

Advanced Software Development for Robotics Assignment Manual (v2022.3)

Gijs van Oort and Jan Broenink

Friday 31st March, 2023

Contents

0	Introduction	1
0.1	Context	1
0.2	Organisation	1
0.3	Support from the Teaching Assistants	2
0.4	Responsibility and fraud	2
0.5	Document History and Disclaimer	3
1	Assignment set 1 — ROS2	4
1.0	Assignment 1.0: Getting familiar with ROS2	4
1.1	Assignment 1.1: Image processing with ROS2	4
1.2	Assignment 1.2: Sequence controller	5
2	Assignment Set 2 — Timing in Robotic / Cyber-Physical Systems	8
2.0	Introduction	8
2.1	Timing of periodic non-realtime thread	9
2.2	Timing of periodic realtime thread (Xenomai)	10
2.3	<i>Classical only</i> : Timing of two communicating ROS2 nodes	10
2.4	JIWY System Architecture	11
3	Assignment set 3 — Integration, Controlling a robot	12
3.0	Introduction	12
3.1	Assignment 3.1: JiwY system architecture	12
3.2	Assignment 3.2: Integration	13
3.3	Assignment 3.3: Extra functionality (Bonus assignment)	15
A	Changelog	16
B	Getting Linux and ROS2	17
B.1	ROS2 on virtual machine (Virtualbox)	17
B.2	Installing ROS2 directly on Linux	17
C	Ubuntu22.04-RAM Virtual Machine — User Manual	18
C.1	Introduction	18
C.2	VM Contents	18
C.3	VM User name/password	18
C.4	Installation	18
C.5	Tips & Tricks	19
D	Visual Studio Code	20

D.1	C/C++ Extension Pack	20
D.2	Configuration files	20
D.3	Compiling/building and debugging from within VSCode	20
D.4	VSCode and ROS2	21
D.5	Using VSCode for remote editing	23
E	Jiwy simulator	24
E.1	Dependencies	24
E.2	Input topics	24
E.3	Output topics	25
E.4	Parameters	25
E.5	Differences with a real Jiwy	25
E.6	Getting and compiling the node	26
F	Accessing Raspberry Pi's in the lab — For Assignment Sets 2 and 3	27
F.1	Time slots	27
F.2	User name/account	27
F.3	Connecting to the Raspberry Pi (SSH)	27
F.4	File loss	29
F.5	General remarks	29
G	Compiling for Raspberry Pi / Xenomai	30
G.1	Compiling/debugging generic C/C++ code on the Raspberry Pi	30
G.2	launch.json (Raspberry Pi)	30
G.3	c_cpp_properties.json (Raspberry Pi)	31
G.4	Compiling for Xenomai on the Raspberry Pi	31
H	Jiwy hardware	33
H.1	On/off switch	33
H.2	Interface	33
H.3	Pinout of the PMOD connectors (on the FPGA)	35
H.4	Software development for the Raspberry Pi	35
I	The Xenomai-ROS2-20sim framework (Meijer)	36
I.1	The <code>IcoComm</code> class	36
I.2	Using the framework in your project	38
J	20-sim and the 20-sim model	39
J.1	20-sim	39
J.2	The 20-sim model	40
K	Handing in ROS2 projects	41

K.1 Preparing the hand-in	41
K.2 Into more detail...	41
Bibliography	43

0 Introduction

0.1 Context

The goal of this practical work is get familiar with soft real-time and hard real-time software development, and the combination of it, in the context of controlling robotic/mechatronic systems, i.e. physical machines.

For the soft real-time part (Assignment set 1 and 3), we use ROS2 (Robotic Operating System 2), version Humble, heavily used in modern robotics. The algorithms are programmed in C++. For the development environment, you can either install ROS2 on your *own* laptop, or install a virtual machine running ROS2, see Appendix B.

For the hard real-time part (Assignment sets 2 and 3), we work with a Raspberry Pi 4, using a specifically tuned Linux (Raspberry Pi OS with the Xenomai real-time patch) and programs written in C++. The Raspberry Pi's are available at the RaM Lab. In Assignment set 2, only the bare Raspberry Pi is needed, without additional hardware. In Assignment set 3 a robotic system (a two-DOF webcam movement unit) is controlled with the Raspberry Pi.

The focus in the assignments of this course lies on developing *good, well-structured solutions*. Code that is 'hacked together' and lacks good structure or programming style does *not* receive full points, even if the output of the program is perfect.

0.2 Organisation

The practical assignments consist of three assignment sets, and must be done in groups of 2; all three assignment sets with the same team of 2 students. Results must be submitted via the UT Canvas page of the course (other ways of submission are *not* accepted). You must supply a report (as PDF) as well as a structured zip file with *all* source code (separated per sub-assignment). The code should be such that it can be compiled (and run) by the teaching assistants on their own computer. Deadlines are published on Canvas.

General requirements for the reports are:

- Reports must be compact: only answers need to be given, but always provide a motivation.
- Use a readable, *one-column* layout.
- Include screen dumps of models / diagrams and results in the report.
- For ROS2 programs, *always* put a node/topic graph in the report (`rqt_graph`).
- Put also relevant code snippets in the report, but never put the full code in the report, not even as an appendix.
- The sentence *Show that the system works* as regularly written as part of an assignment task, means that you design, run, and report on some tests to make clear that the system does what you specify it should do. You should address and discuss any anomalies.

Note that the teacher only reads the report and the teaching assistants also check the code in the zip file.

0.2.1 CBL version versus Classical version of the course

MSc-Robotics students who take this class as compulsory course must do the CBL-variant of this course. This means, they can skip certain sub-assignments (those are indicated), and contribute to their CBL case using material from this course, which is indicated in the *framing* Section of the CBL manual.

Those who do not need to do the CBL version, do the *Classical* version of this course, that is all the assignments in this document.

0.3 Support from the Teaching Assistants

You can ask questions regarding the assignments (most notably Assignment Set 1) via *Discussions* available on the Canvas page.

During the Lab sessions of Assignment Set 2 and 3, teaching assistants are of course present in the Lab.

0.4 Responsibility and fraud

All group members share responsibility for the work handed in. When you distribute work among group members, check each other's work, discuss the work among all group members, do you understand it all?

You must submit original work.

Plagiarism, i.e. copying of someone else's work without proper reference, is a serious offence which in all cases will be reported to the Examination Board. Refer to UT's Student Charter Student Charter (2020).

In cases where a non-trivial amount of work is copied *with* proper reference, indicate which parts are copied and which parts are your own original work. The copied work will not be considered for grading.

0.4.1 What constitutes fraud?

Material written by Arend Rensink is reused here.

When it comes down to handing in assignments, every year there are students who do not understand the borderline between, on the one hand, cooperating and discussing solutions between groups (which is allowed), and on the other, copying or sharing solutions (which is forbidden and counted as fraudulent behaviour). Here are some scenarios which may help in making this distinction.

- **Scenario.** Peter and Lisa are quite comfortable with programming and have pretty much finished the assignment. Mark and Wouter, on the other hand, are struggling and ask Lisa how she has solved it. Lisa, a friendly girl, shows her solution and takes them through it line by line. Mark and Wouter think they now understand and go off to create their own solution, based on what they saw. Is this allowed or not?
Verdict. No problem here, everything is in the green. It is perfectly fine and allowed for Lisa to explain her solution, even very thoroughly. The important point is that in implementing it themselves and testing their own solution, Mark and Wouter are still forced to think about what is happening and will gain the required understanding, though probably they will not get as much out of it as Lisa (explaining stuff to others is about the best possible way to learn it better yourself!).
- **Scenario.** The start is as in the previous case. However, while Mark and Wouter implement their own solution, inspired by that of Lisa, some error crops up which they do not understand. Lisa has left by now; after they mail her, still trying to be helpful she sends them her solution for them to inspect. They inspect it so closely that in the end their solution is indistinguishable from Peter and Lisa's, except for the choice of some variable names and the comments they added themselves. Is this allowed or not?
Verdict. This is now a case of fraud. Three are at fault: Lisa for enabling fraud by sending her files (even if it was meant as a friendly gesture) and Mark and Wouter for copying the code. Peter was not involved, developed his own solution (together with Lisa) and is innocent.
- **Scenario.** Alexandra and Nahuel are not finished, and the deadline is very close. The same holds for Simon and Jaco. On the Friday night train home, Jaco and Nahuel meet

and during the 2-hour train ride work it out together. They type in the same solution and hand it in on behalf of their groups. Is this allowed or not?

Verdict. This is also a case of fraud. Actually there are two problems here. The first is that both Nahuel and Jaco handed in code on behalf of their groups that had been developed by them alone, without their partners. This is unwise and against the spirit of the assignment (Alexandra and Simon also need to master this stuff!) but essentially undetectable and not fraudulent. The second problem is that the solution was developed, and shared, in collaboration between two groups; this is definitely forbidden. All four students are culpable; Alexandra and Simon cannot hide behind the fact that they did not partake in the collaboration, as they were apparently happy enough to have their name on the solutions and pretend they worked on it, too.

Note that we are not on a witch-hunt here: let us stress again that cooperating and discussing assignments is OK, even encouraged; it is at the point where you start copying or duplicating pieces of code that you cross the border.

0.5 Document History and Disclaimer

When you find a mistake or have remarks about this document, please contact one of the TAs.

As we also continuously improve this document, newer versions appear regularly, see its date and version number. Changes are summarized in the Changelog, in Appendix A. The latest version is on Canvas. Be sure to always use the latest version of this guide.

This document has seen quite some versions over the years. Contributors to this document, next to the authors: Arnold Hofstede, Muriel vd Spek, Timon Kruiper, Alejandro Lopez Tellez. Marcel Schwirtz supported on the hardware.

Special thanks to Arend Rensink for the subsection “What constitutes fraud?”.

1 Assignment set 1 — ROS2

In this assignment you get familiar with ROS2 and build some projects with it. For Assignment sets 2 and 3 you *must* use Linux. Therefore it is recommended to already start using Linux from the first assignment set (switching halfway is not a good idea). See Appendix B for how to get access to Linux and the required software.

1.0 Assignment 1.0: Getting familiar with ROS2

ROS2 has good tutorials on the website:

<http://docs.ros.org/en/humble/Tutorials.html>.

Do (at least) the following tutorials:

1. Using turtlesim and rqt
2. Understanding nodes
3. Understanding topics
4. Launching nodes
5. Creating a workspace
6. Creating a package
7. Writing a simple publisher and subscriber (C++)
8. Using parameters in a class (C++)

You do not have to hand in anything for this assignment.

1.1 Assignment 1.1: Image processing with ROS2

To get more familiar with ROS2, we do some simple image processing work. However, the focus of this assignment is *not* the image processing itself, but rather the structure of the nodes and topics in ROS2.

1.1.1 Camera input with standard ROS2 tools

Use ROS2 standard tools to capture webcam footage, publish to a channel and view the channel. You can use the following commands¹:

```
ros2 run image_tools cam2image --ros-args -p depth:=1 -p history:=
↳ keep_last
ros2 run image_tools showimage
```

Show that the system works.

Note: you might check the tips in Section 0.2 on finalizing this assignment and producing the report.

Questions

1. Describe in your own words what the parameters `depth` and `history` do and how they might influence the responsiveness of the system. Are the given parameter values good choices for usage in a sequence controller? Motivate your answer.
2. What is the frame rate of the generated messages? How can you influence the frame rate? Is there a maximum? What determines the maximum?

¹If it does not work, you may try to add to the first line: `-p device_id:=n` where `n` is 0, 1, ... This will pick a different camera stream from the list of available streams (check with `ls /dev/video*`).

1.1.2 Adding a brightness node

Create a node that determines the average brightness of the image and, using some threshold, sends on a new topic if a the light is turned on (it is light) or off (it is dark).

Show that the system works.

1.1.3 Adding ROS2 parameters

Change the node in such a way that the threshold is a ROS2 parameter. Show that it is settable from the command line when starting the node (`ros2 run ... --ros-args -p ...`) as well as during run time (`ros2 param set ...`).

