MOLCURE Summary

Johannes Prizbilla CAMILLE Ulrich

Summary

Chapitre 1

Introduction

To introduce the HAIVE, the HAIVE is a bio-tech robot that aim to :

Accelerate drug discovery through robot and IA.

These documentation aim to introduce and defines all specification of the robot team and how the current system run.

Let's evaluate the HAIVE according to next figure : Currently HAIVE Robot is around level 2 :

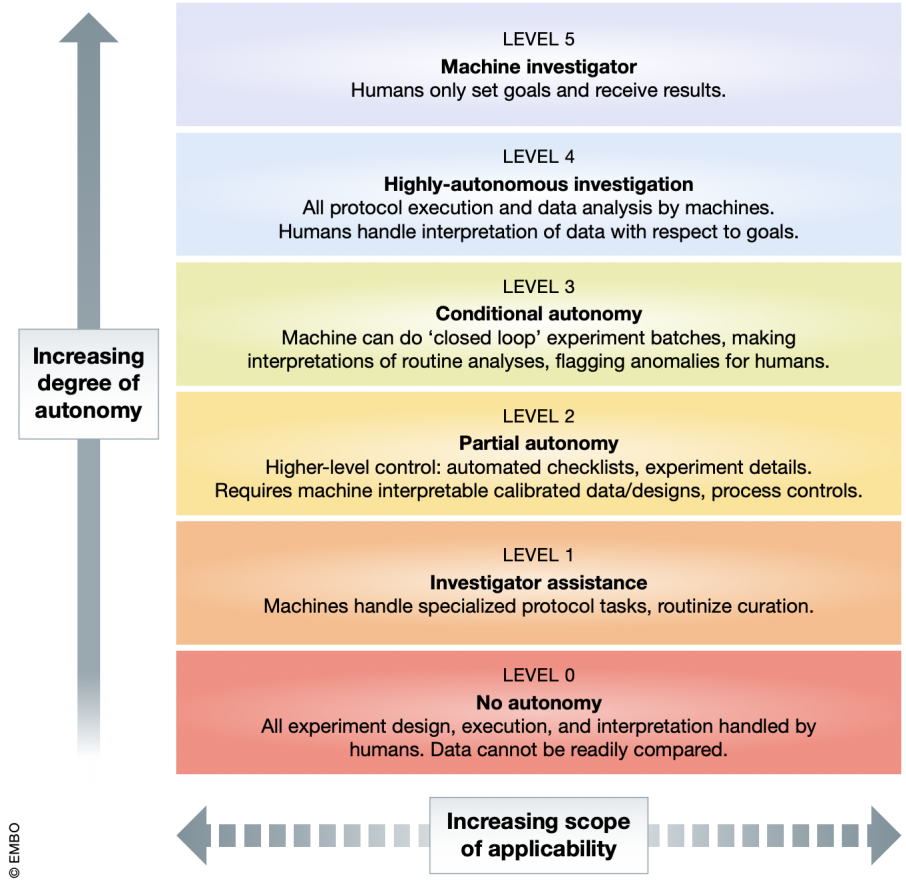


FIGURE 1.1 – Biotechnology levels of automation

"Partial autonomy". The minimum is to reach level 3 which implies a closed loop experiment at the moment

Chapitre 2

HAIVE OS

2.1 ROS2

R.O.S., which stands for Robot Operating System, is an *middleware* and therefore a tool that bridges the gap between hardware and software, as it provides the tools needed to handle both the hardware and software (AI, interface, etc.).

Robotic systems call on a wide range of skills, including mechanics, electrical engineering, control, computer vision, and many areas of computer science such as real-time computing, parallelism and networking. Advances in robotics often depend on technological breakthroughs in these scientific fields. A robotic system can therefore be complex, involving many different people. In order to be efficient in the design of its robotic system.

R.O.S. has practical aspects on several levels : simpler architecture design, debugging tools, multiple possible programming languages (c++, python, java, ...). HAIVE-OS is a ROS2 software platform responsible based on ROS2 humble for connecting and orchestrating the devices and services of the HAIVE robotics platform. originally developed by Johannes Prizbilla, we both highly contributed to this software

2.2 Command architecture

The system architecture of HAIVE OS is based on the design principle of separation of concerns. (For now) we distinguish 5 layers from bottom to top :

— Layer 0 | Hardware : This layer represents the distributed system of HAIVE. It contains all endpoint devices that implement some kind functionality. What all of these devices have in common from the viewpoint of HAIVE OS is that their hardware interface for communication is usually a WiFi chip (e.g. ESP or Raspberry Pi). The responsibility of this layer lies with implementing a functionality and exposing that functionality through some device API.

— Layer 1 | Device Layer : This layer mainly consists of a centralized system that manages the connections of all Layer 0 devices. The system has access to a device database which describes the nature of Layer 0 devices and their APIs. It also provides an interface to devices to upper layers and consumes device API requests. (Currently) the system does not handle any kind of request scheduling, which leaves it open to the developer to have a device of layer 0 receive commands on a on-by-one basis or in bulk. Any scheduling that does not happen in the hardware layer should therefore be implemented in Layer 2 or an intermediate Layer 1.5.

— Layer 2 | Behavior Layer : This layer contains software modules that implement abstract behaviors using information provided by Layer 1. An example of a Behavior Layer module is the

hos_l2_proxy package which implements the L2 protocol or the path planing behavior shown in this example. Most of the work of the Behavior Layer still has to be done.

— Layer 3 | Protocol Layer : This layer is supposed to manage user requests for jobs for the HAIVE OS. It will be the interface that data from the UI will be passed to.

— Layer 4 | User Data : This layer should describe the data format of the data needed to execute protocols on the HAIVE system. This data will be produced by an updated version of the HAIVE UI.

This graph shows the entire ROS architecture we will try to explain through the following sections.

2.3 Device API

Let's take a look at the generated file in hos_device_layer/hos_device_api.py and see what it does for the example of the delta_arm_move function :

```
# ...
from hos_device_layer.device_manager import (create_device_api_call,
    send_device_api_call, DeviceAPIArg, DeviceAPICallInfo)

# ...

#
# Moves endeffector of Delta HAIVE to 3D coordinate (x, y, z) in mm
#
def delta_arm_move(node: Node, device_id: str, x: float, y: float, z: float) ->
    DeviceAPICallInfo:
  function_name = inspect.stack()[0][3]
  args = [
    DeviceAPIArg(
      'x',
      'float32',
      str(x),
    ),
    DeviceAPIArg(
      'y',
      'float32',
      str(y),
    ),
    DeviceAPIArg(
      'z',
      'float32',
      str(z),
    ),
  ]
  request = create_device_api_call(device_id, function_name, args)
  return send_device_api_call(node, request)

# ...
```

In every device API function, we first have to pass in a reference to the calling node and a device_id (e.g. H4001). Next, follow the actual device function arguments, in this case, x, y, and z for the Cartesian coordinates of the delta arm.

The function then first looks up its name on the stack and generates a list of DeviceAPIArg that hold the device function argument information. A DeviceAPICall request is then generated using the create_device_api_call function of the hos_device_layer.device_manager module. Finally, the DeviceAPICall is sent by using the send_device_api_call function (also exposed by the hos_device_layer.device_ma

