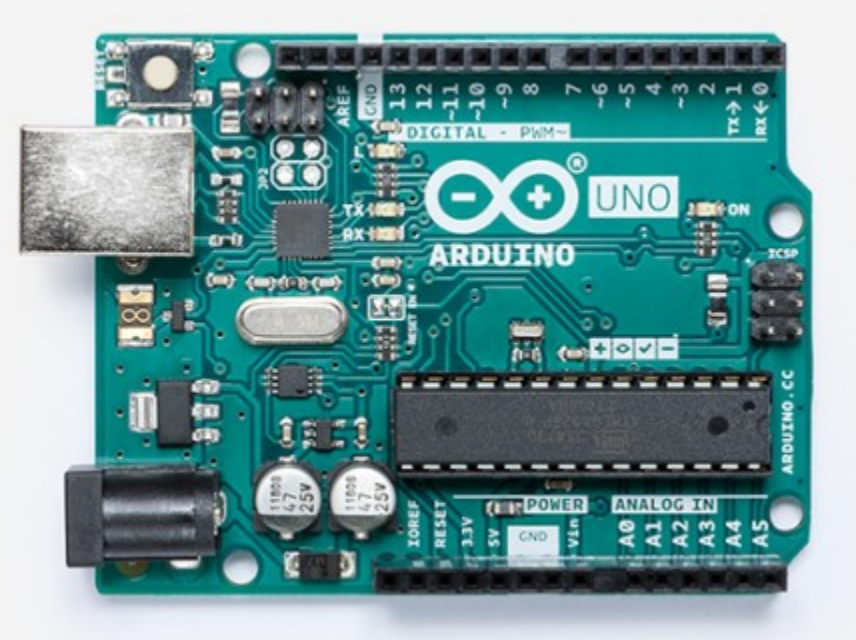


Fig. 10.1: Arduino Uno *(Source: https://store.arduino.cc/arduino-uno-rev3)*

The following course provides you with basic understandings of microcontrollers using Arduino as application example. Furthermore, a short introduction in C programming is provided.

10.2 Embedded Systems

Embedded systems are computer systems in which a processor is embedded in a technical device. The processor takes over all monitoring, control and operating functions. In general, the processor of an embedded system is a **microcontroller**. In contrast to PC processors the memory as well as inputs and outputs are usually integrated in a single chip. This is a processor with different peripheral interfaces.

Microcontroller can be separated by the number of bits of the internal data bus (4 bit, 8 bit, 16 bit, 32 bit, etc.). This number describes the length of data which can be processed by the microcontroller. The biggest in 8 bit representable number is 255, thus a 8 bit microcontroller can only add numbers less or equal 255 in one step. Adding higher numbers results in a longer calculation time as several commands are necessary.

Embedded systems are not always required to have a user interface. However, they can have I/O interfaces to handle analog or digital data.

Characteristics:

- Application specific customized hardware and software
- Compact devices which execute one or more specific tasks
- Can work without or with a reduced operating system (depending on complexity)
- Special real-time operating systems available

Systems are optimized for (examples):

- Code-size efficiency
- Optimised program execution
- Reduced system costs
- Reduced energy consumption

10.2.1 Difference between Universal Computers and Embedded Systems:

Universal Computer	Embedded System
Optimized for high processing power and graphics performance	Optimized system solutions where the microcontroller is often not visible
High compatibility with application software	Provides comprehensive functions. Available with sensor and actuator interfaces
Uses a variety of programs for individual applications	Is optimized for a specific application
Computers are interchangeable, software is the capital good	Is often designed as a real-time system
Mostly uses standardised Operating Systems, e.g. MacOS	

10.2.2 Software Development

In conventional, so called "self-hosted" computers (like your Windows PC), the development system and the target system are identical. The development system in general has a highly specialized and highly integrated interface that includes all the tools of software development (e.g. Visual Studio).

In case of embedded systems, the operating system of the target can offer only basic functionalities like for example loader, debugger or scheduler. Therefore, the host system is different from the target system which makes configuration and debugging more complicated than for "self-hosted" computers. A **cross development** is necessary.

In this case, the programming and compiling is done using development tools on the host computer. Furthermore, a connection to the target system is established using a communication interface (e.g. ISP or Ethernet interfaces).

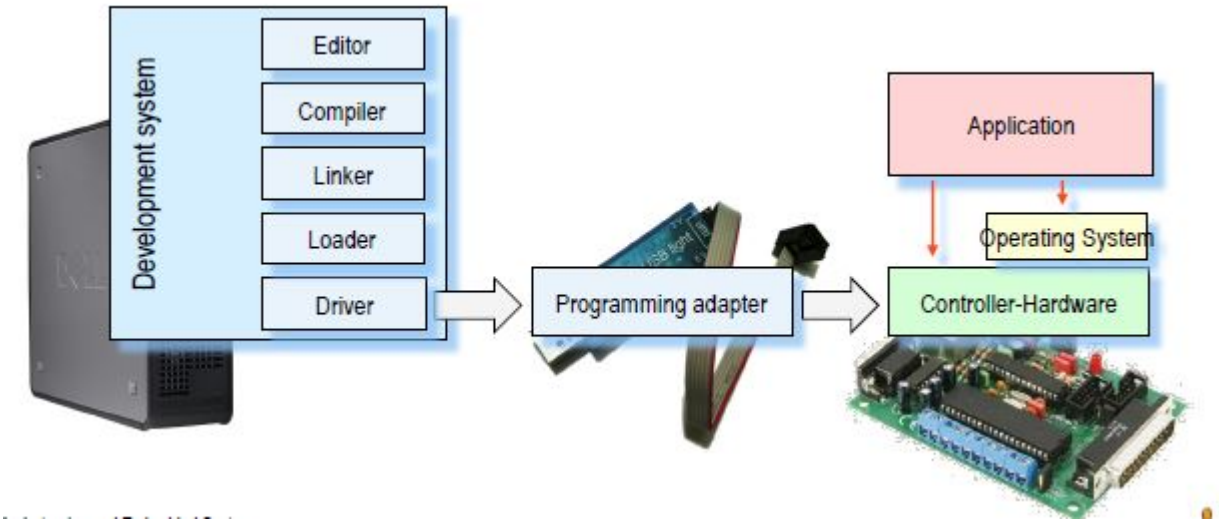


Fig. 10.2: Cross Development using a host computer and an embedded system

10.3 Programming in C

10.3.1 Variables

A variable is a name given to a storage area that programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

Standard data types for C programming (typical values):

Integer types:

Datatype	Size	Range
signed char	1 Byte	-128 ... 127
unsigned char	1 Byte	0 ... 255
unsigned short	2 Bytes	0 ... 65535
signed short	2 Bytes	-32768 ... 32767
unsigned int	4 Bytes	$0 \dots 2^{32} - 1$
signed int	4 Bytes	$-2^{31} \dots 2^{31} - 1$
unsigned long	8 Bytes	$0 \dots 2^{64} - 1$
signed long	8 Bytes	$-2^{63} \dots 2^{63} - 1$

Floating point types:

Datatype	Size	Range	Precision
float	4 Bytes	1.2E-38 ... 3.4E+38	6 decimal places
double	8 Bytes	2.3E-308 ... 1.7E+308	15 decimal places

Void type:

The void type specifies that no value is available. It is used in three kinds of situations:

- A function with no return value has the return type as void.
- A function with no parameter can accept a void.
- A pointer of type void * represents the address of an object, but not its type.

Variables can be defined like this (examples):

- `int i, j, k;`
- `char x = 'x';`

10.3.2 Operators

The C language has different types of operators:

- Arithmetic Operators, see below
- Relational Operators, see below
- Logical Operators, see below
- Bitwise Operators, see below
- Assignment Operators, e.g. =
- Misc Operators, e.g. `sizeof`

Arithmetic operators

Operator	Description	Example
+	Adding	`A + B = 30`
-	Subtracting	`A  B = -10`
*	Multiplication	`A * B = 200`
/	Division	`B / A = 2`
%	Modulus	`B % A = 0`
++	Increment value by one	`A++ = 11`
--	Decrement value by one	`A-- = 9`

Relational operators

Comparison	<, >, <=, >=
Equality	==
Inequality	!=

Logical operators

And:	&&
Or	||
Not	!=

Bitwise operators

Bitwise operators work on bits and perform bit-by-bit operation.

Binary And:	&
Binary Or	|
Binary XOR	^
Binary Left Shift	<<
Binary Right Shift	>>

10.3.3 Decision Making

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

- if statement: Consists of a boolean expression followed by one or more statements.
- if…else statement: An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.
- switch statement: A switch statement allows a variable to be tested for equality against a list of values.

10.3.4 Loops

Loops are used if a block of code must be executed several times. It is possible to use one or more loops inside any other loop.

- while loop: Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
- for loop: Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
- do…while loop: It is more like a while statement, except that it tests the condition at the end of the loop body.

Loop control statements change execution from its normal sequence.

- break statement: Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
- continue statement: Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.

- goto statement: Transfers control to the labelled statement.

10.3.5 Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is main(), and all the most trivial programs can define additional functions.

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{ body of the function }``
```

The return type specifies the data type of the value which is returned by the function, for example `int`. If no value is returned, the return type is `void`. The function name is the actual name of the function. The parameters act like a placeholder to pass values to the function. The function body contains the statements which define what the function does.

Example:

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;

    if (num1 > num2)
    {
        result = num1;
    }
    else
    {
        result = num2;
    }

    return result;
}
```

To use the function, it must be called in the program. Therefore, the required parameters and the function name must be passed. If the function returns a value it must be stored in a variable.

```
/* calling a function to get max value */
ret = max(a, b);
```

Note: Variables inside a function or a block which are called local variables. Variables outside of all functions are called global variables. Variables in the definition of function parameters are called formal parameters.

10.3.6 Arrays

Arrays can store a fixed-size sequential collection of elements of the same type.

Declaration: `type arrayName [ arraySize ];`; example: `double size[3];`

Initialization: `double size[3] = {10.0, 2.0, 3.4};`

In this case, the number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [ ].

If the array should just big enough to hold the initialization, an empty square bracket is used: `double size[] = {10.0, 2.0, 3.4, 9.4};`

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array: `timSize = size[2];`. This will take the third element from the array and assign it to the variable "timSize".

10.3.7 Pointers

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. It is used for example for dynamic memory allocation.

10.3.8 Strings

Strings are actually one-dimensional array of characters terminated by a null character '0'. Thus a null-terminated string contains the characters that comprise the string followed by a null.

Example: `char word[] = "Hello";`

10.3.9 Structures

Structures is a user defined data type available in C that allows to combine data items of different kinds. For example it might be helpful to track different information about a person like name, age and city.

Structures can be defined like this (structure tag is optional):

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
}[one or more struct variables];
```

Example:

```
struct person
{
    char name[50];
    int age[25];
    char city[100];
};
```

(continues on next page)

(continued from previous page)

```
/* Initialization */
struct person tim; /* Declare tim of type person */

/* Access member of structure */
timAge = tim.age;
```

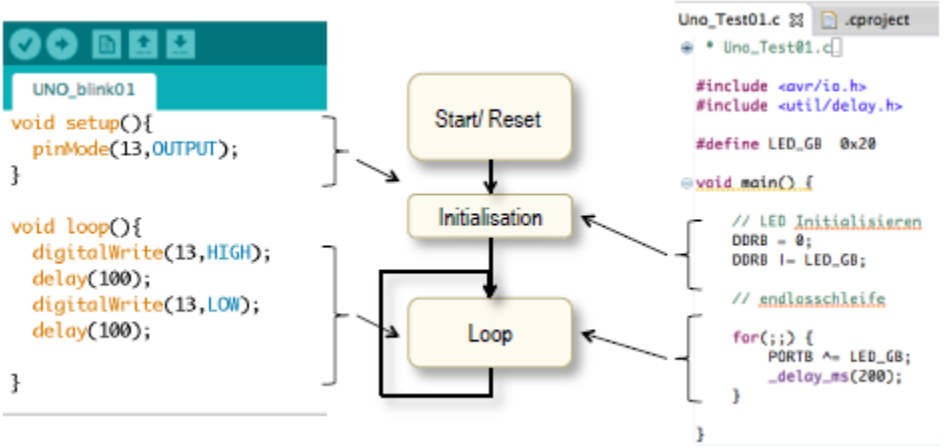
To access any member of a structure, we use the member access operator (`.`).

10.4 Arduino Programming

An Arduino-Program „Sketch“ consists of the „setup“ function for one time initialization during starting time and the „loop“ function for the endless loop.

10.4.1 Difference between Arduino and conventional C

The Arduino has a minimal Operating System. The program consists of an initialization part and an endless loop. As it can be seen in the example below, the Arduino development platform encapsulates many cryptic functions and the processor code. Additionally, the Arduino code is significantly larger in size than a code written in conventional C.



10.4.2 Arduino vs. Microcontroller Programming

Microcontrollers can be programmed in different languages. A very hardware-related, low-level language is Assembler. It is used to create code for a specific device/ a particular computer architecture and typically selected for time or memory critical tasks.

In contrast to Assembler Arduino has a minimal Operating System and library functions to interact with the Hardware. Thus, it is portable across multiple systems (the generated code can be used for different microcontrollers). Programming of C-Register is done with the help of the library functions. This leaves the programmer free to take care of his individual program.

10.4.3 Arduino basic commands

Read and Write:

- To set up the PIN in (INPUT,OUTPUT) mode: `pinMode(PIN,OUTPUT);`
- To write digital values: `digitalWrite(PIN,value);`
- To read digital values: `value=digitalRead(PIN);`
- To write analog values: `analogWrite(PIN,value);`
- To read analog values: `value=analogRead(PIN);`

Useful Arduino commands:

- Delay of the program execution in milliseconds: `delay(1000);`
- Start of serial communication: `Serial.begin(9600);`
- Send data via serial interface: `Serial.println(analogRead(1));`

Create own functions (example):

```
void blink(int thePin, int time)
{
    digitalWrite(thePin, HIGH);
    delay(time);
    digitalWrite(thePin, LOW);
    delay(time);
}

// Calling the function
blink(3, 1000);
```

Switch-case: Used to test if a variable is in a specific condition.

```
switch (myVariable)
{
    case1:
        command1;
    break;
    case2:
        command2;
    break;
    default:
        command3;
    break;
}
```

Note: For more information please refer to the Arduino page: https://www.arduino.cc/reference/en/

Note: For most steps Arduino examples are available in the Arduino IDE. Use these examples!

10.5 Getting started with Arduino IDE

The Arduino IDE is a really simple programming environment. Arduino projects are called “sketches”. Main features of Arduino IDE are:

10.5.1 Simple GUI with only a few Menus and Buttons

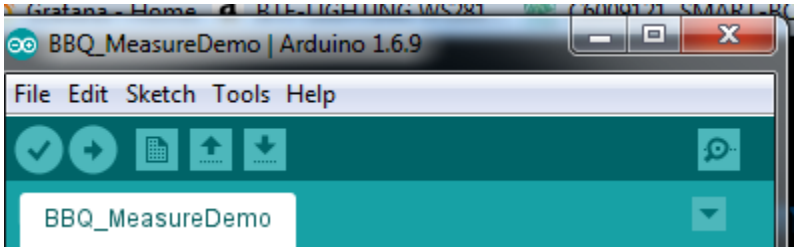


Fig. 10.3: Arduino GUI Overview

- The first button starts the compilation of your program.
- The second button compiles and downloads the program to the board.
- The next three buttons are New, Open and Save.
- On the right side is the button for the serial monitor.

10.5.2 Managed and unmanaged Libraries

The Arduino project provides a lot of libraries which are version managed. These libraries are available via the `Library Manager` in the Sketch menu.

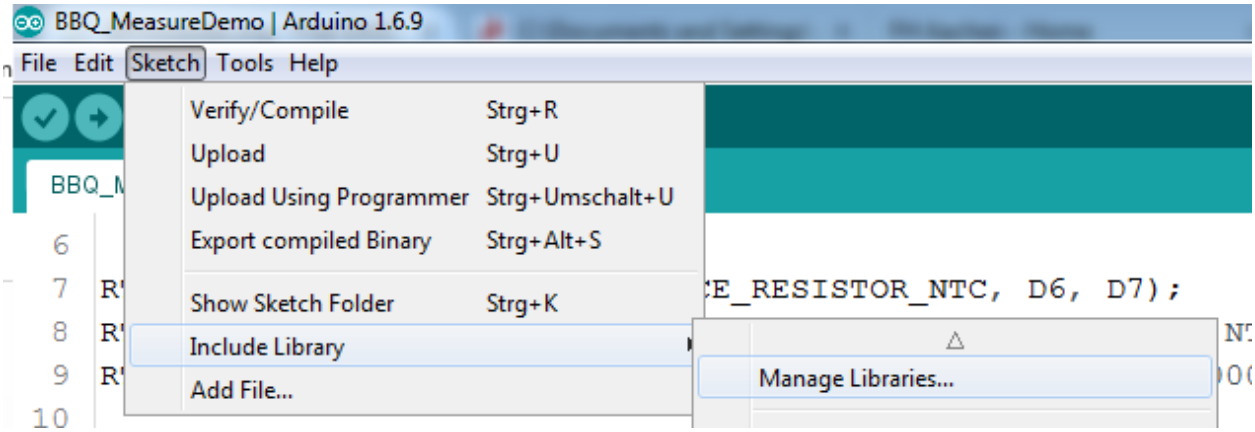


Fig. 10.4: Arduino Library Manager

With the Library Manager you can search for libraries and install them. Also library updates can be installed with this tool.

Unmanaged, own or third party libraries are supported as well. These can be placed in the user folder.

```
..\Documents\Arduino\libraries\
```

After adding a new library into this folder, a restart of the IDE is necessary.

10.5.3 Serial monitor and Serial plotter

With the serial monitor you have a direct access to the serial port. Via serial port you can exchange information between the developer and the program running on the board. It is mostly used for debugging software or exchanging data between a pc program and the Arduino board.

The serial plotter is a really simple tool which is able to plot a received number in a time series diagram. You can plot multiple values into one waveform, by adding a blank between the values. If you use a tabulator `\t` as separator the values are plotted in different waveforms.

10.5.4 Examples

Most Board Support Packages and Libraries will provide example source code that demonstrates how it should be used. All examples can be found in the `File` menu. You can often realize quickly and vertically prototypes by combining some examples.

10.5.5 Arduino Basic functions

Every Arduino program contains two basic functions:

```
void setup(){
}

void loop(){
}
```

The `setup()` function is called once in the beginning of the program start-up. Here you can call initialization and setup functions, which are needed in your program.

The `loop()` function will be called over and over again as fast as possible. In this part you will implement every functionality which will be done during the runtime of the board.

10.6 Problem Solving

This chapter should help you to fix problems which are occurring frequently.

10.6.1 Arduino

You cannot find the Node MCU?

Check if you can find the microcontroller in the `Device Manager` (press `Windows` and `X`) of your Laptop (COM Port).

Option 1: At the `Device Manager` look for a USB to Serial kind of device that seems to be broken (e.g., showing an exclamation mark; showing an USB2.0 device without specific name). Double click on it and head over to the Drivers tab. Click on Update Driver to search for updated driver software. Then, you should see the NodeMCU at your COM port.

Option 2: If you do not see the microcontroller you may need to install the drivers. You can download them here: https://github.com/nodemcu/nodemcu-devkit/tree/master/Drivers

Sources: *https://medium.com/@cilliemalan/installing-nodemcu-drivers-on-windows-d9bffdbad52*

Sources:

- https://learn.sparkfun.com/tutorials/i2c/all
- https://learn.adafruit.com/adafruit-mcp9808-precision-i2c-temperature-sensor-guide/overview
- https://roboindia.com/tutorials/ds18b20_temp_sensor_nodemcu
- https://create.arduino.cc/projecthub/TheGadgetBoy/ds18b20-digital-temperature-sensor-and-arduino-9cc806
- https://www.instructables.com/id/Calibration-of-DS18B20-Sensor-With-Arduino-UNO/
- https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/
- https://www.arduino-tutorial.de/arduino-und-mqtt/
- https://www.baldengineer.com/mqtt-introduction.html
- https://www.tutorialspoint.com/cprogramming/index.htm
- https://www.arduino-tutorial.de/

CHAPTER

ELEVEN

TIA PORTAL BASICS

In this module, the TIA Portal environment for PLC-programming is introduced.

11.1 Learning Outcome

- You will get familiar with the TIA Portal environment.
- You will be able to program Siemens PLCs using FBD, LAD, and SCL programming languages.

11.1.1 Introduction

TIA Portal is a Siemens software that provides a programming environment for PLCs. Different programming languages, like FBD, LAD, and SCL (a variant of ST), are supported in TIA Portal.

This module will go through the basics of starting a new project in TIA Portal.

11.1.2 Requirements

- *PLC-Basics*

11.1.3 What you need

Software

- TIA Portal V15 or newer
- PLCSIM Advanced V2.0 or newer

11.2 TIA Portal

TIA Portal is a Siemens software for programming PLCs. Also, PLCs can also be simulated using *PLCSIM Advanced*.

The TIA Portal environment includes a networks window. Networks act like code lines; code (each network) is run by the CPU from up to down every PLC cycle (i.e., network 1, then network 2, and so on). Programming in networks is useful for good code organization, especially if the programming language used is LAD or FBD.

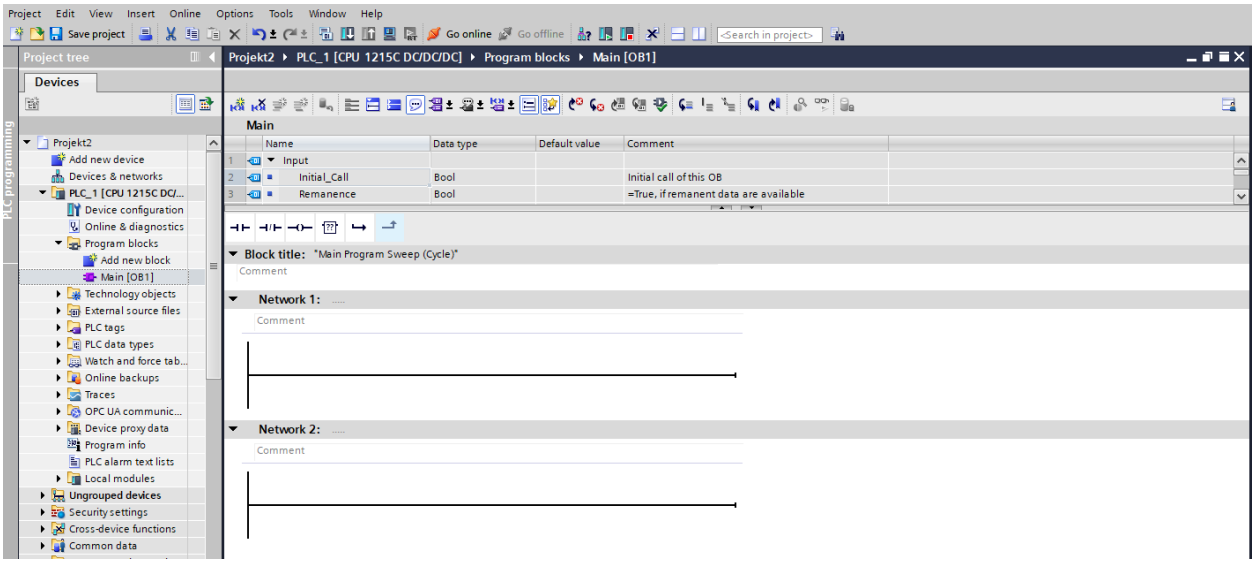


Fig. 11.1: Part of the TIA Portal environment - a new project

11.2.1 Programming Blocks

The design of PLC codes in TIA Portal is done with modular blocks. There are divided into 4 types according to their use.

These are:

- **Organizational block**: These provide structure to program. They serve as a link between the user program and the OS of PLC. Main [OB1] is a type of this block.
- **Function block**: This type of block is used to create code snippets which has its own data storage i.e., it has its own variable memory where the values are stored after the code snippet finishes its intended task.
- **Function code**: This type of block is generally used to create a reusable code structure. This block does not have a database linked to it i.e., it does not save values it calculates. Thus, a local stack of temporary values is used and gets deleted once the code snippet is exited (after executing its intended task).
- **Data block**: This is used to store data received from the code snippets. The data block can be either global DB or can be an instance DB.

11.2.2 Creating a new function block

A new project will only have a block called `Main [OB1]`. This is the project's main function. All user created functions and function blocks will be called inside the main function.

The block `Main [OB1]` in TIA Portal can be written using either LAD or FBD standard programming languages. In the context of this module, LAD will be used.

Note: The programming language of `Main [OB1]` can be changed through right click on `Main [OB1]` -> `Switch programming language`.

To create a block, click on `Add new block` under `Program blocks`.

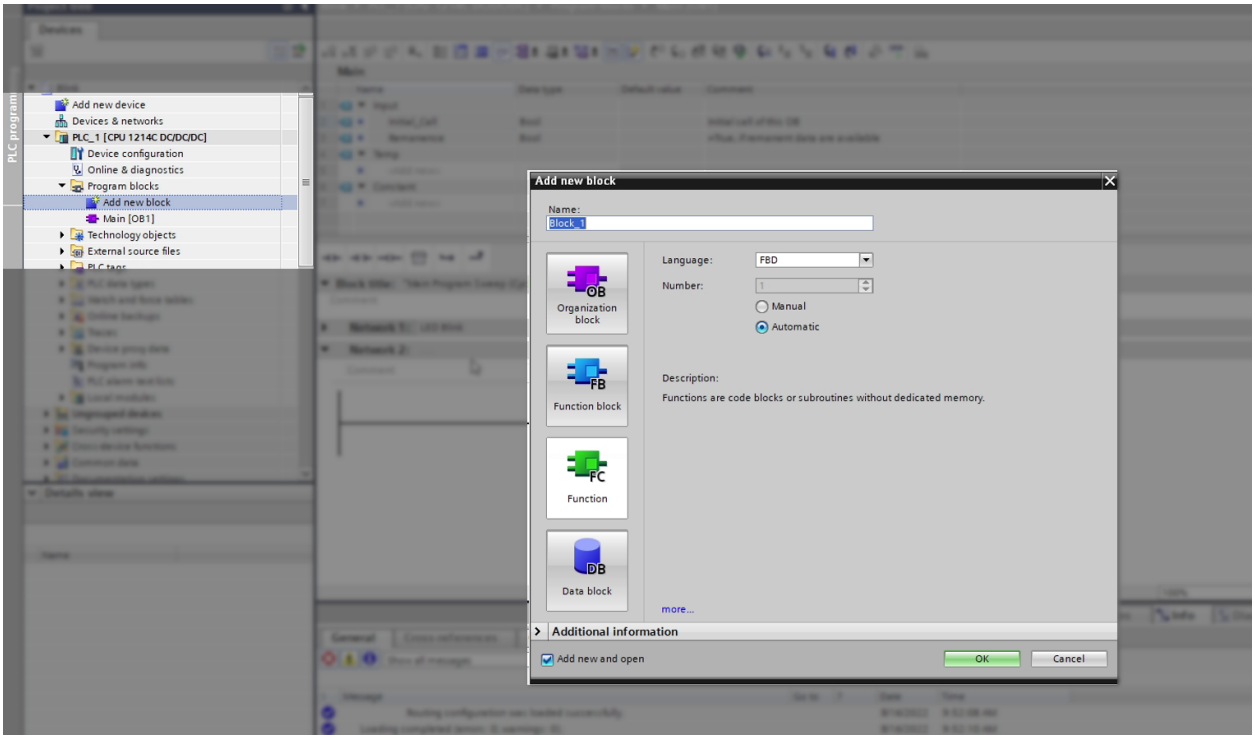


Fig. 11.2: Types of Programming blocks

Note: Blocks are useful for organizing and structuring a program.

The following figure shows a new `function block` that is named `State_machine`.

Variables can be declared and used inside the `function block`.

To run this FC, you will have to include it inside the `Main [OB1] block`.

To do so, drag and drop the `function block` into a `Main [OB1]` network to create instance of the `function block`. Once you drag it on the rail of Main, a pop-up to create a DB would appear. Click `OK` to create the DB. This `data block` stores all the instance variables related to that instance of the FC.

Note: All types of `Functions` have to be called in `Main [OB1]` to execute them. If they are not in `Main [OB1]`, they won't run or do their task.

A `data block` is not created to save variables of `function code / functions`. No instances of `function code / functions` exist.

`Function blocks`, however, are initiated as instances. Variables that belong to a specific instance of a `function block` are saved in a `data block` that specifically stores variables of that instance only. Creating another instance of the same `function block` type will create another `data block` for the new instance. Hence DB instance for LED_1 will be different from LED_2 and will operate separately.

Note: `Functions` with inputs have to have inputs connected to them in `Main [OB1]`. If no inputs are connected to the function, a compilation error will occur.

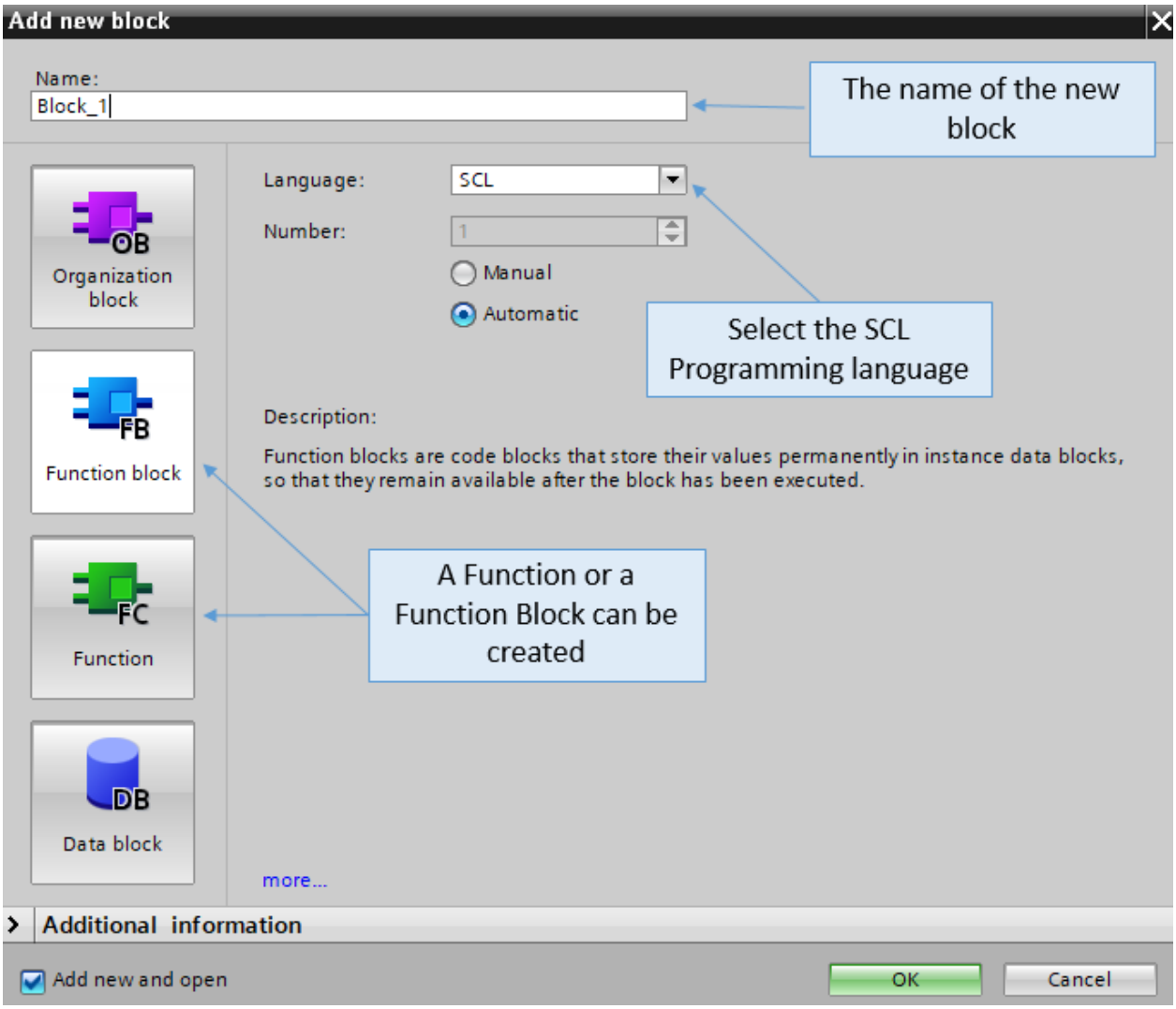


Fig. 11.3: Creating a new block

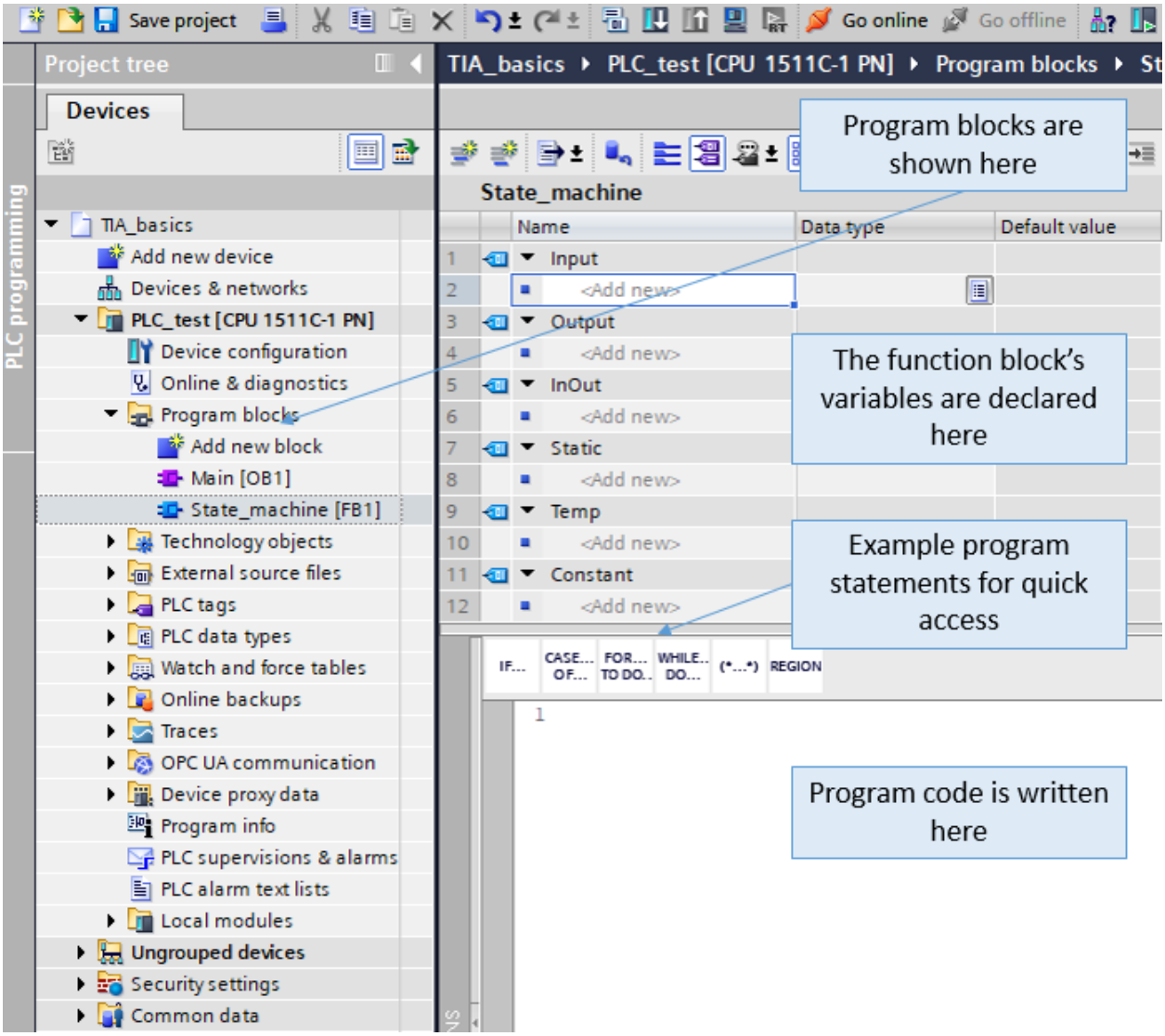


Fig. 11.4: TIA Portal function block environment

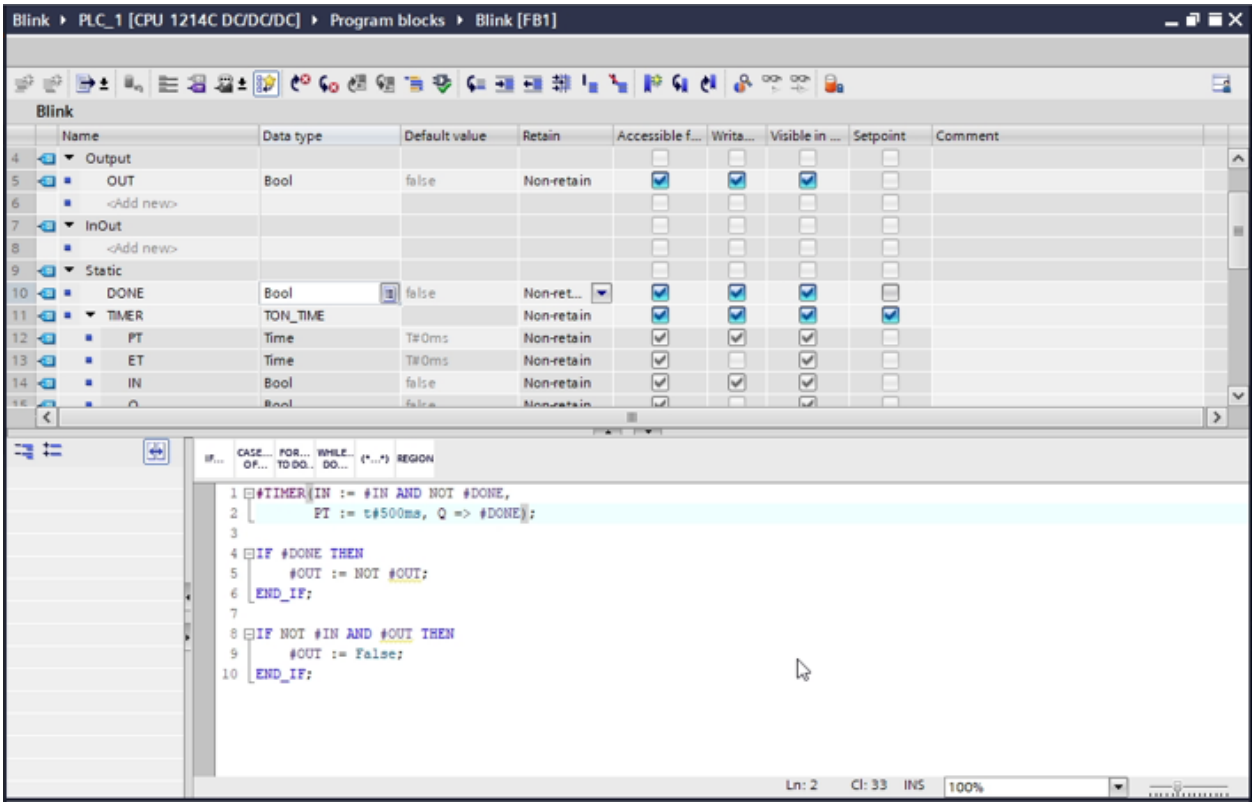


Fig. 11.5: Example Function Block

`Function blocks`, on the other hand, do not require inputs to be connected to them in `Main [OB1]`. Each instance of a function block has its own `Data block`, in which the inputs of that specific instance is saved.

11.2.3 Using Predefined Functions

TIA Portal provides various predefined functions ready for use. These can be found on the right side under `Basic instructions`.

For example, if you need a Mathematics based function, expand `Math functions` and select the required function.

Note: User-defined `functions` and `function blocks` are used in the same way in LAD.

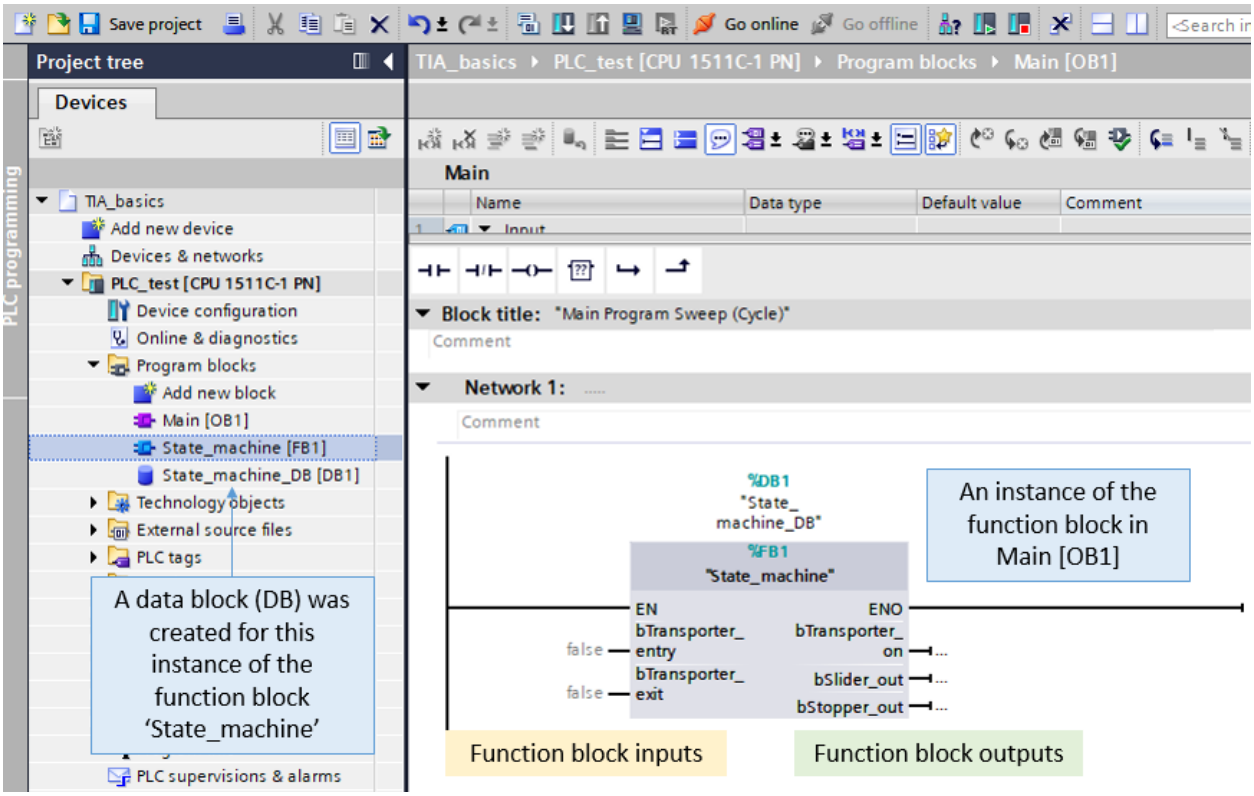


Fig. 11.6: A function block instance in Main

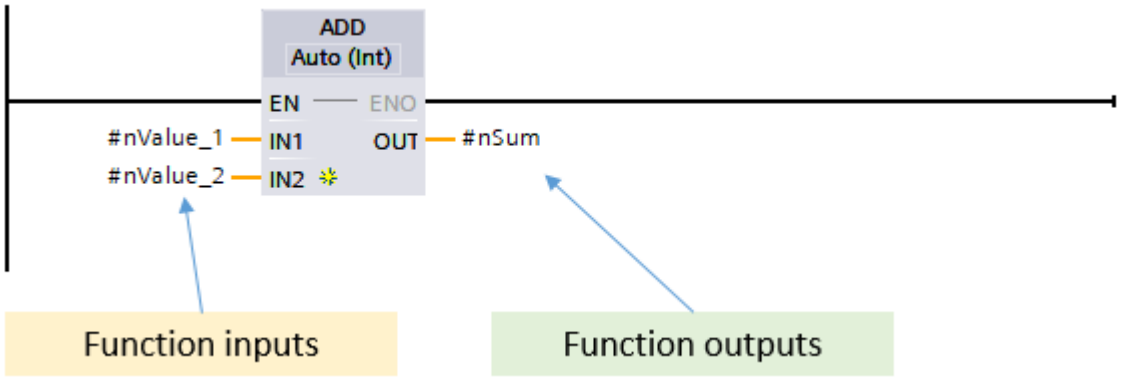


Fig. 11.7: An example of a standard function.

11.2.4 Watch and Force Tables

Watch and force tables are tables that are used to monitor variables in real-time while the PLC is running. Watch and force tables can also be used to overwrite variables in real-time. Because they can change variables' values, they are useful for testing a program. In this module, watch tables will be used to monitor and set variables while the program is running.

Deep Dive: Watch vs Force Tables

Force tables manipulate the peripherals. In this module, watch tables will be used to monitor and test the program. Watch tables are recommended for testing the program. For big programs, every scenario of inputs and outputs could be summed up in a watch table.

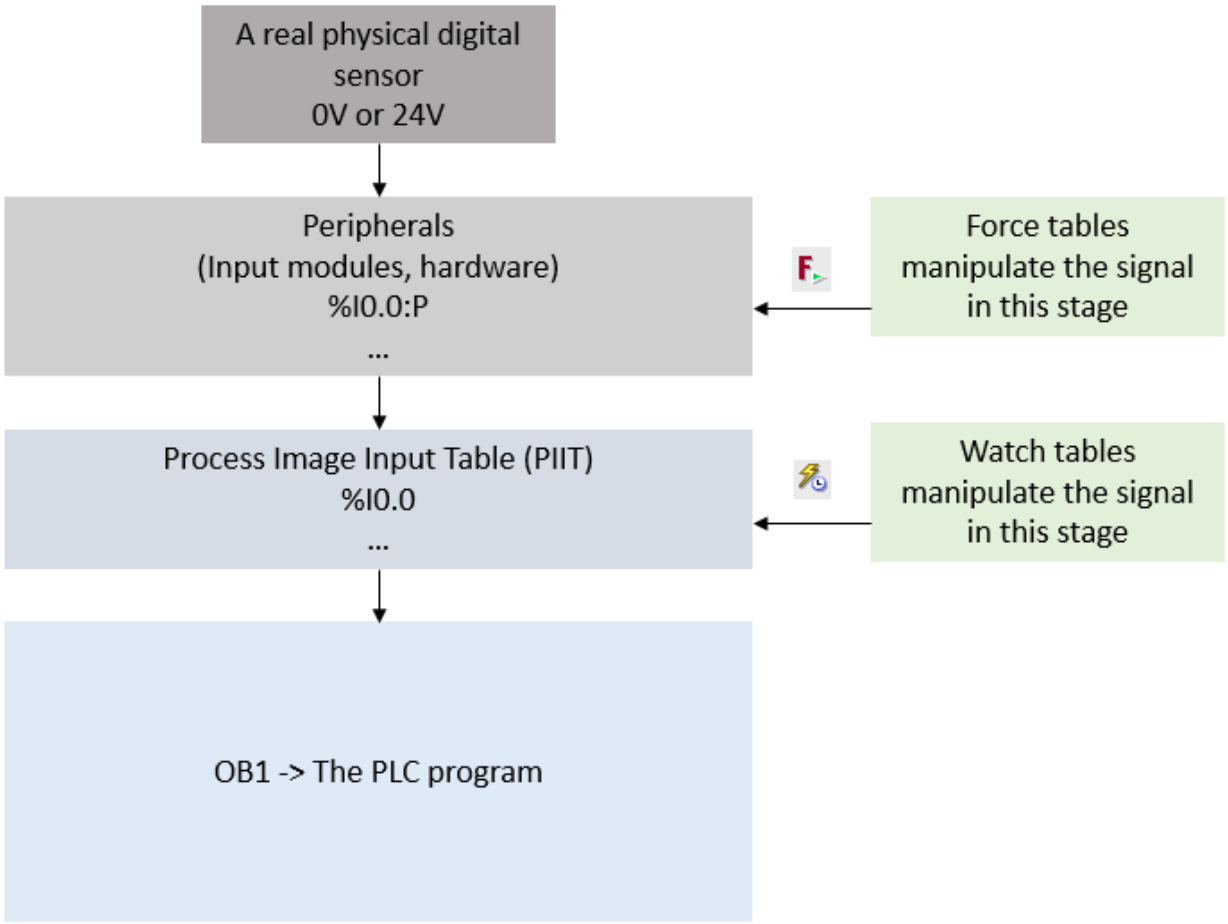


Fig. 11.8: The stage at which watch and force tables manipulate the signals

Creating Watch Tables

To create a new watch table, select `Add new watch table` under `Watch and force tables` in the `Project tree`.

Add the variables you need in the rows.

Set the PLC in *Online* mode (using `Online` -> `Go online`) and in *RUN* mode (`Online` -> `Start CPU`)

Once done, select the *Monitoring tool from the menu above*. Now you can watch the changes in the variables and also modify their values if required.

Hint: Watch tables are on PC hence you do not have to download the code again to the PLC.

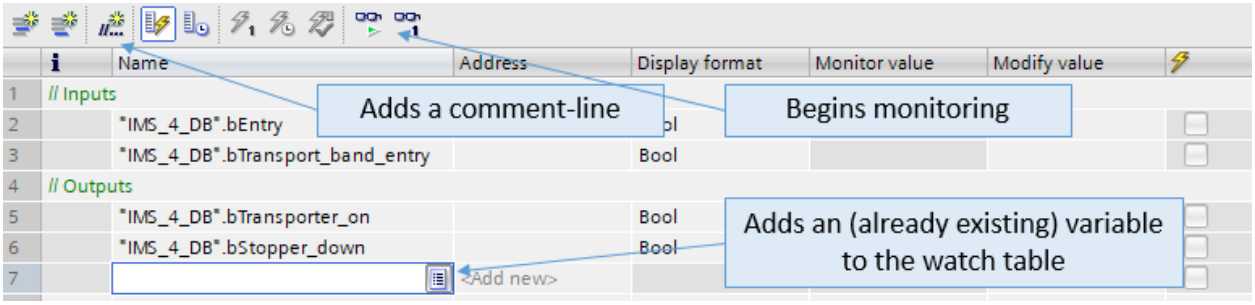


Fig. 11.9: An example set-up of a watch table

Note: All the variables shown in this watch table are part of a created `data block` called `IMS_4_DB`, hence the `"IMS_4_DB".` at the beginning of the variables' names. The address column is empty because these variables in this example were not mapped to a PLC I/O module, but are rather variables that belong to an instance of a user-defined `function block` called `IMS_4`.

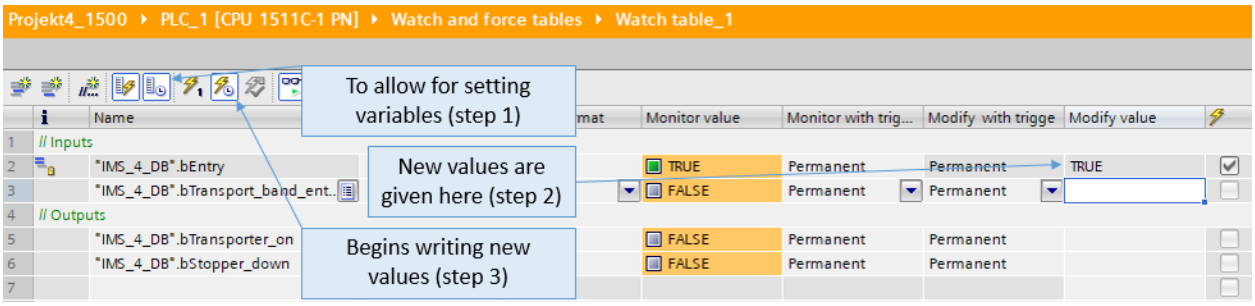


Fig. 11.10: An example watch table in action

Danger: Using a watch or a force table to **manipulate** variables influence the real system and overrides the program on the PLC. Be sure that it is safe to do so when working with a real PLC as it may be connected to moving parts or could unintentionally release liquids through valves.

11.2.5 Timers

There are various types of timers available according to IEC standard. Here are the commonly used timers with their timing diagram.

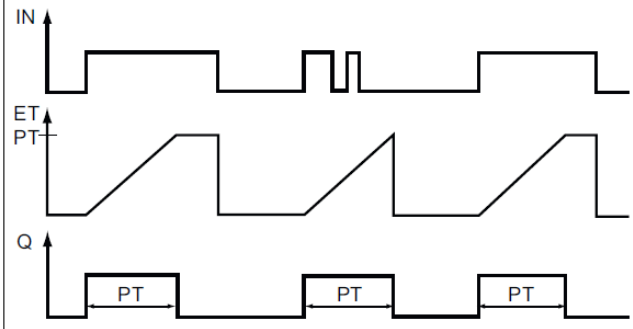
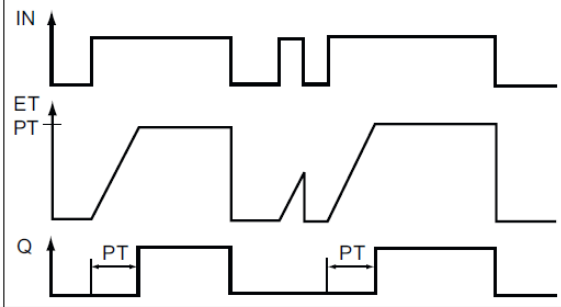
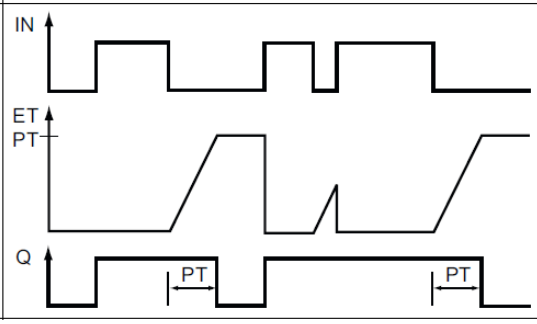
Timer	Timing diagram
TP: Generate pulse The TP timer generates a pulse with a preset width time.	
TON: Generate ON-delay The TON timer sets output Q to ON after a preset time delay.	
TOF: Generate OFF-delay The TOF timer resets output Q to OFF after a preset time delay.	

Fig. 11.11: Timers (ref: Siemens S7-1200 Programmable controller System Manual)

11.2.6 Monitoring runtime values

You can monitor runtime values using *Monitor Tool*. It looks like the image below. You will find it at various locations.

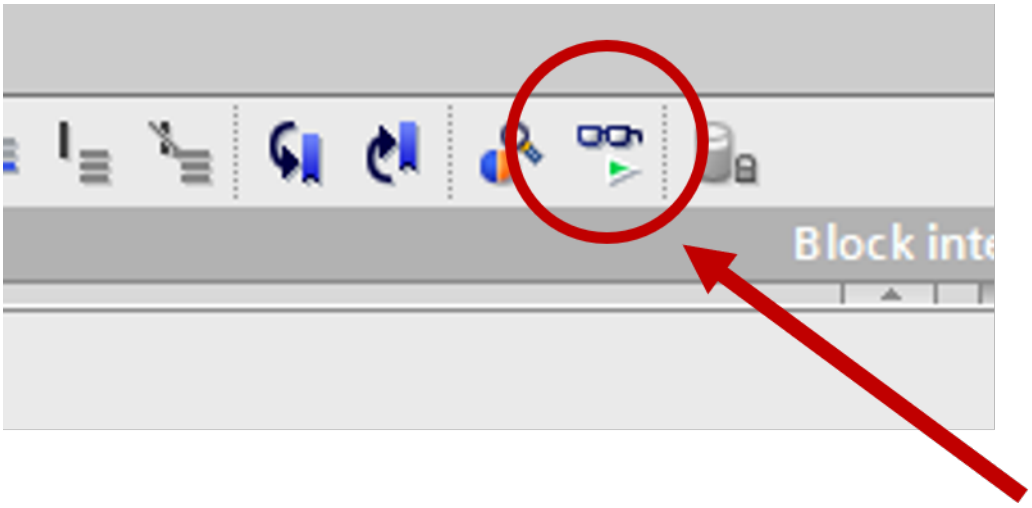


Fig. 11.12: Monitor Tool

CHAPTER

TWELVE

TIA OPC-UA

12.1 Configuring OPC-UA server in TIA Portal

To activate an OPC UA server:

- Go to `Device configuration` from `Project tree` from left side.
- Set the view to `Device view`
- Click on the PLC image and open `Properties`
- Go to `OPC UA` -> `Server` -> `General`
- Under Accessibility of the server, check the option `Activate OPC UA server` to activate it.
- In case a security pop-up appears, accept it (click OK).
- Find `Runtime licenses` in the same window (Properties)
- Under `Type of purchased license`, select SIMATIC OPC UA S7…
- Go back to `OPC UA` -> `Server` -> `General`
- Copy the server address.

Note: Write down (and copy it) the complete server address of the PLC. Please DO NOT forget opc.tcp and port number!

This information will be needed later when a client (for example UA Expert, Prosys OPC UA Browser or Siemens NX MCD) tries to communicate to the PLC over OPC UA.

Note: The OPC-UA server will not work if license is not set properly.

Warning: FOR THIS MODULE, PLEASE DO NOT CHANGE ANY MORE SETTINGS.

Danger: Be careful in setting security for the OPC-UA server when working on REAL PROJECTS.

Here, you are not working on hazardous equipment, hence *No Security* option under `Properties` -> `OPC UA` -> `Server` -> `Security` is enabled. Learn more in user manual before changing these settings.

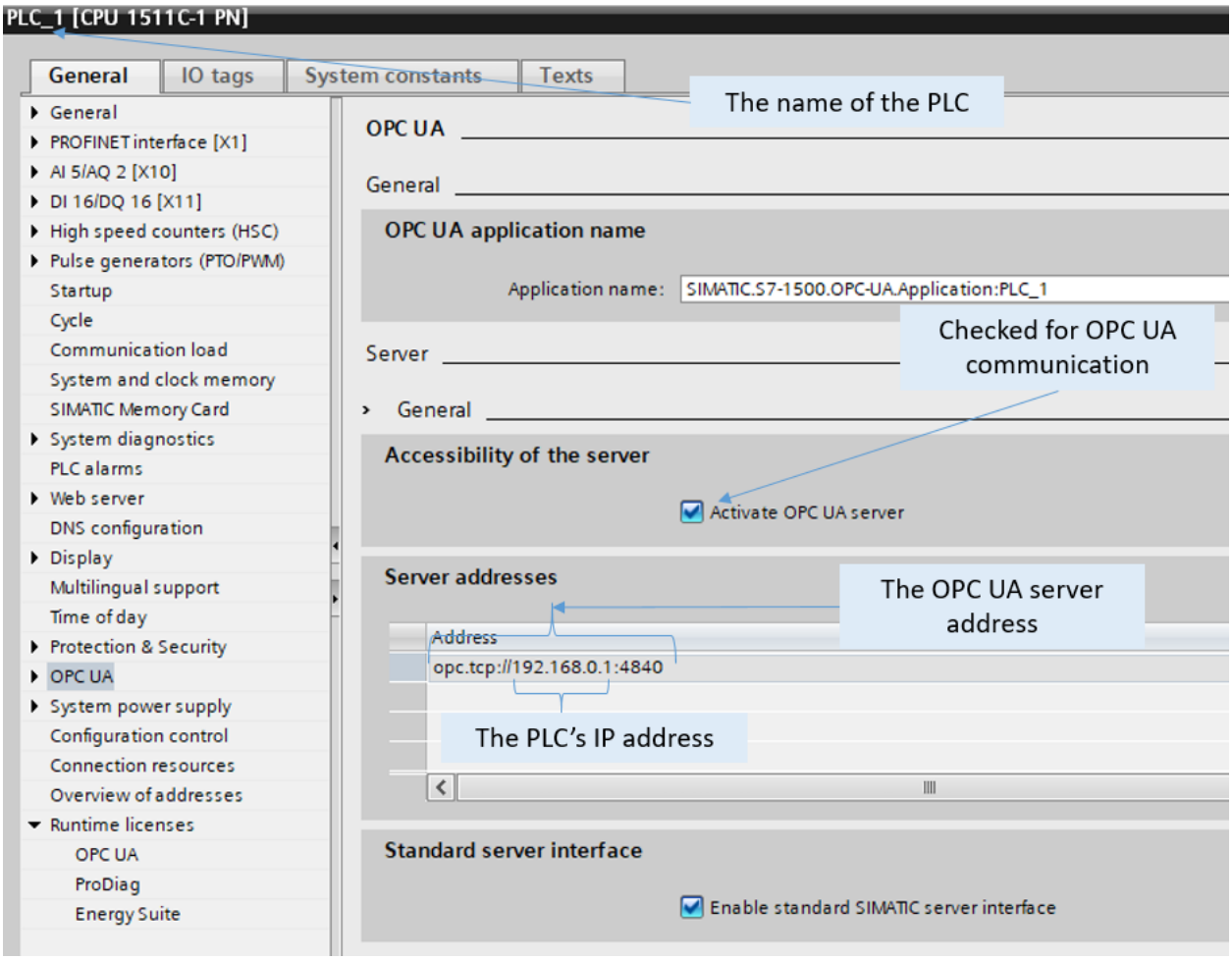


Fig. 12.1: Activating OPC UA server

12.2 Creating OPC-UA Server Interface

To add a server interface, perform the following steps:

1. Select the PLC name from the Project tree.
2. Expand the and select *OPC UA communication*
3. Select *Server interfaces* and expand it
4. Click on *Add new server interface* and rename the interface created
5. From the *Add new server interface* pop-up, select *Server interface* and click OK
6. Add the tags required to be monitored or to be modified using OPC Client. You will have to drag the tags from *OPC UA elements* to *server interface*
7. Upload the changes to the PLC

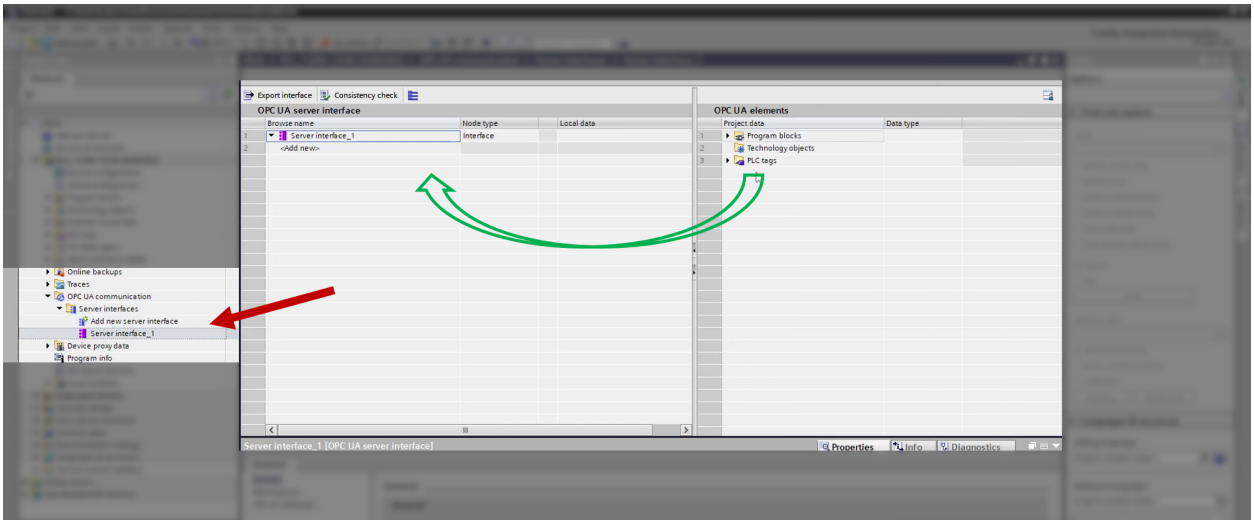


Fig. 12.2: OPC-UA Server Interface

Note: Keep the PLC connected with LAN cable.

- When uploading any hardware changes, keep the PLC in `STOP` mode and in offline mode. i.e., `Online -> Stop CPU` & `Online -> Go offline`
- When uploading only software changes, you can keep PLC in Online mode.

Danger: Be cautious in setting access of variable. You should not provide open access to all variables (tags).

Set the access in *Default Tag Table*

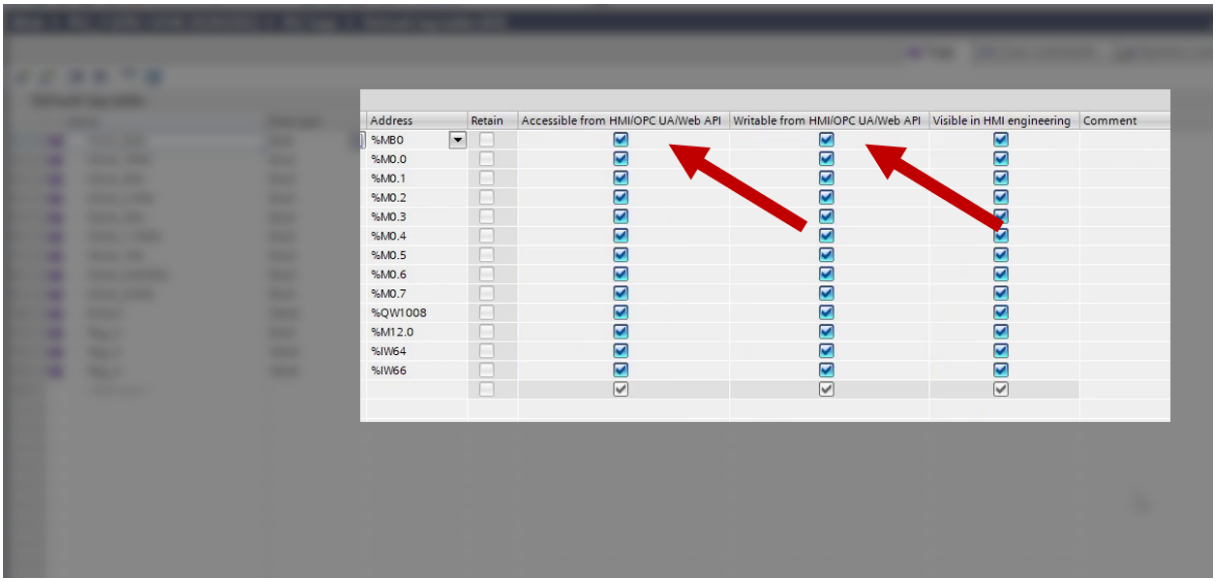


Fig. 12.3: Default Tag Table

CHAPTER

THIRTEEN

PLCSIM ADVANCED

