Technische Universität München

Lehrstuhl für Steuerungs- und Regelungstechnik

Prof. Buss

Technical autonomous systems (TAS)

Introduction to the hard- and software for the course

Contents

1 Overview

The following chapter will give a quick overview of the hard- and software which is currently used on the remote-controlled car. The whole system can be seen in picture 1.1

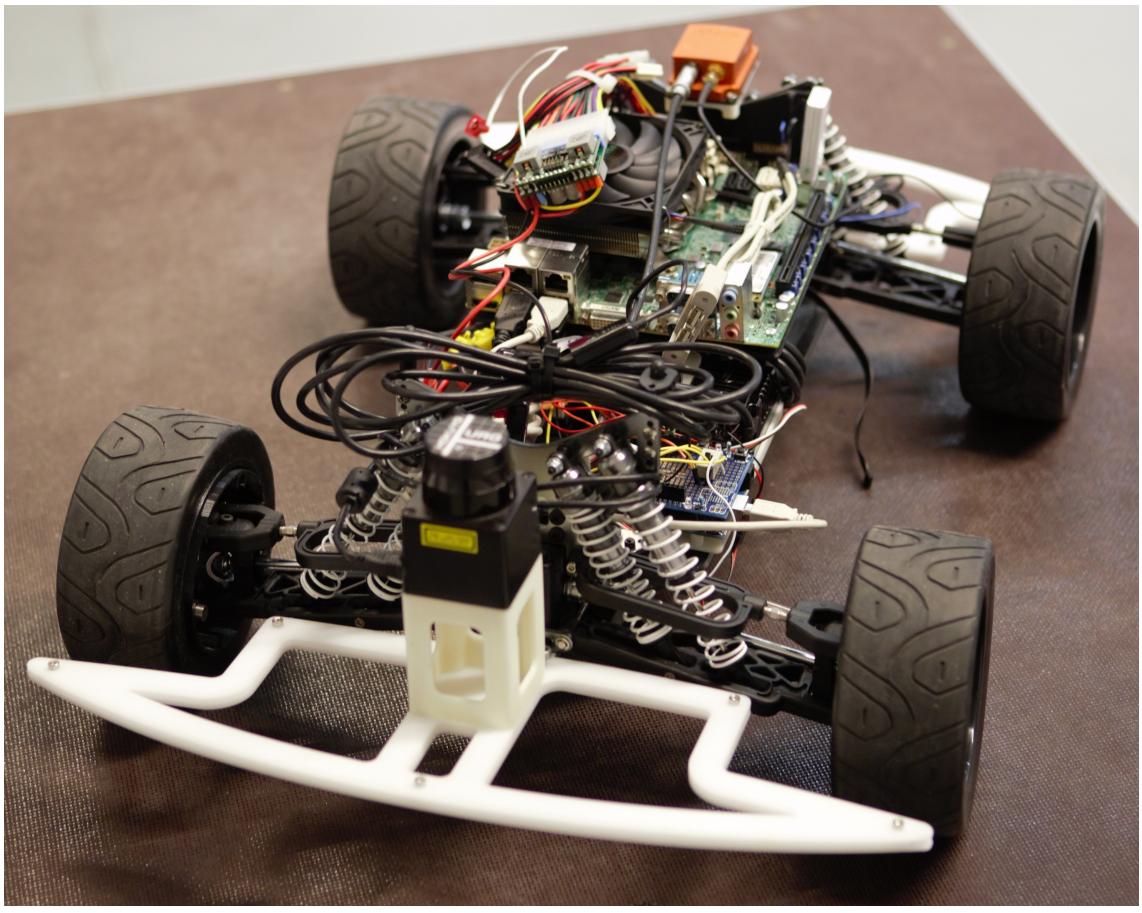

Abb. 1.1: Picture of the rc-car

1.1 Hardware

This section will give a short introduction to the hardware which can be found on the car.

1.1.1 Motors and controllers

On the car we can find two electrical motors, one for driving the car and one for the steering.

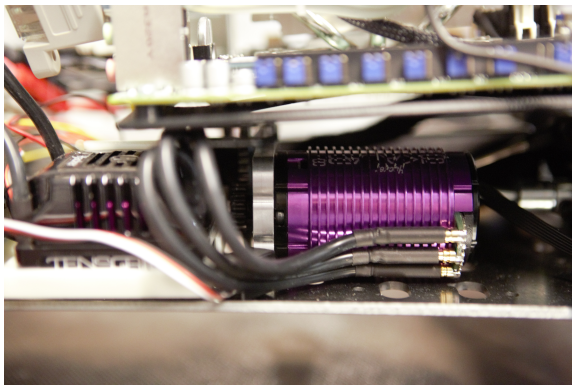

Abb. 1.2: HACKER Skalar 8 brushless motor

In picture 1.2 we can see the electrical brushless motor which is used to drive the wheels.The motor is connected to the wheels over a gear and two differentials. Table 1.1 gives some details of the used type. More information can be found here:

http://www.hacker-carline.de/produkte/skalar-motoren/hacker-skalar-10/

For steering, there is a second motor at the front of the car (see picture **??**).

