

## Exercise 2

Rick Veens      Studentno: 0912292  
r.veens@student.tue.nl

Huib Donkers      Studentno: 0769015  
h.t.donkers@student.tue.nl

July 7, 2015

### 1 Timers

#### 1.1 Model

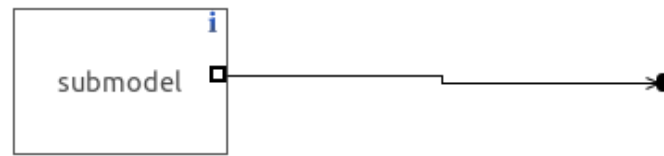
The model is shown in figure 1, and code used for the first two tests in codeblock 1. The improved code used for the third test is shown in codeblock 2. We used `octave` to parse the output as csv and perform some statistical analysis.

Codeblock 1: Pcode::execute

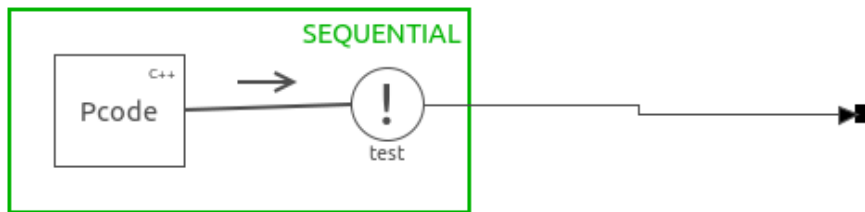
```
1 void Pcode::execute()
2 {
3     // protected region execute code on begin
4     static struct timespec t1;
5
6
7     struct timespec t2;
8     struct timespec res;
9
10    clock_gettime(CLOCK_REALTIME, &t2);
11    clock_getres(CLOCK_REALTIME, &res);
12    //long int elapsedTime = (t2.tv_nsec - t1.tv_nsec);
13
14    printf("%ld.%ld, %ld.%ld, %ld.%ld\n", res.tv_sec, res.tv_nsec, t2.tv_sec, ↵
15           t2.tv_nsec, t1.tv_sec, t1.tv_nsec);
16
17    t1 = t2;
18
19    // protected region execute code end
20 }
```

Codeblock 2: Pcode::execute (improved)

```
1 void Pcode::execute()
2 {
3     // protected region execute code on begin
4     static struct timespec t1;
5     static uint64_t clockcycle;
6
7     struct timespec t2;
8     struct timespec res;
```



(a) Main model.



(b) submodel

Figure 1: Model used for testing the timer

```

9  uint64_t currentcycle = ClockCycles();
10
11  clock_gettime(CLOCK_REALTIME, &t2);
12  clock_getres(CLOCK_REALTIME, &res);
13  int64_t dcycle= currentcycle-clockcycle;
14  uint64_t cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
15
16  if(currentcycle > clockcycle)
17      printf("%lld, %lld, %lld, %ld.%.9ld, %ld.%.9ld, %ld.%.9ld\n", currentcycle, ↵
            clockcycle, cps, res.tv_sec, res.tv_nsec, t2.tv_sec, t2.tv_nsec, t1.↵
            tv_sec, t1.tv_nsec);
18
19  t1 = t2;
20  clockcycle = currentcycle;
21
22  // protected region execute code end
23  }

```

## 1.2 Questions

1. Measured over 1522 intervals, the variation is  $3.667 \cdot 10^{-8} \text{ s}^2$  or  $0.03667 \text{ ms}^2$ . Since the clock resolution of QNX in virtual box ( $0.000999848 \text{ s}$ ) is so much larger than the observed variation, our measurements are not accurate enough to result in a variation that is representative for the actual variation. With a clock resolution of  $0.000999848 \text{ s}$ , and a timer that ticks every  $0.250000000 \text{ s}$  without jitter, we expect  $\lceil 0.25/0.000999848 \rceil - 0.25/0.000999848 = 96.2\%$  of the measurements to be  $\lceil 0.25/0.000999848 \rceil \cdot 0.000999848 = 0.249962 \text{ s}$ , and the other  $3.8\%$  to be  $\lceil 0.25/0.000999848 \rceil \cdot 0.000999848 = 0.250962 \text{ s}$ . In 1522 measurements we found  $96.2\%$  of the intervals to be  $0.249962 \text{ s}$  and  $3.8\%$   $0.250962$

s. Therefore, our measurements do not provide evidence of jitter.

Up to a jitter bounded by  $0.250000 - 0.249962 = 0.00038 \text{ s} = 380 \text{ } \mu\text{s}$ , we would still expect the same results. For larger jitter, we would expect to occasionally measure intervals of  $0.248962 \text{ s}$ . So we can strengthen our claim: our measurements do not provide evidence of jitter larger than  $380 \text{ } \mu\text{s}$ .

