

RELbot Software Tools and RELbot Simulation (v2024.3)

Jan Broenink, et al.

Thursday 27th March, 2025

Contents

1	Introduction	1
1.1	Origin of this documentVersion	1
1.2	Document History and Disclaimer	1
A	Change Log	2
B	Install C++ Integrated Development Environment	3
B.1	Introduction	3
B.2	VS Code as IDE and g++ as C++ compiler	3
B.3	Installing VS Code as IDE	4
B.4	Compiling (building) from the command line	5
B.5	Enable C++14 or newer	6
B.6	Compiler settings for C++14 or C++17 threads	7
B.7	The test case	7
B.8	Common issues	7
B.9	Configuration files in VS Code	8
B.10	Compiling / building and debugging using VSCode	8
C	Install for Remote Development	10
C.1	Introduction	10
C.2	Install virtual-machine managing software and a Linux virtual machine	11
C.3	Install ROS2 Jazzy	13
C.4	SSH key: Arrange the connection between a host and a VM or RELbot	14
C.5	Configure VS Code for remote development, on a VM or RELbot	15
C.6	Start developing C++ programs on a remote site	16
C.7	Install VS-Code extensions for ROS2	17
C.8	Arrange showing graphics from a VM or RELbot on the host machine	17
C.9	Installing ROS2 directly on Linux	18
D	Webcam stream to your VM	20
E	Accessing Raspberry Pi's in the Lab	21
E.1	User name/account	21
E.2	Connecting to the Raspberry Pi (SSH)	21
E.3	Avoid loosing files	21
E.4	General remarks	22
F	Structuring and Handling in ROS2 projects	23

F.1	Structuring ROS2 projects	23
F.2	Preparing a hand-in for ROS2 projects	24
F.3	How TAs grade your work	25
F.4	Example README.txt	26
G	RELbot simulator	27
G.1	"Installing" the package	27
G.2	Running	27
G.3	Parameters	27
G.4	Input topics	28
G.5	Output topics	28
G.6	From coordinates to image	29
G.7	Differences with a real RELbot	29
G.8	Troubleshooting the relbot simulator node	29
H	RELbot	30
H.1	Overview	30
H.2	Hardware parts	31
H.3	Firmware / Embedded Software	31
I	XRF2, the Xenomai 4 – ROS 2 framework	32
I.1	Structure of the Framework	32
I.2	Using the framework in your project	32
I.3	Background Information on XRF2	35
J	Using the XenoFrtLogger	38
K	Tips & Tricks and cheatsheets	41
K.1	Introduction	41
K.2	General terminal commands	41
K.3	VS Code	41
K.4	Compiler (g++ or clang)	41
K.5	WSL (Windows)	42
K.6	Multipass (Mac)	42
K.7	SSH	42
K.8	ROS 2	42
K.9	Colcon	43
K.10	Terminal applications useful for ROS	43
	References	45

1 Introduction

1.1 Origin of this documentVersion

This document contains instructions and tips for software development and testing of the resulting software for the RELbot – *Robotics Education Lab robot*. However, this document is also useful for developing software using VS Code or ROS2 for robotic / embedded control systems containing a Raspberry Pi computer.

Originally, these appendices are from course material of:

1. ASDfR – Advanced Software Development for Robotics
2. SDFR – Software Development for Robotics
3. Prog2 – Programming 2

This implies that this document is also fitted towards use for software development for these courses.

If you use this document for software development using VS Code or ROS2 or maybe not using a Raspberry Pi, some stuff might not be applicable for you. You might need to read ‘through’ these specifics to get the material applied for your own development work.

1.2 Document History and Disclaimer

When you find a mistake or have remarks about this document, please contact one of the contributors, most notably the first author.

As we also continuously improve this document, newer versions appear regularly, see its date and version number. Updates and changes are summarised in the Changelog, in Appendix A. The latest version of this document to be used in the mentioned courses is on the Canvas of the course and on the GIT server of RaM (in RoboticsEducationLab). Be sure to always use the latest version of this guide.

This document has seen quite some versions over the years, originated as appendices in the software development course manuals. Contributors to this document, next to the first author, are: Gijs van Oort, Luuk Lenders, Jelle Hierck, Jurriijn Pott, Ilyas Raoudi, Jasper Vinkenvleugel.

Development and documentation of the RELbot is by Gijs van Oort, Sander Smits, Marcel Schwirtz, Marjon Kuipers, Artur Gąsienica, and Jan Broenink.

A Change Log

- **version 2024.3**, 27 March 2025
 - Added appendix on XRF2 the Xenomai4 - ROS2 Framework.
 - Added appendix on the Logger of XRF2.
- **version 2024.2**, 13 March 2025
 - Updates in Appendices on accessing RPi in the Lab.
- **version 2024.1**, 3 February 2025
 - First version of the RELbot Software Tools and RELbot Simulation Manual.
 - **Note** Appendix E still to be checked / updated for the 2024 / 2025 version.

B Install C++ Integrated Development Environment

B.1 Introduction

The big advantage of IDEs is that, next to editing and look-up support, a lot of details on how to compile and link are hidden behind a graphical interface. The downside is that it may be hard to understand what happens behind the scenes of the graphical interface. Furthermore, relying on the IDE keeps the compilation commands implicit, taking more work when you have to share / submit the code including those compilation instructions. Therefore, it is good to know how things work under the hood, especially compiling and linking from the command line.

Note that for submitting assignments on software-development course work, you have to produce `README.txt` files in which you describe how to compile and run each program from the command line (including the exact commands that you use). This to be sure that those who grade your work can easily compile and run your code.

Next to instructions about installing the Integrated Development Environment (IDE) and C++ compilers, we provide pointers to more documentation and information on:

- Calling the compiler from the command line in Section B.4.
- Specifying the correct C++ language version in Section B.5.
- Specifying compiler settings for using threads in Section B.6.
- A test case to test your IDE plus C++ compiler setup in Section B.7.

About VS Code as the 'best' IDE and on installing it, is in Section B.2 and Section B.3 respectively.

B.2 VS Code as IDE and g++ as C++ compiler

Although many IDEs exist, we recommend using VS Code as IDE as that is one of the mainstream IDEs and allows the teaching assistants to give better support compared to using another IDE.

Furthermore, it is available for Windows, Mac, and Linux and has an enormous amount of available extensions. Both VS Code itself and its website provide documentation and help including tutorials.

Note that VS Code is *not* the same as Visual Studio. Visual Studio is another popular choice for IDE and is also made by the same developers as VS Code, but the TAs cannot support you with Visual Studio in this course.

The C++ compiler depends on the Operating System (OS) you have. It can actually be any compiler that is fully compliant with the C++14 language definition. Fortunately, all mainstream C++ compilers comply with C++ standards, and you can tune for which standard code is checked / compiled. The code you hand in must be portable C++ code (that is, it will compile and run on any of the OSes used, most notably, those of the TA who checks your work).

Local or 'remote' development

If you only need to create and run code on your computer / laptop "as is", *the native OS*, mostly called the *host OS*, your code creation and compilation workflow looks like the left leg in Figure B.1. This is, for instance, the situation in which you learn the tools and a programming language.

For robot-software programming or embedded-systems programming, the code you develop has to run on that robot / embedded-systems computer with often another processor architec-

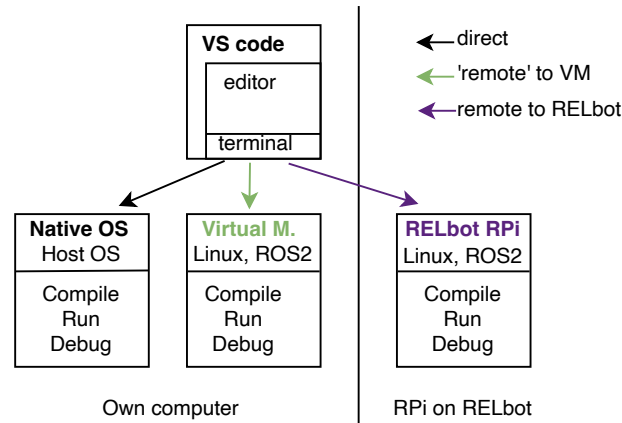


Figure B.1: Software development workflow: (left) local; (middle) 'remote' using a VM; (right) remote.

ture and another operating system than your development station (laptop). The code creation and compilation work flow is shown in the right leg in Figure B.1. This is called *remote development*.

As in general your workstation is more powerful than the embedded computer, you can install a *virtual machine* (VM) mimicking the processor and OS of that embedded computer (so without the specific peripherals and interfaces to the machine to be controlled), thus experiencing faster development compared to compiling on the embedded computer itself. Furthermore, you can develop software to run on the embedded computer, without having it at your disposal, so developing 'at home' while the embedded computer is 'at the lab', and thus easier available. In this form of remote development, the remote computer is on your own laptop. This workflow is shown in the middle leg in Figure B.1.

For programming on our Raspberry Pi or RELbot, you must do remote development. In that case, it is beneficial to start immediately using this remote development, using a virtual machine.

Advice for the courses

For the courses, we advice the following:

- *Programming 2*: local development.
- *Software Development for Robotics*: remote development. However, the first part can be done using local development, which can be beneficial in case you are rather new to this kind of software development.
- *Advanced Software Development for Robotics*: remote development.

For *Programming 2* and the first part of *Software Development for Robotics*, the local development approach as presented in this appendix will do, to start with installing VS Code, Section B.3, below here.

For remote development, to be used in *Advanced Software Development for Robotics* and *Software Development for Robotics*, for sure during the latter part, a few more things need to be arranged. These are explained in Appendix C.

B.3 Installing VS Code as IDE

VS Code is an editor which uses plugins ("Extensions") to connect with various compilers (not only C++; for almost every language extensions are available). VS Code runs on Windows, MacOS and Linux.

For elegantly installing VS Code as IDE and the C++ compiler, we can let VS Code do the work. Therefore, start with installing VS Code, add its C++ extensions, and install C++ compiler according to the instructions given in those extensions and using the information below.

Install VS Code

Download VS Code from <https://code.visualstudio.com/>, and follow installation instructions given there.

On Linux (Ubuntu) you might use `sudo snap install --classic code` to install VS Code from the command line.

Install VS Code Extensions for C++ and C++ compilers

Install the C++ compiler and VS Code extensions *C/C++*, *C/C++ Extension Pack*, and *C/C++ Themes*¹ via the documentation area of the VS Code website (<https://code.visualstudio.com/docs>), by choosing C++ in the overview on the left-hand side and then selecting:

- for Windows: [GCC on Windows](#).
- for Mac: [Clang on MacOS](#).
- for native Linux: [GCC on Linux](#).

We use the g++ compiler, and that is installed following the selections above. Do select the g++ compiler when compiling from within VS Code.

Follow the instructions about installing compilers up to and including "Create Hello World app". Next to that, we have a different test focussing on the topics of the course, which you can also run when reaching that section of this installation guide, (Section B.7).

Remarks

- The information in the description of the C/C++ extension is also on the installation page mentioned above.
- For MacOS, only Clang is treated, but g++ is installed in the same go using those instructions.
- When compiling using VS Code's interface (as the installation manual suggests), be aware to select the g++ compiler, so *not* the gcc compiler, which is often the first to select from in the drop-down box of the command palette.

The *C/C++ extension pack* contains popular and useful extras for developing C/C++ code.

The documentation part of the VS Code website contains a lot more documentation including videos. We have summarised some essential parts in Section B.9 and further sections.

B.4 Compiling (building) from the command line

Although the VS Code IDE supports building, that is compiling and linking, from its IDE (and thus hides it behind its graphical user interface; tuning it is in Section B.9), we present here compiling and linking from the command line.

Note that building is the complete process of generating an executable from source files. For most C++ projects, building is actually compiling and linking, and can be started from the command line by running the compiler.

¹Extension Pack and Themes are extra packages, and not mentioned in the documentation area of VS Code.

B.4.1 Getting a command prompt

The default Command Prompt (on Windows a.k.a. Terminal) or Terminal (on Mac), being a separate program, suffices. However, terminal programs like *PowerShell* or *Windows Terminal* (on Windows) or *iTerm* (on Mac) have quite some configuration possibilities, also for the remote development we are also doing, namely several terminal panes next to each other in one application window.

VS Code has a Terminal integrated in the *Panel*, the part below the editor region, which is already navigated to the directory of the project you are working on.

Use `cd` to navigate to your project directory.

B.4.2 Compiling and linking simultaneously

By default, the compiler does both compile and link in one command. Assume that we have two source files, `a.cpp` and `b.cpp`, and want to compile and link them into one executable, `C.exe` (for Windows) or `C` (for Linux/MacOS):

```
g++ a.cpp b.cpp -o C
```

Remarks

- In Windows the extension `.exe` is automatically generated.
- Mac, Linux executables do not need a specific extension.
- The compiler accepts wildcards as well, so if you want to compile-and-link all `.cpp` files in the directory, you can use `*.cpp`

B.4.3 Compiling only

You can limit the compiler/linker to only compile and not link. When doing this, the output is *not* a single executable, but one object file (extension `.o`) for each `.cpp` file compiled. This is done with the flag `-c`:

```
g++ -c a.cpp b.cpp
```

B.4.4 Linking only

By giving object files instead of source files to the compiler, the compilation process is skipped and the compiler only does linking. For example, after having generated `a.o` and `b.o`, you can create an executable from these with

```
g++ a.o b.o -o C
```

B.4.5 Run the generated executable

It is like any other program you run from the command line:

- *Windows*: `.\C`
- *Linux / Mac*: `./C`

B.5 Enable C++14 or newer

You have to make sure that the C++14 standard of the language is chosen, or a newer standard. C++14 is recommended because this is the standard used in Deitel and Deitel (2017).

Add on the command line `-std=c++14` to let the compiler use the C++14 standard. However, you can also safely use `-std=c++17`

To show the C++ version that is used by the compiler, you can query the long variable `__cplusplus` in a C++ program. See the nice example on <https://www.learncpp.com/cpp-tutorial/what-language-standard-is-my-compiler-using/>

B.6 Compiler settings for C++14 or C++17 threads

- *Windows*: There are no extra settings needed to use threads with Visual Studio.
- *MacOS*: There are no extra settings needed to use threads with the MacOS compiler (either g++ or clang).
- *Ubuntu, Linux*: On Linux, you probably need to add `-pthread -lpthread` to the compiler command line.

B.7 The test case

After installing and setting up your development environment, you should be able to compile and run the example code of Listing B.1 and see an output of “The answer is 42”, using the commands given above. Add arguments to specify the language standard and threads if needed.

Formatting is lost when copying the listing, but VS Code has an auto formatting function, which can be found using the command palette:

- *Linux/ Windows*: Ctrl + Shift + P
- *Mac*: Cmd + Shift + P

And then searching `Format Document`. Do note that the key-combination to activate this function is also shown (if set).

```
#include <iostream>
#include <thread>

volatile int globalVar = 1;

void threadFunction(int x) {
    globalVar = x;
}

int main() {
    // Start a new thread that calls "threadFunction"
    std::thread t(threadFunction, 42);

    // Waits for the thread to finish
    t.join();

    // print results
    std::cout << "The answer is "
    << globalVar
    << std::endl;

    return 0;
}
```

Listing B.1: Test code to see if the compiler is installed correctly

B.8 Common issues

B.8.1 Operating-System specific system calls

Sometimes you might want to use a system call, a sleep function for example. When you use these functions, you have to be careful that the method of calling the function might be different on a different operating system. The sleep function on Windows requires you to include the `<windows.h>` header. However, the sleep function on Linux requires you to include the `<unistd.h>` header. This means that you need to be careful when using these kinds of functions.

In this case, a simple solution to make sure that the sleep function works on both systems can look as shown in Listing B.2.

```
#ifdef _WIN32
    #include <Windows.h>
#else
    #include <unistd.h>
#endif
```

Listing B.2: Example sleep function

B.9 Configuration files in VS Code

This section is here for completeness and further tuning of VS Code for the building / compilation and linking process. You would not need it when simply compiling C++ projects from the command line, or using the ROS build support when compiling ROS nodes (programs).

Configuration of VSCode usually goes per project / (VS Code-)workspace. When you open a folder in VS Code, 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 colour 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 optimise 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.

B.10 Compiling / building and debugging using VSCode

This section is here for completeness and further tuning of VS Code for the building / compilation and linking process. You would not need it when simply compiling C++ projects from the command line, or using the ROS build support when compiling ROS nodes (programs).

B.10.1 Compiling / building

Building a project is started via Ctrl-Shift-B (Windows) / Shift-Command-b (MacOS), see also the *Terminal* menu. 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++). You might need to specify the language standard, for instance `"-std=gnu++14", .`

Note that the default building task *only* compiles the active file.

For a multiple-C++-file project, you need to customise 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 serves as 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 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 specialise 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, specify the directory where the include files are using the `-I` flag.

B.10.2 Debugging

Debugging a project is started by F5, see also the *Run* menu. Debug instructions for VS Code 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 customise `.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 customisation.

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.

B.10.3 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 do not 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 is going on under the hood, such that you can fix any inconsistencies that the extensions might introduce.

C Install for Remote Development

C.1 Introduction

The *remote development environment* presented in this Appendix is for development on a *virtual machine* and for real remote development, like on a RELbot or a Raspberry Pi. So it is about the middle and right leg in Figure C.1, being a recall of Figure B.1.

We do *not* go for a complete virtual machine like those in Virtual Box, UTM, or Parallels, as this approach takes more resources from your laptop. Furthermore, you can use your own tools running on your own laptop to do code editing, file handling etc., so *no* need to completely dig into the linux environment.

By first developing using ROS2 on a virtual machine on your own laptop and thereafter continuing the development on the RELbot, you use a *same* development environment. This makes testing and real execution more or less the same in both cases from the point of view of the development environment. It gives furthermore less dependency on having always a RELbot available.

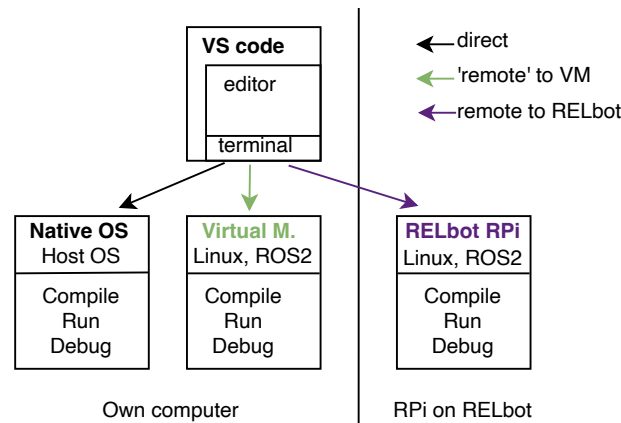


Figure C.1: Software development workflow: (right) local; (middle) ‘remote’ using a VM; (left) remote. This figure is a recall of Figure B.1.

Note that the installation work differs per host computer (Mac or Windows) and per target (Virtual Machine or RELbot / Raspberry Pi), so *not* all parts of this Appendix are applicable to all situations. However, quite some steps are the same, see Figure C.2.

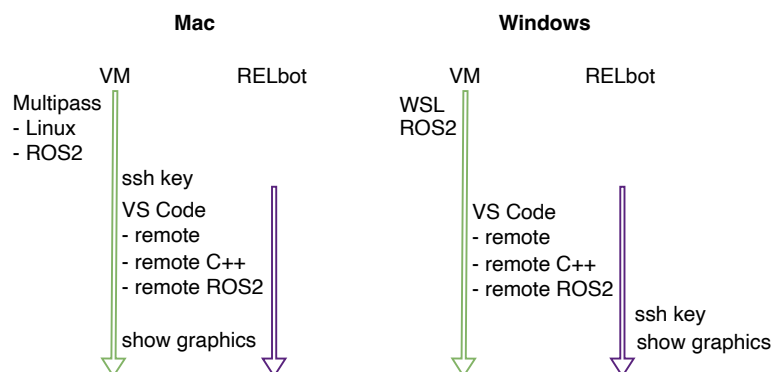


Figure C.2: Different sequences of installation steps. Steps on both sides of the install-flow arrow must be taken to install the complete set of tools for that particular host / target combination.

The installation work consists of the following steps:

- Install virtual-machine managing software (different for Mac and Windows).
- Install Linux, version Ubuntu 24.04 *Noble Numbat*, the newest "Long-Term Stable" version.
- Set up a mount point to reach your data files stored on your host OS.
- Install ROS2 Jazzy, the newest "Long-Term Stable" version.
- Arrange the connection between the host machine and a VM or RELbot.
- Arrange the connection between VS Code and a VM or RELbot.
- Setup remote C++ development on a VM or RELbot.
- Arrange showing graphics from the VM or RELbot on the host machine.

These steps are explained in the next sections. Of course, tests for each step are indicated.

More information, including some background, is in the VS Code Remote Development section of the VS Code documentation: <https://code.visualstudio.com/docs/remote/remote-overview>, although this documentation is sometimes a bit too much inclined towards Windows (VS Code is developed and owned by Microsoft). This appendix is not, obviously.

C.2 Install virtual-machine managing software and a Linux virtual machine

Mac and Windows need a different approach: On a Mac we use Multipass and for Windows, we use WSL (Windows Subsystem for Linux) as virtual-machine managing software.

C.2.1 Mac: Installing Multipass, Linux, and also ROS2

Multipass has a well-structured set of documentation in their documentation area, on their website, <https://multipass.run>, menu entry "Docs". For a quick start, some relevant commands are listed on the install page.

Download and install multipass from its website <https://multipass.run>. It must be at least version 1.14.1+mac.

Start Multipass.

Linux (Ubuntu) and ROS2 can be installed via two different ways, where in both cases tuning the Virtual Machine (*instance* in Multipass speak) via the Multipass GUI:

- Ubuntu via the Multipass GUI and ROS2 via command line.
- Ubuntu and ROS2 in one go via the command line (first start Multipass). This version takes a few GB more disk space.

Installing Ubuntu via the Multipass GUI, by selecting in the Catalogue *Ubuntu 24.04 LTS* and tune its parameters, see below.

Installing Ubuntu and ROS2 via the command line: `multipass launch ros2-jazzy --name jazzy`, and tune its parameters, see below.

Tuning parameters of the Multipass *instance*, (Multipass speak for a virtual machine running via Multipass):

- Name of VM: choose *jazzy*, the name of the ROS2 version we are using, or any other suitable name. Can only be specified when the instance is to be created (launched).
- CPUs: not more than half of the available cores, 4 will do anyway.
- Memory: not more than half of the available memory, 8 GB will do.
- Disk: 40 GB is recommended.
This 40 GB is not immediately taken, fractions are only claimed when needed.
- Bridged Network is not needed.

- **Mounts:** Map your host-OS home directory, usually `/Users/<login name>` on `/home/ubuntu/<login name>`. Be aware, directory and filenames are case sensitive.

We advice to use a mount point, such that your files can stay on the file system of your host OS, and are available in the VM. Using such a *mount* keeps your data (uh... programs) separate from the Ubuntu / ROS2 environment, so you would not loose them in case the VM gets corrupt. This way of working is often done in industry.

You can work in Ubuntu / ROS2 from the Multipass graphical user interface or directly from a MacOS Terminal (a.k.a. *command prompt* or *Shell*) using the command

```
Multipass shell jazzy
```

Note: the Shell from within the Multipass GUI might have some restrictions w.r.t. use of the clipboard. So, using a MacOS Terminal as Shell might be easier.

Start a *shell* (Terminal in Mac speak) from either within the Multipass user interface or from a MacOS Terminal and note the IPv4 IP address, as that is needed later to connect to this VM from the host OS. It is part of the internal local network between the host and the VM. This also shows that installation of Multipass and Ubuntu has succeeded.

C.2.2 Windows: Installing WSL and Linux

WSL, Windows Subsystem for Linux, version 2, supports quite some Linux commands and thus Linux programs.

To install WSL, we follow the guide on the website <https://learn.microsoft.com/en-us/windows/wsl/install>, but have defined some extra steps to make sure everybody is using the right version.

- Start a Command Prompt (or Terminal) and "Run as Administrator" by right clicking and selecting this option.
- Run the command `wsl --install --no-distribution`
If WSL was installed before, it is indicated, most probably as "The operation completed succesfully."
- To be sure to have the latest version (obviously skip this when you installed WSL in the previous step) by running `wsl --update`
- Run the command `wsl --set-default-version 2` in your terminal to ensure using WSL 2.
- Run `wsl --install -d Ubuntu-24.04` to install the right version of Ubuntu for this course.
- Choose a username and password (not always asked for). Choose as user `ubuntu` and as password `ubuntu` This Linux username has no relation with your Windows username.
- Reboot your laptop, as usual on Windows after these kind of installation work.

Some useful commands on handling Linux distributions on WSL are listed on the install page.

Note: for some very old Windows 10 versions, installing WSL take more steps, as explained on the install pages. Please follow those instructions if your system experiences this.

Some information about the VM is presented, including its IP addresses. Note the IPv4 address, as that is needed later to connect to this VM from the host OS when using graphics. It is part of the internal local network between the host and the VM.

Tips and best practices for a Linux VM running on WSL

- Starting the VM (Linux / Ubuntu using WSL) can be done via several ways:

- Starting a *Shell* (Command Prompt in Windows speak) on the VM by just starting Ubuntu 24.04 as a regular app on your windows machine. In that Shell window, navigate to the Linux home directory using the command `cd ~`
- Starting the VM from a Command Prompt or Power Shell using the command `wsl`. After that, navigate to the Linux home directory using the command `cd ~`
- Starting the VM, and navigating to the Linux home directory using the command `wsl ~`
- In Linux, your home directory is (`/home/ubuntu`), which is also where command `cd ~` in the ubuntu environment will bring you. It is recommended to put any files starting from this location. This includes your ROS2 files. Starting a VM in a shell puts you at the location you called the command from, so you must do an explicit navigation to the Linux home directory, with the indicated `cd` command. See the *File storage* Section on the *Best practices for setup* page of the *Tutorials* part for more info on file structure.
- Some status info including the hostname and IP address is shown only *once* per day. The IPv4 address can be queried via `wsl.exe hostname -I` inside the WSL environment. You can also request this info in a normal Windows shell by using `wsl hostname -I`
- File Explorer can be started from a VM (linux) Shell using `explorer.exe .`
Note the dot at the end of this command. This indicates to let the Explorer show the contents of the current directory. Tip: *Ubuntu-24.04* in the file path at the beginning or not, reveals whether a directory on the VM is shown or on 'normal' Windows.
- Windows programs can be started from within a Linux Shell by using the extension `.exe`
- When using either PowerShell or Command Prompt, turn on allow copy / paste via the Clipboard to Windows by enabling this feature in the settings. Reach these options via right click in the top bar of the shell and choose options. This might be called enable "ctrl+shift+C/V".
- You could use the program `wslsettings` to check and update settings of your virtual machines. However, default values are OK for our work.
- The Tutorials section of the online WSL documentation (root of it is <https://learn.microsoft.com/en-us/windows/wsl/>) contains the page *Best practices for set up*, <https://learn.microsoft.com/en-us/windows/wsl/setup/environment> with a few good tips concerning getting this VM running.

C.3 Install ROS2 Jazzy

Mac and Windows have the *same* installation process for ROS2 Jazzy, obviously, as ROS2 is installed in the (virtual) Linux environment. Even for a Linux machine the installation process of ROS2 Jazzy is the same.

Installation instructions are on the ROS2 site: <https://docs.ros.org/en/jazzy/Installation.html>.

You can obviously skip this part when you have installed Ubuntu and ROS2 in one go (MacOS only). However, you need to do the last step.

Do use the recommended approach for Ubuntu Linux ("deb packages"), as that installation is nicely integrated in the Ubuntu we have. Do install the optional development tools, so ROS2 package `ros-dev-tools`. You will be developing ROS2 nodes and packages.

Desktop installation gives the complete set of tools and packages you need, so install `ros-jazzy-desktop`. This installation includes demos, tutorials, and graphical tools. We do *not* use all graphical tools, as we use the graphics on the host OS, but this desktop version is really useful to have.

In case you have not that much disk space, you can opt for the ROS-Base install, and install material for demos and tutorials when needed (at the tutorials, it is indicated what to install).

Add the location of ROS2 to the settings file of the Ubuntu Shell (`.bashrc`), to avoid setting this every time you start a new Shell (Terminal), by typing

```
echo source /opt/ros/jazzy/setup.bash >> .bashrc
```

while being in your home directory, `/home/ubuntu`. Change directory to your ubuntu home directory is via the command `cd ~`

To test ROS2, run some examples, like the talker - listener demo (as last step on the install page), or the turtlesim demo (2nd tutorial, reachable via the *next step* on the install page).

Tips and Notes

- The command sets in the light green boxes can be copied to your terminal using the copy symbol in the top right, visible when you hover in that corner.
- Multi-line command sets starting with `sudo` might need some extra care, as the `enter` included at the end of a `sudo` command line can be interpreted as submitting an empty password, causing that (set of) commands go wrong.
- Most command sets can be run multiple times, except installation of `curl` in the box of *Now add the ROS 2 GPG key with apt*. Omitting the `-y` parameter on the first line, allows it do run more often.
- Mac: when installing Ubuntu and ROS2 in one go, you have to update the permissions of the `.bashrc` file first (seems a flaw) via `sudo chown ubuntu:ubuntu .bashrc`
- Windows: starting a Shell (Terminal) on the Linux VM puts you in the directory where you started this terminal. So, you have to first change directory to the home of the Linux filesystem (`/home/ubuntu`) via `cd ~`
- In case you installed the ROS-base version, you need to install the demo nodes before you can run the talker - listener demo, via the two commands:

```
sudo apt install ros-jazzy-demo-nodes-cpp
sudo apt install ros-jazzy-demo-nodes-py
```
- If you encounter the message *deferred to phasing*, packages / libraries are not immediately installed for reasons of quality / safety, as these parts have been reported on malfunctioning and are in the repair process by the Ubuntu distributors. As it is about updates and the current version is still there, the software still keeps running. So, you can safely ignore this message.

C.4 SSH key: Arrange the connection between a host and a VM or RELbot

On Windows using a VM via WSL, a secure connection is implicitly arranged, so *no extra* installation steps are needed for this situation (middle leg of Figure C.1). So, you can skip this Section.

On Mac using a VM via Multipass (middle leg of Figure C.1), a secure connection via an *ssh key* is needed, see below, Section C.4.1.

To connect to a RELbot / Raspberry Pi (right leg of Figure C.1 and green arrows in Figure C.2), for both Windows and Mac, a secure connection via an *ssh key* is needed, see below, Section C.4.1.

The connection between VS Code and the virtual environment respectively RELbot must be secure as must be for every remote connection. This is realised via a *Secure SHell*, *ssh*, taking care of the authentication when connecting to the remote side, see Section C.5.

C.4.1 Setup to secure connection

The secure connection using *ssh*, between VS Code and the VM from a Mac or between VS Code and a RELbot must be configured explicitly, before you can connect to that VM resp. RELbot. This holds also for a connection through VS Code using its extension *Remote - SSH*.

Establish the secure connection via ssh using key-based authentication, as supported by ssh, by putting the public-key of the private-public authentication key pair to the remote side. If you do not have such a private-public authentication key pair, you have to generate it.

Create a SSH key, if needed

Check if you have an SSH key on your *host* system by seeing if `~/ .ssh/<keyName>.pub` exists¹, where *<keyName>* is the name of the key as was generated, often (and in our case) `id_rsa`.

If there is no key present, generate one by running the following command in a terminal on the host (Mac or Windows):

```
ssh-keygen -t rsa
```

You can safely skip using a passphrase by hitting enter when asked.

This keygen program produces a public and private key in `~/ .ssh` and you can copy the public key to the VM (in case of a Mac) or a RELbot, as stated below.

Mac: Configuring a secure shell connection to the VM

Put the public part of the private-public authentication key on the remote side, VM (Mac only), by copying the public key from the host to the remote side, using the following command, using a shell (terminal) on the host: `multipass exec jazzy -- bash -c "echo `cat ~/ .ssh/<keyName>.pub` >> ~/ .ssh/authorized_keys"` *all on one line*, where *<keyName>* is the name of the public key file, and *jazzy* is the name of the Ubuntu VM (*instance* in Multipass speak). Note that copy-pasting from the pdf might not work, as the backquotes of the echo text appear as single quote, causing the command not to work.

Mac: Connect to a VM

Simply using the `ssh` command and typing the password will do.

Note that you can also start a shell from within the GUI of Multipass. However, showing graphical output is then not possible.

Mac and Windows: Connect to a RELbot

Simply using the `ssh` command and typing the password will do.

To prevent typing the password every time a connection is started, copy the public part of the authentication key to the RELbot in the file `.ssh/authorized_keys`:

- Mac: run the `ssh-copy-id` command instead of the normal `ssh` command from a shell (terminal) on the host while connecting to the Raspberry Pi.
- Windows: run the remote secure copy command: `scp .\<keyName>.pub pi@<RELbot IPv4 address>:./ .ssh/authorized|keys`

C.5 Configure VS Code for remote development, on a VM or RELbot

Mac and Windows are slightly different here. VS Code to the RELbot (its Raspberry Pi) is the same for both types of hosts.

For developing C++ (or code in another language) some specific software on the remote site is needed. This is automatically taken care of when starting to develop code on the remote site, see Section C.6.

¹~ is the shorthand for your home directory. On Mac or Linux it is usually `/Users/<loginName>` or `/home/<loginName>`. On Windows it is `<Drive Letter>:\Users\<loginName>`


For using ROS2, the ROS extension by Microsoft implements support for development for ROS2, see Section C.7.

Install remote-SSH, for both Mac and Windows

On Windows, VS Code needs the WSL extension. Simply install the extension when a request to do so pops up.

Install the *Remote - SSH* extension of VS Code for support for secure shell connections to remote (virtual) machines.

Connect to a remote machine within VS Code

To start the remote connection press the -icon at the lower left of the VS-Code window. Choose

- Mac / Linux VM: *Connect to Host...* and specify `<user name>@<remote IPv4 address>`. User name is `ubuntu` and the IPv4 address as shown when starting a shell on the VM.
- Windows: *Connect to WSL*. Nothing needed to add.
- RELbot: *Connect to Host...* and specify `<user name>@<remote IPv4 address>`. Username is provided for each group, see Appendix E, and the IPv4 address is shown on the display of the RELbot (labeled *eth0* for cabled access, and *wlan0* for WiFi access, both RaM-lab VPN only)

While connecting for the first time, VS Code automatically installs some 'server code' to the remote site to support this connection. It puts even more on it when starting with developing software, see the next subsection.

C.6 Start developing C++ programs on a remote site

While developing C++ code for the first time, VS Code asks to install the *C/C++ Extension Pack* extension. Of course install this extension.

Windows: Install build tools on WSL

Install C++ compiler on WSL in VS Code, via the documentation area of the VS Code website (<https://code.visualstudio.com/docs>), choosing C++ and then GCC on Windows Subsystem for Linux.

Follow the instructions about installing the compiler, debugger, VS Code extension and do the "Hello World" example. Explore a bit using the debugger and maybe intellisense.

Mac: Check your VM development system

The compilers are available on the Linux / Ubuntu VM, and can simply be used.

Start a remote connection to the VM via VS Code, and construct the "Hello World" project as you did while installing C++ compilers using VS Code on your host.

Some specific software on the remote site is automatically installed when starting to develop code on the remote site.

Notes and Tips

- Create this project in a *different* directory than the "Hello World" project on you host. You can safely store these project files on your host system, as these are available on your VM via the mount point point, as set up during configuring of the VM in Multipass.
- You can put your Ubuntu C++ files and projects in a directory tree next to the C++ files and projects that are compiled for your Mac.

Mac and Windows: Remote development on RELbot / Raspberry pi

The compilers are available on the RELbot / Raspberry Pi, and can simply be used.

Start a remote connection to the RELbot / Raspberry Pi via VS Code, and construct the "Hello World" project as you did while installing C++ compilers using VS Code on your host.

Some specific software on the RELbot / Raspberry Pi is automatically installed when starting to develop code over there.

Notes and Tips

- Create a directory (as usual) on the RELbot / Raspberry Pi to put your project in. Putting files on the RELbot / Raspberry Pi is needed as the programs you create must run over there.
- Do copy the files to your host after finishing your development. If you are familiar with file synchronisation tools or code development tools (like Git) you could use these to automate this file synchronisation / copying process.

C.7 Install VS-Code extensions for ROS2

Install the ROS extension by Microsoft ("Develop Robot Operating System (ROS) with Visual Studio Code").

A key activity using the ROS extension of VS Code is to streamline developing and debugging of ROS2 nodes. Simply connecting to a remote ROS2 environment can be done by using a ssh connection, either from a separate Terminal / Shell, or from the Terminal window in VS Code

See the documentation of this extension about its capabilities.

Note that the ROS extension of VS Code is normally installed at the remote site (VM or RELbot / Raspberry Pi) where ROS2 is installed. VS Code shows only extensions of the remote site when connected to that remote site.

Test the very basics by running some ROS2 nodes, like the talker - listener example used for installing ROS2 (see Section C.3). Start these nodes in two different Terminal windows from within VS Code (use + in menu bar at the right-hand side of the Terminal pane).

See tips for attaching to a node etc via this extension, the documentation of this extension, reachable via the Extensions list in the Primary Side Bar (at the left-hand side of the VS Code window).

C.8 Arrange showing graphics from a VM or RELbot on the host machine

Showing graphics on the host is different for Windows and Mac.

Using the VM on Windows via WSL does not need any extras for this.

C.8.1 Mac: Showing graphics from a VM on host

For showing graphics (time plots of signals, for instance), we use an X server running on the host system. Such an X server is *not* part of MacOS, the standard operating system anymore. Fortunately, Apple has an X server available, namely Xquartz.

So, for Mac, simply install Xquartz from xquartz.org

To run a command on the VM that produces graphical output on the host, you need to start the SSH connection using the `-X` parameter, so:

```
ssh -X ubuntu@<IP address VM or RELbot>
```

and start the program on the VM or RELbot that produces graphical output via ssh on the host.

You can test this X server connection over ssh, by running the turtlesim simulator in ROS2, see the ROS2 Jazzy tutorial “Using turtlesim, ROS2, and RQt”, available on <https://docs.ros.org/en/jazzy/Tutorials.html>. It is the second tutorial in the “Beginner: CLI tools” category.

C.8.2 Windows: Showing graphics from a RELbot – Preparation

We recommend X server from <https://sourceforge.net/projects/vcxsrv/>. After you have installed X server, run the following installation steps:

1. Start XLaunch, you can search for it in the start menu.
2. You can go with the default settings, but you *must* enable the `Disable access control`. In case your firewall gives a warning, do allow vcXserv to communicate over networks (default value will do), of course.

Continue with the steps in the next subsection. These are the same for Windows and Mac, and run on the RELbot

C.8.3 Windows and Mac: Prepare the RELbot to show graphics to your host

The steps below are the same for Windows and Mac, as these steps run on the RELbot. Windows users need to take a few extra steps to get -X to an external device working. Other use

Windows preparation

1. First, check if you have a environment variable named `DISPLAY` set correctly. This can be checked by the following command, different per Terminal-application:
 - Powershell: write `$env:DISPLAY`, this should return `:0.0`
 - Command Prompt: write `echo %DISPLAY%`, this should return `:0.0`
2. If this is not set, we need to set this environment variable. Two options exist:
 - *Long-term*: Search for `environment variables` using the start menu search function. This should open a window named `System Properties`, at the bottom there is a button named `Environment Variables`. Click that, which should open a new window. In this window, under the subsection `System variables` click `New`. Then fill in under variable name `DISPLAY` (all caps), and variable value `:0.0` (including the colon). Then press the apply buttons to exit the windows.
 - *Short-term* (only for the current terminal): Write `set DISPLAY=:0.0` to set the variable for this terminal window only. Long term option is recommended for the duration of the course.
3. Check again if the `DISPLAY` variable is set by running the appropriate command mentioned above.

Connection steps

1. Connect with the RELbot via ssh:
 - Windows: `wsl ssh -X <username>@<IPv4 address RELbot>`
 - Mac / Linux: `ssh -X <username>@<IPv4 address RELbot>`
2. Validate the working of XLaunch by running turtlesim via `ros2 run turtlesim turtlesim_node`

Note: This -X argument / facility is *not* supported in shells started from within the graphical user interface of Multipass.

C.9 Installing ROS2 directly on Linux

Installation instructions are presented in Section C.3.

For Ubuntu users, there is a convenient script that does the full installation unattended, see https://github.com/Tiryoh/ros2_setup_scripts_ubuntu.

Use it as follows:

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

D Webcam stream to your VM

Using a VM means that direct access to your webcam is made more difficult. The `cam2image` package makes this easily possible.

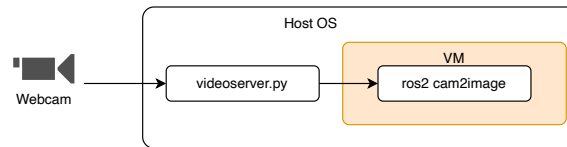


Figure D.1: High-level overview of how the webcam stream is passed to the VM for further use there.

Prerequisites

- Have a recent version of Python 3 installed.
 - If you use python for other projects, a `venv` (virtual environment) is an option. See [documentation](#) for instructions to set up the `venv` functionality.
- Have your host-side IPv4 address of your VM connection at hand (router address of the internal local network between host and VM).
 - **Mac:** Find this by running `ifconfig | grep 192` and find number after `inet`.
 - **Windows:** You can find this by running `ipconfig` and search for the entry with WSL in the line. You can find your IP address under IPv4 address.
- **For Mac:** Increase the UDP data-frame size to the maximum by running `sudo sysctl -w net.inet.udp.maxdgram=65535`. Running this has no adverse effects on your machine. To reset, run the same command with `9216` as `maxdgram`.

Host machine

1. Place `cam2image_host2vm` anywhere on the filesystem of your host computer.
2. Open a terminal in this folder, make sure all required python packages are installed by running `pip install -r requirements.txt`
3. Run the `videosever.py` by calling `python videosever.py`. Some systems might use `python3` as command. Command autocomplete with tab should help.

VM

1. Place `cam2image_vm2ros` in the `src` folder of your ROS workspace.
2. In a terminal, run `colcon build` in the base of your ROS workspace.
 - See also Section E2 for layout.
3. Set up the environment by running `source install/setup.bash`
4. Edit the `config/cam2image.yaml` file in the `cam2image_vm2ros` package, change the `socket_ip` line to your host IP.
5. Run the package by running `ros2 run cam2image_vm2ros cam2image --ros-args --params-file src/cam2image_vm2ros/config/cam2image.yaml`

Note: the `videosever.py` script should be running before starting the node on the VM!

Debugging parameters

- `show_camera` will show the camera image over an `ssh -X` connection. WSL automatically does this, but Multipass and the Raspberry Pi need the `-X` argument!
- `remote_timeout` sets the amount of seconds the node waits before timing out. Default = 7

E Accessing Raspberry Pi's in the Lab

In the assignments where you must use a Raspberry Pi 4 (8 GB), either stand-alone or as main computer of the RELbot, you must use a Raspberry Pi 4, of which we have several available in our Lab (RaM LabOne, Carré 3434), because of specifically prepared software on it¹.

E.1 User name/account

Each group has his own account, being `sdf-r-xx`, `sdf-r-cbl-xx`, or `asdf-r-xx` (e.g., `asdf-r-03`) with a private password which is mailed or given to the group members. You can change the password if you want but be aware that it could be reset as well (see Section E.3). 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.

E.2 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.
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 do *not* have connection to the internet.
 - Connect your own laptop to the RaM-lab VLAN with a *green* ethernet cable². 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 shows on its display its IP address, e.g., `10.0.28.109`. 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.

E.2.1 Logging in

From a terminal: `ssh <user name>@<IP>`, for instance, `ssh asdf-r_03@10.0.28.109`.

For information on X-forwarding, see Section C.8. In general, if you have set up X-forwarding for Multipass and an X client on Windows and macOS, you can use the flag `-X` in the aforementioned command to forward windows from the Raspberry Pi.

E.3 Avoid loosing files

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

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.

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 such case, your files on the Raspberry Pi get lost.

In order to make it easier to manage your files, we 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.

E.4 General remarks

- Using your own laptop for editing the files remotely with Visual Studio Code is recommended; see Appendix C, especially Section C.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. Or `git` as synchronisation tool as mentioned above.

F Structuring and Handing in ROS2 projects

This appendix provides rules on how to properly structure and hand in ROS2 projects. Doing so, your work is consistent with common approaches for ROS development, and is easier understandable by others. Furthermore, it helps the TAs more effectively reproduce your nodes and grade your work.

F.1 Structuring ROS2 projects

F.1.1 ROS Workspace

The *standard* structure of files (for instance, as a result of doing ROS2 tutorials) looks like Figure F.1.

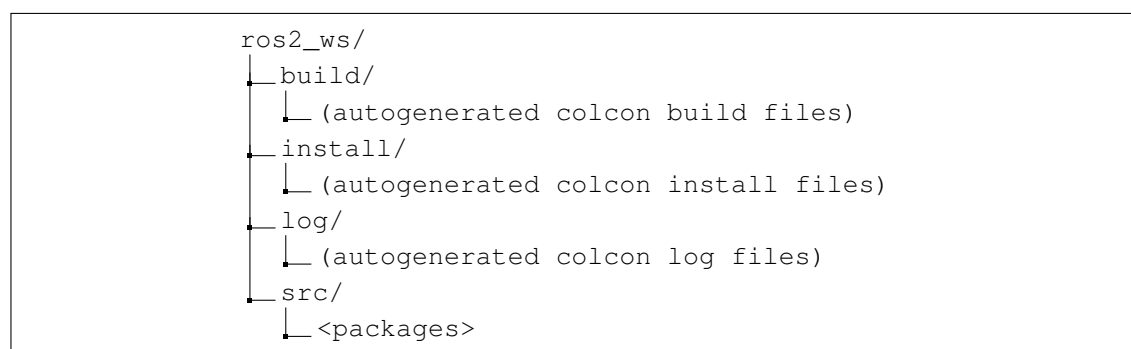


Figure F.1: Project contents

The `ros2_ws` folder is called the *base* of your ROS workspace. It is the real starting point of all the work in your ROS workspace:

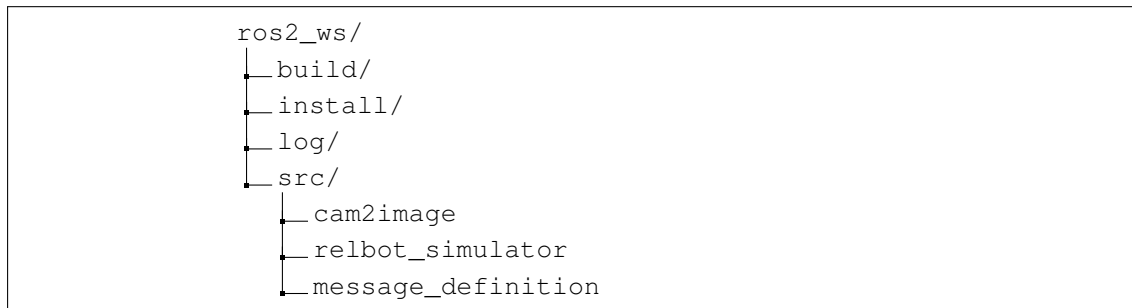
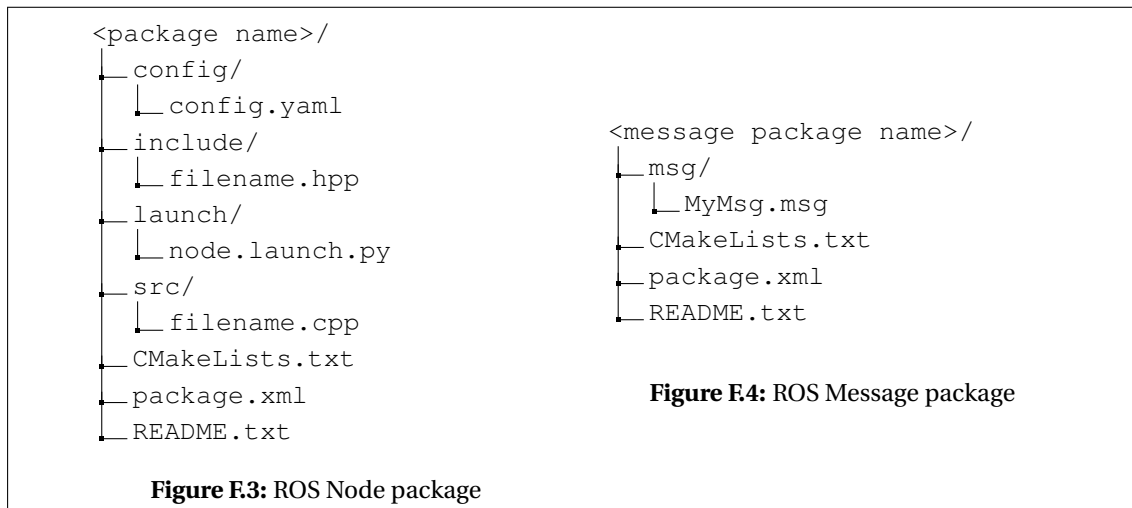
- To start the building / compilation process via the command `colcon build`
- To configure the environment using `source install/setup.bash`
- To put all work (source files, configuration files) in `ros2_ws/src/`

For this reason, we do *not* discuss the content of the *autogenerated* `build/`, `install/`, and `log/` directories.

F.1.2 Packages

During development, code is organised in packages. These contain one or more ROS nodes, or message definitions. These aim to separate different components, such as the `cam2image` and `relbot_simulator` packages described in Appendix D and Appendix G respectively. Messages are also kept separate, such that they can compile separately from the package that needs them. Putting this all together, should look like Figure F.2.

Within these folders, code files are separated by function such as source files in the `src/` and header files in the `include/` folders. The same is done for supporting ROS files such as `config/` and `launch/` files. This structure is also used for messages, to keep consistency between all source files. Both have `package.xml`, `CMakeLists.txt`, and a `README.txt`. The `README.txt` should describe the contents of the package on its own. See Section F.2.2 for minimal requirements when handing in.

**Figure E2:** Project contents**Figure E3:** ROS Node package**Figure E4:** ROS Message package

E1.3 Troubleshooting

Sometimes, errors keep existing even though the code was fixed. This might be due to issues persisting in the auto-generated folders. It can sometimes help to remove these, by running `rm -r build/ install/ log/` in the base of the ROS workspace.

E2 Preparing a hand-in for ROS2 projects

When handing in your assignments, your code should work on the PC of the TA. To make this easier, we have set some rules.

E2.1 Hand in contents

You must submit *one* .zip file containing everything to run your code independently. The automatically generated `build/`, `install/`, and `log/` folders should **NOT** be included. Your submission should look like Figure E5.

Code provided by us *must* be excluded and *must* be described as a requirement in the top-level `README.txt`

E2.2 README files

In the final assignment, you need two types of README files:

- Package README: every package is self-contained. It must include a README file describing that package, and how to use it. See an example in Section E4.2.
- Top-level README: Should explain how to run your packages, and for sub-assignment they relate to. See an example in Section E4.1.

```

assignment.zip
├── my_custom_package/
│   └── README.txt (package level)
├── my_other_custom_package/
│   └── README.txt (package level)
├── my_custom_messages/
│   └── README.txt (package level)
└── README.txt (top-level)

```

Figure F.5: Project contents

For the README-file text, the obvious first steps like unzipping, placing it in the standard directory structure (so like in Figure E.2), and compiling / building using the `colcon build` command do *not* need to be written, as is also omitted in the two example README files. So, steps 1 – 3 in Section F.3.

F.2.3 Submission instructions

Before each submission, make sure of the following

- Make sure your package works in a clean environment. This can be done by removing the `build/`, `install/` and `log/` folders, rebuild it, and remove those folders again.
- Make sure that each package includes a *package-level* `README.txt` file.
- Make sure that the *top-level* `README.txt` file describes how to run each sub-assignment. Assume ROS is installed.
- Test the generated zip file and various `README.txt` files on a different installation, such as your group-mates laptop.

Assuming everything above passes, and you followed the structuring guidelines, select the top-level README, all packages related to the assignment in the `ros2_ws/src/` folder and zip them.

F.3 How TAs grade your work

This section describes how the TAs will grade your work.

1. TAs download your submitted zip file from Canvas and unzip it.
2. The packages will be copied into their own ROS workspace, similar to Figure E.2. None of the packages we supply will be copied, as these are already present (the original ones).
3. The TA runs `colcon build` and `source install/setup.bash` from the base of their ROS workspace.
 - Any issues during this building and configuring process can cause reduction in your grade. TAs do *not* debug your work. Top-level README should explain any further steps if necessary. TAs have ROS2 Jazzy desktop and our provided packages installed.
4. TAs grade your work.
 - The design document (submitted separately as PDF) must contain your design decisions and all answers to questions in the assignment. Your code must match this.
 - Top-level README must contain instructions for how to run all code components. It should also describe where something is implemented so that the TAs can easily find it in your code. See also Section F.4.1. Any command will be run from the base of the workspace.
5. Based on answers and rubrics you are graded. Rubrics are available on Canvas.

F4 Example README.txt

F4.1 Top-level README file example (src/README.txt)

```

Assignment 25.1
-----
1. Run
   ros2 run my_custom_package node1
   Location in code: lihgt_detection() function in image_analysis.cpp

Assignment 25.2
-----
1. Open two new terminals, in the first run
   ros2 run my_custom_package node2
   Location in code: coluor_detection() function in image_analysis.cpp

   In the second run
   ros2 run my_other_custom_package node1
   Location in code: steer_caluclation() function in steer()

Assignment 25.3
-----
1. Run
   ros2 launch my_custom_package start_all_nodes.launch.py

   and send a velocity
   ros2 topic pub /my_topic example_msgs/msg/Float64 "{data: 2.0}"

   Assignment connects node 1, 2 and node from other-package together.

```

F4.2 Package README file example (src/relbot_simulator/README.txt)

```

Package relbot_simulator
-----
Description: This package implements a simulated relbot. Model generated by 20-sim

Inputs
/image
    Type: sensor_msgs/msg/Image
/input/left_motor/setpoint_vel
/input/right_motor/setpoint_vel
    Type: example_interfaces/msg/Float64

Outputs
/output/camera_position
    Type: geometry_msgs/msg/PointStamped
/output/moving_camera
    Type: sensor_msgs/msg/Image
/output/robot_pose
    Type: geometry_msgs/msg/PoseStamped

Run
    In a terminal run either of the following commands:
    ros2 run relbot_simulator relbot_simulator
    ros2 launch relbot_simulator relbot_simulator.launch.py

Parameters
    double image_stream_FPS : Sets the output rate of image stream. Default = 30
    FPS, which is most webcams

Core components
    dynamics_timer_callback(): Calls and runs the model every timestep
    image__stream_timer_callback(): Ingests and passes on webcam image stream
    CreateCVSubimage(): Creates sub-image which is then published on
    /output/moving_camera

```

G RELbot simulator

The `relbot_simulator` package provides a modelled RELbot as a ROS node for software evaluation. This appendix describes the use of this node.

G.1 "Installing" the package

For first use, you will need to first download the package from Canvas.

1. Place the folder named `relbot_simulator` in your ROS workspace (should look like: `ros2_ws/src/relbot_sim`).
2. `colcon build` from the base of the workspace.
 - Depending on the install you did in Section C.3, you might need to run `sudo apt install ros-jazzy-cv-bridge`. This will be clear if you get an error.
3. Configure the environment by running `source install/setup.bash`

G.2 Running

G.2.1 Running the node

To get the simulator node running, use `ros2 run relbot_simulator relbot_simulator`. This will run the package, but lacks any form of input. Figure G.1 shows the available inputs and outputs of the node while running. These are discussed in the next sections.

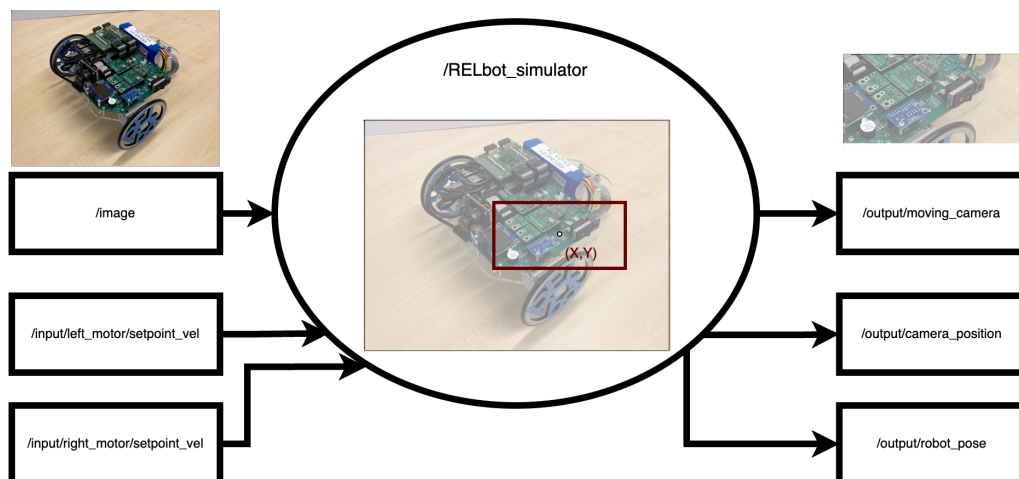


Figure G.1: Schematic drawing of the functionality of the RELbot Simulator node.

G.3 Parameters

This package has two parameters which can be used to change image output rate, and if the simulator accepts a different command. Parameters can be added to the normal launch command by writing `--ros-args -p <parameter name>:=<value>`

- `code` : Controls how often the image sub-functionality is run. Default = 30 FPS.
- `use_twist_cmd`: Controls if you use wheel velocity set-points or a twist to control the robot. Default = False. Turning this on also changes the active input topics.

G.4 Input topics

G.4.1 `/input/left_motor/setpoint_vel` and `/input/right_motor/setpoint_vel`

. These topics are used to set the desired velocity of the wheels in radians/s. Mutually exclusive with the twist topic.

- The message type is `example_interfaces/msg/Float64`.
- Commands need to be sent to both wheels separately.
- You can give a velocity set-point to the node from the command line with the following ROS2 command: `ros2 topic pub input/left_motor/setpoint_vel example_interfaces/msg/Float64 "{data : -0.5}"`

Note: you *must* place a whitespace after the :

G.4.2 `/image`

An image, can be used in combination with the `cam2image` package (see Appendix D) to stream a webcam.

- Message type is `sensor_msgs/msg/Image`

G.4.3 `/input/twist`

If you are unaware what a twist is, or how to use it, we recommended using the wheel velocities instead.

When setting the parameter `use_twist_cmd:=true` when starting using launch or run commands, you can turn on commanding the system with a singular Twist, rather than two wheel-velocities. Mutually exclusive with wheel input topics

- Message type is `geometry_msgs/msg/Twist`

G.5 Output topics

G.5.1 `/output/camera_position`

Center of the sub-image camera position in pixels.

- Message type is `geometry_msgs/msg/PointStamped`
- Get published after an input image has been processed.
- Only X and Y components are used.

G.5.2 `/output/moving_camera`

The simulated output image of the simulated camera.

- Message type is `sensor_msgs/msg/Image`
- This topic is published to whenever a new message from `/image` arrives.
- It is a part of the original input image, scaled and rotated as needed
- Default dimensions are `0.5*height of original /image topic at initialisation`.
- The centre of the output image is looking at `/output/cam_position`.

G.5.3 `/output/robot_pose`

Location of the robot in 3-dimensional space.

- Message type is `geometry_msgs/msg/PoseStamped`
- Only X, Y, and Rotation_Z are used. These describe X, Y, θ_Z respectively.
- These values are the ones used to calculate the `/output/moving_image` topic. Using this topic to debug your controller (through e.g. `rqt_plot`) is recommended!
- Published at 1000 Hz

G.6 From coordinates to image

In Figure G.2 shows two components that affect `/output/moving_image`. The pose is defined from the viewpoint of your camera, so a positive rotation means the RELbot shows more towards your right.

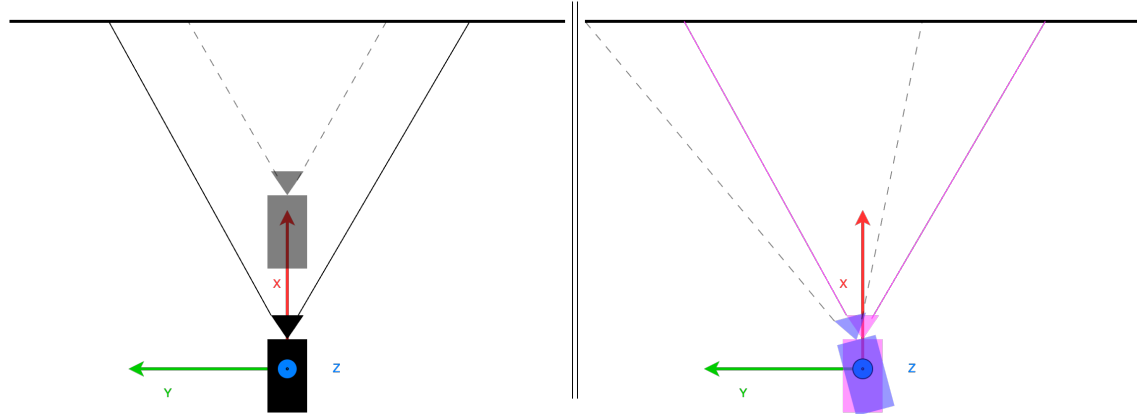


Figure G.2: Top view of how input affects the output image. Left: Increasing x implies zooming; Right: Positive rotation (from blue camera position to the pink one) shifts centre of what the cut-out camera sees, to the left

G.7 Differences with a real RELbot

There are some difference with the real RELbot The most important ones are:

- The physical RELbot setup uses PWM signals as inputs which encode the motor voltage.
- The simulated RELbot has some limits, outside of which the behaviour is non-crashing but the output image will generally not work:

	Simulator	Physical RELbot	Unit
x position	± 5	$\pm \infty$	m
θ_Z	$\approx \pm 2$	$\pm \infty$	rad

G.8 Troubleshooting the relbot simulator node

- After downloading, make sure you first build the node, after which you source the setup file again: `source install/setup.bash`
- Make sure an image stream is being sent to the `/image` topic, without this the simulator will not show an output image. The model will continue to update.
- Encountering a GTK or QT error? Try restarting the terminal shell you are in. Make sure the `ssh` argument `-X` is used!

H RELbot

The RELbot (Robotics-Education-Lab robot) is a differential-drive robot, driven by two high-quality electromotors, high-precision encoders on the motor axes, and a high-quality gearbox. The RELbot can be considered as a rigid body that can move in space. In the context of current use, the RELbot moves on a flat, horizontal surface, so a 2D space. This limits possible movements to driving forward or backward or turning (also due to the layout of the wheels of the RELbot). For describing the pose (position and orientation) of the robot, we only need x , y , and θ_z rotation around the vertical axis (z-axis).

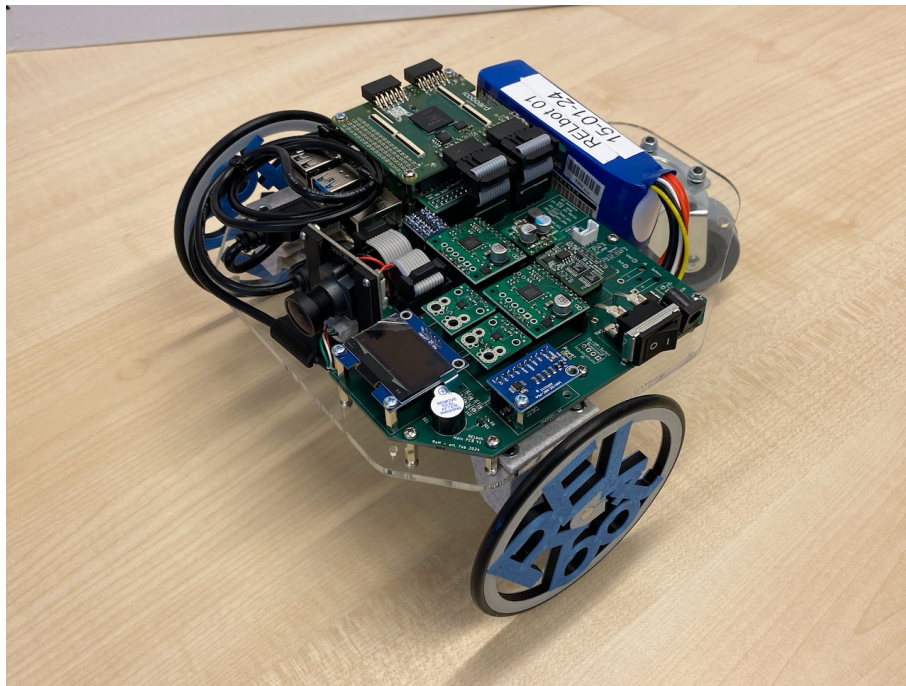


Figure H.1: RELbot v1.0.

H.1 Overview

An overview of the structure is in Figure H.2.

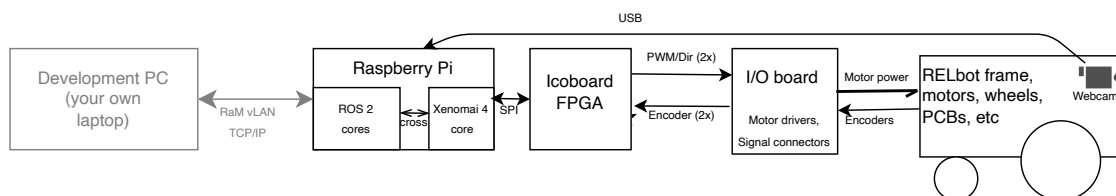


Figure H.2: CPS structure of RELbot v1.0. Connection to development machine also shown.

The connection between the FPGA on the Icoboard and the motor amplifiers / encoders is via PMOD connectors, one per degree of freedom. So, on one PMOD connector, both PWM and encoder signals of one motor–encoder pair are situated.

Relevant parameters of the RELbot are in Table H.1.

Name	Symbol	Value	Unit
Mass	m	1.35	kg
Wheel diameter	d	0.101	m
Gear ratio	n	15.58 : 1	
Motor Constant	K_m	39	rad/s /V
Encoder		1024	counts per turn
Supply voltage (motors)	V_{cc}	12	V

Table H.1: Relevant parameters of the RELbot.

H.2 Hardware parts

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:

- PWM value $[-2047 \dots 2047]$, maps to $-100\% \dots +100\%$ of full motor power. Values outside this range are clipped to -2047 or $+2047$.
- The left motor is connected to icoBoard port P2.
- The right motor 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.

Encoders

Each degree of freedom has an incremental encoder, and are connected on the motor axis. 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 encoder count is *relative to the initial orientation*, i.e., the orientation the RELbot wheels were in when the Raspberry Pi was turned on. The encoder count values are reset to zero when driver program is started, so upon start of your program.

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.

H.3 Firmware / Embedded Software

H.3.1 Connection to the RELbot via network

Connect the RELbot's Raspberry Pi to the network of the *RaM-Lab*, either via the *green* network cable or WiFi.

Connect to the Raspberry Pi of the RELbot via SSH using your account for this practical (see Section E.2) and the IP address shown on the display of the RELbot after start up (allow at least 10 s for the start-up protocol to finish and to show the IP address).

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

I XRF2, the Xenomai 4 – ROS 2 framework

The Xenomai 4 – ROS 2 Framework, second generation (XRF2) provides a skeleton to set up data communication between a ROS 2 node and code running on EVL / Xenomai 4. This Xenomai 4 code consists of a controller generated by 20-sim (or another tool). The Framework provides the FRT methods to monitor and log data in the FRT loop on Xenomai.

Note: The controller function must be called `LoopController()`.

I.1 Structure of the Framework

The file structure of the framework is shown in Figure I.1.

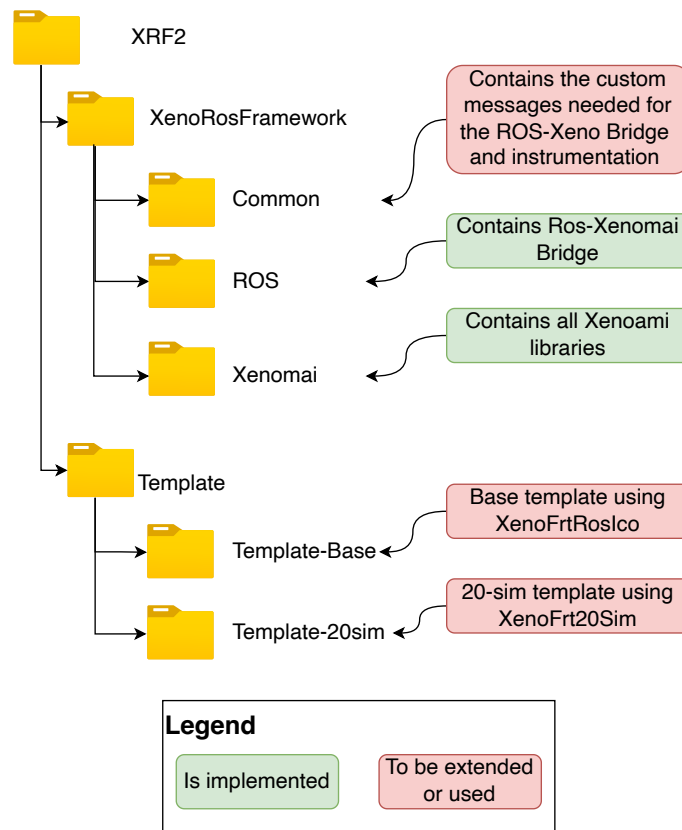


Figure I.1: File structure of the framework. Files in red annotated folders must be extended

I.2 Using the framework in your project

The framework must be downloaded on the RELbots before it can be used.

Note: Always start the Xenomai 4 side before starting the ROS-Xeno-Bridge ROS 2 node.

Use the following step-by-step instructions on how to use the framework and to extend / fill the template for the FRT loop, and to build and run the code. Some background information is given in Section I.3.

Preparation

- Download XRF2, uncompress it and put the framework in the `src` folder of your ROS2 workspace (`ros2_ws`).

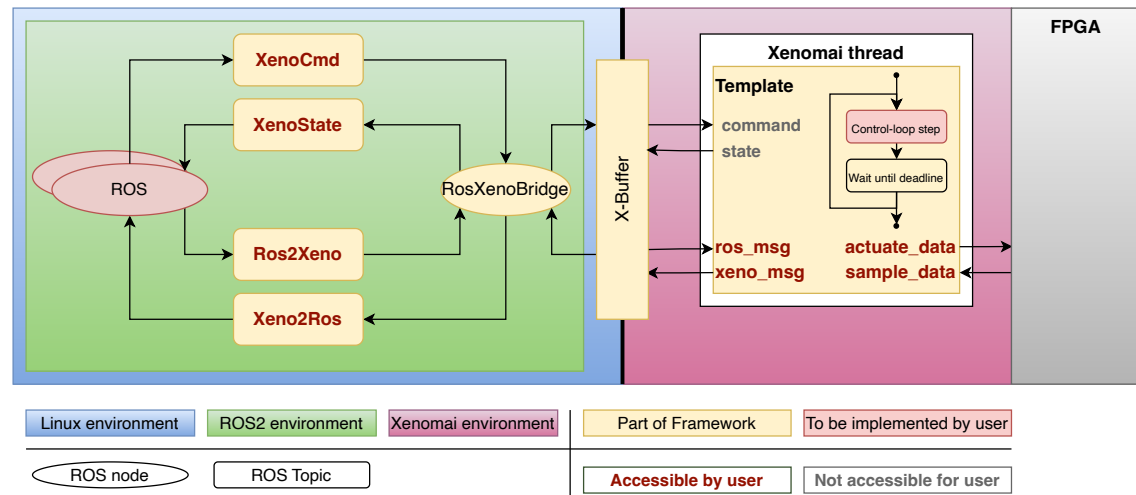


Figure I.2: Typical use of XRF2 connected to a ROS 2 node and a Xenomai program

- Copy one of the templates from the `XRF2/Template` folder in the `src` folder of your workspace. Keep the file structure of that template intact.
- Rename the copied template to the name of your project part that runs Firm Real-Time on Xenomai.
- Your folder tree should look like Figure I.3.

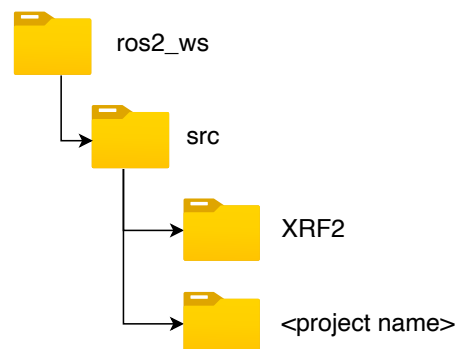


Figure I.3: Top-level of your workspace after Preparation.

Messages

- Go to the folder `XRF2/XenoRosFramework/Common/XRF2_msgs/msg`.
- Change the messages and types in the two `msg` files to fit your project's needs. One message file per direction of the messages forming the data stream between the ROS2 side and the Xenomai side. These names will be used in your code.
- Build this message package via

```
colcon build --packages-select xrf2_msgs
```
- Set the (updated) environment variables using the usual command:

```
source install/setup.bash
```

being in the root or your workspace.

Incorporate the code generated by 20-sim

- The controller submodel from which 20-sim generates code must be named `LoopController()`, such that the name of the generated class is also `LoopController`. Indeed, hardcoded.
- Go to the folder `ros2_ws/src/<project name>`
- Fill the template with 20-sim generated code, in the folder `controller` as follows:

- Remove all files in this directory, as these are skeleton files, and 20-sim produces new files.
- Generate 20-sim code from your controller model and copy it to this folder.
- **Note:** The class of the generated code must have the name `LoopController`.
- **Note:** *All* of the code in this folder is generated, so do *not* edit these files.

Finite State Machine

First update the names of the project and files, then edit the header and cpp files of the FSM, shown in Figure I.4.

- Go to the folder `ros2_ws/src/<project name>`
- Open the `CMakeList.txt` file and rename the project name on line 2 to match with your project.
- In the folder `include` rename the header file `Template.hpp` to match with your project name.
- Edit this `.hpp` file as follows:
 - Rename the class and (de)constructor to match with your project and thus file name.
 - If the logger is used see Appendix J about how to use it.
 - if the logger is *not* used, remove the `struct GenericLogStruct`.
 - Change the size of the arrays `u` and `y` to the correct size specified by your 20-sim-generated controller code.
- In the folder `src` rename the `.cpp` file to match with your project / header file.
- Edit this `.cpp` file as follows:
 - Remove all code lines *in* the functions implementing states that you are *not* going to use. Do *not* remove the return statements in these state functions.
 - Specify the actions / code of the states you need in its corresponding function. These are the states *Initialising*, *Run*, *Pausing*, and *Stopping*, as shown in Figure I.4.
 - **Note:** The *Run* state function is prefilled with a call to the `LoopController` function.

Tweak the Computation in the *run* state function

The 20-sim model is already imported and is called `controller`. Through the `controller.calculate` function one computation step is performed. The `controller.finished` function indicates when the simulation is finished.

Tip: Use `monitor.printf()` instead of `evl_printf()` for debug messaging to reduce the output flow of the `printf` statements to a lower frequency. It is an FRT-friendly function.

- Go to the folder `ros_ws/src/<project name>`
- Edit `main.cpp` and replace the template class with your own class.
- On line 13 the object of the class is made. The parameters of the class are :
 1. Communication frequency to ROS (Hz)
 2. Monitor frequency (Hz) of the `monitor.printf()`

Build and Run

- If not already done add all your ROS2 projects to the `ros2_ws/src/` folder.
- Go to the base of your workspace (folder `ros2_ws`).
- Build all code present in this workspace (in `/src`) via `colcon build`

To exclude a project from being compiled add a file called `COLCON_IGNORE` to the source folder of that project.

- Set the (updated) environment variables using the usual command:
`source install/setup.bash` being in the root of your workspace.
- Start the Xenomai project *always* first, via:
`sudo ./build/<project name>/<project name>`
- Start the ROS-Xenomai bridge in a different terminal being in the root of your workspace:
`ros2 run ros_xeno_bridge RosXenoBridge`
 Do set the environment variables first via `source install/setup.bash`
- **Note:** The Xenomai4 project has to be started first in order to let it work.
- Start other ROS2 nodes in separate terminals, as usual:
`ros2 run <package name> <node name>`
- Initialise the FSM and thus to start the *computation* at the Xenomai side by publishing the *initialise* command to the *XenoCmd* topic.

Tip: Create a ROS2 launch file to make it easy to start all needed ROS2 nodes and topics. This reduces the amount of terminals needed. Commands to publish data to certain topics can also be added to launch files.

I.3 Background Information on XRF2

I.3.1 Finite State Machine

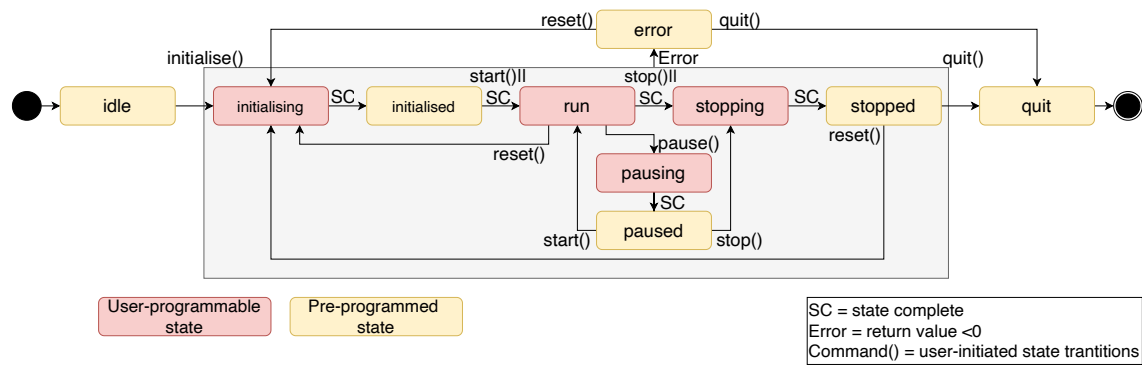


Figure I.4: Diagram of state machine implemented – from slide A-3-52.

Each function except the (de)constructor in the class represents a state. The state diagram is shown in Figure I.4. The state machine can be controlled by the ROS2 topic *XenoCmd*. The resulting state value is published in the *XenoState* topic. To change the state of the FSM, publishing the integer value representing the new state to the *XenoCmd* topic. The commands are shown in Table I.1.

Command	Value	Discription
initialise	1	Go from idle to initialising state. Before any movement can happen this command has to be given from the ROS side of the communication.
start	2	Go from the initialised or paused state to the run state.
stop	3	Go from the run or paused state to the stopping state.
reset	4	Go from the run, stop, error or pause state to the initialising state.
pause	5	Go from the run state to the pausing state.
quit	6	Go from the error or stopped state to the quit state. After this command is called, the Xenomai side will shut down.

Table I.1: Commands that can be given through the ROS2 topic *XenoCmd* to the FSM on the Xenomai4 side.

I.3.2 FPGA data

The variable `sample_data` provides all the sampling data from the FPGA and has the following structure:

```
struct IcoRead {
    int channel1;      // Encoder value from channel 1
    int channel2;      // Encoder value from channel 2
    int channel3;      // Encoder value from channel 3
    int channel4;      // Encoder value from channel 4
    bool channel1_1;   // Digital input 1 from channel 1
    bool channel1_2;   // Digital input 2 from channel 1
    bool channel2_1;   // Digital input 1 from channel 2
    bool channel2_2;   // Digital input 2 from channel 2
    bool channel3_1;   // Digital input 1 from channel 3
    bool channel3_2;   // Digital input 2 from channel 3
    bool channel4_1;   // Digital input 1 from channel 4
    bool channel4_2;   // Digital input 2 from channel 4
};
```

The variable for controlling the FPGA output is called `actuate_data` and its structure is:

```
struct IcoWrite {
    int16_t pwm1;      // PWM value for channel 1
    bool val1;         // Digital output 1 of channel 1
    int16_t pwm2;      // PWM value for channel 2
    bool val2;         // Digital output 1 of channel 2
    int16_t pwm3;      // PWM value for channel 3
    bool val3;         // Digital output 1 of channel 3
    int16_t pwm4;      // PWM value for channel 4
    bool val4;         // Digital output 1 of channel 4
};
```

I.3.3 Controller data

As specified in the package `XRF2_msgs`, the data sent from ROS2 to Xenomai is called `ros_msg` and has the structure of the `Ros2Xeno` message. The data to be sent from Xenomai to ROS2 is set in the variable called `xeno_msg` and has the structure of the `Xeno2Ros` message.

The message files are in `XRF2/XenoRosFramework/Common/XRF2_msgs/msg`

I.3.4 UML diagram of the framework

The class diagram of the framework is shown in Figure I.5.

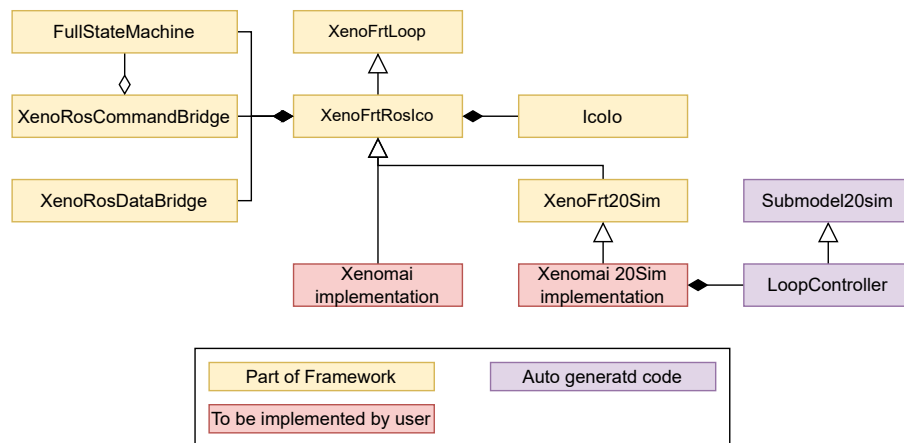


Figure I.5: Class diagram of the framework. One of the two parts in red must be extended

J Using the XenoFrtLogger

XenoFrtLogger provides methods to firm real-time log data on Xenomai when the control loop is running. This is done through a proxy that communicates the data from the Xenomai to the Linux kernel to log the data on a file. This class works on both the Xenomai and the Linux kernel.

The logger works by having one variable to log data from. This variable has to be defined by the user. Each element in the variable has to be registered. Start and stop commands can be given to determine when the logger has to log the data.

Below, step-by-step instructions are given on how to implement and use the XenoFrtLogger. Under each step, a code example is shown.

1. Create a variable that is going to be logged. This variable can consist of multiple elements like an array or struct. When using structs make sure to use "#pragma pack (1)" before the struct and "#pragma pack (0)" after the struct. This will remove the space within the memory between different elements. In the definition of the struct, default values can be given.

```
// Definition of the struct ThisIsAStruct
// A normal struct name is PosistionData
#pragma pack (1)
struct LoggedVariable {
    int this_is_a_int;
    double this_is_a_double;
    float this_is_a_float;
    char this_is_a_char;
    bool this_is_a_bool;
};
#pragma pack(0)

// Variable that is going to be logged
struct LoggedVariable data_to_be_logged;
```

2. Go into the main function and create a XenoFileHandler instance which represents the log file.

```
file =
XenoFileHandlerfile(1, "/home/pi/Logger_Test", "bin");
```

3. Create the XenoFrtLogger instance.

```
logger = XenoFrtLogger(&file,
&data_to_be_logged);
```

4. Register all elements in the variable to the logger. Table J.1 shows all the variable types that can be registered. The registration of the elements has to happen in the same order as the allocation in the variable.

Name	Size	Type
id_bool	1	bool
id_char	1	char
id_float	4	float
id_int	4	int
id_double	8	double
id_long_int	8	int64, timespec.tv_sec, timespec.tv_nsec

Table J.1: Different constants of the enum VariableTypes.

```

logger.addVariable("this_is_a_int", id_int);
logger.addVariable("this_is_a_double",
id_double);
logger.addVariable("this_is_a_float", id_float);
logger.addVariable("this_is_a_char", id_char);
logger.addVariable("this_is_a_bool", id_bool);

```

5. After having added all elements, initialise the logger once. If done twice, the program will crash. A check can be done to avoid this. This call might take a while therefore it is advised not to do this within a control loop.

```

if (!logger.isInitialised())
    logger.initialise();

```

6. Start the logger by calling:

```

logger.start();

```

7. The data has to be allocated to the variable given by the logger before it can be logged. Afterwards, the log function has to be called, to sent the data to be logged. This has to happen each cycle the data has to be logged. In the example below 100 cycles are simulated in a for loop.

```

// Setting initial values for data_to_be_logged
data_to_be_logged.this_is_a_bool = 0;
data_to_be_logged.this_is_a_int = 0;
data_to_be_logged.this_is_a_char = 'R';
data_to_be_logged.this_is_a_float = 2.0;
data_to_be_logged.this_is_a_double = 4.0;

//Cycling through 100 cycles
// And changing the data each cycle
for (size_t i = 0; i < 99; i++)
{
    // Allocating data to data_to_be_logged to be logged
    data_to_be_logged.this_is_a_bool =
!data_to_be_logged.this_is_a_bool;
    data_to_be_logged.this_is_a_int++;
    if (data_to_be_logged.this_is_a_char == 'R')
        data_to_be_logged.this_is_a_char = 'A';
    else
        data_to_be_logged.this_is_a_char = 'R';
}

```

```
        data_to_be_logged.this_is_a_float =  
data_to_be_logged.this_is_a_float/2;  
        data_to_be_logged.this_is_a_double =  
data_to_be_logged.this_is_a_double/4;  
  
        // Sent data to the log file  
        logger.log();  
    }
```

8. Stop the logger when not more needed, by calling:

```
logger.stop();
```

9. After being done with logging, the user can decode the log file into a .mat or .csv file. This can be done by going to 'ros2-xenomai4-framework/XenoRosFramework/Common/XenoFrtLogger_decoder' folder in the framework. In this folder, two python scripts reside for decoding the log file to the preferred file type. This is done by calling one of the scripts in the terminal and following the given instructions.

```
python binary_to_matlab.py
```

K Tips & Tricks and cheatsheets

K.1 Introduction

Over the length of this document, a lot of commands are used, here we provide a collection of some often used commands. General "cheatsheets" containing more information are referenced. In line with commonly used styles, *<name>* denotes a name to be filled in by the user, with the name filled in providing a hint of what the 'type' should be. A *[]* will be used to describe different options if needed, with the pipe character (*|*) used to separate the different options.

Generic shell explainer: <https://explainshell.com/>

K.2 General terminal commands

A lot things will be done through the terminal, whatever system you have. Most of these commands will also work on Windows. Most terminal applications have auto-complete build in, which can be used by pressing tab. If no option can be given, a sound will play. Sometimes the application is not aware of all options: try writing them out, an error will pop up if there was a mistake, sometimes with hints or shows the string to type to get concise help info on the command itself.

Some general commands are:

- `cd [~ | ../ | <folder-name>]`: used for navigation. `~` denotes home directory in Linux and Mac, `C:` does the same for windows. `../` moves you back one folder. `<folder-name>` moves you into the folder named.
- `mkdir <name>`: Makes a directory called name.
- `ls`: List contents
- `pwd`: path to working directory
- `cp <location a> <location b>`: Copy file from a to b.
- `mv <location a> <location b>`: Move file from a to b.
- `rm <file>`: Remove a file.
- `<command> --help`: Generally works to present the descriptions of options for a command. Longer explanations can be found online.
- `nano <filename>`: Show and edit file contents in the terminal, useful for checking if file matches expectations. Exit with `Ctrl + X`.
- `Ctrl/Cmd + R`: Reverse search, allows you to search back in the previous commands you have given.
- Arrow keys: Allows you to search through old commands (up/down), or move your text cursor around a command (left/right).
- `code .`: Open VS Code on this location.

K.3 VS Code

General cheatsheet: [Windows](#), [Mac](#), [Linux](#).

- `Ctrl/Cmd + Shift + P`: Opens the Command Palette, allows searching for commands. Also shows the key combination attached to a command.
- `F1`: Same as above

K.4 Compiler (g++ or clang)

The focus is on the command line here, not on using VS Code to build files.

Note: when compiling using VS Code's interface, be aware to select the g++ compiler, so *not* the gcc compiler, which is often the first to select from in the drop-down box of the command palette.

Cheatsheet: [From GNU itself](#), [a course given somewhere else](#)

First, some core arguments:

- `-o <name>`: define the name of your output file.
- `-c`: Compile but do not link.
- `-std=c++14`: Set C++ Standard to C++ 14 or `std=c++17` for C++ 17, obviously.
- `-pthread -lpthread`: Linux environments only; Makes sure that threads can be used in the program (Windows / Mac do *not* need to have this flag to properly deal with threads).

Using these arguments in combination with each other allows to write:

- `g++ a.cpp b.cpp -o C`: Compile a.cpp and b.cpp, and create an output called C (automagically becomes C.exe on windows)
- `g++ a.cpp b.cpp -c`: Only compile, but do not link. Generates output `a.o b.o`.
- `g++ a.o b.o -o C`: Take two compiled files, and link them into output file C.

K.5 WSL (Windows)

WSL does not have that many specific commands. Mostly, you use the general terminal commands for more information, see Section K.2.

Cheatsheet: <https://learn.microsoft.com/en-us/windows/wsl/basic-commands#install>

- `wsl`: Start WSL at the given location, this is also where the WSL prompt will be.
- `wsl ~`: Start WSL, but go to the home of your wsl environment (`\home\ubuntu`)
- Inside WSL instance: `explorer.exe .` : opens a file explorer window at given location.

K.6 Multipass (Mac)

Multipass can be accessed through the GUI, or through a terminal / shell from the host (of course). Some Terminal commands are mentioned here. The `<name>` of the instance (VM running within Multipass) is "jazzy", in case you followed our install guide.

- `multipass start <name>`: start the instance (VM) with the given name.
- `multipass shell <name>`: Open a shell in the given instance, that needs to be running.
- `multipass stop <name>`: Stop multipass instance from running.
- `multipass restart <name>`: Restart multipass instance.

K.7 SSH

After setting up the SSH connection to a device, there is one main command to use:

- `ssh <user>@<ip>`: User is the username, IP is the ip address. After this, fill in the information ssh requests.
- `ssh -X <user>@<ip>`: Same as before. `-X` enables X11 forwarding, which lets ssh be used for showing graphics on the host produced on the VM.

K.8 ROS 2

ROS 2 is a large package, with lots of commands.

Cheatsheet: https://github.com/ubuntu-robotics/ros2_cheats_sheet/blob/master/cli/cli_cheats_sheet.pdf

- `ros2 run <package> <node name>`: Runs a specific executable (node) from the given package
- `ros2 param [get | set] <name>`: Get the value of parameter name, or set the value of parameter name.
- `ros2 topic list`: Give a list of all topics currently active
- `ros2 topic echo <name>`: Print the message currently being published on the topic called name
- `ros2 topic type <name>`: Give information about the type of the information on the topic
- `ros2 launch <package> <launch-file>`: Launch the executables in the launch file.

K.9 Colcon

Colcon is the build system for ROS 2. Colcon is based on CMake and Make. Colcon basically searches for all packages 'below' the location of where it is called and searches for CMakeLists.txt and package.xml to identify packages. After this, it tries to run these and build these packages, with knowledge of the ROS 2 install location for possible package-linking.

Cheatsheet: https://github.com/ubuntu-robotics/ros2_cheats_sheet/blob/master/colcon/colcon_cheats_sheet.pdf

- `colcon build`: Builds all package in the workspace. Call it from workspace root, as it will place a /build, /install and /log folder in the directory from which it is called.
-

K.10 Terminal applications useful for ROS

ROS often requires multiple processes called via separate terminal windows to run at once. We present some options which we consider useful for this. Main reason for usage is the splitting of windows into *panes*, each of them having their own terminal application.

K.10.1 Terminator

[Terminator](#) is a terminal application for Linux.

- `sudo apt install terminator`: run once, installs the application. Run by searching for terminator.
- `Ctrl + Shift + E`: Split window vertically.
- `Ctrl + Shift + O`: Split window horizontally.
- `Ctrl + (Shift +) Tab`: Move to next window, or back when also pressing shift.

K.10.2 Windows Terminal

[Windows Terminal](#) should not be confused with the Command Line or Powershell applications. Installed via [Microsoft Store](#), no account should be required. Has many options and can also run other shell types when installed.

- `Alt + Shift + '+'`: Split window vertically.
- `Alt + Shift + -`: Split window vertically.
- `Alt + Shift + D`: Split window vertically.
- `Ctrl + Shift + W`: Closing Pane; if one pane, the tab closes; if one tab, the application closes.
- `Alt + Arrow-keys`: Move focus to other pane.

- `Ctrl + Shift + P`: Command Palette, allows for searching for commands

K.10.3 iTerm2

[iTerm2](#) is a Mac terminal application which does multi-pane terminals. Install via the link above. Under Settings, Profiles you can set "Reuse previous session directory" to start the new terminal in the directory of the terminal which has focus while starting a new terminal.

- `Cmd + D`: Split Window vertically.
- `Cmd + Shift + D`: Split Window horizontally
- `Cmd + W`: Close Pane; if only one pane, the window closes.
- `Cmd + Option + Arrow-keys`: Move between panes.

References

Deitel P J, Deitel H M, 2017 *C++ How to Program*, Prentice Hall, 10th edition ISBN 978-1-292-15334-6, or 978-0-134-44823-7