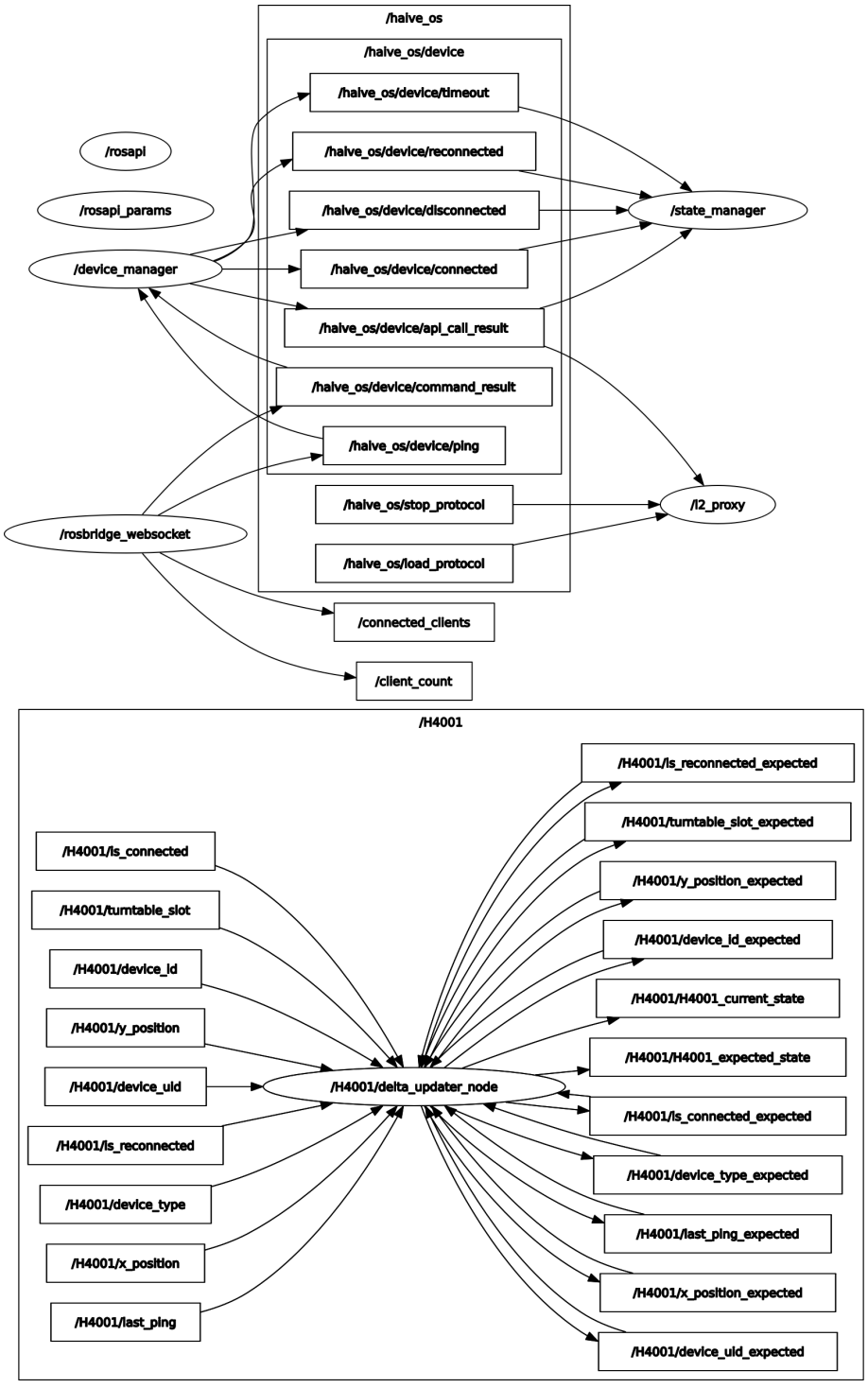

/haive_os
/haive_os/device
/haive_os/device/timeout
/haive_os/device/reconnected
/haive_os/device/disconnected
/haive_os/device/connected
/haive_os/device/api_call_result
/haive_os/device/command_result
/haive_os/device/ping
/haive_os/stop_protocol
/haive_os/load_protocol
/rosapi
/rosapi_params
/device_manager
/rosbridge_websocket
/state_manager
/l2_proxy
/connected_clients
/client_count
/H4001
/H4001/is_connected
/H4001/turntable_slot
/H4001/device_id
/H4001/y_position
/H4001/device_uid
/H4001/is_reconnected
/H4001/device_type
/H4001/x_position
/H4001/last_ping
/H4001/delta_updater_node
/H4001/is_reconnected_expected
/H4001/turntable_slot_expected
/H4001/y_position_expected
/H4001/device_id_expected
/H4001/H4001_current_state
/H4001/H4001_expected_state
/H4001/is_connected_expected
/H4001/device_type_expected
/H4001/last_ping_expected
/H4001/x_position_expected
/H4001/device_uid_expected

FIGURE 2.1 –

module). To be able to track the result of the DeviceAPICall the function then returns a DeviceAPICallInfo object to the caller of the function.

The DeviceManager is the node that contains all logic of the device layer. The public interface of the node consists of the following ROS topics and services :

Subscribed

```
DEVICE_PING_TOPIC = 'haive_os/device/ping': Pings HAIVE OS from device
DEVICE_CMD_RESULT_TOPIC = 'haive_os/device/command_result': Sends result of command call from device to HAIVE OS
DEVICE_STREAM_TOPIC(uid, stream_name) = 'haive_os/device/uid{uid}/stream/{stream_name}': Streaming data from device to HAIVE OS
Published
DEVICE_API_CALL_RESULT_TOPIC = 'haive_os/device/api_call_result': Sends result of device api call from HAIVE OS upwards
DEVICE_CONNECTED = 'haive_os/device/connected': Sends connection info when a new device connects to HAIVE OS
DEVICE_TIMEOUT = 'haive_os/device/timeout': Sends connection info when a device timed out
DEVICE_RECONNECTED = 'haive_os/device/reconnected': Sends connection info when a known device reconnects to HAIVE OS
DEVICE_DISCONNECTED = 'haive_os/device/disconnected': Sends connection info when a known device disconnects from HAIVE OS
Served
CONNECT_DEVICE_SERVICE = 'haive_os/device/connect': Requests device connection to HAIVE OS
DEVICE_COMMAND_SERVICE(uid: int) = 'haive_os/device/uid{uid}/command': Serves device command interface
SERIALIZED_DEVICE_COMMAND_SERVICE(uid: int) = 'haive_os/device/uid{uid}/serialized_command': Serves device command interface using serialized data (e.g. used for L1)
DEVICE_API_CALL_SERVICE = 'haive_os/device/api_call': Serves device api from the device layer upwards
DEVICE_CONNECTION_INFOS_SERVICE = 'haive_os/connected_devices': Serves device connection infos from the device layer upwards
```

The DeviceManager also loads device specification information about the HAIVE system to have knowledge about device types, device functionality and fleet constellations. This information is stored in an online database on Airtable, which serves as the human-interface to edit the specification. Once data is loaded from the online database, a copy of that information is also stored locally and can be used to start the node in an offline environment (see device_db launch argument in hos_run.launch.py).

2.4 Serialization

The Airtable database interface saves the record of all DeviceAPI and refers to their input, output, and serialization format. the hos_generator package generates a python function as seen previously. The API table has several properties :

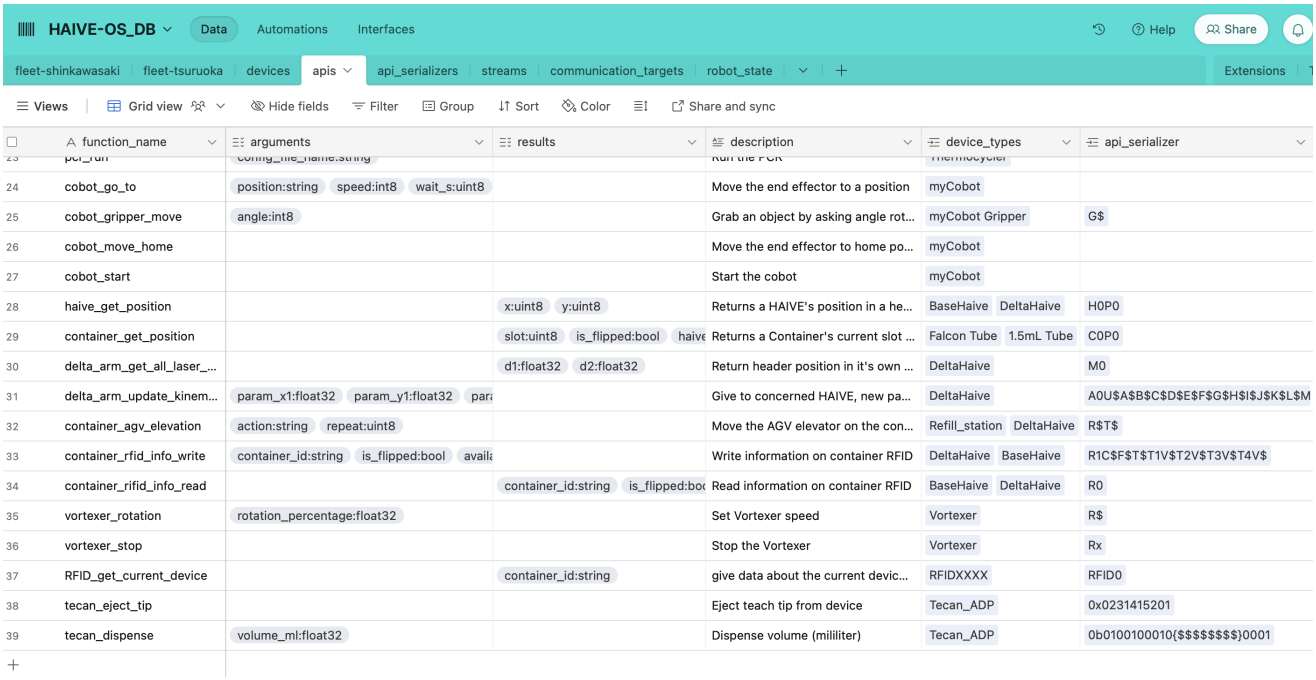
- "api_serializer" holds the string format to send to the device (example : A$U0V$)
- "input" refers to the number, type, and order of input for the DeviceAPI
- "output" refers to the number, type, and order of the output of the DeviceAPI return by a device. each device return values to the HAIVE_OS will be with that format : VVV$. Though we can't understand the nature of the data, thus the output information is here to give the order and type of each output value.

Now let's understand how it works on the micro-controller side. to allow to communicate with the HAIVE-OS there were 2 possibilities so far :

— WiFi, using rosbridge allowing to send information to a ROS node through WiFi
— Serial communication, port, and baud rate are referred to in the fleet device table of the database

Actually, firmware devices only send a VVV$ format, after the hos_client takes care to convert these data as a ROS Message, in the case of WiFi communication it converts the string to a JSON string and then sends as a ROS message through the rosbridge. This hos_client is held either directly by the device if they are using a WiFi protocol or can also be held directly by the computer handling the serial communication with the device and running the HAIVE-OS, then the api_serialized is converted to JSON string and sent as a ROS message from the HAIVE-OS itself.