Both motors bring its own controllers which take a PWM-signal as input to set the desired revolution rate or angle. The principal is shown in picture 1.3. Both input signals are provided by a little Arduino board (see next section 1.1.2).

Name:	HACKER Skalar 8 1750
max. Power (W):	1650
max. RPM (1/min):	25900
max. voltage (V):	14.8
max. current (A):	110

Table 1.1: Hacker Skalar 8 - Details

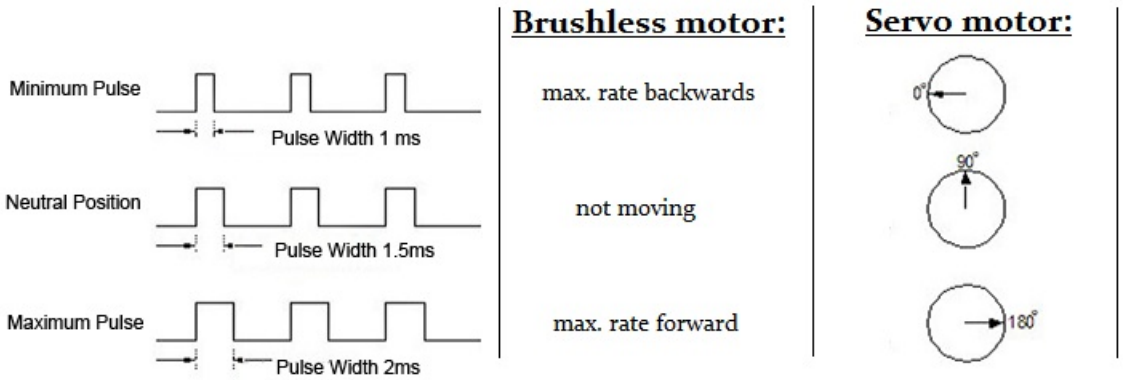

Abb. 1.3: PWM-signals for the motors

1.1.2 Arduino board

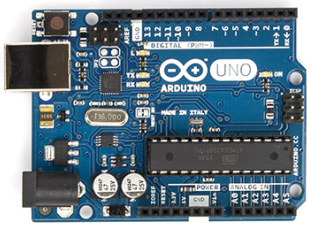

Abb. 1.4: Arduino UNO board

To provide the signals to the motor controllers there is a little Arduino UNO board on the car which can be seen in picture 1.4. It receives the PWM-values from the mainboard of the car via USB and sets them to the output pins which are connected to the controllers.

1.1.3 Linux-Board

For computing there is a Supermicro X10SLV motherboard with a Intel Core i7 processor on it. To control the system we use Ubuntu 14.04 and a special software framework called ROS (see section 1.2). Table 1.2 gives a short overview of the computer system. More information can be found on:

http://www.supermicro.com/products/motherboard/Core/H81/X10SLV.cfm

Motherboard:	Supermicro X10SLV
Processor:	Intel Core i7 8 x 3GHz
Memory:	16GB DDR3
	128GB Flash memory

Table 1.2: Computer system - Details

1.1.4 Laserscanner

Abb. 1.5: Hokuyo URG-04LX-UG01

At the front of the car we can find a Hokuyo URG-04LX-UG01 laserscanner (see picture 1.5). The scanner is used for location in a given 2-D map or to create maps while the car is moving through the environment. For detailed information about the scanner refer to:

https://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx_ug01.html

Name:	Hokuyo URG-04LX-UG01
Measuring area:	20 to 5600mm(white paper with 70mm×70mm), 240°
Accuracy:	60 to 1,000mm : ±30mm,
	1,000 to 4,095mm : ±3 percent of measurement
Scanning time	100ms/scan

