KAL Toolchain Tutorial

Danial Kamran, Sascha Wirges, Christopher Wecht, Martin Lauer

I. Introduction

This document provides you with the necessary information to use Anicar, the build tool chain and supports you in order to develop your own software. We do not provide detailed information about C++, git or ROS but instead refer to comprehensive resources.

We first describe the vehicle hardware in section II. Please read this chapter carefully as the same hardware is used by all teams and one of the few things that can be damaged when handled improperly. We then give an overview on our build system in section III while explaining the fundamental details that are necessary to get an idea of the build process. Section IV provides a detailed overview on the software packages used in our lab and main modules such as localization or stereo matcher are explained here. Finally, we provide a recipe of how to get you started in section V. After working through this section you are able to develop your own software using our tool chain.

II. Hardware

Most parts of Anicar were completely designed by the MRT in order to fulfill the requirements of the cognitive automotive laboratory, such as robustness and simplicity of operation. Figure 1 gives an overview of the hardware provided, not including the vehicles.

Fig. 1. Lab Hardware available. There are three silver (one big and two small) **Camera** and four black **Motor** battery packs. There are two notebooks, which can be used for any car.

The vehicles are equipped with an electric motor for driving, a servo for steering, a second-generation Kinect in the front and a top camera for localization via a modified version of the stargazer-system (originaly from Hagisonic).

In order to use the stargazer-system, you should plug in the power cable in the back to the power supply shown in Figure 2 to switch it on. It is located near the window opposite to the computer room. After running the correct ROS nodes, you will see the LED-Landmarks in the images from the top camera. Please disconnect the power cable after you finish using the stargazer.

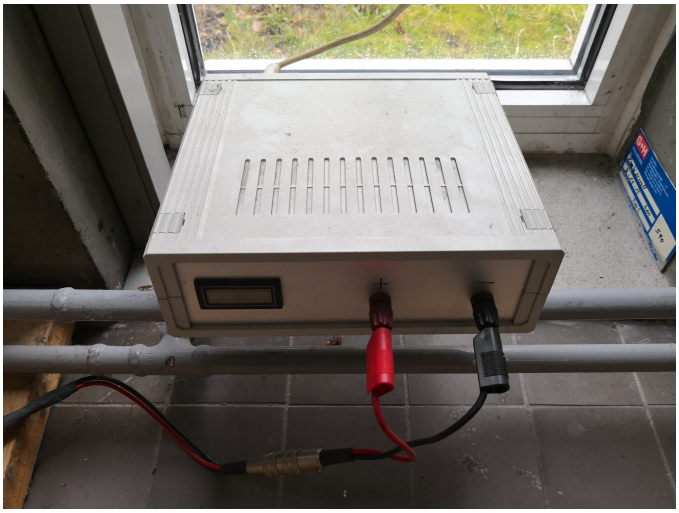

Fig. 2. The power supply for Stargazer.

The Kinect uses a wide-angle time-of-flight camera that resolves distance based on the known speed of light, measuring the time-of-flight of a light signal (invisible by using an active IR sensor) between the camera and the subject for each point of the image. There are three resolutions for the depth map, a FullHD resolution (1920x1080), a quarter FullHD resolution (960x540) and the raw IR and depth images from the sensor (512x424). The camera also captures 1080p color video up to 30 frames per second.

The computational work is done by a notebook which can be placed on top of the car. The electrical power is provided by two Lithium-Ion-battery-packs: one for the motor (its cable is labeled with *motor*) and one for all other components (its cable is labeled with *camera*).

For setting up the car, follow these steps:

- Voltage supply: Connect the batteries and switch them on
- Data connection: Plug in the ethernet- and the USB-cable

Note: Make sure that the batteries operate with **12V**! The

silver battery-pack is already configured correctly,. The black battery-pack has to be configured every time it has been turned on. The the off-lo-hi-switch has to be at lo, the LED at 26V/12V must be active. To switch between the different voltages press the button at the left as long as the current LED is flashing. After the release of the button, the next LED gets active. The correct configuration is shown in Figure 3.