Show that the system works.

1.1.4 Simple light position indicator

A very simple way of detecting the position of a bright light in an (otherwise no so bright) image is the following:

1. Gray-scale the image.
2. Apply a threshold on the brightness of each pixel.
If the threshold is well-chosen (and there are no other bright parts in the image), the result is a black image with a large white dot where you shine the light. You may need to reduce background light.
3. Compute the ‘center of gravity’ (COG) of the white pixels. This COG (in pixel coordinates) gives an indication of the position of the center of the light.

Create a node that, given a camera input, outputs the position of a bright light (if there is any) in pixel coordinates. The node should be easy to use, e.g., also when using in a larger project, on a different webcam or with different lighting conditions.

Show that the system works. Also, discuss your design choices.

1.2 Assignment 1.2: Sequence controller

The final assignment set of this course is on controlling a pan-tilt camera unit, called *Jiwy* (Figure 1.1). In order to prepare for the final assignment set, in this assignment we explore the ways of creating a sequence controller in ROS2 and control a simulation model of Jiwy.

1.2.1 Unit test of the Jiwy Simulator

A ROS2 node simulating the Jiwy behaviour is available for this assignment (see Canvas). The node needs a webcam stream from a steady webcam (e.g., the webcam of your computer). For details about the node, refer to Appendix E.

Create a node that generates a sequence of setpoints to test the Jiwy Simulator node. Also create a launch file that sets up all required nodes.

Show that the system works and discuss the design decisions you made.

Questions

1. Plot the setpoint and actual position (use of `rqt` is ok). Explain the results. Don’t forget to look at the time constant of the first order system.

1.2.2 Integration of image processing and Jiwy Simulator

This sub-assignment can be seen as an intermediate step towards the ultimate goal of having the Jiwy Simulator use its own (moving) camera output. In this sub-assignment the data from

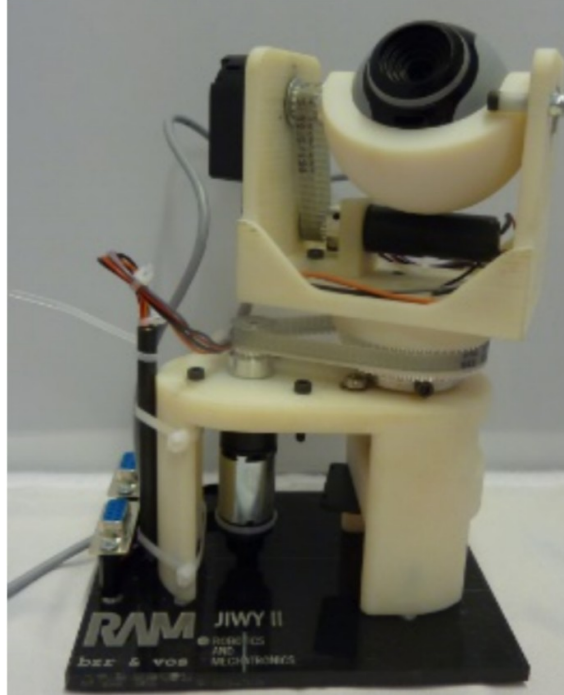


Figure 1.1: The pan-tilt camera unit *Jiwy*.

the (stationary) webcam is used directly (software development goes small, testable, steps at a time...).

Make a new ROS2 node that makes the Jiwy Simulator follow a bright light that you move around in front of your webcam. Use the steady (non-moving) stream of your laptop camera (not the `/moving_camera_output`) as an input for the light position indicator node of Section 1.1.4 and compute appropriate setpoints from that. Remember to keep the system modular (i.e., keep the nodes easily re-usable; thus not too specialized).

Show that the system works and discuss the design choices you made.

Questions

1. Is the system you created an open-loop or closed-loop system (why)? .
2. Is ROS2 (which is a non-realtime platform) in general, appropriate for this type of systems? Why? Are there any limitations (i.e., when do things go wrong)?

1.2.3 Classical only: Closed loop control of the Jiwy Simulator

Classical Only

This sub-assignment is *not* for CBL students. Only for those who take the *Classical* version of ASDfR.

In this part you make Jiwy look at the bright light again, but this time the input of the light position indicator should be the `/moving_camera_output` of the Jiwy simulator (just as in the real case, where the camera mounted on Jiwy moves around during operation).

In order to make this work, you need to steer the camera orientation, $\mathbf{x}_{\text{jiwy}} = [x_{\text{jiwy}} \ y_{\text{jiwy}}]^T$ (in radians), towards the (absolute) position of the light $\mathbf{x}_{\text{light}}$ (in radians) such that the error $\mathbf{x}_{\text{light}} - \mathbf{x}_{\text{jiwy}}$ becomes zero. A convenient controller choice is a first order controller, where the rate of change of the Jiwy orientation, $\dot{\mathbf{x}}_{\text{jiwy}}$, is proportional to the error, i.e.,

$$\dot{\mathbf{x}}_{\text{jiwy}} = \begin{bmatrix} \dot{x}_{\text{jiwy}} \\ \dot{y}_{\text{jiwy}} \end{bmatrix} = \frac{1}{\tau} \cdot (\mathbf{x}_{\text{light}} - \mathbf{x}_{\text{jiwy}}) = \frac{1}{\tau} \cdot \begin{bmatrix} x_{\text{light}} - x_{\text{jiwy}} \\ y_{\text{light}} - y_{\text{jiwy}} \end{bmatrix}; \quad \mathbf{x}_{\text{jiwy}} = \int \dot{\mathbf{x}}_{\text{jiwy}} dt, \quad (1.1)$$

where τ is some time constant (a good start would be $\tau \approx 1$ s).

Notes:

- The position of the bright light in the camera image is precisely $\mathbf{x}_{\text{light}} - \mathbf{x}_{\text{jiwy}}$.
- The only way to influence the camera orientation is by sending a setpoint \mathbf{x}_{set} to the simulator node. Fortunately, the Jiwy can track its setpoint well, so we can assume that $\mathbf{x}_{\text{jiwy}} = \mathbf{x}_{\text{set}}$, resulting in

$$\dot{\mathbf{x}}_{\text{set}} = \begin{bmatrix} \dot{x}_{\text{set}} \\ \dot{y}_{\text{set}} \end{bmatrix} = \frac{1}{\tau} \cdot (\mathbf{x}_{\text{light}} - \mathbf{x}_{\text{jiwy}}) = \frac{1}{\tau} \cdot \begin{bmatrix} x_{\text{light}} - x_{\text{jiwy}} \\ y_{\text{light}} - y_{\text{jiwy}} \end{bmatrix}; \quad \mathbf{x}_{\text{set}} = \int \dot{\mathbf{x}}_{\text{set}} dt. \quad (1.2)$$

Develop the system described above. For the integration (second equation of (1.2)) you can use the Forward Euler integration method.

Show that it works (i.e., that the simulated Jiwy can track the bright light). Also discuss your design choices.

Questions

1. Is the system you created an open-loop or closed-loop system (why)?
2. Is ROS2 (which is a non-realtime platform) in general, appropriate for this type of systems? Why? Are there any limitations (i.e., when do things go wrong)?

Note

Keep the nodes you have programmed for this assignment set; you might need them again in assignment set 3.

2 Assignment Set 2 — Timing in Robotic / Cyber-Physical Systems

2.0 Introduction

In this assignment, you are analyzing the timing of a Robotic or Cyber-Physical System, in which the SRT parts are ROS2 nodes. More specifically, it is about the timing of communication between ROS2 nodes. By doing so, you can find out to what extent ROS2 is SRT-capable. In particular, what jitter appears in the communication between ROS2 nodes. Factors such as computational load and how busy a processor is obviously affect this, and thus require investigation. Furthermore, of importance is characterising roundtrip times: a bidirectional, synchronised communication between two nodes, like in closed-loop control systems. And of course, how such a roundtrip behaves under several load conditions.

Next to this, by characterising the Raspberry Pi 4 with respect to timing, using a kind of bare implementation (that is, without the ROS2 stuff), you are also setting up a basic timer experiment on the Raspberry Pi 4.

2.0.1 Raspberry Pi 4 in the Lab

The larger part of this assignment set consists of doing timing measurements on a Raspberry Pi 4. They are situated in the RaM Lab (Carré 3434).

Each group is allowed 3×2 hours on a Raspberry Pi, spread across the range of 8 March 2023—24 March 2023. You *must* subscribe to a time slot in Canvas before you are allowed to use it. See Appendix F for details on the time slots and on how to log in to the Raspberry Pis.

The time you have available at a Raspberry Pi is limited. Therefore it is vital to be well prepared:

- The programs of Section 2.1 and Section 2.3 can be fully prepared and tested at home on the Virtual Machine. If you program correctly, the code can run both on the VM and on the Raspberry Pi without any code modifications. Note that you *must* recompile though (throw away all object files (.o) and executables before recompilation).
- The program of Section 2.2 can not be fully tested on the VM because the VM has no Xenomai functions. However you can do a best-effort to write the program without compiling and testing the final version. As an example, some POSIX commands require settings for running in real-time which are not available on the VM (e.g., `SCHED_X`). During development you can keep them at the normal settings and only change them when you can work on the Raspberry Pi.
- Do any post-processing and report-writing later.

2.0.2 Documentation/references

- Documentation for all POSIX C functions is conveniently embedded in Linux; you can use the command-line command `man` for this, e.g., from the Linux command line:
`man pthread_create`
Alternatively, an online version of the man-pages is at
<https://man7.org/linux/man-pages/index.html>
- Specifically for documentation on `CLOCK_REALTIME` and `CLOCK_MONOTONIC` (used in Section 2.1), refer to the manpage of `clock_gettime` (they are described in many other man-pages as well, but in less detail).
- Documentation on Xenomai 3.1 is at
<https://source.denx.de/Xenomai/xenomai/-/wikis/home>.

2.1 Timing of periodic non-realtime thread

In Linux there are two ways to get timed periodic execution of code:

- using `clock_nanosleep`, in which you repeatedly tell the process to sleep until a certain absolute time, or
- using `timer_create` and `sigwait`, in which you set a periodic timer once and subsequently call `sigwait` each sample to sleep until the next time the timer fires (interrupt).

Usually, the part of the code that is executed periodically is situated inside a `while` (or `for`, if you only want to run for a known, finite amount of time) loop.

Write a program that spawns a POSIX thread which has a `while` (or `for`) loop inside that contains timed periodically executed code. Choose either `clock_nanosleep` or `timer_create/sigwait` to accomplish this. The code should be executed every 1.0 ms. Let the loop do some computational work so that it actually spends some time inside the loop.

Run the program twice: without and with extra processor load (from another program).

Investigate the timing of the loop. Discuss any particularities. Does extra computational load significantly affect the timing performance?

- The timing measurements should be taken when the program runs on the Raspberry Pi 4. However, you can prepare and test the program at home. See Section 2.0.1.
- Use `clock_gettime` for accurate time measurement.
- Avoid printing to screen/file IO inside the loop as this may heavily influence the timing of the loop.
- Make sure that the program exits nicely; e.g., the main thread waits for the thread to end, files and other resources are properly closed etc.
- For computational work, consider doing some math. Experiment a little with the amount needed to get a nice processor load (do not flood it though).
- You can use the command line tool `htop` to view the processor load.
- To generate extra processor load extra, you can use the command line tool `stress` (both on the virtual machine¹ and the Raspberry Pi). For this, open an extra terminal or extra SSH connection and start `stress` in there before running your program. Think of suitable parameters yourself (motivate them).

Questions

1. Explain what timing aspects we are interested in and why. Keep in mind that the course is on software development *for robotics*; a typical use case here is doing closed-loop control with a given sample time. Give at least two ways of representing the measured data. Discuss advantages and disadvantages of them.
2. Explain why you have chosen the method you used (either `clock_nanosleep` or `timer_create/sigwait`). What are advantages and disadvantages of both methods?
3. What is the difference between `CLOCK_MONOTONIC` and `CLOCK_REALTIME`? Which one is better for timing a firm/hard real-time loop?
4. When using `timer_create/sigwait`, what happens with the timer you initialized when the program ends? And what happens when it crashes due to some error?
5. Is it a good idea to use the approach used in this subassignment for closed-loop control of a robotic system? Why (not)?

¹If this command is not found, you can install it with `sudo apt install stress`.

2.2 Timing of periodic realtime thread (Xenomai)

Make a variant of the program of Section 2.1 in which the periodic loop runs real-time under Xenomai.

Investigate the timing of the loop, that is, display the jitter in a diagram. Discuss any particularities. Does extra computational load significantly affect the timing performance?

- This program must run on Xenomai on the Raspberry Pi 4. Obviously, you can *not* fully prepare and test this at home on the VM. See Section 2.0.1.
- Most POSIX functions can stay the same; those functions are made real-time by linking to real-time variants of the functions (see Appendix G).
- However, some extra work needs to be done to make the environment real-time capable.
- See Appendix G for details on how to compile for Xenomai.
- You can still use `htop` to show processor load. However, this command only shows load from processes run from Linux; not from Xenomai. To show the load by Xenomai processes, use the command `cat /proc/xenomai/sched/stat`. Apart from the load (%CPU), this command shows some other statistics, such as the number of *mode switches* (MSW) made.

Questions

1. For each POSIX command used in your program, explain *in your own words* what it does and why you need it for a real-time program. If the command has arguments that influence the real-time behaviour (e.g., the scheduling type `SCHED_X`, or the clock type `CLOCK_X`), discuss why the chosen option is the best for real-time performance.
2. Compare the timing performance of this program with the program from Section 2.1. What can you conclude on timing with respect to real-time capability?
3. Is it a good idea to use the approach used in this subassignment for closed-loop control of a robotic system? Why (not)?

2.3 Classical only: Timing of two communicating ROS2 nodes

Classical Only

This sub-assignment is *not* for CBL students. Only for those who take the *Classical* version of ASDfR.

Create two ROS2 nodes, `Seq<nn>` and `Loop<nn>`, where `nn` is your group number. Let each node run on 1 kHz, using *its own, individual* timer. Let `Seq<nn>` send a message each sample time to `Loop<nn>`. Let `Loop<nn>` send a message back to `Seq<nn>` *in reply to* the message it got from `Seq<nn>`.

Run the program twice: without and with extra processor load. Investigate the timing, that is show jitter, round-trip time, and jitter of round-trip time graphically. Discuss any particularities. Does extra computational load significantly affect the timing performance?

- Again, the timing measurements should be taken when the program runs on the Raspberry Pi 4. However, you can prepare and test the program at home. See Section 2.0.1.
- Take into account the hints in Section 2.1.
- To match which sent and received messages belong together `Seq<nn>`, it is wise to send unique messages (e.g., with unique reference numbers).
- When testing the round-trip time of messages, take into account that the policies which are set for the communication between ROS2 nodes affect the results. An example is enabling or disabling message buffering. Recall from Section 1.1.1 that you can affect these policies with

```
ros2 run ... --ros-args -p ...
```


For a list of arguments, see <https://docs.ros.org/en/humble/Concepts/About-Quality-of-Service-Settings.html>

- Usually, a ROS2-node executes indefinitely and is stopped by completely stopping the program containing the node with `Ctrl + C`. However, when doing measurements, you probably want to run the experiment for a known, finite amount of time and do some post-processing in the same program afterwards. One way of accomplishing this, is to stop the node from within, which is done by calling `rclcpp::shutdown()` inside its functions. Doing this is not mandatory though.

Questions

1. Compare the results from this subassignment with the results from Assignment 2.1. Discuss the results.
2. Motivate your tuning of communication policies, that is, which parameters you have used, or not.
3. Is it a good idea to use the approach used in this subassignment for closed-loop control of a robotic system? Why (not)?

2.4 JIWI System Architecture

As preparation of the integration of image processing in ROS2, calculating the control law in FRT on Xenomai on a Raspberry Pi 4, and controlling a real JIWI robot, you design the architecture of the cyber (software) part of the total system.

Do the subassignment described in Section 3.1.1, that is, construct a high-level block diagram of the software you are developing.

3 Assignment set 3 — Integration, Controlling a robot

3.0 Introduction

3.0.1 How to do this assignment set

The larger part of this assignment set consists of doing work on a Raspberry Pi with an actual robot. This part of the work is done in the lab of the RaM group, Carré building, CR-3434.

Each group can use 2 half days (9:00 - 12:30 or 13:30 - 17:00) using one of the five *Jiwy* + Raspberry Pi 4 setups. You *must* subscribe via Canvas to a time slot before you are allowed to use it.

The time you have at a setup is limited, therefore it is vital to be well-prepared. Just like for Assignment Set 2.

How to connect to the setup with your own laptop is presented in Appendix F. Furthermore, presenting graphical output of a program while running it on a Raspberry Pi, so-called *X-forwarding* is explained in the same Appendix.

Handing in a ROS2 project needs some specific action, see Appendix K.

3.0.2 Preparation

As preparation for this assignment set, you need to read the MSc. report by Bram Meijer (Meijer, 2021), especially Chapter 4. See the Handouts page on Canvas for a link to this MSc. report.

3.1 Assignment 3.1: Jiwy system architecture

In this final assignment, set you control a real robot, the Pan-Tilt unit *Jiwy*, such that it follows a bright light. You make use of the MSc. work of Bram Meijer (Meijer, 2021), who developed a framework that allows for code generation from 20-sim to Xenomai on a Raspberry Pi 4 with an interface to ROS2. Appendix H describes the provided hardware. Appendix I describes how to use the framework by Meijer.

In this Assignment Set, you should re-use your own work from Assignment Set 1.

In this Assignment Set, you need to develop the software implementing (at least) the following:

- Closed-loop position controller — given a position setpoint \mathbf{x}_{set} and the actual position \mathbf{x}_{jiwy} , it computes the required motor power to reduce the error $e = \mathbf{x}_{\text{jiwy}} - \mathbf{x}_{\text{set}}$ to zero (PID control). This controller runs on 1 kHz.
- Closed-loop sequence controller — given the (moving) camera image with some bright light in it, it computes the required setpoint \mathbf{x}_{set} to steer the Jiwy such that the bright light ends up in the middle of the camera image. The sample frequency of this loop controller is much lower than 1 kHz. Think of a suitable sample frequency yourself (motivate). Developing this component was done in subassignment 1.2.3. Because CBL-students have not done this subassignment, we provide an implementation of the node on Canvas. Anyone (also Classical students) can use this instead of their own implementation if they wish. The node implements equations (1.2) (including the integration equation). Note that, together with the notes just above (1.2), it appears that you do *not* need to feed back the actual position of Jiwy (\mathbf{x}_{jiwy}) to this node!
- Relevant signals should be made available on ROS2 topics for monitoring/debugging.

3.1.1 Block diagram

Note: This subassignment (Section 3.1.1 only) must be done as part of Assignment Set 2!

Construct a high-level block diagram of the structure of the software you are developing.

- Detailing should be to the level of ROS2 nodes and topics; similar detail is expected for the non-ROS side.
- Clearly indicate the boundaries between non/soft/firm/hard real-time.
- Refer to Figure H.2 for the hardware structure; it gives information on the expected inputs and outputs of the Raspberry Pi.
- Thoroughly read all of Chapter 3 before doing this subassignment.

3.1.2 Details of each block

Note: If you received considerable feedback on the subassignment of Section 3.1.1 given as part of the feedback on the assignments in Chapter 2, you can update the block diagram, as this helps the work in the rest of this assignment. You may also change the earlier-made block diagram as result of new insight. Motivate your changes in any case.

Per block/communication channel, specify at least the following aspects:

- For each module/node/thread:
 - Name
 - Description of its function
 - Environment (e.g., ROS2, Non-RT, Xenomai)
 - Sample time (if applicable)
- For each communication channel:
 - Signal Name
 - Type of channel (e.g., ROS topic, XDDP, IDDP, SPI, electrical, ...)
 - Data type and (for vectors) size and element assignment (which element is what)
 - For signals representing physical quantities: unit (deg, rad, ...)
 - Buffered/unbuffered (buffer size); Real-time/non-real-time
 - Sample time (if applicable)

3.2 Assignment 3.2: Integration

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

3.2.1 ROS2 on the Raspberry Pi

Reproduce the experiment from Section 1.2.3 on the Raspberry Pi.

Show that the system works.

- Re-use the nodes you developed in Assignment Set 1.
- The Jiwy USB camera is connected to the Raspberry Pi.
- `rqt` is not installed on the Raspberry Pi (it *cannot* easily be installed). Fortunately, `showimage` is available. Because the Raspberry Pi does not have a graphical environment, you cannot see the window directly on the Raspberry Pi. Instead, the graphics stream can be forwarded to your laptop (*X-forwarding*). See Appendix F, Section F.3.4 for details.
- You can use the command line tool `htop` to view the processor load.
- Store results of this subassignment separately, that is, prepare it for submission (see Appendix K), and set it apart in a separate directory.

Questions

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

2. Did you have to make changes to the experiment to get it running on the Raspberry Pi? If so, why was it necessary? Could you have avoided it if you had done the experiment in Assignment 1.2.3 differently?
3. Did you have to make changes in the nodes 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?

3.2.2 Code generation of the position controller from 20-sim

Implement the loop-controller in a Xenomai thread by creating C-code from 20-sim (see below) and using the framework from Meijer.

Design a way to test the module, at least to see if it runs (it may be hard to check correctness because unfortunately there is no simulation model in Xenomai available).

Show that the system works, discuss your design choices.

- Keep in mind the block diagram you made in Assignment 3.1.
- Provided for this part is a 20-sim model with a simulation model of Jiwy as well as an appropriate position loop-controller. Appendix J gives more details about how to use 20-sim and about the Jiwy-model.
- Use the `wrapper` class. See the MSc. report by Bram Meijer (2021, Chapter 4) and Appendix I.
- You are not allowed to modify the files from the framework (that would destroy modularity). If you need extra functionality, make a subclass that implements the extra functionality. If you think there is no other way than to modify the framework, discuss with the teaching assistant first.
- You may create code from the plant block in 20-sim and use that to validate the controller, but this is not compulsory (bonus).
- If you do not want to use 20-sim, you can use any other modeling package (e.g., Simulink) for code generation. In that case, you need to design the position controller yourself. You cannot use the `wrapper` class but you need to base your work on the `ownThread` class. Note that you *must* do code-generation from some modeling package; writing the controller directly in C/C++ is not accepted.
- Store results of this subassignment separately, that is, prepare it for submission, and set it apart in a separate directory.

Questions

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

3.2.3 Measurement and Actuation block

Implement the Measurement and Actuation functionality, using the provided classes (see bullet points below) as a starting point. In particular, create the functions `ReadConvert` and `WriteConvert`. Motivate the choice of scaling factors and equations.

Design and perform a *unit test* of this block (i.e. test this block separately, so without using the controller class from Section 3.2.2).

Show that the system works and validate the correctness of the Measurement and Actuation block.

- Make sure the directions (i.e., which direction is called positive) adhere to those mentioned in Appendix E.

- In general, the *implementation structure* does not need to be exactly identical to the *functional structure*. For example, in some cases it can be beneficial (e.g., from a computational point of view) to merge two *functional blocks* into one C-class or node. In this situation it makes sense to embed the Measurement and Actuation conversion functions in the classes that take care of the SPI communication to the icoBoard FPGA. The provided classes already support this. See Appendix I.
- One of the ways to test the module is by creating a new 20-sim model which relays the SPI data to/from ROS2 topics, and generating code from that (but that is not the only way; it is up to you to choose how to do it).
- To test the encoders, you can carefully rotate the camera by hand.
- An oscilloscope is available to measure the PWM signals; see Appendix H.
- Store results of this subassignment separately, that is, prepare it for submission, and set it apart in a separate directory.