Table 1.3: Laserscanner - Details

1.1.5 Inertial measurement unit (IMU)

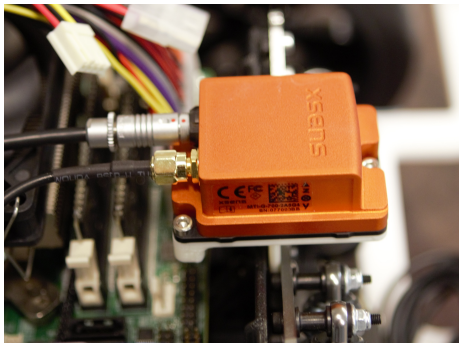

Abb. 1.6: XSens MTi-G-700

At the back of the car there is an X-Sens MTI-G-700 Inertial Measurement Unit (IMU). It contains sensors for accelerations in every spatial direction, for angle velocities and orientation about each axis and a GPS-module. Table 1.4 gives some details of the module. For more information see:

http://www.xsens.com/products/mti-g-700/

Name:	XSens MTi-G-700
Output frequency:	Up to 2000Hz
Standard full range gyro:	450 °/s
Standard full range acc	50 m/s2
In-run bias stability gyro	10 °/h

Table 1.4: IMU - Details

1.2 Software

The operating system currently running on the linux board of the car is Ubuntu 14.04. To work with the hardware there is installed a software framework called Robot Operating System (ROS) which provides a big number of libraries and tools to work with robots. On the car we use the currently newest version: ROS Indigo [1].

At the beginning of the course you should get familiar with the functionality of ROS. A good start are the official tutorials on:

http://wiki.ros.org/

After working with the tutorials you should understand the concept of nodes and topics and be able to create and build your own package. To work with the car

[1] Since ROS Fuerte there are two different build systems. We use the catkin build system

there is a package on GitHub you can download and install on your system. See the next chapter for more information.

2 Set up the hardware

The following chapter will show you how to connect the car to different power supplies.There are two possibilities to power the car electrically:

- Power from battery:
  Of course the whole car can be powered by battery to drive it around wireless.

- External power source:
  The Linux Board can also be powered by an external power source for long programming and debugging sessions. This avoids data loss in case of an empty battery. If only the board is supplied and the brushless motor is disconnected from the power supply the battery will last for about one hour.

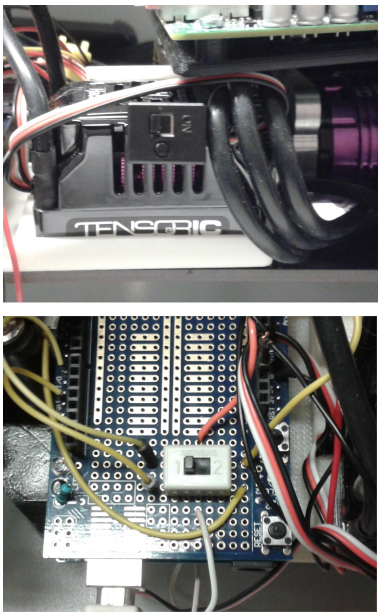

Abb. 2.1: Switches for brushless motor

Power supply for the brushless motor can disconnected by a switch, which you can see in picture **??** on the top. With the switch shown on the bottom you can select the source of the input signals for the brushless motor controller:

- Position 1:
  In this position the input signal is taken from the ROS `/servo` topic (see section **??** for more information).

- Position 2:
  In this positon the input signal is taken from the receiver for the original remote controller (see section **??** for more information).

2.1 Power from external power source

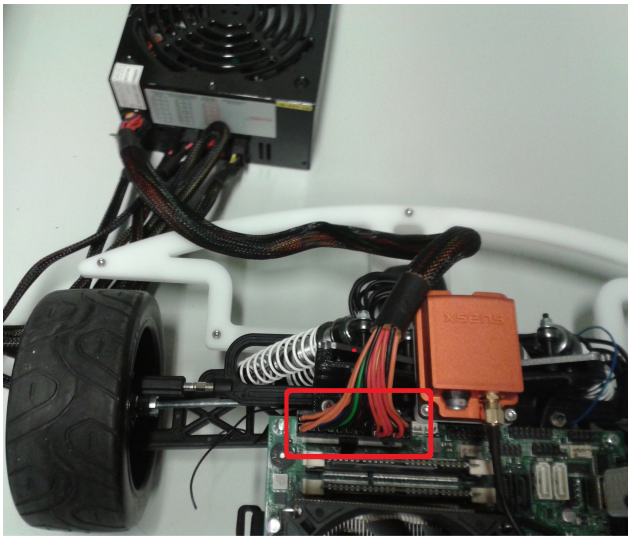

Abb. 2.2: Connection to the external power source

To power the board externally just connect the large 24-pin connector like it is show in picture 2.2. Pay attention: This will not supply the motors. These can only be powered by the battery.