This message is first sent to DEVICE_CMD_RESULT_TOPIC, giving the same message format for any device response. Once received, it will create another ROS message to DEVICE_API_CALL_RESULT_TO giving information to DeviceAPI call upward.



	function_name	arguments	results	description	device_types	api_serializer
24	cobot_go_to	position:string, speed:int8, wait_s:uint8		Move the end effector to a position	myCobot	
25	cobot_gripper_move	angle:int8		Grab an object by asking angle rot...	myCobot Gripper	G$
26	cobot_move_home			Move the end effector to home po...	myCobot	
27	cobot_start			Start the cobot	myCobot	
28	haive_get_position		x:uint8, y:uint8	Returns a HAIVE's position in a he...	BaseHaive, DeltaHaive	H0P0
29	container_get_position		slot:uint8, is_flipped:bool, haive	Returns a Container's current slot ...	Falcon Tube, 1.5mL Tube	C0P0
30	delta_arm_get_all_laser_...		d1:float32, d2:float32	Return header position in it's own ...	DeltaHaive	M0
31	delta_arm_update_kinem...	param_x1:float32, param_y1:float32, para		Give to concerned HAIVE, new pa...	DeltaHaive	A0UABCDEFGHIJKL$M
32	container_agv_elevation	action:string, repeat:uint8		Move the AGV elevator on the con...	Refill_station, DeltaHaive	RT
33	container_rfid_info_write	container_id:string, is_flipped:bool, availa		Write information on container RFID	DeltaHaive, BaseHaive	R1CFT$T1V$T2V$T3V$T4V$
34	container_rifid_info_read		container_id:string, is_flipped:boo	Read information on container RFID	BaseHaive, DeltaHaive	R0
35	vortexer_rotation	rotation_percentage:float32		Set Vortexer speed	Vortexer	R$
36	vortexer_stop			Stop the Vortexer	Vortexer	Rx
37	RFID_get_current_device		container_id:string	give data about the current devic...	RFIDXXXX	RFID0
38	tecan_eject_tip			Eject teach tip from device	Tecan_ADP	0x0231415201
39	tecan_dispense	volume_ml:float32		Dispense volume (milliliter)	Tecan_ADP	0b0100100010{$$$$$$$$}0001

FIGURE 2.2 – Airtable database : api table

2.5 Robot State

Robot space (can also be called State space representation) is a mandatory tool for any robotics system to make protocol dynamics but please make a difference between creating a dynamic behavior and actually knowing the status of the system.

another example : if we add devices during runtime, we can't change the protocol while it is running, while having robot state and sensor allow us to do so, behave according to the device status that might change for any reason (users, errors).

There are two main steps :
— Creating an estimator : after a firmware API call has been done, we should update the theoretical state which can be deduced by the command ask to robot (so here we assume that the robot

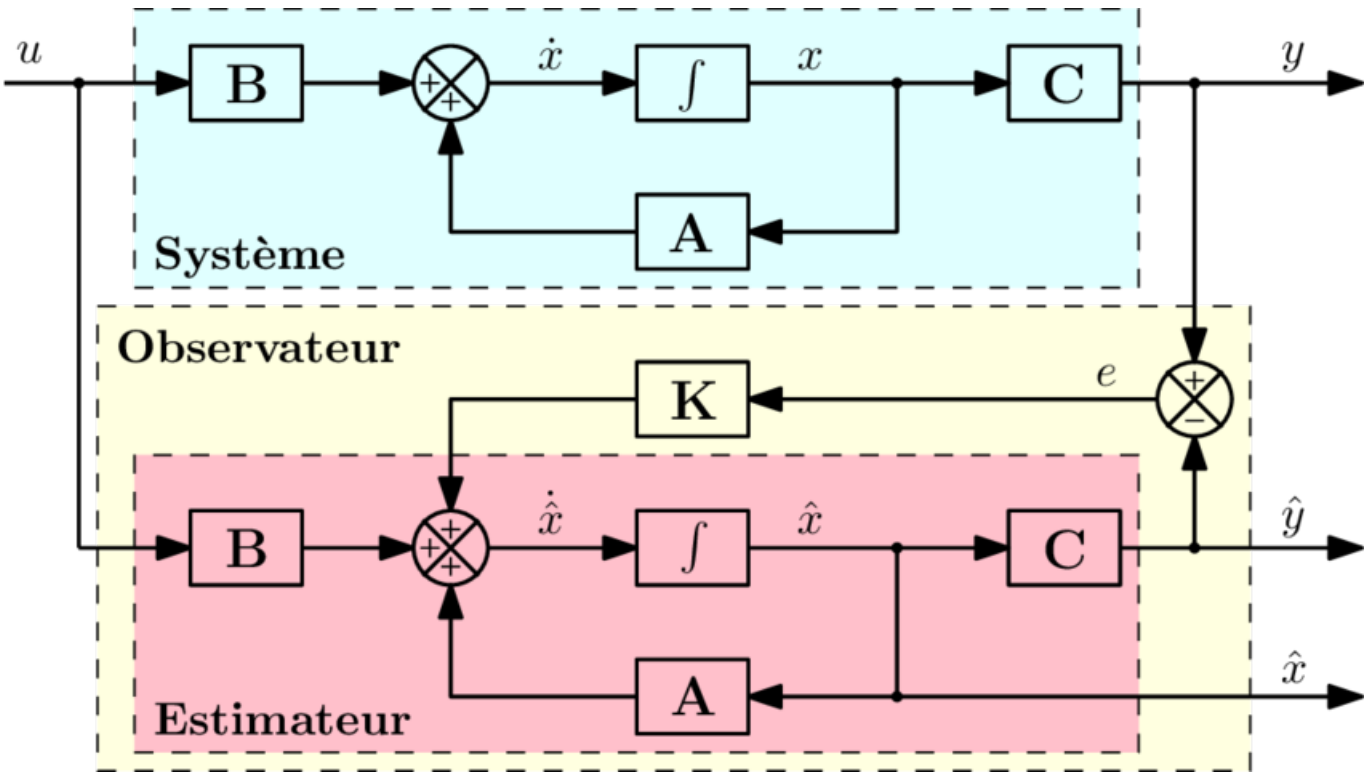


FIGURE 2.3 – State space representation

does exactly what he has been asked). but obviously, this will sometimes not happen so we need an observer !

— Creating an observer : this observer is sensors related, from the information gets on the real world, we translate this as robot state information. for instance, a simple rotation encoder can give more information than just an angle such as a boolean is_open, ... or combined with another sensor it might give something else

it's important to understand in Figure 2.3 :

— the blue zone corresponds to real-word physics behavior, thus B, A, and C are unknown oherwise it would mean that we know 100% with no error of our real life/condition parameter.

— The red zone is the estimator corresponding to the mathematical model that tries to simulate the physic and robot state. B, A and C need to be the closest to the real system B, A, and C in order to have an accurate estimator.

— the yellow zone is the observer, where all sensor information, reading real word parameters, and condition give information about the robot state. They may be not be directly related to the robot state but can be translated and fused with other sensor data with the system K.

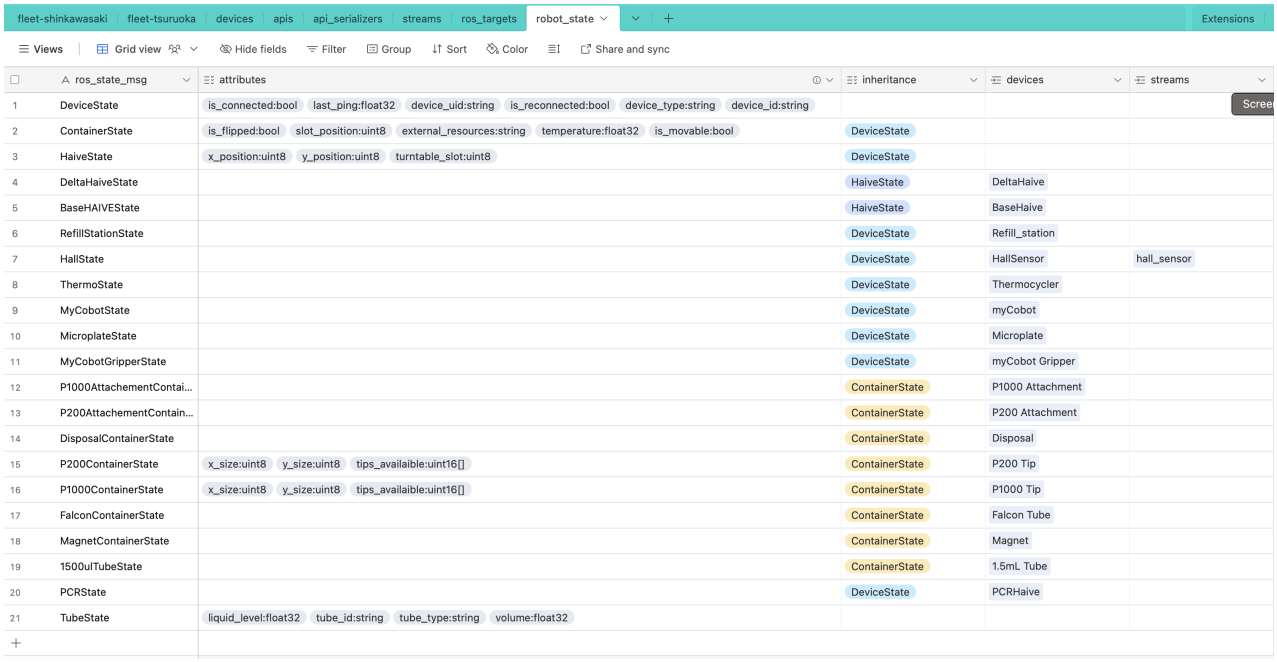
2.5.1 StateAPI

How to update robot state :

— Create new State/Update attributes Robot State model new device state or attributes can be updated/added in the robot_state table in the database.

— ros_state_msg : the name of the state message returned by the GetStateAPI and will give access to all its attributes : list of attributes that will define the state of the device.