Questions

1. 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. Discuss advantages and disadvantages (in terms of maintainability, ease of implementation, modularity, computation speed, ...). Is the chosen implementation your preferred one? Why?

3.2.4 Final integration

Create the full system of ‘Jiwy following a bright light’ by composing all blocks you have built and (individually) tested in the previous sections. Think of a good order of integrating the blocks and testing, motivate why you chose that order. Show and discuss intermediate results.

Show that the final system works; discuss your design choices.

- Simply “connecting all blocks at once and hope for the best” is not a modular way of working. If it does not work, it is hard to track down the faults (‘integration hell’).
- Store results of this subassignment separately, that is, prepare it for submission, and set it apart in a separate directory. Prepare the ROS2 parts as indicated in Appendix K.

Questions

1. 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?

3.3 Assignment 3.3: Extra functionality (Bonus assignment)

In this bonus assignment you can add functionality to the, already working, setup. Some ideas:

- Homing procedure
- Safety layer: endstops
- Advanced behaviour in the case no bright light is seen
- Extra safety: stop the movement of the whole robot if one of the degrees of freedom does not function as expected (test by disconnecting one of the PMOD cables)
- Your own idea...?

Describe what extra functionality you want to make and how it will influence the behaviour of the whole system. Explain how you implemented the extra functionality. Motivate the design choices. Show and discuss test results.

- Note that, as with all assignments, the focus is on developing *good, well-structured solutions*. A very cool feature which is just ‘hacked’ into the system will not give you points.
- If you have developed code, store it separately, as is done with the other subassignments.

A Changelog

- **Version 2022.3, 31 March 2023**

New parts:

- Appendix K (Handling in ROS2 projects)

Updates:

- Chapter 3 (Assignment3) added tips on updating block diagram and tips on submitting work, including reference to Appendix K on handling in ROS2 projects.
- Repaired typos and inconsistencies throughout the whole text.

- **Version 2022.2, 6 March 2023**

New parts:

- Chapter 2 (Assignment 2)
- Chapter 3 (Assignment 3)
- Appendix F (Accessing Raspberry Pi in the Lab)
- Appendix G (Compiling for Raspberry Pi and Xenomai)
- Appendix H (Jiwy Hardware)
- Appendix I (Xenomai-ROS2-20sim framework)
- Appendix J (20-sim and the 20-sim model)

Updates:

- Appendix E (Jiwy Simulator): added output topic `/position` to Figure E.1.
- Repaired typos and inconsistencies throughout the whole text.

- **Version 2022.1, 8 February 2023**

Initial version, consisting of:

- Chapter 0 (Introduction)
- Chapter 1 (Assignment set 1)
- Appendix A (Changelog)
- Appendix B (Installing ROS2)
- Appendix C (Ubuntu 22.04 VM)
- Appendix D (Visual Studio Code)
- Appendix E (Jiwy Simulator)

B Getting Linux and ROS2

For the first assignments you need ROS2 on your own computer. For the later assignments you *must* use Linux. These requirements combine into two options for software installation:

- *Run ROS2 on a virtual machine (Virtualbox) running Ubuntu 22.04.*
This is the recommended way. We provide a virtual machine which has everything you need pre-installed.
- *Run Ubuntu natively on your laptop and install ROS2 yourself.*
If you happen to run Ubuntu on your laptop you can install ROS2 directly on it. Note that in the later assignments you need 20-sim as well. You can either use a separate Windows PC for that or install it on Linux by using Wine (installation instructions will follow in an update of this manual).

B.1 ROS2 on virtual machine (Virtualbox)

See Appendix C.

B.2 Installing ROS2 directly on Linux

Installation instructions are on the ROS2 site: <https://docs.ros.org/en/humble/Installation.html>. Installing binary packages is usually easier than compiling from source.

For Ubuntu users, there is a convenient script that does the full installation unattended; have a look at https://github.com/Tiryoh/ros2_setup_scripts_ubuntu. We used this script to install ROS2 on the virtual machine provided, as follows:

```
wget https://raw.githubusercontent.com/Tiryoh/ ros2_setup_scripts_ubuntu
↪ /main/ros2-humble-desktop-main.sh
chmod 755 ros2-humble-desktop-main.sh
./ros2-humble-desktop-main.sh
```

C Ubuntu22.04-RAM Virtual Machine — User Manual

C.1 Introduction

This document/chapter¹ gives some information on how to work with the *Ubuntu22.04-RAM* Virtual Machine. It was prepared for usage with the courses *Advanced Software Development For Robotics* (ASDFR), *Programming 2* (optional) and *Software Development For Robotics* (SDFR), but of course usage is not limited to these courses; anyone who can benefit from this readily-installed VM can use it.

C.2 VM Contents

The Virtual Machine does *not* contain course-specific software (such as templates for assignments; these have to be downloaded by the students from Canvas). However, it does contain software and tools that are used by the courses, being:

- Ubuntu Desktop 22.04
- Visual Studio Code (VS Code) with some extensions:
 - cpptools, cpptools-extension-pack *Working with cpp*
 - cmake-tools *Working with cmake*
 - doxygen *Doxygen support*
 - githistory, gitlens *Better support for git from within VS Code*
 - latex-workshop *Work with L^AT_EX files*
 - remote-ssh *Directly edit files on a remote target such as a Raspberry Pi*
- ROS2 (Humble Hawksbill)
- SDL2
- Various small tools, e.g., git, vim, wget, ...
- Wine (required for 20-sim)
- 20-sim 5.0.2 (license key until 1 February 2024)

C.3 VM User name/password

There is one user installed:

- User: ram-user
- Password: ram-user

This user has admin rights.

C.4 Installation

1. Download and install Virtualbox (www.virtualbox.org). Version 7.0 is recommended.
2. If your host OS is Linux, you might need to add yourself to the `vboxusers` group. This can be done by typing into a terminal (on your host OS):

```
sudo usermod -a -G vboxusers `whoami`
```

and logout/login afterwards.
3. For webcam support (required for the assignments): go to www.virtualbox.org/wiki/Downloads and download and run the Virtualbox Extension Pack.
4. Download the virtual machine file `Ubuntu22.04-RAM.ova`. Where you get it may vary. If you are attending a course, look on Canvas. Alternatively, ask the person you got this user manual from. The file is approximately 11 GB.
5. In Virtualbox, *Choose File | Import appliance* and import the virtual machine.

¹Depending on if the text was embedded in a larger document

6. Start the virtual machine.
7. Depending on the configuration, you may need to log in. For usernames and passwords, see Section C.2.
8. To use the webcam in the virtual machine, from the drop down menu item *Devices|Webcams* of the Virtualbox window, check the appropriate webcam. Test if this works by typing (in a terminal on the virtual machine):

```
ros2 run image_tools cam2image --ros-args -p show_camera:=true
```

C.5 Tips & Tricks

- The `ros2` command is available directly (i.e., *source*ing the ros installation directory is not needed). If you don't want this, comment out the lines

```
source /opt/ros/humble/setup.bash
source /usr/share/colcon_argcomplete/hook/colcon-argcomplete.bash
```

from the file `~/.bashrc`.

D Visual Studio Code

At this moment, Visual Studio Code¹ is the most-used editor for coding. It is available for Windows, Mac and Linux and has an enormous amount of available extensions. We encourage you to use this editor (if you don't already). In the remainder of this chapter we will use the term VSCode for the editor. On the internet there are thousands of helpful pages on how to use VSCode in specific circumstances. Also, the application itself provides help; e.g., there are quick tutorials linked on the opening screen.

D.1 C/C++ Extension Pack

To make use of all features listed below, make sure you have the *C/C++ Extension Pack* extension installed. This pack contains nine useful extensions for developing C/C++ code, including the Remote-SSH extension (Section D.5).

D.2 Configuration files

Configuration of VSCode usually goes per project/(VSCode-)workspace. When you open a folder in VSCode, it creates a directory `.vscode` in the base directory, containing one or more settings files. These files are human-readable (and -editable):

- `.vscode/settings.json` General project settings such as the color scheme used.
- `.vscode/tasks.json` Compiler/Build settings
- `.vscode/launch.json` Debugger settings
- `.vscode/c_cpp_properties.json` Settings for correctly parsing/checking C/C++ files. For example, you supply include-paths here so that the VSCode can look up header files and use the information in there for code completion, real-time error checking, hinting etc.

You can edit these files so that they optimize the integration for your system. For example, they can contain paths to header files that VSCode then can use for syntax highlighting and code-completion. Some VSCode extensions provide (some of) the configuration files automatically.

D.3 Compiling/building and debugging from within VSCode

Building refers to the full process of generating an executable from source files. In simple C++ projects this usually boils down to compiling and linking (for more complex projects, such as ROS projects, more steps may be involved). The steps described here do not apply to ROS project (for those, the configuration files can be generated automatically; see Section D.4).

D.3.1 Compiling/building

Building a project is done by pressing Ctrl-Shift-B. Building instructions for VSCode are in the `.vscode/tasks.json` file. If there is no such file, a default building task can be chosen from a popup menu. Choose the building task with the recommended compiler for your OS (g++ for macOS and Linux, `cl.exe` for Windows). The default building task only compiles the active file.

For a multiple-C++-file project, you need to customize the build task. Make sure you have `.cpp` as active (editor) file. Press Ctrl-Shift-P to open the Command Palette, and type `Tasks: Configure default build task`. Choose the appropriate build task that will be your starting point. This creates a `.vscode/tasks.json` file which you can edit. Some things which you may want to change as stated below.

The file that is compiled is `"${file}"`. You can modify this to, for example, `"${workspaceFolder}/*.cpp"` or `"${workspaceFolder}/src/*.cpp"` to compile and link all cpp files that are in the VSCode

¹<https://code.visualstudio.com/>

base directory or the `src` directory under the VSCode base directory respectively. Since the `tasks.json` file is specific for your current project, it makes sense to specialize it and explicitly list all `cpp` files that need to be compiled. In that case, each file is a separate entry, e.g., `"${workspaceFolder}/src/main.cpp", "${workspaceFolder}/src/divide.cpp"`.

The name of the generated executable is the line after the `"-o"` argument; its default is `"${fileDirname}/${fileBasenameNoExtension}"`. This implies that as soon as your active file changes (i.e., you start editing a different file), the executable name changes (because `${fileBasenameNoExtension}` changes value). This is often not what you want. Instead, you can supply a fixed file name, e.g., `"${workspaceFolder}/divide.exe"` (or without the `.exe` part for Linux/macOS).

If you have problems with getting your header files included, try the `-I` flag

D.3.2 Debugging

Debugging a project is done by pressing F5. Debug instructions for VSCode are in the `.vscode/launch.json` file. If there is no such file, a default debugging configuration can be chosen which only debugs the active file. Prior to debugging the file is (re)built as well.

Debugging a multiple-C++-file project is handled automatically. If you have configured the building task correctly, the default debugging task is chosen accordingly; no need to change things here.

If you want more control on the debugging process, you can create a custom `.vscode/launch.json`. From the pull-down menu, choose "Run|Add configuration". An 'empty' `.vscode/launch.json` file is created. In the edit window, click the 'add configuration' button and choose 'C/C++ (gdb) Launch'. This gives you a starting point for your customization.

Debugging only starts to become fun if you add breakpoints. To add/remove a breakpoint at a certain line, put the cursor on the line and press F9. A red dot appears in front of the line. Alternatively, click on the place where the red dot is about to appear.

D.3.3 Troubleshooting

If you think you may have messed up your compile/build/debug configuration, you can remove the `.vscode` directory in your base directory to start from scratch again (actually, just removing `tasks.json` and `launch.json` suffices; the other configuration files don't have anything to do with building/debugging).

There are also extensions that claim to do this configuration work for you, but our experience is that it is still good to know what's going on under the hood, so that you can fix any inconsistencies that the extensions introduce.

D.4 VSCode and ROS2

There is a nice VSCode Extension to make Visual Studio Code work properly with ROS2. The extension is inconveniently called ROS, but it works for both ROS(1) and ROS2. This extension configures VSCode (via the files described in Section D.2) so that it does code-completion, building and debugging (including breakpoints) of ROS nodes.

Install the ROS extension from Microsoft (ms-iot.vscode-ros) yourself; it may not be installed in the Virtual Machine that we provided.

D.4.1 Preparing your workspace for VSCode

The extension does not work great for individual nodes; if you want debug a node you *must* create a launch file for it and configure your workspace accordingly. Do the following:

1. Create a launch file in the appropriate directory
(`<ros_ws>/src/<package name>/launch/mylaunchfile.launch.py`). See the ROS2 tutorials.
2. Add the following line to `package.xml`:
`<exec_depend>ros2launch</exec_depend>`
3. Add the following lines to `CMakeLists.txt`:

```
!# install the launch directory
install(DIRECTORY
  launch
  DESTINATION share/${PROJECT_NAME}/
)
```

4. Remove the directories `build`, `install` and `log` from your workspace (to allow VSCode to do a complete rebuild of the workspace).

D.4.2 Opening the Workspace and enable ROS Extension in VSCode

In VSCode, choose File|Open Folder and open the root of your workspace folder (i.e., the folder where `src`, `build`, `install` and `log` are situated normally). This acts as the *base directory* for VSCode. VSCode now automatically recognizes the ROS2 workspace. Hence it will (later) create `.vscode/c_cpp_properties.json` and `.vscode/settings.json` with ROS-friendly settings. This enables syntax highlighting and code completion (even of your own custom message types!).

D.4.3 Building your ROS workspace

Press Ctrl-Shift-B to build your entire workspace. VSCode asks you what/how to build:

- Choose 'Colcon: build' as build task.

This should invoke `colcon build` with some additional parameters for convenient debugging.

D.4.4 Debug a node (with the built-in debugger)

This requires a `launch.json` file to be automatically generated. For this:

- Go to 'Run and Debug' in the side bar (Ctrl-Shift-D or View|Run from the pull-down menu)
- Close all files in VSCode (failure to do this will not give the ROS option in the next step)
- In the side bar, choose 'Create a launch.json file'.
- Choose ROS as debugger
- Choose ROS Launch
- Choose the appropriate package name and node name.

Now a `launch.json` file is created. To start debugging, set some breakpoints in your `cpp` files and press the green triangle in the 'Run and Debug' sidebar (or press F5).

D.4.5 Run a node

The equivalent of `ros2 run` is:


- Press Ctrl-Shift-P to open the command window of VSCode
- ROS: Run a ROS Executable
- Choose the package and node name

D.4.6 More information on VSCode and ROS2

A good source on how to work with VSCode and ROS2 is the *Visual Studio Code ROS Extension* video series on Youtube: <https://www.youtube.com/playlist?list=PL2dJBq8ig-vihvDVw-D5zAYOArTMIX0FA>.

D.5 Using VSCode for remote editing

You can use VSCode on your own computer work with files on a remote computer, for example a Raspberry Pi. This is not limited to editing the files, but you can also compile, run and debug remote files as well as using `git` on those files.

In order to use this feature, you need the *Remote - SSH* extension of VSCode installed. Also, you need SSH access to the remote computer (test in a terminal window). Then, to use it, press the -icon on the lower left of the screen.

After connecting, VSCode automatically downloads some ‘server code’ to the remote computer to support the editing/developing process. However, not all functionality is available right away. In particular, for syntax checking and debugging/running C/C++ files you need to install the ‘C/C++ Extension Pack’ extension on the remote computer. This is how to do it:

1. In VSCode, log in to the remote computer as described above.
2. Press Ctrl-Shift-P to open the Command Palette
3. Choose the *Remote: Install Local Extensions In ‘Remote’* command
4. Choose the *C/C++ Extension Pack* extension.

E Jiwy simulator

Provided for Assignment set 1 is a ROS2 node that simulates the Jiwy behaviour. The node uses the webcam of your computer (or, actually it accepts an image stream on a topic; you can offer webcam footage to that) as input.

Instead of physically rotating the camera, it outputs a part of the full camera view in a topic. Additionally, it outputs its actual pan and tilt angle. See Figure E.1.

As an input, the node expects a setpoint, i.e., a desired pan and tilt angle. Dynamics are modelled as a rather slow (to see the effect) first order system. End stops are included as well.

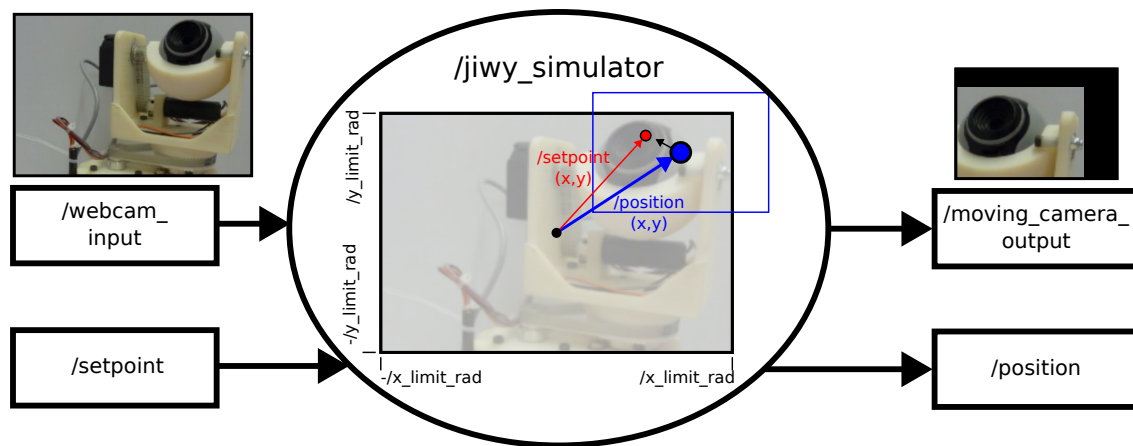


Figure E.1: Schematic drawing of the functionality of the Jiwy Simulator node.

E.1 Dependencies

The node used the custom message type `asdfr_interfaces/msg/Point2`. Before invoking the node, make this message type known to ROS2 (in the `ros2_nodes` directory, invoke `. install/local_setup.bash`).

The message type of `Point2` is:

```
~$ ros2 interface show asdfr_interfaces/msg/Point2
float64 x
float64 y
```

E.2 Input topics

`/setpoint`

The desired pan (x) and tilt (y) angle of the simulated camera in radians.

- The message type is `asdfr_interfaces/msg/Point2`.
- A positive pan (x) angle makes the camera look to the right.
- A positive tilt (y) angle makes the camera look upward.
- `[x=0, y=0]` makes the camera look straight forward.
- Setting a setpoint outside the working area (i.e., beyond the end stops) is allowed; the camera will just move to the end stop and not further.
- You can publish to this topic as often (or as little) as you like.
- The initial setpoint is `[x=0, y=0]`.
- You can give setpoints to the Jiwy Simulator from the command line with the following (standard) ROS2 command:

```
ros2 topic pub --once /setpoint asdfr_interfaces/msg/Point2 "{x: 0.3,y: 0.2}"
```

Note that you *must* place a whitespace after the `:`'s. Also make sure you have executed `. install/local_setup.bash` first.

`/webcam_input`

A webcam stream of a stationary webcam (e.g., your laptop's webcam).

- Consider using `ros2 run image_tools cam2image` with appropriate topic remapping.

E.3 Output topics

`/position`

The actual pan (x) and tilt (y) angle of the simulated camera in radians.

- This topic is published to at approximately (but not necessarily precisely) 100 Hz.
- Refer to the `/setpoint` topic for details about the positive orientation and zero angle.

`/moving_camera_output`

The simulated output image of the simulated camera.

- This topic is published to whenever a new message from `/webcam_input` arrives.
- It is a part of the original input image; padded with black pixels if necessary.
- The width and height of the output image are half the width and height of the input image (rounded down if necessary).
- The center of the output image is looking at `/position`.
- The position scaling is such that at the end-stop positions the center of the output image coincides with the edge of the input image.

E.4 Parameters

`/tau_s`

The time constant of the first-order dynamics system, in seconds. You can freely change this while running.

`/x_limit_rad`

`/y_limit_rad`

The orientation, in radians, at which an end stop is modelled. A negative endstop is at `-/*_limit_rad`. The values influence the scaling from pixels to orientation angles; see above (`/moving_camera_output`). You can freely change these parameters while running. Doing so gives an instantaneous jump in camera image and may give an instantaneous jump in `/position`. In particular, you can adjust the end stops so that they align with the physical field of view of your webcam.

E.5 Differences with a real Jiwy

This simulation is a very crude one; there are dozens of differences. The most important ones are:

- The physical Jiwy setup accepts PWM signals as inputs which encode the motor voltage. Normally, one would add a (software) closed-loop position controller to this which determines these signals based on setpoints \mathbf{x}_{set} (this is what you will develop in Assignment set 3). So, in fact, this simulator does not simulate the bare Jiwy setup, but rather the setup plus the closed-loop position controller.
- The field of view of the physical Jiwy is much larger than the field of view of the simulator:

	Simulator	Physical Jiwy	Unit
Pan	$-0.8 \dots 0.8$ (adjustable)	$\approx -3 \dots 3$	rad
Tilt	$-0.6 \dots 0.6$ (adjustable)	$\approx -1 \dots 1.5$	rad

- In the physical Jiwy, the camera image rotates (and, thus, distorts) while in the simulation the camera image translates over the input image.
- The speed of the dynamics is not the same; also the physical jiwy acts as a 4th order system (though dominant 2nd-order behavior); the simulation generates first order trajectories only.

E.6 Getting and compiling the node

E.6.1 Getting

You can download the code from Canvas. Unzip into the directory of your choice.

E.6.2 Compiling

Make sure that the ROS2 commands are sourced. Since this is a normal ROS2 package, the standard ROS2 compilation instructions apply:

```
cd <your_root_dir>/ros2nodes
colcon build
```

If this fails, you may have luck trying to build the `asdfr_interfaces` package first and sourcing it before building the `jiwy_simulator` package:

```
cd <your_root_dir>/ros2nodes
colcon build --packages-up-to asdfr_interfaces # Build only asdfr_interfaces
. install/local_setup.bash
colcon build # Build all
```


F Accessing Raspberry Pi's in the lab — For Assignment Sets 2 and 3

In Assignment Sets 2 and 3 you use a Raspberry Pi 4 (8 GB). These assignment sets are done in the Lab¹, RaM LabOne, Carré 3434.

F.1 Time slots

Each group gets 3×2 lecture hours access to a Raspberry Pi. The available time period (8 March 2023—23 March 2023) is divided into three time blocks of 4 or 5 working days; each group is allowed *one* time slot in each block. The blocks and time slots are available on Canvas Calendar.

For Assignment Set 2, you do not need to use the same Raspberry Pi all three time slots. The Raspberry Pis are identical and you should have a copy of your files on your local computer anyway, so switching to a different Raspberry Pi should not be a problem.

F.2 User name/account

Each group has his own account, being `asdf_r_XX` or `asdf_r_cbl_XX` (e.g., `asdf_r_03`) with a private password which is mailed to all group members. You can change the password if you want but be aware that it could be reset as well (see Section F.4). Each group has his own (encrypted) home directory on each Raspberry Pi. Note that the home directories are local on each Raspberry Pi, so when you log in on another Raspberry Pi you will not find your files there.

F.3 Connecting to the Raspberry Pi (SSH)

When working with the Raspberry Pi in the Lab of the RaM group, you can access the Raspberry Pis through the Virtual LAN of the Lab. You use your own laptop as the development PC. In order to connect to the Raspberry Pi, do the following:

1. Make sure that the Raspberry Pi is turned on and has a *green* ethernet cable²
2. Choose one of the following options:
 - With your own laptop, connect via WiFi to *RaM-lab*, password *robotics*. You are now connected to the virtual LAN. When on this network, you don't have connection to the internet.
 - Connect your own laptop to the RaM-lab VLAN with a *green* ethernet cable. This way your WiFi connection is still free to, simultaneously, connect to a WiFi access point providing internet (e.g., Eduroam or EnschedeStadVanNu).
3. Access the Raspberry Pi via SSH as described below.

Obviously, you are only allowed to log in onto the Raspberry Pi on your own lab table. Each Raspberry Pi has a label indicating its IP address, e.g., `10.0.15.53`). Logging in onto the Raspberry Pis goes through the SSH protocol; you need an SSH-client program on your own laptop to do that.

Note that you can have multiple SSH connections to the Raspberry Pi at the same time; e.g., one for running your program, another for running the load-generating program and a Visual Studio Code connection.

¹Preparation and post-processing are done at home though.

²The lab at RaM provides two separated networks: the UT net (red cables) and a private Virtual LAN called RaM-lab (green cables). The network needed is the latter.

F3.1 Logging in from Linux/macOS — Example

From a terminal:

```
ssh asdfr_03@10.0.15.53
```

and fill in your password.

F3.2 Logging in from Windows — Example

The best-known SSH client is `putty`. However, we recommend to use `MobaXterm` instead. Fill in the IP address and user name in the GUI.

F3.3 Logging in from Visual Studio Code — Example

Use the Remote-SSH Extension; use as the address ('Select configured SSH host or enter user@host'):

```
asdfr_03@10.0.15.53
```

See also Section D.5

F3.4 X-forwarding

This might be convenient for Assignment Set 3. Graphic output (e.g., of `showimage`) can be forwarded to your own laptop. This is called X-forwarding or X11-forwarding. Here is how to do it.

- *Linux*: just add `-X` to the SSH command, e.g.,

```
ssh -X rtsd_03@rtsd-rpi-4
```
- *OSX*: just add `-Y` to the SSH command, e.g.,

```
ssh -Y rtsd_03@rtsd-rpi-4
```
- *Windows/MobaXterm*: X-forwarding is enabled by default.
- *Windows/Putty*: When setting up the connection, enable X11-forwarding in the Connection|SSH|X11 tab as shown in Figure F.1:

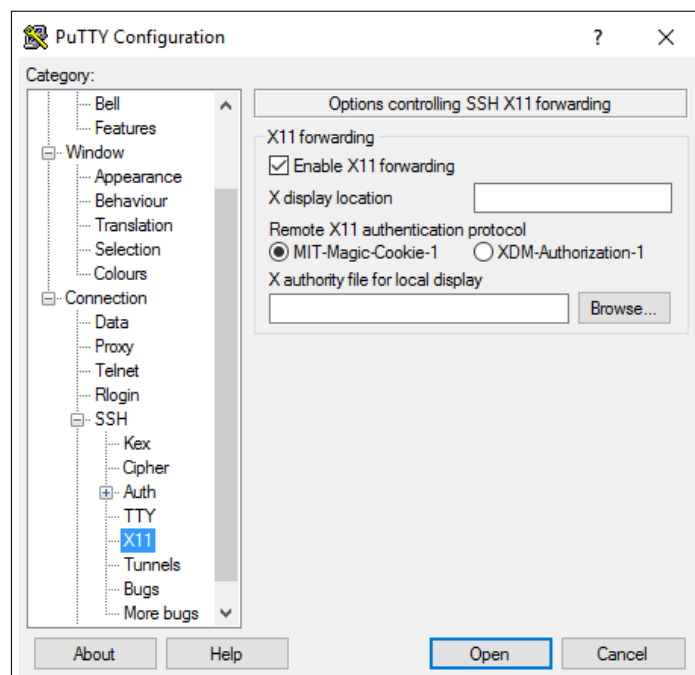


Figure F.1: Enabling X11-forwarding in Putty

You can test whether X-forwarding works by running a program with graphics output on the Raspberry Pi. When doing so, the window should pop up on your own laptop (possibly in the background). Simple programs that you can try are `xclock` and `xeyes` (or any other program from the `x11-apps` package).

F.4 File loss

Read this section carefully! If you do not comply with the information here, you could lose all your files!

In rare circumstances, the SD card of the Raspberry Pi can become corrupt. In this case we might decide to wipe the SD card and start with a fresh install. Then all user files on the Raspberry Pi are irreversibly lost. So, be aware that, in rare circumstances, your files on the Raspberry Pi could be lost.

In order to make it easier to manage your files, we highly recommend using `git` (be aware that ignored files are not pushed to the server though). This also makes it much easier (and less error-prone) to work on your own pc first and then on the Raspberry Pi later. `git` is installed on the Raspberry Pis.

F.5 General remarks

- Using your own laptop for editing the files remotely with Visual Studio Code is recommended; see Appendix D.5. Alternatively, you can use a text editor running on the Raspberry Pi (`nano` and `vim` are installed).
- No GUI/Windows manager is installed on the Raspberry Pis.
- To copy your files you can use the command line tool `scp` or use a GUI `sftp/scp` program like `winscp`. For Windows, `MobaXterm` has it built-in.
- A watchdog is installed on each Raspberry Pi. If you mess up your real time threads, starving all Linux processes, the Raspberry Pi will reboot automatically after 30 seconds. Alternatively you can power-cycle the Raspberry Pi.

G Compiling for Raspberry Pi / Xenomai

This appendix gives some hints on how to develop, run and debug code on the Raspberry Pi from within Visual Studio Code (VSCode). It continues on Appendix D, so read that first.

G.1 Compiling/debugging generic C/C++ code on the Raspberry Pi

Use the Remote-SSH extension to login onto the Raspberry Pi. Make sure you have installed the C/C++ Extension Pack, both locally and on the Raspberry Pi (see Section ??)

The following, standard, `tasks.json`, `launch.json` and `c_cpp_properties.json` should get you started. When using these, you can press F5 to build and debug the C-file you are currently editing.

G.1.1 `tasks.json` (Raspberry Pi)

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: gcc build active file",
      "command": "/usr/bin/gcc",
      "args": [
        "-fdiagnostics-color=always",
        "-g",
        "${file}",
        "-o",
        "${fileDirname}/${fileBasenameNoExtension}",
        "-lpthread", "-lrt"
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": "build",
      "detail": "compiler: /usr/bin/gcc"
    }
  ]
}
```

G.2 `launch.json` (Raspberry Pi)

```
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit:
  // https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "gcc - Build and debug active file",
      "type": "cppdbg",
      "request": "launch",
      "program": "${fileDirname}/${fileBasenameNoExtension}",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${fileDirname}",
    }
  ]
}
```

```

    "environment": [],
    "externalConsole": false,
    "MIMode": "gdb",
    "setupCommands": [
      {
        "description": "Enable pretty-printing for gdb",
        "text": "-enable-pretty-printing",
        "ignoreFailures": true
      }
    ],
    "preLaunchTask": "C/C++: gcc build active file",
    "miDebuggerPath": "/usr/bin/gdb"
  }
]
}

```

G.3 c_cpp_properties.json (Raspberry Pi)

```

{
  "configurations": [
    {
      "name": "Linux",
      "includePath": [
        "${workspaceFolder}/**"
      ],
      "defines": [],
      "compilerPath": "/usr/bin/gcc",
      "cStandard": "gnu99",
      "cppStandard": "c++14",
      "intelliSenseMode": "linux-gcc-arm"
    }
  ],
  "version": 4
}

```

G.4 Compiling for Xenomai on the Raspberry Pi

G.4.1 xeno-config

To compile a program for real-time running in Xenomai, you need to make the compiler link to real-time functions instead of the normal non-realtime variants. In order to do this easily, Xenomai provides a command `xeno-config`, which can output all required compiler and linker flag in an easy manner. The command is located in `/usr/xenomai/bin`. In the simplest case, where you want to compile and link a C-file (or multiple files) in one `gcc` command, the required flags can be obtained as follows (try it to have an idea what's going on):

```
/usr/xenomai/bin/xeno-config --skin=posix --cflags --ldflags
```

One can embed the output of `xeno-config` directly inside a new command with the bash command substitution feature¹, for example the compile command. With that, compiling a file, say `test.c` for Xenomai from the command line becomes as easy as

```
gcc test.c -o test $(/usr/xenomai/bin/xeno-config --skin=posix --cflags
↪ --ldflags)
```

This can also be used in `tasks.json`; it should just be given as one of the `"args"`. Unfortunately, arguments containing spaces are treated special; in order to avoid that, we need to separate the whole command into separate non-space-containing strings, so the arguments to

¹When using `$(command)` in a command, the `command` is executed and its output is substituted there

be added to the args are:

```
"$(/usr/xenomai/bin/xeno-config", "--skin=posix", "--cflags", "--ldflags) "
```

Also, the arguments `-lpthread` and `-lrt` can be removed since they are added by `xeno-config`. Finally, the argument `-g` ("compile only, do not link") must be removed because we want to compile and link in one command.

G.4.2 tasks.json (Raspberry Pi, Xenomai)

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "cppbuild",
      "label": "C/C++: gcc build active file",
      "command": "/usr/bin/gcc",
      "args": [
        "-fdiagnostics-color=always",
        "${file}",
        "-o",
        "${fileDirname}/${fileBasenameNoExtension}",
        "$(/usr/xenomai/bin/xeno-config", "--skin=posix", "--cflags", "--ldflags) "
      ],
      "options": {
        "cwd": "${fileDirname}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": "build",
      "detail": "compiler: /usr/bin/gcc"
    }
  ]
}
```

G.4.3 launch.json (Raspberry Pi, Xenomai)

Running a real-time program in Xenomai can only be done as root, i.e., you should run the program (say, `test`) as `sudo ./test`

This can be done from within VSCode, although it seems that using breakpoints does not work out-of-the-box. We added a small script, `/bin/gdb-sudo` the file system of the Raspberry Pi which simply invokes the debugger `gdb` as root:

```
#!/bin/bash
sudo /usr/bin/gdb "$@"
```

Now in `launch.json` the only thing to do is to change the `"miDebuggerPath"` from `"gdb"` to `"gdb-sudo"`.

H Jiwy hardware

Provided is a pan-tilt unit with webcam, called *Jiwy*, including motor drivers, see Figure 1.1. An *icoBoard* FPGA board, which is plugged directly on top of the Raspberry Pi (as so-called HAT) takes care of creating the PWM pulses for the motors as well as counting encoder pulses. It communicates with the Raspberry Pi via SPI (going through some of the pins of the 40-pins connector on the Raspberry Pi).

Connection between the hardware an FPGA is done via two short PMOD cables, one for each degree of freedom. A 12 V adapter provides power for Jiwy, see Figure H.1. The webcam is a normal USB webcam and is connected to the Raspberry Pi.

On the Raspberry Pi a C-library is available for setting the PWM value for each motor as well as reading the encoder count for each encoder. This function sends an SPI message to the FPGA, exchanging the necessary data. See Appendix I for details.

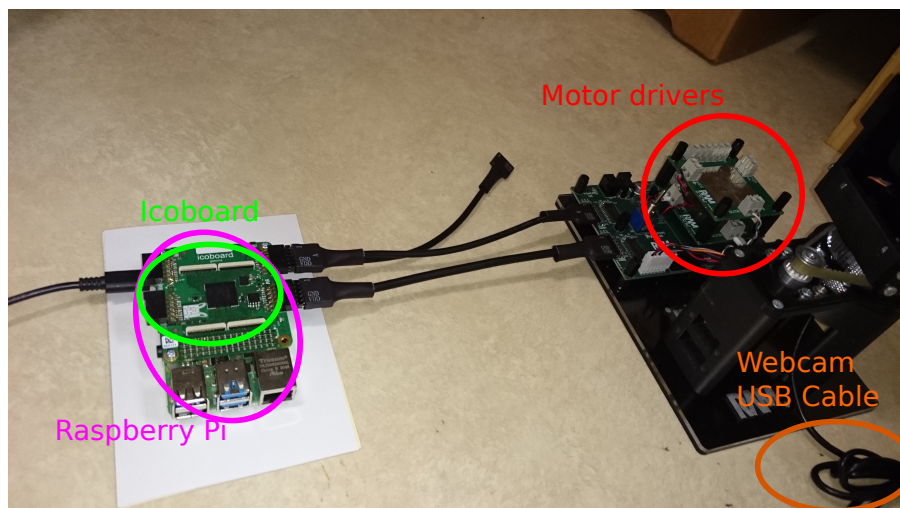


Figure H.1: The full Jiwy system, with on the left the Raspberry Pi.

An overview of the system architecture is shown in Figure H.2.

H.1 On/off switch

The PCB stack on the Jiwy setup contains a signal divider PCB (the bottom one) as well as two motor driver PCBs, one for each degree of freedom. On the signal divider PCB an on/off switch is present. It switches on and off the power supply to the motor drive PCB's, thereby enabling/disabling the motors. The rest of the circuit, e.g., sending PWM signals to the motor drive PCB's and reading encoder signals, is not influenced by the state of the switch.

The white ON and OFF labels close to the switch are wrong; when the switch is set to the ON position, the motors are off and vice versa. Check the LED close to the switch to see if the motors are enabled or not.

H.2 Interface

H.2.1 Motors

Each motor has its own motor driver PCB. It receives a PWM signal and direction¹ signal from the FPGA. From the Raspberry Pi, you send an integer value in the range:

¹Actually, two direction signals, PWM_ENA and PWM_ENB, where always PWM_ENA = !PWM_ENB.

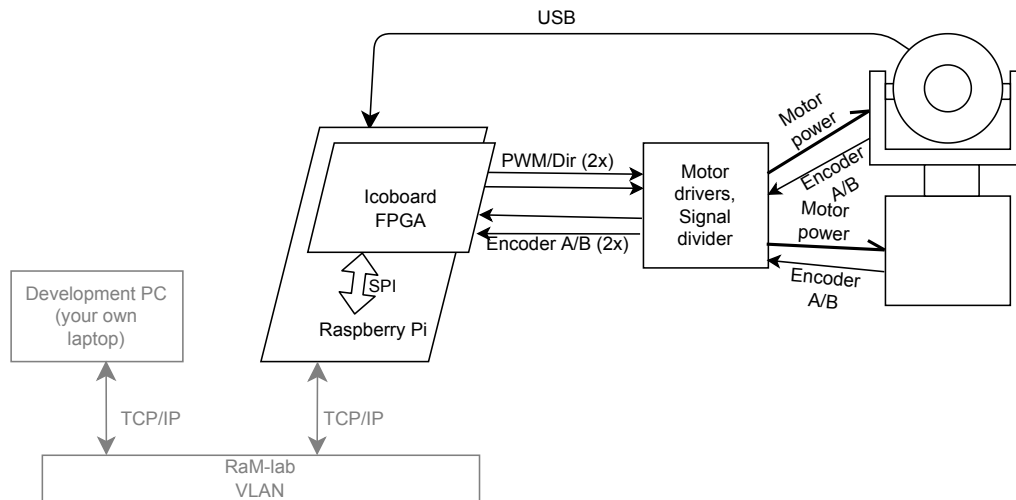


Figure H.2: Jiwy system architecture

- PWM value $[-2047 \dots 2047]$, maps to $-100\% \dots +100\%$ of full motor power. Values outside this range are clipped to -2047 or $+2047$.
- Pan = Yaw = left-right movement is connected to iCoBoard port P2.
- Tilt = Pitch = up-down movement is connected to iCoBoard port P1.
- The direction of rotation for positive/negative values is undefined, though identical for all setups. You need to try and, if needed, correct for it in software.

H.2.2 Encoders

Each degree of freedom has an incremental encoder. This encoder outputs A/B pulses (no zero/index pulse) when rotated. The corresponding signals on the PMOD connectors are called ENC_A and ENC_B. The FPGA counts the pulses and outputs this count to the Raspberry Pi (via SPI). Note that this count is *relative to the initial orientation*, i.e., the orientation the Jiwy was in when the Raspberry Pi was turned on. You can reset the encoder count by re-flashing (sort of re-booting) the FPGA firmware (do not forget to put the Jiwy in the desired zero-position by hand first):

```
icoprogram -b
```

The encoder pulses are counted by the FPGA, and sent to the Raspberry Pi via SPI:

- The value sent is encoded as a *14 bit unsigned int*, i.e., a value in the range $[0 \dots 16383]$. Note that this wraps around, e.g., if the encoder value is 0 and it decreases by one, the encoder value becomes 16383.
- The direction of counting is undefined, though identical for all set-ups. You need to try and, if needed, correct for it in software.
- See also Table H.1.

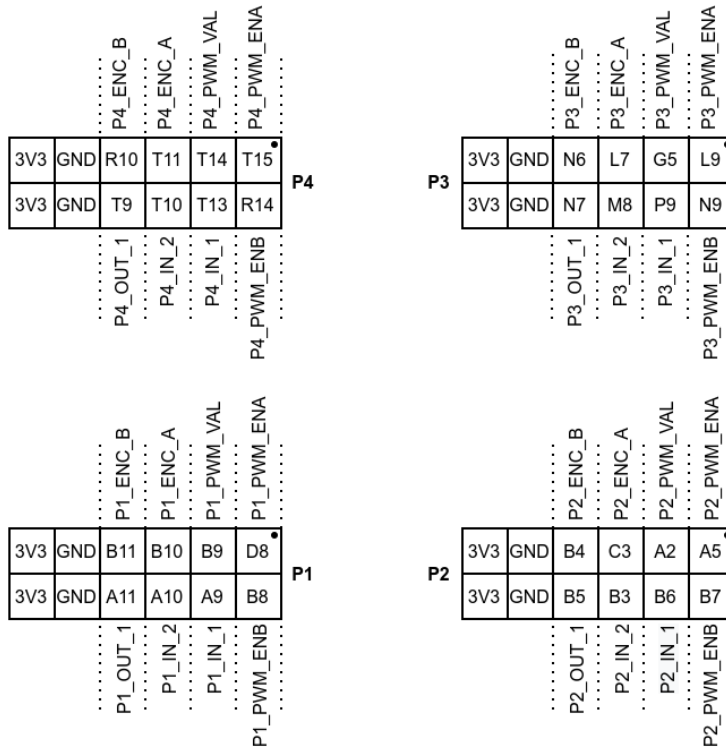
DOF	Encoder mount point	Counts per revolution	Gear reduction motor/output axis	Direction	Zero
Pan	Motor axis	1250	4:1	Undefined	Orientation at startup
Tilt	Output axis	2000	1:1	Undefined	Orientation at startup

Table H.1: Encoder details.

H.2.3 Extra digital inputs and outputs

Each PMOD connector also features two extra digital inputs `IN_1` and `IN_2` (which, for example, could be used for end-stop switches) as well as one extra digital output `OUT_1`. These inputs/outputs are not used in the Jiwy setup (the C-library does support them though).

H.3 Pinout of the PMOD connectors (on the FPGA)



The codes inside the connector squares (T15 etc.) are the internal ‘pin’ names in the FPGA; they are not relevant for using the system (only for programming the FPGA).

Note that, on the Jiwy side, the connectors are mirrored.

H.4 Software development for the Raspberry Pi

To develop software for the Raspberry Pi, you need to access it from a development PC, e.g., your own laptop (similarly to the way of work in Assignment Set 2). See Appendix ?? for details.

I The Xenomai-ROS2-20sim framework (Meijer)

This appendix extends the documentation from Bram Meijer's MSc. thesis (Meijer, 2021), Chapter 4. It gives some more details on how to implement the code and describes additions specifically for Jiwy control.

I.1 The IcoComm class

Specifically for interfacing with the *icoBoard* FPGA (Humenberger, 2008) an additional class for the xenomai-ros2-20sim framework was created. The class, called `IcoComm` descends from the `frameworkComm` (Meijer, 2021, Section 4.2.3) and act as a wrapper around the `ico_io` C-library. Basically, instead of sending/receiving the stream of bytes through the IDDP or XDDP communication channel, this class sends/receives a stream of bytes through the SPI port to the *icoBoard* FPGA.

The usage of this set of classes is almost identical to using the `IDDPComm` and `XDDPComm` class sets (which is logical because they all descend from the same parent class).

The differences are:

- `IDDPComm` / `XDDPComm` are unidirectional communication channels: they either send or receive data. Trying to use one class instance for both sending and receiving will immediately yield undefined behaviour.

`IcoComm` on the other hand is *always* bidirectional: sending and receiving data is done with only one instance. The reason behind this is that only one SPI port is used for the bidirectional data transfer; and one cannot open a port twice (also, sending and receiving is cleverly mixed in time to save communication time). Thus, instead of creating a separate instances of the class for the `controller_uPorts` and `controller_yPorts` (as done on page 24 of Meijer (2021)), you need to instantiate once and refer to it twice, e.g.:

```
icoComm = new IcoComm(...);
frameworkComm *controller_uPorts[] = {
    new IDDPCom(...),
    icoComm};
frameworkComm *controller_yPorts[] = {
    new IDDPComm(...),
    icoComm};
```

- The size of the communication packets is fixed, so, on construction, no `size` argument needs to be given
- The data type of the variables communicated through the SPI interface is integers and booleans. To keep compatibility with `FrameworkComm` however, they are typecast into an array of doubles (without scaling).
- The SPI communication packets have a fixed format and are translated from/into arrays of doubles as follows:

The data sent through the SPI interface (called the *output array*) is converted from an 8-element array of doubles having the following structure:

```
[ P1_PWM, P1_OUT_1, P2_PWM, P2_OUT_1,
  P3_PWM, P3_OUT_1, P4_PWM, P4_OUT_1 ]
```

The data received through the SPI interface (called the *input array*) is converted into a 12-element array of doubles having the following structure:

```
[ P1_ENC, P1_IN_1, P1_IN2, P2_ENC, P2_IN_1, P2_IN2,
  P3_ENC, P3_IN_1, P3_IN2, P4_ENC, P4_IN_1, P4_IN2 ]
```

- Just as with `IDDPComm` / `XDDPComm`, arrays of *Element Parameters* (Meijer, 2021, page 24) need to be given, specifying, for each element of the input and output array, to which element in `u[]` or `y[]` it goes. However, many of the provided elements in those arrays are not used for the Jiwy (e.g., only two of the four available encoders are read). In order to ignore (i.e., not use) an element from the input or output array, you can specify -1 as its associated element parameter.¹
- Since there is only one SPI port, it makes no sense to supply a port number. Hence, on construction, no `ownPort` / `destPort` need to be given.
- Immediately after reading the input array through SPI, the input array is fed through the function `ReadConvert()`, which can apply scaling/filtering to each of the elements of the input array.

Similarly, immediately before sending the output array through SPI, the output array is fed through the function `WriteConvert()`, which can apply scaling/filtering. See Section I.1.1.

The constructor of `IcoComm` has the following parameters:

```
IcoComm::IcoComm(int _sendParameters[], int _receiveParameters[])
```

I.1.1 ReadConvert() and WriteConvert()

These functions are provided to implement the Measurement&Actuation functionality. `ReadConvert()` does signal scaling and filtering for incoming signals (in the case of Jiwy: the encoders). `WriteConvert()` does signal scaling and filtering for outgoing signals (in the case of Jiwy: motor power). You provide the two as separate functions which you connect to the class instance with `SetReadConvertFcn()` and `SetWriteConvertFcn()`.

The function `ReadConvert` should be of the following type:

```
void ReadConvert(const double* src, double* dst)
```

Immediately after `IcoComm` has received the input array, the array is handed over to `ReadConvert()` as its first argument, `src`. This function should then fill the same-sized array `dst` with the scaled/filtered version (`IcoComm` has already allocated the memory). Note that it does make sense that no array size is given as argument; if you write this function you already know exactly what the array looks like (you even process each element individually). A `ReadConvert()` example is given below (comments in the code), together with how to connect it to the class instance. `WriteConvert()` works in the same fashion; no example is provided

```
void MyReadConvert(const double* src, double *dst)
{
    static double lastKnownGoodValue = 0;
    /* We assume that src is a 3 element double array, which has:
     * src[0] = temperature reading in degrees Fahrenheit (scaling)
     *          -> We want the output in degrees Celcius
     * src[1] = some measurement with unreliable communication; once
     *          in a while a (faulty) 0 is received; this should be
     *          filtered and the previous value should be passed (filtering)
     * src[2] = some good measurement which does not need
     *          scaling neither filtering
     */
}
```

¹To clarify: for each element in the input or output array, the associated element parameter specifies from/to which index of `u[]` or `y[]` the element should be copied. For example, if we specify the following element parameter array: `[•,4,•,-1,•,•,•,•]` as `_sendParameters`, `y[4]` will be copied to the second element of the output array, effectively binding `P1_OUT_1` to `y[4]`. The fourth element of the output array is not connected to any element of `y[]` (due to the value -1), so `P2_OUT_1` it is not bound to anything and will stay zero always.

```
dst[0] = (src[0]-32.0) / 1.8; // scaling

if ( src[1]!=0) // filtering
{
    dst[1] = src[1];
    lastKnownGoodValue = src[1];
} else
{
    dst[1] = lastKnownGoodValue;
}

dst[2] = src[2]; // passing
}

// Using the function (typically in some main/setup function):
IcoComm icoComm = new IcoComm(...);
icoComm.SetReadConvertFcn(&MyReadConvert);
```

I.2 Using the framework in your project

You can download a zip file with all necessary files (including all sources from the framework) from Canvas. When extracted you get a convenient workspace directory.

I.2.1 On your own computer

Unzipping and code development can be done on your own computer; though a full compilation is not possible due to the absence of Xenomai. Individual files can sometimes be compiled (but not linked), depending on their dependencies. This can save you some debugging work on the Raspberry Pi later.

I.2.2 On the Raspberry Pi

When working in the lab on a Raspberry Pi, you can copy/move the directory from your development computer to the Raspberry Pi (see Section E.5). Or, better, use git.

Compiling the project (which now consists of multiple files), cmake is used. A CMakeLists.txt file is provided which should be updated to include your own developed files (typically at least main.cpp). See the comments inside CMakeLists.txt.

ROS2 is installed and can be used right away; no need to source them. Also, the ROS2 nodes from Meijer (2021), talker and listener, both from the ros2-xenomai workspace, are installed and reachable by default (no need to source them).

J 20-sim and the 20-sim model

J.1 20-sim

20-sim (Controllab Products, 2008) is a modeling and simulation package by Controllab Products. It consists of a model editor, a simulator and much more.

J.1.1 Installation

Windows

1. Download 20-sim 4.8 here: <https://www.20sim.com/downloadnext/>
2. Double-click to install
3. Start 20-sim
4. In the License Activation window, select 'Activation'
5. Choose 'Single' / 'Only me'
6. Choose 'I received a license key by email'
7. Use as license key **MZavX-wZxkk-azLaV-vVGkZ-Dqy6v**.

Linux

20-sim is only available for Windows. On Linux (e.g., on the virtual machine from Assignment Set 1) you can use `wine` (or WineHQ):

- Install WineHQ. On Ubuntu 20.04 (e.g., the RTSD virtual machine):

```
sudo dpkg --add-architecture i386
wget -nc https://dl.winehq.org/wine-builds/winehq.key
sudo apt-key add winehq.key
sudo add-apt-repository 'deb https://dl.winehq.org/wine-builds/
↳ ubuntu/_groovy_main'
sudo apt update
sudo apt install -y --install-recommends winehq-stable
```

- Follow all steps for the Windows installation but instead of double clicking to install, run the executable under wine as follows:

```
wine 20-sim.exe
```

and, instead of double-clicking to start 20-sim run the executable (which is somewhere in a deep secret directory) under wine as follows:

```
wine ~/.wine/dosdevices/c\:/Program\ Files\ \ (x86\)/20-sim\ 4.8/bin
↳ /20sim.exe
```

J.1.2 Code generation

Code generation is a function found in the simulator (Model|Start Simulator), not in the editor (this is because the code generator needs already an checked-and-precompiled model and that's done when invoking the simulator). The code generation function can be found in the simulator under Tools|Real Time Toolbox. You need to generate a 'C++ class for 20-sim submodel'.

You can only generate code of individual submodels; if you want to generate a single C++ class from multiple submodels then you need to group the submodels first ('implode').

The default location where the generated files are saved is `c:\temp`. In Linux under Wine, this translates into `~/.wine/drive_c/temp`.

J.2 The 20-sim model

We provide a 20-sim model of Jiwy (the *plant*) and a position controller, see Figure J.1. The model can be downloaded from Canvas, on the Assignments page (Jiwy-with-controller.emx).

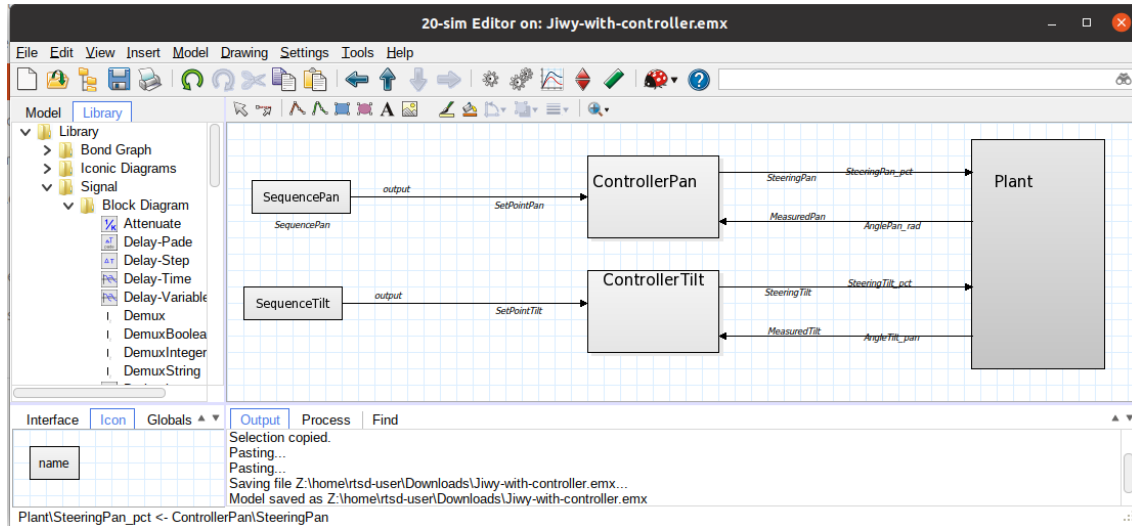


Figure J.1: The 20-sim editor with the Jiwy model under Linux.

In the model, the controllers for pan (x-movement) and tilt (y-movement) are separated. The contents of the ControllerPan and ControllerTilt blocks are identical and consist of a PID controller and appropriate signal subtraction, Figure J.2.

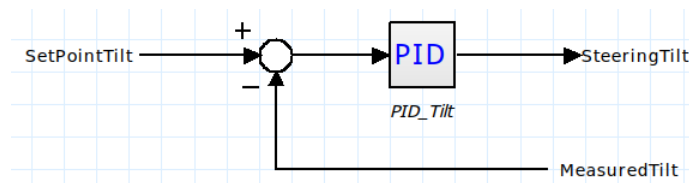


Figure J.2: The contents of the ControllerTilt block from Figure J.1.

The inputs of the controller block are the setpoints and measured joint angles in radians; the outputs are the desired motor powers in percent (i.e., a value in the range of $[-100 \dots 100]$). No provisions are made for keeping the steering values or measured joint angle within range (i.e., no safety layers).

The sequence blocks can be seen as test inputs for simulation of the full system.

K Handing in ROS2 projects

This appendix gives some hints on how to properly hand in ROS2 project. The essence is to properly package all required files (but not more) so that anyone (including the TA's) can unpack, compile and run it without hassle. In normal life this is an important aspect of software development, as software that you develop often needs to be run by others on other machines.

K.1 Preparing the hand-in

After completing each sub-assignment:

- Remove the `build`, `install` and `log` directories of your ROS2 workspace and do a complete rebuild of the project on your own PC. Solve any problems that arise.
- Make a zip file of the top-level `src` directory (see Section K.2.1).
- Create a small readme document (may be inside the zip file, but make sure it can be found easily) listing *everything* that needs to be done to compile and run the project on a different computer, see Section K.2.2. You may assume that ROS2 and SDL2 are correctly installed and that the `ros2` directory (not your own workspace) is source'd.
- Test the generated zip file and readme document on a different installation, for example your group-mate's laptop or a (fresh) virtual machine. It may help to choose a similar OS as test machine (i.e., test a Windows build on Windows and a Linux build on Linux). Even though everything should work on both platforms, you could get nasty configuration issues when mixing up Windows and Linux. It suffices if it runs on the similar platform; cross-platform compatibility is not required.
- Don't forget to adhere to the generic 'Report and File Requirements' as well (e.g., also hand in a PDF as a *separate* file).

K.2 Into more detail...

K.2.1 The ROS2 directory structure explained (and which files are required)

If you use the directory structure as suggested by the ROS2 tutorials, you probably have something like the structure below (depending on if you have split your work among different packages; file and deeper nested directories are not shown):

```

jiwy_ws
├── build
├── install
├── log
└── src
    ├── jiwy_msg
    │   └── msg
    ├── jiwy_pkg
    │   ├── include
    │   ├── launch
    │   └── src
    └── jiwy_pkg2
        ├── include
        └── src

```

Note that there are `src` directories at two different levels: one as a subdirectory of the `pong_ws` directory, and one inside each package directory. They serve different purposes:

- The top-level `src` means 'all files that are required to build the executables in `build` and `install`': all ROS2 configuration files such as `package.xml`, message type files (`.msg`),

images, icons and other resources, as well as the actual programming code (in C++ or Python). Note that below this `src` directory, multiple package directories may exist.

- The low-level `src` directories, inside each package, are used to store the actual C++ source code related to the specific package.

The directories `build`, `install` and `log` are all generated automatically when compiling the ROS2 project with `colcon`. You can remove them without regrets and they will be recreated from source when compiling (this sometimes solves untraceable compile problems as well, btw.). These directories are thus not required and should not be included in the package.

From the text above it should be clear that, if you have adhered to the suggested directory structure, you should package the full top-level `src` subdirectory; not more and not less.

You may sometimes find `build`, `install` and `log` directories inside a package directory. This happens when you accidentally are in the wrong directory when invoking `colcon build`; the directories are then created at the wrong place. You can safely remove these directories again.

K.2.2 Example `readme.txt` file

```
Steps to compile and run pong game
-----
For this example readme.txt, we take a pong game
that is controlled from the keyboard.
-----
1. Unzip in an empty directory <pong_ws>
2. Adjust SDL library path in src/pong_pkg/CMakeLists.txt
3. Go to <pong_ws>
4. colcon build
5. In new terminal:
   cd <pong_ws>
   install/local_setup.sh
   ros2 run pong_pkg pong_core
6. In another new terminal:
   cd <pong_ws>
   install/local_setup.sh
   ros2 run pong_pkg2 keyboard_input
7. Set focus on SDL2 window and control left bat
   with Q, A keys; control right bat with I, K keys

Alternative to steps 5 and 6:
   ros2 launch src/pong_pkg/launch/pong_keyboard.py
```


Bibliography

Controllab Products (2008), 20-sim.

<http://www.20-sim.com/>

Humenberger, E. (2008), icoBoard.

<http://icoboard.org/>

Meijer, A. (2021), Real-time robot software framework on Raspberry Pi using Xenomai and ROS2, MSc Thesis 070RaM2021, University of Twente, msc.

<https://cloud.ram.eemcs.utwente.nl/index.php/s/TPtt2yjixfN5DCc>

Student Charter (2020), Student Charter at University of Twente.

<https://www.utwente.nl/en/ces/sacc/regulations/charter>