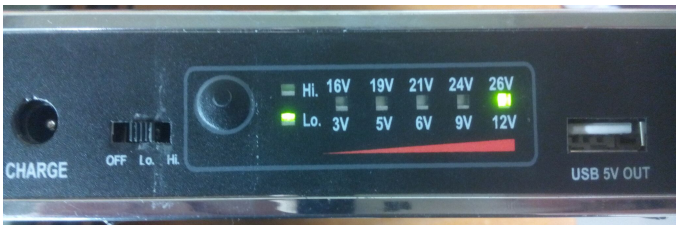

Fig. 3. Correct configuration of the black battery-pack.

If you are done using the batteries or the notebooks, please switch them off and connect them to the charger again.

For manual control every vehicle has its own game pad. On the downside of every game pad the name of the associated car is imprinted. The game pads only work if the notebook has been plugged in and the correct ROS nodes are running (for more details see section IV-B). The gamepad button mapping is described in table I.

Fig. 4. Configuration of the silver battery-pack. Don't change the voltage

III. Build System

A. C++

The software provided is programmed and built in the C++ programming language. In contrast to *interpreted languages* like Python, C++ is a *compiled language*. This means, you first need tools that translate your *source code* into a set of instructions that can be executed by the processing units. In case you have worked with Java before (probably most of you), transition guides such as [**?**] exist. These guides explain the major commonalities and differences between the two languages.

Using our build system, you do not need to know details about the translation process. However, we encourage you to get an understanding of C++ programming as it is an efficient and fast way of programming.

B. git and gitlab

Git [**?**] is used as distributed version control system (VCS). With a VCS, you do not need to keep multiple copies of files, but instead commit changes at certain stages and tagging them with a meaningful name. A VCS tracks changes made on text files and stores them efficiently. Git also makes it convenient to work on one project with different users using multiple *branches*. If two users commited changes to the same file, git detects this and either merges those changes automatically or asks you to resolve these *conflicts* manually.

Here, we run our own *gitlab* server [**?**]. Gitlab is a system that makes it more convenient to manage many git repositories. It contains a remote repositories that you can use to *push* your files and make them available to your team. Also, it's webinterface provides a good overview on documentation, different branches or other events. Furthermore, you can use our gitlab server to run tests on software.

C. ROS

We make extensive use of the *Robot Operating System* (ROS) [**?**], the de-facto standard in prototyping and development of robotic systems.

ROS has some important advantages for our development. It is well-documented, easy to learn and based on the publisher / subscriber paradigm (message base communication), which helps to create clean interfaces from the beginning. Different processes (*nodes*) in ROS usually either exchange messages over *topics*, or *services* that serve as interface between them. Topics can be recorded and stored in *rosbags* which can be replayed later. Many useful nodes such as sensor interfaces for cameras or laser scanners or a stereo-matcher are shipped with ROS. Comprehensive tools for visualization and debugging such as *rviz* and *rqt* complete the ROS-ecosystem. ROS has a large community. Many projects are open-source and available on github.

Please understand that we can just give an overview about ROS here, but instead refer to the very well maintained wiki [**?**].

However, we provide a list with important tools that you might use during your lab:

- *rosbag*: A set of tools that is used for recording to and playing messages from `.bag` files.
- *rviz*: Visualize lots of messages, including images, point clouds, odometry, localization.
- *rqt_graph*: Visualize running nodes and their interaction. This might be used to diagnose improperly routed messages.
- *rosrun*: Run single nodes.
- *roslaunch*: Start launch files. A launch file usually contains a list of nodes to be started, e.g. a processing pipeline.
- *rostopic echo*: Subscribe on a given topic and show its contents.
- *rostopic info*: Show information about a given topic, e.g. subscribers, pusblisher message-type.
- *rosmsg show*: Show the message definition of a given message-type.

In section V-F a quick-start-guide is given how to create nodes with MRT-tools.

D. Packages

In ROS, software is structured in packages. Packages are a collection of nodes, libraries, configuration files and other resosources which belong together in any way.