13.1 Setting up simulation support in TIA Portal

Before starting a PLC simulation, one should first make sure the TIA Portal program allows simulations. To do that, right click on `Project's name` in the `Project tree``and select ``Properties`. In the `Protection` tab, a tick has to be placed next to `Support simulation during block compilation`.

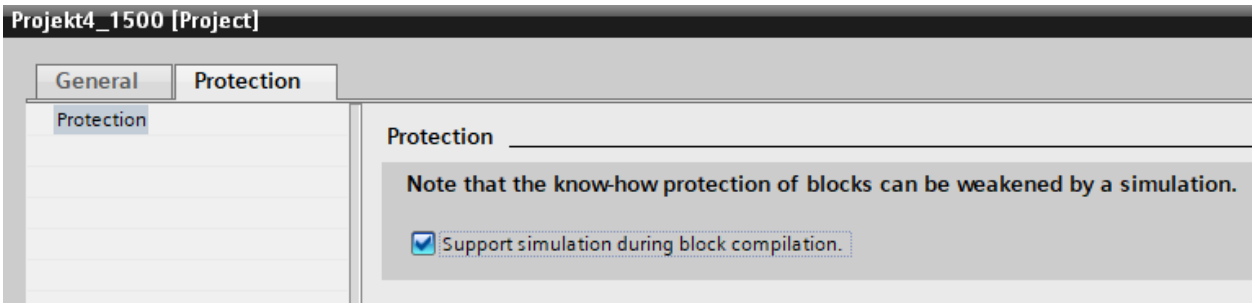


Fig. 13.1: Adding support for simulation in TIA Portal

Warning: PLCSIM does not offer simulated PLCs with communication features. If communication features, such as OPC-UA are required, simulations must be carried out using PLCSIM Advanced.

13.2 Launching PLCSIM Advanced and creating a PLC instance

Now PLCSIM can be launched. For that, double click on the `PLCSIM Advanced` icon on the desktop or search for `PLCSIM Advanced` in the search bar and click `Enter`.

Note: PLCSIM Advanced does not launch a window (in older PLCSIM Advanced versions). When it runs, it shows up on the tray icon. A right click on the tray icon will show the control panel.

Note: The PLC name and IP address have to be identical to those in the TIA Portal program. Use a subnet mask of `255.255.255.0`. Please note that the subnet mask depends on the network settings.

After `Start` is clicked, a virtual PLC instance should run.

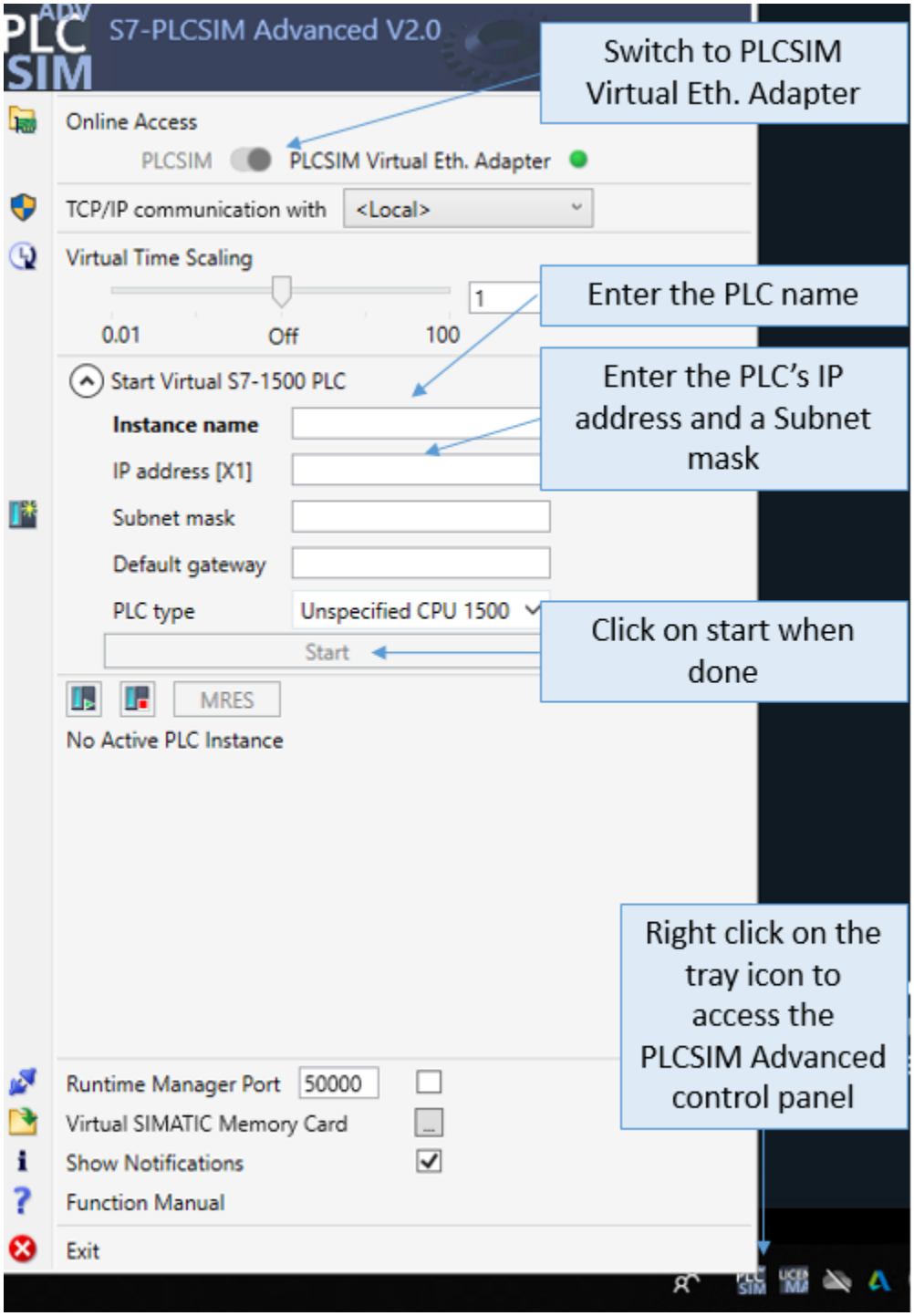


Fig. 13.2: Setting up a PLC instance

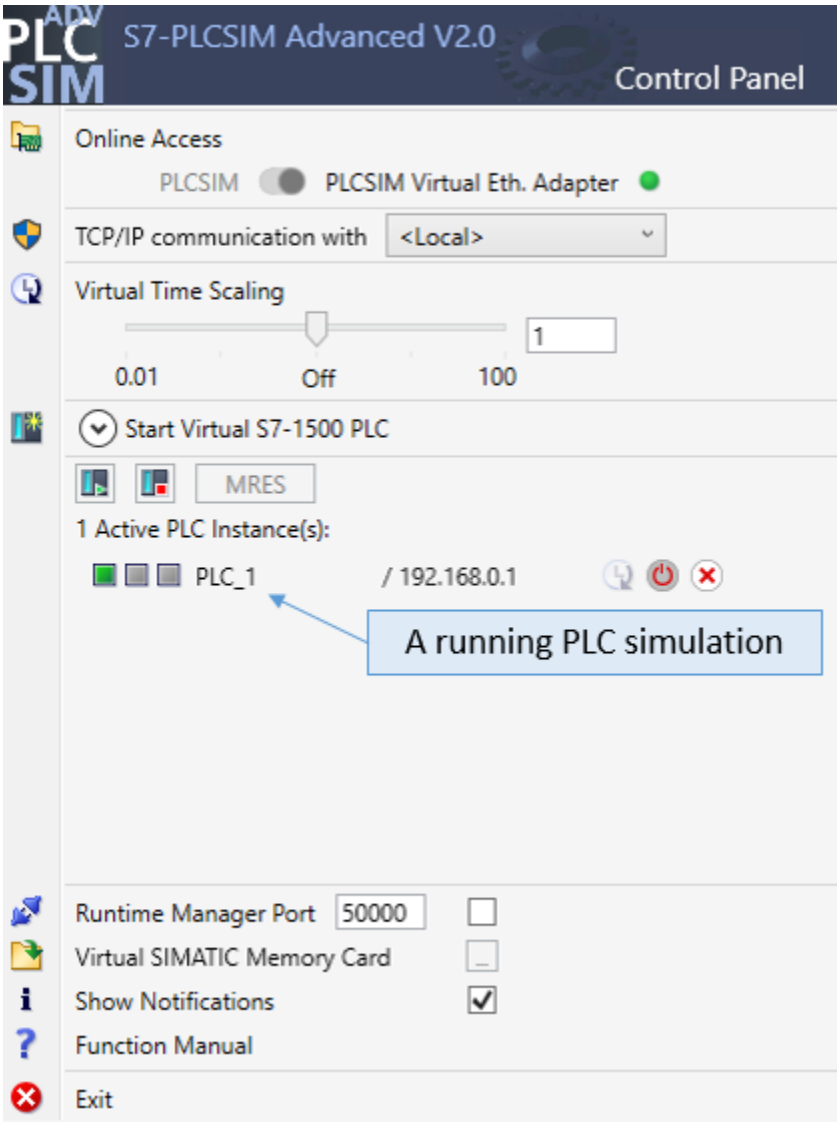


Fig. 13.3: A running virtual PLC instance

13.2.1 Deep Dive: PLCSIM vs PLCSIM Advanced

With TIA Portal, it is possible to simulate a Siemens PLC and test a program on the simulated PLC. With the software PLCSIM, it is possible to create such a simulation (12xx and 15xx PLCs), however, the simulated PLC has no communication capabilities. The software PLCSIM Advanced (a separate software than PLCSIM) can simulate Siemens 15xx PLCs with communication capabilities.

13.2.2 Summary

Simulating a PLC using PLCSIM Advanced allows for communication. The simulated PLC can communicate with other devices (through OPC-UA for example), allowing for a more realistic validation of control code.

CHAPTER

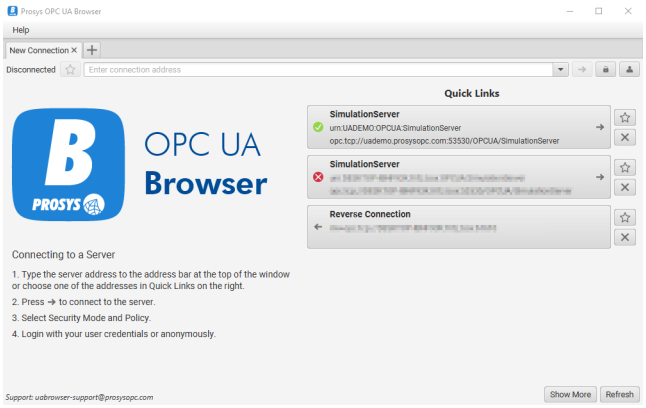
FOURTEEN

OPC-UA CLIENT

14.1 Prosys OPC-UA Browser

There are a lot of different OPC tools available. In this project the free to use tool Prosys OPC UA Browser from Prosys OPC Ltd is selected.

Warning: Prosys OPC-UA Browser requires the IP-address and port of the opc-ua server.



Start the tool and select the field `Enter connection address`. Fill in the address and port to the OPC-UA Server (i.e. the IP Address of the PLC and the default Port 4840). Next, click on the arrow and open the connection to the OPC-UA server.

Hint: A PLC might need to be in running mode for the OPC-UA server to be enabled.

Now you can browse through your OPC-UA `Objects` and search for the variables your OPC-UA server exposes. It is also possible to subscribe to data changes of variables.

CHAPTER

FIFTEEN

PLC-BASICS

Programmable Logic Controllers (PLC) are computers which are commonly used in commercial and industrial control applications. Generally speaking, PLCs monitor inputs and other variable values, make decisions based on a stored program and control outputs to automate a process or machine.

A PLC typically consists of input and output modules, a Central Processing Unit (CPU) and a programming device. The primary function of the input unit is to convert the signals at the inputs into logic signals which can be used by the CPU. The CPU analyzes the status of inputs, outputs and other variables and executes the stored program. After that, the CPU changes the output signals.

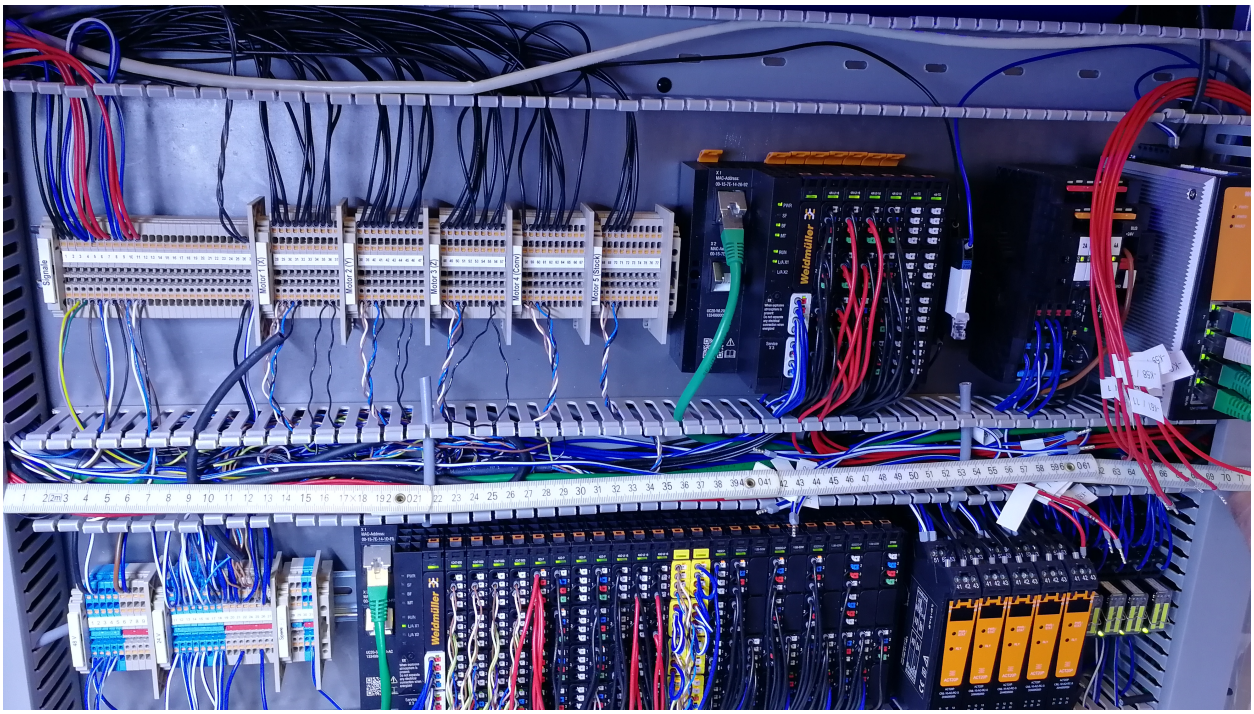


Fig. 15.1: Example for PLC wiring for industrial applications

Advantages in contrast to hard-wired control:

- Easy to modify input and output devices
- High flexibility due to modular design
- Significant reduction of cabling
- Solid-state, no moving parts

- Smaller physical size
- Integrated diagnostics and override functions
- Easy and cost-effective duplication possible
- Communication capabilities

15.1 Information processing

Information at the inputs of the PLC are processed in cycles. First, the CPU queries the input channels and stores the data in the working memory. This storage area is called “input image” as the stored input data does not show the current state of the inputs but the available data at the time of sampling.

Next, the program is executed step by step and the variables are stored in the memory. Finally, the calculated output parameters are stored in the “output image” and transferred to the output channels to control the connected machine.

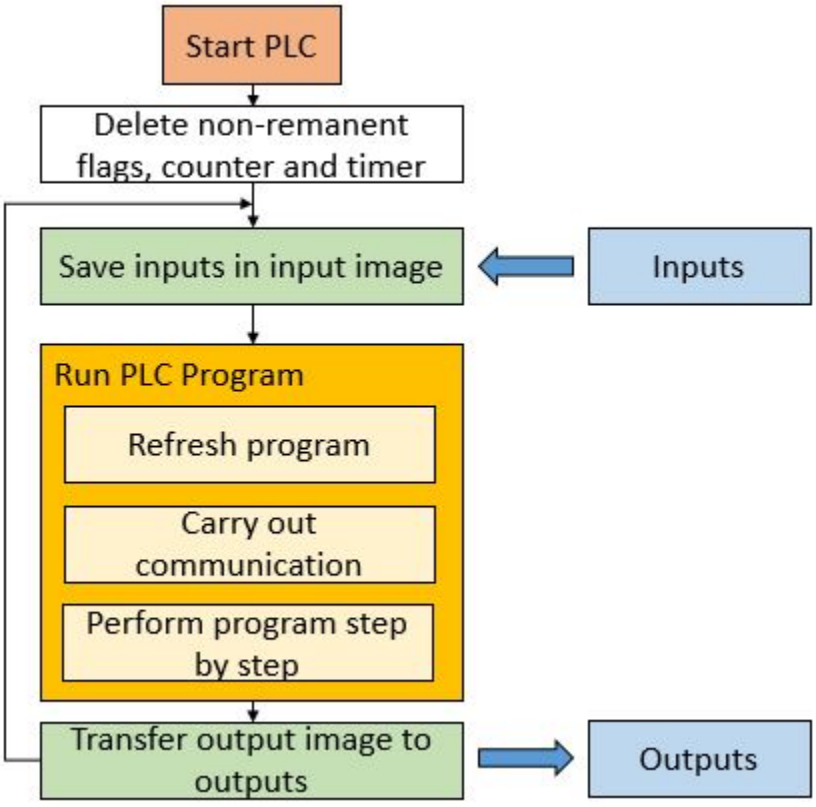


Fig. 15.2: Cyclic information processing of PLCs

Types of input and output signals:

- Binary inputs: Input module can distinguish between a high and a low level of the input signal (typical value: 0V and 24V).
- Binary outputs: Output modules generate TRUE (5V) or FALSE (0V) values which are typically intensified (24V) using transistors or relays.
- Analog inputs: A physical measurand is transformed into a voltage or current signal using a transducer. The resulting signal is transferred to the analog inputs of the PLC.

- Analog outputs: A voltage or current signal is generated to control actuators.

15.2 PLC Programming (IEC 61131)

PLC programming is standardized in the IEC 61131.

15.2.1 Controller configuration and resources

Creating a controller configuration, the hardware structure is specified in the programming environment. All resources which are controlled by the PLC software (input, output modules, interface cards, etc.) are declared. Nowadays, this is usually done automatically by scanning the connected components or by drag-and- dropping the parts.

Input channels are declared using `%I` and outputs specified writing `%Q`. The following letter defines the data type of the signal (bit `X`, byte `B`, word `W`, doubleword `D`). In this context, the declaration `%IX1.2` calls the second channel of the first binary input card.

Labelling of input and output addressing	1. Letter	2. Letter	Example
AT%	I Input	x Bit	AT%IX1.2
	Q Output	B Byte, 8 Bit	AT%QB0
		W Word, 16 Bit	AT%QW7
		D Doubleword, 32 Bit	AT%QD5

15.2.2 Tasks

Tasks organize the time schedule of programs. It works a bit like a cyclic step counter, which selects the instructions of the program in a clock-controlled way. Based on that, the CPU processes the instruction which is stored in the selected storage area.

Several programs can be assigned to one task running all in the same cycle time. The following process is conducted (IPO principle; input-processing-output):

- Read input data of all programs.
- Processing of all programs.
- Output of the output data of all programs.

The cycle time of all programs assigned to one task is identical as the output data of all programs is transferred simultaneously at the end of the processing cycle.

One CPU can process multiple tasks with different cycle times (multitasking). Therefore, the computing power is distributed between the different tasks using interrupt signals for changing.

15.2.3 Program Organization Units (POUs)

An object of the type `POU` is a Program Organization Unit in a CODESYS project. You write source code for the controller program in `POUs`. There are the following types of POUs:

- Programs
- Function Codes `FC`
- Function Blocks `FB`

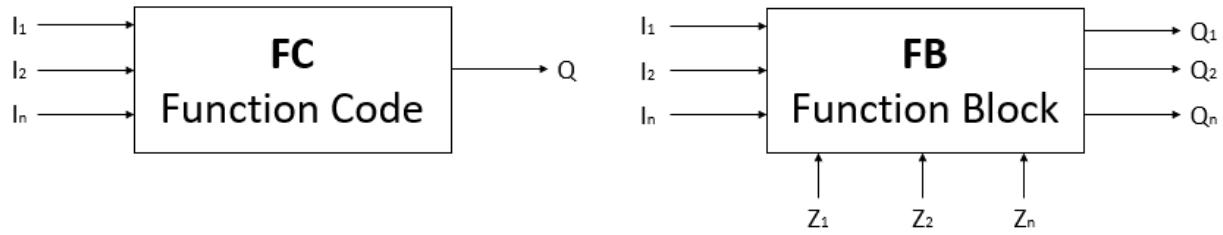


Fig. 15.3: Difference between Function Codes and Function Blocks

Function Codes can have several input variables but only one output variable. This variable is the return value of the Function Code. Function codes have no internal memory, thus they cannot store intermediate values. Function Codes can be separated between standard and user-specific functions. Standard functions like for example AND, ADD or BYTE_TO_WORD can be used directly during programming without declaring it separately.

Function Blocks can also be standardized or user-specific. Examples for pre-defined FBs are triggers, counters and timers. FBs can store intermediate values for the next IPO cycle and can have several outputs `Q`. The output signals are created by connecting the inputs `I` and the states `Z`.

15.2.4 Variables

Variables are used to establish a communication between different program organization units. Different kinds of variables are available:

- `VAR`: Local variables which are only valid in the associated POU.
- `VAR_GLOBAL`: Global variables are valid in all POUs. They are used for communication between different programs.
- `VAR_INPUT`: Input variables which are used to write inputs in FCs or FBs.
- `VAR_OUTPUT`: Output variables of FCs or FBs.
- `VAR_IN_OUT`: Input and output variables can be changed in the FB and then be released.
- `VAR_RETAIN`: Variables retain the value when the PLC is put off and on again.
- `VAR_PERSISTENT`: Variables retain the value if the software is loaded on the PLC.

Standard data types:

	Datatype	Size	Range	Example
Bit sequence	BOOL	1 Bit	FALSE / TRUE	FALSE
	BYTE	8 Bit	16#00 … 16#FF	16#00
	WORD	16 Bit	16#0000 … 16#FFFF	16#0000
	DWORD	32 Bit	16#00000000 … 16#FFFFFFFF	16#00000000
Whole numbers	SINT	8 Bit	-128 … 127	0
	INT	16 Bit	-32768 … 32767	0
	DINT	32 Bit	-2147483648 … 2147483647	0
Whole numbers without sign	USINT	8 Bit	0 … 255	0
	UINT	16 Bit	0 … 65535	0
	UDINT	32 Bit	0 … 4294967295	0
Floating point figure	REAL	32 Bit	-3.4 * 10^38 … 3.4 * 10^38	0
Time	TIME			
Time of day	TIME_OF_DAY			
Date	DATE			
Character sequence	STRING			

A typical variable declaration consists of five parts:

`<variable name> AT<address> :<data type> :=<initial value>; (* comment *)`

example: `buttonStart AT%IX0.0 :BOOL :=FALSE; (*Start Button*)`

Note: It is helpful to use speaking names instead of numbers to make it easy to understand the program (e.g. buttonLight and buttonVentilator instead of button1 and button2).

Derived data types:

Standard data types can be modified to fulfil special needs: `TYPE Name : ... END_TYPE`

Four different kinds can be distinguished:

- Enumeration: `TYPE A_STATUS : (START,RUN,WAIT,STOP); END_TYPE`
- Area: `TYPE A_DATA : UNIT(0..16#3FF); END_TYPE`
- Field: `TYPE A_4IN : ARRAY[1..4] OF A_DATA; END_TYPE`
- Structure: `TYPE str_Motor : STRUCT ... END_STRUCT; END_TYPE`

Enumerations are typically used if different options are possible and if easy readable code should be produced.

15.3 Programming languages (IEC 61131)

Programmable Logic Controller (PLC) programming as other programming tasks has defined set of rules described in the IEC 61131-3. In this Standard, information about programming concepts and industry accepted programming languages is provided. For this module, SCL, LAD and FBD are selected as they are most commonly used.

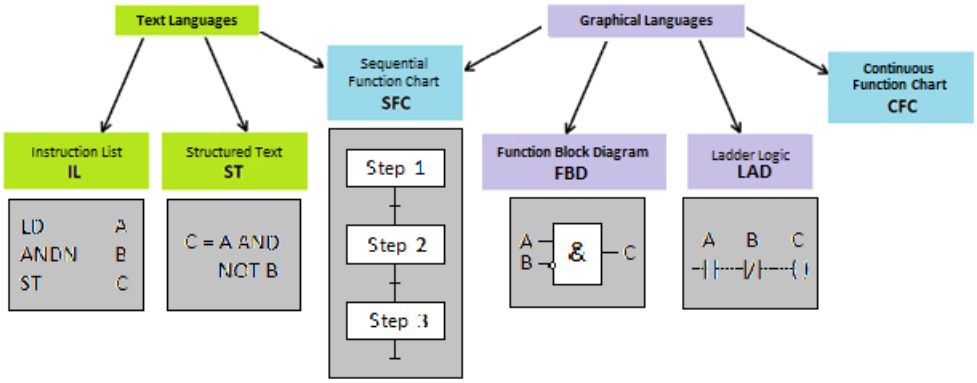


Fig. 15.4: Five programming languages of the IEC 61131-3 standard (source: https://www.motioncontroltips.com/iec-61131-3-plcopen/)

LAD	SCL

"IN"	IF in THEN
	Statement;
	ELSE
	Statement;
	END_IF;

"IN"	IF NOT (in) THEN
	Statement;
	ELSE
	Statement;
	END_IF;

LAD	FBD	SCL
"OUT"	"OUT" =	out := <Boolean expression>;

FBD	SCL[1]
& "IN1" "IN2"	out := in1 AND in2;
>=1 "IN1" "IN2"	out := in1 OR in2;

Fig. 15.5: SCL, LAD & FBD (ref: Siemens S7-1200 Programmable controller System Manual)

15.3.1 Structured Control Language (SCL)

SCL (Structured Control Language) is a high-level text-based programming language. It is based on PASCAL.

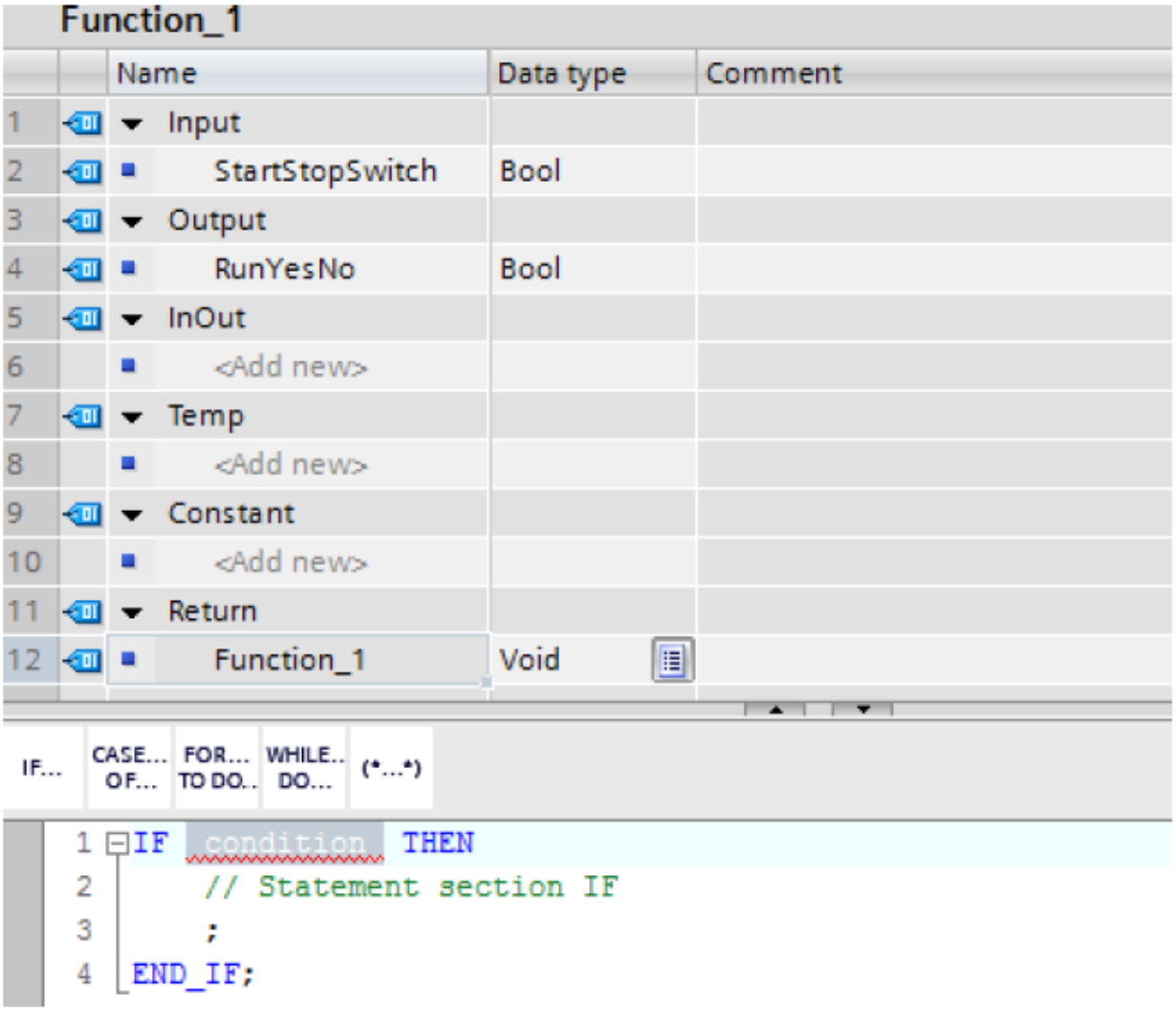


Fig. 15.6: An example of a SCL code 1.

Various important syntax for SCL are provided below:

Assignments: `A := 10;` (The variable A is assigned the value 10.)

Statements: `blink : BOOL;` (Create a variable `blink` and assign the type `BOOL`.)

Mathematical functions: +, -, *, /

Addressing of global variables (tags): “<tag name>” (Tag name or data block name enclosed in double quotes)

Addressing of local variables: #<variable name> (Variable name preceded by “#” symbol)

Comments

Single line comment: `// comment`

Multi line comment and comments after end of ST line: `<statement>; (* comment *)`

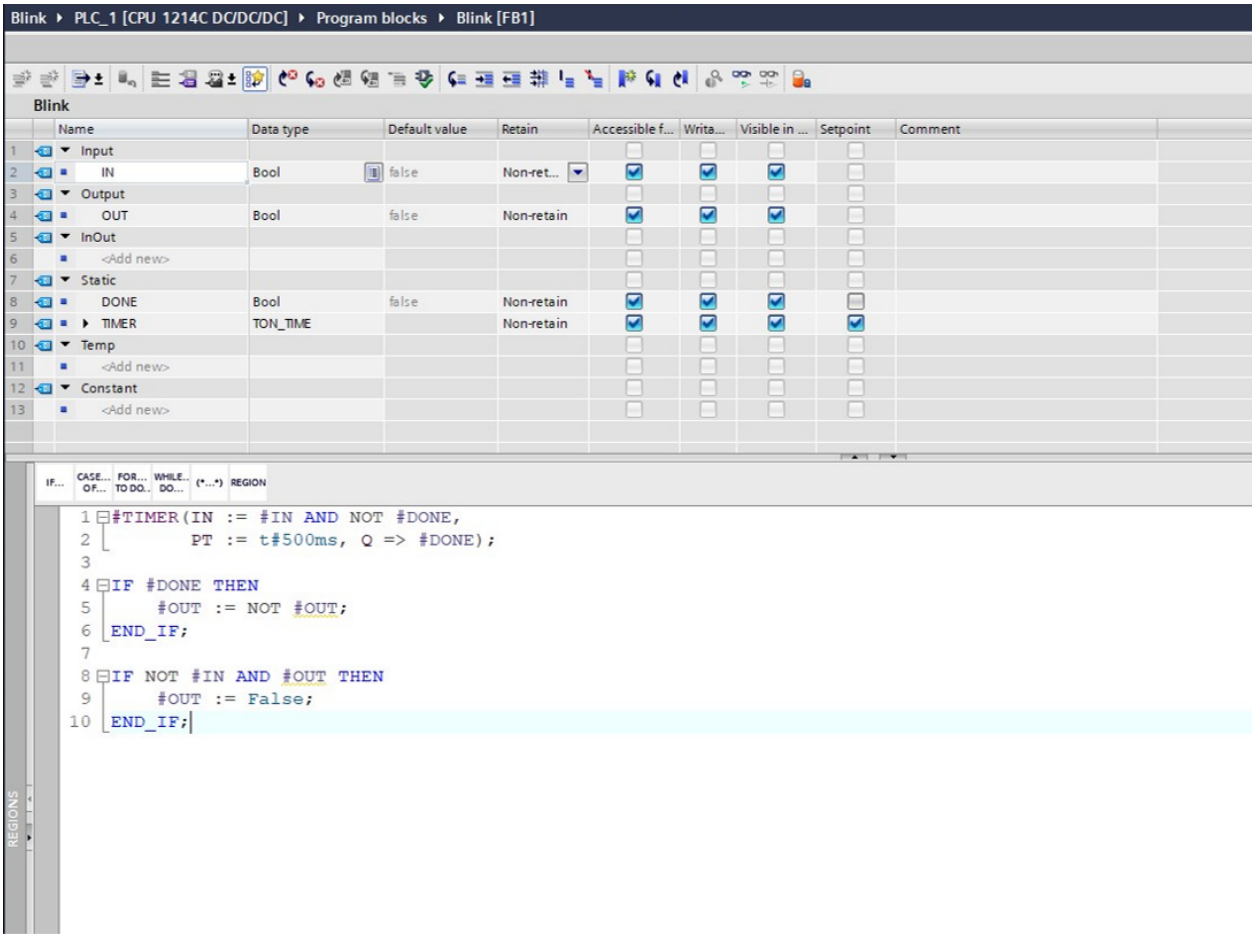


Fig. 15.7: An example of a SCL code 2.

Operators: Standard operators in ST ordered after precedence:

Operation	Symbol	Precedence
Parentheses	`(expr)`	Highest
Function Evaluation	`MAX(A,B)`	
Negation Complement	`NOT`	
Exponentiation	`**`	
Multiply	`*`	
Divide	`/`	
Modulo	`MOD`	
Add	`+`	
Subtract	`-`	
Comparison	`<,>,<=,>=`	
Equality / Inequality	`= / <>`	
Boolean AND	`& AND`	
Boolean Exclusive OR	`XOR`	
Boolean OR	`OR`	Lowest

Note: `A = B` and `A := B` are NOT SAME!

= (equality operator) evaluates if the left and the right side is equal. If value of `A` is equal to value `B`. It returns `TRUE` if yes, else it returns `FALSE`.

`:=` denotes a statement. It is used for assigning value of right side to the left side. In this case, the value of B is assigned to A.

Combining operators:

```
IF (Input1) AND (Input2) OR (Input3) THEN
    Output1 := True;
END_IF
```

IF Statements: If statements are used for boolean queries.

```
IF [boolean expression] THEN
    <statement>;
ELSIF [boolean expression] THEN
    <statement>;
ELSE
    <statement>;
ENDIF;
```

CASE Statements: Case statements are one of the most important structuring methods to generate readable code. Case statements are typically used to program state machines.

```
TYPE
    Steps:(INIT:=0, START, RUN, END);
END _TYPE

VAR
    state: Steps; (*use of enumeration*)
END_VAR
```

(continues on next page)

(continued from previous page)

```
CASE state OF
    INIT: Instruction_A;
    START: Instruction_B;
    RUN: Instruction_C;
    END: Instruction_D;
END_CASE
```

FOR Loops: It is used to repeat code a specific number of times.

```
FOR count := initial_value TO final_value BY increment DO
    <statement>;
END_FOR;
```

WHILE Loops: It is used to repeat the loop as long as some conditions are TRUE. A WHILE loop will repeat as long as a boolean expression evaluates to TRUE.

```
WHILE [boolean_expression] DO
    <statement>;
END_WHILE;
```

REPEAT Loops: It works the opposite way of the WHILE loop. This loop will stop repeating when a boolean expression is TRUE.

```
REPEAT
    <statement>;
UNTIL [boolean_expression]
END_REPEAT;
```

15.3.2 Ladder Diagram (LAD)

Ladder diagrams are specialized schematics commonly used to document industrial control logic systems.

In terms of TIA Portal program for Siemens PLCs, SCL plays an important role in programming of `functions codes` and `function blocks`. The `Main [OB1]` block is preferred to be programmed using the Ladder Diagrams.

A program written in LAD consists of networks. The language is based on relay logic. The flow of the logic 'true' across a network is similar to the flow of electricity across a relay logic. One can think of the left rail as being positive, and the right rail as being negative. The logic 'true' flows from the left rail to the right rail.

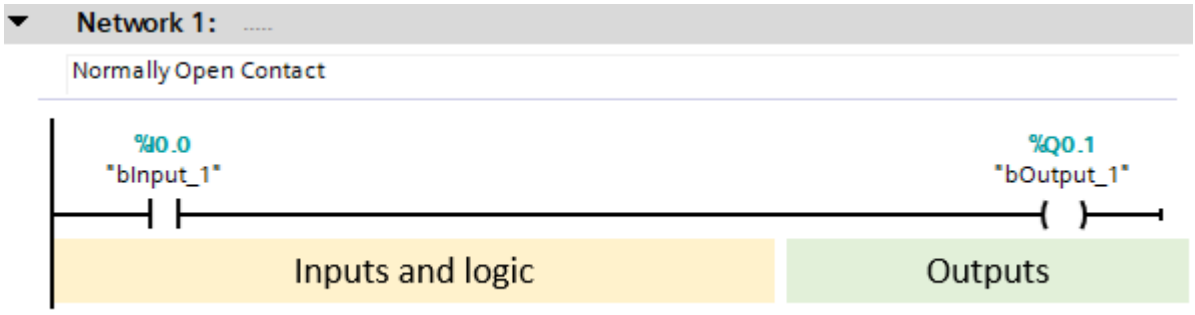


Fig. 15.8: An example of a network in LAD.

Various logical operations can be thus programmed in LAD. The following figure shows AND & OR logic after starting the PLC or simulated PLC.

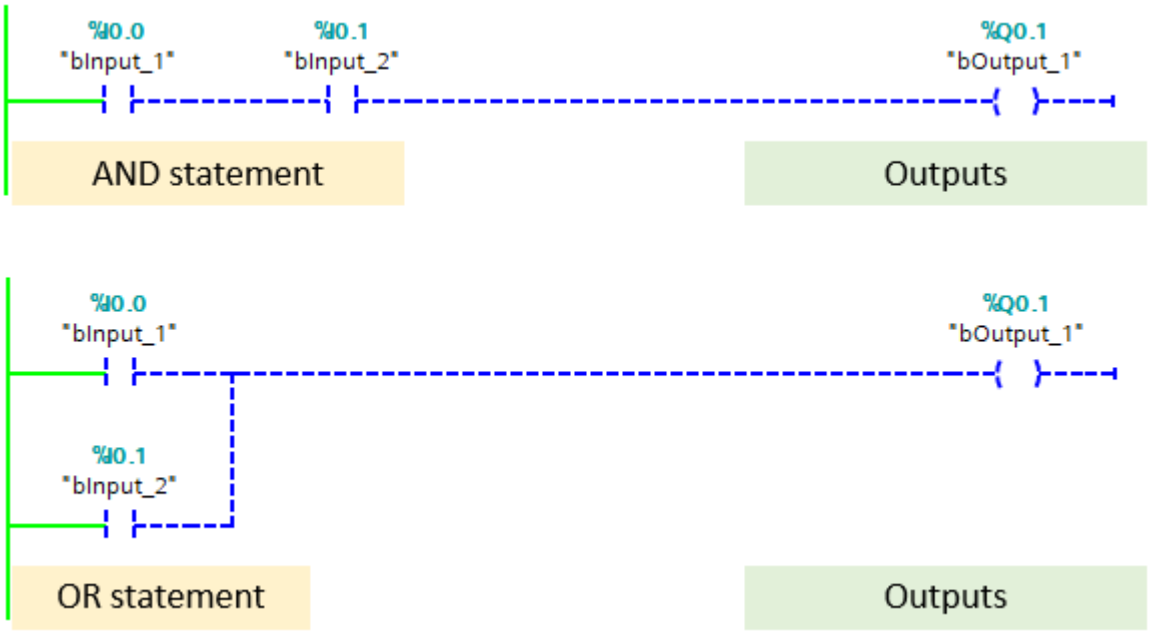


Fig. 15.9: An example of logical operations in LAD.

A general list of Bit Logic instructions used in LAD are mentioned below

Operation	Symbol
Normally Open Contact (Address)	`---| |---`
Normally Closed Contact (Address)	`---| / |---`
Save RLO into BR Memory	`---(SAVE)`
Bit Exclusive OR	`XOR`
Output Coil	`---( )`
Midline Output	`---( # )---`
Invert Power Flow	`---|NOT|---`
Insert branch	
Merge branch	

15.3.3 Function Block Diagram (FBD)

FBD is another graphical programming language. It uses Boolean algebra-based blocks to develop codes. A function block is depicted as a rectangular block. The inputs are on the left and outputs on the right side. It can have standard functions, such as logic gates, mathematical operations, counters, or user defined functions.

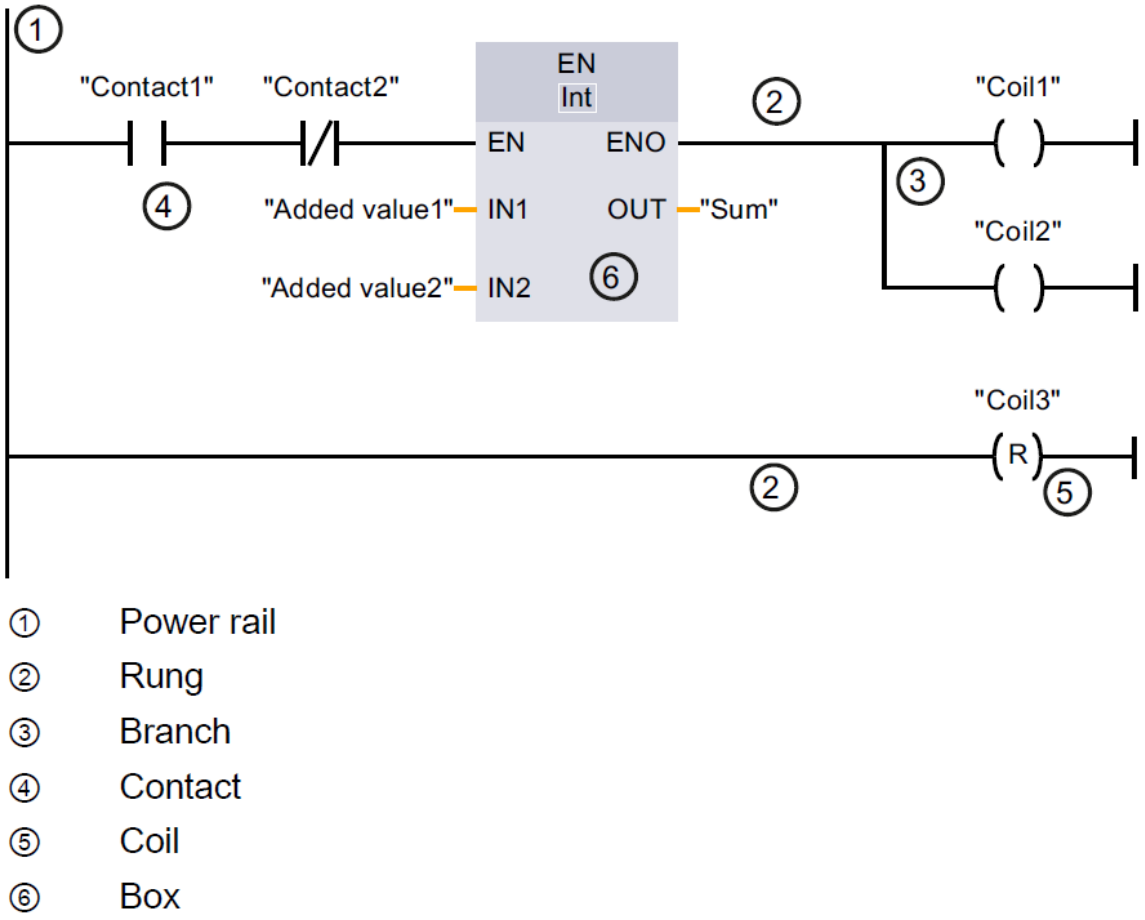


Fig. 15.10: An example of a network in LAD. (ref: SIEMENS TIA Portal STEP 7 Basic V10.5)

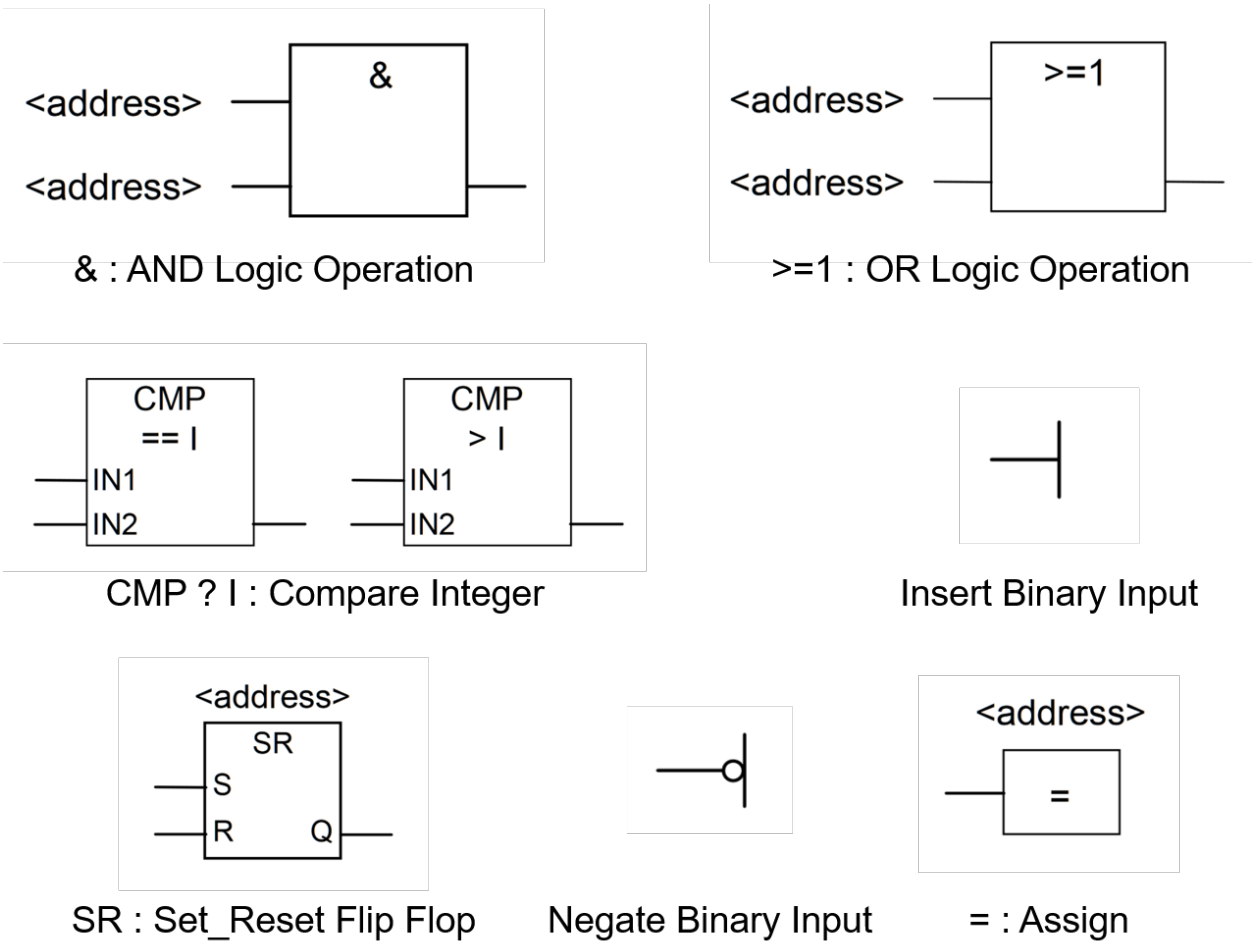


Fig. 15.11: Common FBD blocks (ref: SIEMENS Function Block Diagram (FBD) Reference Manual)

15.3.4 Additional languages

The Instruction List (IL) is very similar to the programming language Assembler and typically used to limit the computing time. In the example, the operand `A` is loaded in the working memory in the first line. Next, an `AND` operation with the negated operand `B` is conducted. Finally, the result is stored in the variable `C`. However, IL gets very long and confusing if complex tasks are programmed. As computing time usually does not play an important role today, high-level languages like Pascal, C or Structured Text (ST) are used typically.

The Sequential Function Chart (SFC) can only be used to program sequential control tasks in the form of step chains.

15.3.5 Examples

SCL

A complete example (programming of a traffic light) is shown below. First, an enumeration including all process states is defined. Next, all necessary variables are declared (a variable called `state` is defined using the new created data type `stateType`). Finally, the program sequence is established using a `CASE OF` statement.

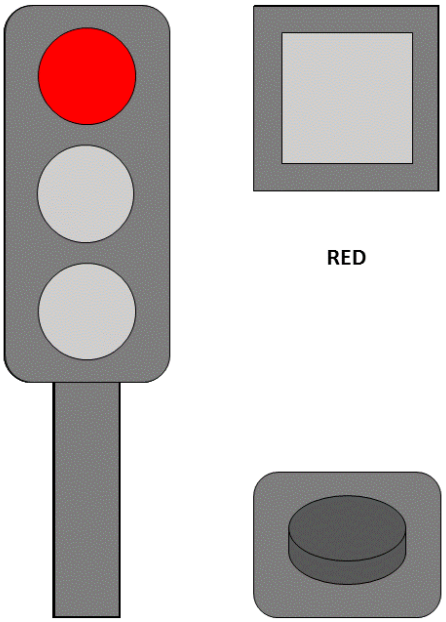
```
TYPE stateType :
    (RED:=1,REDYELLOW,GREEN,YELLOW);
END_TYPE