2.2 Power from battery

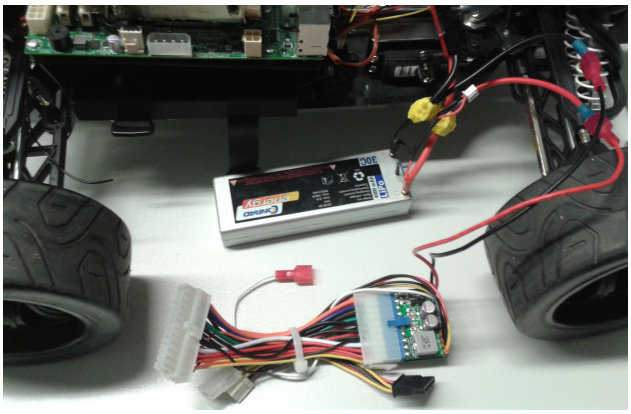

Abb. 2.3: Connection to battery

To supply the board from battery, there is a 12V DC-DC Converter which you can see in picture **??**. Just connect the battery and the converter like in the picture and the plug the 24-pin connector into the board.

3 The TAS-package

This chapter will show you how to install the TAS-package and get the car running step-by-step. It will give you a short desription of each of the different components. The package contains several self-written nodes and launchfiles to run the hardware (Laserscanner, IMU, Arduino) and the so called Navigation Stack properly on the car. The Navigation Stack provides tools for high level tasks like path planning and localization. The TAS-package also brings some shell scripts to make the work quicker and more comfortable. The folder structure can be seen in picture 3.1.

Most of the files we need in this tutorial are found in the folder `tas/launch`.

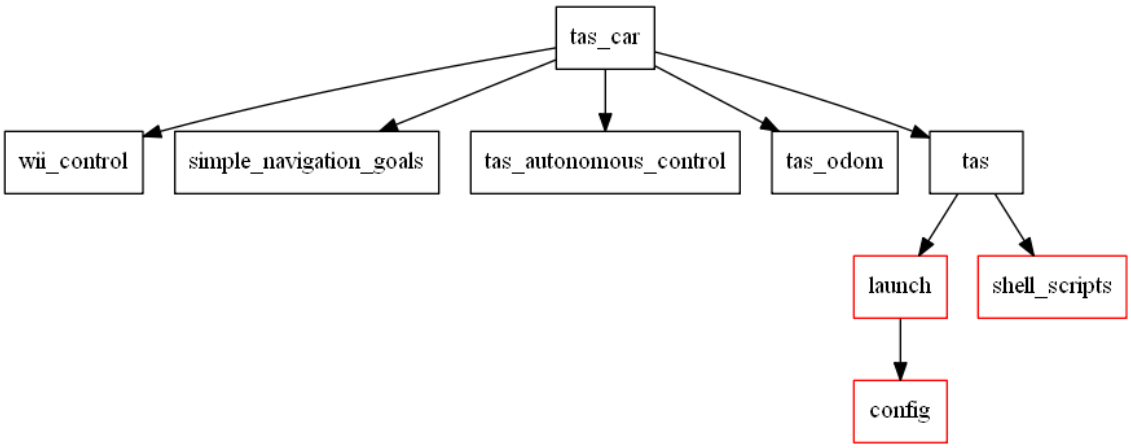

Abb. 3.1: Folder structure of the TAS-package

3.1 Installing the TAS-package

As it was mentioned in the previous chapter the TAS-package can be found on GitHub. To install it open a terminal and switch to your catkin workspace directory. In this example we assume your working directory in `~/catkin_ws`. To get and build the package on your car type in:

```
# cd ~/catkin_ws/src
# git clone https://github.com/LSR-TAS/tas_car
# cd ..
# catkin_make
```

If there occured no errors, the package has been built and the car is now ready to run. Before running anything you should also do the following preparations to make your work in the terminal a little more comfortable:
Open a terminal and type in:

```
# sudo gedit ~/.bashrc
```

The `.bashrc` file is called everytime you open a terminal in Ubuntu. Each command you add to this file will be executed automatically when you open a new terminal. Add the following lines to the end of the textfile[1]:

```
PATH=~/catkin_ws/src/tas_car/tas/shell_scripts:"$PATH"
source ros_setup
```

TODO: Add ros/opt..../setup.bash to ros_setup file, and change file extension!

This will take the the following effects:

- Add script folder:
  The first command will add the `shell_scripts` folder (see picture 3.1) to the standard pathes. After that it is possible to run files from this folder without giving the full path. You can place your own shell scripts here to run them more quickly.

- Set USB access rights:
  The second command runs the `ros_setup` file from the `shell_scripts` folder. It mainly sets the correct access rights for the USB ports for the laserscanner and the Arduino Board.

[1] Eventually you have to change the path of your catkin workspace directory

3.2 Hardware drivers

The follwing section will show you how to run the hardware drivers on the car. After launching them it is already possible to drive the car around and control it manually with the wii controller.