E.g. for `path_generator_ros_tool`, one of the example packages, the structure is shown below.

```
|-- cfg
|   |-- PathGenerator.mrtcfg
|-- CMakeLists.txt
|-- launch
|   |-- params
|   |   |-- path_generator_parameters.yaml
|   |-- path_generator_node.launch
|   |-- path_generator_nodelet.launch
|-- nodelet_plugins.xml
|-- package.xml
|-- README.md
|-- res
|   |-- raw_path.osm
|   |-- fitted_road_map.osm
|-- src
|   |-- path_generator
|       |-- bag.hpp
|       |-- path_generator.cpp
|       |-- path_generator.hpp
|       |-- path_generator_node.cpp
|       |-- path_generator_nodelet.cpp
|-- test
```

A package in ROS consists of the following parts:

- `src` contains the source files, e.g. the nodes and libraries
- `test` contains tests
- `launch` contains the launchfiles, might contain parameter yaml-files in `launch/param`
- `cfg` contains the files needed for using `dynamic_reconfigure` [**?**]
- `msg` / `srv` contains message and service definitions respectively
- `res` contains other resources like rviz-configuration files or the map generated by the package
- `package.xml` defines the package, e.g. the name, dependencies
- `README.md` contains the documentation of the package

E. ROS Build System: catkin on top of cmake

A build system is responsible for generating *targets* from raw source code that can be used by the end user. These targets may be in the form of libraries, executable programs, generated scripts, exported interfaces (e.g. C++ header files) or anything else that is not static code. In ROS terminology, source code is organized into *packages* where each package typically consists of one or more targets when built. In addition, virtually all integrated development environments (IDEs) such as QtCreator or Eclipse add their own build system configuration tools. Often the build systems in these IDEs are just front ends for console-based build systems such as CMake. To build targets, the build system needs information such as the locations of tool chain components (e.g. C++ compiler), source code locations, code dependencies, external dependencies, where those dependencies are located, which targets should be built, where targets should be built, and where they should be installed. This is typically expressed in some set of configuration files read by the build system. In an IDE, this information is typically stored as part of the workspace/project meta-information. With *CMake*, it is specified in a file typically called *CMakeLists.txt* and with GNU Make it is within a file typically called *Makefile*. The build system utilizes this information to process and build source code in the appropriate order to generate targets.

ROS uses *catkin*, a custom build system that extends CMake to manage dependencies between packages. In particular, catkin combines CMake macros to provide some functionality on top of the normal workflow of CMake. Catkin allows for better distribution of packages, better cross-compiling support, and better portability.

F. IDEs

We encourage you to use an *Integrated Development Environment* (IDE) that supports you with referencing between files, indentation and syntax highlighting. Documentation on how to set up *QtCreator* or *Eclipse* as IDE for our build tool chain is explained in [**?**].

G. MRT Build Tools

The *MRT build tools* support you in building your workspace, testing code, creating documentation, downloading packages from gitlab, creating new packages, creating rosbags and many more things.

The MRT build tools provide a command line interface (CLI) with a comprehensive help menu. To try this menu, simply type `mrt` when you are inside your workspace. If you need help for some of the functions, e.g. `pkg`, just type `mrt pkg` and a help menu for this function will be printed.

In order to start the MRT tools, the setup file located at `/opt/mrtsoftware/setup.bash` must be sourced.

Although we do not explain all features, we like to highlight a few important features that you find helpful during your work.

The `mrt ws update` command provides a simple way to update the complete workspace.

IV. Packages

Here, we give an overview about the packages provided for the lab. The most important packages are briefly explained. For more information, please refer to the package documentation in the package itself (`README.md` in the main folder of the package) or in gitlab.

All packages (except of `stargazer_ros_tool`) described here are part of the

Button/Stick	mapping
Left/Right Axis stick right	steering
Up/Down Axis stick left	velocity
B - button	activate motor interface
Y - button	deactivate motor interface

TABLE I
GAME PAD BUTTON MAPPING

kognitive_automobile_labor-namespace in gitlab [**?**].

A complete overview of the system, its components and their interactions is given in figure 5.

Note: If you encounter a problem with the provided software-packages or miss a feature, create an issue for the respective package in gitlab. Also feel free to modify the provided software and create merge-requests, your help will be welcome.

A. motor_interface_ros_tool

The package motor_interface_ros_tool provides the interface to the motors (motor and servo) of the vehicle. The motors can be controlled by sending message of the type motor_interface_ros_tool/MotorCommand to this node. On startup this node is deactivated. For activation the activation service must be called (e.g. by tui_ros_tool or joy_converter_ros_tool).