— inheritance : create an inheritance pattern (for instance, most of the devices will inherit from DeviceState, which means they all inherit from DeviceState attributes such as is_connected, device_id, ...) StateAPI and estimator



	ros_state_msg	attributes	inheritance	devices	streams
1	DeviceState	is_connected:bool, last_ping:float32, device_uid:string, is_reconnected:bool, device_type:string, device_id:string			
2	ContainerState	is_flipped:bool, slot_position:uint8, external_resources:string, temperature:float32, is_movable:bool	DeviceState		
3	HaiveState	x_position:uint8, y_position:uint8, turntable_slot:uint8	DeviceState		
4	DeltaHaiveState		HaiveState	DeltaHaive	
5	BaseHAIVEState		HaiveState	BaseHaive	
6	RefillStationState		DeviceState	Refill_station	
7	HallState		DeviceState	HallSensor	hall_sensor
8	ThermoState		DeviceState	Thermocycler	
9	MyCobotState		DeviceState	myCobot	
10	MicroplateState		DeviceState	Microplate	
11	MyCobotGripperState		DeviceState	myCobot Gripper	
12	P1000AttachementContai...		ContainerState	P1000 Attachment	
13	P200AttachementContain...		ContainerState	P200 Attachment	
14	DisposalContainerState		ContainerState	Disposal	
15	P200ContainerState	x_size:uint8, y_size:uint8, tips_availaible:uint16[]	ContainerState	P200 Tip	
16	P1000ContainerState	x_size:uint8, y_size:uint8, tips_availaible:uint16[]	ContainerState	P1000 Tip	
17	FalconContainerState		ContainerState	Falcon Tube	
18	MagnetContainerState		ContainerState	Magnet	
19	1500ulTubeState		ContainerState	1.5mL Tube	
20	PCRState		DeviceState	PCRHaive	
21	TubeState	liquid_level:float32, tube_id:string, tube_type:string, volume:float32			

FIGURE 2.4 – Airtable database : ros_state_msg

To have a proper estimator, each time a firmware API is called (API from Airtable) we need to update target devices state. Upper-level Software APi will also create behavior that will change the robot state, but most of these software APIs will be broken down to firmware APIs call that is why if every DeviceAPi is called along with the StateAPI call, the quality of the StateAPI doesn't depend on the developer which might forget some attributes to update if they do it manually.

Unfortunately, StateAPIs need to be updated by humans. Indeed after generating robot_state msg and the StateAPIGen, SetStateAPI is automatically called with DeviceAPI args.

by default StateAPI calls look like this :

```
def state_delta_arm_move(self, node: Node, device_id: str, x: float, y: float, z: float) -> StateAPICallInfo:
    function_name = inspect.stack()[0][3]
    args = [
      DeviceAPIArg(
      'x',
      'float32',
      str(x),
    ),
    DeviceAPIArg(
      'y',
      'float32',
      str(y),
    ),
    DeviceAPIArg(
      'z',
      'float32',
      str(z),
    ),
    ]
    raise Exception("This state function has not been updated, please override this function in StateAPI Class")
```

By default if the function is not overwritten, it will throw an Execption, forcing the developper to update those. In order to generate first the StateAPI library :

```
$ ros2 launch hos_run hos_generator.launch.py
```

This will generate all the robot_state_msg (.msg files), update DeviceAPI and StateAPI files (hos_device_api_raw.py and hos_state_api_raw). then we need to build again.

```
$ colcon build
```

Before the build, we should also check that every StateAPI is overridden in hos_state_api.py, otherwise, it will raise an exception forcing the developer to update this function. Sensors callback

2.5.2 State updater

Whenever a Device is connected to the HAIVE-OS, or we create a DeviceAPI that target a device, it will create a state updater and a different one according to its device_type in the fleet information. state updater is a threaded Node handling all the StateAPI request concerning his device. thus it's different for each device because they have the same state to handle, they have their own model or behavior

Each attribute defined in the Airtable database, will be instantiate in 3 variations :

— variable_name_expected : that gives the expected state of a device if on going DeviceAPI are done

— variable_name_before : that gives the initial state of the device from the last DevciceAPI call done

— variable_name_current : Interpolate robot state attributes (example : temperature sample only at t1 and every 1 minute, during this 1 minute we can interpolate the temperature of the device knowing room temperature and some physics coefficient.

2.5.3 Sensor

Whenever a sensor sends some information data directly to the HAIVE-OS, it needs to trigger callback methods for some devices. Automatically, every device referred as a child device in the Airtable database, his state_updater will automatically subscribe to this sensors

```
  def _on_device_stream(self, stream_name: str, message: object) -> None:
    device_uid = message.uid
    data = message.data
    self.get_logger().warning(f"THIS IS A DEBUG MSG: stream event from {self.
   node_name}>> {stream_name}:{device_uid}:{data}")

    #TODO:Ricky:Here we have to identify the stream type from UID using self.db
    sensor_device_id = self._db.get_device_id(device_uid)
    stream = self._db.get_device_streams(sensor_device_id)
    for s in stream:
      self.get_logger().info(s.stream)
    #TODO:so the sorting/switch case should be done here ? indeed each sensor
   should behind trigger different call backs
    match s.stream:
      case "hall_sensor":
        pass
      case "rfid_info":
        rfid_info_callback(self, data)
      case _:
        self.get_logger().warning(f"No callback found for this stream type : {s.
   stream}")
```

In the switch case, according to the sensor type we call a certain callback. Of course you can override the _on_device_stream method so that according to the device you can change the callback function.

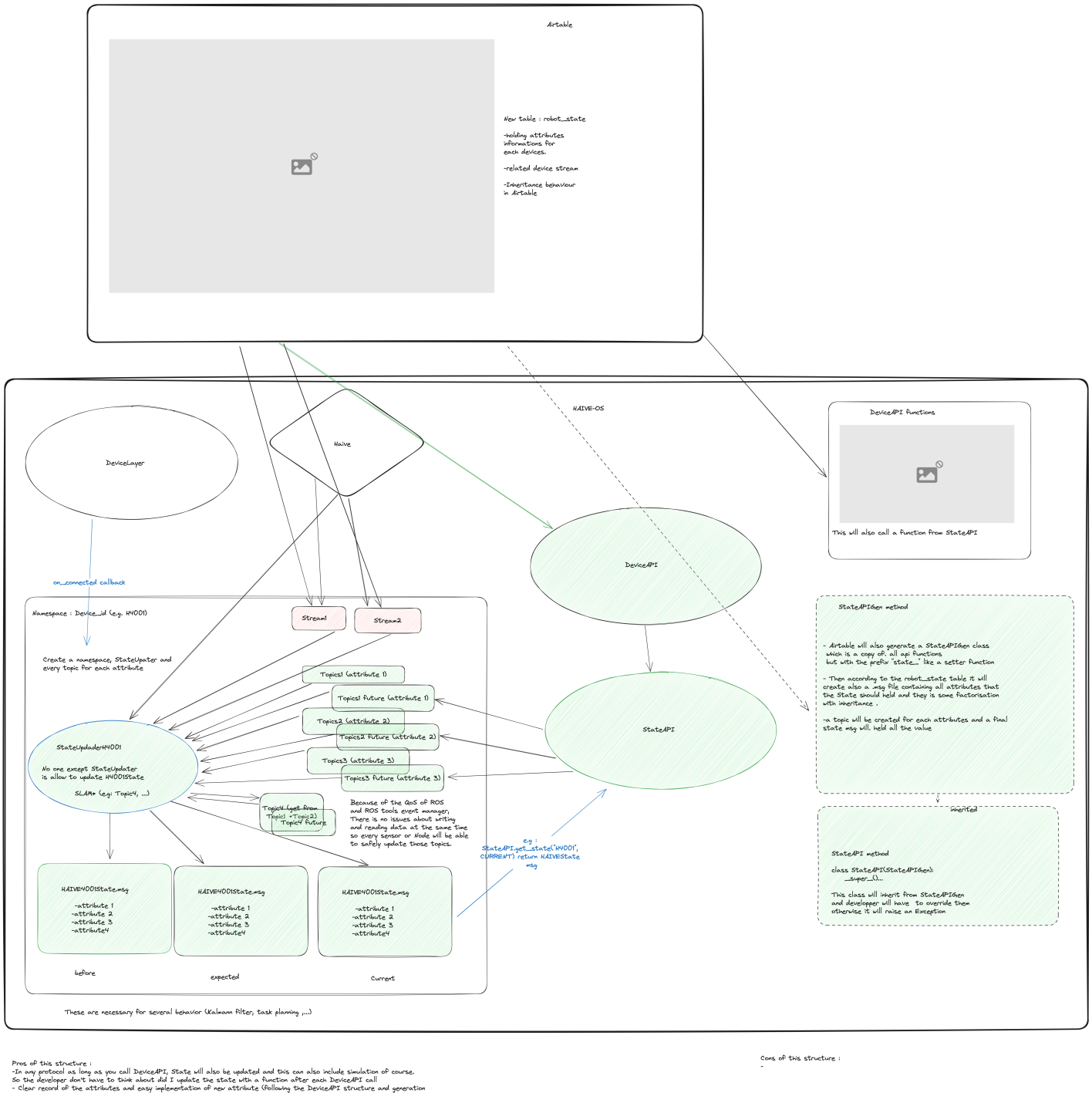


FIGURE 2.5 – Excalibur StateAPI pipeline

Chapitre 3

Calibration

Calibration for any robotics system is necessary to increase their accuracy which is mostly sensors and software-related, where precision/repeatability relies on the hardware structure.

