ASSIGNMENT 3: INTEGRATION / CONTROLLING A ROBOT.

Bishoy Essam Samir Yassa Gerges, bishoy.gerges@student.utwente.nl, s2921073, M-ROB
Panteleimon Manouselis, p.manouselis@student.utwente.nl, s3084493, M-ROB

3.1 – JIWY SYSTEM ARCHITECTURE

3.1.1 – BLOCK DIAGRAM & 3.1.2 – DETAILS OF EACH BLOCK

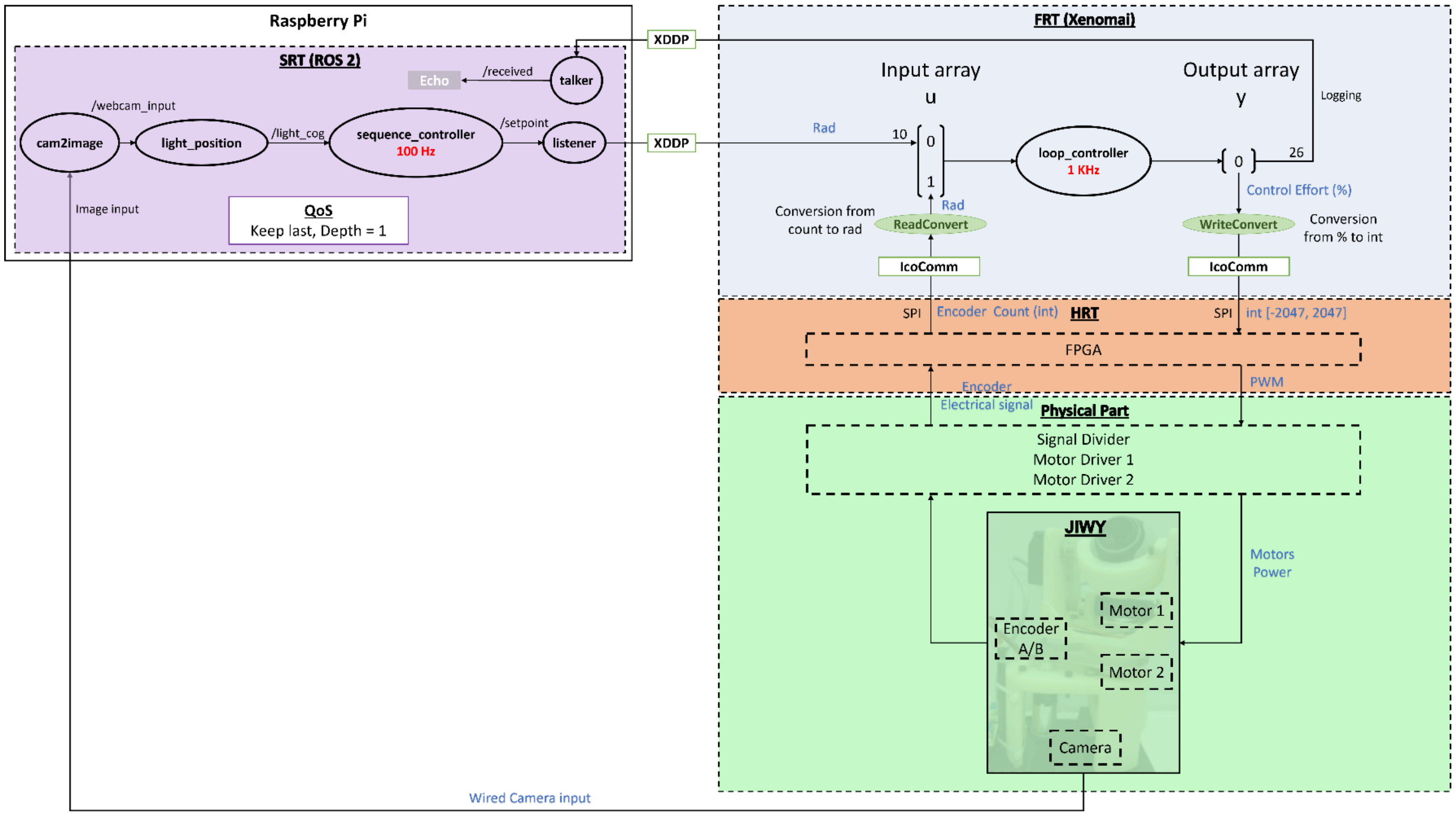


Figure 1: Block diagram with details given to each block/signal.

