Table of Contents

PART 1

➢ 3.1.1

We extended the timer model from exercise 2 so that we can run it on the Xenomai device. The timer was set to 0.01 sec . In order to measure the execution time using the oscilloscope we further extended the code on the producer’s side, in a way that it will generate a periodic square signal that will go from high to low every 0.01 sec. This signal is what we display on the oscilloscope.

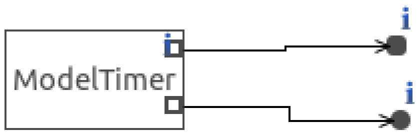


Figure 1. Architecture model . One port is for the timer and the other for the oscilloscope

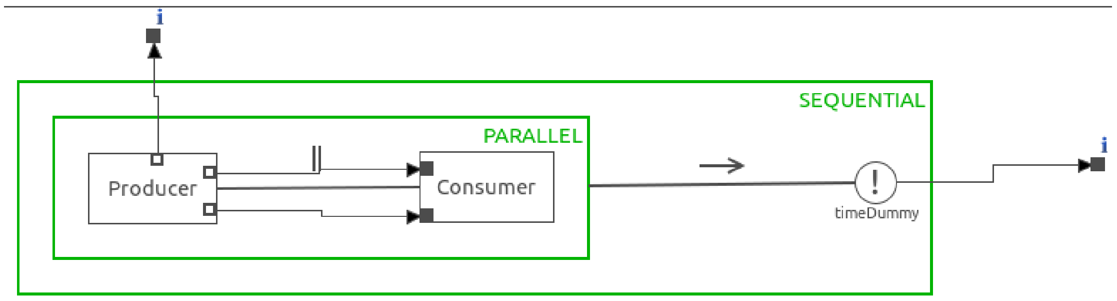
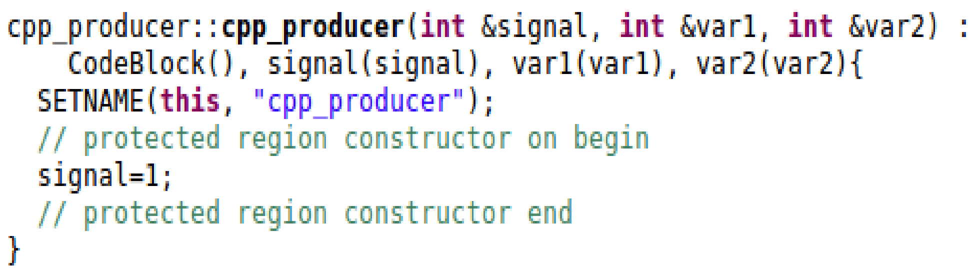


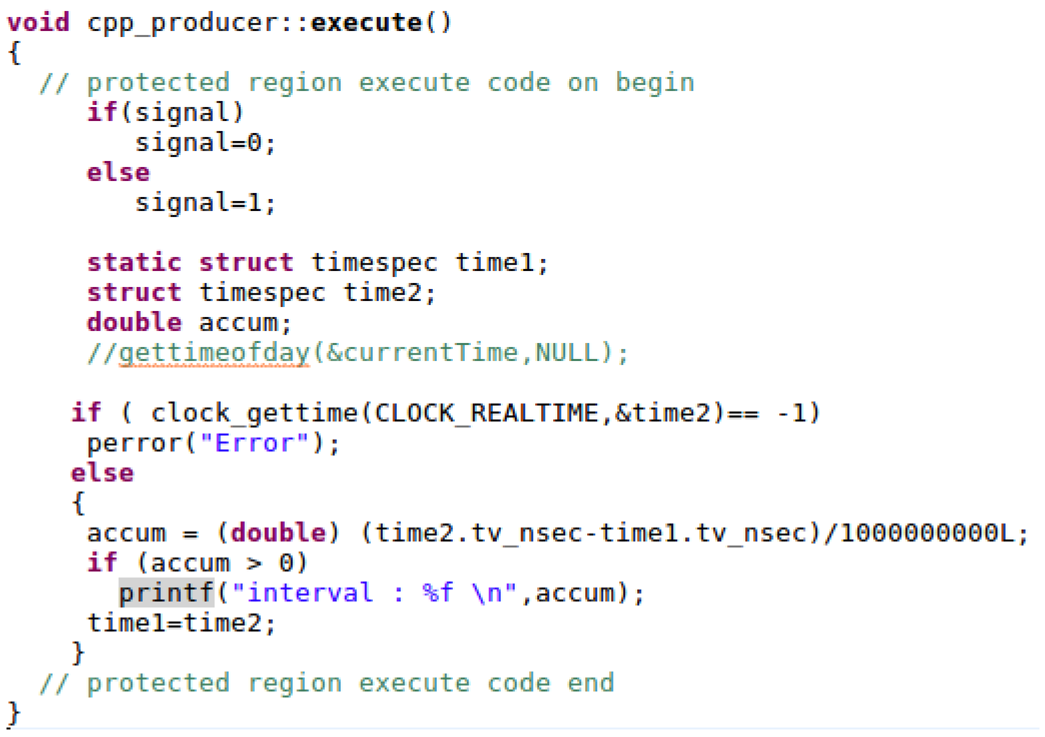
Figure 2.The cspm model of the timer with an extra port



```
cpp_producer::cpp_producer(int &signal, int &var1, int &var2) :
    CodeBlock(), signal(signal), var1(var1), var2(var2){
  SETNAME(this, "cpp_producer");
  // protected region constructor on begin
  signal=1;
  // protected region constructor end
}
```

Figure 3.Producer's code constructor

We start by having the signal high and we alter it at every execution.



```
void cpp_producer::execute()
{
  // protected region execute code on begin
     if(signal)
        signal=0;
     else
        signal=1;

     static struct timespec time1;
     struct timespec time2;
     double accum;
     //gettimeofday(&currentTime,NULL);

    if ( clock_gettime(CLOCK_REALTIME,&time2)== -1)
     perror("Error");
    else
    {
     accum = (double) (time2.tv_nsec-time1.tv_nsec)/1000000000L;
     if (accum > 0)
       printf("interval : %f \n",accum);
     time1=time2;
    }
  // protected region execute code end
}
```

Figure 4.Producer's code

➢ 3.1.2

Running the program on the device and having the oscilloscope connected what we expect to see is a square pulse waveform with its half period being 0.01 sec.

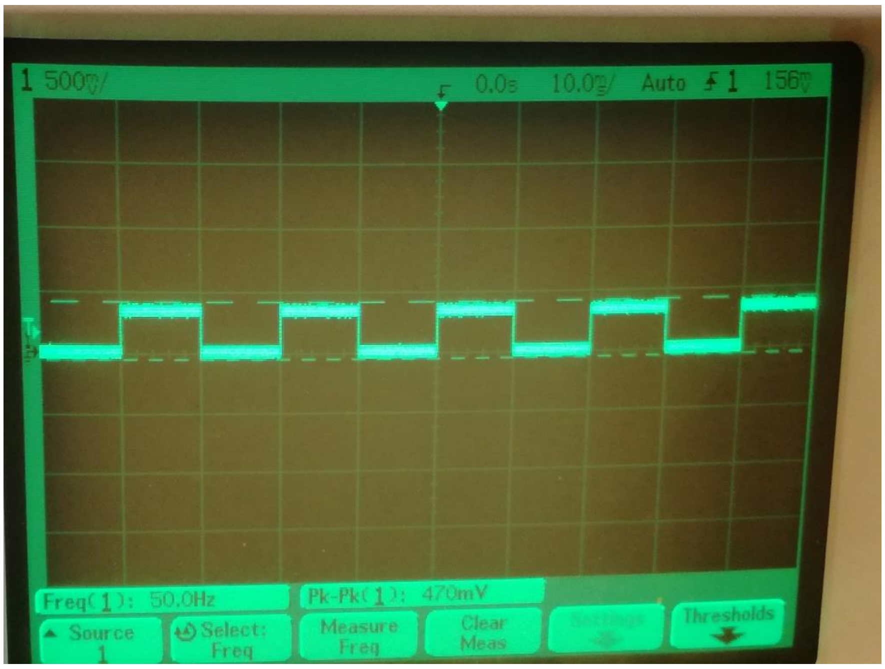


Figure 5.Periodic signal on oscilloscope

The horizontal distance between two grid boxes is 0.01 sec. Thus the result is the expected one.

➢ 3.1.3

Testing the timing behavior of our system in the VM (what we did on exercise 2) was an unorthodox way of trying to figure out precisely the system's performance. However, knowing the fact that those kind of measurements don't give precise results on VMs doesn't mean that they are useless. It is a fast and easy method of figuring out quickly if the general behavior of our system is the one that was supposed to be, before even testing it on a real time OS.
On the other hand, running the software on the Xenomai device which is real time we can have precise timing measurements that fully correspond to the real time system's performance. The only disadvantage of the hardware solution is probably the fact that we don't always have available FPGAs and real time devices for testing purposes. Also using an oscilloscope for figuring out precisely the jitter is difficult especially when it is small.

➢ 3.1.4

In exercise 2 we hade some significant variations on timing which were unacceptable for a real time system. On the real time system those variation have been minimized greatly.

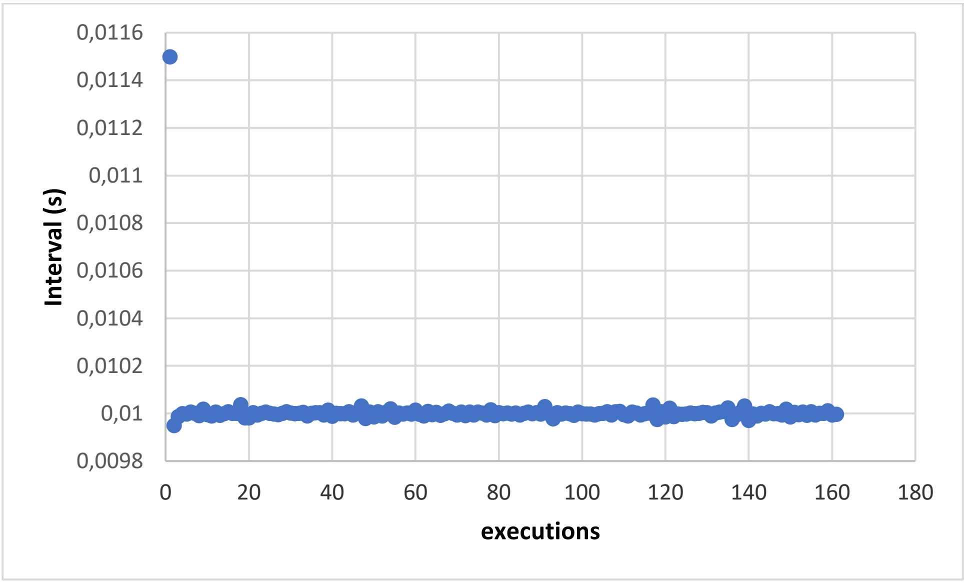


Figure 6.Timing measurements on xenomai

The average interval is at 0.010024 sec and the standard deviation is at 0.000118 sec . The delay here is significantly smaller than in the pervious measurements.

For understanding even better the timing differences between the real time system and the VM we combined both data sets in one chart.

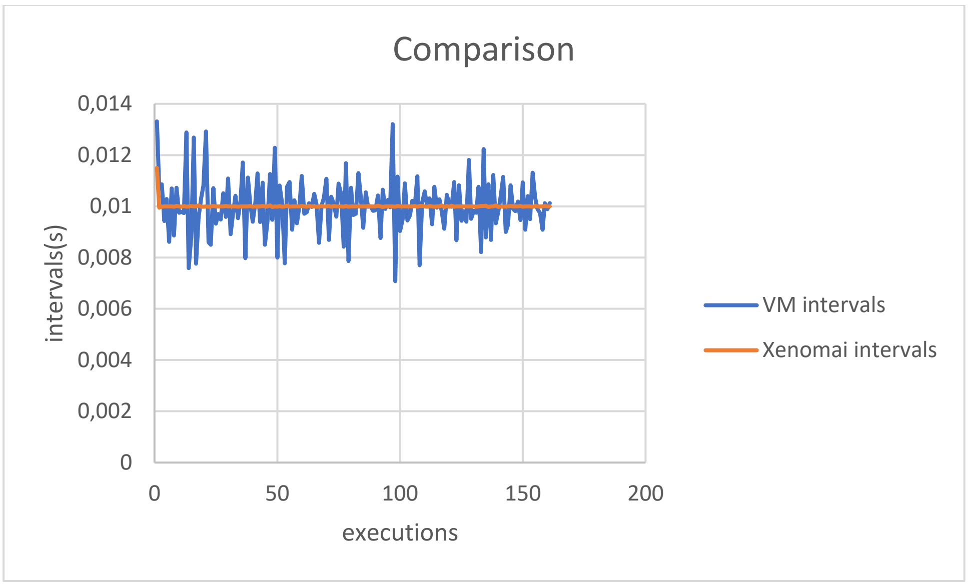


Figure 7.Timing comparison

From Fig.7 we can see clearly the quality difference in the two measurements.

➢ 3.1.5

Yes, the above timing differences are exactly what we expected because at first we used a VM and after that a real time OS. Contrary to other OS RTOS are extremely consistent about the time it takes for each task to be executed and they have fixed time constraints. This is because they are designed to run time critical applications which aren't supposed to have any timing variations.

<u>PART 2</u>

➢ 3.2.1

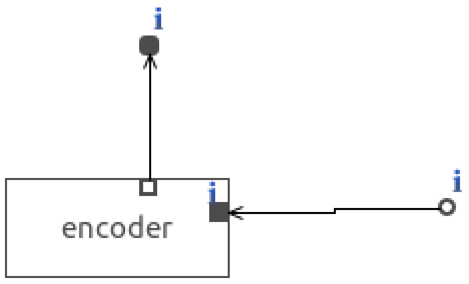


Figure 8.Encoder architecture model

The encoder takes input from the RaMstix Encoder port that is connected to the wheel . The other port is a periodic timer set at 0.01 sec.

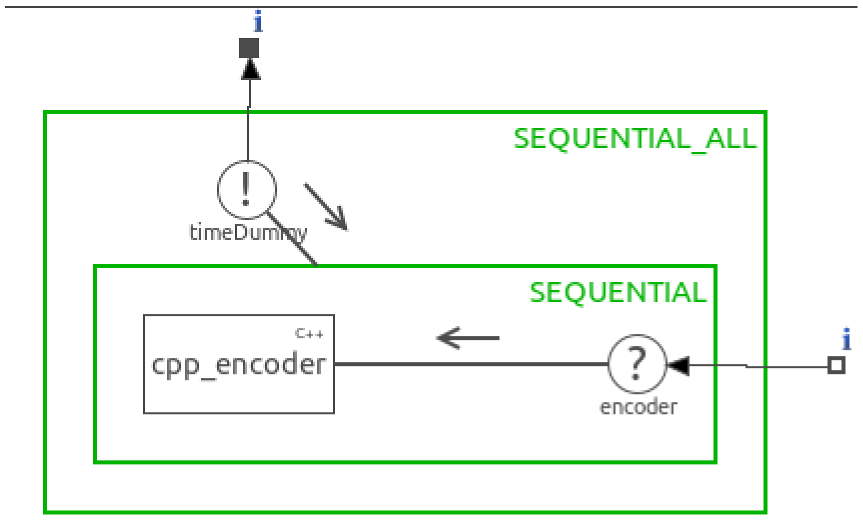
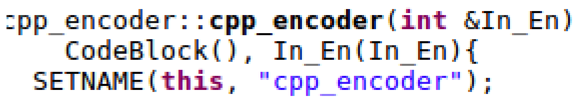
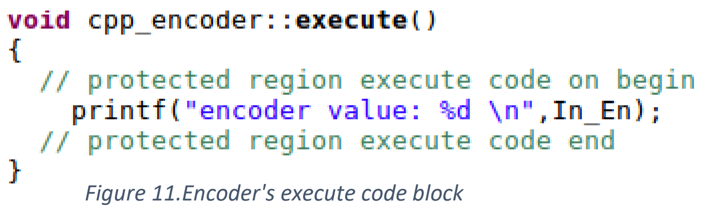


Figure 9.cspm model of the encoder



```
cpp_encoder::cpp_encoder(int &In_En)
    CodeBlock(), In_En(In_En){
  SETNAME(this, "cpp_encoder");
```

Figure 10.Encoder's input from RaMstix



```
void cpp_encoder::execute()
{
  // protected region execute code on begin
    printf("encoder value: %d \n",In_En);
  // protected region execute code end
}
```

Figure 11.Encoder's execute code block

The encoder takes an integer number showing the wheel’s position as an input and prints its value on the terminal.

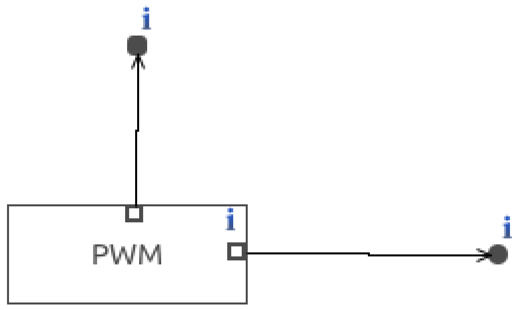


Figure 12.PWM architecture model

PWM sends a double number output signal with a value between -1 and 1 to a PWM RaMstix port which is connected to the oscilloscope.

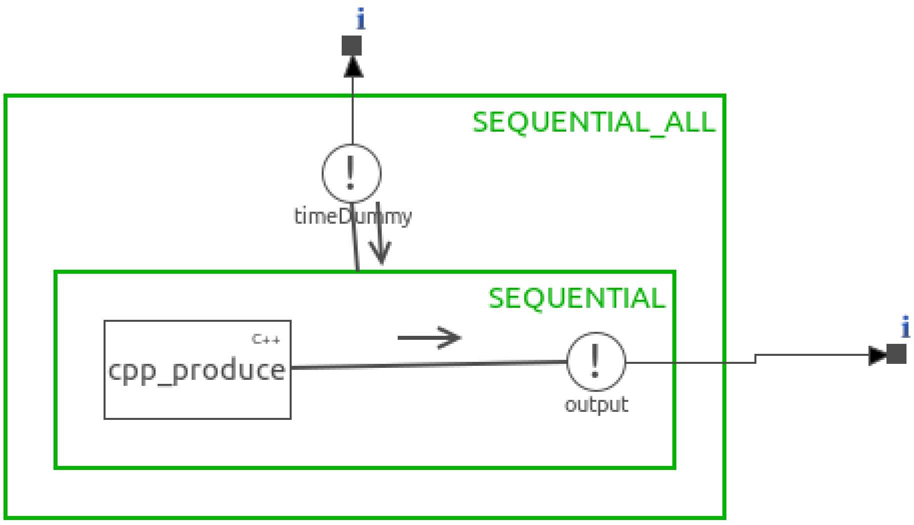


Figure 13.PWM model

The timer that synchronizes its execution is set at 0.01 as well.

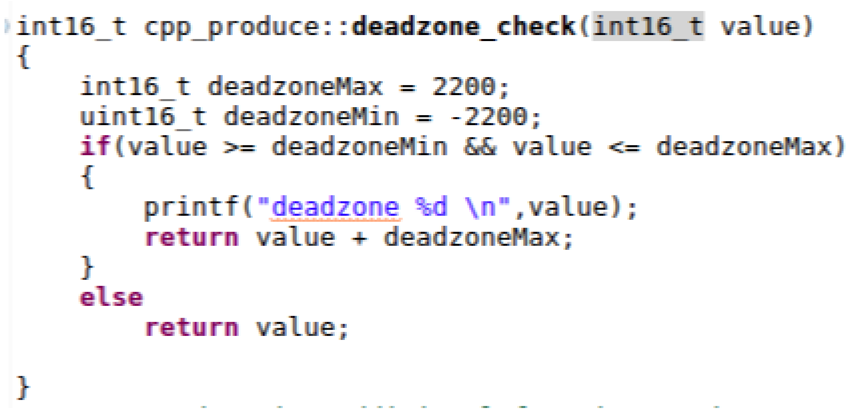
Inside the code block we initialize a test value at -32678 and we keep increasing it up to 32768 changing the duty cycle in that way. Every time that this process is executed we call a *deadzone_check* function that checks if the test value is between -2200 and 2200. If it belongs in that zone then it returns a different valid value. Finally, we call a *translation* function that normalizes that value. In some cases we are using int16_t instead of int to ensure that we are having 16-bit numbers as mentioned in the description. However, probably this is not necessary as it works fine in any way.

```
void cpp_produce::execute()
{
  // protected region execute code on begin
    static int value=-32768;
    this->outSignal = translation(deadzone_check(value));
    value++;
    if(value==32768)
        value=-32768;
  // protected region execute code end
}

// protected region additional functions on begin
double cpp_produce::translation(int16_t value)
{
    int16_t maxValue=32768;

    double normOut= (double) value / maxValue;
    return nornOut;
}
```

Figure 14.Code that changes the duty cycle gradually and the translation function



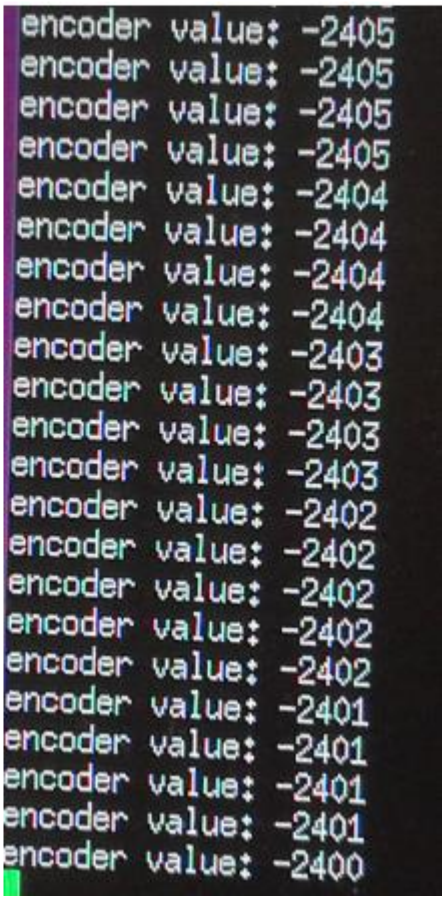
```
int16_t cpp_produce::deadzone_check(int16_t value)
{
    int16_t deadzoneMax = 2200;
    uint16_t deadzoneMin = -2200;
    if(value >= deadzoneMin && value <= deadzoneMax)
    {
        printf("deadzone %d \n",value);
        return value + deadzoneMax;
    }
    else
        return value;

}
```

Figure 15.The deadzone_check function

➢ 3.2.2

On the encoder we connected the chosen pins on the board and then we moved the wheel waiting for seeing position encoding values on the terminal.



```
encoder value: -2405
encoder value: -2405
encoder value: -2405
encoder value: -2405
encoder value: -2404
encoder value: -2404
encoder value: -2404
encoder value: -2404
encoder value: -2403
encoder value: -2403
encoder value: -2403
encoder value: -2403
encoder value: -2402
encoder value: -2402
encoder value: -2402
encoder value: -2402
encoder value: -2402
encoder value: -2401
encoder value: -2401
encoder value: -2401
encoder value: -2401
encoder value: -2400
```

Figure 16.Encoder values on terminal by pushing the wheel in one direction

For the PWM , we connected the output RaMstix port to the oscilloscope , expecting to see the duty cycle gradually changing as the value variable changes over time.

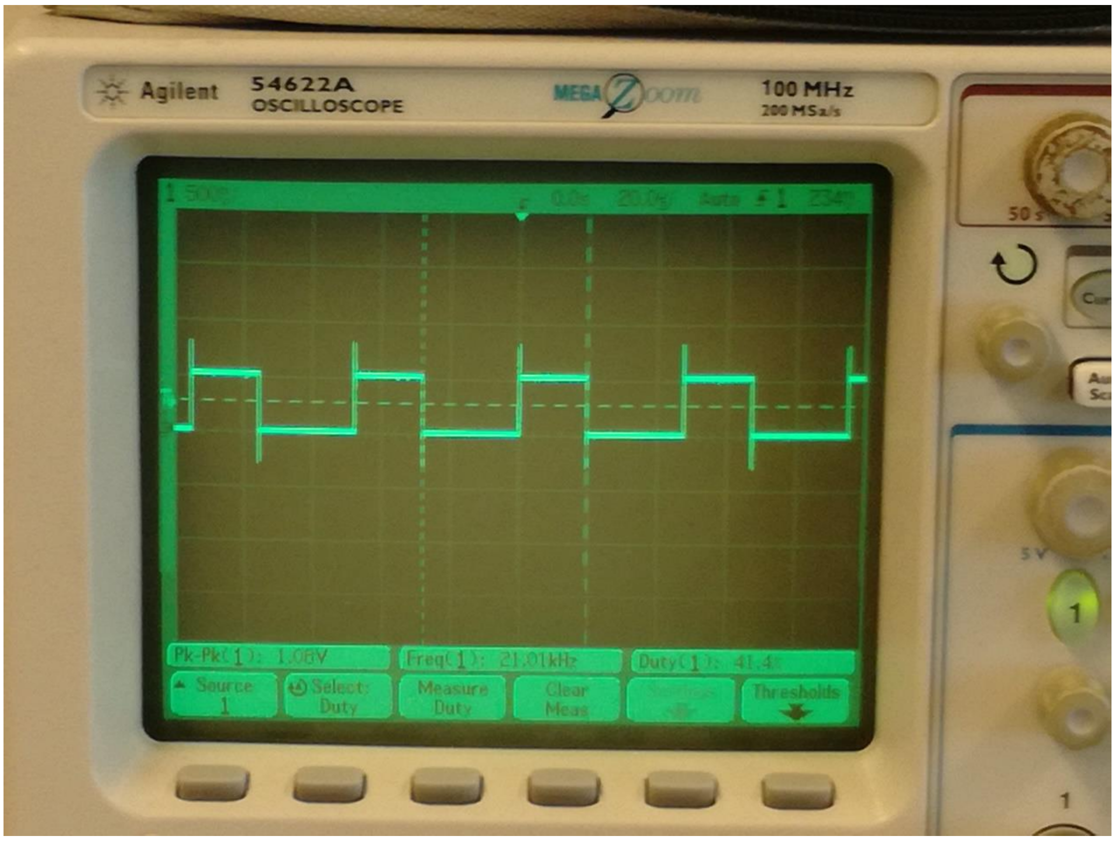


Figure 17.showing the duty cycle on the oscilloscope

At the above photo we can see that the duty cycle is about at 41%.

➢ 3.2.3

Testing the link drivers in an earlier stage and after that using them in the real system (robot, joystick) is part of the stepwise design method that we've followed so far. It can be considered as a unit test for both the encoder and the PWM device that we make to validate their correctness and check them under different inputs. In that way we can ensure that those systems operate as expected and that they will not fail once we integrate them in the actual control system. Furthermore, as we don't always have the actual setup available it is a good method to validate as many subsystems as possible beforehand in order to achieve a first-time-right result.

➢ 3.2.4

Yes, it is effective as discussed in 3.2.3. Apart from helping us to spot problems with the design of those units , it did help us to get familiar with the Xenomai device and with the board before using them in the actual setup.

PART 3

➢ 3.3.1

For controlling the robot with the joystick we created first an architecture model (similar to exercise 2) containing all the subsystems.
A timer with a period of 0.01 sec was added to the controller in order to have a recursive system.

- The joystick uses the provided API and generates two inputs on for the horizontal and one for the vertical motion. Those two inputs were initially normalized to belong to [-1,1].
- EncoderH and EncoderV are both connected to RaMstix encoder ports and what they do is to convert the input to a 32-bit number and passing it to the controller.
- PWM receives two inputs that correspond to the horizontal and vertical movement. Those values are already normalized in the joystick so this part does nothing but to transfer the input signals to the outputs. It sends the control signals to the robot through two RaMstix PWM ports.
- JIWY uses 20-sim extracted models that receive inputs from the joystick and the encoders and pass them to the PWM.

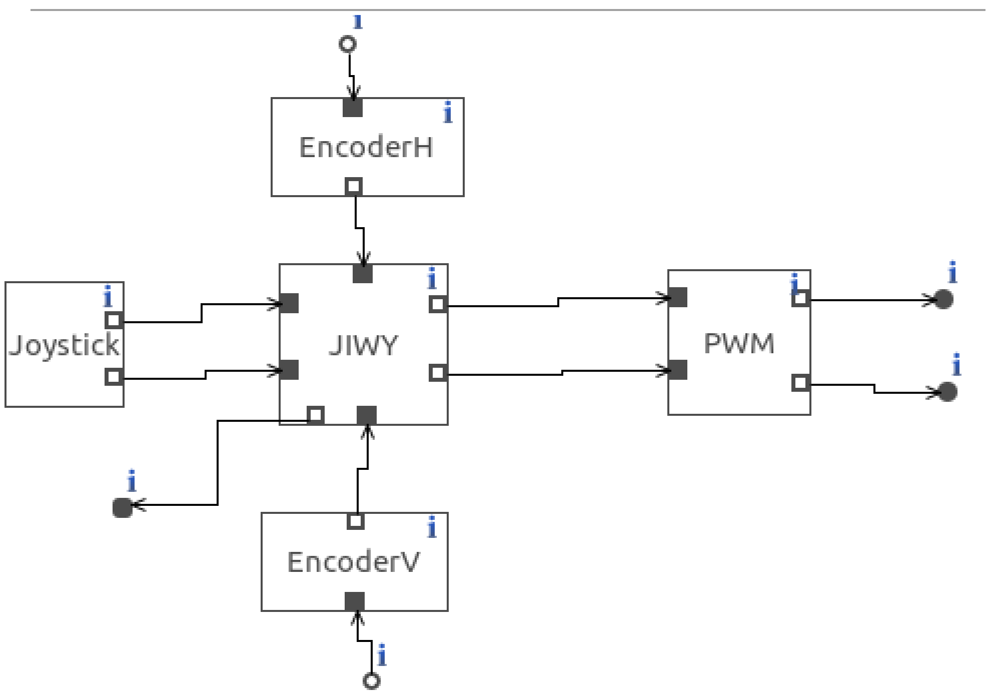


Figure 18.Architecture model

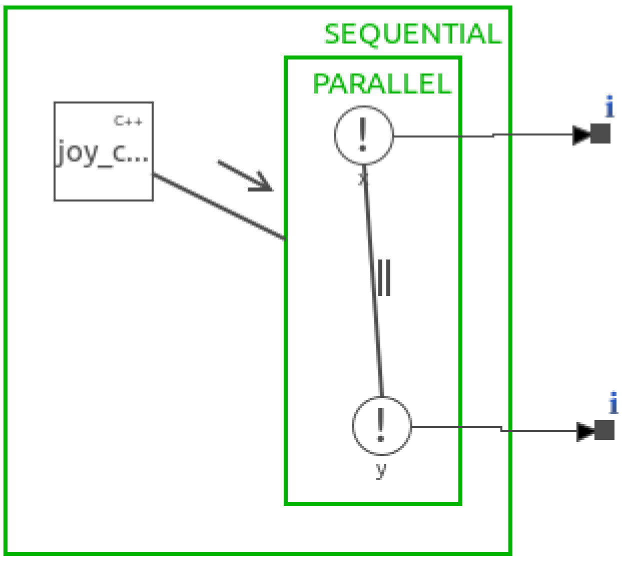


Figure 19.Joystick model

The top writer sends the input to the pan controller and the bottom to the tilt controller . Both are inputs generated from the joystick's movement.

```
void joy_code::execute()
{
  // protected region execute code on begin
    read(fd, &event, sizeof(event));
      if(event.type == JS_EVENT_AXIS)
      {
          if(event.number == 0)
          {
              x = (double)event.value/16000; //TODO

              //printf("Axis X: %f,\t Axis Y: %f\n", var_hor_joy, var_ver_joy);
          }
          else if(event.number == 1)
          {
              y = (double)event.value/16000; //TODO

              //printf("Axis X: %f,\t Axis Y: %f\n", var_hor_joy, var_ver_joy);
          }

      }
      else if(event.type == JS_EVENT_BUTTON)
      {
          if(event.number==3)
              printf("Smile\n");
      }
  // protected region execute code end
}
```

```
joy_code::joy_code(double &x, double &y) :
    CodeBlock(), x(x), y(y){
  SETNAME(this, "joy_code");

  // protected region constructor on begin
  fd = open("/dev/input/js0", O_NONBLOCK);
  // protected region constructor end
}
```

At first the x and y inputs were normalized using 32768 but after that in order to increase the joystick's sensitivity the robot's ability to turn more degrees we normalized it using 16000.

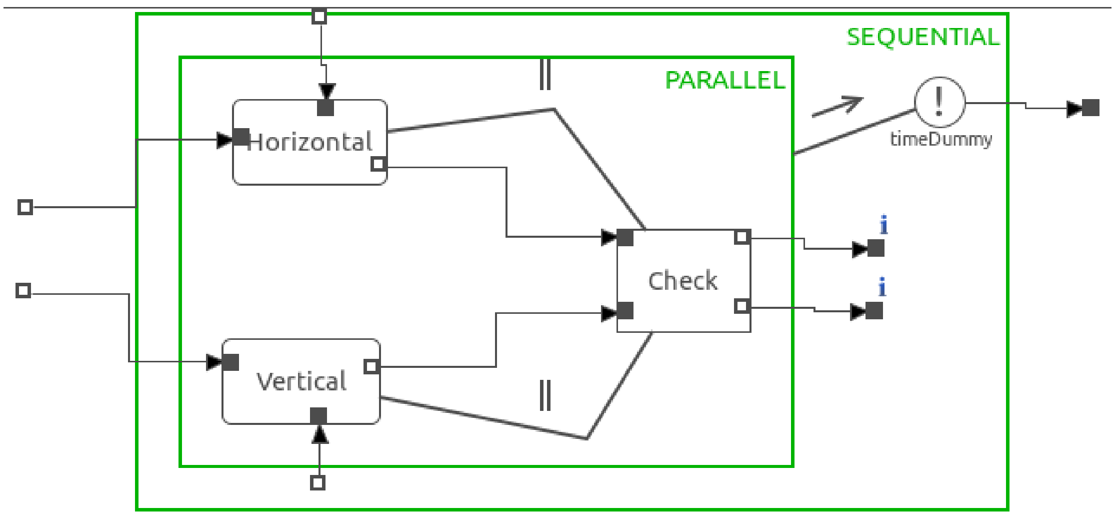


Figure 20.JIWY model

The JIWY controller is composed by a tilt controller (Vertical) a pan controller (Horizontal) and a check sub-system that is supposed to verify that the controller's outputs are correct before sending them to the PWM device and to the plant.

The models for the Horizontal and Vertical sub-systems were extracted from 20-sim.

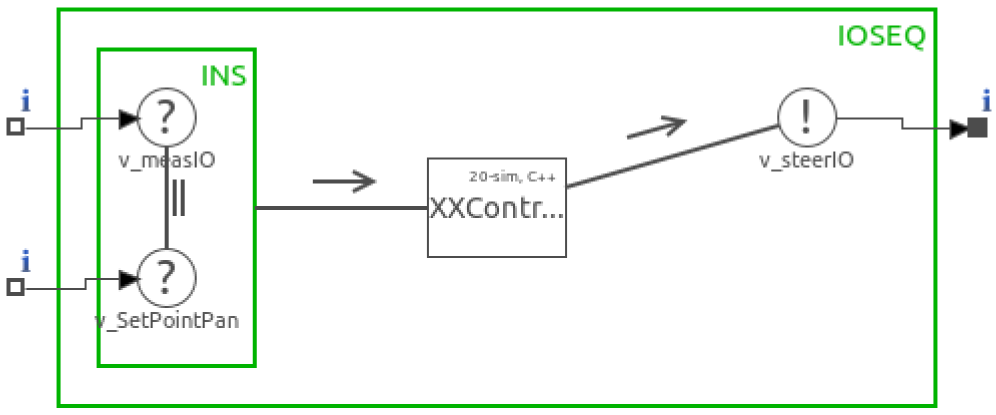


Figure 21.Horizontal-Pan controller

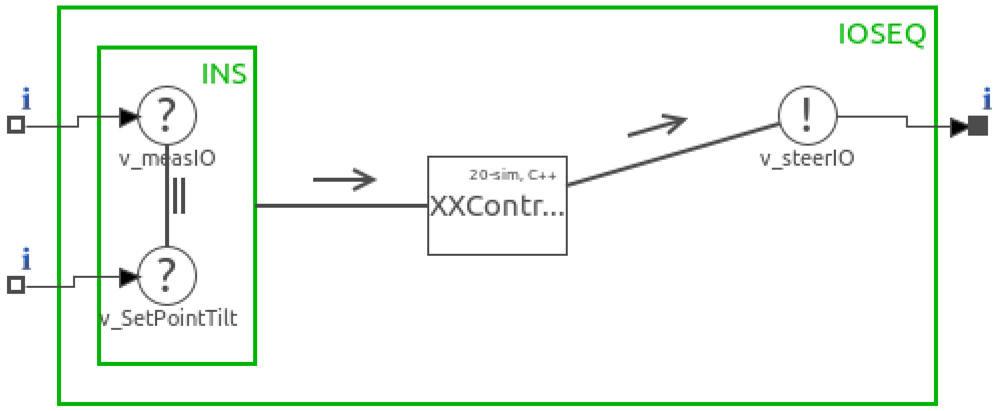


Figure 22.Vertical-Tilt controller

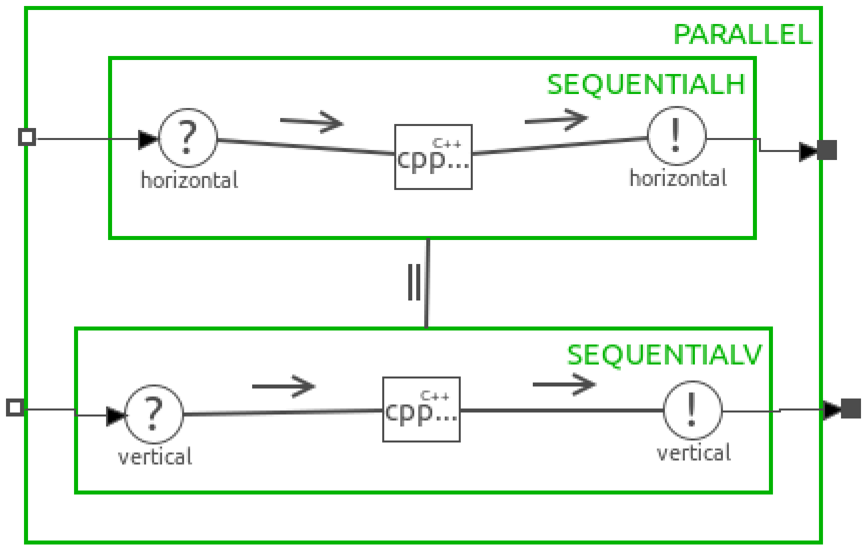


Figure 23.Check system

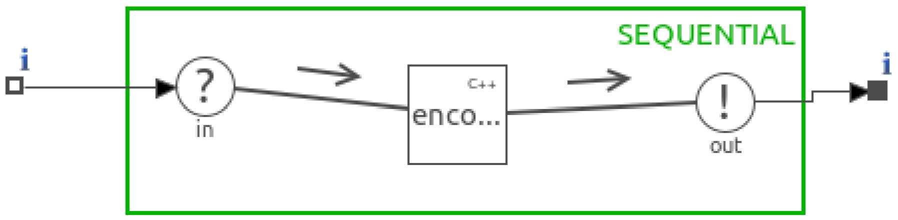
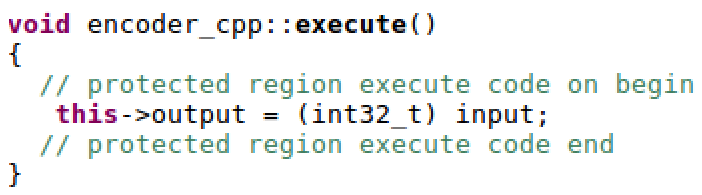


Figure 24.Encoder's model

Both encoders ensure that the inputs from the RaMstix ports are 32-bit numbers and send them to the Horizontal and Vertical controllers.



```
void encoder_cpp::execute()
{
  // protected region execute code on begin
   this->output = (int32_t) input;
  // protected region execute code end
}
```

Figure 25.Encoder's code

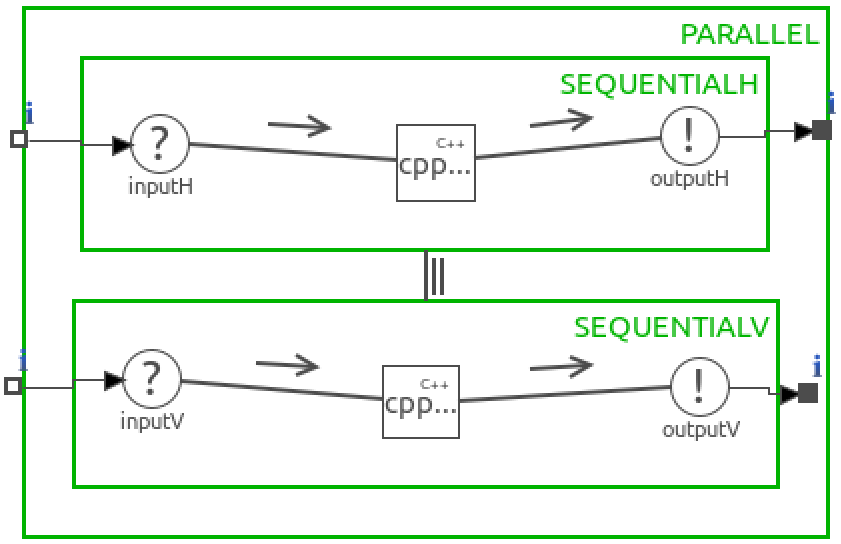


Figure 26.PWM model

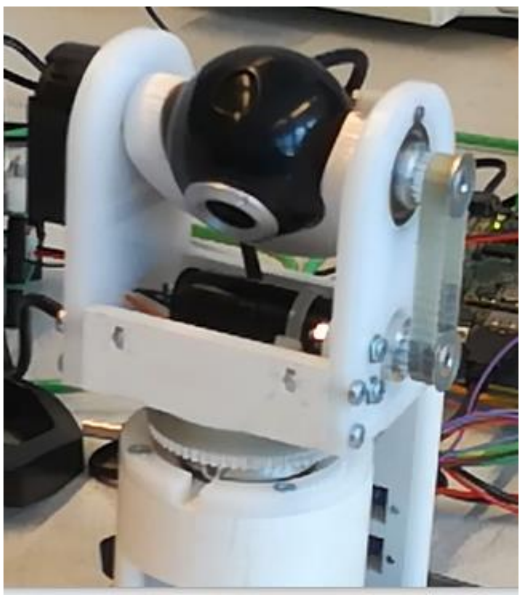


Figure 28.Moving the robot vertically

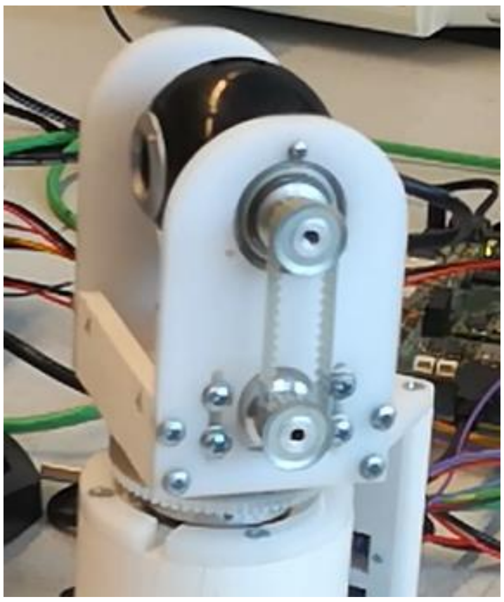


Figure 27.Moving the robot horizontally

➢ 3.3.2

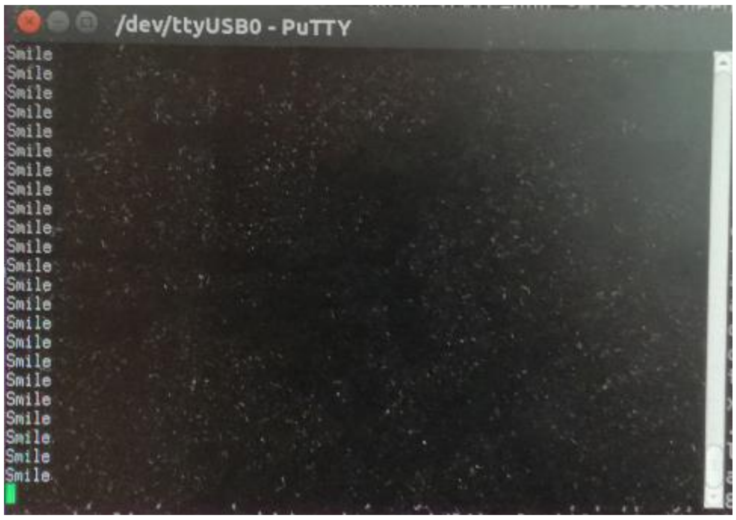
Extra scaling code segments were placed in the joystick in order to normalize the values that are taken from it . In the other systems (PWM, Encoders) what we mainly do is to pass the inputs to output without changing them.
Placing that scaling code inside the joystick is the easiest approach that we could follow because we scale the input the moment that we take it and then we pass it normalized to the other systems. A different approach would be to put the scaling code inside the PWM part and inside the controllers but that would make the whole design more complicated.

➢ 3.3.3

We could use the encoders to find out the maximum angles that the robot's joints can rotate. Then using those values as the maximum valid outputs that the controllers may give we could prevent it from hitting end stops. This approach could be implemented inside the check sub-system of the JIWY controller. We would check if the controllers give outputs bigger than the maximum angles of rotation and if that was true either we would change the control signals or we would stop the execution.

➢ 3.3.4

We implemented the extra functionality of pressing button 3 on the joystick and printing a message on the terminal.



```
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
Smile
```

Figure 29.Message on terminal

It is impossible for a human to press the button only once and print only one message on the terminal.

- 3.3.5

We didn’t implement any method for preventing from hitting end- stops . However, the idea is described on 3.3.3

- 3.3.6

In general faults may happen because of problems in sensors. devices that connect environment with the system or by coding mistakes (for example null values , non-initialized arrays etc.).In our case, encoders that don’t work properly or a joystick that doesn’t interpret its movement to the correct output values could produce failures on the system. To make the system fault tolerant we could make it redundant (probably having multiple encoders and having a “voting” protocol) or we could create segments in the code that will check the inputs from the environment before processing them. For example, when we take the x and y values from the joystick we should check if those values make sense before sending them to the controller, the same should also be done for the encoders.

- 3.3.7

We could do fault modeling. Creating a separate model and describing what faults and how often they can occur. Also, by fault injection we can test different real faulty cases and check how the system respond on them. For example, changing the values that the encoders send or even changing the inputs that the controller receives from the joystick would be ways to simulate faulty unexpected cases. On those fault-triggering tests we should try to run as many different scenarios as possible in order to be sure that the system will not fail.

- 3.3.8

The process that was followed is the concurrent design flow as explained in the slides, with the exercises focusing on the software side of the design. We received the control models from 20-sim and then we had to integrate them in the architecture that we designed using TERRA. Also, in exercise 2 we performed co-simulation between our TERRA model and the CT control system from 20-sim, all these are steps in the design flow. Finally in exercise 3.3 we reached the final step , testing our real time control system on Xenomai. The course doesn’t focus on the plant dynamics and on the physics of the control system. This makes sense, because control engineering and computer engineering are two different fields having different people who frequently have to collaborate for their projects.