B. joy_converter_ros_tool

The package joy_converter_ros_tool provides control via game pad. It interfaces the motor_interface_ros_tool with joy_node (part of the joy-package [**?**]). If you start the launch-file joystick.launch of this package joy_converter_ros_tool and joy_node are started with the mapping of buttons in tabular I.

C. tui_ros_tool

The package tui_ros_tool provides control via a text-based user interface. The node prints out a short help at startup or if you press h.

D. stargazer_ros_tool

The package stargazer_ros_tool provides a landmark-based indoor localization solution. By default, the localization is provided by a transform from stargazer-frame to camera_top-frame (tf2 is explained in [**?**]). Additionally, it provides visualization of the landmarks. Usually you will not use this package directly but by using the localization.launch-launch-file provided by the vehicle_setup-package.

E. vehicle_setup

The package vehicle_setup provides launch-files to setup up the given functionality of the vehicle including localization and stereo-vision. It also includes launch-files for visualization. Additionally the stargazer calibration files for each vehicle are stored in this package. To create a new calibration, a guide is given in the README.md of this package.

F. kal

The meta-package kal is intended to simplify the check-out process of the kognitive automobile laboratory software stack. By resolving the dependencies of this package, every other package is downloaded. It also provides top-level launch-files for runnig all parts of the kal-software-stack together. You can use master_live.launch to control the anicar with joystick, you can also run demo_path_follower.launch to let anicar autonomous driving.

G. simulation_ros_tool

The package simulation_ros_tool contains vehicle_simulator, which can replace motor_interface for simulation purposes.

H. path_generator_ros_tool

The package provides the roadmap for the contolling part.it can convert the trajectory inside the rosbag into a osm map file. launch path_generator_node.launch to generate new map.

I. kinect2_bridge

The package kinect2_bridge provides a driver and the tools needed to receive data from the Kinect-2 sensor. The received data contains point-clouds, depth-clouds and images. You can find further information in [**?**]. To start the communication with Kinect, launch kinect2_bridge.launch.

V. EXAMPLES

Here, we demonstrate an exemplary pipeline that consists of setting up the build environment, downloading and building the relevant source code and executing a simple program. Furthermore, we show up how to modify a program in order to modify the functionality of programs. After reading this chapter you should be able to develop your own software using the MRT build system.

A. Create Build Environment

Before downloading and building packages, you need to create a catkin workspace and set up relevant ROS environment variables. You can find a guide on how to do this in [**?**]. You will then be able to download packages from our gitlab server.

B. Download and Build Packages

From within your workspace, you can download any package, using the mrt pkg command. To simplify the download of all packages, we created a meta-package, named kal. By executing mrt pkg add kal, you can download all packages relevant to the lab into your workspace.

Then, call mrt catkin build or mrt b to build all packages within the workspace. **Attention** : When you want

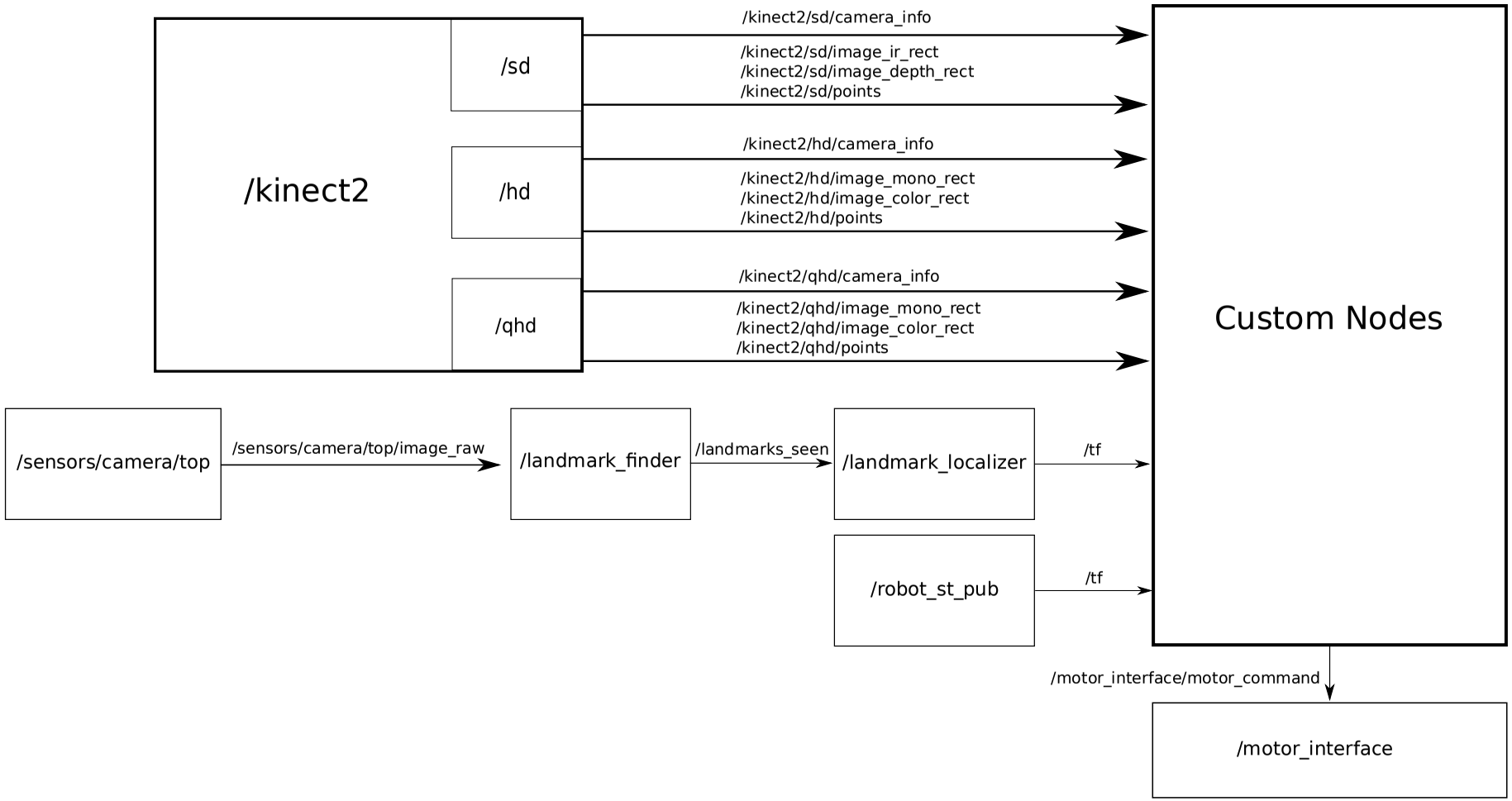

Fig. 5. System Overview: boxes are nodes (or multiple nodes) and arrows are topics. `/robot_st_pub` publishes a model of the vehicle which can be used for visualization or by any other node. The topics are described in table II.

Topic	Message-Type	Description
/kinect2/sd/camera_info	sensor_msgs/CameraInfo [?]	Camera info for raw image
/kinect2/sd/image_ir_rect	sensor_msgs/Image [?]	rectified raw RGB-image of the camera
/kinect2/sd/image_depth_rect	sensor_msgs/Image [?]	rectified raw depth-image of the camera
/kinect2/sd/points	sensor_msgs/PointCloud2 [?]	raw point cloud from kinect
/kinect2/hd/camera_info	sensor_msgs/CameraInfo [?]	Camera info for raw image
/kinect2/hd/image_mono_rect	sensor_msgs/Image [?]	rectified HD resolution mono-image of the camera
/kinect2/hd/image_color_rect	sensor_msgs/Image [?]	rectified HD resolution RGB-image of the camera
/kinect2/hd/points	sensor_msgs/PointCloud2 [?]	HD resolution point cloud from kinect
/kinect2/qhd/camera_info	sensor_msgs/CameraInfo [?]	Camera info for raw image
/kinect2/qhd/image_mono_rect	sensor_msgs/Image [?]	rectified QHD resolution mono-image of the camera
/kinect2/qhd/image_color_rect	sensor_msgs/Image [?]	rectified QHD resolution RGB-image of the camera
/kinect2/qhd/points	sensor_msgs/PointCloud2 [?]	QHD resolution point cloud from kinect
/sensors/camera/top/image_raw	sensor_msgs/Image [?]	raw mono-image of the top camera
/landmarks_seen	stargazer_ros_tool/Landmarks	landmarks recognized by the stargazer
/tf	geometry_msgs/TransformStamped [?]	part of *tf2* [?]). **Do Not Subscribe!**
/motor_interface/motor_command	motor_interface_ros_tool/MotorCommand	set velocity and steering angle

TABLE II

DESCRIPTION OF TOPICS

to build only one pacage, you can get into the package and call `mrt bt` or `mrt bt -rd`.