When we started working for the HAIVE, the accuracy and precision of the system were low. that's why we started to propose a new calibration protocol.

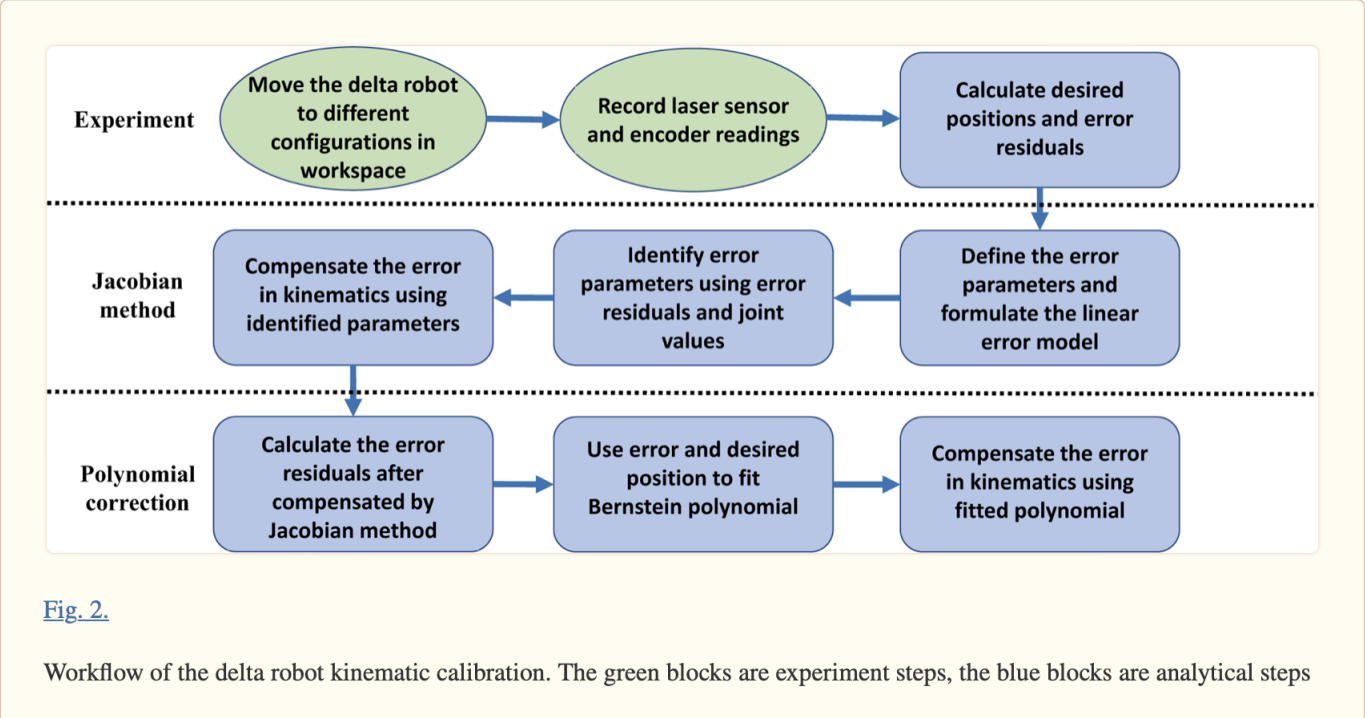


FIGURE 3.1 – Calibration road map

3.1 Previous calibration

The previous calibration used a developer tool and only neither the encoder's value nor the position sensors' value. The developer tools give the ability to a technician to manually calibrate the HAIVE. it will ask the technician using the HAIVE, to move the delta arm to the different configurations in the workspace (slots, container plate, tools), this would update the X, Y, and Z coordinates of the command because for each HAIVE it would have been different.

Considering the previous roadmap (Figure 3.1), we barely do the first and the third steps, skipping the second steps. So the first goal was to implement laser and encoders into the HAIVE. kinematics information wasn't updated and to make people realize this, we had to make an evaluation.

3.2 Evaluation

The evaluation procedure consists of estimating what the kinematics parameter uncertainty can imply for the delta arm position.

the current kinematics model relies on the intersection of 3 circles representing the range of each rod from each actuator :s

$$\begin{cases} (x-x1)^2+(y-y1)^2+(z-z1)^2 = R1^2 \\ (x-x2)^2+(y-y2)^2+(z-z2)^2 = R2^2 \\ (x-x3)^2+(y-y3)^2+(z-z3)^2 = R3^2 \end{cases}$$

with :

$$\begin{cases} x1 = cos(u).DeltaRodRadius1 \\ y1 = sin(u).DeltaRodRadius1 \\ x2 = cos(v).DeltaRodRadius2 \\ y2 = sin(v).DeltaRodRadius2 \\ x3 = cos(w).DeltaRodRadius3 \\ y3 = sin(w).DeltaRodRadius3 \end{cases}$$

but considering the previous equation, u,v, and w were considered equal, the same goes for DeltaRodRadius1, DeltaRodRadius2, and DeltaRodRadius3 were considered the same. Instead, it should have been considered different if we wanted the robot to be more accurate. Solving those previous equations led to these coefficients

$$a_{11} = 2(x_3 - x_1) \qquad a_{21} = 2(x_3 - x_2)$$
$$a_{12} = 2(y_3 - y_1) \qquad a_{22} = 2(y_3 - y_2)$$
$$a_{13} = 2(z_3 - z_1) \qquad a_{23} = 2(z_3 - z_2)$$
$$b_1 = r_1^2 - r_3^2 - x_1^2 - y_1^2 - z_1^2 + x_3^2 + y_3^2 + z_3^2$$
$$b_2 = r_2^2 - r_3^2 - x_2^2 - y_2^2 - z_2^2 + x_3^2 + y_3^2 + z_3^2$$

$$a_4 = -\frac{a_2}{a_1} \qquad a_5 = -\frac{a_3}{a_1} \qquad a_1 = \frac{a_{11}}{a_{13}} - \frac{a_{21}}{a_{23}} \qquad a_2 = \frac{a_{12}}{a_{13}} - \frac{a_{22}}{a_{23}} \qquad a_3 = \frac{b_2}{a_{23}} - \frac{b_1}{a_{13}}$$

$$a_6 = \frac{-a_{21}a_4 - a_{22}}{a_{23}} \qquad a_7 = \frac{b_2 - a_{21}a_5}{a_{23}}$$

$$a = a_4^2 + 1 + a_6^2$$
$$b = 2a_4(a_5 - x_1) - 2y_1 + 2a_6(a_7 - z_1)$$
$$c = a_5(a_5 - 2x_1) + a_7(a_7 - 2z_1) + x_1^2 + y_1^2 + z_1^2 - r_1^2$$

FIGURE 3.2 – System resolution part 1

Finally : In this equation configuration, y is the up vector, which means motors move along y where y1,y2, and y3, are the motor height value. if we talk about y in the equation/excel context, y is the

$(x-x_1)^2+(y-y_1)^2+(z-z_1)^2=r_1^2$ $(x-x_2)^2+(y-y_2)^2+(z-z_2)^2=r_2^2$ $(x-x_3)^2+(y-y_3)^2+(z-z_3)^2=r_3^2$				Current model :		New Model		Note : Interesting model error/issue given by David (xi,zi) are maybe yi dependent since the rail is maybe not straight
$y_{\pm}=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$	MOTOR Y1	400	X =	3.58882950090273	X =	5.21269817173947	Xdiff =	-1.62386867083674
$z_{\pm}=a_6y_{\pm}+a_7$	MOTOR Y2	700	Y =	913.852889216061	Y =	912.585595415032	YDiff =	1.267293801029
$x_{\pm}=a_4y_{\pm}+a_5$	MOTOR Y3	900	Z =	-290.000259240801	Z =	-287.235297495898	Diff =	-2.764961744903

FIGURE 3.3 – System resolution part 2

up vector, and outside this context, z is still our up vector. This is the main interface. y1,y2,y3 values give : The first X, Y,Z is the theoretical position considering previous equations The second (X', Y', Z') is the theoretical position considering error on xi,yi,zi, and ri (where i = 1,2,3) In the last column, we evaluate the difference between the 2 vectors. In this example, we considered only a 1 degree and 1mm uncertainty on each parameter such as DeltaRodRadius, u,v,w.

As a result, we can see that it can induce between 1 and 3 mm errors in our position. This condition can't be accepted in the context of conducting experiments with the Haive if we want to reproduce wall pipetting or surface dispensing.

3.3 New calibration

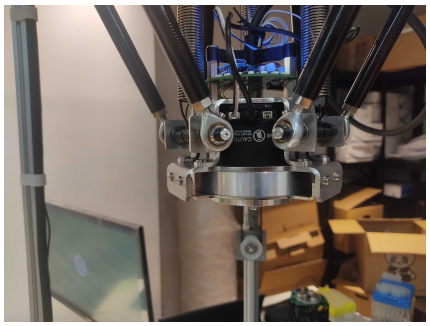
3.3.1 Laser and encoders

Sensors are necessary for any mobile robotics system, perception is a pillar of robotics systems. Encoder implementation was delegated to the electrician. PCBs have been replaced and the motor has been changed also accordingly.

Before implementing and buying any sensors, we had to design a protocol where lasers could give the X, Y, and Z positions. First, we bought a cheap laser and tried to deduce the arm position according to laser information. We record all the positions x, y, z, z1, z2, z3 (with z1,z2,z3 the height of the rods on pillars) and also we keep track of the command asked so it can be compared to measurement.