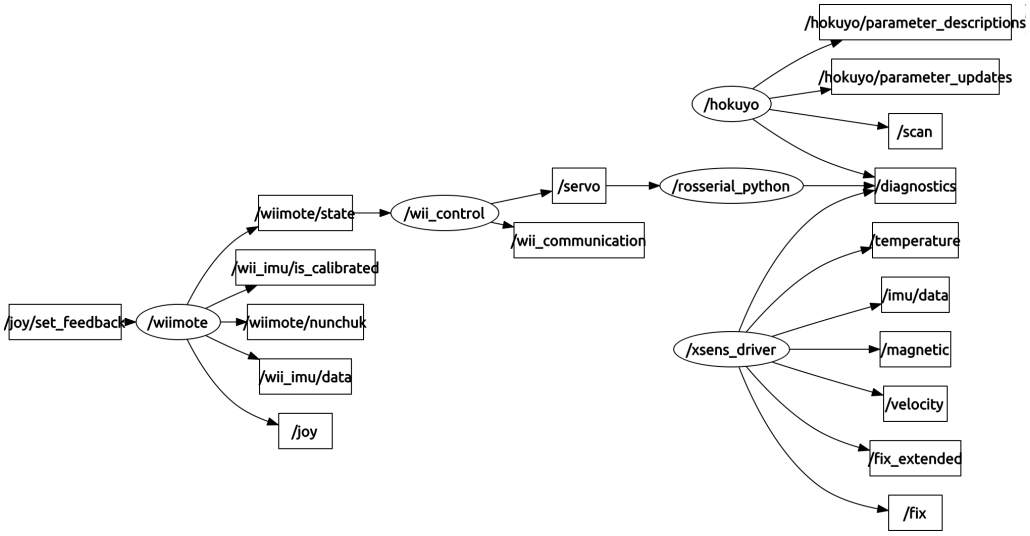

Abb. 3.2: ROS nodes for the hardware

To launch the hardware drivers open a terminal and type in:

```
# roscd tas/launch
# roslaunch hardware.launch
```

Abb. 3.3

This will run several nodes to communicate with the hardware which will be explained in detail in the following sections. To establish the connection to the wii-controller press the the two two buttons at the bottom of the controller (see picture 3.3 on the right) for about 5 seconds until you see a notification in the terminal window. **If the motors are powered (see section 2), the car is now ready to run!**

In picture 3.2 you can see a graphical overview of the running nodes and topics. If you open a terminal and type in:

```
# rosrun rqt_graph rqt_graph
```

you should get the same picture.

3.2.1 wiimote

The `wiimote` node establishes a bluetooth connection to the wii-controller. It reads the inputs from the sticks and buttons of the

controller and publishes these information to several ROS topics. Like you certainly have noticed, we use this data as control inputs for the car in manual mode. See the following link for a more detailed description:

http://wiki.ros.org/wiimote?distro=indigo

3.2.2 wii_control

The motor controllers require a special PWM signal as control inputs (see picture 1.3). The `wii_control` node takes the inputs from the wii-controller, calculates the corresponding PWM values and publises them to the `/servo` topic.

3.2.3 rosserial_python

The `rosserial_python` node takes the PWM values from the `/servo` topic and sends them to the arduino via the USB port. It mainly implements the communication channel between nodes and topics over a (virtual) serial port. For correct communication there has to be installed the so called `rosserial_client` on the arduino. For more information refer to:

http://wiki.ros.org/rosserial

3.2.4 hokuyo_node

The `hokuyo_node` is the driver for the laserscanner. It reads the measured data from the scanner via USB and publishes the to the `/scan` topic.

http://wiki.ros.org/hokuyo_node

3.2.5 xsens_driver

The package provides a driver for several different IMUs from the XSens company. It reads data from the sensor via USB and publishes it to the `/imu/data` topic just like the `hokuyo_node` does with the laser data.