The detailed block diagram is shown in Figure 1,

- The details of the blocks and signals are shown in the Figure.
- The image processing algorithm is running on ROS 2, the topics are shown in the diagram. This means that it is running in a soft real-time manner. This is because image acquisition (~33 ms) and processing take time thus, it is more suitable to run it on ROS 2. However, a QoS of keep-last and depth of 1 is used to decrease latency as much as possible.
- The loop_controller runs in Firm Real Time on Xenomai kernel. The ReadConvert and WriteConvert functions are used to map inputs and outputs of the controller to the expected types by the controller and FPGA.
- Communication between Xenomai and the FPGA is done using IcoComm which interfaces the SPI protocol. Communication between Xenomai and ROS is done via XDDP protocol, the port numbers are illustrated in the block diagram. The talker and listener ROS nodes available in the given framework, interface the XDDP communication to and from ROS.

- To make the block diagram simple, we include the diagram for one controller (tilt/pan). It will be the same block diagram (only port numbers changed) for the other controller.
- The sequence controller was chosen to have 100 Hz. This is motivated as follows: the image acquisition takes about 33 ms, this means that the sequence controller period time should be lower than this value. In addition, by looking into these lines of code of the sequence controller:

```
double ddt_setpoint_x = 1/tau * cog_x; // Derivative
double ddt_setpoint_y = 1/tau * cog_y; // Derivative

// Forward euler integration
setpoint_x += ddt_setpoint_x * ToSeconds(sampleTime);
setpoint_y += ddt_setpoint_y * ToSeconds(sampleTime);
```

  and by taking into consideration that the value of tau = 1, then setpoint equals:

$$\text{setpoint} += \frac{cog}{frequency}$$

  So, actually the trajectory to this cog will be reached after a number of steps equal to the frequency of the node. In other words, the frequency of the node determines how smooth the trajectory to the light spot is. Based on the previous, a sampling period of 10 ms was chosen (100 Hz) which is nearly 10 times faster than the image acquisition speed. This means that for each image (and a corresponding cog), the sequence controller produces three points in the trajectory to this cog.

3.2 – INTEGRATION

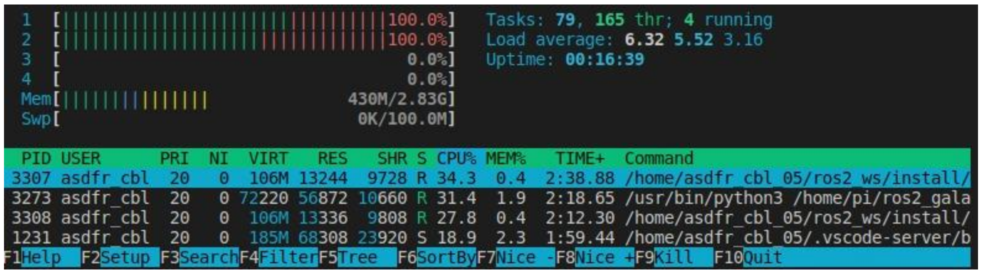
In this sub assignment we develop the missing blocks from the diagram created in Assignment 3.1 and integrate everything into a working product.

3.2.1 – ROS2 ON THE RASPBERRY PI

This section will focus on ROS2 implementation on the Raspberry Pi. To begin with, the experiment from Assignment 1 were reproduced on the Raspberry Pi. The nodes developed in Assignment Set 1 were re-used for this purpose (except the closed loop controller which was provided to us as we are CBL students). The Jiwy USB camera was connected to the Raspberry Pi, and the graphics stream was forwarded to our laptop using X-forwarding as the Raspberry Pi does not have a graphical environment.

The system was tested to ensure that it works as intended. The ROS2 implementation on the Raspberry Pi successfully communicated with the other nodes in the system, and the video stream from the camera was transmitted to the laptop.

The processor load of the Raspberry Pi (`htop`) can be seen in the figure below:



```
  1  [||||||||||||||||||||||||||||||||||100.0%]   Tasks: 79, 165 thr; 4 running
  2  [||||||||||||||||||||||||||||||||||100.0%]   Load average: 6.32 5.52 3.16
  3  [                                    0.0%]   Uptime: 00:16:39
  4  [                                    0.0%]
  Mem[||||||||||||||||             430M/2.83G]
  Swp[                              0K/100.0M]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 3307 asdfr_cbl  20   0  106M 13244  9728 R 34.3  0.4  2:38.88 /home/asdfr_cbl_05/ros2_ws/install/
 3273 asdfr_cbl  20   0 72220 56872 10660 R 31.4  1.9  2:18.65 /usr/bin/python3 /home/pi/ros2_gala
 3308 asdfr_cbl  20   0  106M 13336  9808 R 27.8  0.4  2:12.30 /home/asdfr_cbl_05/ros2_ws/install/
 1231 asdfr_cbl  20   0  185M 68308 23920 S 18.9  2.3  1:59.44 /home/asdfr_cbl_05/.vscode-server/b
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figure 2: Raspberry Pi processor load

Considering the Raspberry Pi has only two cores, it is apparent that performing image processing tasks places a significant strain on the device.

As the Raspberry Pi does not have `rqt` installed, in order to visualize the blocks of our system we reproduced the experiment locally (in our machines). The results can be seen in the following two Figures.

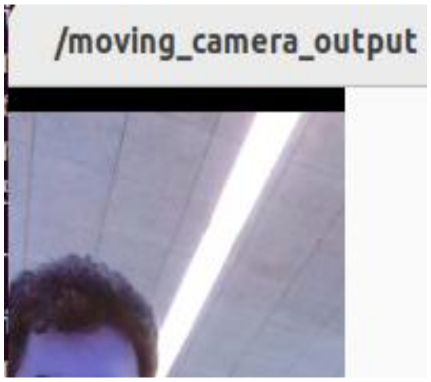


Figure 3:Ouput of camera after it has moved to the COG (i.e., lamp)

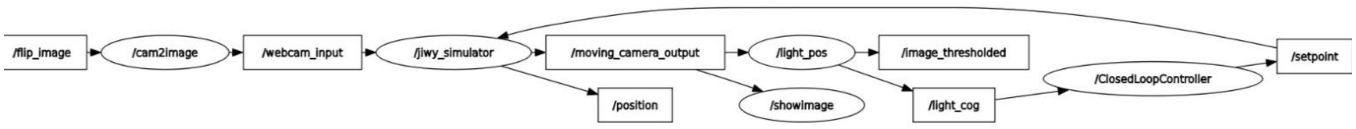


Figure 4: Nodes and topics of our system

Does the system perform as well as on your own laptop? Why? What is the processor load?

How well each system performs (i.e., how well the tracking of the COG was performed) is mainly dependent on the time constant (τ). This parameter defines how quickly the system can respond to changes in input (the smallest the time constant the fastest the response). When testing the system on both a laptop and the Raspberry Pi, using the same time constant, it was observed that the Raspberry Pi had a slower response time compared to the laptop. Specifically, the tracking of the center of gravity was slower, and the motion of the image appeared more sluggish. Furthermore, based on our observations of the processors load (`htop`) we saw that the processor load on the Raspberry Pi was higher compared to the laptop. This is because the Raspberry Pi had to work harder to execute the same tasks as the laptop, due to its lower processing power. Other processes running simultaneously on the Raspberry Pi had a significant negative impact in its performance.

Did you have to make changes in the nodes/experiment you developed in Assignment 1.2.3? If so, why was it necessary? Could you have avoided it if you had developed the nodes in Assignment 1.2.3 differently?