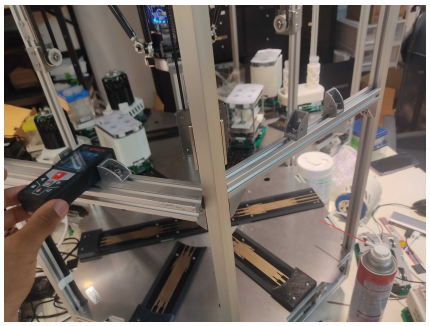
Calculation of the z position of the header is straightforward, but x and y are different because of the hexagon shape and the open environment, making it difficult to have any side surface contact, it also needs a plane surface perpendicular to the side laser.

From this need, we implemented side panels so that side lasers can touch a surface. That implies that the laser should be internal (along with the arm) This paper [3] shows an example of an external implementation. A big cube is attached to the header so that the laser can have a surface to detect. they are also using a microscope in addition. but in our case, we can't afford each HAIVE to have lasers, and making them external can be difficult because they are surrounded by other HAIVE, we have a little range to work with. and it would be difficult to touch a plane surface and can't afford to attach a cube to the header so that lasers have a surface to touch because it will considerably reduce the moving range

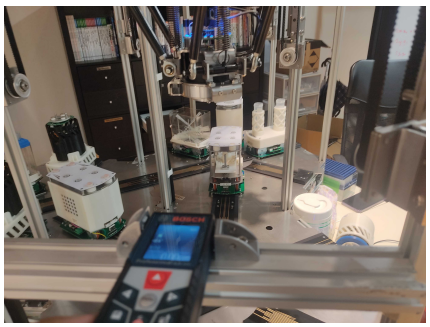


(a) Delta arm header

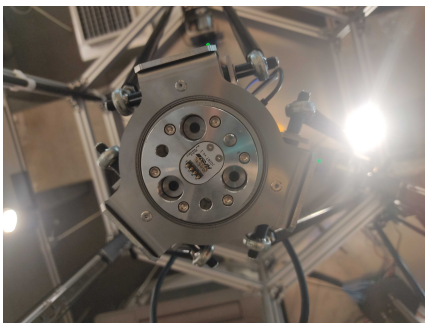
(b) Header height measurement



(c) X,Y header position reading 1



(d) X,Y header position reading 2



(e) Header view from bottom

(f) Rod height measurement

FIGURE 3.4 – Calibration proof of concept

because the cube will collide with pillars

Once the proof of concept was done, the laser should have been ordered, but at the moment the estimation cost was around 160 000¥, 30 000¥ x2 laser, and 100 000¥ for the laser sensors control box, thus it has been postponed. of course, other solutions have been investigated such as cameras.

3.3.2 Camera

Though our perception skills in the team couldn't afford to work with cameras, it would have been a cheaper and more secure way to calibrate. Indeed lasers can be harmful for bio lab experiments, and samples. As much as laser this method has been also evaluated at some point. we need to consider that from the camera we need to retrieve x,y, and z and there are several methods. the main ones would be : - Stereo camera, refers to a technique used in computer vision and perception systems to extract depth information from images by mimicking the human visual system's process of perceiving depth through binocular vision. only 1 camera would have given only x, and y information but the stereo camera allows to give also the z information

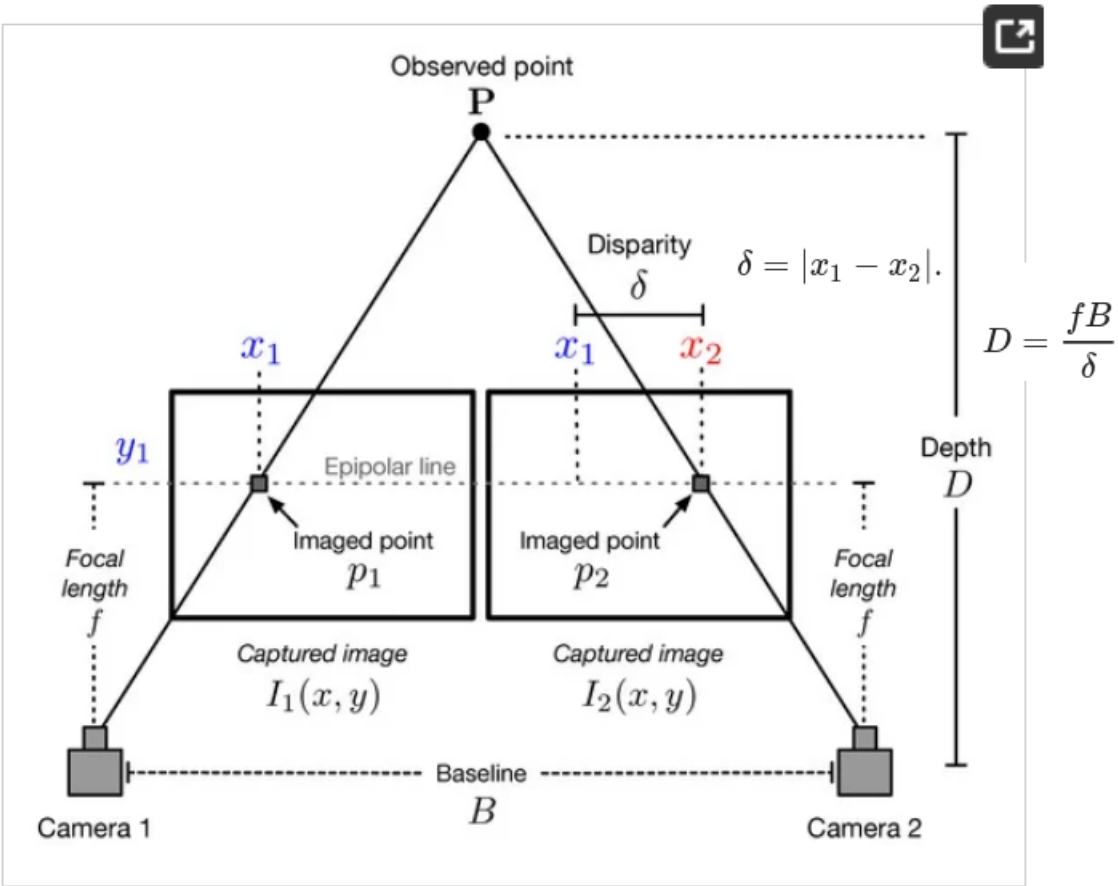


FIGURE 3.5 – Stereo camera method

- Mathematical model linking the coordinates of a point in three-dimensional space three-dimensional space and its projection onto the image plane of an ideal pinhole.

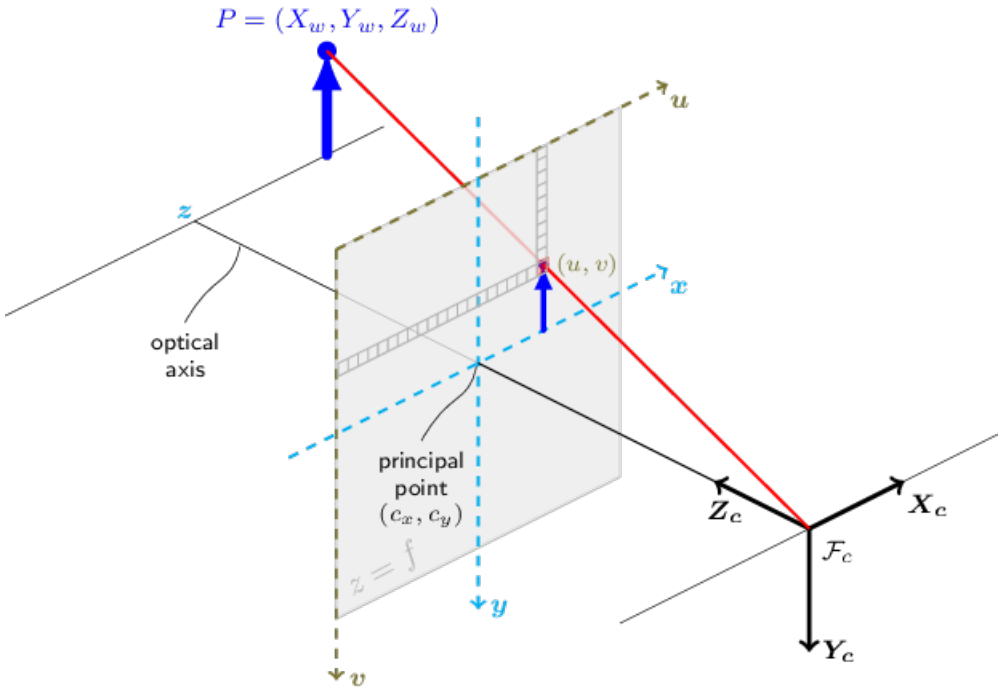


FIGURE 3.6 – pinhole method

3.3.3 Implementation

In collaboration with the hardware and electronician engineer, we designed an add-on/tool header that could be fixed to the HAIVE having serial port and alimentation.

the micro-controller used was a Seeed Studio XIAO SAMD21 and here the firmware architecture :

The WiFi communication between the HAIVE-OS and the M5Stick is handle by the hos_client as refer this chapter 2.4. Once the command receive it will be parsed and execute some action. the main main will be :

- Check laser condition (connected or not and potential errors)
- Get lasers distance value, sending then distance and light info to the HAIVE-OS, reflection condition can be important either so estimate which kind of material we hit with the laser, or to ensure that the accuracy is good or not, indeed low light information would mean that we can't really trust the laser acquisition.

The entire code has been design to have non-blocking loop, including the Serial reading or calculation allowing to thread every communication, each serial reading will have a buffer to keep the current result. When we ask for laser distance, we wait for both Serial reading to be done before sending back to the HAIVE-OS. it will also allow to not block any communcation such as an emergency stop while it's trying to read another serial port.

3.3.4 Result

As a result, through iterations we can see that the x and y axis can have up to 4-5 mm difference between the command and the measurement, considering the 1 mm accuracy of the sensors it can't be only the sensor uncertainty.

After this so far, the more accurate lasers couldn't be ordered.

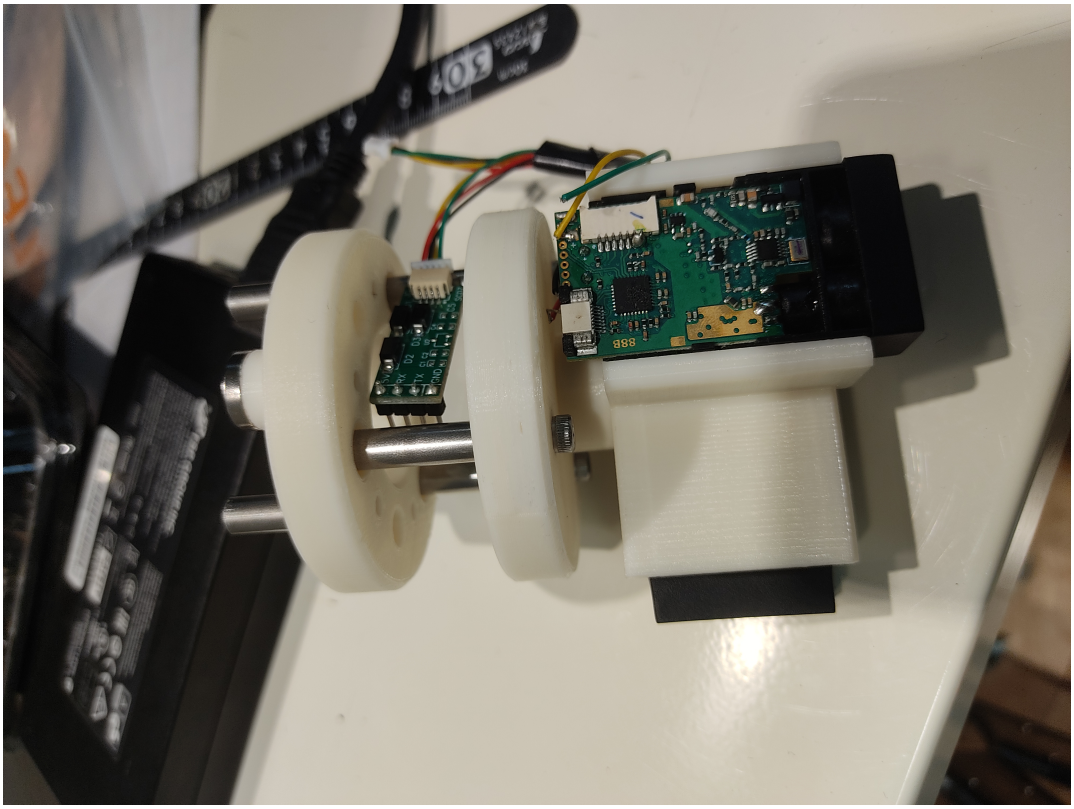


FIGURE 3.7 – Laser header module

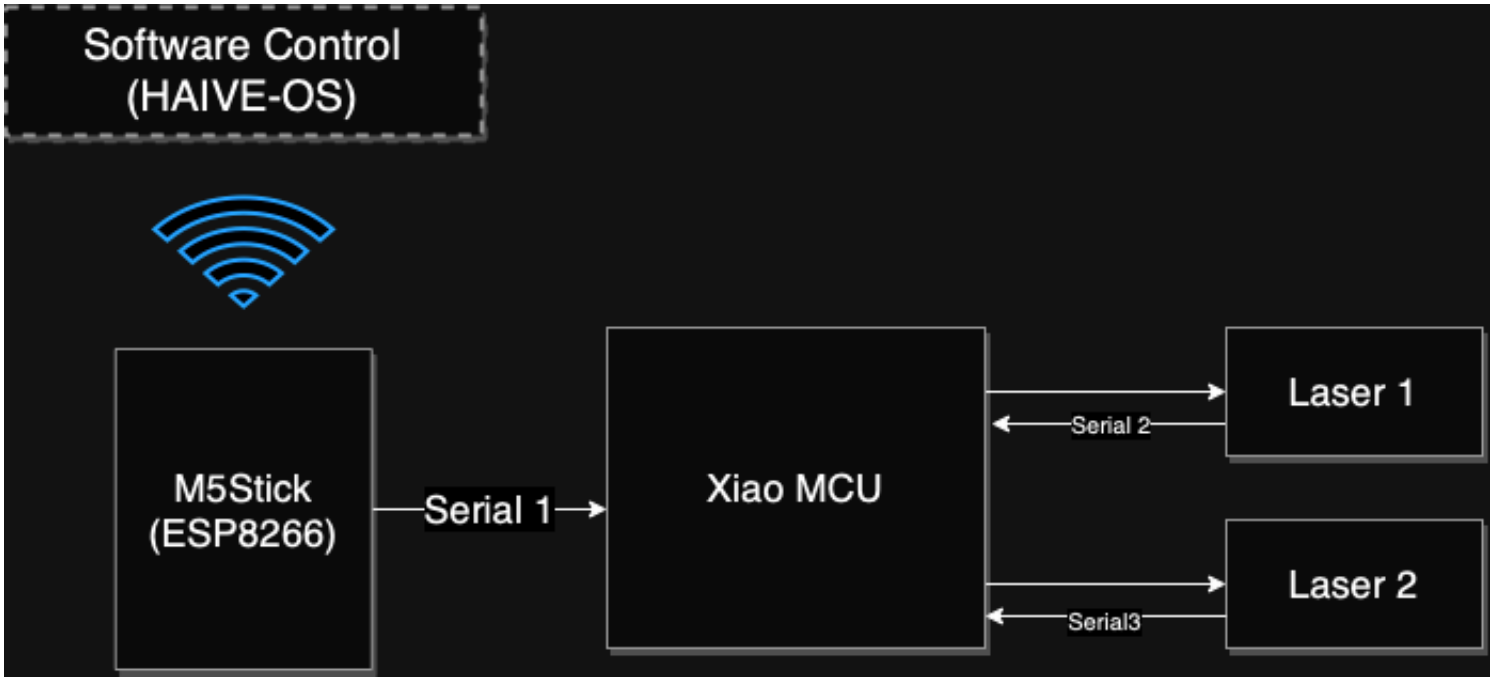


FIGURE 3.8 – Firmware architecture for lasers

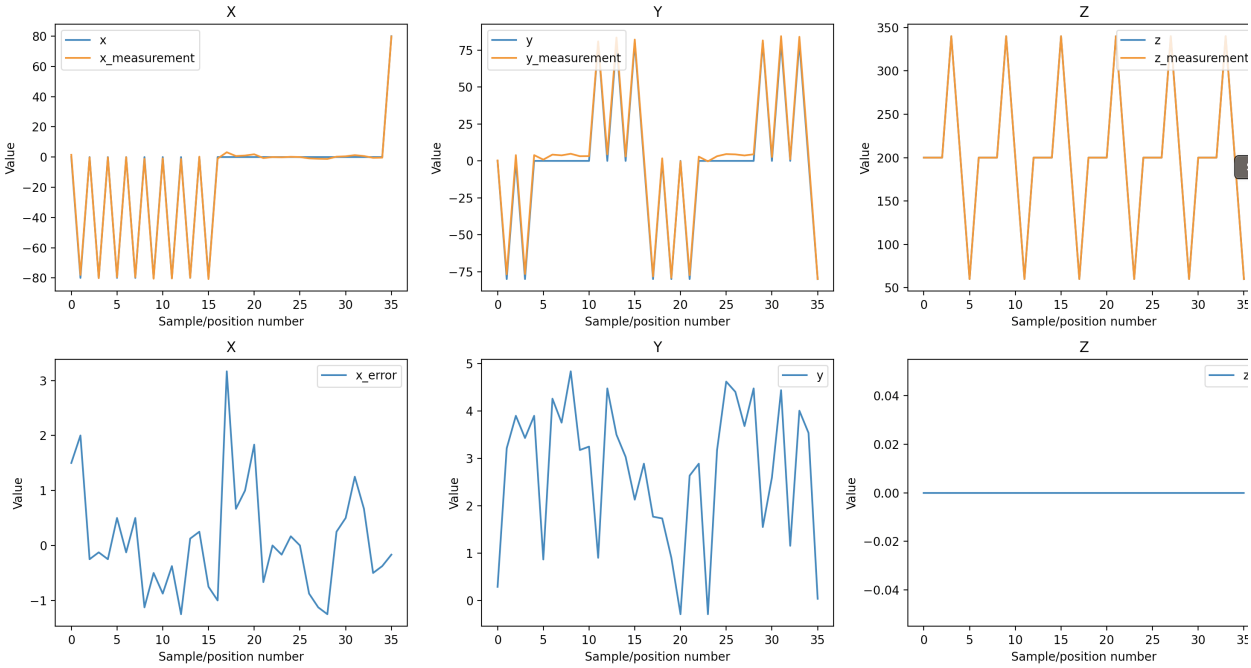


FIGURE 3.9 – Stereo camera method

Chapitre 4

Software development

For the software team development, We were using Jira, Bitbucket, and Sourcetree, allowing us to have an easy Pull Request (PR) environment, Kanban with tickets to keep properly track of tasks and their dependencies.

4.1 Management

Jira, Github and Bitbucket were the main software tools for developpement. Github and Bitbucket are similar, though Bitbucket offers a cleaner interface for PR. Github was used as a release repository and Bitbucket as a development repository. Sourcetree is just personal software that gives a UI interface for GitHub commands. Despite having the UI, working with RaspberryPi we also had to use terminal command properly, create branches and handle 2 remote repositories.

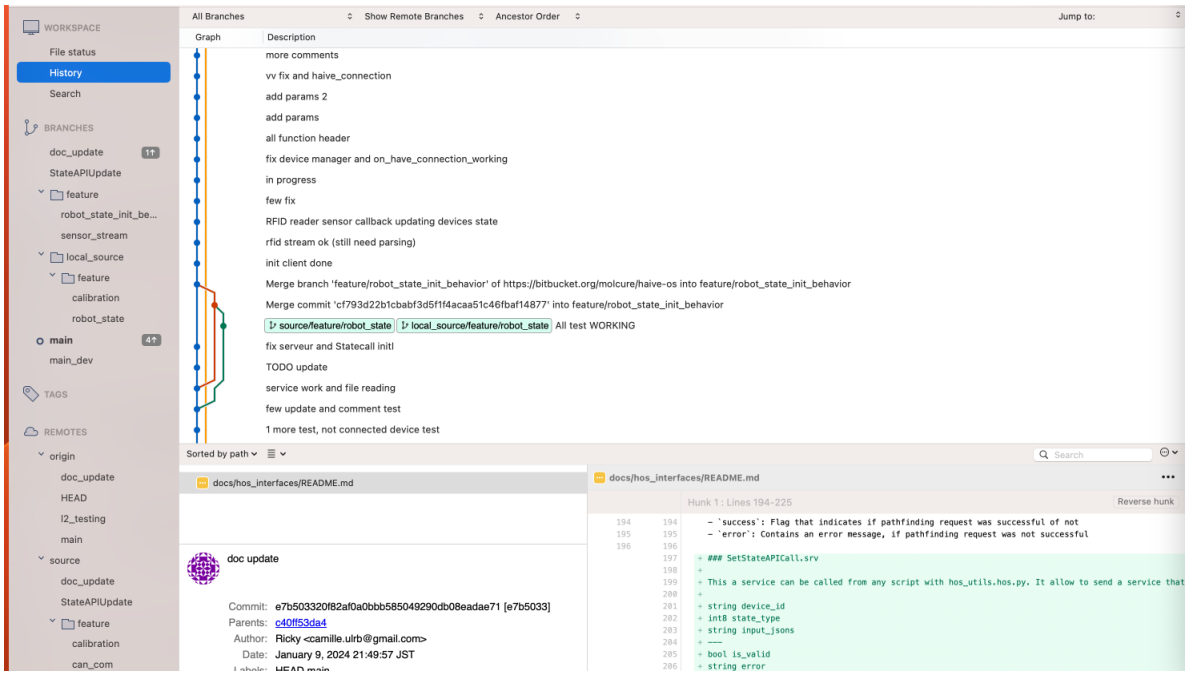


FIGURE 4.1 – Sourcetree application : Github desktop UI

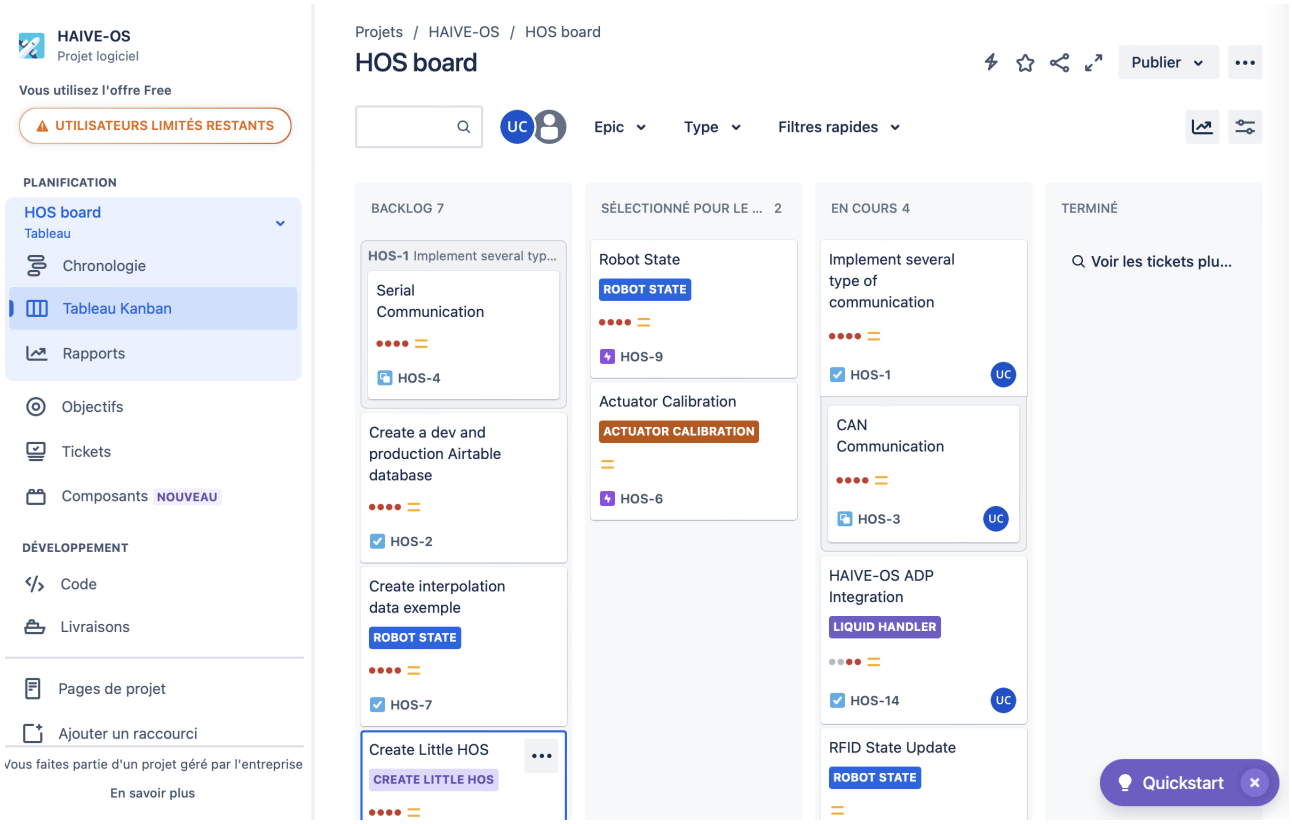


FIGURE 4.2 – Kanban of HAIVE-OS project

4.2 Test

Differentiation between the Unit test and Functional test is important for testing behavior. In the end, both are needed to validate a solution but the functional test are upper-level test (achieving or not the target/function), whereas the unity proves the stability and integrity of the system. here are some example from the HAIVE-OS project :

Functionality/Upper level test : We need to execute a protocol and since the StateAPI updater will be linked to the DeviceAPI, the protocol just needs to hold DeviceAPI calls and then by human confirmation ensure that the state represented is the state of the robot in real-time like container position, liquid volume, ... (without sensors first).

- Functional test 1
  - input : DeviceAPI protocol
  - parameter : None
  - expected output : container theoretical state space representation is updated

  This test should give validation to the estimator architecture sensors needed to feed information into the robot state for example, an RFID Reader should be able to detect a container position (this can be both true for DeltaHAIVE and HAIVE Cube) and update container position info in the robot state
- Functional test 2
  - input : device moving from a slot to another.
  - parameter : device_name, slot a, slot b.
  - expected output :device state space representation is updated (slot_position attribute)

  This test should give a validation to the observer architecture In the end we want to be able to

create dynamic protocol

- Functional test 3
  - input : a "dynamic" protocol script/feature
  - parameter : protocol and 2 different initial state for device
  - expected output : record all the Device API call and compare

  the protocol for those 2 different initial state and ensure that those records are different showing that the protocol was dynamics and reacting to devices status.

Unit test : in the case of the HAIVE-OS, Unit tests could be generated automatically along with the StateAPI and DeviceAPI, ensuring future proof development.

- Unit test 1
  - input : 1 DeviceAPI
  - parameter : DeviceAPI args
  - expected output : depend on DeviceAPICall Test if StateAPI is called along with DeviceAPI
- Unit test 2
  - input : 1 DeviceAPICall
  - parameter : DeviceAPI args expected output : time estimation for the service call, and ensure that the right device is update and call state_updater and ros event_call.
- Failure check 1
  - input : updating a non-existent device
  - parameter : None
  - expected output : WARNING message and different behavior

  if device not in fleet and not connected : refuse state update if device in fleet and not connected : connection warning and estimated update if device not in fleet and connected : accept state update as long as it is DeviceState.msg attributes

Bibliographie

[1] https ://www.embopress.org/doi/full/10.15252/msb.202010019

[2] https ://www.researchgate.net/figure/Observateur-detat-pour-un-systeme-lineaire-La-dynamique-de-lerreur-dobservation-x_fig20_303446268

[3] https ://www.ncbi.nlm.nih.gov/pmc/articles/PMC9484559/

[4] https ://medium.com/@satya15july_11937/depth-estimation-from-stereo-images-using-deep-learning-314952b8eaf9