In order to set up your Eclipse or QtCreator IDE properly, you need to append a `--eclipse` or `--qtcreator` flag to the build command. Please call `mrt catkin --help` for a list of all flags.

After successfully building the make sure that the workspace environment variables are set. E.g. you can do this by calling `source ~/catkin_ws/devel/setup.bash` or `mrtsrc`.

C. Setup IDE

As mentioned in section III-F, [**?**] provides a comprehensible guide on how to setup two IDEs for our build system.

D. Start Example Program

We recorded some rosbags with camera images that you can directly start and use for diagnostics or debugging. It is located in the `/mrtstorage/rosbags/unit_test/kal` directory. To start a simple processing chain consisting of stereo matching and localization, type `roslaunch kal play_bag.launch file:=</path/to/bagfile>` and provide a valid rosbag file.

Attention : To play data offline, you need to specify which car you used by setting the environment variable `CAR_NAME`, e.g. by calling `export CAR_NAME=anicar3` prior to starting processing launch files.

After starting the launch file, rviz will show up and the rosbag will be paused. You can toggle between start and pause hitting space.

E. Record and Replay Measurements

With ROS comes *rosbag*, a tool that makes it easy to record and replay ROS messages. If you want to record messages, you can use `rosbag record <topic/name>` to start a recording. That data is then written to a *bag* file ending in `.bag`. You can then replay this data with the `rosbag play </path/to/bagfile>` command. For a full list of functions, please refer to [**?**]. We also prepared record and play launch files in the `kal` package that you can adapt to your needs.

F. Create New Packages and ROS Nodes

To extend Anicar's functions, a simple and modular way is to create new packages. From inside your workspace, you can simply call `mrt pkg create <name>` to create a new package called <*name*>. For our development, we distinguish between four different package structure: Library or executable, with / without ROS dependencies. It is a good strategy to create ROS-agnostic libraries and write a small ROS wrapper around these libraries in a separate ROS-dependent tool. In general, we recommend to create two new packages for each main module you want to develop. However, for prototyping it is also convenient to first integrate all functions into a ROS executable package.

The `mrt pkg create` command will guide you through this process. After creating your new packages, simply call the build command with the IDE flag to make the packages visible in your IDE.

You can also add an exemplary ROS node with the `mrt pkg create_executable <node_name>` command. After creating your node with this command, new files are generated according to the default package structure described in section III-D.

In case you want to know how to implement basic functions such as image processing, dealing with `tf2`, subscribers or services, please refer to the demo packages `path_provider_ros_tool`, `lateral_control_ros_tool`, `longitudinal_control_ros_tool` and `demo_image_processing_ros_tool` available on gitlab.

G. Run On Private PC

Although we recommend to work on the MRT-Pool-PCs, it is also possible to setup up the needed environment on your private PC or a virtual machine (e.g. in VirtualBox). First of all you need an installation of Ubuntu 16.04 (or any of its derivatives, e.g. Xubuntu or Lubuntu) [**?**].

Now you can install ROS Kinetic Kame. Just follow the official ROS-Kinetic-Install-Guide [**?**] and install the package `ros-kinetic-desktop`.

After the installation of ROS you can now install the mrt-toolchain.

1) Download the `mrt_software` archive [**?**].
2) Unpack the archive using e.g.
   `tar -xvf mrt_software.tar.gz.`
3) Install `mrt-software`:
   `sudo dpkg -i mrt-software*.deb`
4) Install `mrt-build-tools`:
   `sudo dpkg -i mrt-build-tools*.deb`
5) Resolve dependencies:
   `sudo apt-get -f install`
6) Update packages:
   `sudo apt-get upgrade`

Note: The qtcreator version shipping with Ubuntu 16.04 is broken. You can download a newer – not broken – version from the official QtCreator website [**?**]. To be able to execute this binary, you need to make it executable first, by executing `chmod +x <filename>`.