http://wiki.ros.org/xsens_driver

3.3 TAS-Odometry

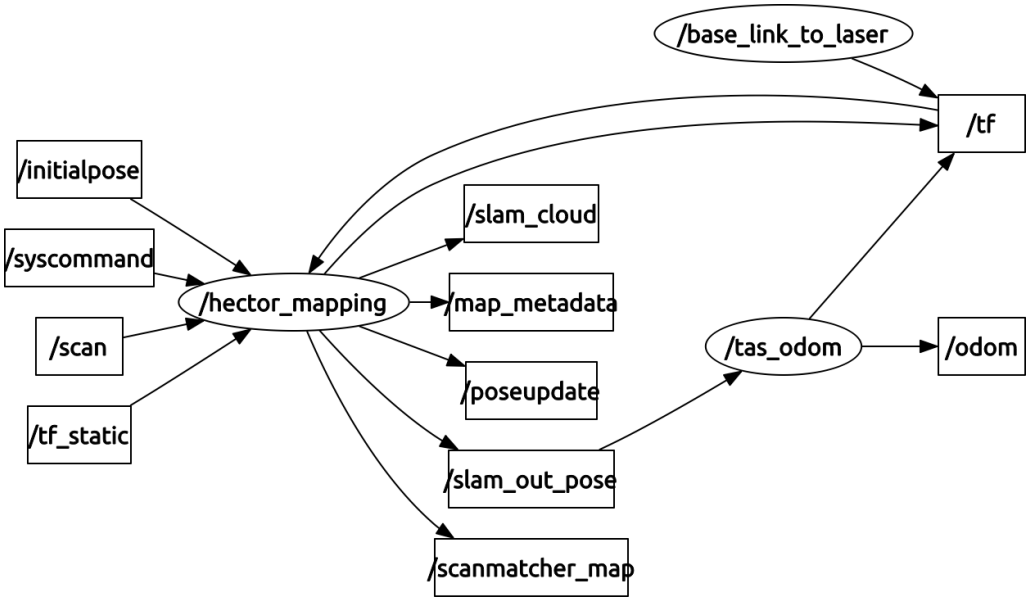

Abb. 3.4: Overview of the TAS-Odometry

Currently there is no real odometry on the car available. For the navigation packages though odometry data is required (see picture 3.6). This was solved by implementation of a "fake-odometry". It uses position and orientation estimated from the laserscanner data, calculates the velocities and provides it as odometry data.

To launch the fake odometry open a terminal and run the odom.launch file:

```
# roscd tas/launch
# roslaunch odom.launch
```

Picture 3.4 shows the running nodes and topics for the fake odometry.

3.3.1 hector_mapping

The hector_mapping node implements a SLAM-algorithm [2] to build a map and localize the car in it simultaneously at runtime. In picture 3.5 you can see the building process. It shows the created map at three different timesteps.

Unlike other solutions for localization (see the `amcl package` in section 3.5.2) it only needs data from the laserscanner and does not require any odometry inputs. This is the reason why we use it to provide the fake odometry.

[2] Simultaneous Localization and Mapping

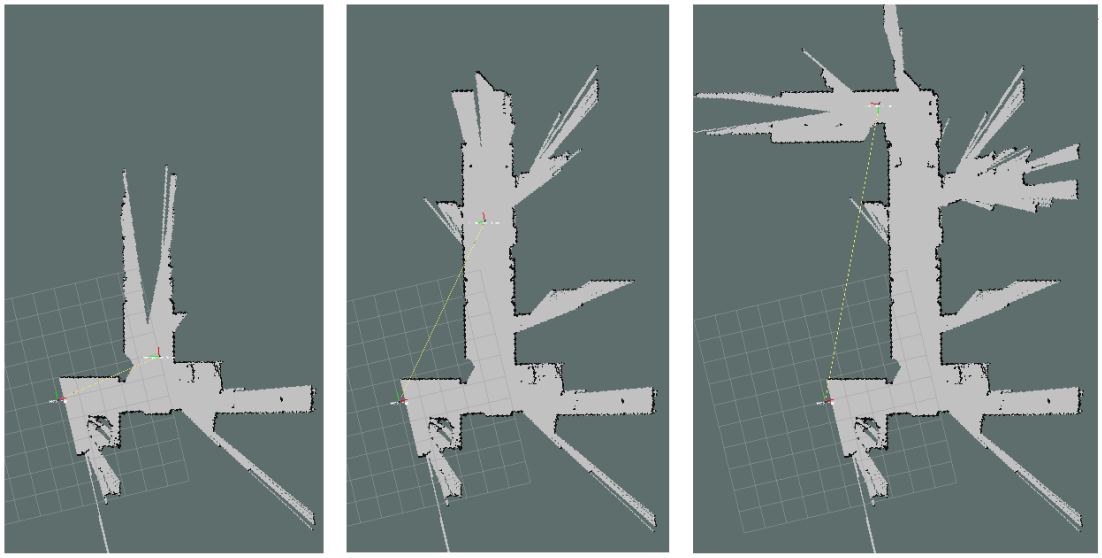

Abb. 3.5: Building process of a map with hector_mapping

The estimated position and orientation is published to the `/poseupdate` (with covariance) and the `/slam_out_pose` topics (without covariance). These information are later used to calculate the velocity of the car (see next section 3.3.2).

For correct position estimation the node needs information about the relation between coordinate frame of the laserscanner and the coordinate frame of the car (refer to **??** for an introduction to transformations and frames).