As we were CBL students, the node developed in Assignment 1.2.3 was provided to us. However, we did have to make changes in multiple other nodes we created in Assignment 1. More specifically, both the `brightness node`, which defined whether there is light or not in the image, and the `light position node` which calculated and displayed the COG had to be changed as they were heavily reliant in the libraries OpenCV and CV Bridge. Those two libraries were used for image processing and conversion of ROS image messages to OpenCV images, respectively. Unfortunately, these libraries were not supported by the Raspberry Pi in the lab and therefore the nodes had to be adapted.

3.2.2 – CODE GENERATION OF THE POSITION CONTROLLER FROM 20-SIM

The Tilt/Pan controllers were provided to us in the 20-sim software and now need to be implemented in a Xenomai thread. To achieve this, C-code was generated from the 20-sim model and then integrated using the framework from Meijer [1]. The generated code was then modified to work within the Xenomai environment.

In this part of the assignment, we focused on ensuring that the module runs smoothly within the Xenomai environment. To achieve this, we created a test system consisting of the following components:

1. Two `Listener nodes`, which receive input from outside the module. The first listener node produces the tilt setpoint, while the second listener node produces feedback from the plant. Since this is a test system, we haven't yet connected our plant (Jiwy Camera), so we produce our own feedback. If the feedback is the same as the tilt setpoint, and hence the error is zero, we expect that our controller will output a value close to zero.
2. Two buffer arrays named u and y. The array u has size $2 \, x \, 1$ and stores the current tilt setpoint (in radians) in its first element and the feedback (in radians) in its second element. The array y has size $1 \, x \, 1$ and stores the output of the Controller (values range from -100 to 100 indicating a percentage).
3. The Controller that was generated from 20-sim (`ControllerTilt`) which accepts as input the first element of the buffer array u and its output is written in the buffer array y. `ControllerTilt` is a PID controller that outputs the desired motor power in percentage.
4. A `talker node` (again counterintuitive as it actually “listens” to the output). This node receives the output of the controller in the form of a percentage.

In order for our listener and talker ROS2 nodes to communicate with the Xenomai, we utilized the XDDP communication protocol. XDDP is a communication protocol between ROS2 and Xenomai where data is exchanged through shared memory buffers (in our case arrays $u \, and \, y$). More specifically, we created unidirectional (one-way) communication channels using the XDDPComm class. This class takes an array of element parameters, an input and an output port, and the number of element parameters. The element parameters define on which index of the buffer array the incoming/outgoing data is set or taken from [1]. The implementation can be seen in the following code snippet:

```
    frameworkComm *controller_tilt_uPorts [] = {
new XDDPComm(10, -1, 1, xddp_uparam_setpoint_tilt),
new XDDPComm(11, -1, 1, xddp_uparam_feedback_tilt),
                            };

    frameworkComm *controller_tilt_yPorts [] = {
 new XDDPComm(26, 27, 1, xddp_yparam_logging_tilt),
};
```

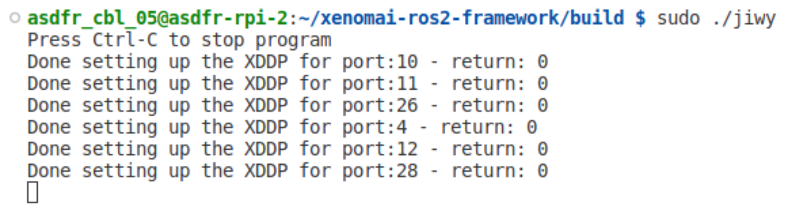
To create our controller in a form that can be run by a Xenomai Thread we use the following code snippet:

```
                ControllerTilt *controller_tilt = new ControllerTilt;
runnable *controller_tilt_runnable = new wrapper<ControllerTilt>(controller_tilt,
                controller_tilt_uPorts, controller_tilt_yPorts,2, 1);
```

Here the 20-sim generated class `ControllerTilt` is used to declare a pointer variable of the same class. Then we wrap this newly created controller into a runnable class. This step is necessary if we want to run our controller in a Xenomai real time thread as the interface between Xenomai and our class is provided through the runnable class. The wrapper class is used in the creation of the runnable in order to specify which controller we are going to use, the frameworkComm class from which the controller will receive an input (effectively the buffer array u), the frameworkComm class to which the controller will output (effectively the buffer array y) and the length of each buffer array.

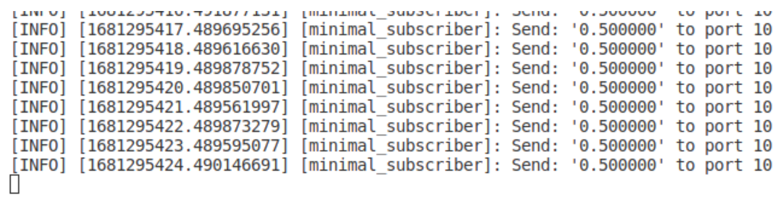
Showing that the system works:

We proceeded with the testing of the aforementioned test module, which includes the listeners, buffer arrays, controller, and talker. The results are illustrated in the following four Figures:



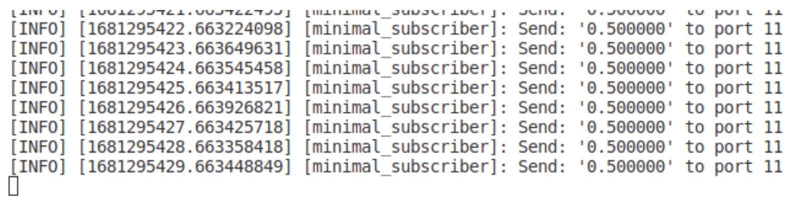
```
asdfr_cbl_05@asdfr-rpi-2:~/xenomai-ros2-framework/build $ sudo ./jiwy
Press Ctrl-C to stop program
Done setting up the XDDP for port:10 - return: 0
Done setting up the XDDP for port:11 - return: 0
Done setting up the XDDP for port:26 - return: 0
Done setting up the XDDP for port:4 - return: 0
Done setting up the XDDP for port:12 - return: 0
Done setting up the XDDP for port:28 - return: 0
```

Figure 5: Running program, communication ports open successfully.



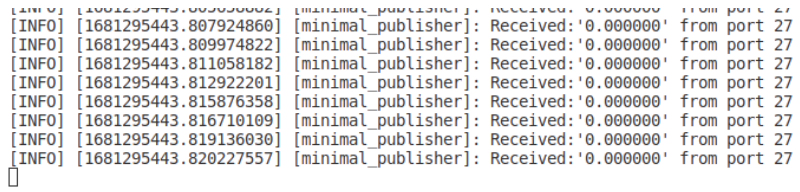
```
[INFO] [1681295417.489695256] [minimal_subscriber]: Send: '0.500000' to port 10
[INFO] [1681295418.489616630] [minimal_subscriber]: Send: '0.500000' to port 10
[INFO] [1681295419.489878752] [minimal_subscriber]: Send: '0.500000' to port 10
[INFO] [1681295420.489850701] [minimal_subscriber]: Send: '0.500000' to port 10
[INFO] [1681295421.489561997] [minimal_subscriber]: Send: '0.500000' to port 10
[INFO] [1681295422.489873279] [minimal_subscriber]: Send: '0.500000' to port 10
[INFO] [1681295423.489595077] [minimal_subscriber]: Send: '0.500000' to port 10
[INFO] [1681295424.490146691] [minimal_subscriber]: Send: '0.500000' to port 10
```

Figure 6: Data send from first listener to first element of buffer array u (i.e., tilt setpoint)



```
[INFO] [1681295422.663224098] [minimal_subscriber]: Send: '0.500000' to port 11
[INFO] [1681295423.663649631] [minimal_subscriber]: Send: '0.500000' to port 11
[INFO] [1681295424.663545458] [minimal_subscriber]: Send: '0.500000' to port 11
[INFO] [1681295425.663413517] [minimal_subscriber]: Send: '0.500000' to port 11
[INFO] [1681295426.663926821] [minimal_subscriber]: Send: '0.500000' to port 11
[INFO] [1681295427.663425718] [minimal_subscriber]: Send: '0.500000' to port 11
[INFO] [1681295428.663358418] [minimal_subscriber]: Send: '0.500000' to port 11
[INFO] [1681295429.663448849] [minimal_subscriber]: Send: '0.500000' to port 11
```

Figure 7: Data send from second listener to second element of buffer array u (i.e., simulated feedback)



```
[INFO] [1681295443.807924860] [minimal_publisher]: Received:'0.000000' from port 27
[INFO] [1681295443.809974822] [minimal_publisher]: Received:'0.000000' from port 27
[INFO] [1681295443.811058182] [minimal_publisher]: Received:'0.000000' from port 27
[INFO] [1681295443.812922201] [minimal_publisher]: Received:'0.000000' from port 27
[INFO] [1681295443.815876358] [minimal_publisher]: Received:'0.000000' from port 27
[INFO] [1681295443.816710109] [minimal_publisher]: Received:'0.000000' from port 27
[INFO] [1681295443.819136030] [minimal_publisher]: Received:'0.000000' from port 27
[INFO] [1681295443.820227557] [minimal_publisher]: Received:'0.000000' from port 27
```

Figure 8: Data that talker node receives from buffer array y (i.e., output of controller)

As the controller's output indicates the power percentage, we anticipated that when the setpoint and feedback values were the same, the controller would output 0. This is what we also observed from our experiments. More specifically, when the tilt angle was equal to 0.5 radians and the feedback angle was also equal to 0.5 radians, the output was 0.

What are the advantages and disadvantages of doing code generation (as opposed to writing the controllers directly in C/C++)?

Advantages:

- Time-saving: Code generation can save an amount of time and effort from the user, as it allows for the automatic generation of code based on the design of the system. This can also help speed up the development process and reduce the time required to get the system up and running.
- Reduces errors: Code generation tools can minimize the potential for coding errors, as they provide a standardized and automated process for generating code. This can help to reduce the number of coding errors and improve the reliability of the system.
- Reusability: Code generated from modeling and simulation tools can be reused across different platforms and environments. This can make it easier to develop and maintain systems across different hardware and software platforms.

Disadvantages:

- Limited customization: Generated code can be limited in terms of its customization compared to manually written code. This can make it difficult to implement certain features or make changes to the code in the future.
- Over-reliance on code generation tools: Relying too heavily on code generation tools can lead to a lack of understanding of the underlying code and the system as a whole. This can make it difficult to troubleshoot issues or make changes to the system in the future.
- Cost: Code generation tools can be expensive. Furthermore, training staff to use code generation tools can also be costly.

3.2.3 – MEASUREMENT AND ACTUATION BLOCK

For measurement, the ReadConvert function is written to implement the following equations:

$$\text{current angle} = \text{previous angle} + (\text{current encoder count} - \text{last encoder count}) \times \text{rad per count}$$

where rad per count = 2*pi/2000 for the tilt and 2*pi/(4*1250) for the pan. These values were obtained using the info provided in Appendix H of the manual. In addition, over/underflow detection is implemented as follows:

```
if (current_count - last_count > 300){
real_current_count = -1 - ((int)current_count % (int)max_count);
}else if (current_count - last_count < -300){
real_current_count = max_count + ((int)current_count % (int)max_count) + 1;
}else{
real_current_count = current_count;
}
```

where this piece of code detects a sudden jump (here a jump of 300 counts or more) as an indication of over/underflow. If this occurred, then it means that the encoder count went above the maximum value or below the minimum value.

The WriteConvert function was simply implemented to map the control effort from percentage (%) to int values in the range [-2047, 2047] which was mentioned in the manual Appendix H, that is:

$$\text{output mapped} = (\text{int})(\text{output}(\%) * 2047)$$

To test measurement and actuation, a simple 20-sim block was designed. This block takes two inputs and passes them to two outputs (first input to first output and second input to second output).

The first input takes its value by manually sending a known value from the ROS side (via XDDP). This value will be stored in the u array of this block and then passed to the first output of the y array. This output will be sent via IcoComm thus it will be passed through WriteConvert method. An oscilloscope will then be used to measure the sent PWM signal which was set by the WriteConvert method and see if it meets our expectation according to the sent value from the ROS side and the implemented equations in the WriteConvert method. Sending from the ROS side is by using the listener node.

The second input of the block is set by IcoComm via SPI which goes first through the ReadConvert, stored in the u array and passed to the second output of the y array. This time, this output is logged (via XDDP) to ROS. We can then see the logged value if it meets our expectation according to the implemented equations of the ReadConvert function.

Implementing the Measurement and Actuation block by 'embedding it in the Communication class', is just one way to do it. Describe three alternatives (including the one just mentioned) to implement the M&A block in the framework you are using.

1. Method 1: adjust the input and output type/mapping inside the controller itself. That is, during the design of the controller in 20-sim before exporting it to C++ code.
2. Method 2: to design the input and output mapping on the FPGA.
3. Method 3: is the one used here which is to implement the input and output mapping in the communication class.

Discuss advantages and disadvantages.

Method 1 is easier to implement if it does not need to be maintained. That is, if the mapping needs to be changed from time to time, it is not the best method to use. In addition, usually the control algorithm is designed by a control engineer or a person different from the one implementing the algorithm on the software-hardware architecture. So, this method will hinder working of both persons in parallel because the CPS architecture will affect the work of the control engineer.

Method 2 has the advantage of high computation speed. However, it lacks modularity and maintainability. This is because the FPGA will need to be reprogrammed for each application. In addition, instead of focusing on only one piece of development at a time (Raspberry pi only for example), the focus will be also on the FPGA programming.

Method 3 has the advantage of modularity. That is, it is very flexible to be changed and adapted to different applications without effort. However, it may hinder the speed of the communication if the M&A

calculations are complicated or lead to errors that can cause large or even infinite runtime of the mapping functions leading to the malfunctioning of the whole program.

Is the chosen implementation your preferred one? Why?

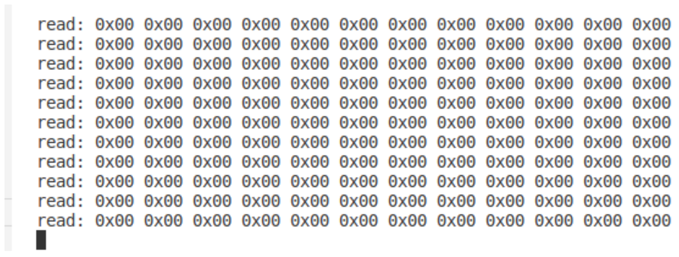
Yes, because it can be setup and tested easily. If the required mapping is changed for any reason, it will be easy to implement that by just changing the WriteConvert and ReadConvert methods. In addition, no additional knowledge is needed (for example, no extra programming on the FPGA is needed using VHDL).

3.2.4 – FINAL INTEGRATION

The full integration is composed of the previously mentioned components together:

- The controllers C++ code was exported from the 20-sim model and imported inside the framework to be used.
- For each controller (pan/tilt), communication protocols was setup as follows: each controller has two inputs, one input is the set point coming from ROS via XDDP protocol and another input coming from the feedback of the JIWY encoder via IcoComm communication (interfacing the SPI). Each controller has one output which is the control effort in percentage which is then sent via IcoComm to the FPGA. An XDDP communication is also set up for this output to log it to ROS for debugging. This is summarized in the block diagram in Figure 1.
- The mapping as described in 3.2.3 is implemented using the WriteConvert and ReadConvert functions. Each controller has its pair of functions.
- Thread creations in Xenomai and cleaning after exiting are implemented by following the MSc report from Meijer [1].

We did not manage to make the full system work, but we managed to run the ROS part alone as mentioned in 3.2.1 in addition to attempting to get data from the JIWY encoders. The read data from the encoders were as follows:



```
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
read: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
```

Figure 9: SPI input readings.

Actually, the data printed in Figure 9 are the 12-size array retrieved from the SPI communication using IcoComm where encoder 1 (tilt) and encoder 2 (pan) readings are elements number 0 and 3 respectively in the array. As shown in the figure, the readings are all zeros. We tried changing the camera position by hand, but the values did not change. We did not have enough time to debug the problem, specially that it could be a problem with the hardware setup as well.

Compare the resulting system with the block diagram from Assignment 3.1. Are there any differences? If so, why did you deviate from the original plan?

The code is an implementation of the block diagram without differences. Actually, the final block diagram (the one submitted in this report, not the one from Assignment 2) was implemented after going through the code and finalizing the overall system so that the block diagram gives and overview of the whole architecture.