VAR
    state:stateType;
    button:BOOL;
    lred,lyellow,lgreen:BOOL;
    atime:TON;
END_VAR
```

```
CASE state OF
    stateType.RED:
                IF NOT lred THEN
                    lred:=TRUE;
                END_IF
                IF button THEN
                    state:=stateType.REDYELLOW;
                END_IF
    stateType.REDYELLOW:
                IF NOT lyellow THEN
                    lyellow:=TRUE;
                    atime(IN:=TRUE,PT:=#5s);
                END_IF
                IF atime.Q THEN
                    state:=stateType.GREEN;
                END_IF
    stateType.GREEN:
                IF NOT lgreen THEN
                    lgreen:=TRUE;
                    atime(IN:=TRUE,PT:=#40s);
                END_IF
                IF atime.Q THEN
                    state:=stateType.YELLOW;
```

(continues on next page)

(continued from previous page)

```
                END_IF
    stateType.YELLOW:
                IF NOT lyellow THEN
                    lyellow:=TRUE;
                    atime(IN:=TRUE,PT:=#5s);
                END_IF
                IF atime.Q THEN
                    state:=stateType.GREEN;
                END_IF
END_CASE
```



15.4 Sources

- https://www.plcacademy.com/structured-text-tutorial/
- Book Speicherprogrammierbare Steuerungen für die Fabrik- und Prozessautomation, Matthias Seitz, 2012*

CHAPTER

SIXTEEN

SIEMENS NX MCD

In this module, Siemens NX MCD is introduced and the most common functions that are used in constructing kinematic models are presented.

16.1 Learning Outcome

- You will get familiar with virtual commissioning.
- You will get to know the basics of Siemens NX MCD.
- You will understand and be able to create physics-based models.

16.2 Virtual Commissioning

Virtual commissioning is the process of testing and debugging control code on virtual models, possibly before the machine/ components materialize. It allows for early validation of code and decreases real commissioning time.

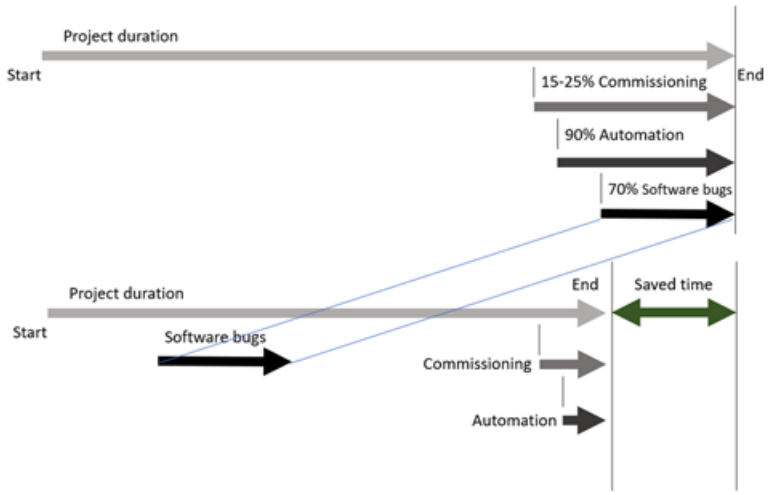


Fig. 16.1: Influence of control software debugging on project time with (figure below) and without (figure above) virtual commissioning

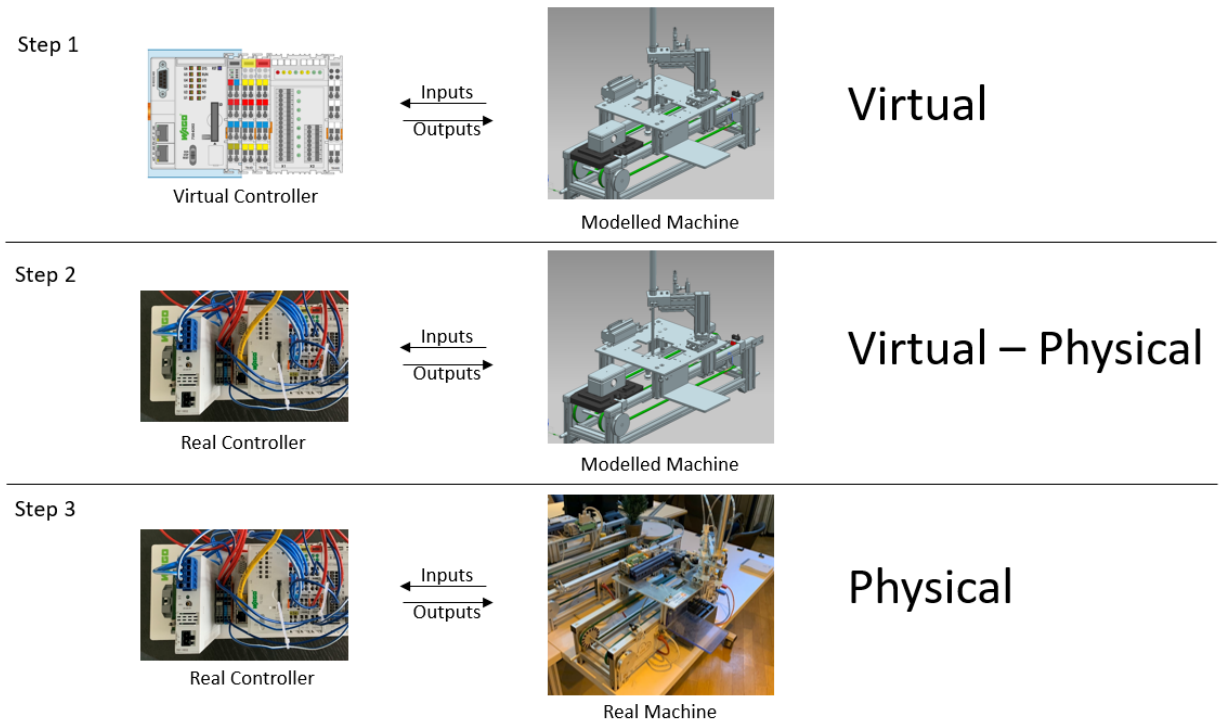


Fig. 16.2: Typical steps of virtual commissioning

Virtual commissioning can start during the development stages of the product/ asset. At stages where the material objects are not yet available, virtual commissioning allows for early detection of bugs in the PLC code and design flaws in the asset.

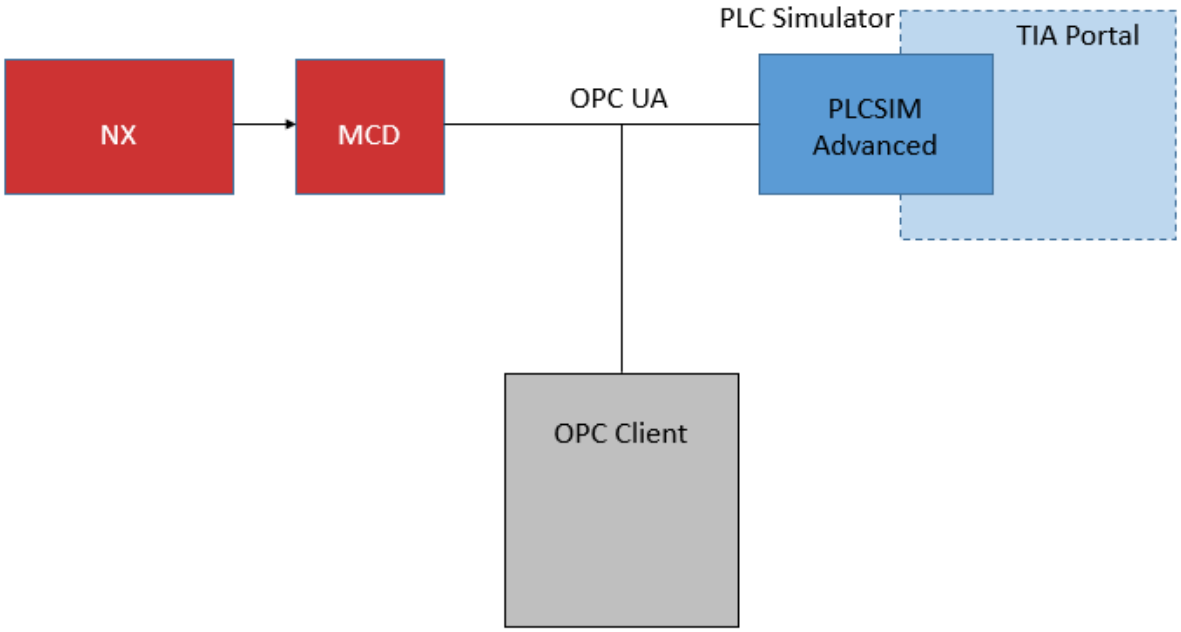
16.3 Mechatronics Concept Designer

16.3.1 Siemens NX General Actions

NX is an advanced CAD/CAM/CAE software. The following are some common actions used when working with a CAD assembly in NX.

Table 16.1: General NX Commands

Action	How to do it
Zoom in/out	Mouse wheel
Move the view left/right/up/down	Click and hold both the mouse wheel + the right mouse button and move the mouse
Rotate the view	Click and hold the mouse wheel and move the mouse
Move/Rotate an object relative to other objects/global coordinate system	Assemblies tab > Move Component > Select Components > Specify Orientation with arrows



16.3.2 Modelling with MCD

Mechatronics Concept Designer is an application inside Siemens NX that allows the creation of physics-based models and simulations based on CAD files. The goal of the following sections is to guide you through the basics of creating a physics-based model and influencing it from outside the simulation using external signals. To navigate to MCD, open a part or an assembly in Siemens NX. In the Application tab, click on more and choose Mechatronics Concept Designer.

Rigid Bodies

Solids (i.e., CAD models with no assigned physical properties in MCD) are only there to visualize the assembly and do not move during the simulation. Once a part gets assigned as a rigid body, it starts participating in the simulation. As a rigid body, the part is changed from just a CAD model to a part with physical characteristics. Rigid bodies have weight, center of mass and inertia. These parameters influence the dynamics of the body in the simulation. It is recommended to assign a material to the solid body, so that a realistic weight can be calculated.

Rigid bodies react to the acceleration due to gravity, which can be defined along any axis in MCD. Usually, acceleration due to gravity is defined in the negative z or y directions. The direction of gravitational acceleration can be changed in `File` -> `Preferences` -> `Mechatronics Concept Designer` -> `General` -> `Acceleration due to Gravity`.

Collision Bodies

Rigid bodies without collision bodies do not collide when they come in contact. Collision is only possible when both objects have collision bodies. Without collision bodies, objects will go right through each other. An object with a collision body, however, will collide with another object that also has a collision body.

Collision is a property which is independent of whether the part is a rigid body or not i.e., a collision object that has no assigned rigid body status will still collide with other objects that have a collision body.

The collision shape of a collision body can be defined in many ways. It is best to choose simple shapes (box, cylinder, etc.) as a collision shape.

Collision mesh

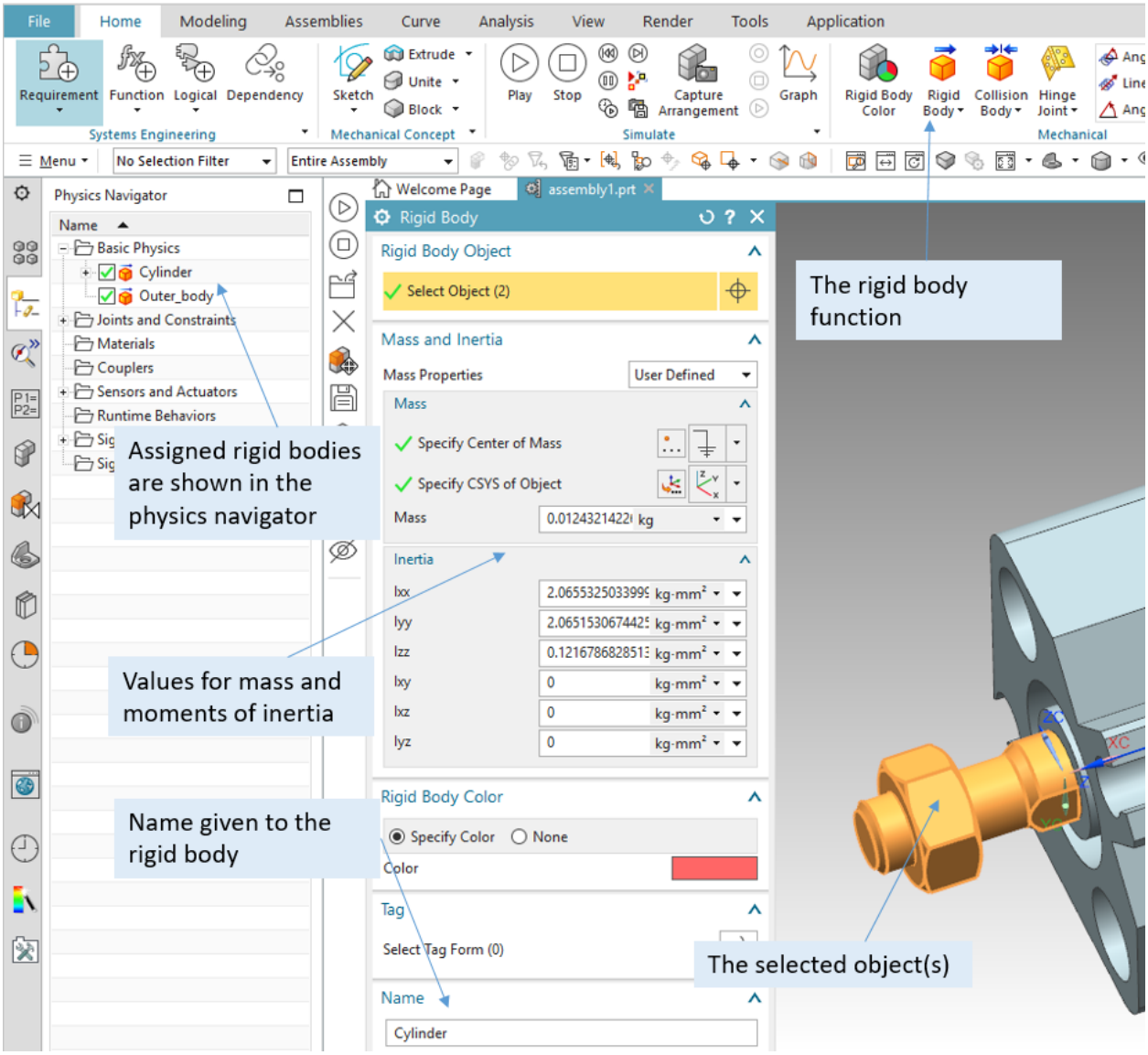


Fig. 16.3: Assigning a rigid body to a part. One rigid body can consist of multiple parts connected to each other that will always undergo the same motion together.

The option "mesh" offers the highest complexity. It allows for a more realistic simulation of the real object, but requires a high simulation performance. The `Convex Factor` sets the resolution of the mesh. The higher the factor, the more detailed the collision mesh. Collision shapes that consist of more than 300 triangles will automatically set a Siemens NX warning pointing to potential performance losses.

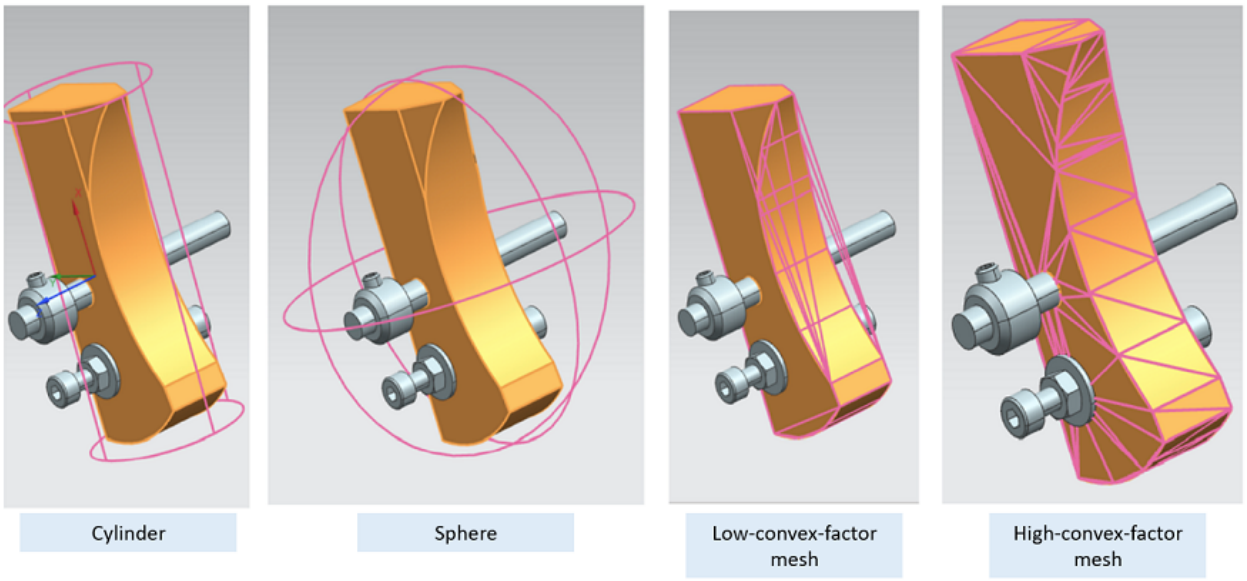


Fig. 16.4: Different ways to set a collision body; the pink area is the active collision area that will collide with other collision objects.

A decision must be made whether an added complexity to the collision shape serves the purpose of the simulation.

Category

The category is a number that can be assigned to a collision body to specify collision possibilities with other collision bodies. Collision bodies only collide within the same categories and with collision bodies in the category 0. Category 0 is an exception; bodies in the category 0 collide with all other collision bodies, regardless of the object's category. Categories can be used to simulate inductive sensors.

Collision material

The choice of collision material influences the physical coefficients dynamic friction, static friction, rolling friction, and restitution. It is recommended to enter accurate friction values for all parts that will take part in collisions to reflect reality.

In order to create a custom material in NX MCD: Go to the `Physics Navigator`, right click on `Materials` and select `Create Physics` -> `Collision Material`.

Prevent collision

The prevent collision function allows two collision objects to not collide with one another if they come in contact.

Transport surface

Transport surfaces are used to simulate conveyor belts. Siemens NX MCD provides a function that can define any surface in an assembly as a transport surface.

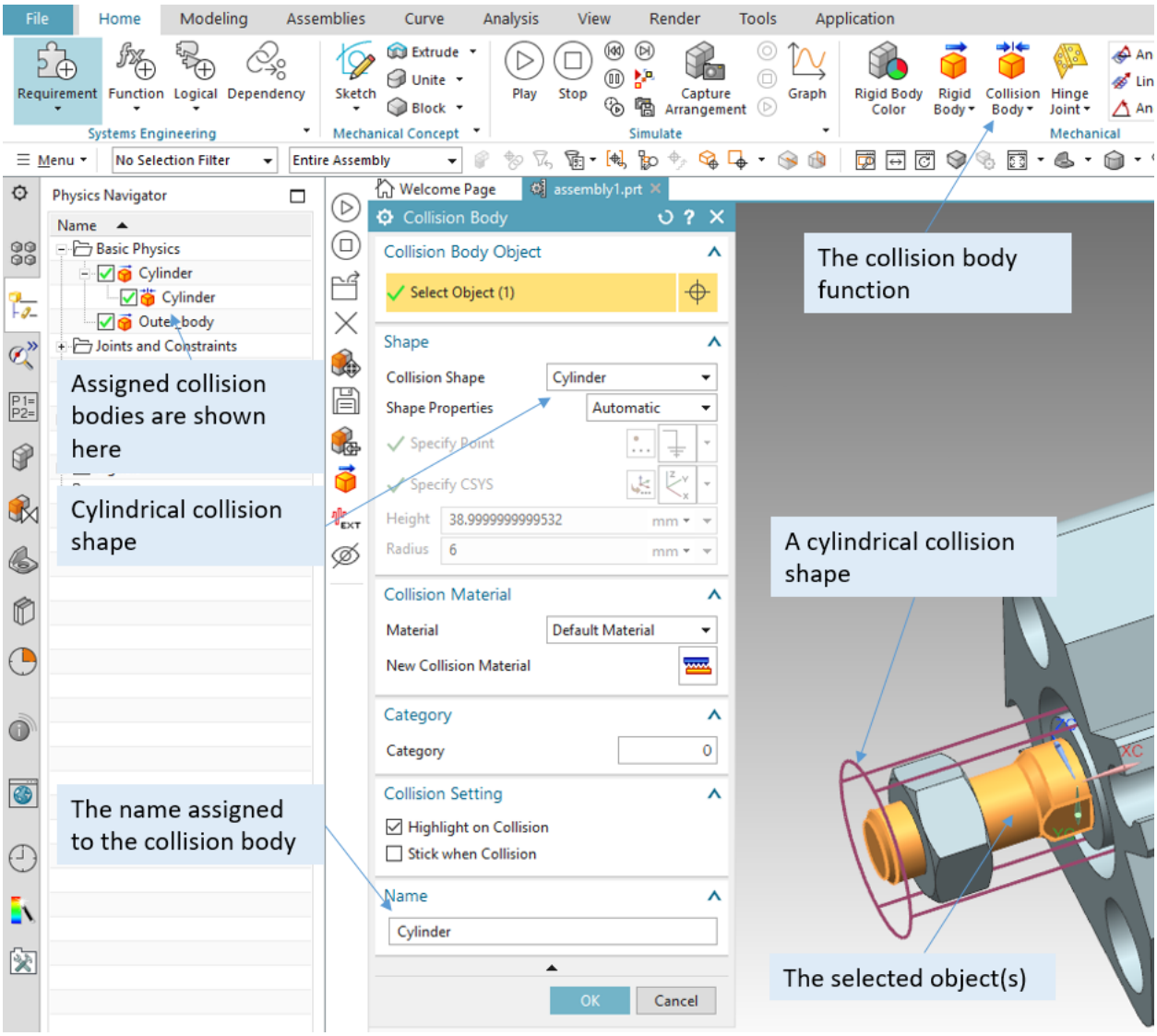


Fig. 16.5: Assigning a collision body

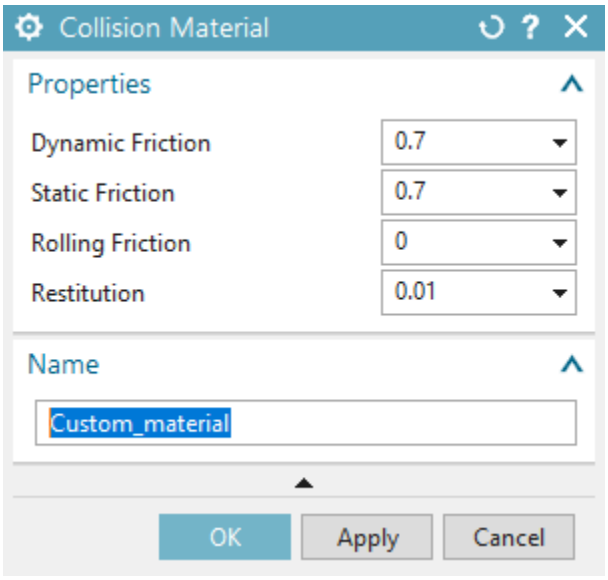


Fig. 16.6: Creating a collision material

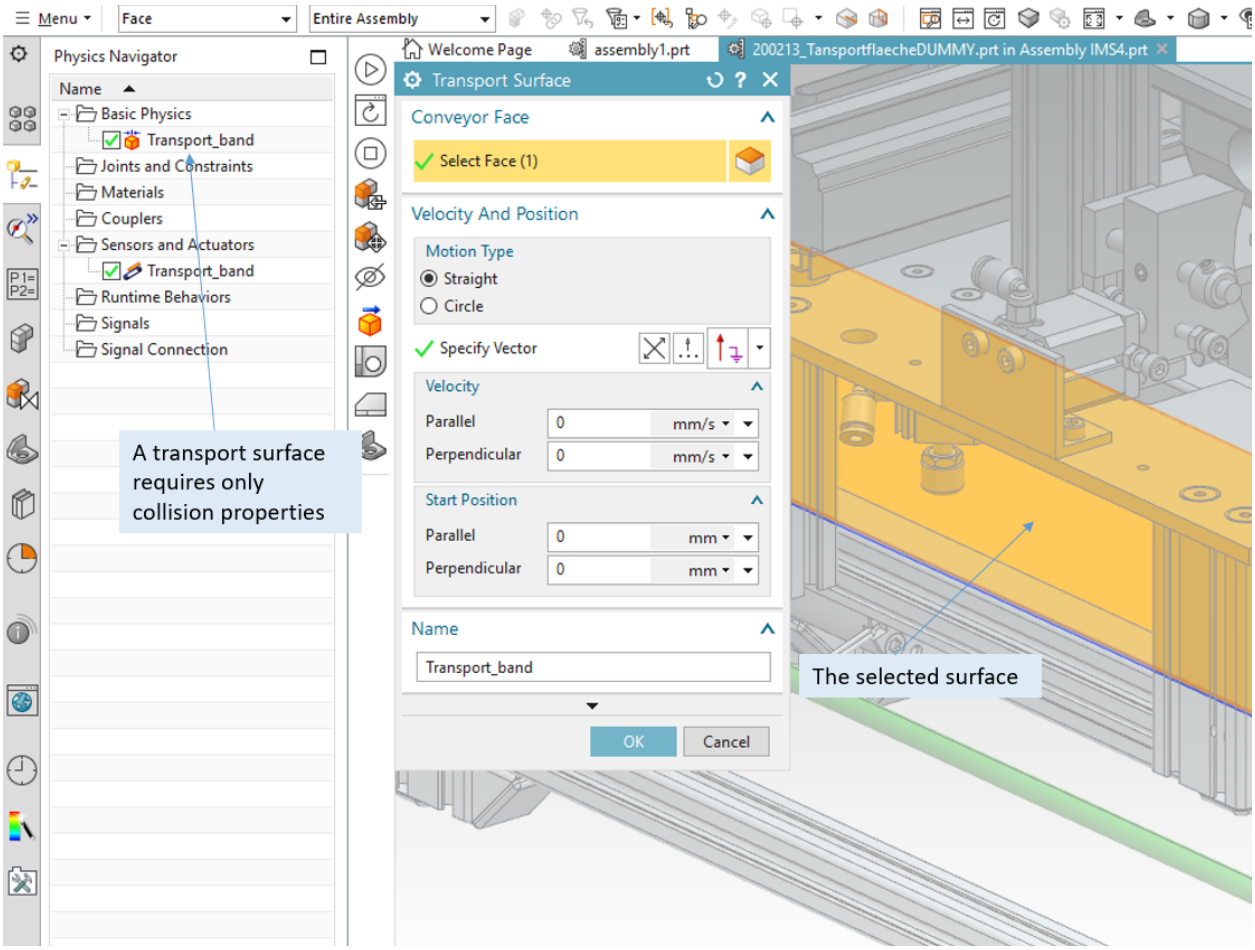


Fig. 16.7: Assigning transport area properties to a surface

Collision Sensors

One of the sensor functions available in Siemens NX MCD is a function that turns objects into collision sensors. When such an object collides with another object (the second object must have collision properties), a boolean signal is set to true. Collision sensors are used to simulate different types of real life sensors (e.g., optical sensors, inductive sensors, etc.). Collision sensors are usually helping elements and are not displayed (blinded out) during the final simulation.

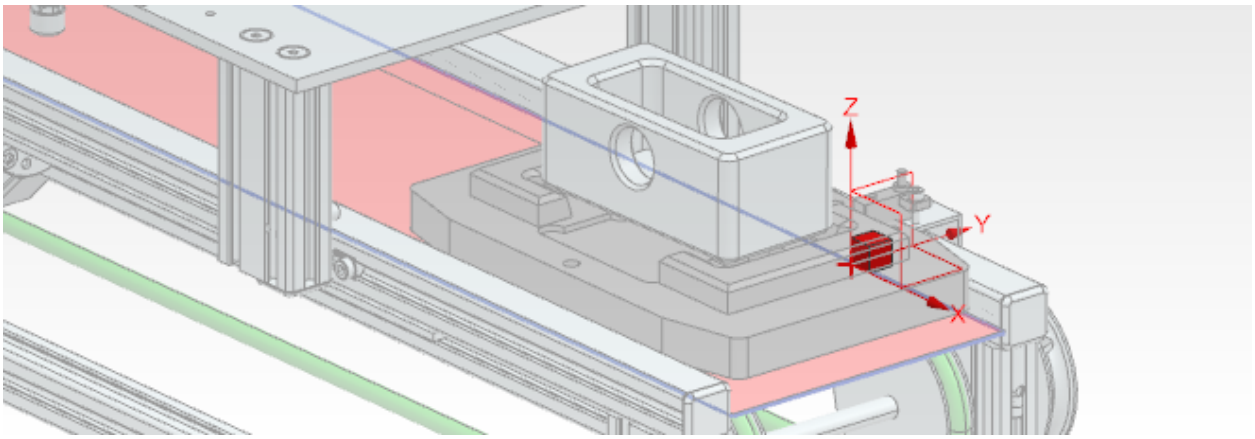


Fig. 16.8: A collision sensor used to simulate the function of an inductive sensor in the assembly. The sensor detects if a part is at the beginning of the conveyor belt.

Joints

Rigid bodies can be fixed, positioned in space or connected to each other by joints of different degrees of freedom. These connections are independent of the assembly constraints from the design application. The joints are available in different degrees of freedom: Hinge (swivel joint f=1), sliding joint (f=1), and cylindrical joint (combined sliding joint and hinge (f=2)).

The following figure shows a sliding joint that is assigned to a rigid body. The rigid body has only one degree of freedom and can be moved only along the axis specified for the sliding joint. Moreover, upper and lower limits are defined along the specified axis which the body cannot exceed during motion.

The following figure shows a hinge joint assigned to a part within an assembly. For a hinge joint (and a sliding joint), an attachment object and a base object should be provided. A base object can be ignored if the base of rotation (or sliding) is not movable in the simulation. In case the base is movable, it must be specified.

Position Controls

The positions of the joints created earlier can be controlled with objects called `Position Controls`, which can be found in the `Electrical` tab. The following figure shows the creation of a position control object to control a sliding joint object.

Position controls can also be created to control the angle of a hinge joint. Position control objects have two important variables: position and speed. Both variables can be manipulated during the simulation using operations and external signals.

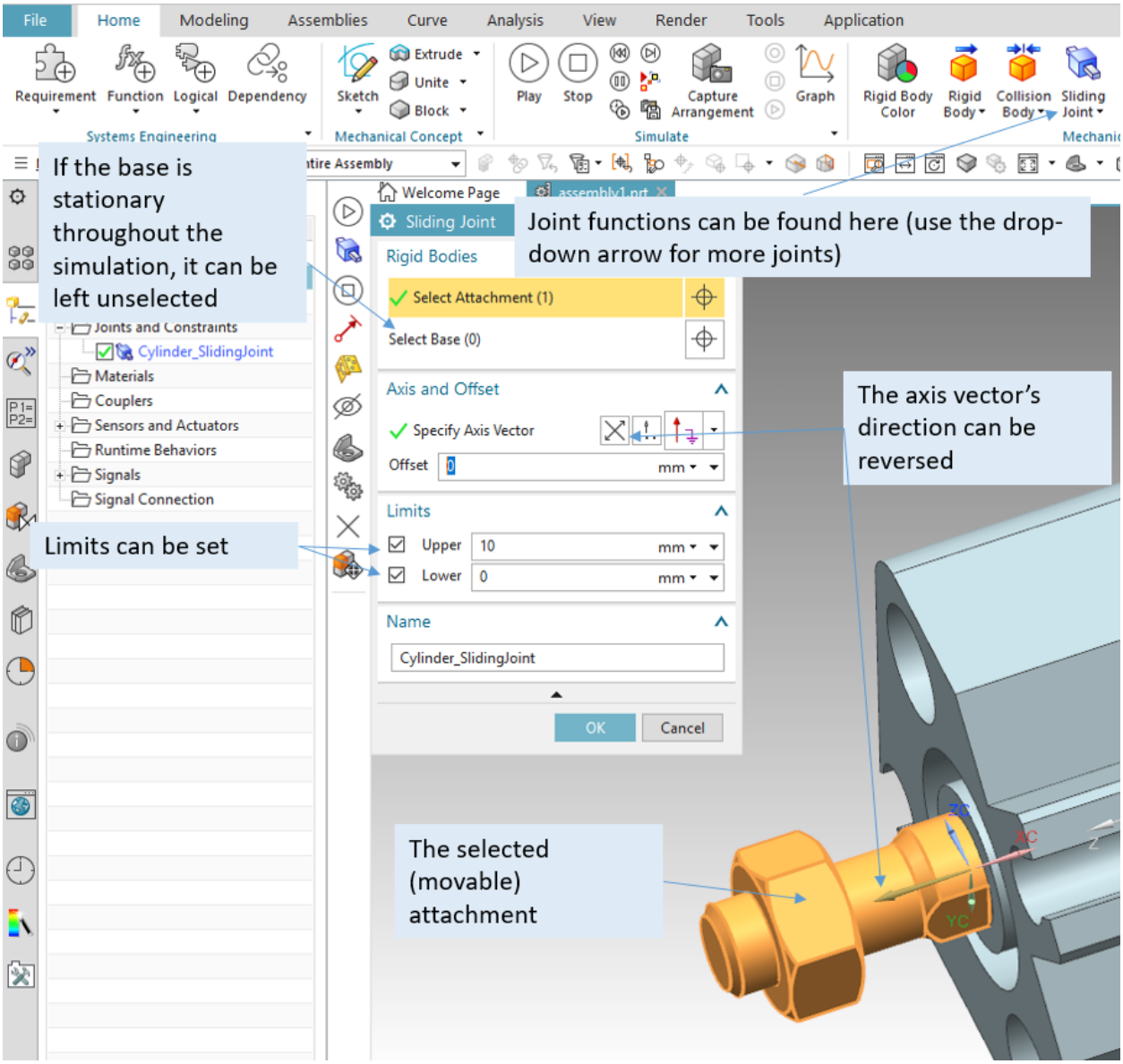


Fig. 16.9: Assigning a sliding joint to an object

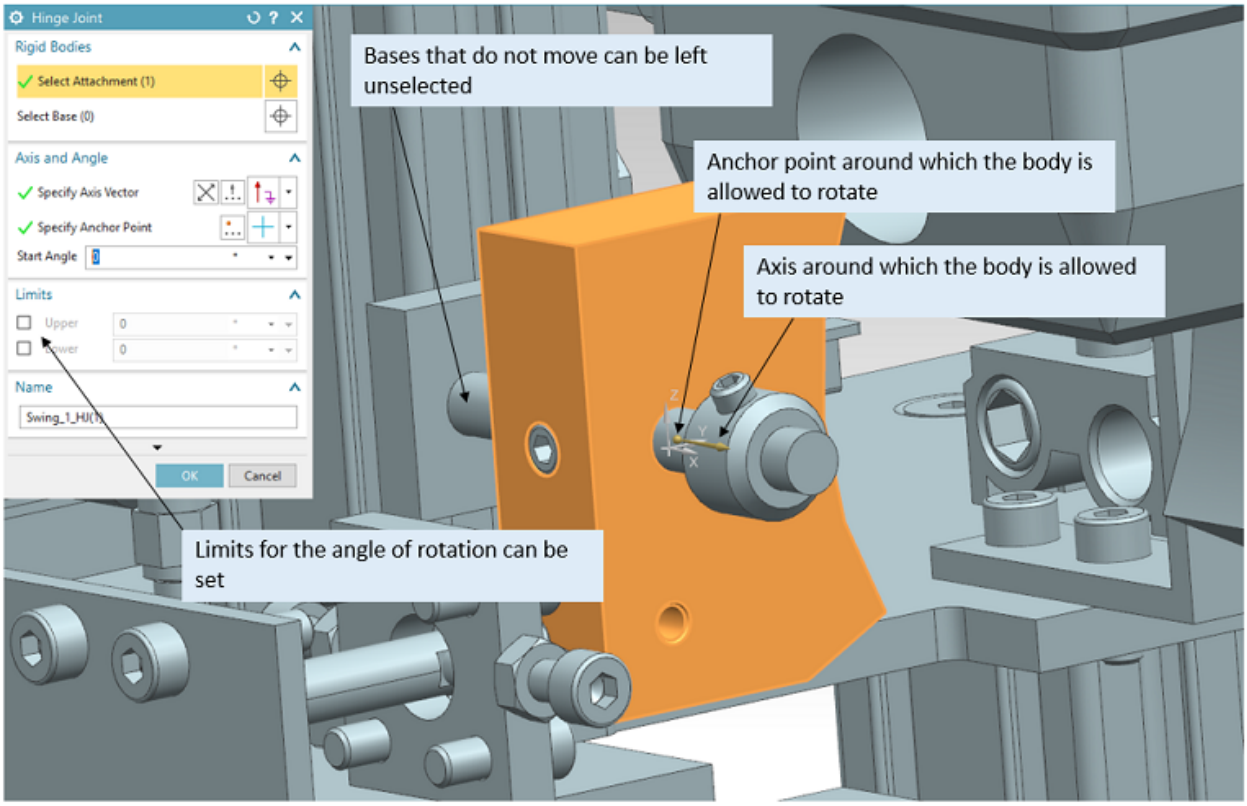


Fig. 16.10: Assigning a hinge joint to an object within an assembly

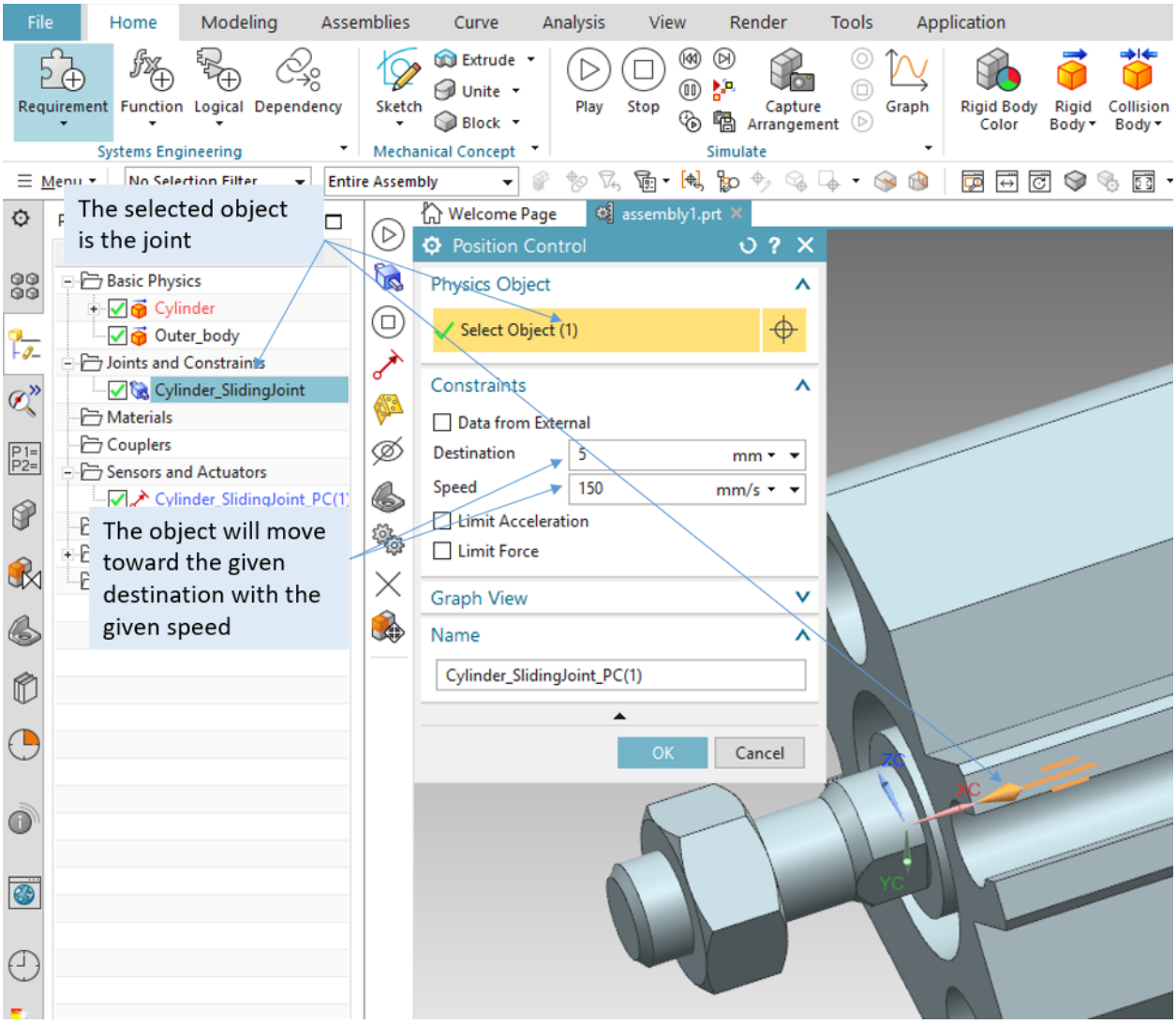


Fig. 16.11: Creating a position control to control the position of an object along its sliding joint

Signals

Signals are objects that allow the communication to programs outside of the simulation. A signal in MCD can have the type bool, int, or double. A signal can be an input signal or an output signal, allowing a two-way communication from and out of MCD.

Note: When communicating with a PLC, an output signal from the PLC's perspective (for example, an motor_on signal) is an input signal from MCD's perspective and should be defined in MCD as an input signal.

Signals can be linked to runtime parameters, in case of a collision sensor for example, where the collision sensor's value (true/false) is the runtime parameter. Signals can also trigger action in the simulation, e.g. setting a motor to turn on. When interacting with a PLC, an MCD signal should be created for each PLC input and output variable. In order to allow auto mapping procedures and prevent confusion, the names of the signals both in the PLC program and in MCD should be identical.

The following figure shows an input signal (input into MCD, output from PLC) that is meant to control the cylinder's position.

Symbol Tables are tables that show related signals together. They are a way to not lose track of signals.

16.3.3 OPC UA and Signal Mapping

To establish a communication over OPC UA, click on `External Signal Configuration` from the `Automation` tab. The following figure shows the configuration of an OPC-UA communication. Two boolean variables are selected which are relevant to the given simulation.

Note: To be able to communicate to a simulated PLC over OPC UA, make sure that OPC UA is configured in the PLC. (See section on TIA Portal basics)

Now that the external signals are imported, it is time to map them to the signals that were created inside MCD. The following figure shows auto mapping of 3 inputs and 2 outputs (from MCD's viewpoint).

16.3.4 Operations

Operations are one way to allow external or internal signals to manipulate the simulation in real time.

The following figure shows an operation that influences the value of the position variable of a position control object. The shown operation extends a cylinder. The condition object is an MCD signal that is mapped to an external signal. Thus, this cylinder can be controlled from outside the simulation while the simulation is running.

16.3.5 A Note on Assembly Hierarchy

The following figure shows a robotic arm in an assembly. The robotic arm itself (`UR3_step`) is an assembly that is made of single parts. The robotic arm is part of a bigger assembly (called `assembly1`). In this example, `assembly1` is the main assembly (the biggest assembly) and the robotic arm is a sub-assembly inside that main assembly.

Assemblies can thus include single parts or other sub-assemblies.

By double clicking on a part/assembly, MCD focuses on that selected part/assembly. Simulations only run with the selected part/assembly. To run a simulation including all parts, the main assembly has to be selected. To run a simulation for only one part/assembly, that part/assembly has to be selected.

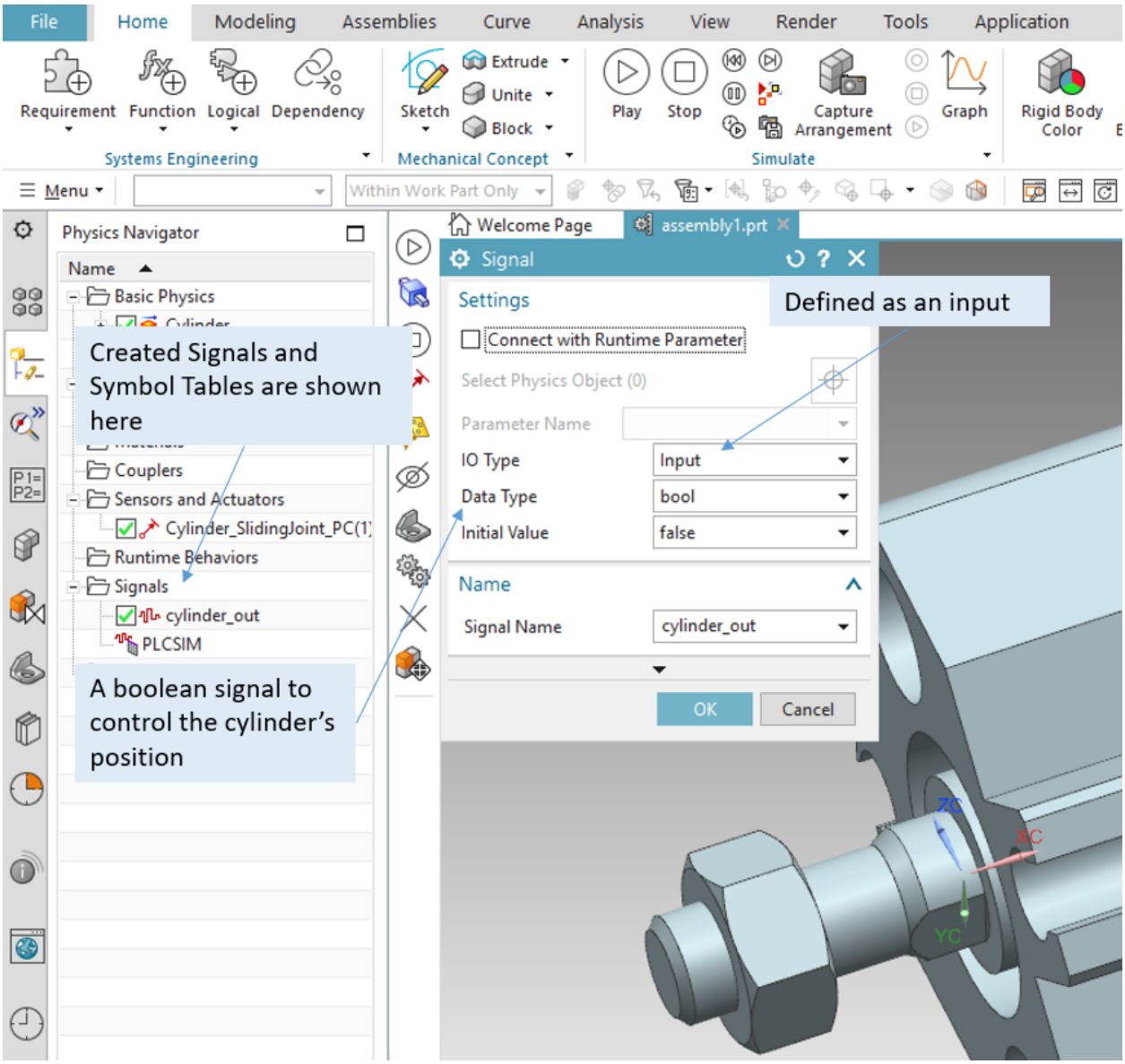


Fig. 16.12: A boolean input signal

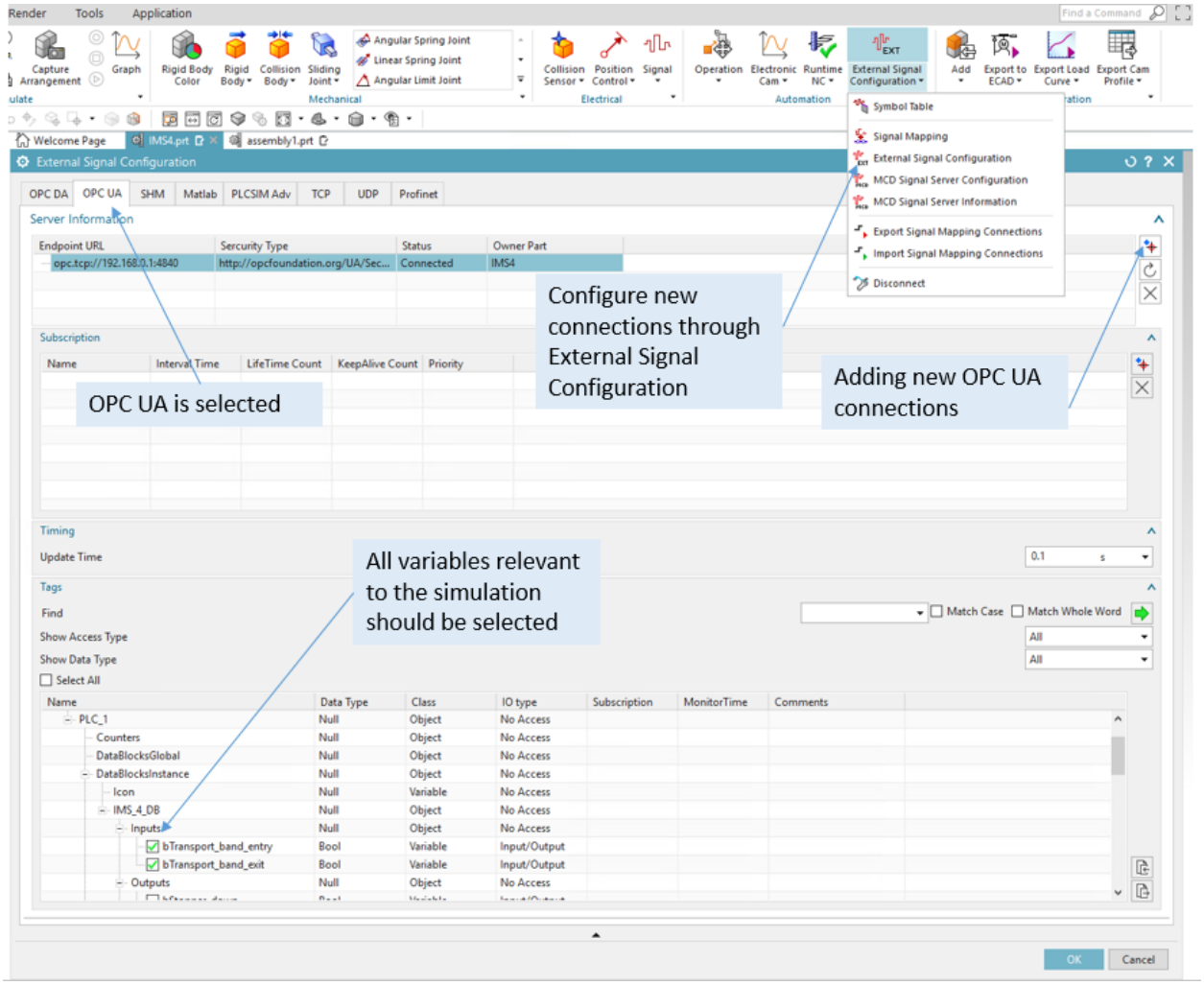


Fig. 16.13: External signal configuration

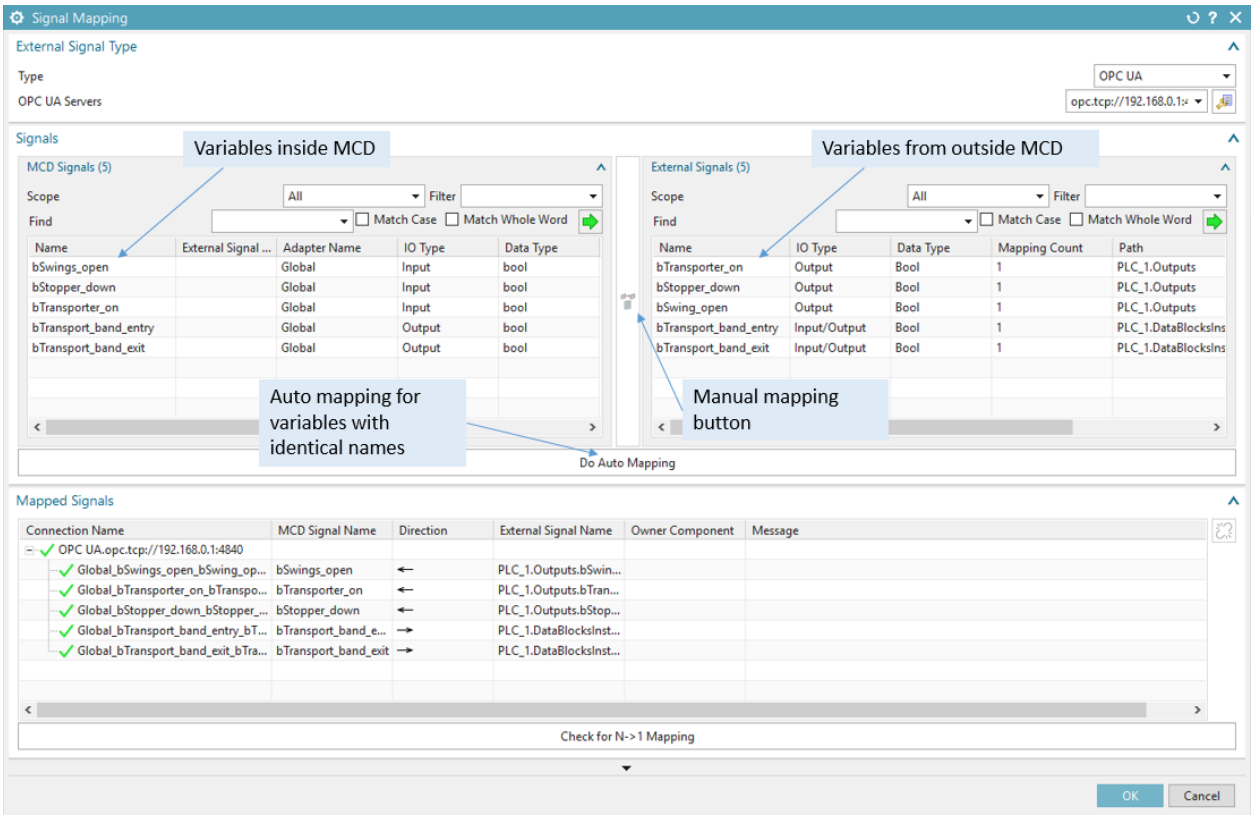


Fig. 16.14: Signal mapping

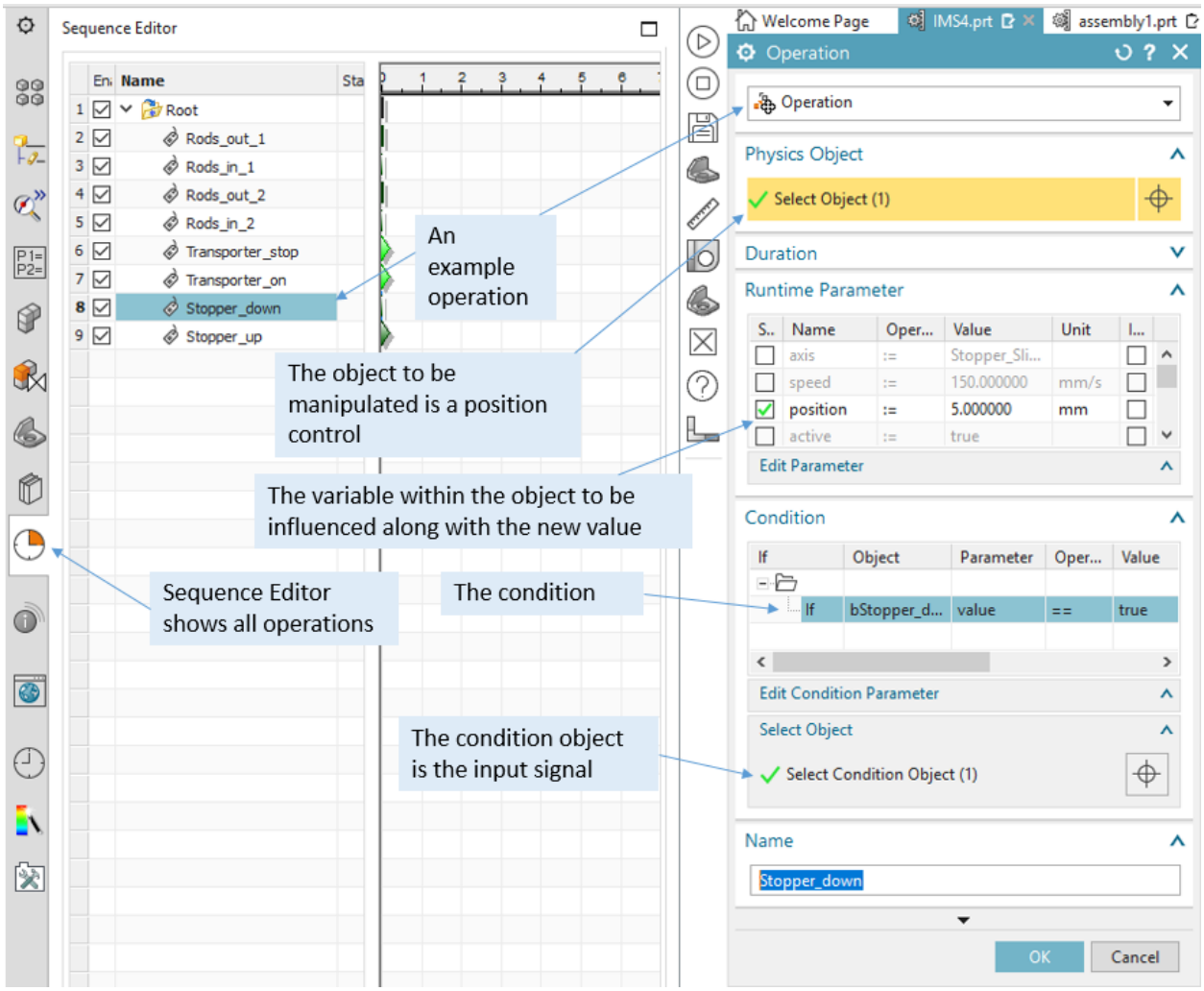


Fig. 16.15: An example of an operation to influence a position control during simulation in real time

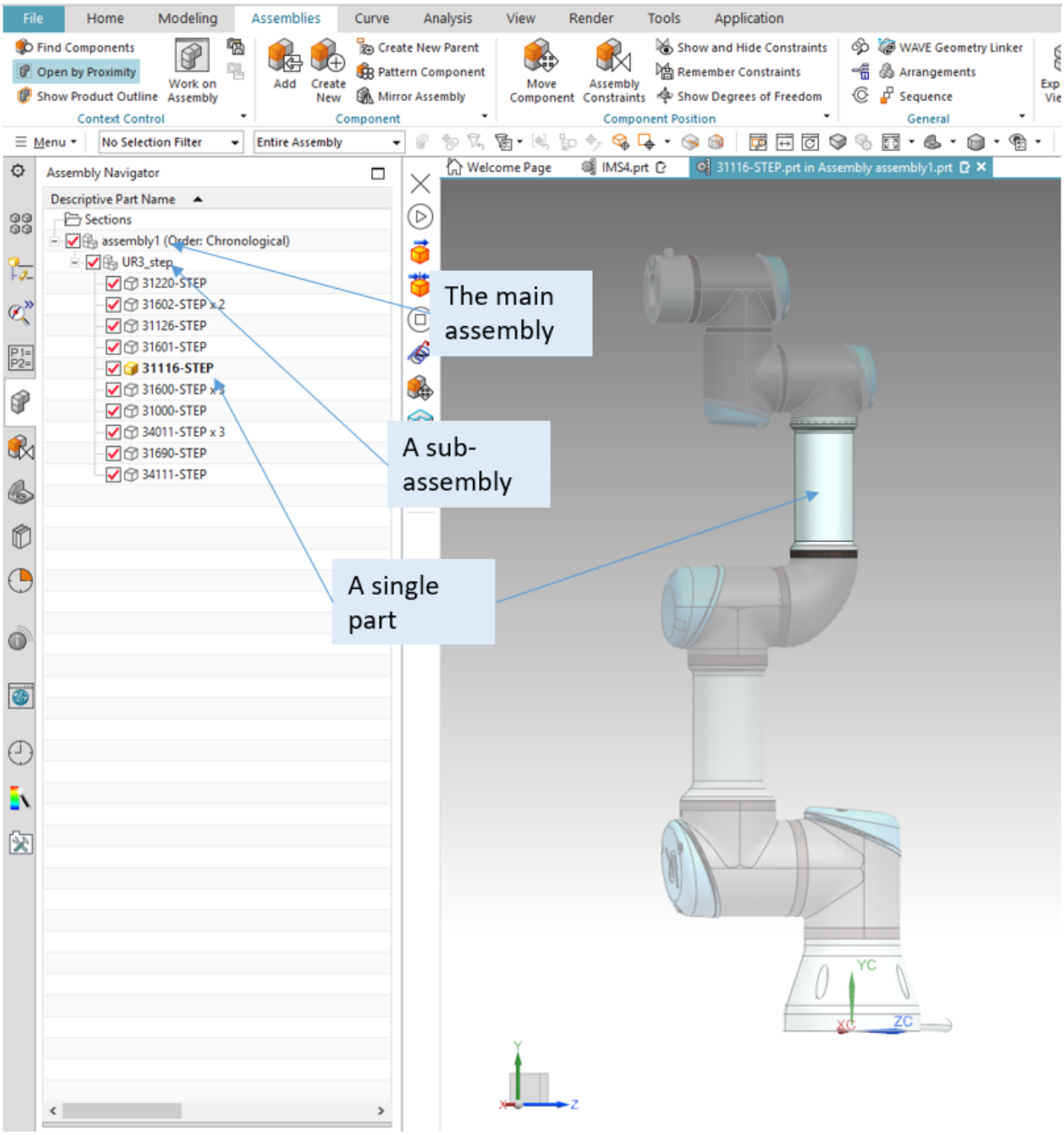
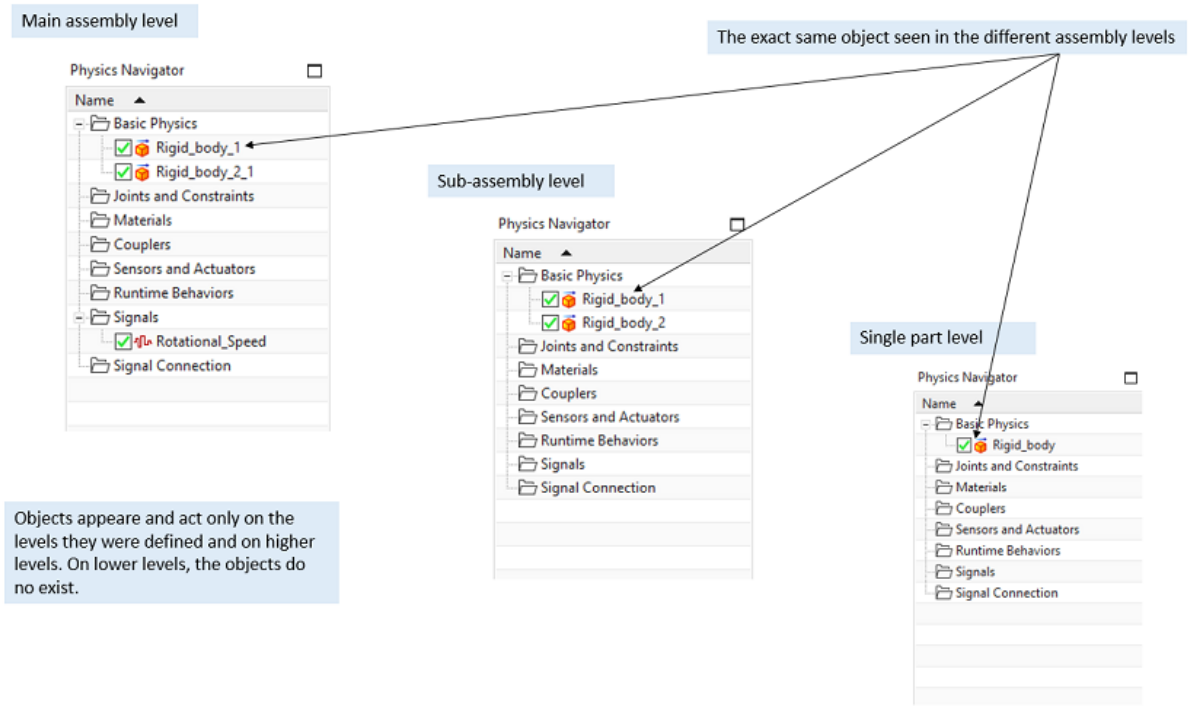


Fig. 16.16: Assemble levels

The location of a single part in the hierarchy can be changed through drag and drop between the assemblies.

The creation of physical properties (rigid bodies, collision bodies, joints, signals etc.) in Mechatronics Concept Designer can be done on any assembly level. Copying parts between files will retain the physical properties created on the copied level and on all lower levels copied along, but not the properties that were created on higher levels.

When a model is copied, the physical properties of all lower levels are copied along.



- Assigning properties on the individual part's level is handy if the part is going to be reused in many other assemblies. This way, the part can retain it's physical properties. This saves the work of reassigning physical properties to the part each time it is copied into a new assembly.
- Assigning the properties on a higher level in the assembly is handy if the part will be used in other assemblies but its properties are not needed anymore. This saves the work of manually deleting all the assigned properties when copying the part into a new assembly.

Because of a bug that causes a communication problem, it is recommended to assign physical properties to parts in a sub-assembly level or on the main assembly level. It is advised to avoid running the simulation on a single part level. Refer to the section on Troubleshooting for more details on communication bugs.

16.4 Summary

Siemens NX MCD offers a tool to create kinematic models using existing CAD models. These kinematic models can be controlled by PLCs (simulated/real), which allows for early verification of control code and a cut in real commissioning time.

CHAPTER

SEVENTEEN

NODE RED BASICS

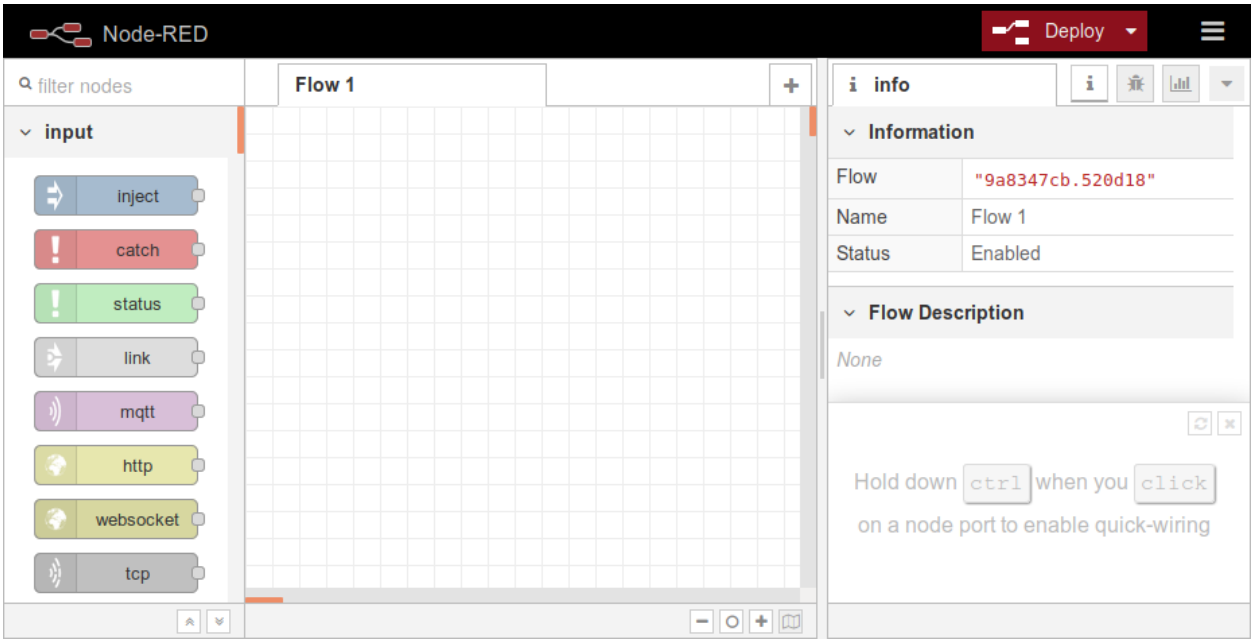


Fig. 17.1: Node-RED UI

Node-RED is a software tool developed by IBM for wiring together hardware devices, APIs and online services as part of the Internet of Things.

17.1 Learning Outcome

17.1.1 Introduction

- You will be able to use node-RED as a middleware to create a dashboard for your devices.

17.1.2 Requirements

- Basic programming skills

17.1.3 What you need

Hardware

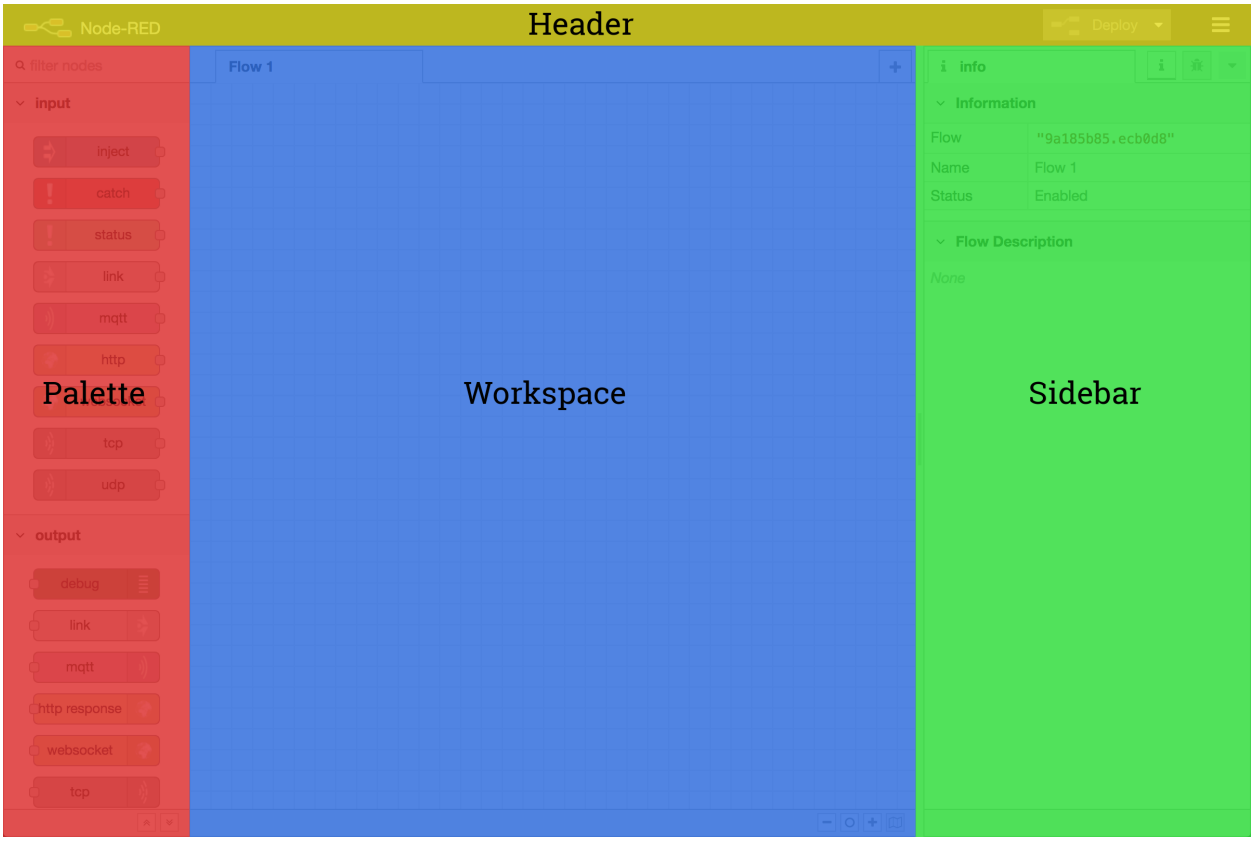
- PC or Laptop
- Arduino UNO with USB Cable

Software

- Node-RED
- Browser (Chrome / Firefox)

17.2 General information

Node-RED is a programming tool for wiring together hardware devices, APIs and online services in new and interesting ways. It provides a browser-based editor that makes it easy to wire together flows using the wide range of nodes in the palette that can be deployed to its runtime in a single-click. (ref: Node-RED)



17.2.1 Start/Stop Commands

To start Node-RED use the following command in Command Prompt (CMD):

```
node-red
```

To stop Node-RED use the following keyboard combination in same CMD window:

```
Ctrl-C
```

Once you start Node-RED, you can then access the Node-RED UI using the IP address. It is usually http://localhost:1880 or http://{[}IP_ADDRESS_OF_PC{]}:1880

(REMEMBER TO REPLACE [IP_ADDRESS_OF_PC] WITH YOUR PCS IP! To find, use **ipconfig**)

17.2.2 About

Node-RED includes the follow concepts (ref: Node-RED) :

- **Node:** Basic building block of a flow. You use these to form the flow of your code.
- **Configuration Node:** Used to store and initialize configuration information. For example providing IP address of PC if you want to receive data via LAN or WiFi.
- **Palette:** Contains the available nodes which can be used to create the flow.
- **Flow:** Nodes interconnected to form a logical data path. It has input, processing functions and output. The flow tab on the top bar shows workspace which can have multiple flow diagrams.
- **Message:** Data which gets passed inside a flow.
- **Context:** Way to store information which can be shared between nodes without using messages that pass through a flow.
- **Subflow:** Creating an abstraction layer for a set of nodes. It helps to group certain nodes and represent them by a single node.
- **Wire:** Connector between nodes.
- **Workspace:** Area where you drag drop nodes and make flows
- **Sidebar:** (on right side) Provides tools and settings to manage and debug the nodes and the messages related to them.

All available nodes are listed on the **left side** of the window. Each node accepts and/or emits a message. By dragging a node into the flow (middle), it is added to the message processing. ‘

On the **right side** the info, debug und dashboard tab can be found. Info displays information about nodes and their documentation and debug shows debug messages.

The red **Deploy** button starts the execution of the flow and applies changes.

Note: When you deploy your flow, it automatically gets saved. You do not need to find a save button. When you open Node-RED next time, your flows will load automatically.

Todo: Please refer the following link for complete information: (https://nodered.org/docs/user-guide/)

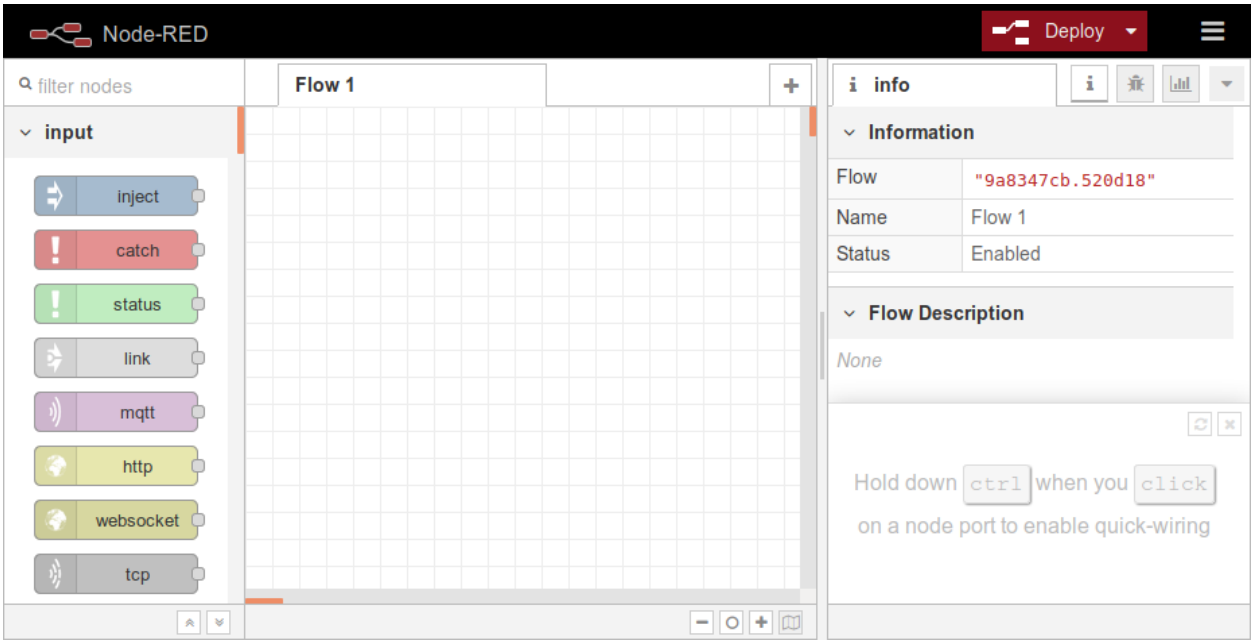


Fig. 17.2: Node-RED UI

Hint: Node-RED is based on Javascript. You need not know in-dept about Javascript but can have basic idea from here: (https://www.w3schools.com/js/default.asp)

17.2.3 Help

You can access help function inside Node-RED. To do so, click on the **book icon** ( ) from the sidebar. Then click on the node you want the information about.

To use this, click on the node -> then click on Book Icon.

17.3 Nodes

To add a node, simply drag them from the panel on the left. There are few core nodes which act as building blocks. These are:

- **Inject:** Pushes fixed data into a flow. You can set repeat interval in its node settings. Supports different types of message types like string, number and timestamp (Source Node)
- **Debug:** Receives all types of messages and displays them in Debug sidebar. Used to analyze messages and find issues. (Sink Node)
- **Function:** To manipulate the message using Javascript as programming language.
- **Change:** To modify message properties as per requirement.
- **Switch:** Switch Statement as in programming terms. Switch message to different output as per given condition.
- **Template:** Creates a text string which fills a specified blank. Example, The temperature today is {{payload}} degrees.

To **configure a node**, double click on it. A panel will open usually on right side.

Hint: Quick-Add dialog provides an easy way to add a node to the workspace by holding the Ctrl or Command key when clicking on the workspace.

17.4 Flows

After adding multiple node, connect the output of nodes start nodes to the input of end nodes. This is how you can create a flow.

Hint: You can comment / disable a flow by selecting all the nodes in that flow and then open **Action List**. You can do this by following key combination CTRL + SHIFT + P. If this doesn't work, change the keyboard shortcuts for **Show Action List** from menu (top right corner).

17.5 Messages

Messages in Node-RED usually have a payload property. This payload consists of the required data. All the messages are in a **Javascript Object** format. A _msgid is added automatically as identifier. This means that the message will be in such format:

```
{
 "_msgid": "12345",
 "payload": "..."
 }
```

To access the useful information of the message, i.e., the payload, you will use the dot(.) command. For example,

```
msg.payload
```

If you have other properties in your message, you can access them as well using the same concept of dot(.) command.

Messages support all the Javascript accepted datatypes. Few of them are mentioned below:

- Boolean - true, false
- Number - eg 0, 123.4
- String - "hello"
- Array - [1,2,3,4]
- Object - { "a": 1, "b": 2}
- Null

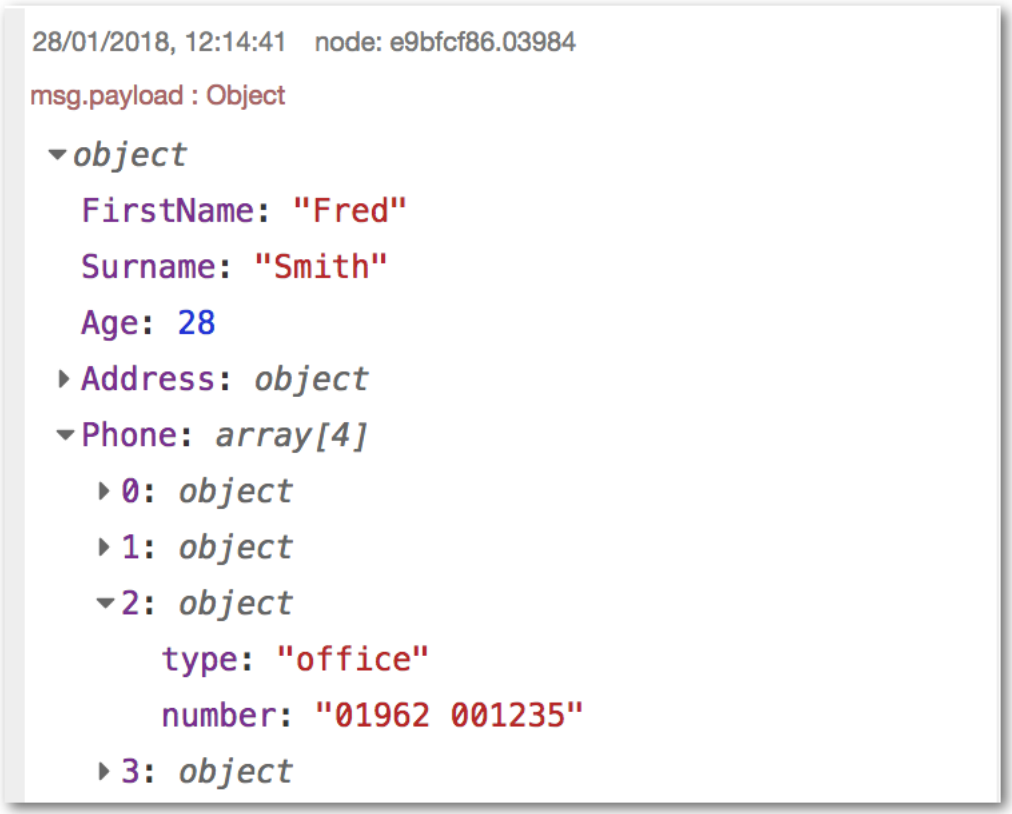


Fig. 17.3: Message structure debug (ref: Node-RED)

17.6 Information and Debug

17.6.1 Information

The Information sidebar shows information about the flows. This includes an outline view of all flows and nodes, as well as details of the current selection.

17.6.2 Debug

The data from the debug node gets printed inside the Debug sidebar. This can be found on the sidebar with icon or a bug.

By default it is selected to **all nodes** but you can set to required nodes if you have multiple debug nodes in your flow.

Hint: You can disable the debug node but clicking on the box on the node. You need not deploy after enabling / disabling only debug node.

17.7 Palette Manager

It is used to install new palettes to your interface. This is like the library manager of Arduino environment.

To access this, click on from top right corner. Then click on **Manage Palette**.

To install a new palette, click on Install and find for the required palette. Once found, click on **Install** button next to that palette.

Hint: Before installing a new palette, it is always good to export all the flows.

To do so, click on from top right corner, then **Export**. Click on **all flows** and then **Download**. Save the file to your PC as backup file.

17.8 Functions

Function node allows writing custom codes using Javascript language. The message is passed inside the function as on object called **msg**. You can access all the properties in the message object using dot(.) command. For example, msg.payload

All the functions have already a **return msg;** statement. This send the message out of the function. You can also pass **null** if your function return nothing, i.e., void function. It is important to always return a msg object. DO NOT RETURN STRINGS OR NUMBERS.

```
// WRONG IMPLEMENTATION

var newMsg = msg.payload.length;  // sends integer value out of function as length is␣
↪number
return newMsg;
```

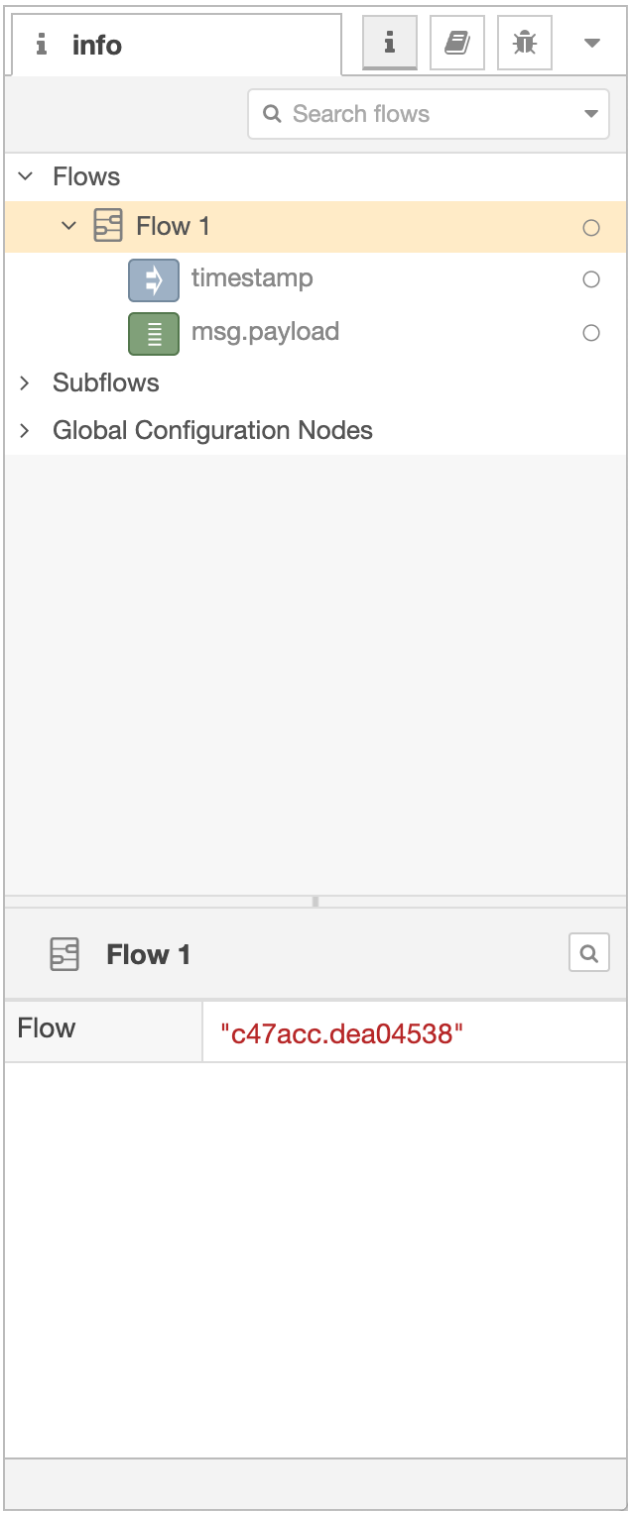


Fig. 17.4: Information (ref: Node-RED)

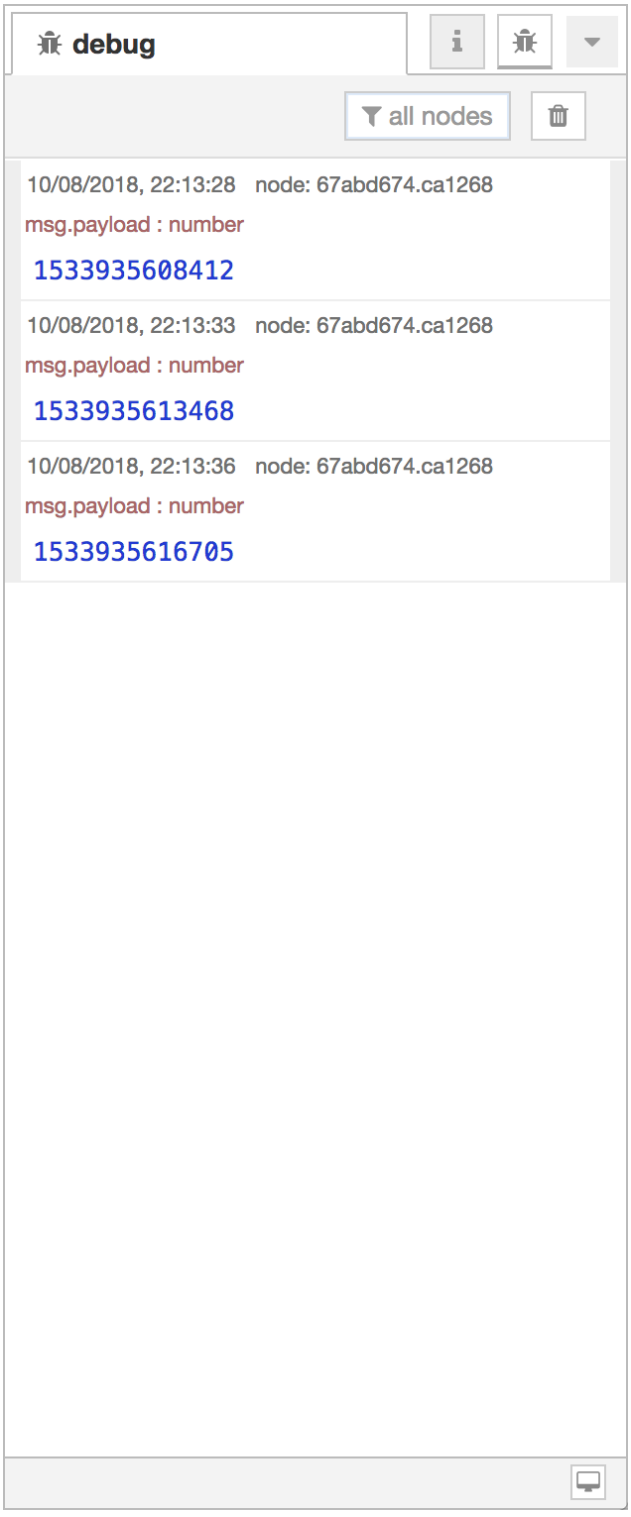


Fig. 17.5: Debug (ref: Node-RED)

```
// CORRECT IMPLEMENTATION

var newMsg =  { payload: msg.payload.length };  // sending length as a property
return newMsg;
```

More information about functions can be found here (https://nodered.org/docs/user-guide/writing-functions)

17.9 Tasks

The tasks below are designed to teach Node-RED UI and its basics

17.9.1 Task 1

Task 1.1

Todo: Create a flow with an **inject** node and **debug** node.

Follow the steps below:

1. Add the nodes on workspace and join them.
2. Deploy the flow.
3. Open Debug window from sidebar.
4. Click on the square button on inject node and observe the output on debug window. You must see some numbers in debug. Check the help to find what these numbers mean.

Task 1.2

Todo: Inject automatically after 2 second. See the complete message in Debug.

Follow the steps below:

1. Open the node configuration of inject and set it to repeat every 2 seconds.
2. Open the node configuration of debug and set it to display complete message.
3. Deploy the changes and observe the Debug window. **You must see the complete message between { } brackets.**

Task 1.3

Todo: Use function to add property to a message.

Follow the steps below:

1. Add a function node in between the inject node and debug node.
2. Add the following code to the function. This function adds a property called **name** to the message and then returns it out from the function.

```
msg.name = "[YOUR NAME]";
return msg;
```

3. Deploy the changes and observe the Debug window. **You must see your name on the debug window.**

17.9.2 Task 2

Note:

You would need the following palettes installed.

- node-red-dashboard (https://flows.nodered.org/node/node-red-dashboard)
- node-red-node-random (https://flows.nodered.org/node/node-red-node-random)

Todo: Use random function to publish values between 15 to 35. It must repeat every 5 seconds. Add a function to convert these values from celsius in fahrenheit. Name the function **Temp Converter**. Use the output of this function to display values on a chart.

The label of chart must be room temperature. Set the value of x-axis to last 10 minutes.

Note: To open dashboard, look for a **bar chart** icon on side panel. Click on it and then find an icon as in image below.

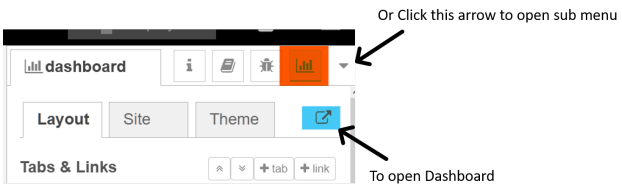


Fig. 17.6: Dashboard Icon

Hint: You will need to add an inject node before the random function generator to repeat values at specified interval.

17.9.3 Task 3

Todo: Create a HTTP endpoint which responds to GET requests and handles the parameters passed in the url.

http://example.com/hello-query?PARAMETER_1=VALUE_1

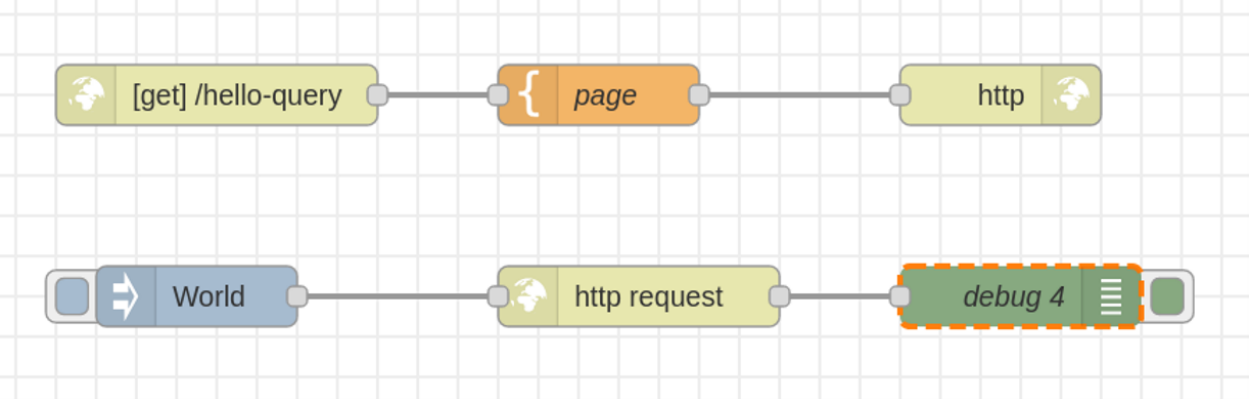


Fig. 17.7: Task flow structure

Tips for First Flow are provided below:

1. **In template node from function section. Enter the following code in the editor.**

```
<html>
    <head></head>
    <body>
        <h1>Hello {{req.query.name}}!</h1>
    </body>
</html>
```

Tips for Second Flow are provided below:

1. In **http request node**. In URL, enter the following and keep the rest setting unchanged.

```
http://localhost:1880/hello-query?name={{{payload}}}
```

or this if the above doesn't work

```
http://127.0.0.1:1880/hello-query?name={{{payload}}}
```

The output must look like this: **Remember to See the blue highlighted text below. You output must be as in blue color.**

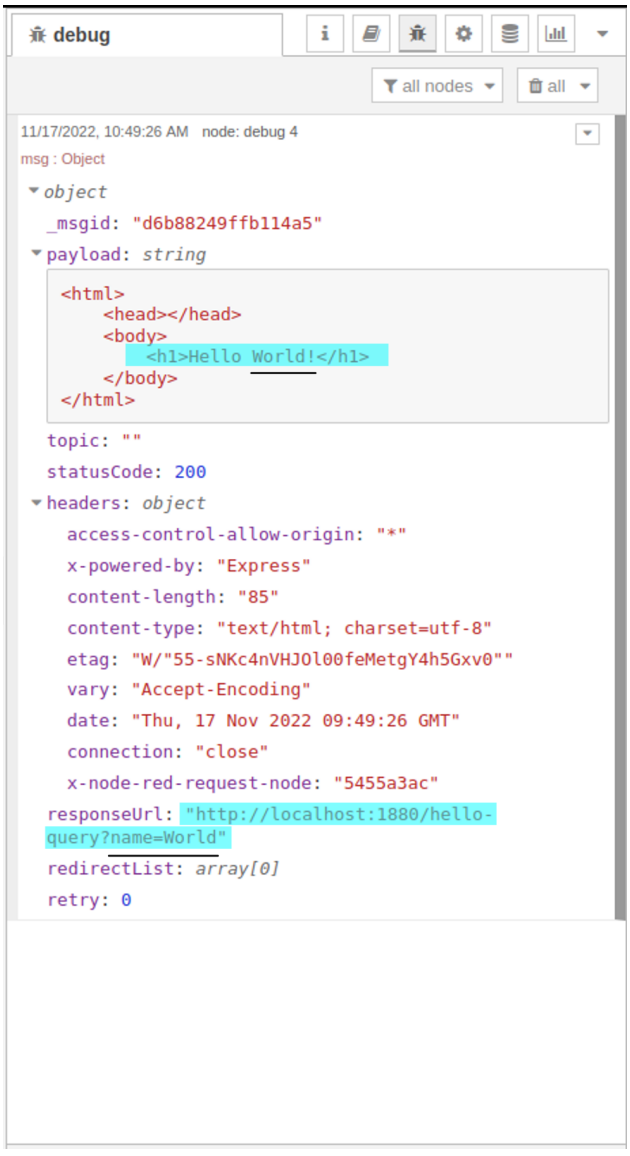


Fig. 17.8: Task output

17.9.4 Task 4

Todo: Create a user input **form** on dashboard. The form should take the following inputs from user: Name, Email-ID and Favorite number.

Use this form to save data to a .txt file.

Use the Node-RED help **book icon** ( ).

CHAPTER

EIGHTEEN

VISUAL STUDIO CODE BASICS

Fig. 18.1: VS Code by Microsoft

Visual Studio Code is a source code editor. It supports various programming languages and can be combined with compilers to work as a package integrated development environment IDE.

18.1 Learning Outcome

18.1.1 Introduction

- You will be able to use Visual Studio Code for code developed and version control.

18.1.2 Requirements

- Basic programming skills

18.1.3 What you need

Hardware

- PC or Laptop

Software

- Git Software
- Visual Studio Code
- Browser (Chrome / Firefox)

References

1. VS Code https://code.visualstudio.com/
2. GitLab https://git.fh-aachen.de/
3. GitLab Extension for VS Code https://marketplace.visualstudio.com/items?itemName=GitLab.gitlab-workflow

18.2 General information

Visual Studio Code is a source code editor. It supports various programming languages and can be combined with compilers to work as a package integrated development environment IDE.

VS Code has IntelliSense which allows word based completions. This you can write code syntax without typos.

You also get access to various extensions. These enhance the working of VS Code and also adds new features as per requirements.

Fig. 18.2: VS Code by Microsoft

18.2.1 Interface

The UI is divided into five areas (ref: VS Code):

- Editor - The main area to edit your files. You can open as many editors as you like side by side vertically and horizontally.
- Activity Bar - Located on the far left-hand side, this lets you switch between views and gives you additional context-specific indicators, like the number of outgoing changes when Git is enabled.
- Side Bar - Contains different views like the Explorer to assist you while working on your project.
- Editor Groups - You can open multiple files and can split the screen to work with two or more files simultaneously.
- Panels - You can display different panels below the editor region for output or debug information, errors and warnings, or an integrated terminal. Panel can also be moved to the right for more vertical space.
- Status Bar - Information about the opened project and the files you edit.

Activity Bar: This includes the followings commands to manage open project: Explorer, Search, Source Control, Run & Debug and Extensions. You can switch between these to handle the project as per requirements.

Side Bar: In this area, you can create, delete and modify files of your project. You can also drag & drop files to organize content.

Note: When creating new files, PLEASE add an extension to the file. If no extension is provided, the file also has no extension!

Status Bar: It is divided in mainly two parts.

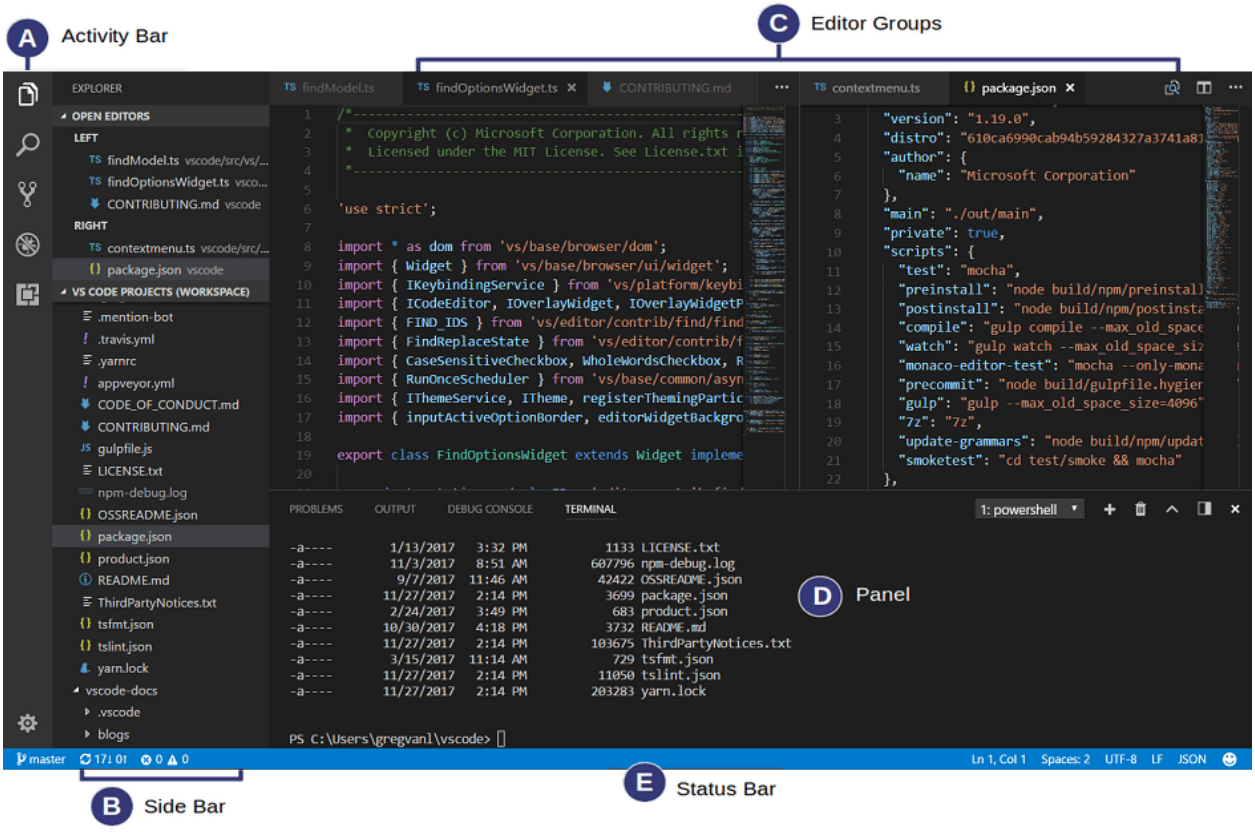


Fig. 18.3: VS Code UI (ref: VS Code)

- The **left part** consists of Project related stuff. Here you see **master** which is the working branch of the project. When you change the branch, it automatically sets it to default upload branch. The refresh icon synchronizes the content between server and local stored code. The icons near that shows information about the tips and errors in all opened files of the project.
- The **right part** of the status bar shows properties related to the editor and user selected settings which apply to editor window.

Panels: The panels include Problems, Output, Debug console and Terminal. These can be opened from top **Menu bar** under **View**.

Note: The garbage can icon is used to kill the current Terminal session. If you kill the last terminal session, the Terminal tab will close automatically.

18.2.2 Command Palette

VS Code allows direct execution of commands for extensions. This can be done using Command Palette. Command Palette can be accessed from View -> Command Palette or using key combination CTRL + Shift + P.

To execute a command for an extension, always check for angle bracket symbol (>). Following the symbol, you can type the name of the extension and then you will see the different options / commands available.

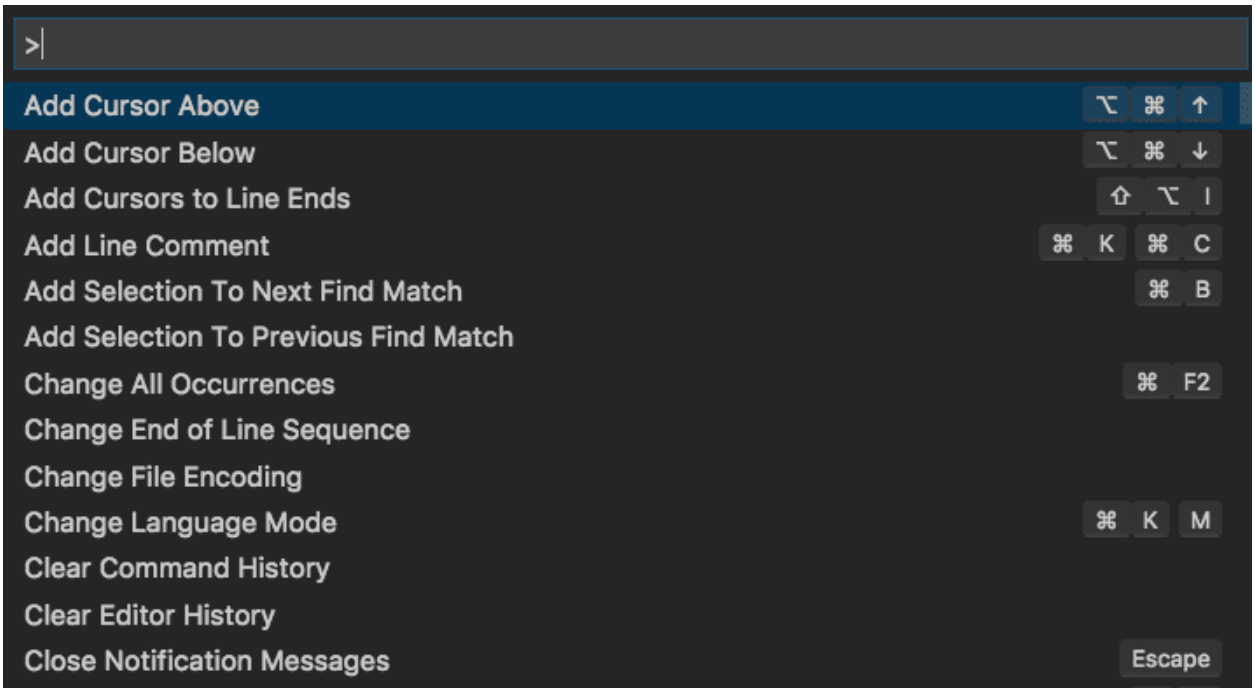


Fig. 18.4: VS Code Command Palette (ref: VS Code)

18.2.3 Saving Settings

When you modify settings, you will usually see 2 options, **User Settings** & **Workspace Settings**.

In **user settings**, the settings are saved for the user of the editor. These settings will be default for all projects opened afterwards.

In **workspace settings**, the settings are saved for only that specific instance of project. A new file gets created in the same project folder which can be shared with others so that everyone has the same settings while working on the project. For example, using tabs instead of spaces for indentation of code.

18.3 Gitlab Extension

Warning: Before starting, please confirm you have access to GitLab server of University. You can check that by logging on the following website using your fh-kennung nummer (AB1234S) and passwort: https://git.fh-aachen.de/

Todo:

Please install the following softwares:

- Git Version Control (https://git-scm.com/downloads)

Follow the steps below to setup GitLab Account with VS Code.

From the Top Right corner, click on your **profile image** then **Preferences**. You will be redirected to new page.

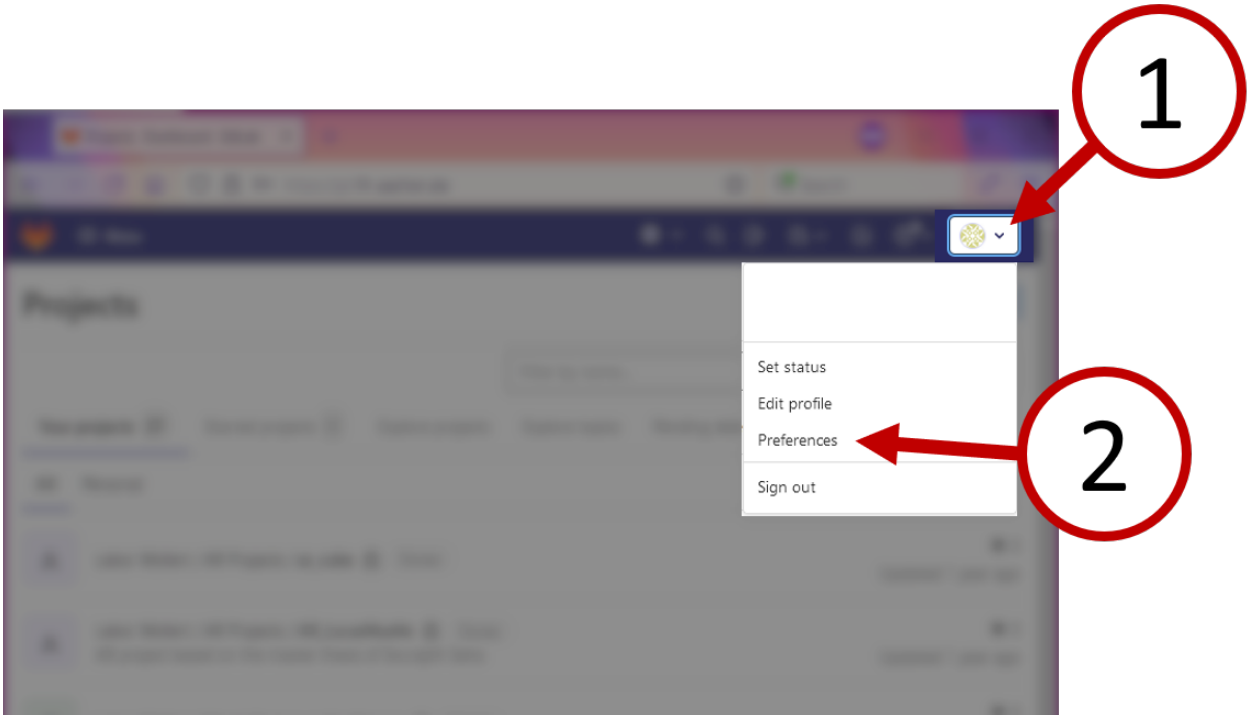


Fig. 18.5: Step 1

On this page, select **Access Token** from the left menu. On the new page, enter a name for token, for example: vs-code_laptop. You can leave the **Expiration date** empty. select all checkboxes under **Select scopes**. Once done, click **Create personal access token**.

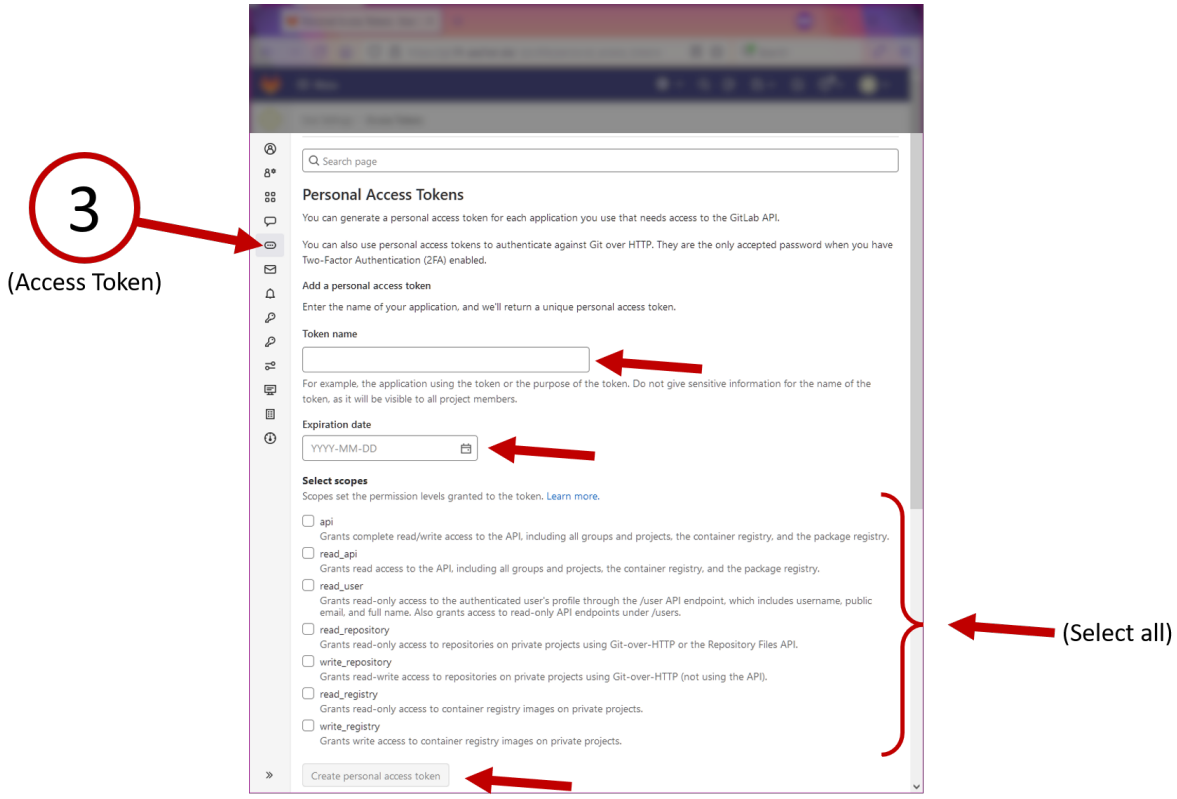


Fig. 18.6: Step 2

When done, you will receive a **Personal Access Token**. Copy and save that for future use.

Inside VS Code, install the extension for GitLab (https://marketplace.visualstudio.com/items?itemName=GitLab.gitlab-workflow). Head to the extension and click on **Add Account**.

On top, you will see a bar requesting for a URL. Use **https://git.fh-aachen.de/**. Then it will ask for access token. Enter the token created from GitLab.

When successful, you will see a confirmation on bottom right corner. If not already done, Open terminal inside VS Code from Terminal -> New Terminal.

For windows users -> Enter **git config --global core.editor "notepad"** in it.

Enter **git config --global --edit** in it. A file will open in text editor. In **name**, enter your **full name** and in **email**, enter your fh-email id example firstname.lastname@alumni.fh-aachen.de

Now you can create / clone new repositories.

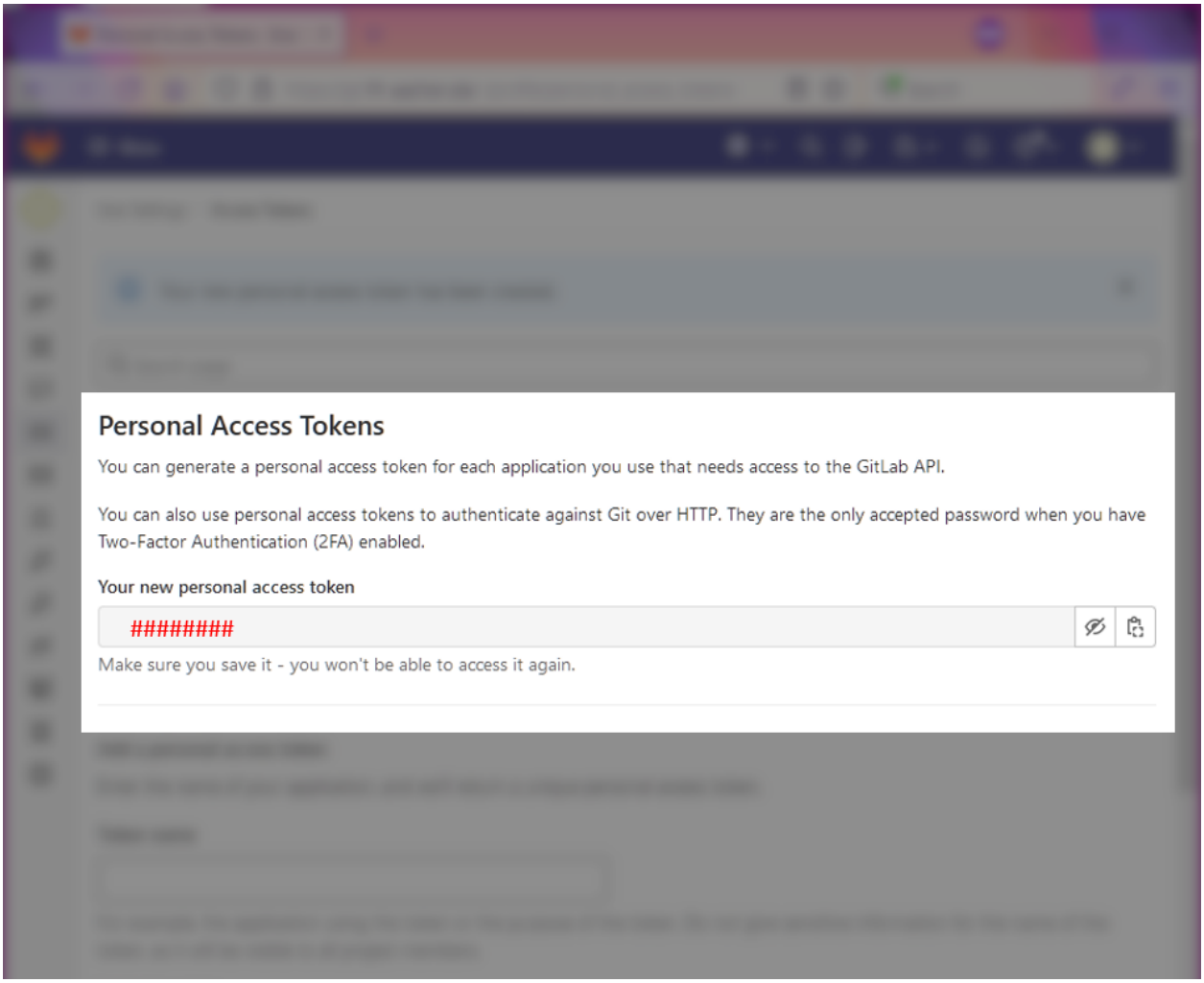


Fig. 18.7: Step 3

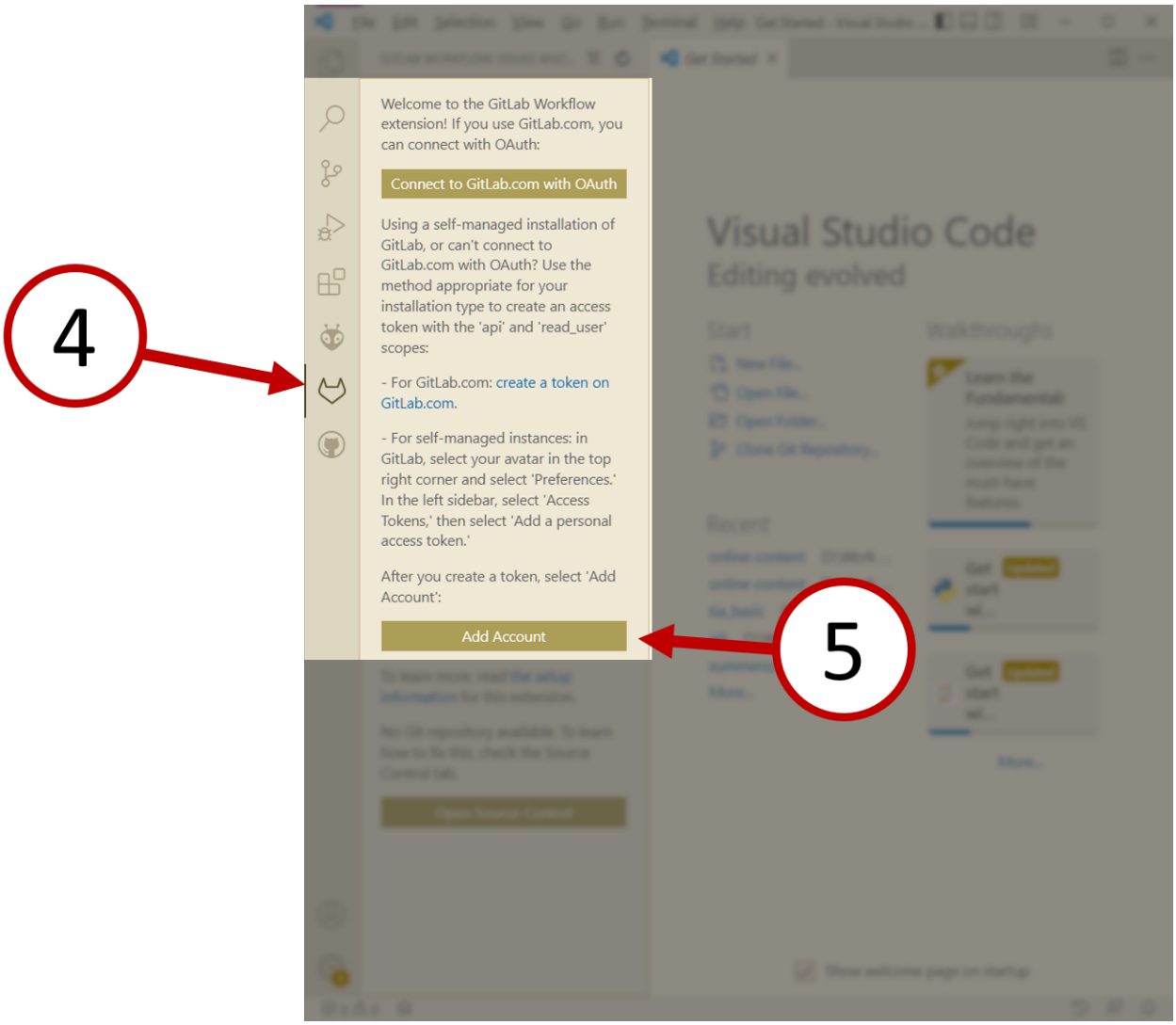


Fig. 18.8: Step 4

Fig. 18.9: Step 5

> **Warning:** When cloning a repository, please use **https://** URL. The git@git is used when setup done using SSH.

After your repository is setup successfully, you can find these options in **Source Control** option from left menu in VS Code.

18.4 Arduino Extension

Todo: Download and install the Arduino IDE to program your Arduino Uno.

Download Link: https://downloads.arduino.cc/arduino-1.8.19-windows.zip

Version: 1.8.X

Note: The Arduino extension for VS Code will automatically detect installed Arduino IDE. If in case it doesn't happen, follow the steps below:

- Open **Extensions** from Activity Bar. Search Arduino extension.
- Click on **small gear** (manage menu) next to it. It will open a new menu.
- Click on **Extension Settings**. New tab will open.
- Scroll down and find **Arduino Path**
- Enter the complete installation address of your Arduino IDE. C:Program Files...

To use the extension, follow the steps below:

- Open VS Code. Click on **Open Folder** from the startup interface. Open the folder **Documents -> Arduino**. If the folder is not found inside Documents, create a new one named **Arduino**. Open the Arduino folder.
- Inside Arduino folder, click on **New Folder** from top bar and name the folder. PLEASE DO NOT INCLUDE SPACES IN NAME. USE UNDERSCORES (_). Example, my_first_code
- Open the new created folder. Once inside the new create folder and click on **Select Folder** from bottom.
- VS Code will open with the selected folder.
- Open **Command Palette -> Arduino Initialize**. A pop-up will ask for file name.
- Enter the file name. PREFER USING SAME NAME AS FOLDER NAME. Press enter once done.
- Select the required board. Enter **UNO AVR board** from the selection list.
- Once done, you will find a project file with extension **.ino** and a folder **.vscode** in your Side Bar.

The following commands have to be used after initial setup:

- **Board Config**: To change the development board.
- **Select Serial Port**: To change the serial port of connected device.
- **Verify**: To compile the code and check for syntax errors. This only compiles the code, does not upload it.
- **Upload**: To build the code and upload it to connected board.

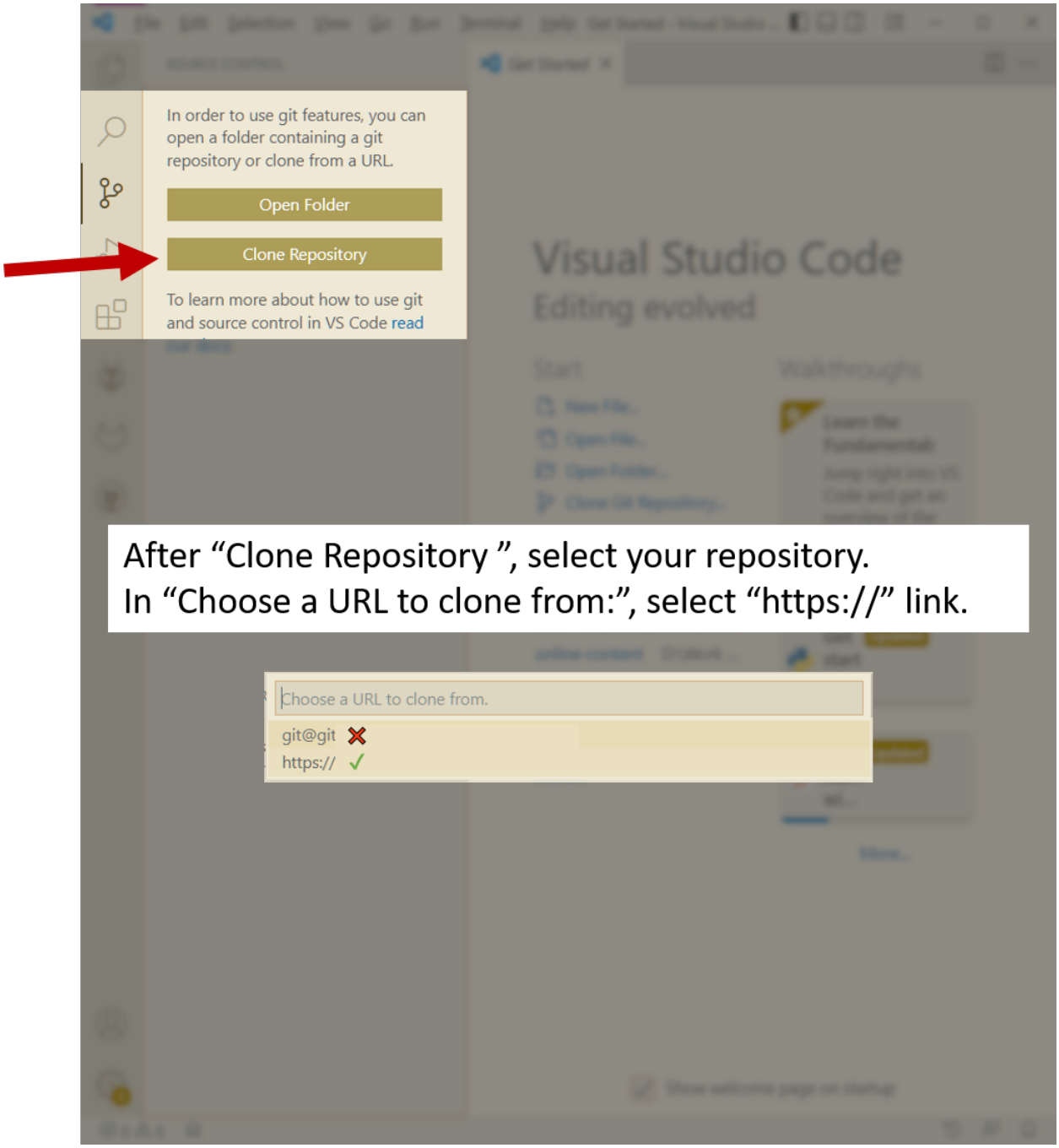


Fig. 18.10: Step 6

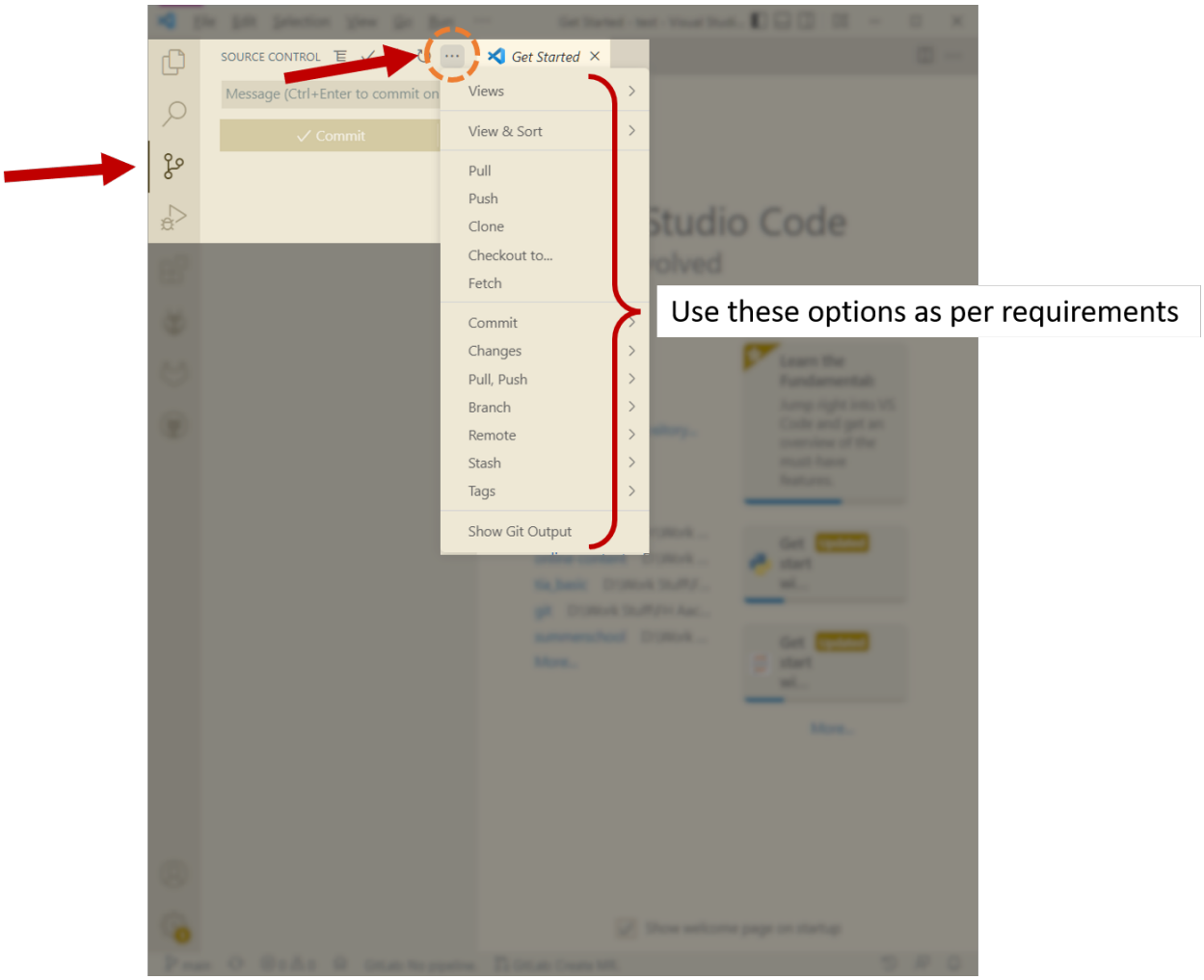


Fig. 18.11: Step 7

- **Library Manager**: Allows management and download of external libraries.
- **Open Serial Monitor**: To open serial monitor to see device logs and user based serial print statements.

Details about rest commands can be found here (https://github.com/microsoft/vscode-arduino#commands)

CHAPTER

NINETEEN

INTRODUCTION TO VERSION CONTROL WITH GIT

Fig. 19.1: GIT SCM

19.1 Learning Outcome

19.1.1 Introduction

- You will be able to use concepts of Git to manage your coding projects.

19.1.2 Requirements

- Basic programming skills

19.1.3 What you need

Hardware

- PC or Laptop

Software

- Git Software
- Visual Studio Code
- Browser (Chrome / Firefox)

References

1. VS Code https://code.visualstudio.com/
2. GitLab https://git.fh-aachen.de/
3. GitLab Extension for VS Code https://marketplace.visualstudio.com/items?itemName=GitLab.gitlab-workflow
4. Git https://git-scm.com/
5. Professional Git by Brent Laster (available online in University Library)
6. Git Book V2 https://git-scm.com/book/en/v2
7. EBC-Tutorials https://github.com/RWTH-EBC/EBC-Tutorials/tree/master/EBC-Git-101
8. Semantic Versioning 2.0.0 https://semver.org/
9. Semantic Versioning Wikipedia https://de.wikipedia.org/wiki/Version_(Software)

19.2 Introduction

Download Link: https://git-scm.com/downloads

A version control is a kind of system which allows you to keep track of the changes that have been made to a code over a duration of time. This means that you can, at any given point in time, revert back to the older versions of the code you are working on.

Git is a popular VC software. Git works on Distributed Version Control Systems. This system provides everyone the copy of all files and allows user to edit them locally. The user can then work on the files locally and then upload the files to the server. The advantage of DVCS is that, if a server crashes, a local user can upload the files and make it running as they have a complete copy of the server data.

Git stores the content in form of trees and blobs. A blob (or Binary Large Object) is a object type used to store file content. It doesn't store metadata like date of creation of file. A commuted checksum SHA-1 is used to represent the file. A tree holds the listing of other trees and blobs. A commit takes the snapshot of the tree at that specific moment and saves it. When you create another commit, a new snapshot is taken with only the required changes in the new commit. The old / unchanged file remain the same and the new commit points to the old files. Thus the storage required is only for changed files.

19.3 Workflow

In a git, you have Working Directory, Staging Area (index), .git Directory (local repository), Upstream Directory (remote repository), and Stash.

Elementary Git workflow (You already have a git initialized repository):

- You modify a file in your Working Directory
- You mark the edited files are confirmed changes and stage them. This saves only the changes as a snapshot. The original file is stored as a reference.
- You commit those files. This takes the staged files and stores the snapshot of changes permanently in your git repository.
- You Push your changes from local repository to remote repository

Small Teams

When you are working in a small team say 2 members, then your workflow will be like in image below.

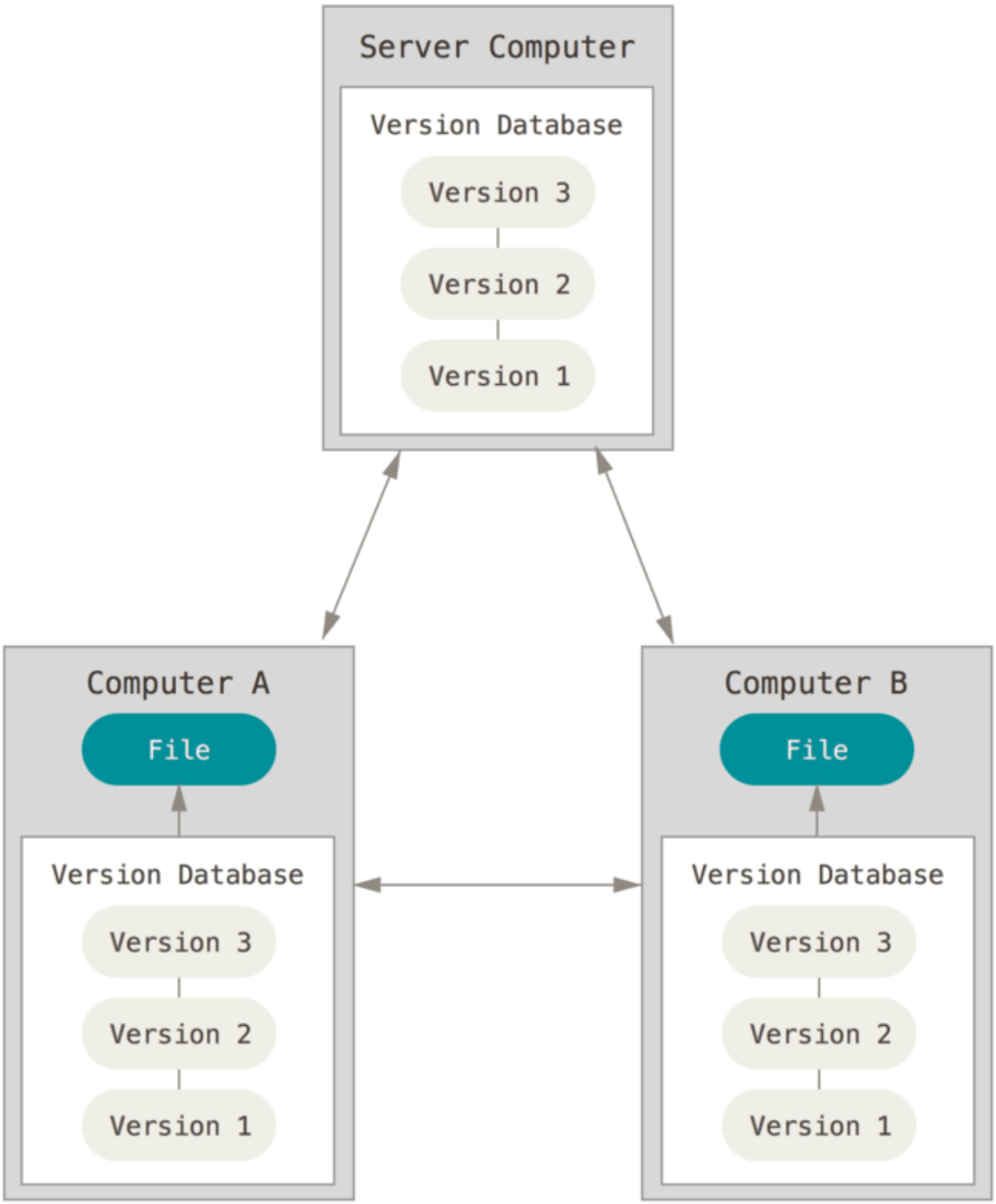


Fig. 19.2: Distributed Version Control System. (ref: Git Book V2)

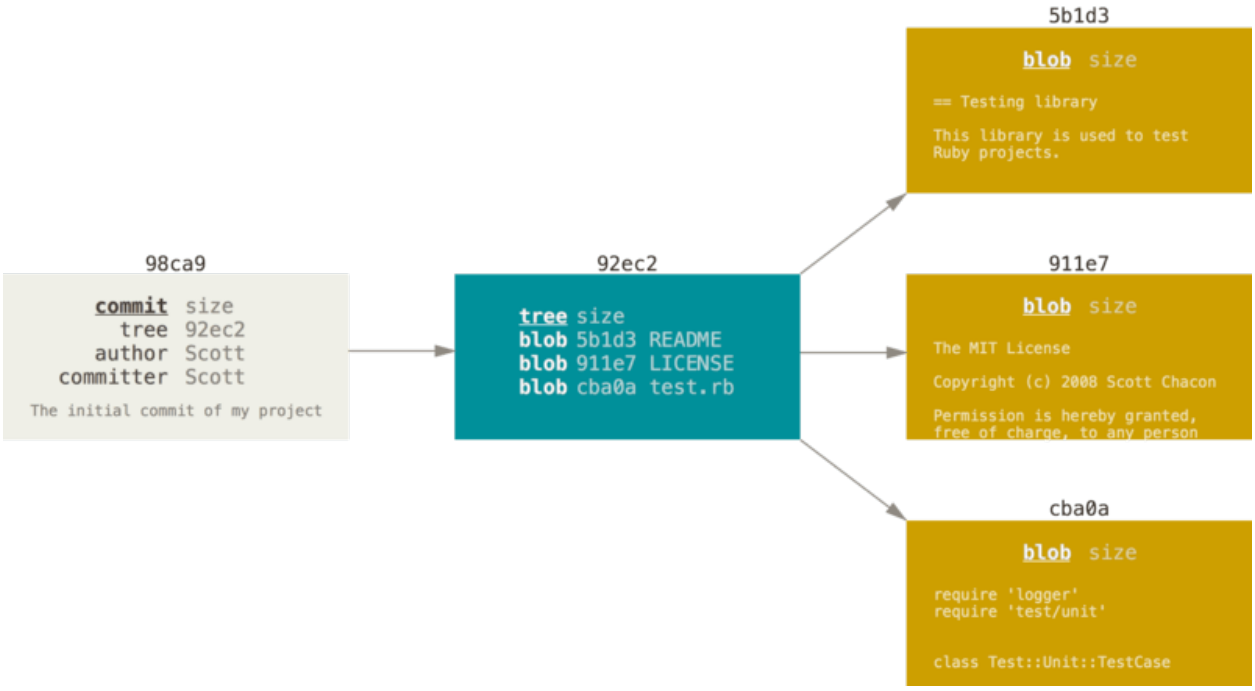


Fig. 19.3: GIT objects (ref: Git Book V2)

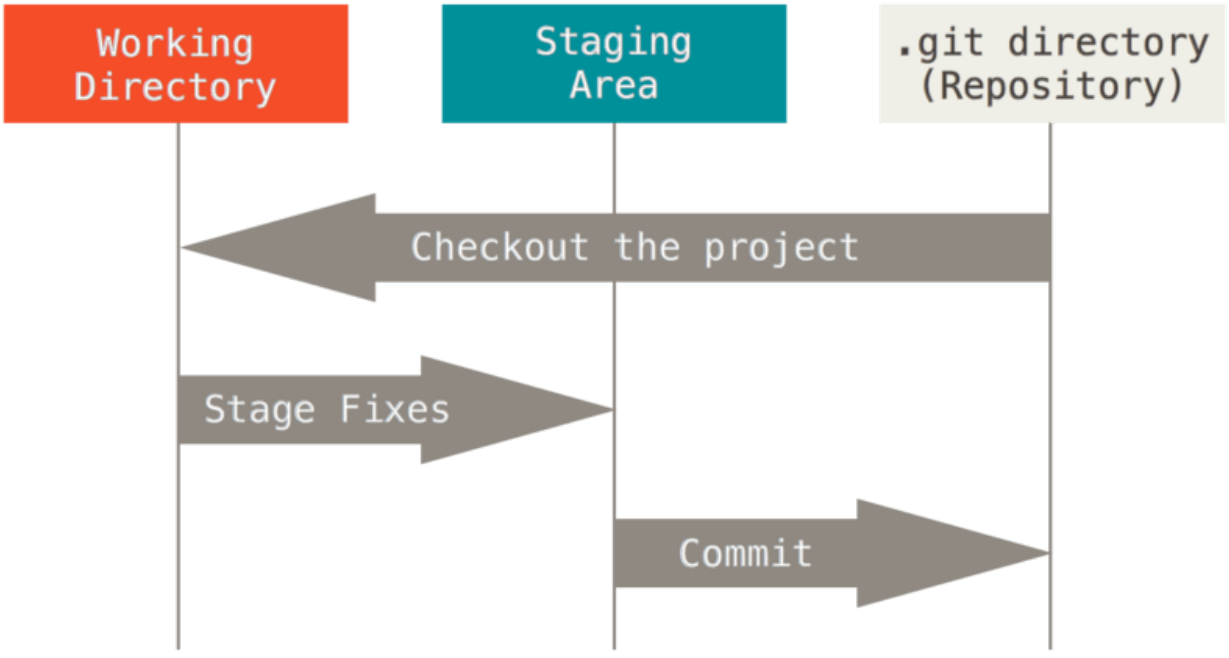


Fig. 19.4: Git States (ref: Git Book V2)

The typical activities done are:

1. Creation of a new repository either online or local on PC.
2. Clone repository if created online. Push repository to server if created offline.
3. Create required branch (say feature)
4. Code and Test it.
5. Commit the code.
6. Repeat steps **4** and **5** till a **certain part** of feature is ready.
7. Push to remote repository.
8. Repeat steps **4** to **7** till the feature is completed.
9. Merge the feature branch into main branch.

Contributing to online projects

When you are contributing to online repositories, the following activities are typically performed (ref: Git Book V2):

1. Fork the project to your account.
2. Create a topic branch from master.
3. Make some commits to improve the project.
4. Push the codes to your account's project.
5. Open a Pull Request on platform (say GitLab).
6. Discuss with the project's team, and optionally continue committing.
7. The project owner merges or closes the Pull Request.
8. Sync the updated master back to your forked project.

19.4 Commits

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the authors name and email address, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents): zero parents for the initial commit, one parent for a normal commit and multiple parents for a commit that results from a merge of two or more branches. (ref: Git Book V2)

There is no certain rule for commits but is suggested to be like **ONE commit per ONE task / bug**.

The commit messages are important as they provide info about what you edited in that specific commit. Thus when you review your changes history, you can recollect what you did by reading the message or when a 3rd person is going through your work can make a sensible understanding of your changes if they plan to work on your repository. The suggested length of text is of 50 characters. You provide to the point info in this.

Ex, if you change a function in a code:

- Fixing file in code *// Not good*
- Fix function ### *// Good*
- Fix function ### in file ### *// Better*

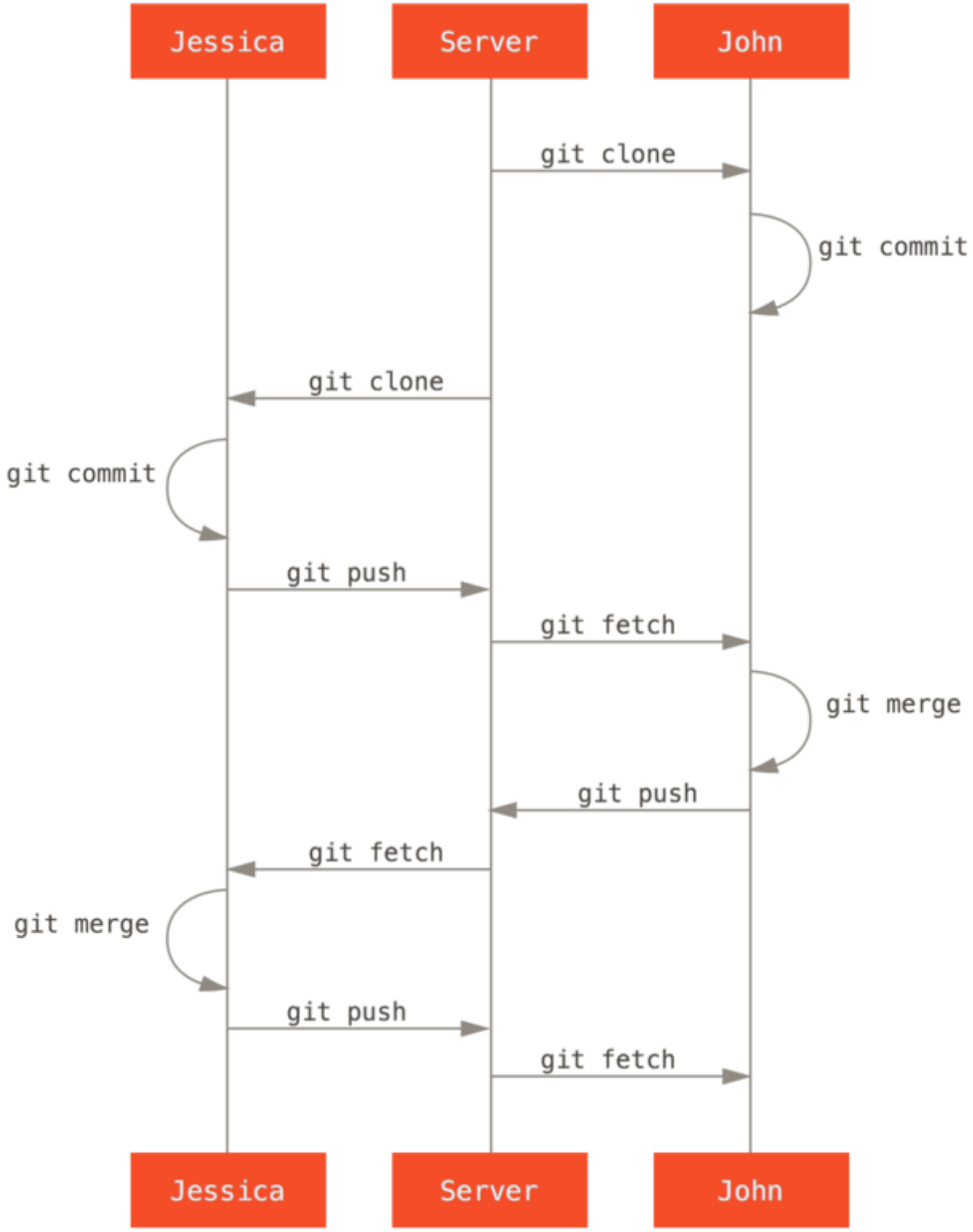


Fig. 19.5: Git workflow small team (ref: Git Book V2)

Hint: The files which are not required to be tracked can be ignored during commit using **.gitignore** file (found usually at root directory). Usual untracked files may include build files or temporary files. To untrack a file simply add the relative location of the file / folder to the .gitignore file.

Hint: If you forgot some files in the last commit or want to update your commit message, you can add the files to the staged area and the use amend command. **You must do this before you push to remote repository.**

19.5 Git Branches

It is used to create a separate directory of your code and then work on that copy. This allows you to create a copy from your main working branch with all the history of changes in it. You can then work on the new branch so as to fix a bug or implement a new feature. Once done, you can merge the branch to your main branch.

This helps in avoiding to edit files on a working environment. Ex, you have a blink LED program which starts automatically when the board boots up. You need to implement a button trigger to start this blink. So, you create a new branch from your main, and then implement and test your code. When everything works, you combine the changes from new branch to main branch.

You can also work on multiple branches simultaneously and can fix a bug while you are working on a new feature. Just be sure when merging your new features, you en-corporate the bugs fixed on the main branch during the development of the new branch.

19.6 Merging and Merge conflicts

To incorporate the new branch into the main branch, you use merge command. This copies the required code from the source branch to the target branch. The source branch can then be deleted if required.

In a situation if two people **A** & **B** are working on same project. **A** updates README.md file and pushed the changes. **B** also adds content to README.md and pushes the changes. But **B** gets a conflict error as **B** did not pull the **A's** updated version of README.md rather just pushed the changes.

To avoid such conflicts:

- Always PULL before PUSH.
- Do not have more that one developer working on a specific file.
- Try merging main branch frequently to your feature branch to keep new changes updated in your feature branch.

19.7 Tags

To mark a specific milestone for the project, tags are used. Tags are initially saved on local repository and have to be pushed to remote.

There are two types of tags (ref: Git Book V2):

- **Lightweight**: It points to a specific commit.
- **Annotated**: It stores the data as full object in git database. It contains tagger name, email and date.

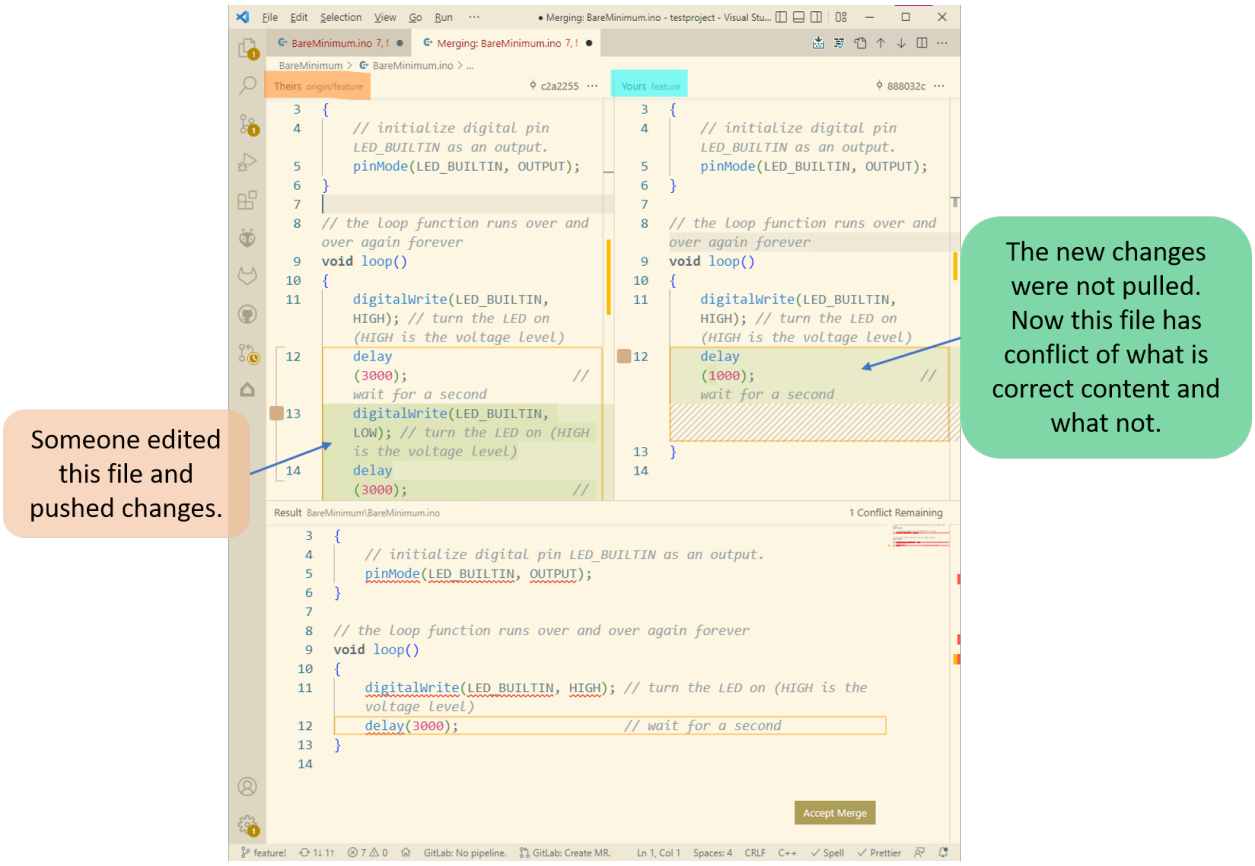


Fig. 19.6: Git conflicts demo.

The tag type depends on what information you want to save. If its a temporary tag, Lightweight is better option.

Hint: You can use semantic versioning for creating tags. Can find more information here: (https://semver.org/)

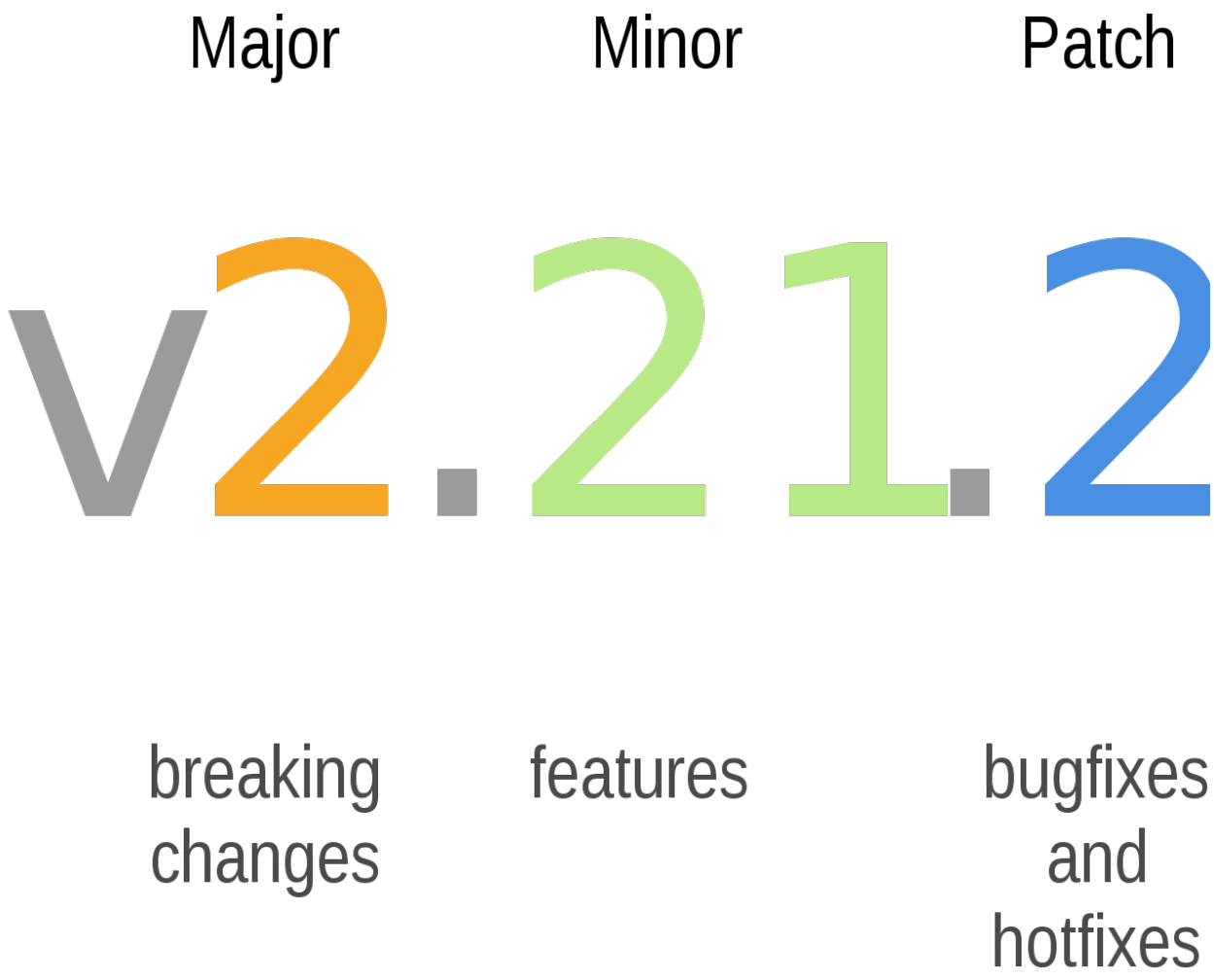


Fig. 19.7: Semantic Versioning (ref: Semantic Versioning Wikipedia)

19.8 Rebase

This allows to move commits and its history to another branch. This means that you are changing the base of your branch and making it appear as if it was created from a different commit. The main use of rebasing is to have a linear commit history.

The difference between a merge and rebase is that, in merge, if the branch moves forward with a commit, merge will create an empty commit. Hence it will be stored in your history.

In rebase, git finds a common commit and saves the differences after that. Then in sequential order applies the changes.

Hint:

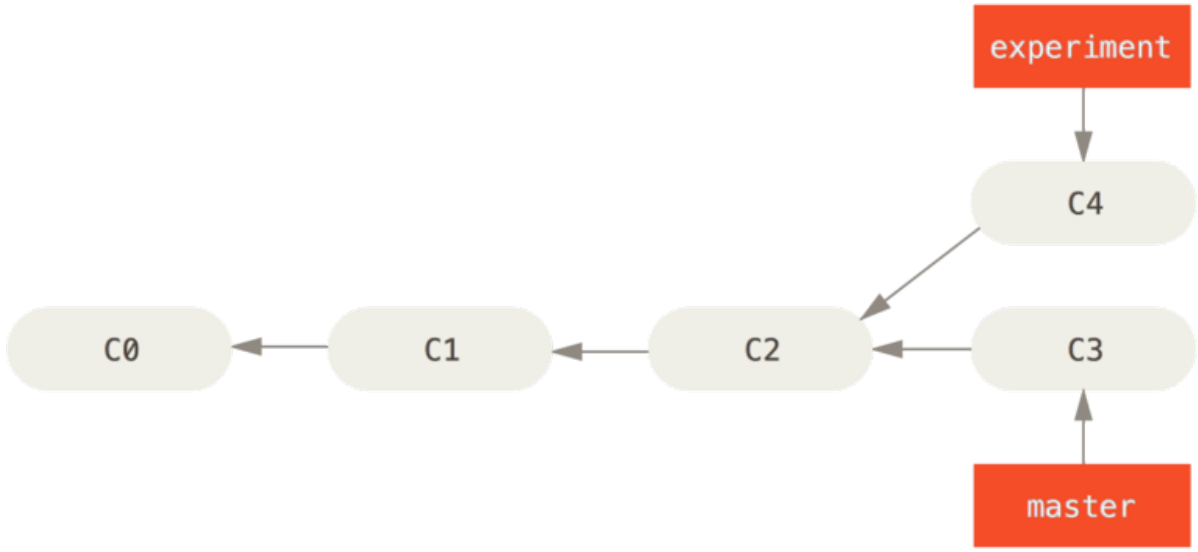


Fig. 19.8: Repository with difference in branches

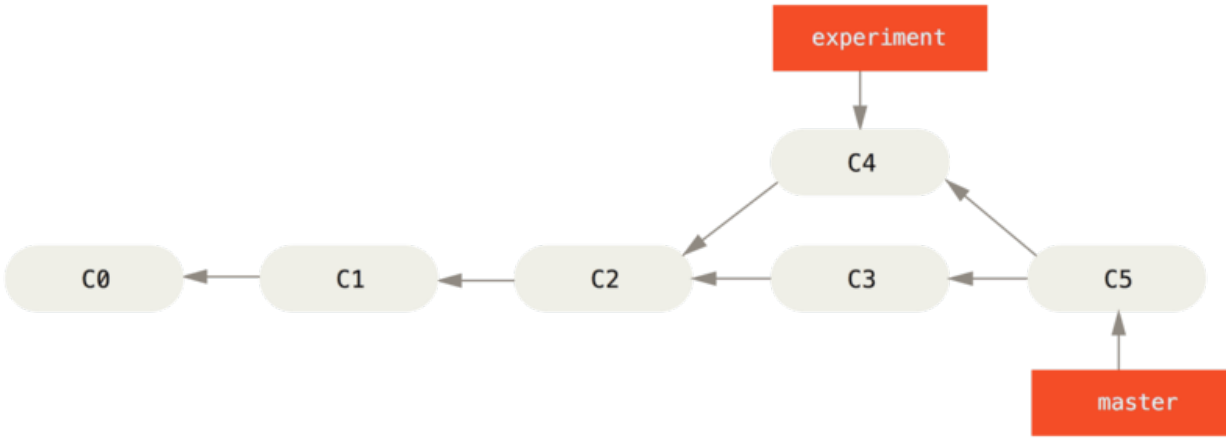


Fig. 19.9: Merge command

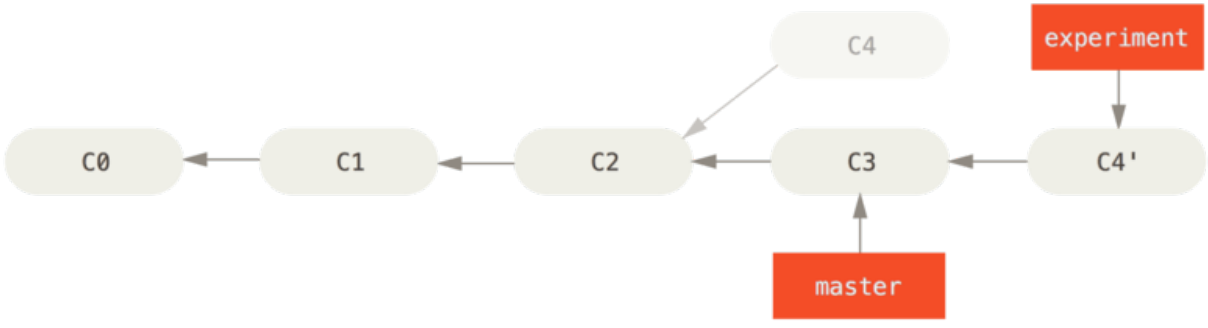


Fig. 19.10: Process of rebase

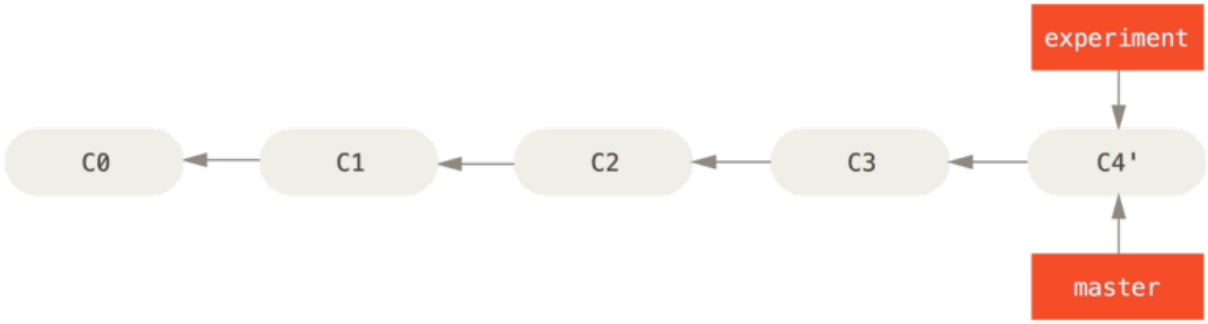


Fig. 19.11: Final repository history after rebase

For more information about this, visit

- https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase
- https://www.atlassian.com/git/tutorials/merging-vs-rebasing
- Git Book V2

19.9 Git Interface

Git is officially available as a command line tool, but is also available in different environments. You can download git command line tool from (https://git-scm.com/downloads). There are various GUI tools and extensions for popular IDE / Code Editors available.

Initially you need to provide some details about you when you first install git. To do so, open a new terminal / cmd.

For windows users -> Enter **git config --global core.editor "notepad"** in it.

Enter **git config --global --edit** in it. A file will open in text editor. Enter your **name** and **email** in that after colon (:).

19.10 Git Commands

A basic list of commands used in git is provided below. For additional commands, please refer the (https://training.github.com/)

Clone new repository:

- **git clone -b <BRANCH_NAME> <REPOSITORY_URL_WITH_HTTPS>**: To download repository to your PC as local repository.

Basic Workflow:

- **git status**: This provides you the current status of your repository. Ex, info about files modified, last commit info, branch info etc.
- **git add PATH_TO_FILE** or **git add .** : It adds the files to staging area. The dot (**.**) adds all changed files inside your working directory to your staging area.
- **git commit -m "commit message"**: It moves the files from staging area to local repository. The -m is a required flag. It is used to provide a sensible commit message to name the changes you did. Ex. Fix blink function in blink.cpp

- **git push**: Publishes the changes from local repository to remote repository.

Make new branch from main:

- **git checkout main**: To set main branch as default work branch.
- **git pull**: To download changes from main branch.
- **git branch NEW_BRANCH**: Create new branch
- **git checkout NEW_BRANCH**: Select NEW_BRANCH as default work branch
- **git push --set-upstream origin NEW_BRANCH**: To set NEW_BRANCH as default when uploading to remote repository.

Tags:

- **git tag**: List current tags.
- **git tag -a [Name_for_Tag] [Commit_Hash_ID]**: Add a tag with specific name for a specific commit.
- **git push –tags**: To push tags to remote.
- **git push –follow-tags**: Push only annotated tags to remote.
- **git show [TAG_Name]**: Display details of tag.
- **git tag -d [Name_for_Tag]**: TO delete a specific tag. To remove it from remote repository, use **git push origin :[Name_for_Tag]**

Amend:

- **git commit –amend**: Adds the files to our last commit.

Log:

- **git log**: To see the complete history of git commits.
- **q / z**: to exit the log.

19.11 Tasks

The tasks below are designed to teach Version Control with VS Code and GitLab.

Hint: You can follow these tasks directly with git commands in CMD/Terminal.

19.11.1 Task 1

Todo: Creating a new repository on GitLab website.

Follow the steps below:

1. Open GitLab website (https://git.fh-aachen.de/)`and login in. You will see a page with all the projects in your account.
2. Click on **New project** button to create a new project. ON next page, click on **Create blank project**.
3. On the next page, enter the name of your project.

4. In **project URL** select your FH-Kennung nummer (FH identifier) (AB1234S). In **Project slug** enter the name of the project. Please no spaces ( ),or dots (.).
5. For **Visibility level**, select Private.
6. Select **Initialize repository with a README** and click on **Create project**

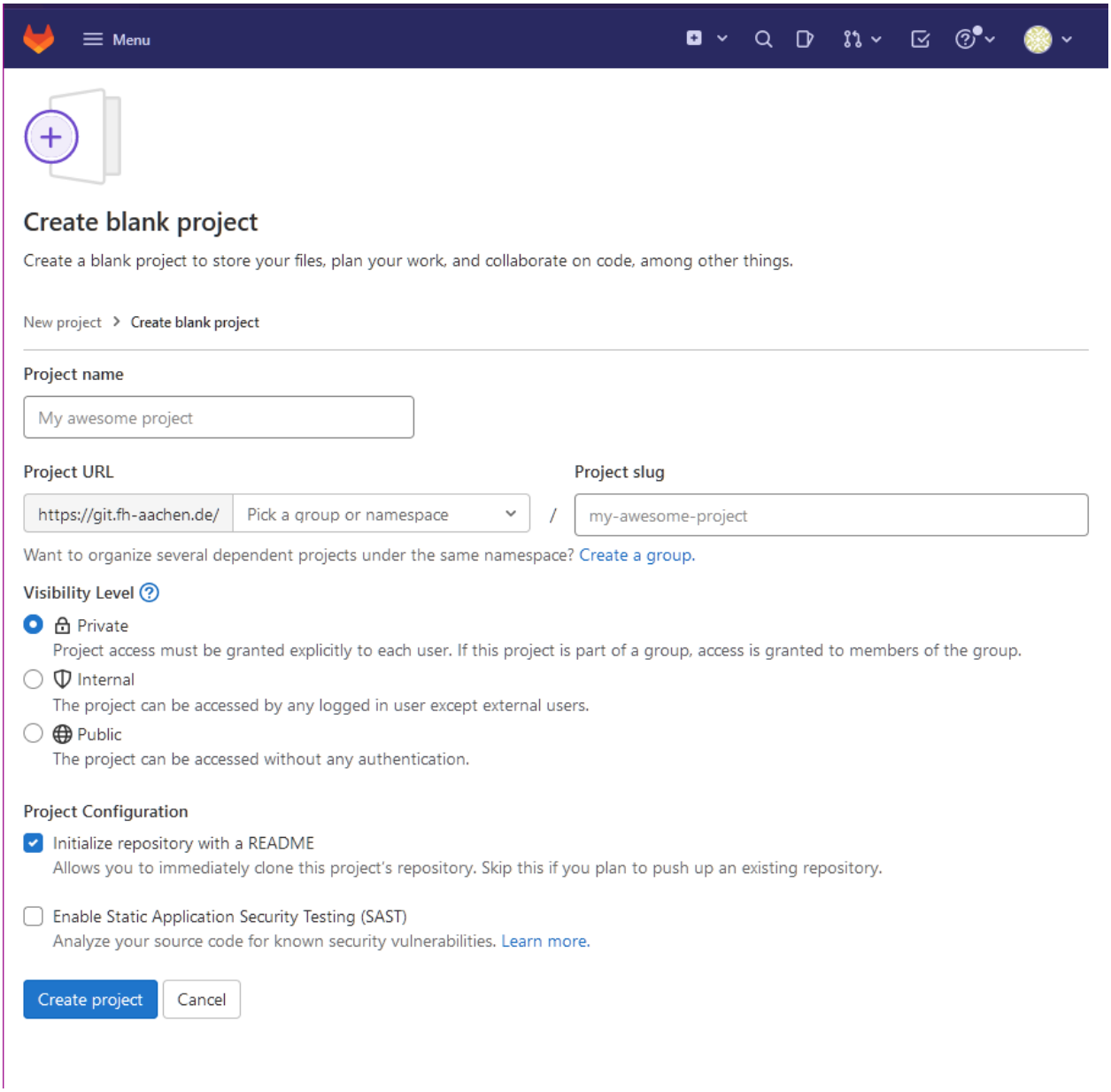


Fig. 19.12: Create new project on GitLab website.

19.11.2 Task 2

Todo: Managing project in VS Code.

Follow the steps below:

1. Have a ready setup of VS Code with connected GitLab account.
2. From the **Get Started** page on VS Code, click on **Clone Git Repository**. You can also clone a repository using Command Palette -> Git Clone.
3. Click **Clone from GitLab** -> Select the created repository from the list. Then select the link with **https**.
4. Select a folder to save the files on your PC. Prefer saving them in **Documents** folder. Then click **Select Repository Location**.
5. Once cloned, VS Code will ask you to whether to open the recent cloned repository in this window or a new window. Select **Open**.

19.11.3 Task 3

Todo: Managing branch and publishing branch.

Follow the steps below:

1. Create a new branch by click on bottom left option. (Initially you will see **Main** written)
2. Select **Create new branch**. Enter the name of the new branch as "**develop**".
3. Add a new folder in the project by clicking on **New Folder** from **Explorer** on the left side. Name the folder as **Blink**
4. Create a New File inside the folder **Blink.ino**. ! Extension is important
5. **Paste the following code in the file.**

```
// the setup function runs once when you press reset or power the board
void setup() {
// initialize digital pin LED_BUILTIN as an output.
pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
digitalWrite(LED_BUILTIN, HIGH);   // turn the LED on (HIGH is the voltage␣
↪level)
delay(1000);                       // wait for a second

}
```

6. Click on **Source Control** from left panel. Enter a commit message "initial commit".
7. Click on "Commit" button. Next, click on **Publish Branch**.

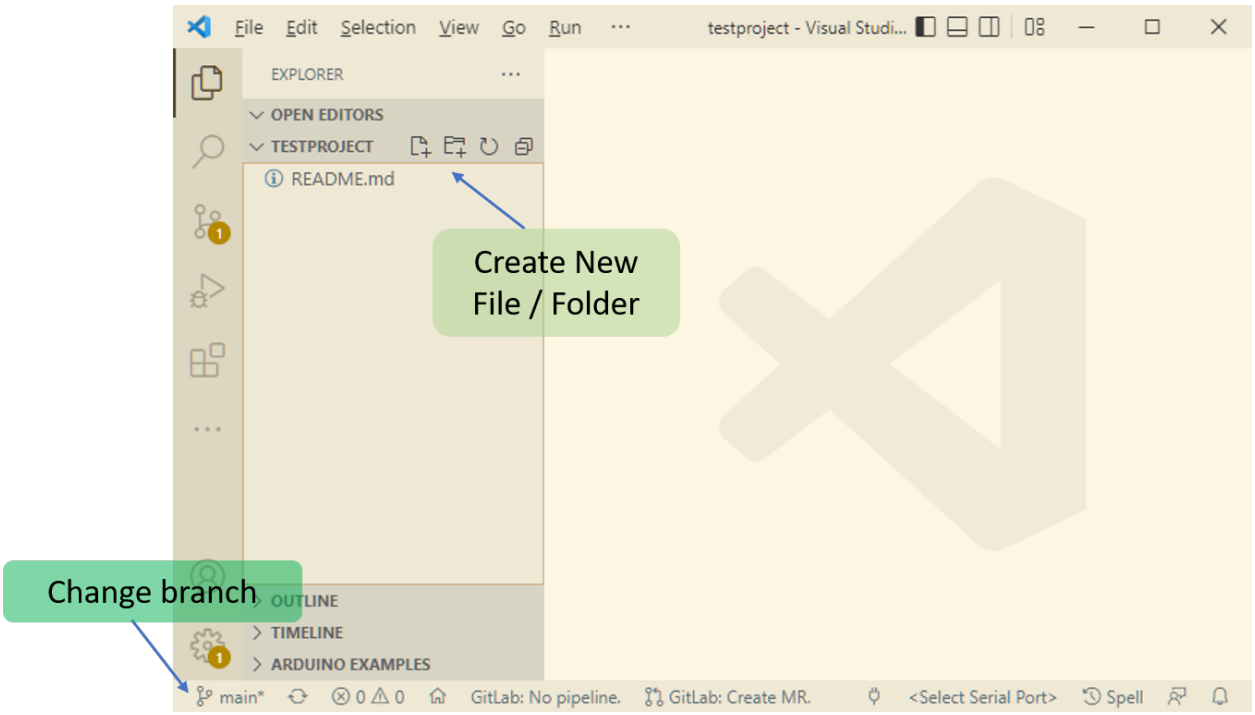
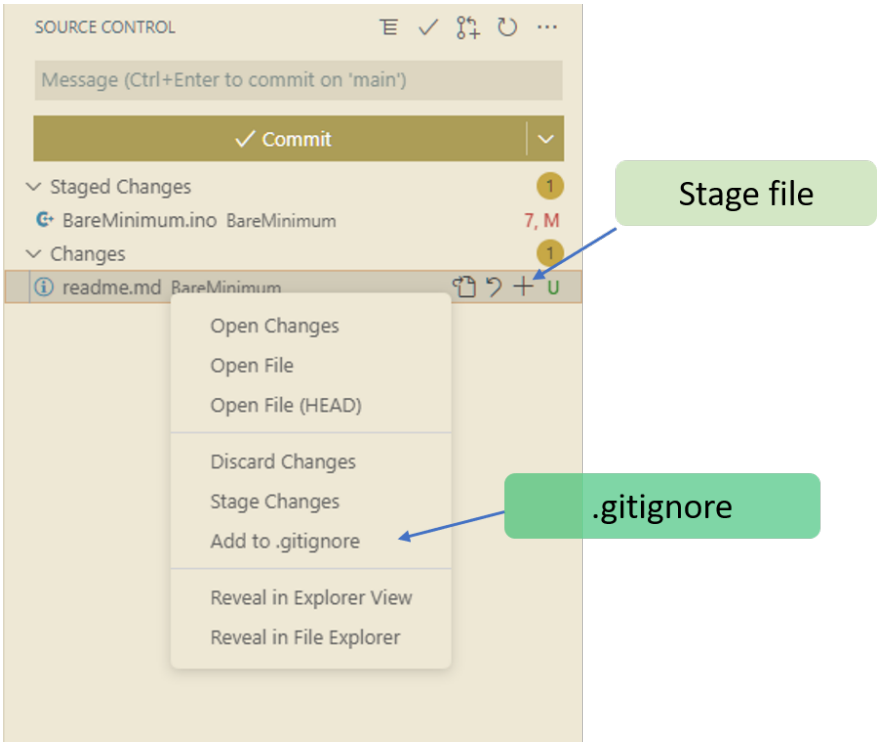


Fig. 19.13: VS Code Branches & File management

Hint: You can stage the required files for commit by using the plus(+) sign next to the files. When the commit button is pressed, only the staged files are committed.

The files which are not required can be added to git ignore by right clicking on the required file.



19.11.4 Task 4

Todo: Updating files online using GitLab website.

Follow the steps below:

1. Open your project online on (https://git.fh-aachen.de/)
2. Switch the branch of the project to "**develop**".
3. Locate the file "Blink.ino".
4. Click on **Open in web IDE**.
5. **Add the following lines in the code between " delay(1000); " and " } "**

```
digitalWrite(LED_BUILTIN, LOW);    // turn the LED off by making the␣
↪voltage LOW
delay(1000);                       // wait for a second
```

6. Once done, click on **Create commit** from the left side.
7. Enter a commit message "update LED blink". De-select **Start a new merge request**.
8. Click on **Commit** button.

Note: When you update code online, this means that the code on your PC is old. To get the latest version of code, you must **PULL** the changes. This is done in next task.

For more information, Please read **Workflow** section of tutorial.

19.11.5 Task 5

Todo: Pulling and Pushing files from VS Code.

Follow the steps below:

1. In VS Code, go to **Source Control**.
2. Click on Source Control menu (...) icon. In that, click on **Pull**. This will download the changes made from GitLab server.
3. Open the file "Blink.ino".
4. Replace the delay time from **1000** to **2000** for both delay functions.
5. Save the file, and go to **Source Control**.
6. Enter a commit message, "Updated Blink duration".
7. Click on Source Control menu (...) icon. In that, click on **Push**. This will upload the changes to GitLab server.

19.11.6 Task 6

Todo: Merging "**develop**" to "**main**" branch.

Follow the steps below:

1. Open your project online on (https://git.fh-aachen.de/)
2. From left side menu, click on **Merge requests**. On the new page, click on **New merge request**.
3. On the next page, select the source branch, "**develop**" and in target branch select "**main**" branch.
4. Next click on **Compare branches and continue**. Enter the title for the merge request.
5. In Description, provide brief information of the content of new feature / part completed with this branch.
6. Change rest of the options as per requirement.
7. Once all things are ready, click on **Create merge request**.
8. After this, another page would show up. This will show the details about the merge request. If everything is correct, it would show **Ready to merge!**.
9. Click on **Merge** to combine the code the main branch. After this, the changes will be moved to the main brnach.
10. Open the repository's main page. Confirm whether the code is merged properly or not.

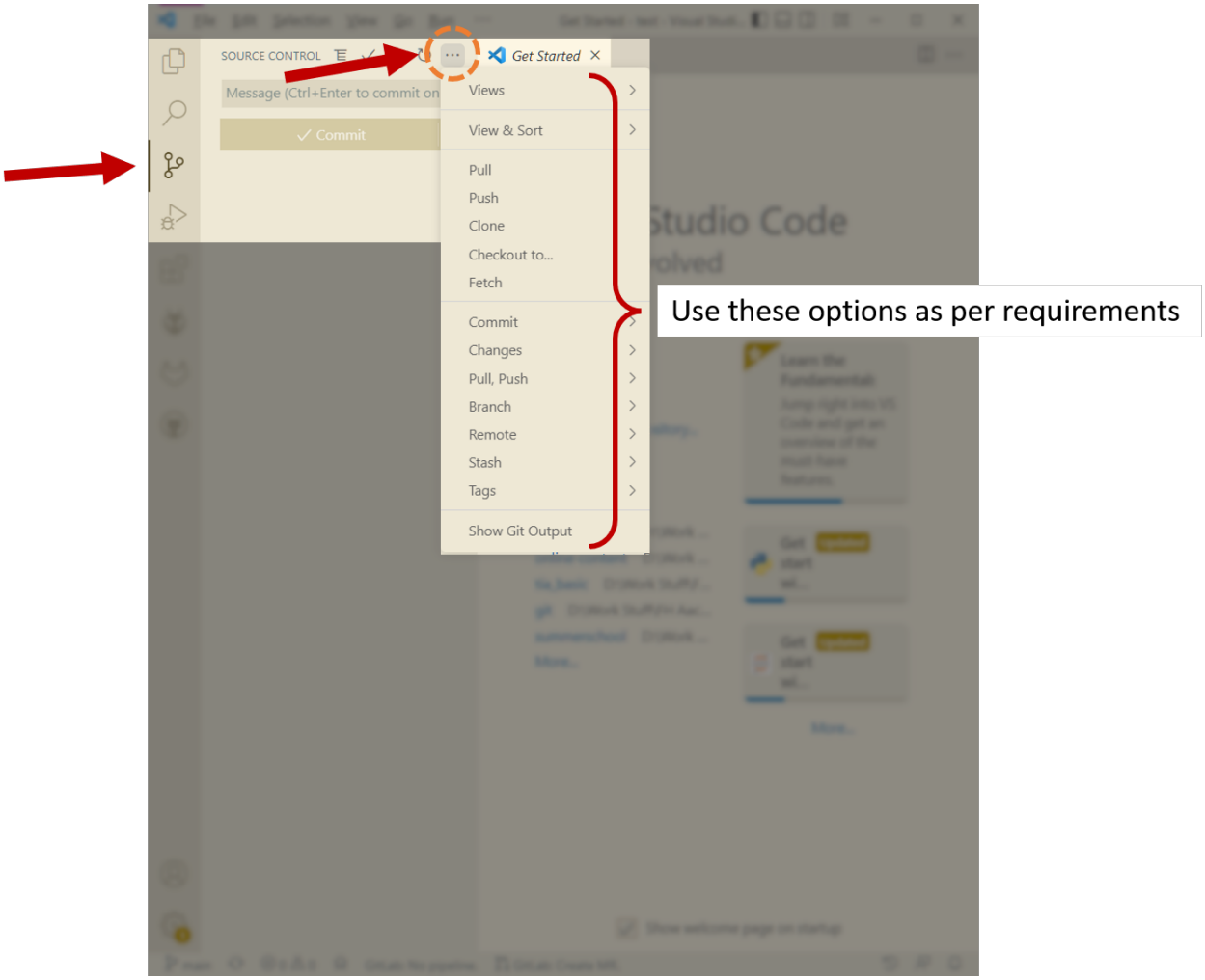


Fig. 19.14: VS Code Source control menu

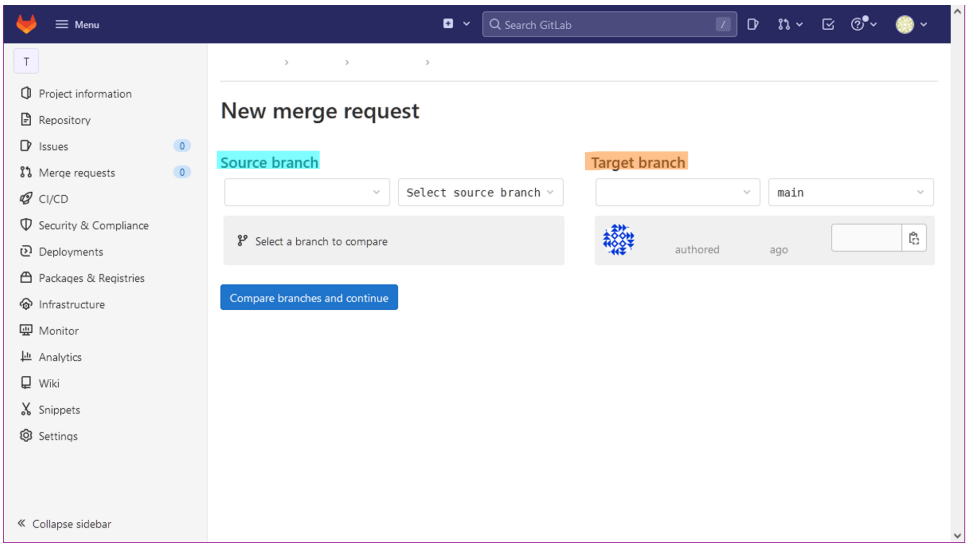


Fig. 19.15: GitLab merge branch page

19.11.7 Task 7

Todo: Create a annotated tag for this version of code.

Follow the steps below:

1. 1. In VS Code, go to **Source Control**.
2. Click on Source Control menu (...) icon. In that, click on **Tags**. Click on **Create Tag**.
3. Enter the required information such as Tag name and annotation message.
4. Click on **View -> Command Palette**.
5. Enter **Git: Push Tags**.

19.11.8 References

- GitLab Manage projects https://docs.gitlab.com/ee/user/project/working_with_projects.html

The Interreg V-A Euregio Meuse-Rhine (EMR) programme invests almost EUR 100 million in the development of the Interreg-region until 2020. This area stretches out from Leuven in the west to the borders of Cologne in the east, and runs from Eindhoven in the north all the way down to the border of Luxemburg. Over 5.5 million people live in this cross-border region, where the best of three countries merges into a truly European culture. With the investment of EU funds in Interreg projects, the European Union directly invests in the economic development, innovation, territorial development and social inclusion and education of this region.

This course is part of the digital twin academy: https://digital-twin-academy.eu/