For more details about the parameters and settings of the package see:

http://wiki.ros.org/hector_mapping

3.3.2 tas_odom

The tas_odom node subscribes to the `/slam_out_pose` topic (see previous chapter 3.3.1) and calculates the velocity from the estimated position by determining the difference quotient. Position, orientation and velocity are then published to the `/odom` topic. See the following page for information about the content of odometry messages:

http://docs.ros.org/indigo/api/nav_msgs/html/msg/Odometry.html

Additionally it publishes the transformation from the odometry coordinate frame to the frame of the car. See the next section for more details about transformations.

TODO: Velocity has to be in base_link frame/tas_odom publishes twist with velocity in odom frame!

3.4 Transformations

3.5 Navigation

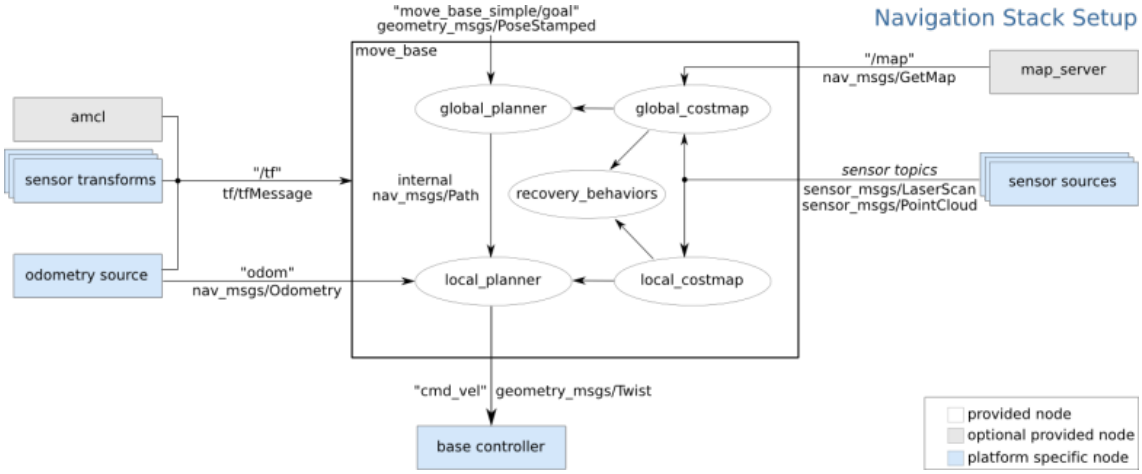

Abb. 3.6: Basic Navigation Stack

In ROS in general the Navigation Stack takes sensor inputs from odometry, laser and other sources and outputs velocity commands to reach a given goal. Picture 3.6 shows the parts of the basic Navigation Stack which is installed on the car.

To run the Navigation Stack open a terminal and type in:

```
# roscd tas/launch
# roslaunch move_base.launch
```

The launch file will run all necessary parts of the Stack. Keep in mind that the hardware drivers and the TAS-Odometry package have to be launched before (see **??** for hardware drivers and **??** for odometry).

To specify a goal for the car you can use the visualization tool RVIZ. Open a terminal and type in:

```
# roscd tas/launch
# rosrun rviz rviz -d config/rviz/tas_rviz.rviz
```

This will run rviz with the configuration file in the `config` directory. At the beginning you have to give an initial estimation of the position to the `amcl node`. After that you can give goals to the `move_base node` which the car will try to reach (see the marked buttons in picture 3.7). To actually drive the car you have to push the `C-button` on the wii-nunchuk.

Alternately you can launch hardware drivers, TAS-Odometry, the Navigation Stack and the RVIZ tool at once by launching the `run_rviz.launch` file.