We performed a second series of measurements, this time with the timer interval set to  $0.249962$ , an exact multiple of the clock resolution. We measured 1967 intervals. With this configuration, we are likely to pick up on a variation larger than  $(0.000999848/1967 \cdot 10^6)^2 \approx 0.258 \text{ } \mu\text{s}^2$ .

Results of the second test is shown in figure 2. This shows that there is indeed jitter. The variation of our measurements is  $0.06 \text{ ms}^2$ .

For a third test we used a method of timing with a resolution that is much higher: `ClockCycles()` from `sys/neutrino.h`. We can retrieve the number of clock cycles per second, so we can deduce how much time has passed from the number of clock cycles that have passed. We measured 3339 intervals using this method. The distribution of intervals is shown in figure 3. We can clearly see how the jitter behaves. It is still distributed somewhat discretely, but smaller deviations are now clearly shown. In this test the variation of the measurements is  $0.078 \text{ ms}^2$ .

2. This jitter is the result of simulating a real time OS in an environment that is not real time. Since the host OS decided when the virtual OS can run, and the host OS cannot guarantee to meet real time requirements, the virtual OS is unable to meet those requirements either.
3. We observe from figure 2 that the timer tick is occasionally delayed by more than 1 ms. Such a delay is not acceptable in many real time applications. **EXAMPLE.**
4. No. Our virtual machines run on a non real time OS (GNU/Linux), so it can happen that the host OS is very busy with tasks that have a higher priority than the virtual machines, postponing execution of the real time OS unboundedly, disabling QNX to meet its real time requirements.

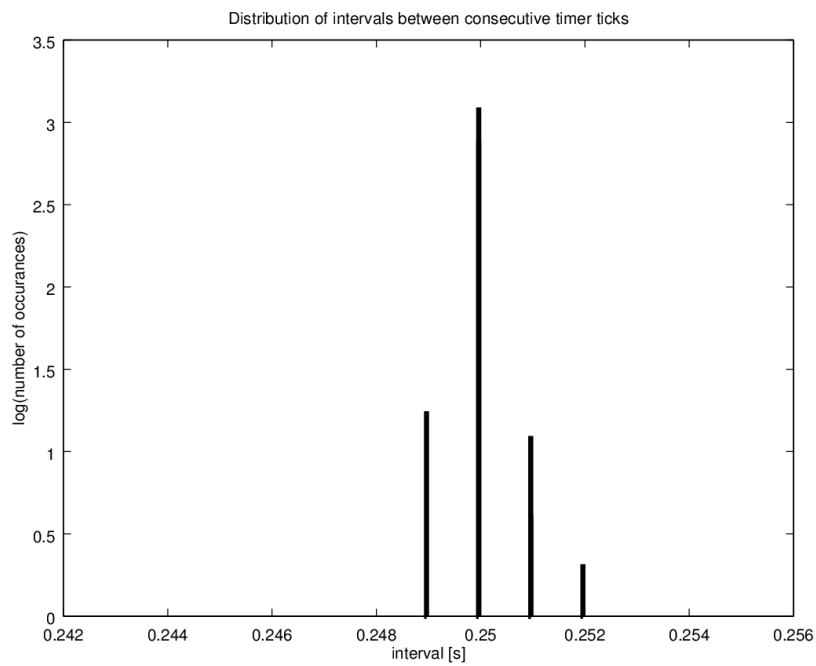


Figure 2: Distribution of intervals in the second test.

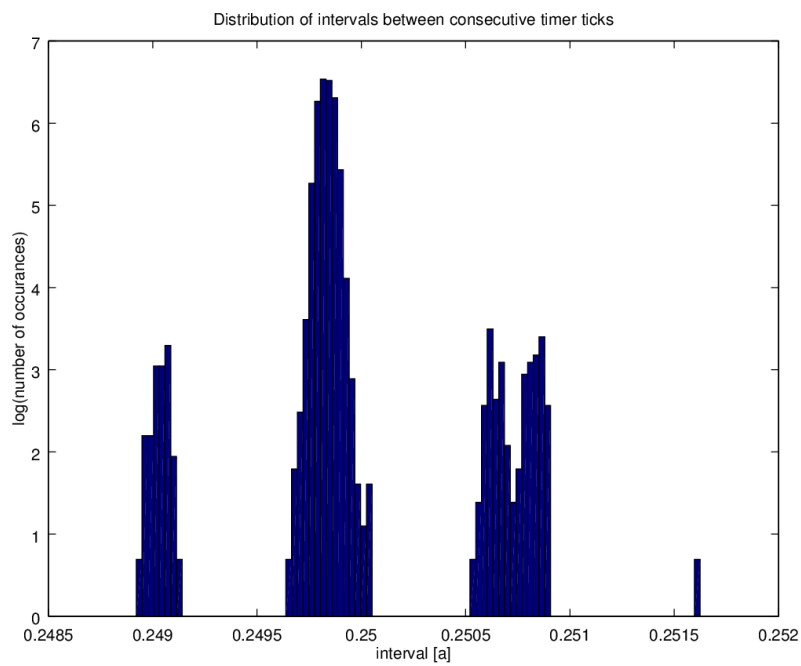


Figure 3: Distribution of intervals in the third test.

### 1.3 Validation

We further extended the code measuring the intervals to send an alternating signal to an output pin (see codeblock 3 AND FIGURE X). The signal switches from high to low, or from low to high at each timer tick, producing a square wave of which we can measure the period using an oscilloscope.

Codeblock 3: Pcode::execute (extended)

```

1 void Pcode::execute()
2 {
3     // protected region execute code on begin
4     static struct timespec t1;
5     static uint64_t clockcycle;
6
7     if (oc)
8         oc = 0;
9     else
10        oc = 1;
11
12    //printf(" oc: %d\n", oc);
13
14    struct timespec t2;
15    struct timespec res;
16    uint64_t currentcycle = ClockCycles();
17
18    clock_gettime(CLOCK_REALTIME, &t2);
19    clock_getres(CLOCK_REALTIME, &res);
20    int64_t dcycle= currentcycle-clockcycle;
21    uint64_t cps = SYSPAGE_ENTRY(qtime)->cycles_per_sec;
22
23    if(currentcycle > clockcycle)
24        printf("%lld, %lld, %lld, %ld.%.9ld, %ld.%.9ld, %ld.%.9ld\n", currentcycle, ↵
                clockcycle, cps, res.tv_sec, res.tv_nsec, t2.tv_sec, t2.tv_nsec, t1.↵
                tv_sec, t1.tv_nsec);
25
26    t1 = t2;
27    clockcycle = currentcycle;
28
29    // protected region execute code end
30 }

```

### 1.4 Questions

1. Using the oscilloscope has the obvious disadvantages: it requires the availability of the oscilloscope, and an output pin. This output pin is not (easily) available for our dry-runs in a virtual machine. An additional disadvantage is that the supplied oscilloscope did not appear to have a function to measure a series of intervals easily. Only one measurement was shown on screen, updating frequently, making it very hard to generate a list of 1000-3000 measurements for an accurate assessment of the jitter like we did with the software method.

The software method uses the same timing hardware to both produce the timer ticks, as to measure the intervals. This method only accurately measures the timer intervals, if the timing hardware is reliable.

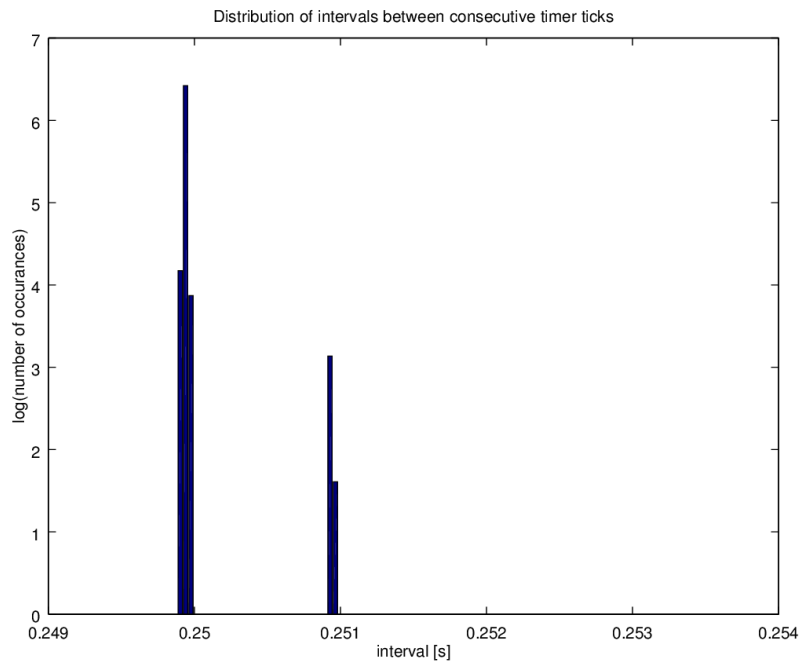


Figure 4: Distribution of intervals in the lab, with the timer set to 4 Hz.

2. We ran two tests in the lab, one with a frequency of 4 Hz, and one with a frequency of 100 Hz. The readings from the oscilloscope seemed to agree with the measurements using the number of clockcycles, showing a period of 500.0 ms, sometimes 501.0 or 499.0 in the first run, and a period of 20.0 ms during the second run. We measured 758 and 936 intervals for 4 Hz and 100Hz respectively, using the number of clock cycles. Distribution of these measurements are shown in figure 4 and figure 5. Variation of these measurements are  $0.053 \text{ ms}^2$  and  $0.0011 \text{ ms}^2$  respectively. We see from the distribution that the test with 4 Hz resulted in two narrow peaks. We don't have an explanation for this (TRY TO EXPLAIN, OR NOT), but clearly observe that the variation is a lot less than when using the virtual machine, more so than the actual variation of  $0.053 \text{ ms}^2$  would suggest.
3. WHAT THEORY?

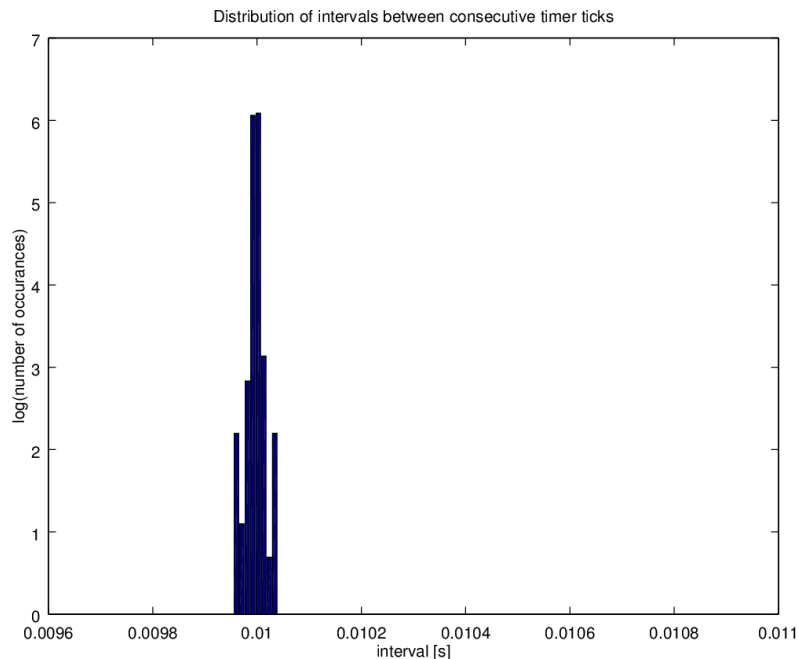


Figure 5: Distribution of intervals in the lab, with the timer set to 100 Hz.

## 2 JIWOYIO Linkdrivers

### 2.1 JIWOYIO driver components

We made the two models suggested in the exercise description.

#### 2.1.1 PWMTester

We tested writing some values to the simulated port first, see codeblock 4. After that we made a function to convert from `int16_t` to `uint16_t` by using bit operations. However, we later found that this convert function could be replaced by a simple cast from `int16_t` to `uint16_t`, see codeblock 5. Additionally, we made sure that any non-zero value would map to a value outside the deadzone of the motor.

At first we understood that the value JIWOY outputs for engine steering would be a real between  $-512$  and  $512$ . So we made a normalise function mapping this range to the range  $[-1, 1]$ .

Codeblock 4: `Pcode::execute`

```

1 void Pcode::execute()
2 {
3     // protected region execute code on begin

```

```

4   static double t = 1.0;
5
6   //t = -32768; // fast reverse
7   //t = 32767; // fast forward
8   //t = 32768; // stop
9   //t = -1000; // deadzone?
10
11  //this->test = (uint16_t) t;
12
13  this->test = convert(normalise(t));
14  printf("%lf, %lf, %d \n", t, normalise(t), convert(normalise(t)));
15  //t+= 6.28318;
16  while(t>512)
17      t-= 1024;
18
19  // protected region execute code end
20 }

```

Codeblock 5: Pcode::convert

```

1  uint16_t Pcode::convert(double f)
2  {
3      int16_t deadzone = 2200;
4
5      int16_t n = f*(32768-deadzone);
6      if(n>0)
7          n += deadzone;
8      else if(n<0)
9          n -= deadzone;
10
11     if(n>=0 && n<32768)
12         return n;
13     else if(n >= -32768 && n<0)
14     {
15         return (uint16_t)n;
16     }
17     else
18         return 0;
19 }

```

Codeblock 6: Pcode::normalise

```

1  double Pcode::normalise(double n)
2  {
3      // linear conversion assumed
4      double min=-512, max=512;
5
6      // normalise to [-1,1]
7      n-= min;
8      n/= max - min;
9      n*= 2;
10     n-= 1;
11
12     return n;
13
14     /*
15     // scale to int16_t
16     if(n=1)
17         return 32767;
18     else
19         return n*32768;
20     */

```



21 | }

### 2.1.2 EncoderTester

The EncoderTester model merely prints values read from the port. We found out that, instead of using bit-shift operations, converting from uint32\_t to int32\_t could be done by a simple cast.

Codeblock 7: Relevant EncoderTester functions

```
1 void Pcode::execute()  
2 {  
3     // protected region execute code on begin  
4     printf("%d, %f - ", this->test, convert(this->test));  
5     // protected region execute code end  
6 }  
7  
8 // protected region additional functions on begin  
9 double Pcode::convert(uint32_t i)  
10 {  
11     return (int32_t)i;  
12 }
```

## 2.2 Questions

1. The advantage of testing the link driver with a software simulation allows us to run the program without requiring hardware. If in a project the hardware and software were to be developed separately, this allows the software to be developed in parallel to the hardware. So, in this project, it allows us to make sure our program (reading/writing from a port) works correctly without being in the lab.
2. One could attempt to make use of C++ sub-classing with a clear defined interface. This would mean one subclass for simulation and one for the actual hardware. This seems to be already implemented. The manual code changes could be solved by clever usage of regular expressions. The user interface is not really an issue here.
3. The most obvious drawback is of course that you need to do the exact same edits every time you generate code from the models. Another drawback is that you have to adapt your code to run simulations for testing. It is not possible to test the actual code that will be run in the real environment.

## 2.3 JIWOYIO driver components in the lab

We used the oscilloscope to capture and show the PWM signal. Using negative values, the direction signal was high, and for positive values the direction signal was low. Using value 1 (in range  $[-512, 512]$ ), the PWM signal was high for  $4.2 \mu\text{s}$ , with a period of  $61.4$

$\mu\text{s}$ , corresponding to an 16 bit integer value of  $32768 \cdot 4.2/61.4 = 2241$ , so just outside the deadzone, as expected. Using a value of  $-1$ , the PWM signal was high  $57.2 \mu\text{s}$ , with the same period of  $61.4 \mu\text{s}$ . We expected the same PWM signal as with value 1, apart from the direction bit. After verification with the Teaching Assistant, we understood that this is normal behaviour, the signal we found corresponds to a motor moving backwards slowly. We tested a couple of other values  $-512$ ,  $-128$ ,  $0$ ,  $256$  and  $512$ , and these all showed the results we expected.

We wired up the fly wheel and started the EncoderTester. In the console, the values corresponded with 2000 pulses per revolution of the wheel. Counting up when turning clockwise, counting down when turning anti-clockwise. We did not test for overflow of the counter, because of time restrictions (over a million revolutions were needed to reach overflow).

## 2.4 Questions

1. The differences between the dry run and the testing in the lab were of course that in the lab all input and output was tied to some actual physical action that we could not observe during the dry run. We could not observe the PWM signal nor how the position of a wheel related to the value of the encoder.
2. The 'first a dry-run and then testing in the lab'-approach is effective, because, it allows the software to be developed in parallel to the hardware. This requires the interface to the hardware to be clearly defined.

One should take note that the dry-run software simulation is not a substitute for testing the code on real hardware. The code might work in the simulation, but not on the real hardware.

## 3 Controlling JIWIY with QNX & CSP

Before the lab, we prepared a TERRA project that contains Joystick, JIWIY, IO and encoder conversion submodels. See figure 6 for a screenshot of the top architecture model.

### 3.0.1 Joystick input handling.

This model (figure 7) contains C++ code blocks that implement the Joystick functionality as suggested in Appendix B of the assignment document.

See codeblock 8 for relevant code of the `x_code` codeblock. `y_code` is almost identical.

Codeblock 8: `x_code` Joystick C++ code block

```
1 /**
2  * Source file for the x_code model
3  * Generated by the TERRA CSPm2LUNA generator version 1.1.3
```

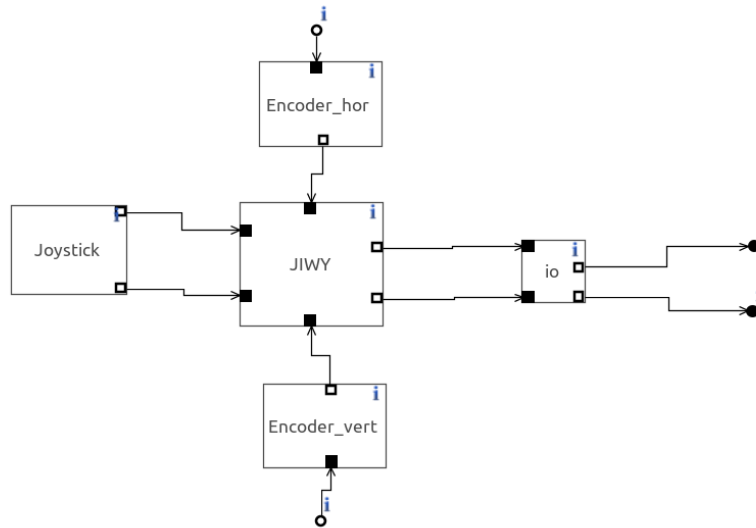


Figure 6: Top architecture model of the Ex2 JIWI model.

```

4  *
5  * protected region document description on begin
6  *
7  * protected region document description end
8  */
9
10 #include "x_code.h"
11 // protected region additional headers on begin
12 // Each additional header should get a corresponding dependency in the Makefile
13 #include "hid/HID.h"
14 #include "csp/Reader.h"
15 #include "string.h"
16 // protected region additional headers end
17
18 namespace Joystick { namespace x_code {
19
20 x_code::x_code(double &x) :
21     CodeBlock(), x(x){
22     SETNAME(this, "x_code");
23
24     // protected region constructor on begin
25     channel = new LUNA::CSP::HIDAbsAxisChannel("/dev/joystick2", 1);
26     reader = new LUNA::CSP::Reader<double>(&x, channel);
27
28     buttonchannel = new HIDButtonChannel("/dev/joystick2", 1);
29     buttonreader = new LUNA::CSP::Reader<bool>(&btn, buttonchannel);
30
31     // protected region constructor end
32 }
33
34 x_code::~x_code()
35 {
36     // protected region destructor on begin
37     delete channel;
38     delete reader;
39     delete buttonreader;
40     // protected region destructor end
41 }
42
43 void x_code::execute()

```

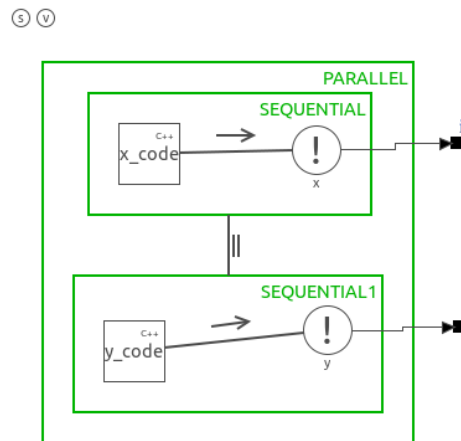


Figure 7: The JIWI Joystick submodel

```

44 {
45     // protected region execute code on begin
46     channel->read(reader);
47     buttonchannel->read(buttonreader);
48     // scale
49     this->x = (x - 512.0) / -512.0;
50
51     //printf(" Joystick_x: i read %f\n", x);
52
53     if (btn)
54     {
55         printf(" Smile !!!\n");
56         std::terminate();
57     }
58     // protected region execute code end
59 }
60
61 // protected region additional functions on begin
62 // protected region additional functions end
63 // Close namespace(s)
64 } }

```

### 3.0.2 JIWI model.

The overview model's (figure 6) JIWI model (figure 8) is very similar to the model we looked at in exercise 1. The only difference is that It has imported models from the given 20sim JIWI model, 'Horizontal' and 'Vertical'.

See figure 11 for the submodels of Horizontal, Vertical and Check.

### 3.0.3 IO output handling.

The IO model of our top-level model (figure 6) converts input from JIWI (double) to HW port (uint16\_t). See figure 9. Codeblock 9 presents the relevant code that is used to do the

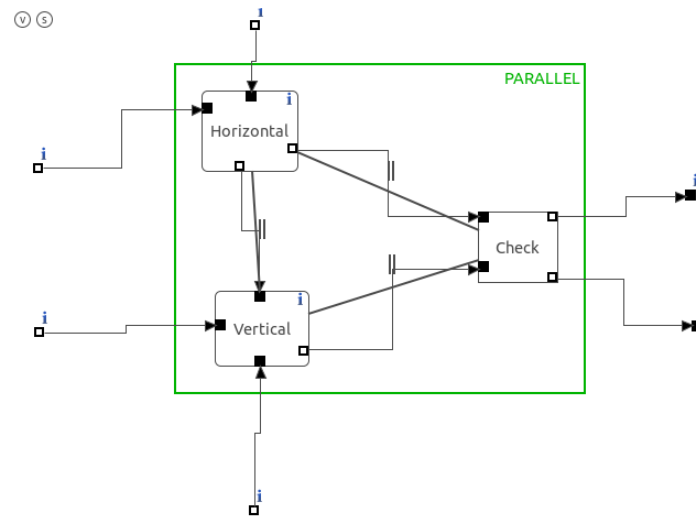


Figure 8: JIWI

conversion. Note that the code of `hor_convert` and `ver_convert` is identical.

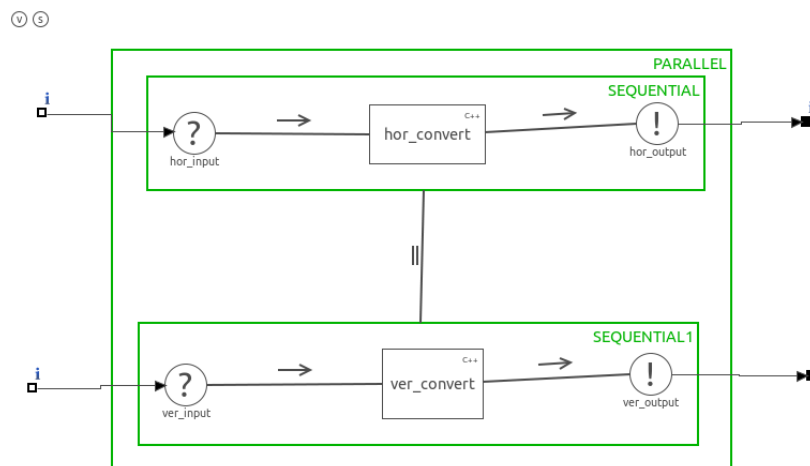


Figure 9: IO

Codeblock 9: Relevant code of the IO block

```

1 void ver_convert::execute()
2 {
3     // protected region execute code on begin
4     this->ver_output = convert(this->ver_input);
5     printf("Vertical: in: %f, out: %d\n", ver_input, ver_output);
6     // protected region execute code end
7 }
8
9 // protected region additional functions on begin
10 uint16_t ver_convert::convert(double f)
11 {

```

```

12  int16_t deadzone = 1000;
13
14  int16_t n = f*(32768-deadzone);
15  if(n>0)
16      n += deadzone;
17  else if(n<0)
18      n -= deadzone;
19
20  return (uint16_t)n;
21 }

```

### 3.0.4 Vertical and Horizontal encoding handling.

Encoder\_hor and Encoder\_vert of the top-level model (figure 6) are very similar. They both consist of a C++ codeblock that does some conversion. Specifically, converting from uint32\_t to double.

See figure 10 for a screenshot of the encoder model, and codeblock 10 for the code (same code in Encoder\_Hor and Encoder\_vert).

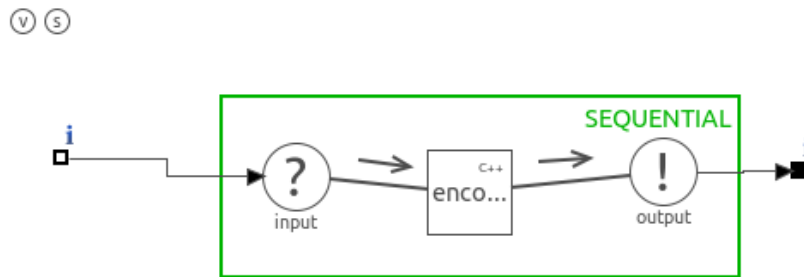


Figure 10: Encoder

Codeblock 10: Relevant code of the Encoder\_vert and Encoder\_hor models

```

1  void encodervert::execute()
2  {
3      // protected region execute code on begin
4      this->output = scale(convert(this->input));
5      printf("%f\n", this->output);
6      // protected region execute code end
7  }
8
9  // protected region additional functions on begin
10 double encodervert::convert(uint32_t i)
11 {
12     return (int32_t)i;
13 }
14
15 double encodervert::scale(double n)
16 {
17     return (n/6000)*2*M_PI;
18 }

```

### 3.1 Test at the lab

During the lab test we had a few problems. However, we got it working in the end.

Here is a list of things that we had to adjust in the lab:

- Joystick input was not working: when creating the `HIDAbsAxisChannel` we entered an arbitrary string instead of the `‘/dev/joystick2’` device file.
- The output of our Joystick component was not normalized to  $-1 .. 1$ .  
We understood from the 20sim model that the JIWIY controller model input was a value between  $-512$  and  $512$ , this is wrong.
- The IO sub-module’s C++ codeblocks had their variables erroneously linked with the wrong parent variables.
- The Joystick input was reversed, so we multiplied it by  $-1$ .
- The conversion code of the IO block was initially in the check part of the JIWIY model. We changed this to a separate model because it makes more sense. The conversion to `uint16_t` has nothing to do with the JIWIY model.
- The JIWIY model expected radian values for the angles of the motors. We had to do some scaling to achieve this.
- The JIWIY model output for PWM ranged from  $-1$  to  $1$ , not from  $-512$  to  $512$  as we initially thought. This was easily fixed.
- JIWIY had trouble keeping the camera still when we mapped every non-zero value to a PWM value outside the deadzone. So we reintroduced a deadzone, slightly smaller than the original one, by assuming a deadzone between  $-1000$  and  $1000$ , instead of  $-2200$  and  $2200$ . This maps small values to within the actual deadzone of the motor.

### 3.2 Questions

1. We put scaling code in two models called `Encoder_hor` and `Encoder_vert`. See figure 6 and codeblock 10 for the code.

We put scaling code in the Joystick submodel, see codeblock 8 line 49.

We put scaling code in the IO submodel, see codeblock 9

We put the scaling code models outside of the JIWIY model because we see it as an extra layer to the JIWIY model, instead of a part of the JIWIY model itself.

2. This is best implemented inside the check JIWIY submodel, shown in figure 11c, as a layer between the JIWIY calculation, and the actual IO conversion.

### 3.3 Further functionality

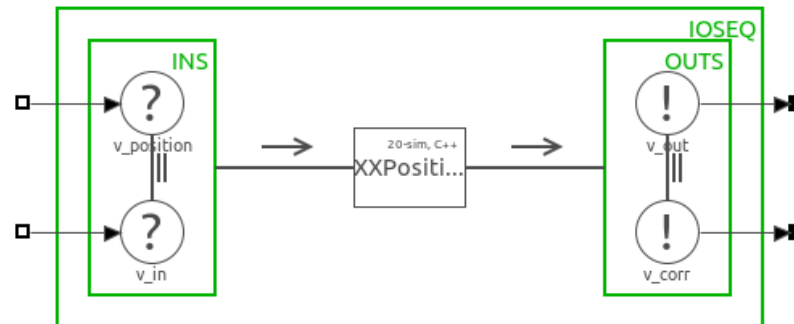
We implemented joystick button functionality and a way to exit the program without CTRL-C.

### 3.4 Questions

1. Not applicable
2. **TODO**
3. **TODO**
4. **TODO**

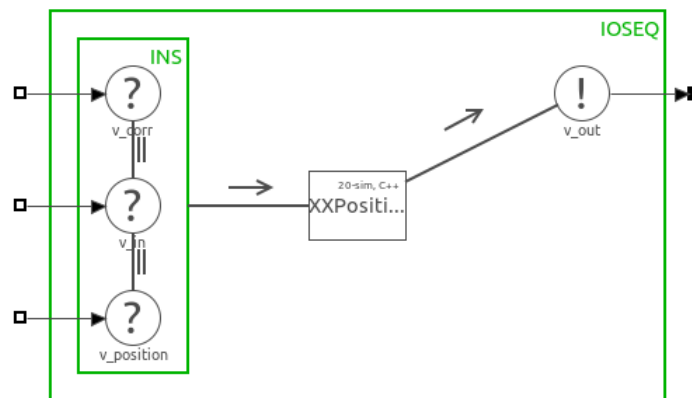


(S) (V)



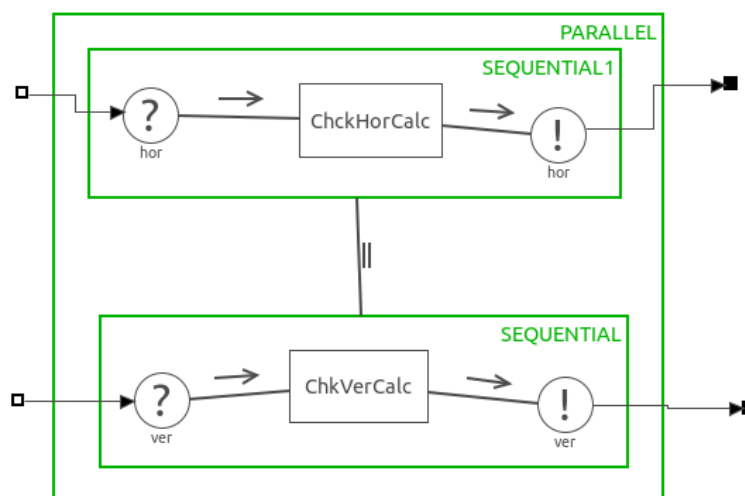
(a) Horizontal

(S) (V)



(b) Vertical

(V)



(c) Check

Figure 11: JIYW submodels.