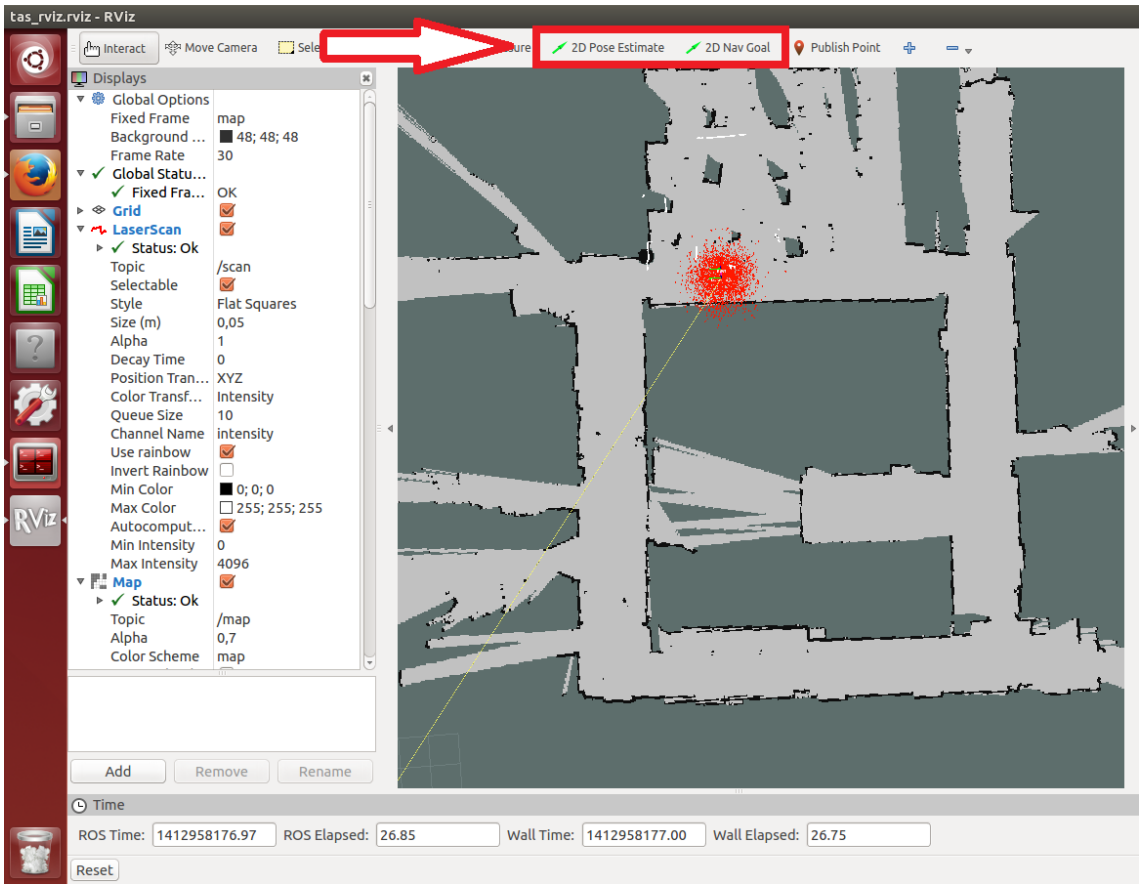

Abb. 3.7: RVIZ visualization tool

3.5.1 map_server

The `map_server` node takes a 2-D map of the environment and provides it as a ROS-service to other nodes. For every map there has to be a configuration file. The map file and the configuration file are located in the `config` directory of the TAS-package (see picture **??** for the folder structure). The path of these files can be changed in the launchfile. For more information about the parameters of the `map_server` package refer to:

http://wiki.ros.org/map_server

3.5.2 amcl

The `amcl` node implements different Monte Carlo localization algorithms to estimate the position of the robot. It requires a laserscanner, odometry and a map of the environment. The position and orientation with it's covariances is published to the `/amcl_pose` topic.

The node takes odometry data in form of the transformation between the odom-

etry frame and the coordinate frame of the car. See section **??** for information about the `tf package`. Refer also to the package documentation on:

http://wiki.ros.org/amcl

3.5.3 move_base

The `mode_base` implements a local and a global planner to calculate velocity commands to reach the navigation goal. These commands are published to the `/cmd_vel` topic which is a general interface to the hardware of the robot. In our case the commands are taken by the `autonomous_control` node to get the car moving (see next section 3.5.4).

In the `config` folder (see picture 3.1) there are several configuration files for the planners. For information about settings and parameters refer to:

http://wiki.ros.org/move_base?distro=indigo

3.5.4 tas_autonomous_control

The `autonomous_control` node mainly takes the velocity commands generated by the `move_base` node and publishes the corresponding PWM values for the motor controllers to the `/servo` topic.

For safety reasons the car only drives autonomously if the `C-button` on the wii-nunchuk is pressed.

Pay attention, that currently there are only two discrete velocity states implemented. See the following extract from the source code of the node:

```
if(autonomous_control.cmd_linearVelocity>0) {
        autonomous_control.control_servo.x = 1550;
}

else if(autonomous_control.cmd_linearVelocity<0) {
        autonomous_control.control_servo.x = 1300;
}

else {
        autonomous_control.control_servo.x = 1500;
}

autonomous_control.control_servo.y =
 autonomous_control.cmd_steeringAngle;
```

4 Additional topics

4.1 Shell scripts

4.2 External RVIZ

4.3 VNC

4.4 Recording and playing data

4.5 Integrated Development Environments (IDE's)

4.6 Version control with git

4.7 Arduino code

TODO: Put arduino code in repository!

4.8 Switching remote controller

Bibliography

Zu erledigen: