

1. CPM Lab Documentation	2
1.1 Lab Setup	3
1.1.1 Virtual Lab Setup	4
1.1.2 Physical Lab Setup	6
1.1.2.1 Overview	7
1.1.2.2 Indoor Positioning System Setup	8
1.1.2.3 Video Recording Setup	10
1.1.2.4 Map Setup	11
1.1.2.5 NUC Setup	12
1.1.2.5.1 Setup Using a NUC Image	13
1.1.2.5.2 Setup Without a NUC Image	14
1.1.2.6 Vehicle Setup	23
1.1.2.6.1 PCB Assembly	24
1.1.2.6.2 Vehicle Assembly	32
1.1.2.6.3 Raspberry Pi Setup	69
1.1.2.6.4 ATmega Flashing	72
1.1.2.6.5 Current Servo Setup	75
1.1.2.7 Cloning Disks with Clonezilla	76
1.1.3 Doxygen Setup	79
1.1.4 System Test	81
1.2 Getting Started	82
1.2.1 Lab Software Preparation	83
1.2.2 Lab Control Center Introduction	84
1.2.3 Vehicle Commands	85
1.2.4 Physical Experiments	87
1.2.5 Running an HLC	89
1.3 Read More	90
1.3.1 Indoor Positioning System	91
1.3.2 Lab Control Center	95
1.3.2.1 Map View	96
1.3.2.2 Vehicle & NUC Monitoring	98
1.3.2.3 Setup (Tab) (WIP)	99
1.3.2.4 Manual Control (Tab)	102
1.3.2.5 Parameters (Tab, Menu, XML)	103
1.3.2.6 Timer (Tab)	105
1.3.2.7 Logs (Tab)	107
1.3.2.8 Commonroad (Tab)	110
1.3.2.9 Distributed Deployment (NUCs)	116
1.3.2.10 Integrated Debugging Tools	119
1.3.3 Modules	122
1.3.3.1 CPM Library	123
1.3.3.2 Command Line Parser	124
1.3.3.3 Parameter Service	125
1.3.3.4 Timer - Periodicity and Synchronization	126
1.3.3.5 Visualization - Used in the LCC	131
1.3.3.6 Commonroad Information for the HLC	133
1.3.3.7 DDS Reader and Writer	134
1.3.3.8 DDS Participants and Topics	136
1.3.3.9 RoundTripTime	137
1.3.3.10 Logging	138
1.3.3.10.1 Logging Structure	139
1.3.3.10.2 Logging to DDS	141
1.3.4 Monitoring (eProxima)	142
1.3.5 Middleware Usage	143
1.3.6 Matlab + RTI DDS	145
1.3.7 Vehicle	151
1.3.7.1 Vehicle Overview	152
1.3.7.2 Equations	155
1.3.7.3 LiPo Charging	156
1.3.8 Software structure	159
1.3.8.1 High Level Controller	160
1.3.8.2 Middleware Structure	161
1.3.8.3 Mid Level Controller	163
1.3.8.4 Low Level Controller	165
1.3.9 HLC Examples	166
1.3.9.1 Basic Circle Example	167
1.3.9.2 Central Routing Example	170
1.3.10 Computation Hardware	174

CPM Lab Documentation



An Open-Source Testbed for Connected and Automated Vehicles

CPM Lab
GitHub
Remote
Access
Publications

This documentation provides technical information in detail of the [Cyber-Physical Mobility \(CPM\)](#) Lab at the [Chair of Embedded Software](#) at [RWTH Aachen University](#). In our Lab it is possible to test and evaluate strategies for connected and automated vehicles.

In this documentation we show you how to setup your environment for [simulation only](#) as well as for [experiments](#). We provide [tutorials](#) to get you started with the CPM Lab. You can [read more](#) to get a deeper look into special topics.

As you can see in the picture the Lab consists of the [Main PC](#) which provides the access to the infrastructure. It processes information from the [Indoor Positioning System \(IPS\)](#) and information from our user interface called the [Lab Control Center \(LCC\)](#). You can upload your program in the LCC which will send the commands to the vehicles [Data Distribution Service \(DDS\)](#) of the [RTI Connex](#) which works similar to a cloud system by providing all information to all members of the service. All vehicles and NUCs participate in the DDS as well. The NUCs process the commands and execute your program. In each NUC a local DDS provides the information with the middleware. Once all data is evaluated it is shared with the central DDS and sent to the vehicle.

Software architecture

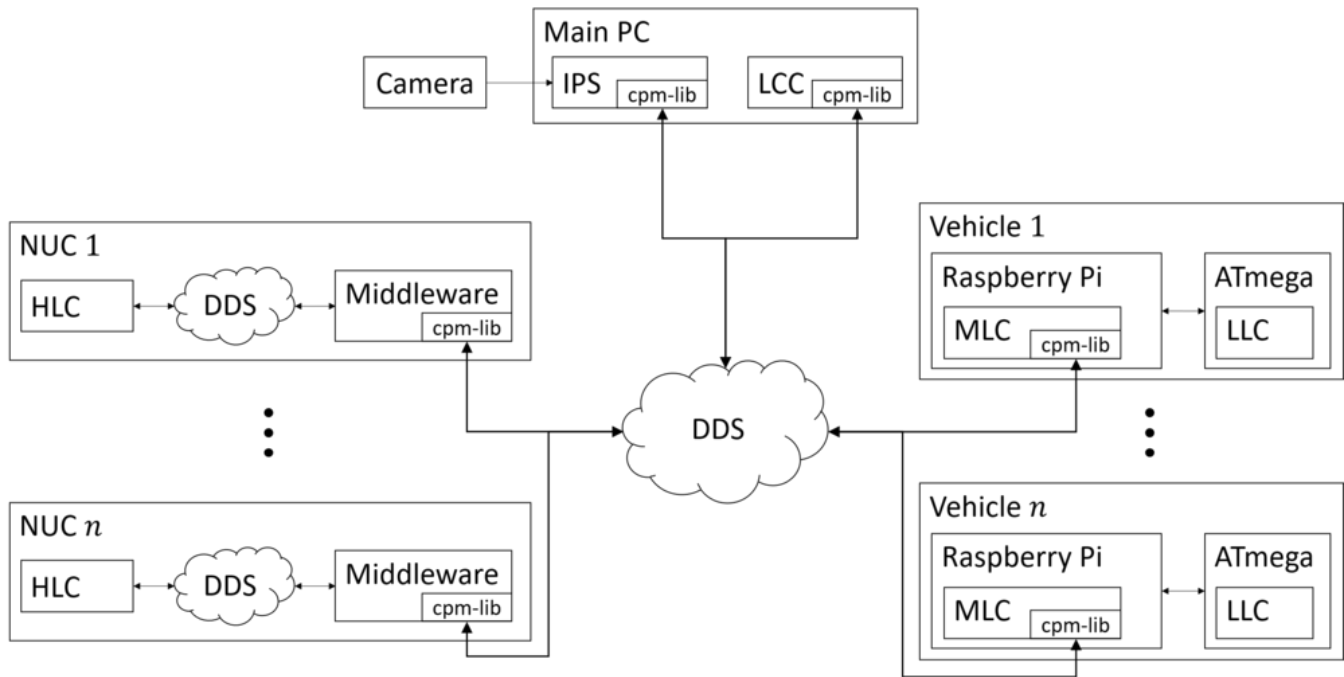
The software for each vehicle is distributed between three major components:

- The [High Level Controller](#) runs on an Intel NUC PC, which is logically associated with a particular vehicle, but located in a shelf next to the track. The HLC is responsible for decision making and coordination with other HLCs.
- The [Mid Level Controller](#) runs onboard the vehicle on a Raspberry Pi Zero W, which is connected via WiFi. The MLC is responsible for accurately following the reference trajectory given by the HLC.
- The [Low Level Controller](#) runs onboard on an ATmega2560. It is responsible for the vehicle hardware interaction and abstraction.

Two supporting software components run on a separate "Master PC":

- The [Indoor Positioning System](#) measures the vehicle poses using a camera.
- The [Lab Control Center](#) gives the user an overview of the current operational state of the lab.

All software components use our [CPM library](#) (cpm-lib). The CPM-lib provides functions to communicate with the vehicle and the IPS system. You can find functions for localization and vehicle states.



- [Virtual Lab Setup](#)
- [Physical Lab Setup](#)
 - [Overview](#)
 - [Indoor Positioning System Setup](#)
 - [Video Recording Setup](#)
 - [Map Setup](#)
 - [NUC Setup](#)
 - [Setup Using a NUC Image](#)
 - [Setup Without a NUC Image](#)
 - [RT Patch](#)
 - [NTP](#)
 - [RTI DDS Setup](#)
 - [eProxima Setup](#)
 - [Matlab Setup \(optional\)](#)
 - [Vehicle Setup](#)
 - [PCB Assembly](#)
 - [Vehicle Assembly](#)
 - [Raspberry Pi Setup](#)
 - [ATmega Flashing](#)
 - [Current Servo Setup](#)
 - [Cloning Disks with Clonezilla](#)
- [Doxygen Setup](#)
- [System Test](#)

Virtual Lab Setup

This page will guide you through the installation process of the Lab:

1. Install [Ubuntu 18.04](#).
2. Create the folder '~/dev' and open it in a terminal:

```
cd ~
mkdir dev
cd dev
```

3. Clone the CPM Lab software repository:

```
git clone https://github.com/embedded-software-laboratory/cpm_lab software
```

4. Prepare to run our installation script by getting a license.dat-file for the [RTI DDS Connex Service](#) (ask your supervisor if you are a student or get into contact with [RTI](#)). This is not required if you are running the eProxima version (which is not yet present in the master branch).
5. Run our script to install all necessary components. In the *software* folder you have just cloned you find *install.sh*.

todo	full installation	simulation setup only
run	<pre>cd software/ sudo bash install.sh</pre>	<pre>cd software/ sudo bash install.sh --simulation</pre>
What will it install?	Unbuntu packages	Unbuntu packages
	Joystick/ Gamepad	Joystick/ Gamepad
	RTI DDS Connex or eProxima	RTI DDS Connex or eProxima
	RTI ARM or eProxima ARM	
	OpenCV 4.0 for the Indoor Positioning System	
	Basler Pylon 5 for the Indoor Positioning System and lab camera	

Change DDS Domain



If you want to change the DDS Domain later, change the corresponding variable DDS_DOMAIN in */etc/profile.d/rti_connex_dds.sh*.

The following domain IDs are used in the CPM Lab:

Domain ID	Purpose
21	Lab (main computer, NUCs, vehicles)
61-66	Student computers

Change RTI license file



If you want to update your license file, change the file at: */opt/rti_connex_dds-6.0.0/rti_license.dat*

6. Reboot your PC.
7. **Optional:**
Setup MATLAB. As a student from RWTH, you can do this with a MathWorks account following [these instructions](#).
For RTI DDS: [Download the DDS Support Package for MATLAB](#) and install the toolbox. The native DDS addon that ships with MATLAB versions from 2021a only supports Simulink. If you are going to code in MATLAB, install the corresponding package from the link.

To use MATLAB with eProxima, [look here](#).

Make sure to create a symbolic link to the MATLAB binary during or after the setup!

8. Run the build script.

full setup	only simulation setup
<pre>./build_all.bash</pre>	<pre>./build_all.bash --simulation</pre>

Troubleshooting



- If you
 - encounter unexpected errors
 - have problems during the building process
 - receive an error regarding timing issues of unittests

make sure that the software is up to date and built correctly with

```
cd ~/dev/software/  
git pull  
git clean -xdf  
./build_all.bash
```

- If you ever run `build_all.bash` as root , you might run into a lot of `permission denied` errors. You need to delete all files and folders that were created and rerun as your usual user.
- If you receive an error like `"dds/someheader.h" not found`, run the command `"rtiddsgen"`, which should build the necessary cpp-files

Physical Lab Setup

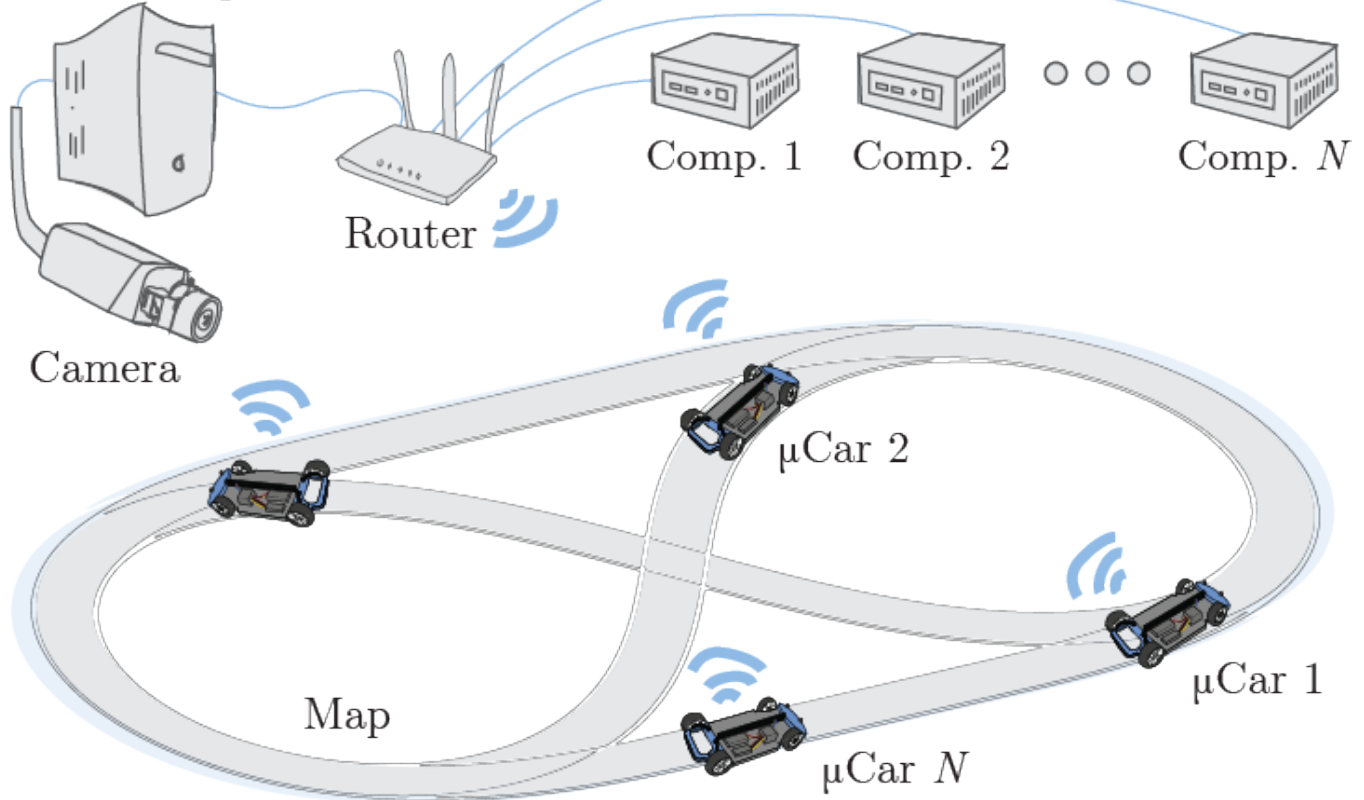
This section describes the hardware for a full physical lab setup. To setup the main PC please follow the instructions in [virtual lab setup](#).

- [Overview](#)
- [Indoor Positioning System Setup](#)
- [Video Recording Setup](#)
- [Map Setup](#)
- [NUC Setup](#)
 - [Setup Using a NUC Image](#)
 - [Setup Without a NUC Image](#)
 - [RT Patch](#)
 - [NTP](#)
 - [RTI DDS Setup](#)
 - [eProxima Setup](#)
 - [Matlab Setup \(optional\)](#)
- [Vehicle Setup](#)
 - [PCB Assembly](#)
 - [Vehicle Assembly](#)
 - [Raspberry Pi Setup](#)
 - [ATmega Flashing](#)
 - [Current Servo Setup](#)
- [Cloning Disks with Clonezilla](#)

Overview

The lab hardware is connected as follows:

Main Comp.



The [IPS](#) camera is connected to the [main computer](#) via USB.

The [main computer](#), the [video recording](#) camera, and the [NUCs](#) (computer 1-N) are connected to the router via LAN.

The [vehicles'](#) Raspberry Pi are connected to the router via WLAN.

We use a [Linksys EA9500](#) router.

Indoor Positioning System Setup

Camera Specifications

A [Basler camera](#) is used for the indoor positioning system in the lab.

The system was originally intended to work in the infrared. The NIR sensor was selected for that reason. Due to eye safety concerns the system was switched to visible light.

The camera is mounted at a height of 3.3 meters to overlook the complete [map area](#). It is centered to the map and orientated straight downwards. In order to simplify the positioning of the camera, we installed three aluminum profiles in an H-form to be able to move the camera along the x- and y-axis.

Model Number	acA2040-90umNIR
Camera Sensor Format	1"
Pixels (H x V)	2,048 x 2,048
Type	NIR Camera
Camera Family	ace
Conformity	USB3 Vision, GenICam
Dimensions (mm)	29.3 x 29.0 x 29.0
Exposure Control	Freely Programmable
Frame Rate (fps)	90
Imaging Device	CMOSIS CMV4000 NIR-enhanced
Input/Output Options	1 Opto-isolated Input/Output, 2 Fast-GPIO Ports
Manufacturer	Basler
Mount	C-Mount
Operating Temperature (°C)	0 to +60
Pixel Size, H x V (µm)	5.5 x 5.5
Power Requirement	Via USB 3.0 interface
Synchronization	Via External Trigger, or Free-Run
Type of Sensor	Progressive Scan CMOS
Type of Shutter	Global
Video Output	USB 3.0

Calibration

A PVC canvas of the size 4.0 x 4.5 m with a 0.1 x 0.1 m checkerboard pattern printed on it is used to calibrate the IPS. This pattern is also used to define the coordinate system in the lab.

Compile and test

```
cd ~/dev
git clone https://git.rwth-aachen.de/CPM/Project/Lab/software.git
cd software
cd ips2
./build.bash
```

To test that the IPS works, open two terminals. In the first, run

```
./build/BaslerLedDetection --visualization=true
```

Switch a vehicle on and place it in front of the camera. The flashing LEDs should be highlighted in the visualization.

In the second terminal, run

```
./build/ips_pipeline --visualization=true
```

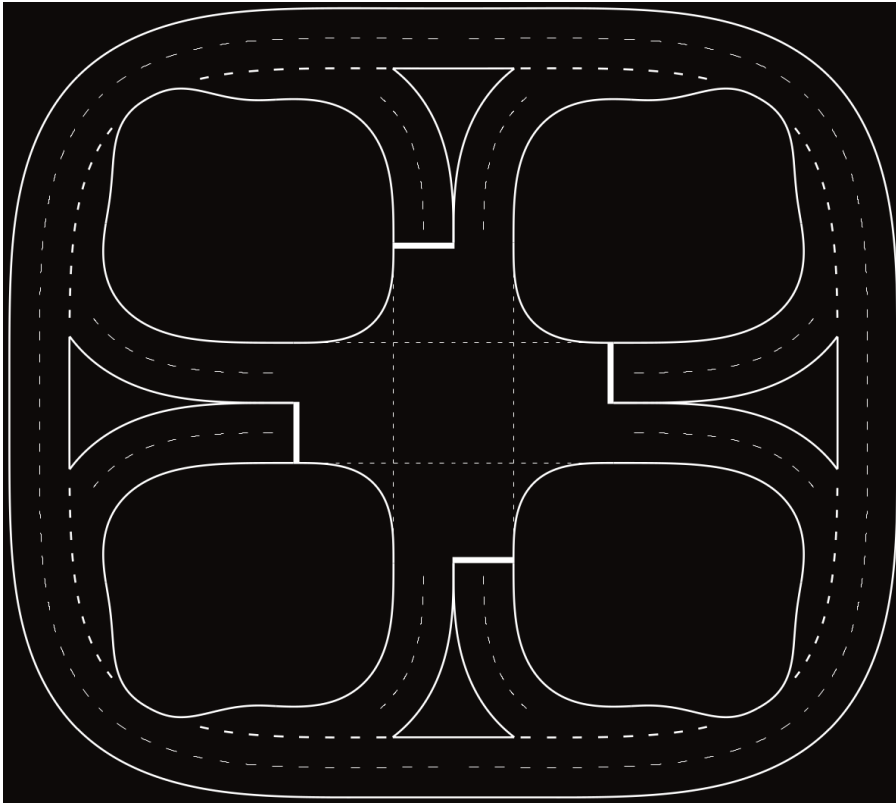
The detected LEDs could also show up in this second visualization.

Video Recording Setup

A [Basler camera](#) (acA1920-50gc) records videos of the experiments.

Map Setup

On the floor of the lab is a 4.0 x 4.5 m PVC canvas with a map printed on it. Its purpose is for visualization and presentation only. It has no direct effect on the vehicle or system behavior.



NUC Setup

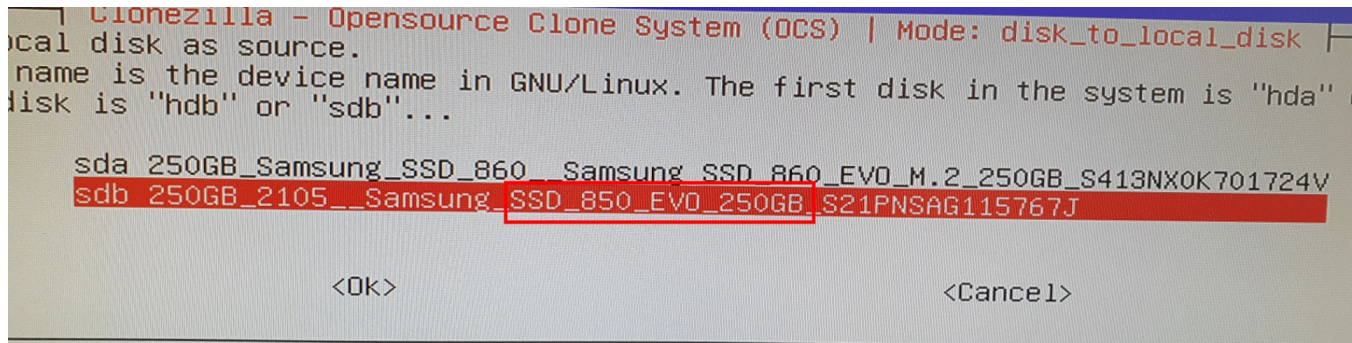
- [Setup Using a NUC Image](#)
- [Setup Without a NUC Image](#)
 - [RT Patch](#)
 - [NTP](#)
 - [RTI DDS Setup](#)
 - [eProxima Setup](#)
 - [Matlab Setup \(optional\)](#)

Setup Using a NUC Image

We already have a NUC image prepared. It was copied onto a SSD which, in combination with an USB stick with Clonezilla, can be used to 'flash' new NUCs.

Install Procedure

1. Insert both the SSD (via an USB adapter) and the stick with clonezilla into the NUC and power it on.
2. Press F10 to get into the boot menu, and select the image on the clonezilla stick
3. The process of using Clonezilla has already been documented [here](#).
4. Make sure to select 850_EVO as source - this is the image on the SSD. The NUCs usually use an M2 stick.



Post-installation

- **Router settings:** You need to register the MAC address of the NUC and set a fixed IP address for it, following the mask 192.168.1.2xx, where xx corresponds to the ID of the NUC. Check if that actually worked after setting it.
- **Known hosts:** You need to connect to the NUC (ssh guest@192.168.1.2xx, no password required) once on the main computer to store it as known host (you will be requested to do so). Otherwise, uploads using "Deploy Distributed" in the LCC won't work remotely (so only locally, on the same machine, not on the NUC).
- **LCC test:** Boot the NUC (connected to the network via LAN) and see if it appears as being online in the LCC. This might take a while, but usually not more than 2 minutes.

Troubleshooting

NUC gets stuck in Ubuntu boot screen after cloning the image from another NUC using Clonezilla?

- Connect NUC to network using an Ethernet connection

NUC shows **Error: out of memory Press any key to continue...** at first startup?

- Change default choice of GRUB
 1. Restart
 2. Select **Advanced options for Ubuntu** (should be 2nd option)
 3. Try using an older Kernel; not (**recovery mode**); memorize the number **x** of the entry, counted from top to bottom, starting at 0
 4. Login as **controller**
 5. Open Terminal and run **sudo <editor of your choice> /etc/default/grub**
 6. Change the Line **GRUB_DEFAULT=0** to **GRUB_DEFAULT="1>x"** where **x** is the number of the entry you want to be chosen by default
 7. Save file and close editor
 8. Run **sudo update-grub**
 9. Restart and check whether the NUC starts without interacting with the GRUB-Menu

Setup Without a NUC Image

Setup

1. Install Ubuntu 18.04.3 LTS
2. Setup the two accounts as described below.
3. Install the [RT Patch](#) (s.t. realtime programming becomes possible)
4. [Setup NTP](#) for time synchronisation
5. Install [RTI DDS Setup](#) for network communication, or **eProsima**. The file `install_eprosima_on_nuc.bash` in the software folder of the repository should contain all vital information for this, regarding required commit IDs for the branches to work with the branches chosen in the repository for thirdparty etc. You can also use this script directly to remotely install eProsima on one or multiple NUCs.
6. Install [Matlab Setup \(optional\)](#) and the RTI Matlab Support Package
7. Install the following packages: git, tree, openssh, tmux, cmake, libgckmm-3.0-dev
8. (Optional) Install the raspbian toolchain and the Raspberry Pi RTI libraries
9. Make sure that you set up a guest account with a simple password. Generate a SSH Key and put the public Key in the LCC main computer, so that a log-in without using a password becomes possible
10. Enable the autostart script as mentioned above. The script is located in the software repository.
11. You also need to make sure that the autostart script is started after the NTP clock sync. This is done by another sudo-autostart script, which communicates with lab_autostart (see below).
12. You need to allow the guest user to restart the machine without "sudo". This is also explained below.
13. If desired, set some environment variables, e.g. the RTI location, the location of the Matlab executable etc
14. Setup the NUC's IP: 192.168.1.2xx, where xx corresponds to the NUC's ID ID 8 - 208, ID 15 - 215 etc.

Accounts

We use two user accounts on each NUC. Nevertheless, the setup can be used with only one user. The LCC will establish a connection to each NUC using `ssh ACCOUNT@IP-ADDRESS`.

Account: controller

The "root" account that can be used to install software or updates, change system settings etc. It is password-protected.

Account: guest

The account used for running the HLC scripts / software on the NUCs. For an SSH connection, no password is required as long as the local SSH key is installed on the current system. This connection is e.g. used by the [Lab scripts](#) to upload scripts, so make sure to use these scripts on a machine that owns the key. If it is not on your device, simply copy the key from the Lab's main computer: It is located in `home/USERNAME/.ssh/id-rsa` and should be copied to a similar location on the desired computer.

This account is also password-protected, but - if necessary - the password can be shared to use the account without the key-based SSH authentication.

Autostart

Enable autostart **for the guest user** (on which all remote deploy actions are performed), using `crontab -e` and insert

```
@reboot ~/autostart/lab_autostart.bash
```

This makes the HLC regularly send messages that it was turned on and allows the user to determine which NUCs are online. (Make sure that `chmod +x` was performed on the file before, and that it does not use relative file locations. Also, make sure that the directory exists and put the autostart script from `software/hlc/autostart` into the same folder. The autostart program (also from software) is downloaded from the apache web server using the autostart script, which runs on the lab's main computer - if the software has been built on it)

Domain ID: 21. Topic name: `hlc_startup`

In the given directory should be `lab_autostart.bash` as well as the executable of the autostart program which is called by the bash script

`lab_autostart.bash` - if it ever gets changed - needs to be uploaded onto each NUC afterwards, to replace the older file.

`lab_autostart.bash` uses `lab_update.bash` in a future update. If it is also part of the current repository, do not forget to put this file on the NUC as well!

Software out of date

 If the software on the NUCs is not up-to-date, check in `/var/www/html/nuc` on the main PC if all (four) packages downloaded by `lab_autostart.bash` on the NUCs are provided (and if they are up-to-date)

Upload not working

 If the upload to the NUCs is not working, check that their fingerprint has already been accepted (needs to be done when no connection has been established before just manually connect via SSH to the NUCs on your PC once before using the LCC to deploy the files remotely)

NTP Sync Before Autostart

Use the controller account for this task. We will create a startup script that runs as sudo to enforce a NTP time sync before lab_autostart is started. This step is important, because all timers, including the one started in lab_autostart, rely on the system clock. If it is changed 'backwards' (into the past), the timer wakes up only when it reaches its next timestep set in the future when the clock had a higher value. This might cause it to stop its periodic execution for minutes up to hours, and is definitely undesired behavior.

The following script needs to be inserted. Read the comments above, as they explain where to put the script and how to activate it:

```
#!/bin/bash

# Place in /etc/rc.local
# The service only starts if the file is formatted properly, like this, with your code in between:

# Also, you need to make sure that the file is executable: sudo chmod +x /etc/rc.local

# Another problem: We want the lab / hlc program to only start if the time sync was finished
# This is done by using pipes between lab_autostart and the enforced time sync in rc.local

# For communication between processes, proposed solution:
# -----
# -----
# -----

# First, set a wrong date (for testing purposes)
# date +%T -s "14:14:00"

# Ping to make sure that an internet connection is available
# Write to /dev/null to suppress output
until ping -c1 ntp1.rwth-aachen.de >/dev/null 2>&1; do sleep 0.1; done

# Now enforce time sync
service ntp stop
ntpd -gq
service ntp restart

# Then, setup communication with the NUC startup script
# We could also run this as another user from here
# But that would be more complicated
# (The script needs to be run from guest)
# We tell the script that we are finished with the clock sync
# It is sufficient to just create a pipe here - we will check for its existence
# in the other script using -p
nuc_ntp_pipe=/tmp/nuc_ntp_pipe
nuc_lab_pipe=/tmp/nuc_lab_pipe

mkfifo /tmp/nuc_ntp_pipe
mkfifo /tmp/nuc_lab_pipe

chmod a+rw /tmp/nuc_ntp_pipe
chmod a+rw /tmp/nuc_lab_pipe

# delete the named pipe on exit
trap "rm $nuc_ntp_pipe $nuc_lab_pipe" EXIT

# Communication: 1 to tell that time sync is done, expect 1 as answer
echo "1" > $nuc_ntp_pipe
while read ans < $nuc_lab_pipe; do
    if [[ $ans -eq 1 ]]; then
        break
    fi
    sleep 1
done

# Also just for testing purposes
echo "Done" >> /tmp/test.txt

exit 0

# The answer is given by lab_autostart, which is already automatically started (see Documentation)
```

The autostart script needs to include the following lines after the export statements. They should already be part of the master branch, so you just need to make sure that the script you put on the NUC is up-to-date:

```

# ----- WAIT FOR TIME SYNC -----
# Wait for clock sync before doing anything else, because starting any program before a clock sync would cause
# problems
# The clock sync is performed in rc.local of the sudo user
# We communicate via pipes, alternatively starting this script as another user would have been possible as well
# Check for existing pipe, wait until it has been created
nuc_ntp_pipe=/tmp/nuc_ntp_pipe
nuc_lab_pipe=/tmp/nuc_lab_pipe

while [[ ! (-p $nuc_ntp_pipe && -p $nuc_lab_pipe ) ]]; do
    sleep 1
done

# Read msg from pipe, which is sent after the time sync, then answer
# 1 -> time sync done, answer with 1 on other pipe
while read sy < $nuc_ntp_pipe; do
    if [[ $sy -eq 1 ]]; then
        break
    fi
    sleep 1
done
echo "1" > $nuc_lab_pipe
# -----

```

Restart / Sudo Change

Use the controller account for this task.

Type `sudo visudo`. Then append the following to the file:

```

# Guest user can reboot the system without password (for LCC reboot option)
guest ALL=NOPASSWD: /sbin/halt, /sbin/reboot, /sbin/poweroff

```

Repositories

- cpm library
- Lab software

RT Patch

See https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/preemptrt_setup and <https://stackoverflow.com/questions/51669724/install-rt-linux-patch-for-ubuntu> (or alternatively <https://medium.com/@patdhik/realtime-linux-e97628b51d5d>).

If the latter is not available (taken from the stackoverflow guide): After applying the patch, you must install some packages (`sudo apt-get install libncurses-dev libssl-dev`) (others will be asked for by `make` / error if they do not exist), then type `make menuconfig` and select the Fully Preemptible Kernel (RT) (*Processor type and features -> Preemption Model -> Fully Preemptible Kernel (RT)*), could be located under general settings depending on the kernel version). After that, install the Kernel and apply it:

```
make -j20
sudo make modules_install -j20
sudo make install -j20
```

```
cd /boot ls sudo update-grub
```

"To change default kernel in grub, edit the GRUB_DEFAULT value in `/etc/default/grub` to your desired kernel." Then reboot, `uname -a` should now indicate that the new kernel is used.

Make sure to install a kernel that fits the current kernel used by your distribution.

Known Problems

If ubuntu is unable to locate the `libncurses-dev` package, try again the ubuntu full-scale installation.

NTP

The Network Time Protocol (NTP) is used to synchronize the clocks of all computers in the CPM lab.

Edit the NTP configuration `sudo nano /etc/ntp.conf` and insert your NTP server. In our case, this is:

```
pool ntp1.rwth-aachen.de iburst
pool ntp2.rwth-aachen.de iburst
```

Restart NTP using `sudo service ntp restart`.

Verify that NTP works correctly by running `ntpq -p`.

Under the st (stratum) column should be at least one "1". This indicates a successful synchronization.

```
cpm@cpm-MS-7A94:~$ ntpq -p
      remote           refid      st t when poll reach  delay  offset  jitter
=====
ntp1.rwth-aache .POOL.      16 p   -   64    0   0.000   0.000   0.000
ntp2.rwth-aache .POOL.      16 p   -   64    0   0.000   0.000   0.000
0.ubuntu.pool.n .POOL.      16 p   -   64    0   0.000   0.000   0.000
1.ubuntu.pool.n .POOL.      16 p   -   64    0   0.000   0.000   0.000
2.ubuntu.pool.n .POOL.      16 p   -   64    0   0.000   0.000   0.000
3.ubuntu.pool.n .POOL.      16 p   -   64    0   0.000   0.000   0.000
ntp.ubuntu.com .POOL.      16 p   -   64    0   0.000   0.000   0.000
*ts-1.rz.rwth-aa .GPS.       1 u  281  512  377   0.486  -1.204   0.122
+ntp2.rwth-aache .GPS.       1 u  187  512  377   0.563  -1.082   0.139
```

The website <https://time.is/> is also useful for checking the clock synchronization.

RTI DDS Setup

RTI DDS is used for real-time communication between programs and devices. It implements a publish-subscribe pattern and serialization/deserialization for the messages. It is mandatory.

If you use on of the eProsima branches, take a look at [eProsima Setup](#).

Downloads

Download and extract the following files:

```
wget https://s3.amazonaws.com/RTI/Bundles/6.0.0/Evaluation/rti_connext_dds_secure-6.0.0-eval-x64Linux4gcc7.3.0.tar.gz
wget https://s3.amazonaws.com/RTI/Community/ports/toolchains/raspbian-toolchain-gcc-4.7.2-linux64.tar.gz
wget https://community.rti.com/static/downloads/connext-dds/6.0.0/rti_connext_dds-6.0.0-core-target-armv6vfphLinux3.xgcc4.7.2.rtipkg
tar xvzf raspbian-toolchain-gcc-4.7.2-linux64.tar.gz
tar xvzf rti_connext_dds_secure-6.0.0-eval-x64Linux4gcc7.3.0.tar.gz
```

A backup of these files can be found in sciebo/CPM/Project/Lab/08_Other/RtiConnextBackup.

The Raspbian toolchain and the ARMv6 libraries may be skipped, if compilation for the lab vehicle Raspberry controller is not required.

Installation

Run the RTI installer

```
sudo ./rti_connext_dds-6.0.0-eval-x64Linux4gcc7.3.0.run
```

and select /opt/rti_connext_dds-6.0.0 as the installation directory.

Copy the Raspberry toolchain to /opt:

```
sudo cp -R raspbian-toolchain-gcc-4.7.2-linux64 /opt
```

Get a copy of the RTI license and move it to /opt/rti_connext_dds-6.0.0/rti_license.dat.

Environment Setup

Register the RTI libraries with the dynamic linker. Edit with `sudo nano /etc/ld.so.conf` and insert:

```
/opt/rti_connext_dds-6.0.0/lib/x64Linux4gcc7.3.0
```

The run `sudo ldconfig`.

Select a unique DDS domain! To avoid interference from other users in the same network, you need to set a DDS domain ID that is different from everyone in the network. The domain ID is assumed to be in the environment variable `DDS_DOMAIN`. Currently occupied IDs are:

ID	purpose
21	main server
60 - 66	Lab PCs

Edit the global profile with the code below while inserting your own unique ID instaed of the "<>", for example using `sudo nano /etc/profile` and insert:

```
export DDS_DOMAIN=<YOUR UNIQUE DOMAIN ID>
export PATH=$PATH:/opt/rti_connext_dds-6.0.0/bin
export PATH=$PATH:/opt/raspbian-toolchain-gcc-4.7.2-linux64/bin
export NDDSHOME=/opt/rti_connext_dds-6.0.0
export RASPBIAN_TOOLCHAIN=/opt/raspbian-toolchain-gcc-4.7.2-linux64
export RTI_LICENSE_FILE=/opt/rti_connext_dds-6.0.0/rti_license.dat
```

Reboot to apply the changes made to the environment variables.

Install RTI ARM libraries

```
sudo /opt/rti_connext_dds-6.0.0/bin/rtipkginstall rti\_connext\_dds-6.0.0-core-target-armv6vfpLinux3.xgcc4.7.2.rtipkg
```

eProsima Setup

eProsima is used for real-time communication between programs and devices. It implements a publish-subscribe pattern and serialization/deserialization for the messages. It is mandatory.

If you use one of the RTI branches, take a look at [RTI DDS Setup](#).

General Installation

This is covered by the install.sh script in the software folder of the repository, in point 3.

The path to the toolchain is set there as well. BUT: **The toolchain is currently not part of the install script.**

How to Install the Toolchain

The Toolchain is required to build for the Raspberry Pi / for the vehicles (cross-compilation).

It can be downloaded using this link <https://rwth-aachen.sciebo.de/s/aw1UGIULZGV4L07>

The Toolchain must be put in /opt/cross-pi-gcc.

Installation on the NUC

The script install_eprosima_on_nuc.bash should include all required information and can be found in the software folder of the repository.

Matlab Setup (optional)

Matlab Installation (or Upgrade)

The Matlab Package Manager (mpm) can be used to install Matlab on a NUC via the command line. An exemplary bash script to remove current versions and install a new one follows.

```
#!/bin/bash

cd /tmp

# remove installed versions if present
rm -rf /usr/local/MATLAB/R2020a/
rm -rf /usr/local/MATLAB/R2022a/
rm -rf /opt/MATLAB/R2020a/
rm -rf /opt/MATLAB

# get mpm
wget -q https://www.mathworks.com/mpm/glnxa64/mpm
chmod +x mpm

# install release and toolboxes
MATLAB_RELEASE=r2023a
TOOLBOXES="Parallel_Computing_Toolbox ROS_Toolbox Statistics_and_Machine_Learning_Toolbox Symbolic_Math_Toolbox"
./mpm install --release=$MATLAB_RELEASE --destination=/usr/local/MATLAB/R2023a/ --products MATLAB $TOOLBOXES

# symbolic link
ln -sf /usr/local/MATLAB/R2023a/bin/matlab /usr/local/bin/matlabR2023a

# remove mpm
rm -f mpm /tmp/mathworks_root.log
```

Network Setup

Matlab is set up on the NUCs using a network license using this [guide](#). We currently use the Matlab 2023a version.

If an existing installation with license is present on a NUC, simply copying the Licenses folder in the installation directory (e.g. /usr/local/MATLAB/R2023a/licenses) to the new installation folder suffices.

DDS Setup

You can find all information required to set up RTI DDS in Matlab here: <https://www.mathworks.com/hardware-support/rti-dds.html>.

A backup copy can be found in sciebo/CPM/Project/Lab/08_Other/MatlabRtiDdsBackup.

To use RTI DDS Support, you must have RTI DDS installed on your system.

Important Environment Variables

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:
    /home/controller/...: //Library files of cpm lib (libcpm.so) and middleware (libadditional_idl.so)
    /opt/rti_connext_dds-6.0.0/lib/x64Linux4gcc7.3.0
PATH=$PATH:
    $HOME/bin:
    $HOME/.local/bin:
    /opt/rti_connext_dds-6.0.0:
    /opt/rti_connext_dds-6.0.0/bin:
    /opt/rti_connext_dds-6.0.0/lib/x64Linux4gcc7.3.0:
    /opt/rti_connext_dds-6.0.0/bin:
    /opt/raspbian-toolchain-gcc-4.7.2-linux64/bin
NDDSHOME=/opt/rti_connext_dds-6.0.0

RTI_LICENSE_FILE=/opt/rti_connext_dds-6.0.0/rti_license.dat
```

Your installation paths might be different, so you might need to change some of them.

Vehicle Setup

- [PCB Assembly](#)
- [Vehicle Assembly](#)
- [Raspberry Pi Setup](#)
- [ATmega Flashing](#)
- [Current Servo Setup](#)

PCB Assembly

The μ Car consists of 2 PCBs. Download the [Eagle files](#).

- [Vehicle Main Board](#)
- [Odometer Board](#)

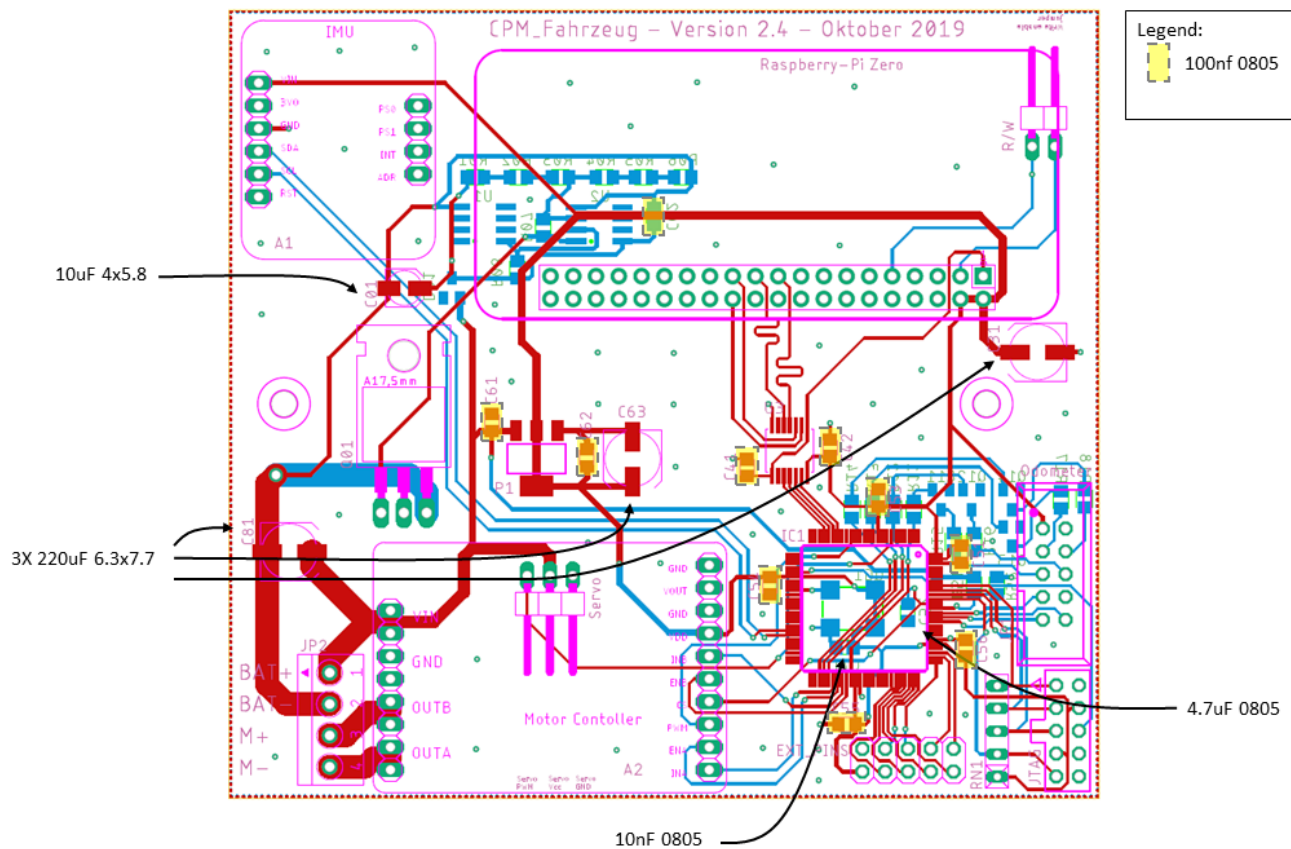
For questions please contact Patrick Scheffe.

Vehicle Main Board

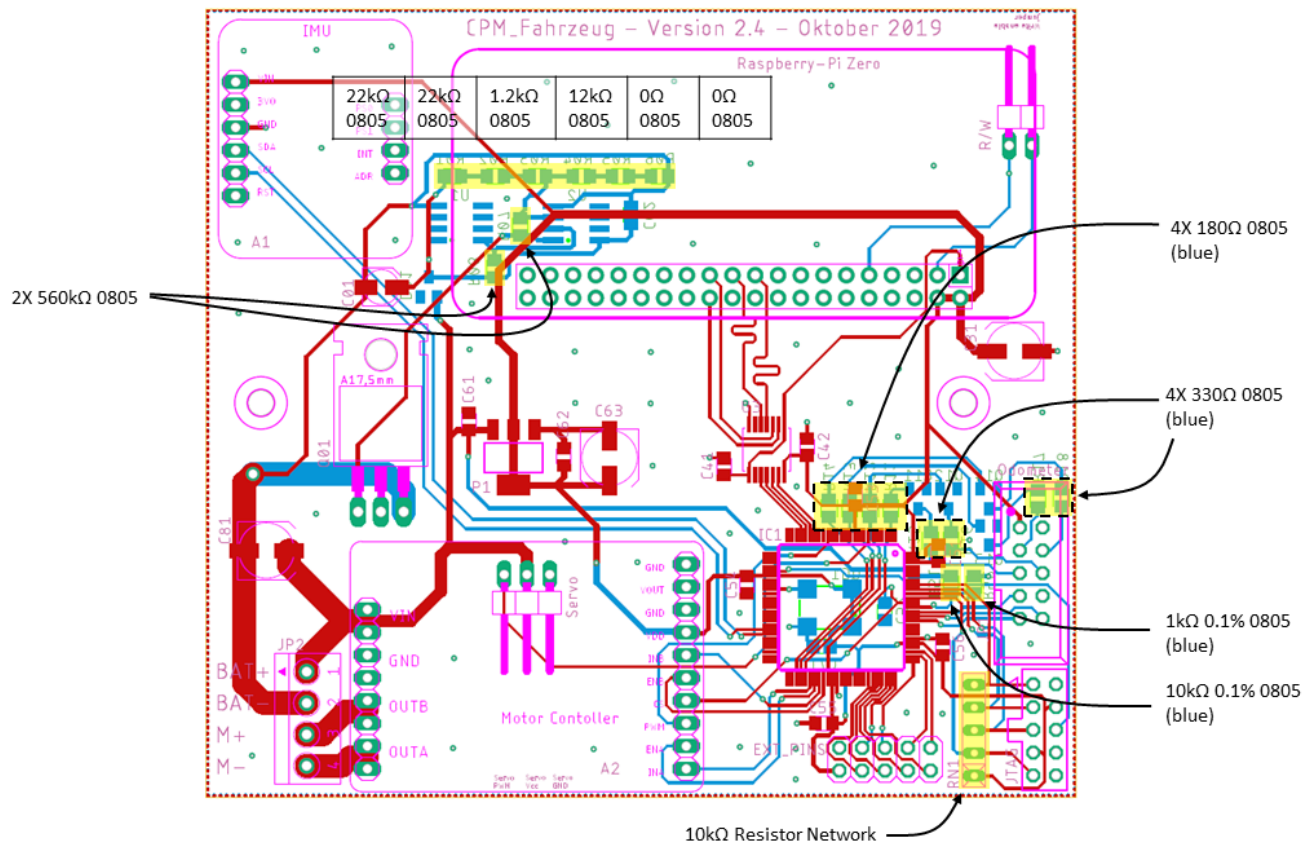
The main board contains most electrical components necessary for controlling the vehicle:

- Raspberry Pi Zero W ([Mid Level Controller](#))
- ATmega2560 ([Low Level Controller](#))
- IMU BNO055
- Motor Controller VNH5019

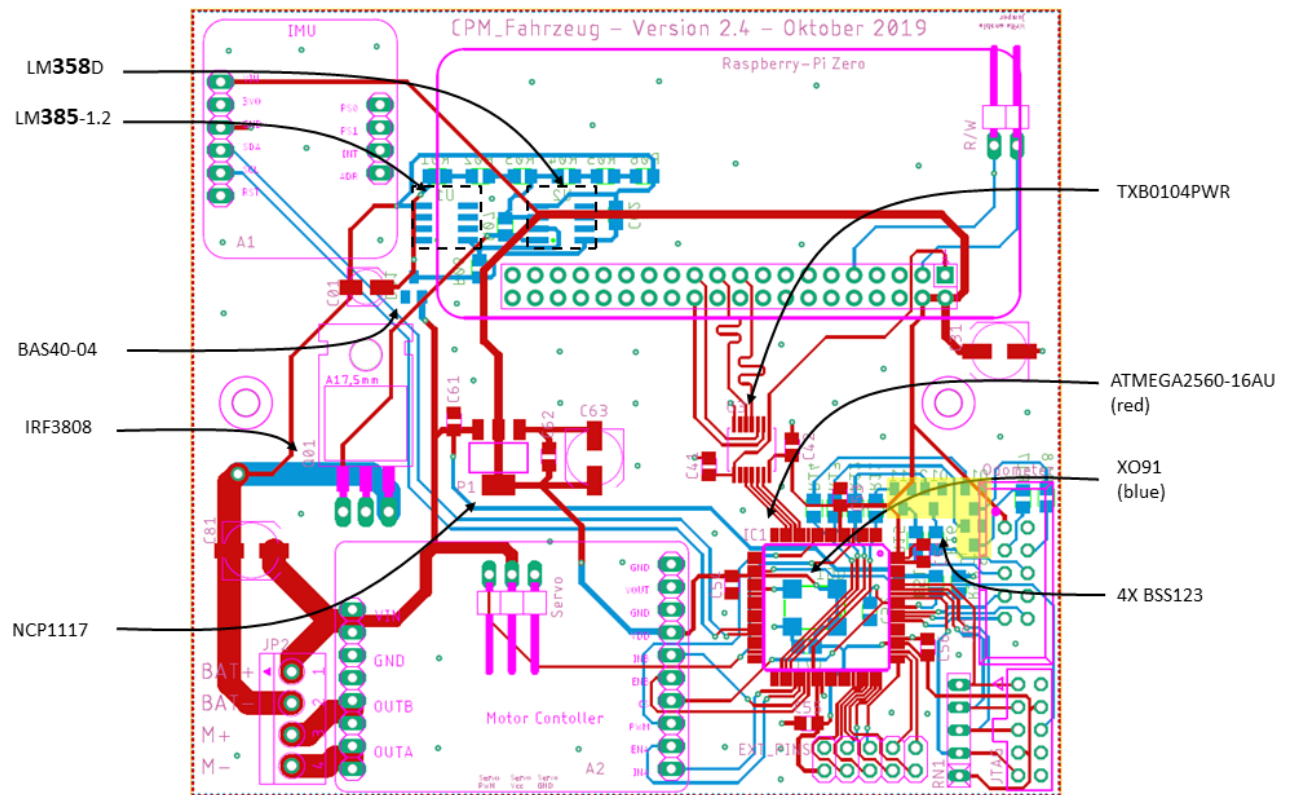
1. Component Placement: Capacitors



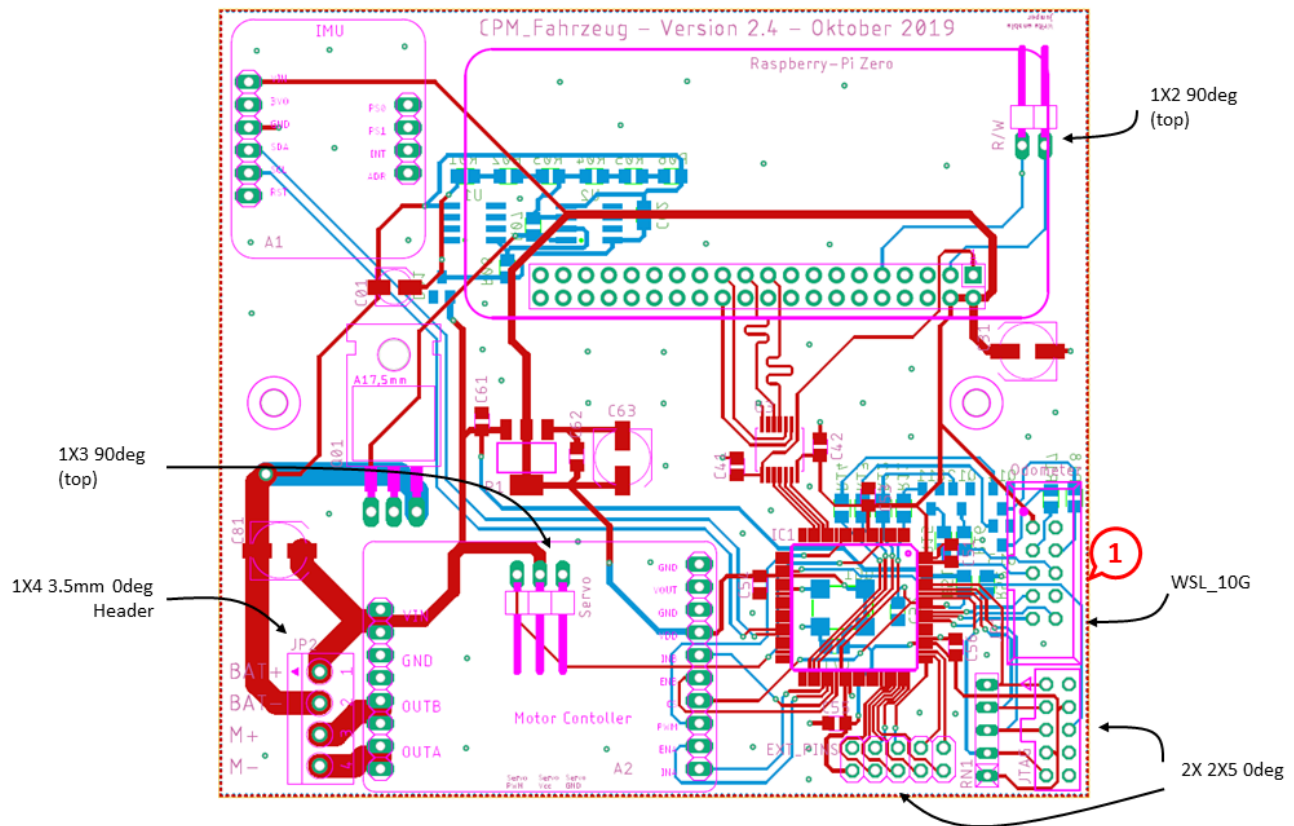
1. Component Placement: Resistors



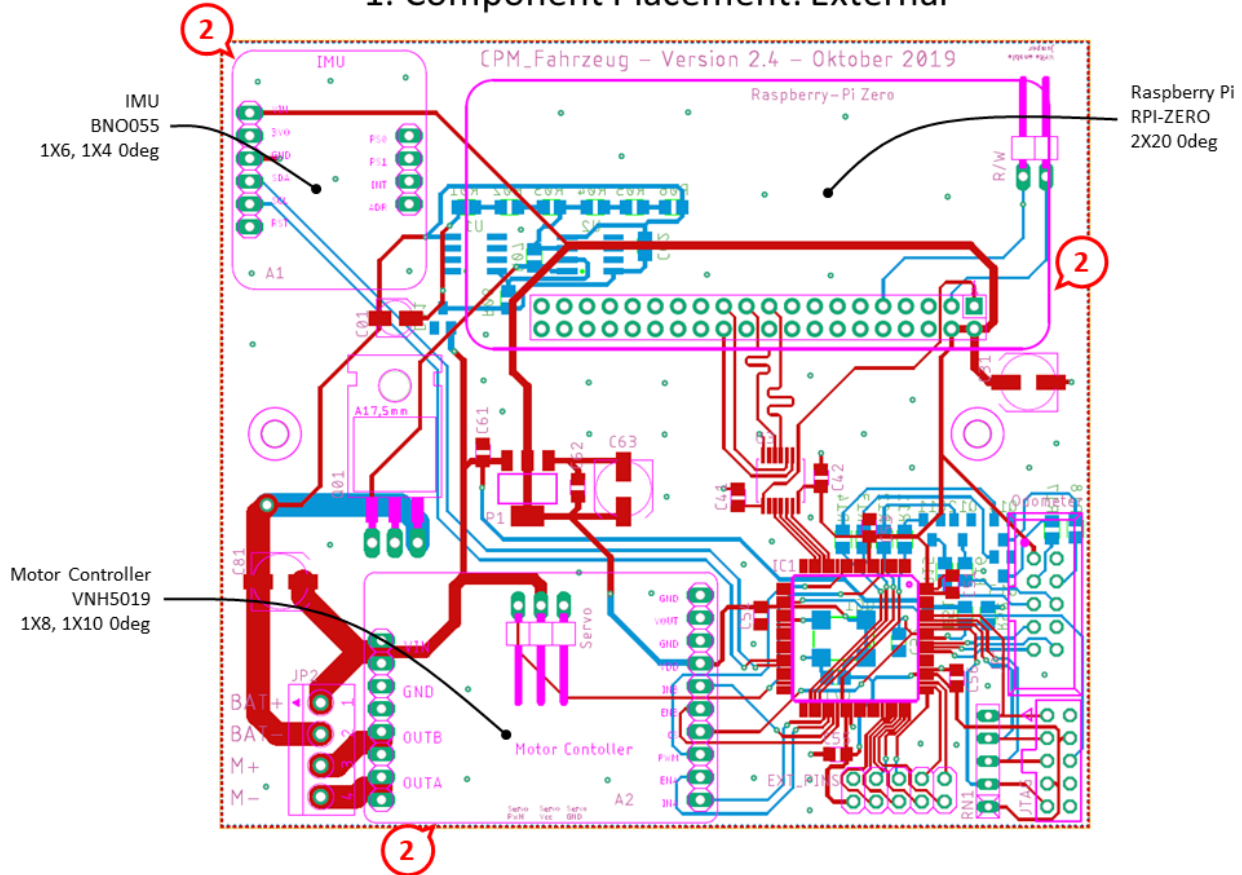
1. Component Placement: ICs



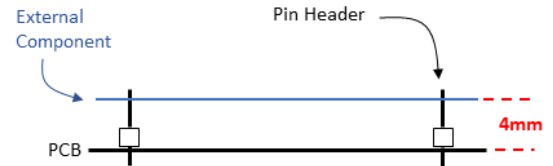
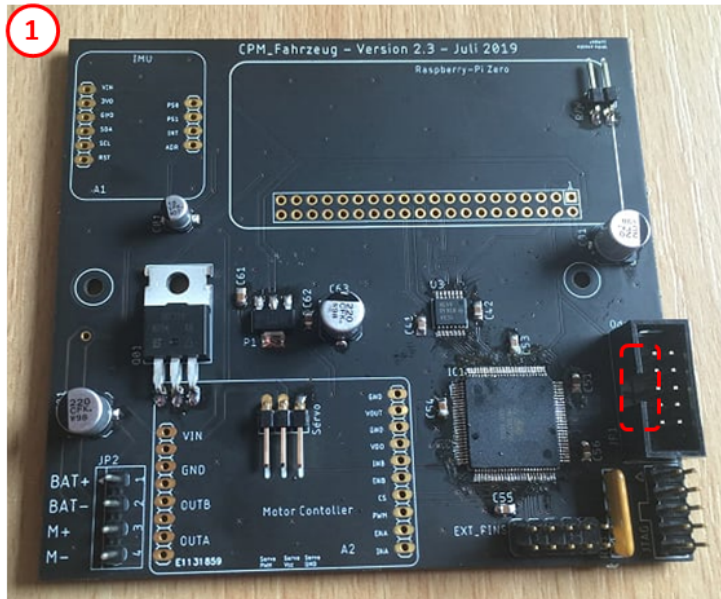
1. Component Placement: Connectors



1. Component Placement: External



2. Notes

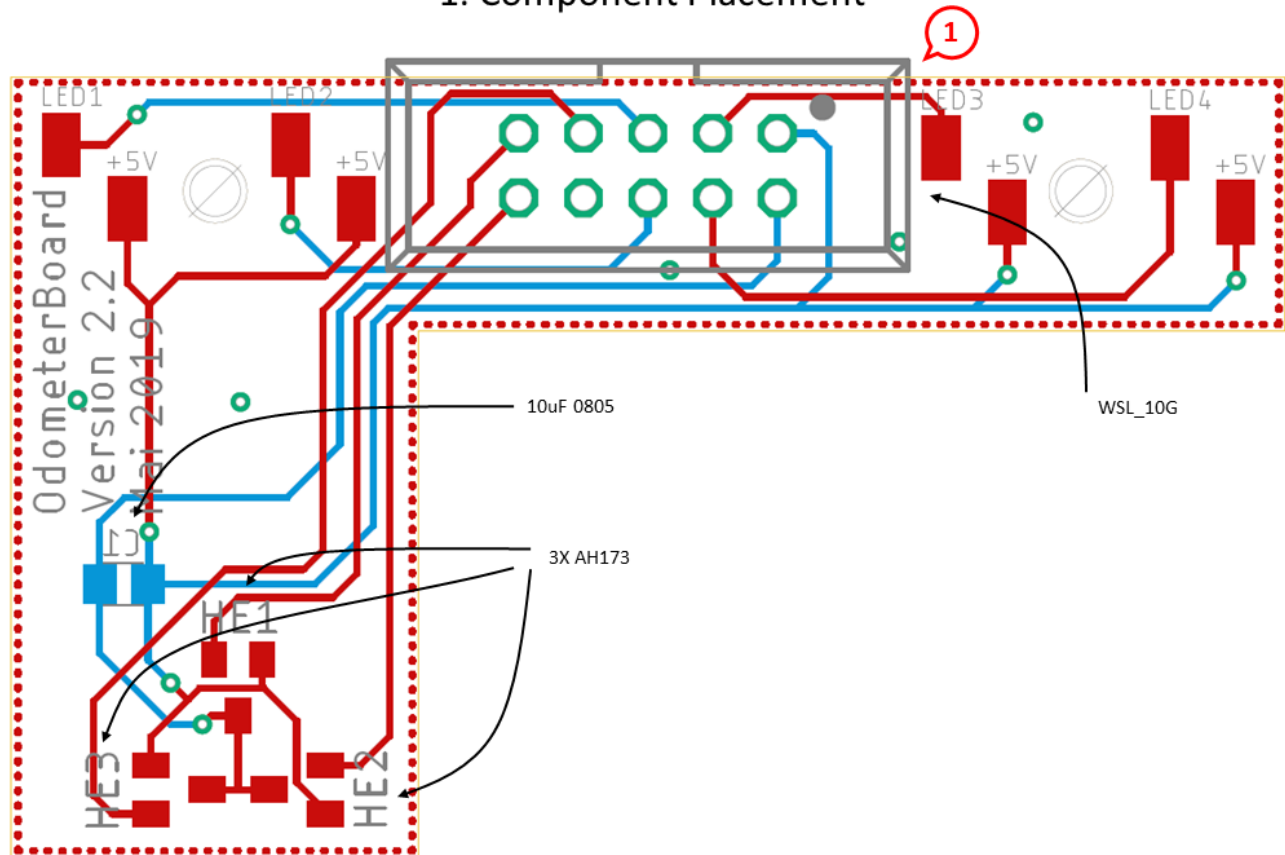


Note	Instructions
1	Take note of orientation Tab (form fit) points inward
2	Use spacers to ensure 4mm gap between external components and PCB Applies to IMU (A1), Motor Controller (A2) and Raspberry Pi

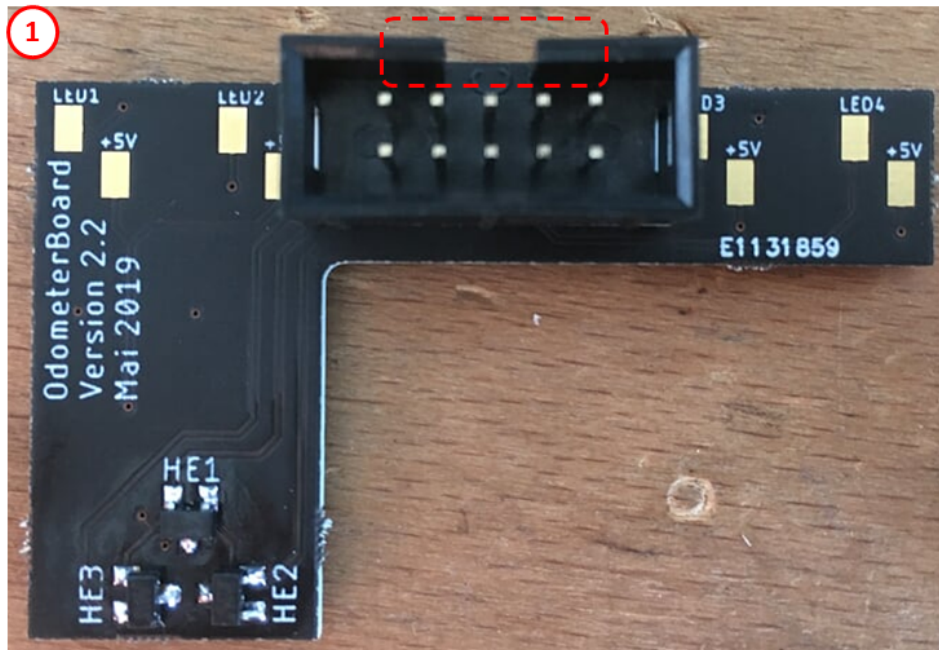
Odometer Board

The odometer board contains two important parts. Firstly, the hall sensors are placed on it and are measuring changes of the magnetic field. These changes result from the rotation of the magnet, which is placed on top of the motor shaft. That allows measuring the driven distance and the velocity. Secondly, the board provides the connections to the LEDs which are necessary for the [IPS](#).

1. Component Placement



2. Notes



Note	Instructions
1	Take note of orientation Tab (form fit) at top

Vehicle Assembly

This section describes the assembly of a vehicle. Firstly, lists with necessary materials and useful tools are given. Afterwards, the assembly is described step by step divided into the following 7 parts:

- [0. Bill of Materials](#)
- [1. Underframe Preparation](#)
- [2. Adhesive Preparation](#)
- [3. XRAY M18 PRO Assembly](#)
- [4. Servo Configuration](#)
- [5. Soldering](#)
- [6. LED Holder](#)
- [7. Final Assembly](#)

These instructions do not include PCB soldering. For instructions on PCB assembly, see [PCB Assembly](#).

For questions regarding assembly, please contact:

- Patrick Scheffe

[Contents](#)

0. Bill of Materials

The following tables show the quantities of the components needed for *one* vehicle. *Note:* The elements needed for PCB-Assembly are not given here.

Mechanical Components		Quantity
XRAY M18 Pro LiPo Kit		1
NF113LG-011 DC motor		1
Motor Magnet (6x2.5 mm, diametrically magnetised)		1
Battery Holder (.stl file for 3D printing)		1
LED Holder (.stl file for 3D printing)		1
Screws	M2x6mm	2
	M2x5mm	2
	M2x10mm Countersink	2
	M2.5x6mm	3
	M3x10mm	2
Nuts	M2	6
	M2.5	5
	M3	4
Washers	M2.5	2
	inner diameter: 2.2mm outer diameter: max. 4.5mm	2

Electrical Components	Quantity
Vehicle Main Board	1
Odometer Board	1
Female DSUB 9 connector	1

HiTEC D89MW servo			1
Banana connector 4mm			2
Banana connector 2mm			1
100nF capacitor			2
Fuse holder 0031.8201.G			1
Fuse 521,027			1
Red 1.0 wire end ferrule			4
4-Pole connector			1
LED NSDW570GS-K1			4
IDC Ribbon cable 10 POL (AWG 28-10G 10M)			11 cm
IDC Socket 10 POL			2
Heat shrinks			
Silver wire			
	Color	Gauge	
Wires	Red	1mm2	31 cm
	Black	1mm2	25.5 cm
	Blue	0.5mm2	8.2 cm
	Red	0.14mm2	35 cm
	Black	0.14mm2	35 cm

Tools
2-Part epoxy adhesive
HiTEC DPC-11 Software
Servo Programmer incl. necessary harnesses
Soldering iron
Crimper
3D-Printer
Diverse drills, screwdrivers and allen keys

[Contents](#)

1. Underframe Preparation

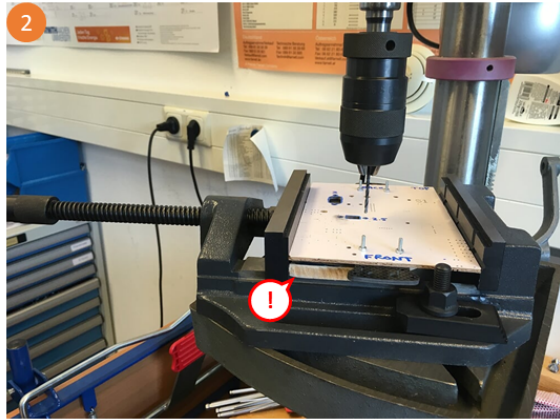
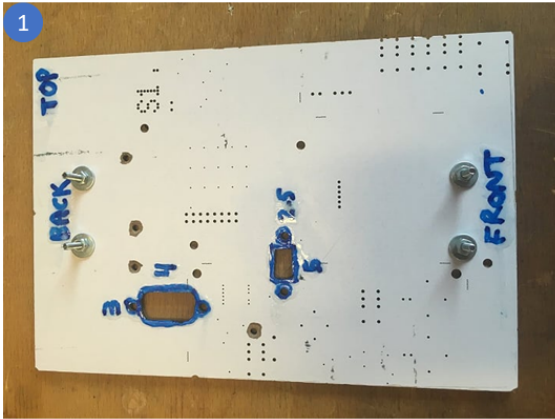
The CPM Lab's vehicles are based on the XRAY M18 Pro LiPo Kit. However, for using it in the Lab, several adaptations have to be applied to it.

In this step, additional holes will be added to the underframe of the M18 to provide mounts for the charging connection, the fuse, and the switch. For this, a wooden template is used onto which even multiple underframes can be attached.

1. Car Underframe Preparation

Part	Quantity
XRAY M18 PRO Underframe	1

Tool
XRAY M18 PRO Drill Template, 4x M2.5mm screws and nut
Wood Drill Underlay

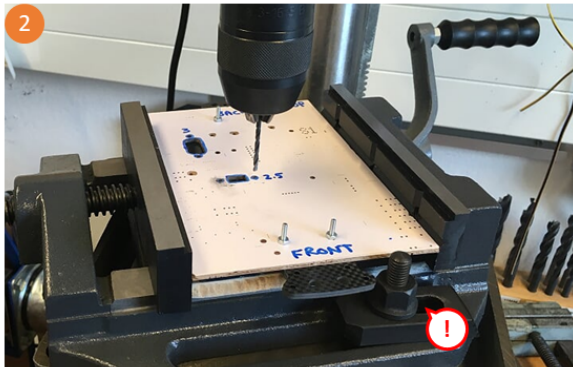


Step	Instructions
1	Attach template to top of underframe NOTE: take note of orientation
2	Mount underframe+template to drill with wood underlay beneath NOTE: without wood underlay, underframe will bend and drilling will not work

2. Car Underframe Drilling

Part	Quantity
XRAY M18 PRO Underframe	1

Tool
2.5mm, 3mm, 4mm, 5mm Drill Bit



Step	Instructions
1	Drill 3mm DSUB holes NOTE: make sure to secure vice to ensure proper hole placement
2	Drill 2.5mm switch holes NOTE: make sure to secure vice to ensure proper hole placement
3	Use 4mm drill to drill out DSUB port
4	Use 5mm drill to drill out switch port

3. Car Underframe Filing

Part	Quantity
XRAY M18 PRO Underframe	1
XRAY M18 PRO Drill Template	1

Tool
Files
Vice

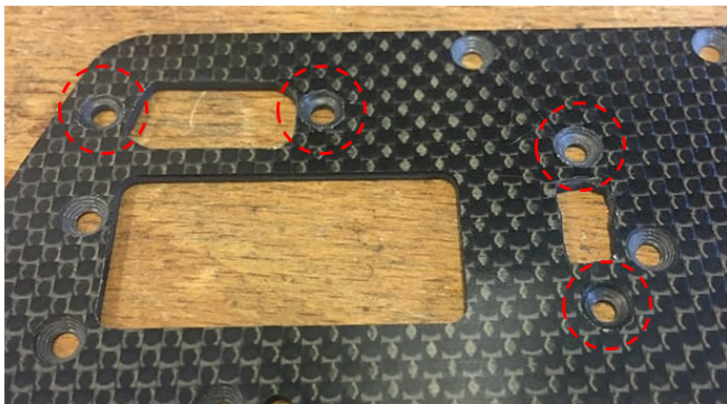


Step	Instructions
1	Using vice and square file to file final switch shape, check results with switch NOTE: consider removing template towards end to avoid damaging
2	Using vice and a combination of files to file out DSUB port, check results with male DSUB NOTE: consider removing template towards end to avoid damaging

4. Car Underframe Post

Part	Quantity
XRAY M18 PRO Underframe	1

Tool
6mm Drill Bit or M3 Countersink
Wood Drill Underlay



Step	Instructions
1	If not already removed, remove template
2	Use 6mm drill bit or M3 Countersink to countersink 3mm and 2.5mm holes NOTE: countersink holes on bottom of underframe

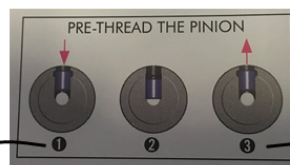
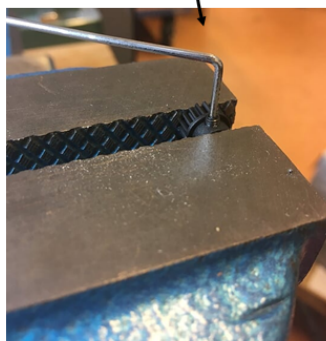
2. Adhesive Preparation

Since the adhesive needs some time for drying, this step is done now. On the one hand, the magnet must be adhered to the cog to allow distance measuring by the odometer. On the other hand, two nuts fixed to the Flex Piece will allow an easier fastening of the Main Board in the end.

1. Magnet Motor Cog Pre-Thread

Part	Quantity
17T Cog from XRAY M18 Pro Kit (tooth count written on back)	1
Threaded Insert	1

Tool
1.5mm Allen Key

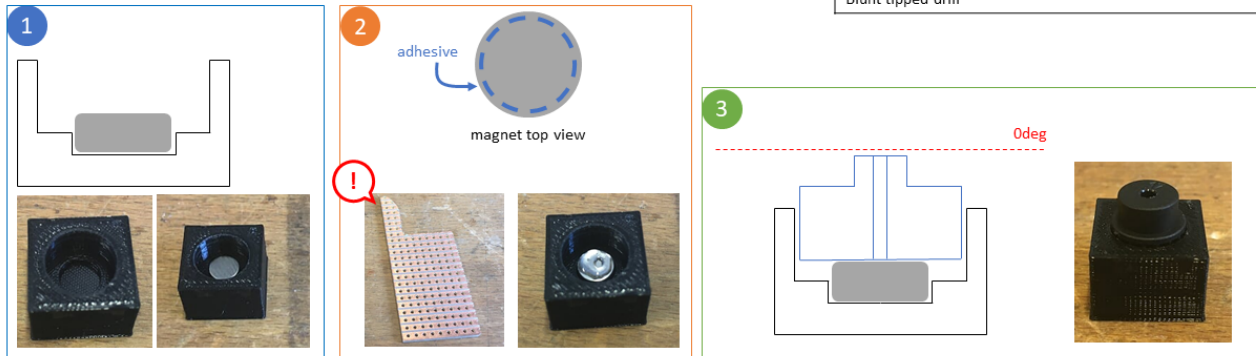


Step	Instructions
1	Use 1.5mm Allen Key to pre-thread cog per manufacturers instructions (p. 26) <i>TIP: clamping cog in vice facilitates steps</i>

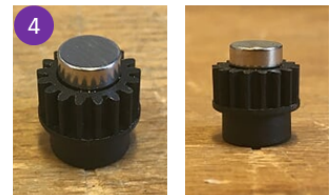
2. Magnet Motor Cog Mating

Part	Quantity
17T Cog from XRAY M18 Pro Kit (tooth count written on back)	1
Bipolar magnet	1

Tool
Cog Magnet Mating Tool (3D Printed)
2-Part Epoxy Adhesive
Blunt tipped drill



Step	Instructions
1	Place magnet into bottom of tool
2	Apply mixed Epoxy to top of magnet using NON-MAGNETIC TOOL/TIP NOTE: apply glue on perimeter of top to avoid shaft hole
3	Slowly and carefully slide cog into tool until it has settled onto magnet. Ensure cog is level. NOTE: do not excessively press on cog to prevent glue spread
4	Allow glue to harden according to specifications. If necessary, use blunt-tipped drill to clear shaft hole.



Recommended 2-Part Epoxy Adhesive



3. PCB Nut Fixation (1/2)

Part	Quantity
M2 Nut	2
M2x6mm Screw	2
M18 XRAY PRO Flex Piece	1

Tool
2-Part Epoxy Adhesive

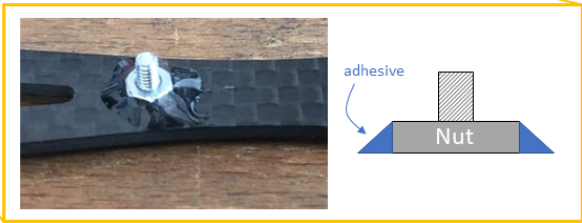
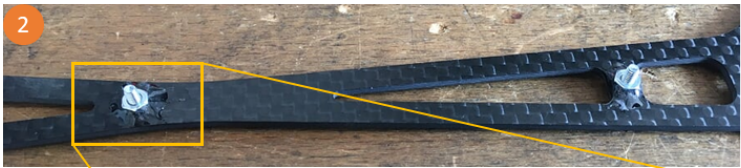


Step	Instructions
1	Use screws to fixate M2 Nuts to Flex Piece NOTE: make sure Nuts are on bottom (see images for orientation)

3. PCB Nut Fixation (2/2)

Part	Quantity
M2 Nut	2
M2x6mm Screw	2
M18 XRAY PRO Flex Piece	1

Tool
2-Part Epoxy Adhesive



Step	Instructions
2	Apply mixed adhesive to perimeter of screw (as fillet) NOTE: make sure to not get adhesive on screw because it can then no longer be removed

3. XRAY M18 PRO Assembly

In the next part, the M18 Pro is built according to its [manual](#), mostly. Consequently, this section focuses on tips and indicates deviations from the original.



XRAY M18 PRO Ins...ction Manual.pdf

XRAY M18 PRO Assembly

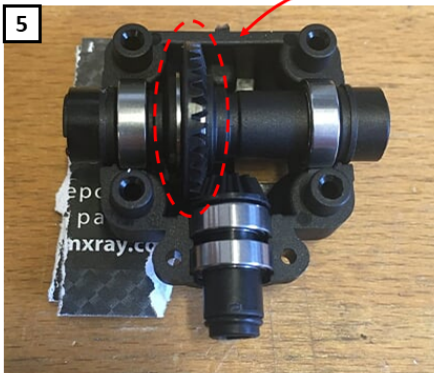
Part	Quantity
XRAY M18 Pro Set	1

Tool
M18 Assembly Manual



Step	Instructions
1	Follow assembly instructions NOTE: each section contains slides with notes and tips for assembly IMPORTANT: read slide to each section before proceeding with section

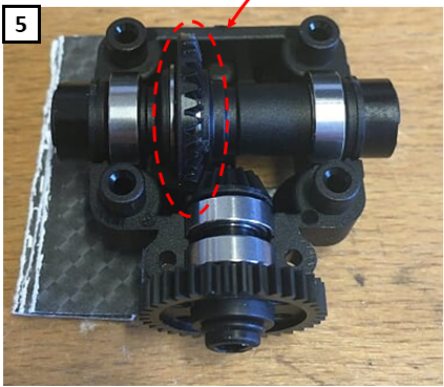
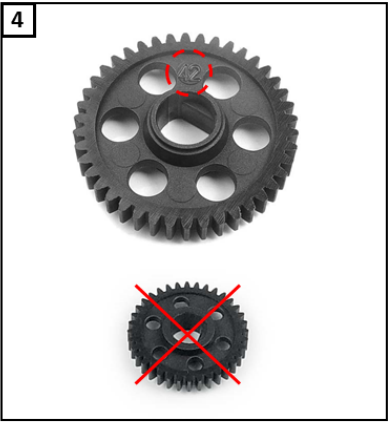
1. Front Differential



Take note of what side gear is on
Otherwise car will drive in wrong
direction given positive motor command

Step	Note
5	Take note of differential orientation
	Differential break-in done post vehicle assembly

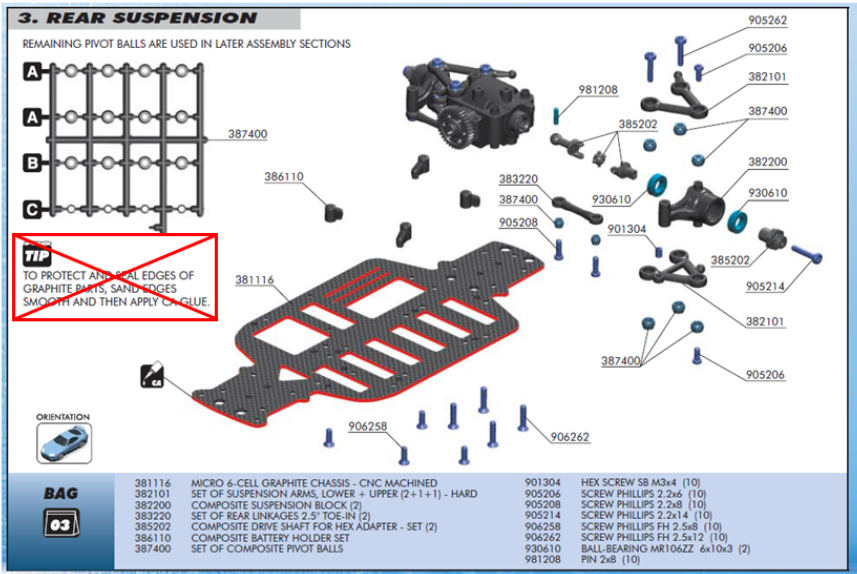
2. Rear Differential



Take note of what side gear is on
Otherwise car will drive in wrong
direction given positive motor command

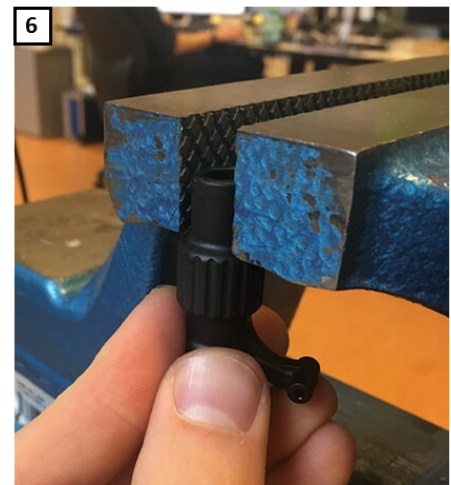
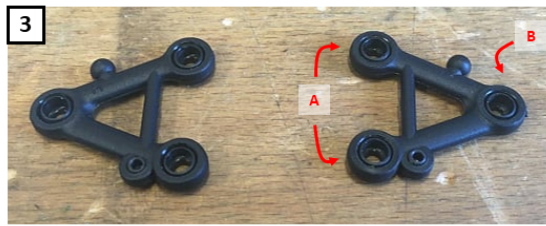
Step	Note
4	Note that for this vehicle the cog wheel with 42 cogs instead of the common one with 36 cogs has to be used.
5	Take note of differential orientation
	Differential break-in done post vehicle assembly

3. Rear Suspension (1/3)



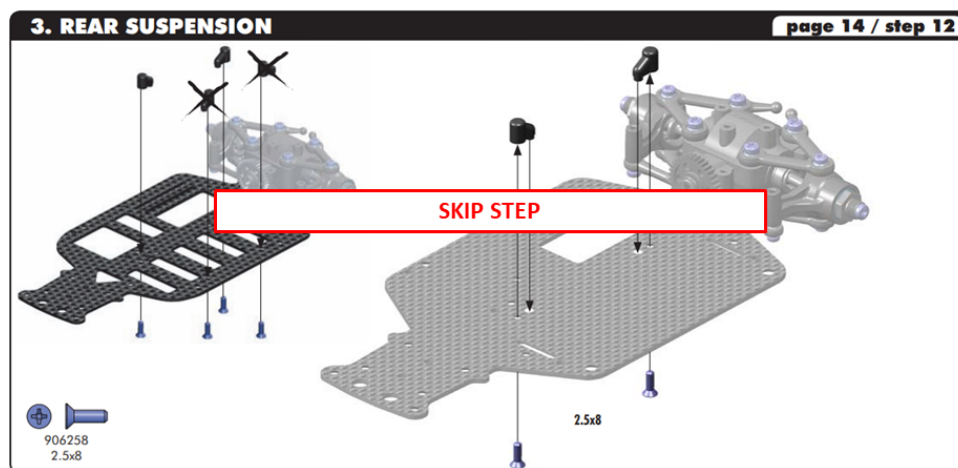
Step	Note
INTRO	Not necessary to seal edges with CA Glue given indoor use of vehicles

3. Rear Suspension (2/3)



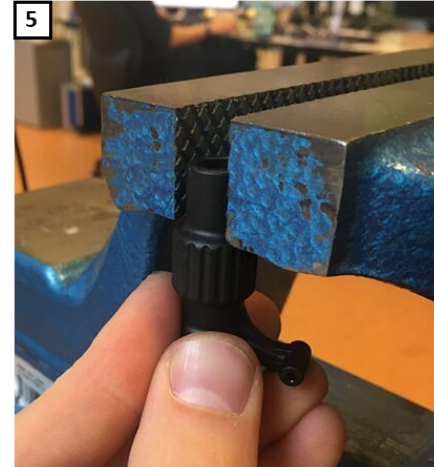
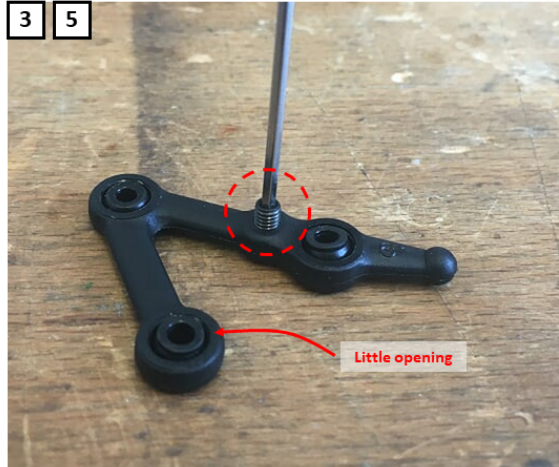
Step	Note
3	Take note of different types of pivot balls Take note of orientation of pivot balls and direction from which downstop screw is inserted
4	Take note of different types of pivot balls Take note of orientation of pivot balls
6	Use vice to clamp tool to facilitate step

3. Rear Suspension (3/3)



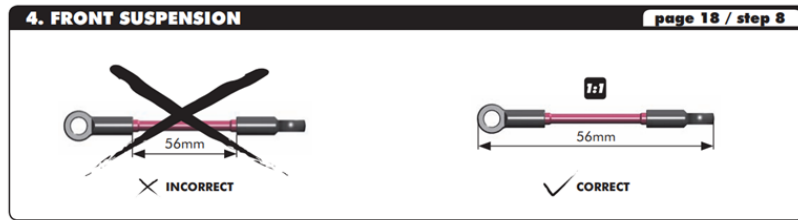
Step	Note
12	Skip step because we have our own battery holding, which will be installed later.

4. Front Suspension (1/2)



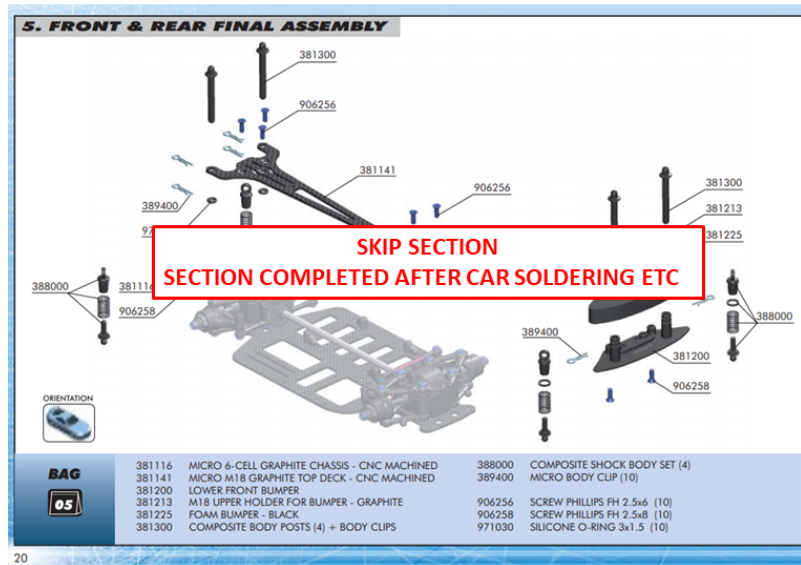
Step	Note
3	Take note of different types of pivot balls Take note of orientation of pivot balls
5	Take note of orientation of pivot balls and direction from which downstop screw is inserted
5	Use vice to clamp tool to facilitate step

4. Front Suspension (2/2)



Step	Note
8	Step wrong, see supplemental instruction sheet NOTE: exact length set during servo setup procedure (see next section)

5. Front & Rear Final Assembly

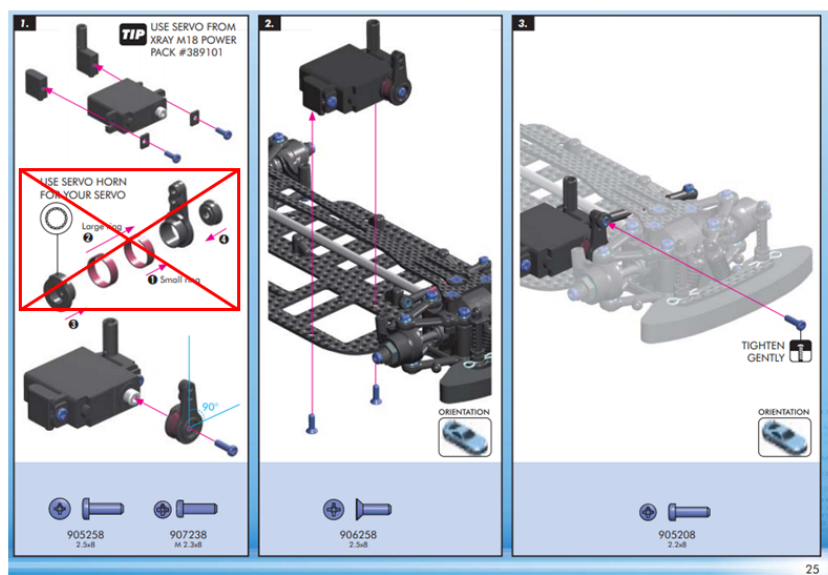


Step	Note
ALL	Skip section for now Modifications and steps to come easier done prior to Section 5

4. Servo Configuration

The assembly of the M18 Pro ends with the configuration of the servo and the wheels. In order to configure the HiTEC D89MW Servo, the [HiTEC DPC-11](#) software is used. For the current configuration of all vehicle's servos look [here](#).

1. M18 Pro Final Assembly: Servo (1/2)



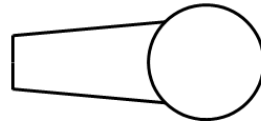
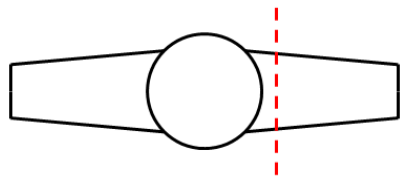
25

Step	Note
1-3	Follow Custom Servo Preparation and Installation Steps in following slides e.g. different servo horn will be used

2. Servo Prep (1/2)

Part	Quantity
HiTec D89MW Servo	1
180deg Servo Arm	1

Tool
Saw, File

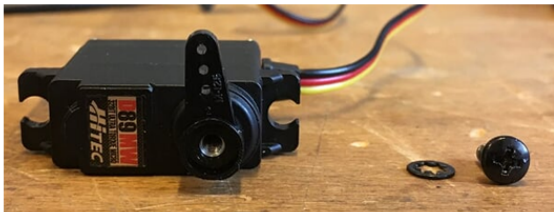
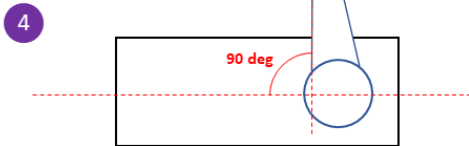
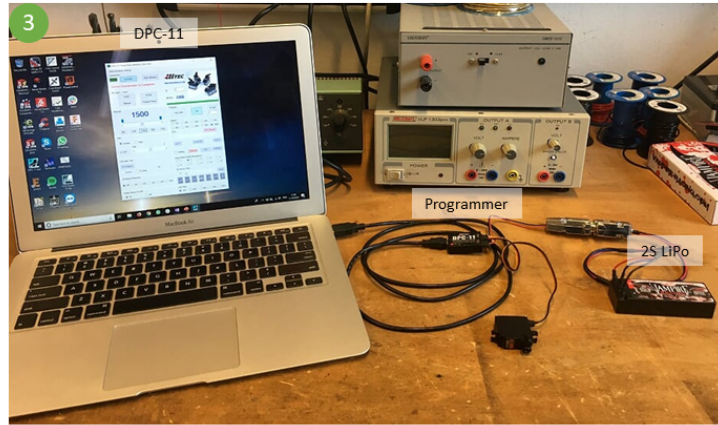


Step	Instructions
1	Remove currently installed servo arm
2	Trim one side of 180deg servo arm (see images above)

2. Servo Prep (2/2)

Part	Quantity
HiTec D89MW Servo	1
180deg Servo Arm	1

Tool
Servo Programmer, 2S LiPo, necessary Harnesses



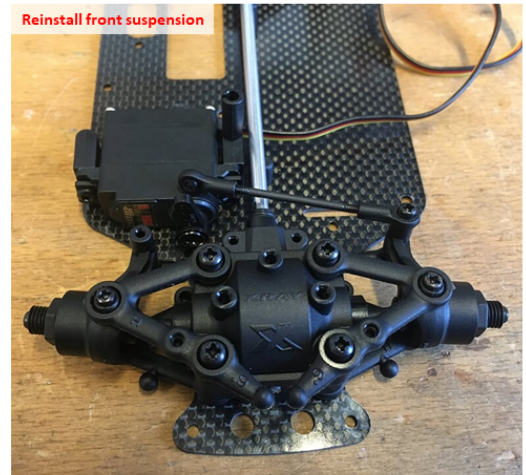
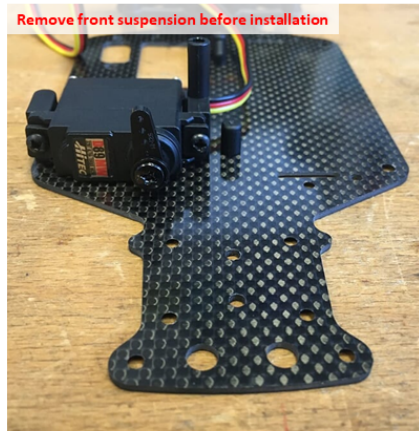
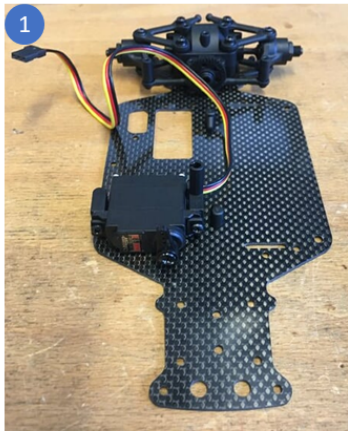
Step	Instructions
3	Using Arduino, HiTec DPC-11 Software, or existing vehicle, power servo and command to 0deg (1500) NOTE: use manual position test, 0deg corresponds to 1500
4	Attach modified servo arm as shown in the figure. Note: servo arm should be located on right side of servo : given splines on servo cog, side of servo arm may not be at exactly 90deg; this will be accounted for later



3. Servo Installation

Part	Quantity
HiTec D89MW Servo	1

Tool
M18 Assembly Manual

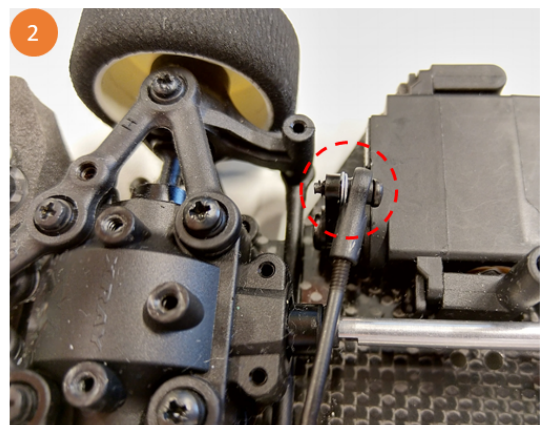
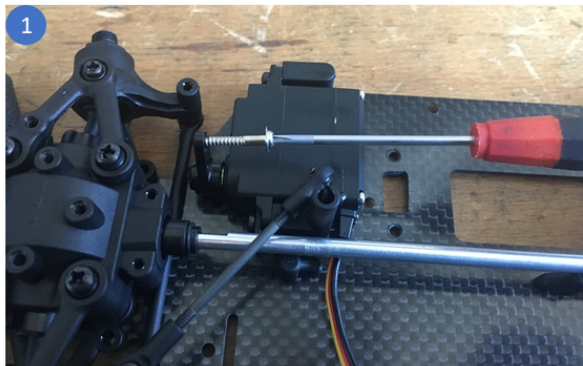


Step	Instructions
1	Attach servo to vehicle according to manual instructions (p. 25 Steps 1-2) Note: try to install servo in the center of servo mounting posts; inconsistencies will be accounted for later TIP: consider removing front suspension to facilitate access

4. Servo Arm Attachment

Part	Quantity
HiTec D89MW Servo	1
Washer (inner diameter: 2.2mm; outer diameter: 4.5mm)	2

Tool
M18 Assembly Manual

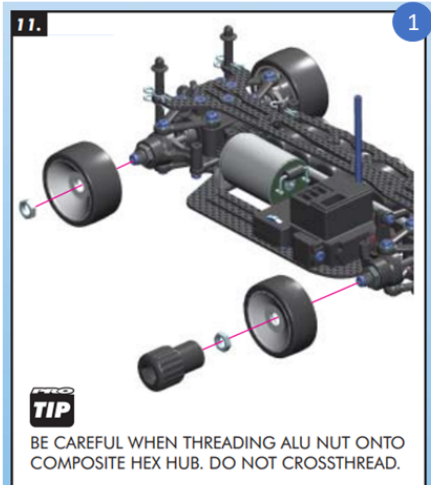


Step	Instructions
1	Use screw included with servo (shown on the left) to pre thread uppermost servo arm hole
2	Attach steering arm to servo arm so that steering arm is behind servo arm (see image) and the two washers are in between. Note: The wheels don't have to be attached here. That will be the next step.

5. Attach Wheels

Part	Quantity
M18 PRO Wheels	4

Tool
M18 Assembly Manual

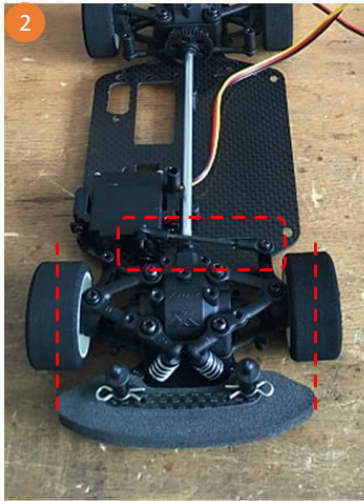
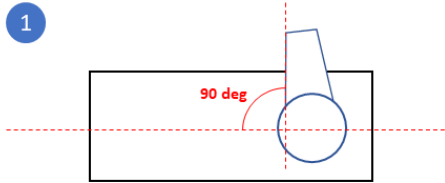


Step	Instructions
1	Attach wheels per manual; instructions (p. 29 step 11) <i>Note: the smaller wheels must be attached on the front axis</i>

6. Servo Arm Adjustment

Part	Quantity

Tool

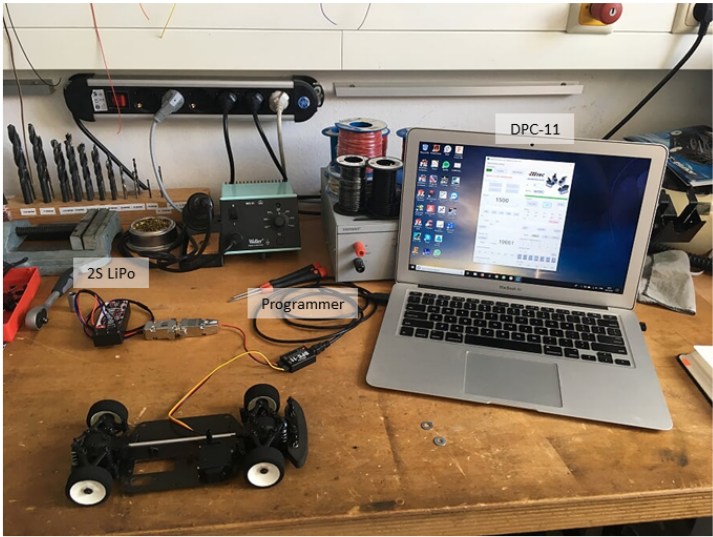


Step	Instructions
1	Move servo arm to the given position.
2	Adjust length of steering arm by turning thread in or out until front wheels are straight when servo arm stays at the shown position.

7. Servo Flashing (1/3)

Part	Quantity
HiTec D89MW Servo	1

Tool
Servo Programmer, 2S LiPo, necessary Harnesses



Step	Instructions
1	Connect Servo to 2S Lipo and Programmer Connect Programmer to computer and start DPC-11 Software.

7. Servo Flashing (2/3)

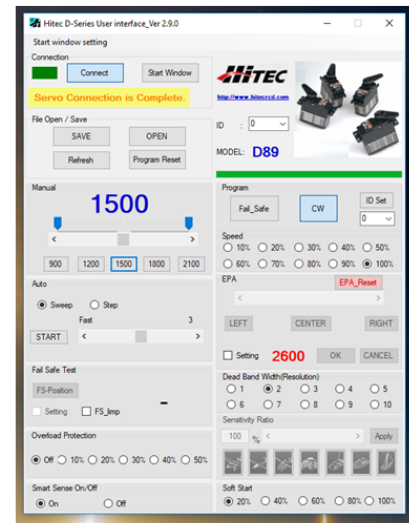
Part	Quantity
HiTec D89MW Servo	1

Tool
Servo Programmer, 2S LiPo, necessary Harnesses

1. Start DPC-11, select D-Series



2. Connect to Servo

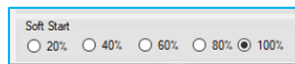
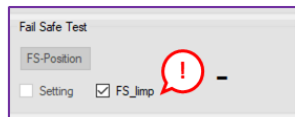
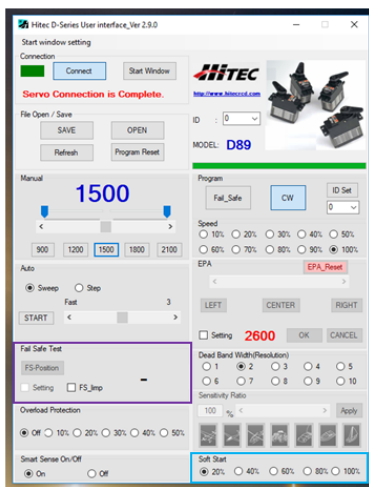


Step	Instructions
2	Start DPC-11 software Select D-Series Connect to Servo

7. Servo Flashing (3/3)

Part	Quantity
HiTec D89MW Servo	1

Tool
Servo Programmer, 2S LiPo, necessary Harnesses

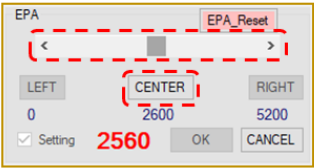
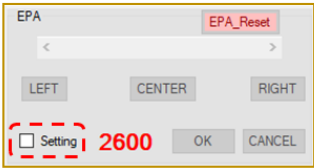


Step	Instructions
3	Only set/change following Settings: 1. Soft Start: 100% (was 20%) 2. FS_LIMP: checked (was unchecked) NOTE: do not set FS-Position, simply check FS_Limp

8. Servo End Point Adjustment (1/3)

Part	Quantity
HiTec D89MW Servo	1

Tool
Servo Programmer, 2S LiPo, necessary Harnesses

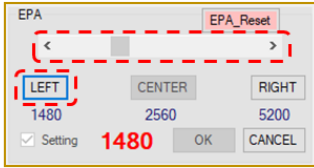
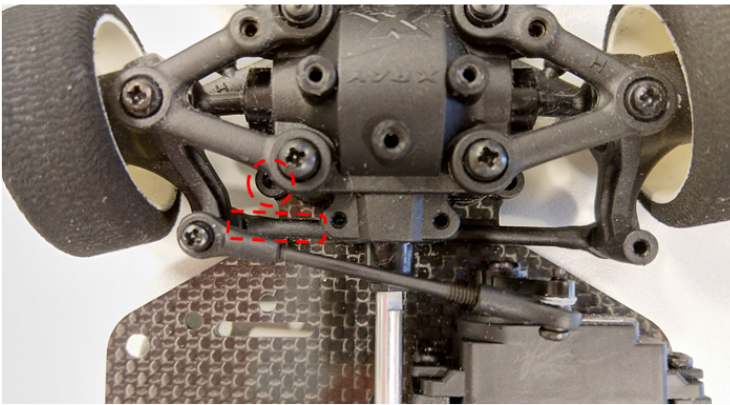


Step	Instructions
1	Activate the End Point Adjustment Setup by setting the hook.
2	Use the scrollbar to find the value at which the vehicle moves completely straight. Confirm this value by hitting "CENTER".

8. Servo End Point Adjustment (2/3)

Part	Quantity
HiTec D89MW Servo	1

Tool
Servo Programmer, 2S LiPo, necessary Harnesses

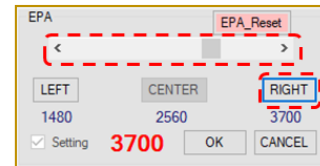
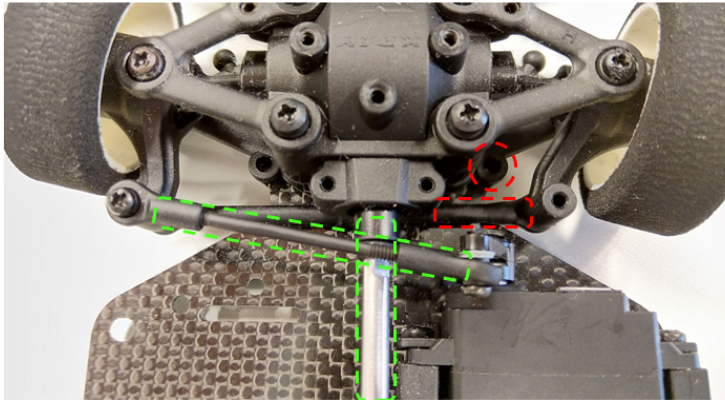


Step	Instructions
3	Steer further and further to the left until the two red marked components hit each other. Confirm this maximum by hitting "LEFT". Note: There should be no mechanical limits for the steering until this point is reached. If that isn't the case, adjust the steering arm accordingly and repeat the process.

8. Servo End Point Adjustment (3/3)

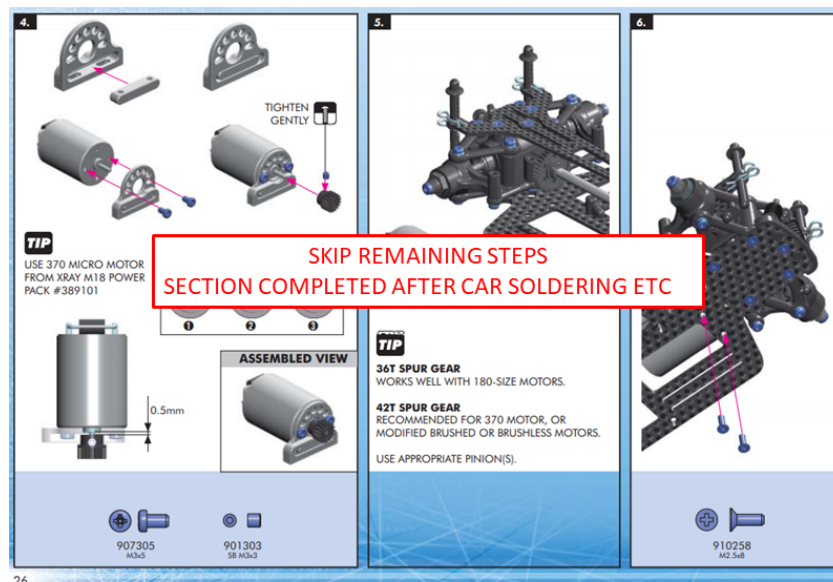
Part	Quantity
HiTec D89MW Servo	1

Tool
Servo Programmer, 2S LiPo, necessary Harnesses



Step	Instructions
4	Steer further and further to the right until the two red marked components hit each other. Confirm this maximum by hitting "RIGHT" (and finish the EPA by hitting "OK"). Note: There should be no mechanical limits for the steering until this point is reached. Especially the green components should NEVER hit each other. If that isn't the case, adjust the steering arm accordingly and repeat the process. Alternatively, it can also be helpful to reduce the suspension of the front wheels.

9. M18 Pro Final Assembly: Motor (2/2)



26

Step	Note
4-11	Skip steps for now Modifications and steps to come easier done prior to Steps 4-11

5. Soldering

This section describes the mounting of most electrical components, i.e., the soldering and mounting of

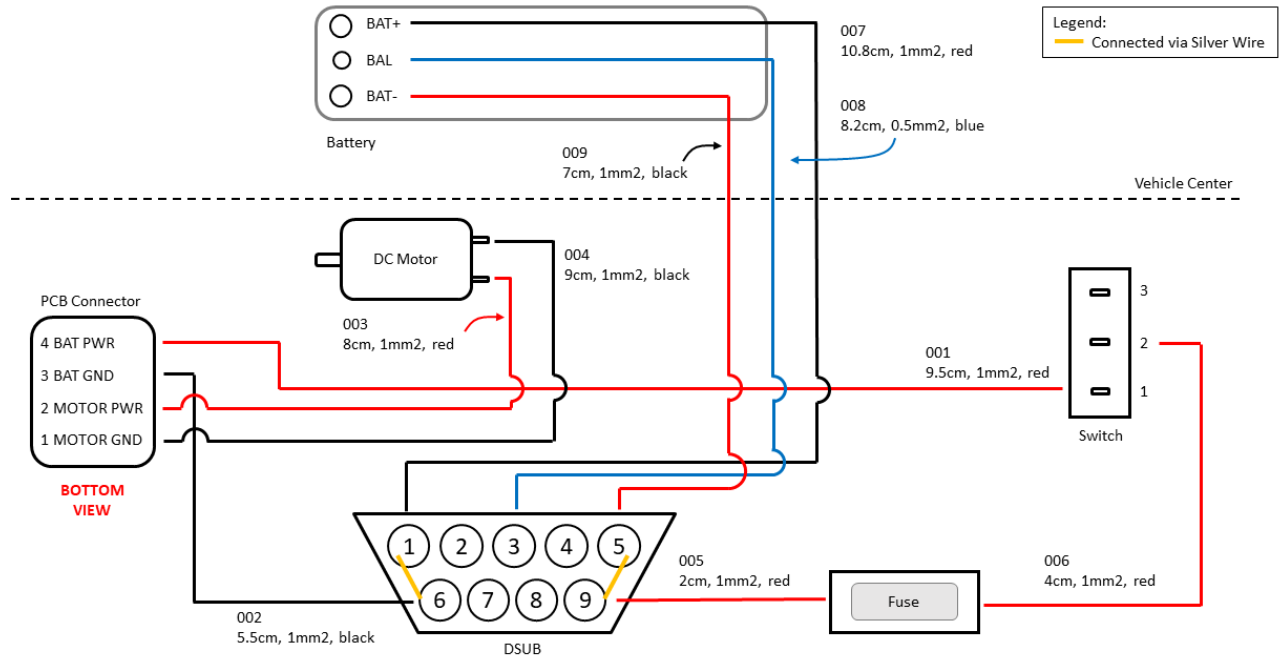
- the motor,
- the fuse,
- the switch,
- the charging connection (DSUB), and
- the battery.

1. Cut Wire (1/2)

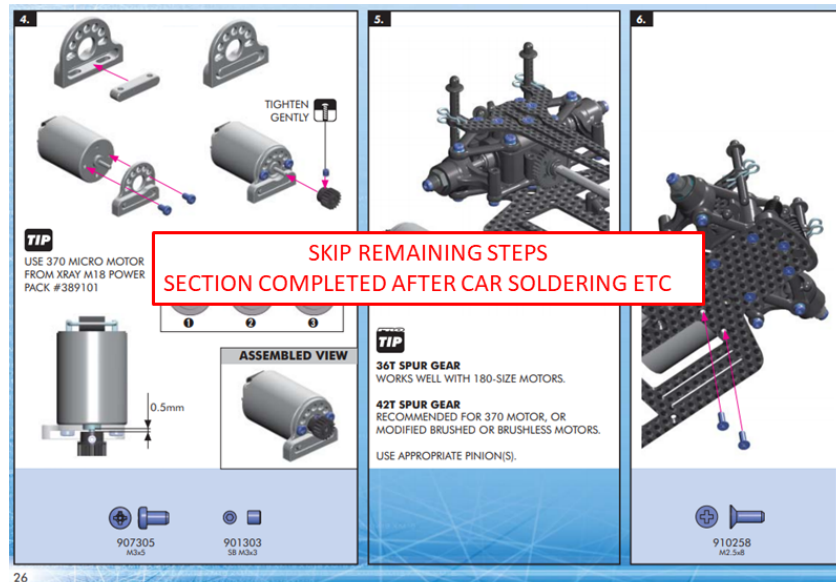
Wire PN	Length [cm]	Color	Gauge	Description	Location/Use
001	9.5 +0.25	Red	1mm2	Battery Power	Switch to PCB Connector
002	5.5 -0.25	Black	1mm2	Battery Ground	DSUB to PCB Connector
003	8.0 +0.25	Red	1mm2	Motor Power	Motor to PCB Connector
004	9	Black	1mm2	Motor Ground	Motor to PCB Connector
005	2	Red	1mm2	Vehicle Power	DSUB to Fuse
006	4	Red	1mm2	Vehicle Power	Fuse to Switch
007	7 -0.25	Red	1mm2	Battery Power	DSUB to Battery
008	8.2	Blue	0.5mm2	Battery Balance	DSUB to Battery
009	10.8 -0.25	Black	1mm2	Battery Ground	DSUB to Battery

Step	Instructions
1	Cut wire to correct length Note: take note of color and gauge

1. Overview Diagram (2/2)



9. M18 Pro Final Assembly: Motor (2/2)



Step	Note
4-11	Skip steps for now Modifications and steps to come easier done prior to Steps 4-11

3. Aderenden Huelsen

Part	Quantity
Red 1.0 Aderenden Huelsen	
Wire 001, 002, 003, 004	

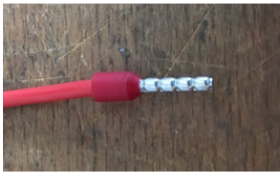
Tool
Crimper

001
9.5cm, 1mm2, red

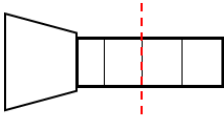
002
5.5cm, 1mm2, black

003
8cm, 1mm2, red

004
9cm, 1mm2, black



2

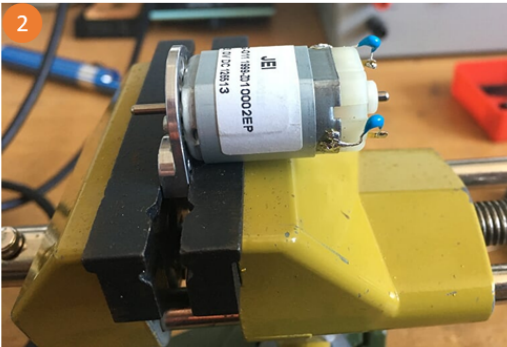
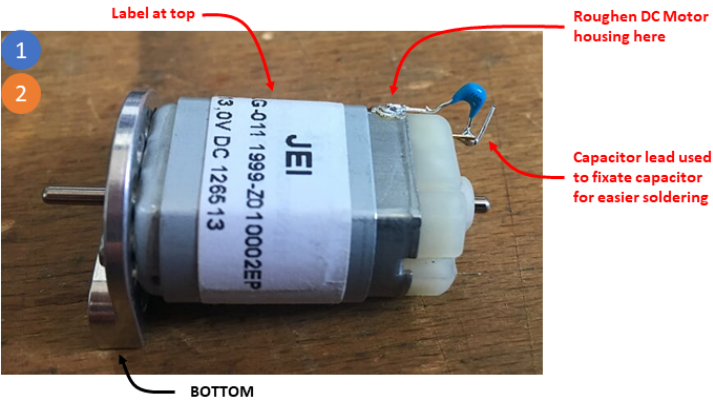


Step	Instructions
1	Crimp red 1.0 aderenden huelsen to PCB connector wires
2	Shorten aderenden huelsen in half (see image) NOTE: otherwise huelsen won't fit into connector

4. DC Motor Prep

Part	Quantity
DC Motor	1
100nF Capacitor (104)	2

Tool
M18 Assembly Manual
Soldering Iron

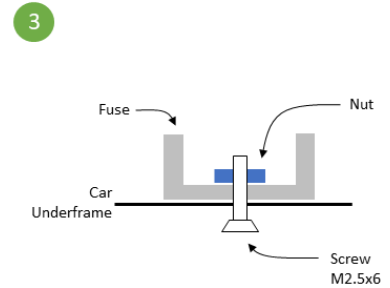
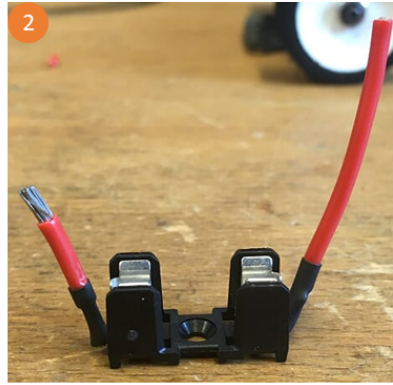
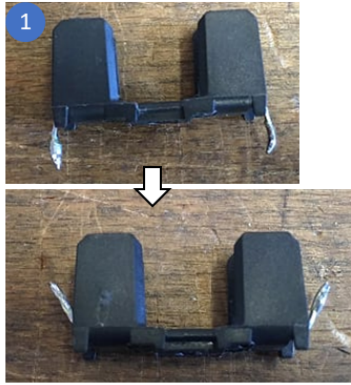


Step	Instructions
1	Attach Aluminum Motor Holder to DC Motor according to manual (p. 26 Step 4), DO NOT INSTALL MOTOR WARNING: use max M2x5 screws to attach motor NOTE: check polarity of motor i.e. spin direction (usually correct when label/sticker on top)
2	Solder 100nF Capacitors to motor leads and housing NOTE: 100nF cap usually labelled as 104 (104 = 10*10^4pF) TIP 1: use sandpaper to slightly roughen motor housing surface for better soldering Tip 2: fixate cap by wrapping capacitor lead wire through motor lead hole

5. Fuse Prep

Part	Quantity
Fuse Holder	1
Wires 005, 006	1
M2.5x6mm Screw; M2.5 nut	1; 1

Tool

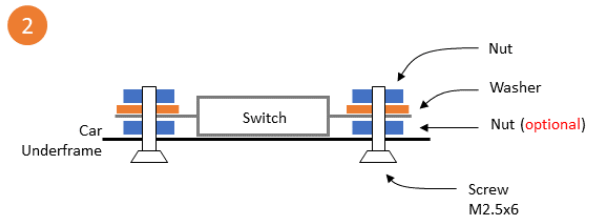
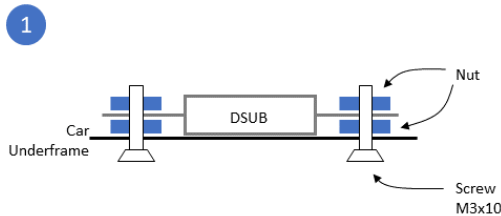


Step	Instructions
1	Bend up fuse leads
2	Solder wires 005 and 006 to fuse leads Add heat shrink
3	Attach fuse holder to frame using M2.5x6mm screw and M2.5 nut NOTE: see CPM Toolbox for pre-cut M2.5x6mm screws. Else, shorten M2.5 screw to 6mm using double nut technique

6. Install Connector/Switch

Part	Quantity
Female DSUB 9 Connector	1
On/Off Switch	1
M2.5x6mm Screw; M2.5 Nut; M2.5 Washer	2; 4; 2
M3x10mm Screw; M3 Nut	2; 4

Tool

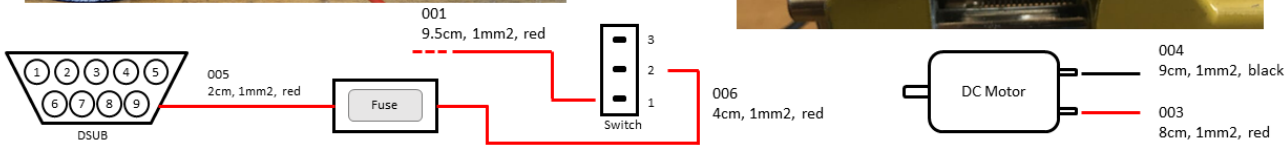
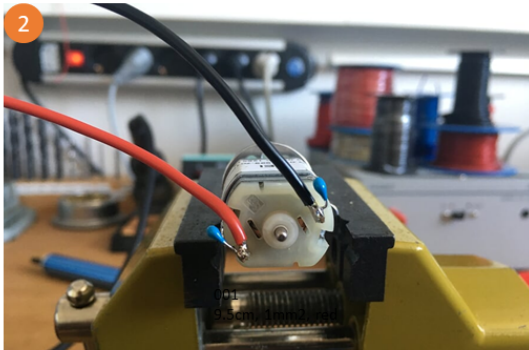
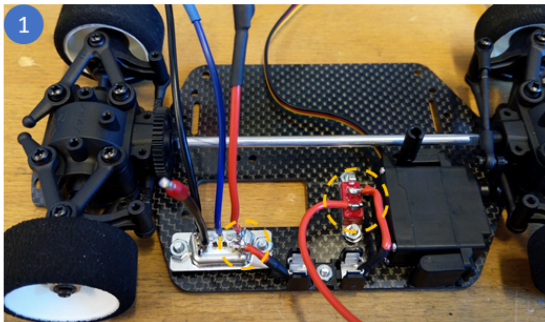


Step	Instructions
1	Attach DSUB to frame using M3 Screws, Nuts and Washers Order: Nut, DSUB, Nut
2	Attach Switch to frame using M2.5 Screws, Nuts and Washers Order: Nut, Switch, Washer, Nut Note: The lower two nuts are optional dependent on the length of the switch's arm. Normally, the nuts can be left out but the arm should not protrude too far.

7. Solder (1/2)

Part	Quantity
Wire 001, 005, 006,	

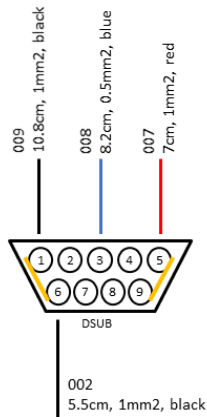
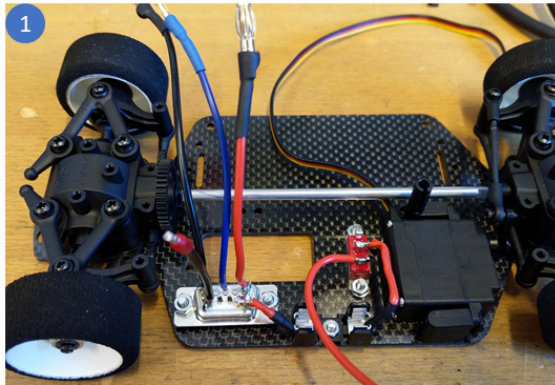
Tool
Soldering Iron



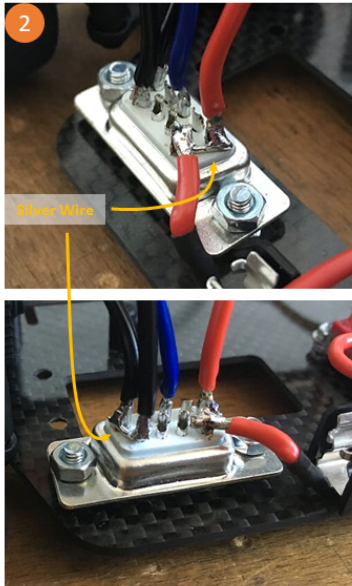
Step	Instructions
1	Solder fuse wire 005 to DSUB pin 9 Solder fuse wire 006 to Switch Pin 2 (middle) Solder wire 001 to Switch Pin 1
2	Solder wires 003 and 004 to DC motor, DO NOT install motor yet NOTE: solder wires slightly slanted, take note of polarity

7. Solder (2/2)

Part	Quantity
Wire 002, 007, 008, 009	1



Tool
Soldering Iron, Silver Wire

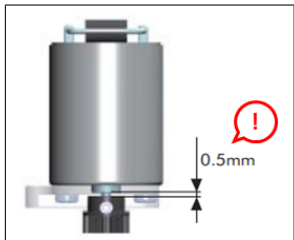
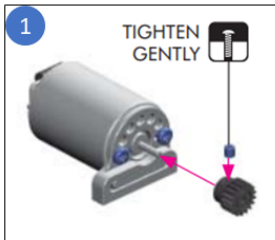


Step	Instructions
1	Solder Wire 007 to DSUB Pin 5 Solder Wire 008 to DSUB Pin 3 Solder Wire 009 to DSUB Pin 1 Solder Wire 002 to DSUB Pin 6
2	Add silver wire to connect Pin 1 and 6 Add silver wire to connect Pin 5 and 9 TIP: leave silver wire long -> spot solder to rear pin (1 & 5) -> then solder to front properly -> then re-solder back properly -> shorten wire

8. DC Motor Installation

Part	Quantity
DC Motor	

Tool
M18 Assembly Manual

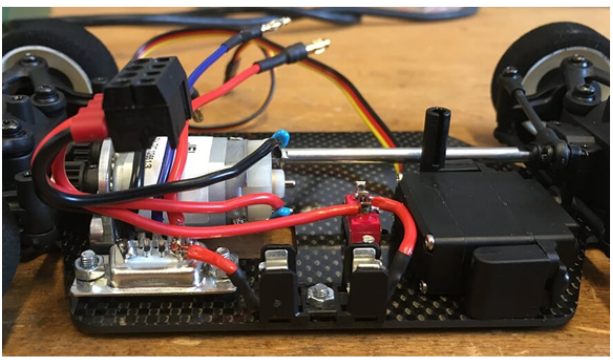
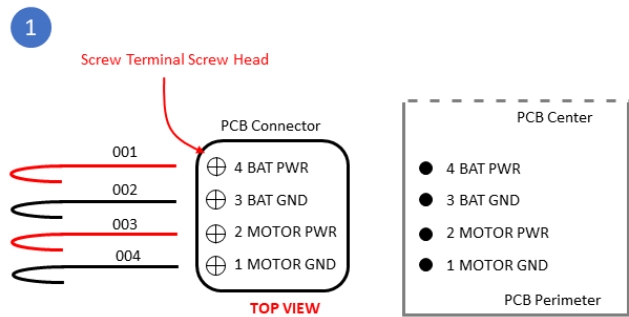


Step	Instructions
1	Install cog (pinion) onto motor shaft per manual instructions (p. 26 Step 4) NOTE: gap should be between 0.5-1mm (exact distance may be adjusted later to ensure proper distance to odometer board hall sensors)
2	Install DC Motor onto vehicle per manual instructions (p. 26-27 Step 6 and TECH TIP)

9. Connector

Part	Quantity
4-Pole Connector	1

Tool
M18 Assembly Manual

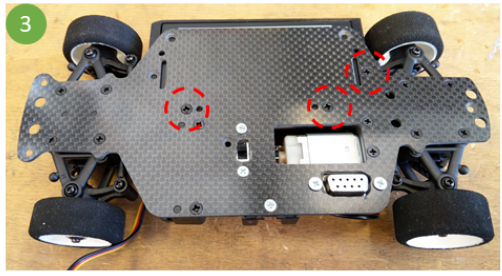
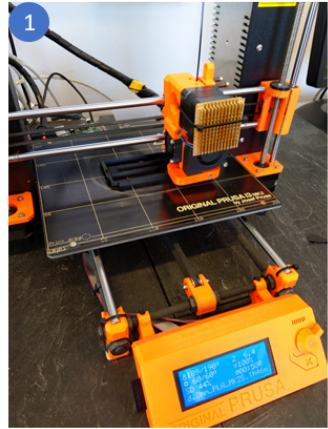


Step	Instructions
1	Attach Wire 001 to Port 4 (top view furthest left on connector) Attach Wire 002 to Port 3 Attach Wire 003 to Port 2 Attach Wire 004 to Port 1 (top view furthest right on connector) NOTE: orientation is important, check PCB pin labels to verify correct connections

10. Battery Holder (1/2)

Part	Quantity
Battery Holder	1

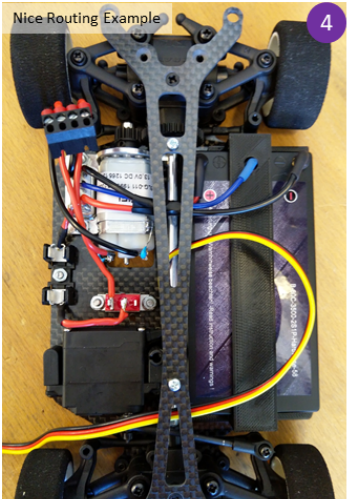
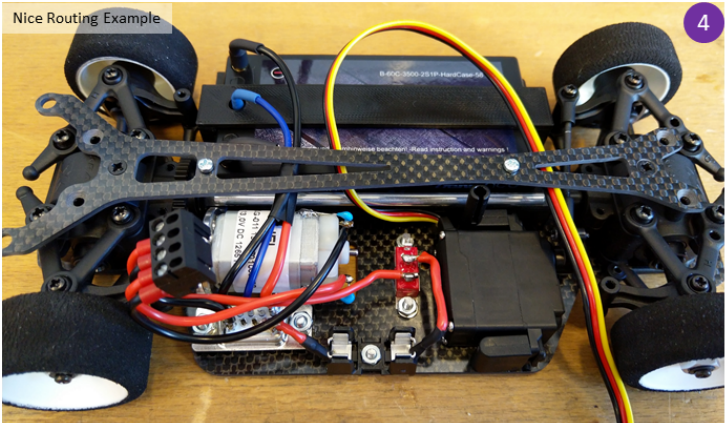
Tool
Prusa i3



Step	Instructions
1	Print Battery Holder with 3D Printer File: lab\docs\Vehicle\Version2\cad\Batterie_Halterung\Battery_Holder_1v13_0.2mm_PLA_MK2S_1h46m.gcode
2	Put the battery into the holder.
3	Use 3 2.5x8 screws to attach the battery holder and the battery to the underframe. TIP: The screws can be found in bag 3 and bag 6.

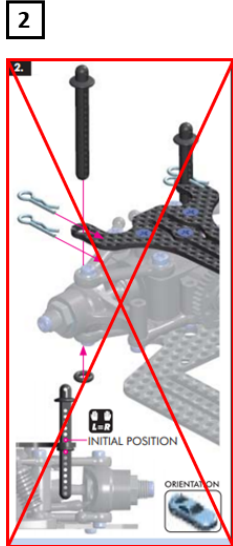
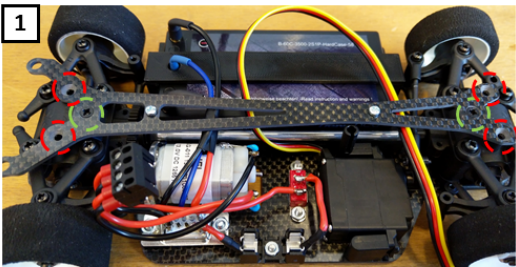
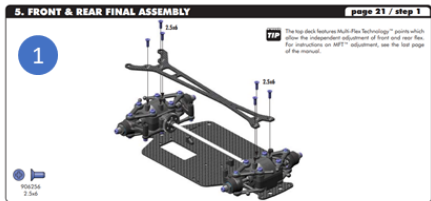
10. Battery Holder (2/2)

Tool
Electrical Tape



Step	Instructions
4	Install Battery and attach banana connectors wires 007, 008, 009 NOTE: route and bundle battery wires 007, 008, 009 nicely

11. Front & Rear Final Assembly



Step	Note
1	Go back in the manual to <i>front & rear final assembly</i> and follow the instructions on pages 20-23 by respecting the following changes.
1	Top deck differs from manual (reference Supplementary Manual) NOTE: only install 1/3 screws, remaining will be installed with LED Holder (following section)
2	Omit. Step 2 is not necessary since car body will NOT be installed

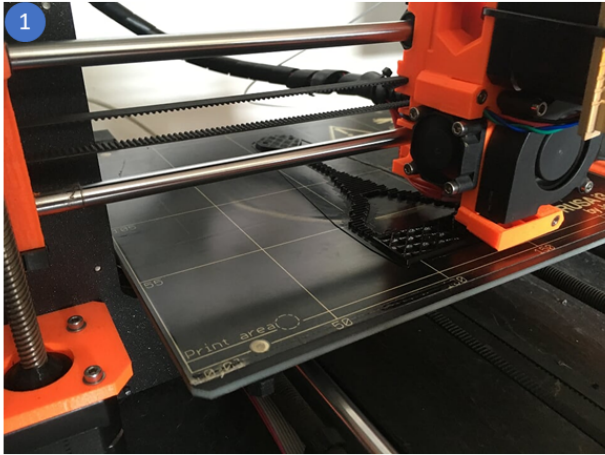
6. LED Holder

This section now describes the soldering and mounting of the LED Holder, which is needed for the [IPS](#). For the assembly of the odometer board, look [here](#).

1. 3D Print LED Holder

Part	Quantity
LED Holder	

Tool
Prusa i3



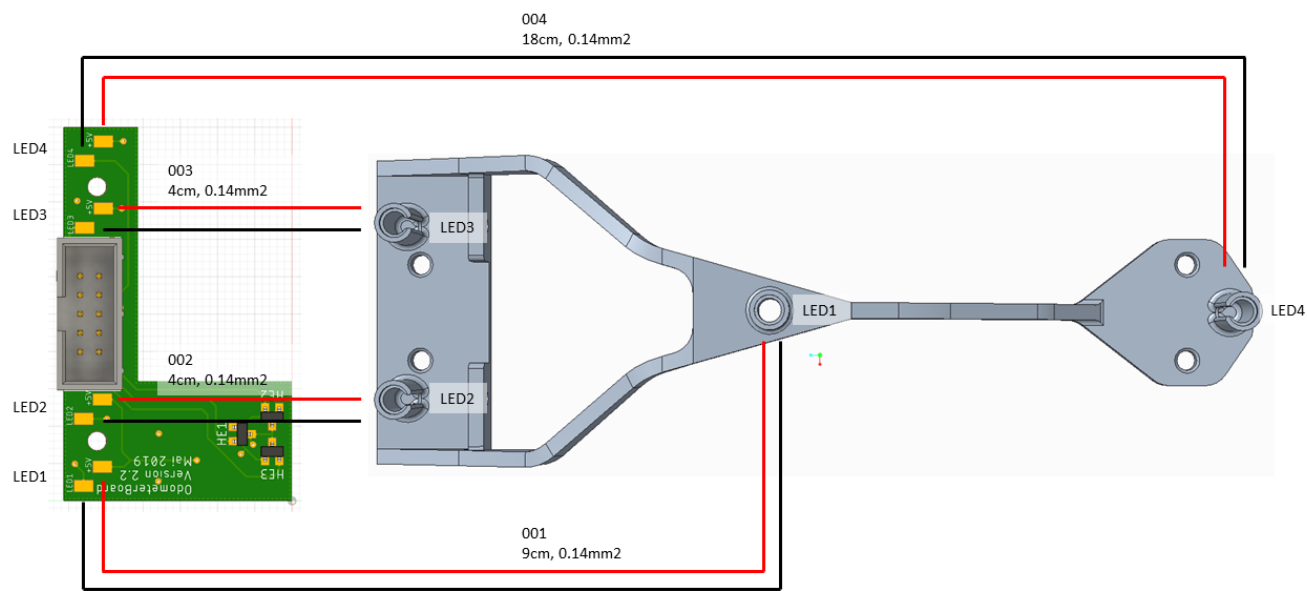
Step	Instructions
1	Print LED Holder with 3D Printer File: lab\docs\Vehicle\Version2\LED_Halterung\003-001_ledholder.stl

2. Cut Wire (1/2)

Wire PN	Length [cm]	Color	Gauge	Description	Location/Use
001	9	1X Red, 1X Black	0.14mm2	Ground and Power for LED	LED1
002	4	1X Red, 1X Black	0.14mm2	Ground and Power for LED	LED2
003	4	1X Red, 1X Black	0.14mm2	Ground and Power for LED	LED3
004	18 +0.25	1X Red, 1X Black	0.14mm2	Ground and Power for LED	LED4

Step	Instructions
1	Cut wire to correct length Note: take note of color and gauge Note: 1X red and 1X black wire per LED

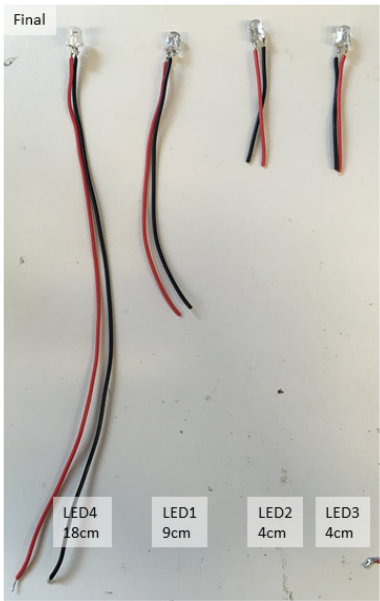
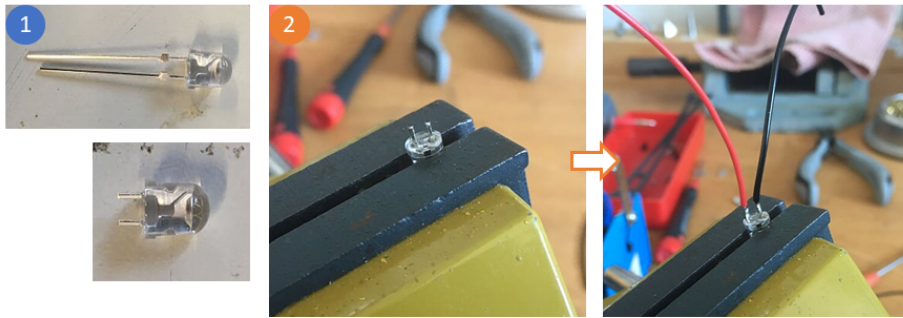
2. Overview Diagram (2/2)



3. LED Prep and Soldering

Part	Quantity
LED	4

Tool
Soldering Iron, Wire Stripper

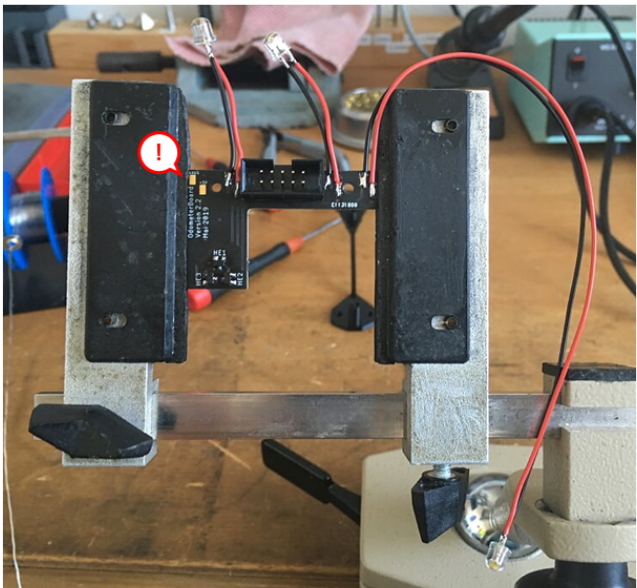
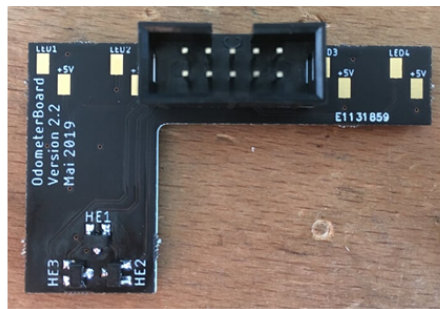


Step	Instructions
1	Short LED leads Note: remember polarity
2	Solder black and red wire to respective lead Note: if necessary, check polarity of LED with multimeter

4. Odometer PCB Soldering

Part	Quantity
LED2, LED3, LED4	1

Tool
Soldering Iron, Wire Stripper

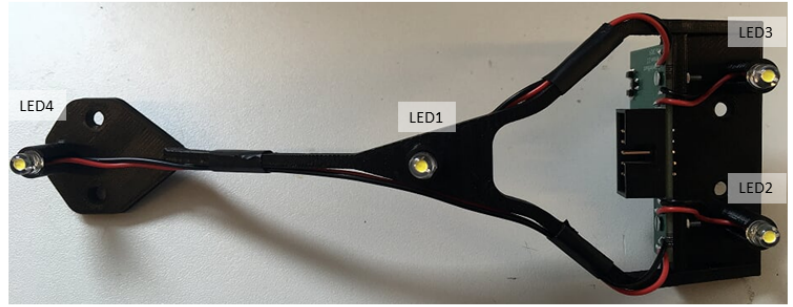
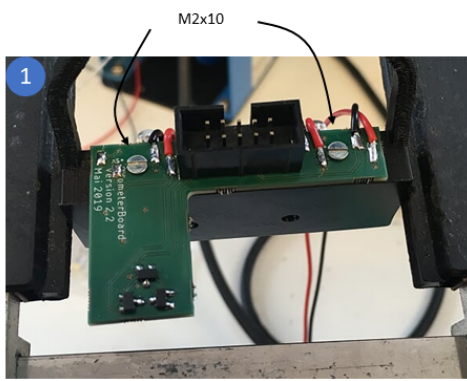


Step	Instructions
1	Solder LED2, LED3, and LED4 wires directly to PCB Open design of LED holders allows LED2-4 to be soldered prior to PCB attachment NOTE: LED1 needs to be soldered after wires are inserted through thru-hole

5. Odometer PCB Attachment (1/2)

Part	Quantity
M2x10 Countersink Screw, M2 Nut	2

Tool
Soldering Iron, Wire Stripper

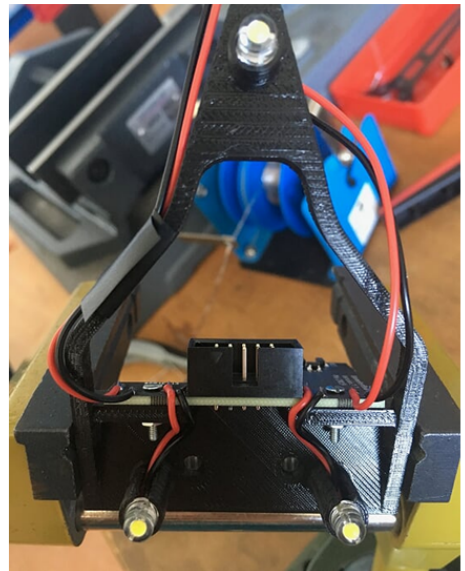
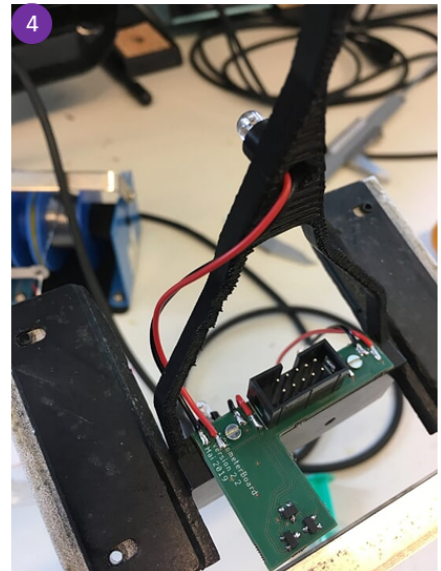


Step	Instructions
1	Attach Odometer PCB to LED Holder using M2x10 Countersink Screws and M2 Nuts
2	Route LED2-4 wires to appropriate holders

5. Odometer PCB Attachment (2/2)

Part	Quantity
LED1	1

Tool
Soldering Iron, Wire Stripper

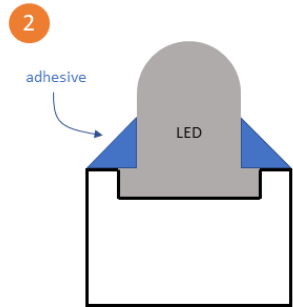


Step	Instructions
3	Route LED1 through thru-hole
4	Solder LED1 to Odometer PCB

7. LED Adhesive

Part	Quantity

Tool
2-Part Epoxy Adhesive



Step	Instructions
1	Use electrical tape or adhesive to fixate/route LED wires nicely
2	Use adhesive to glue LEDs into holder

[Contents](#)

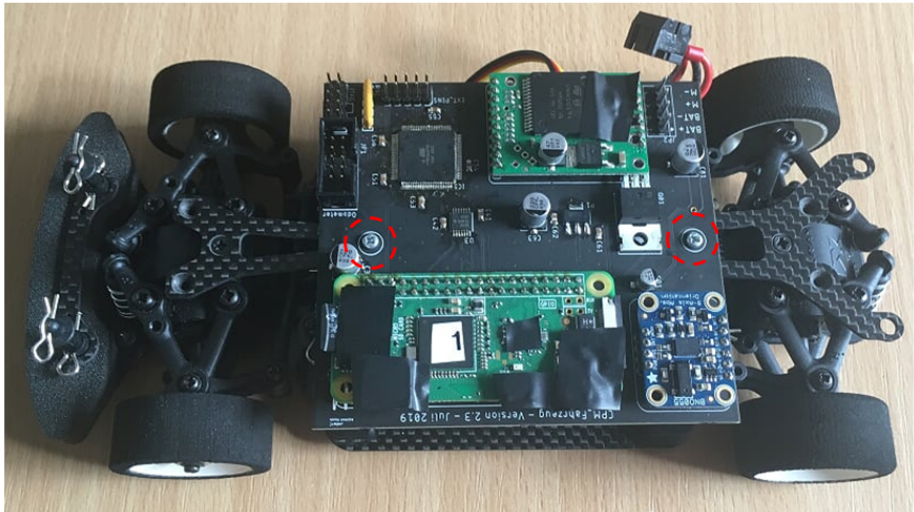
7. Final Assembly

In this last section, the Main Board and the LED Holder will be attached and wired to the chassis. If the Main Board is not soldered yet, see [here](#).

1. Attach PCB

Part	Quantity
M2x6 Phillips Screw	2

Tool



Step	Instructions
1	Attach PCB to flex board using M2x6 Phillips screws

2. IDC Ribbon Cable

Part	Quantity
IDC Ribbon Cable, IDC Connector	1; 2

Tool



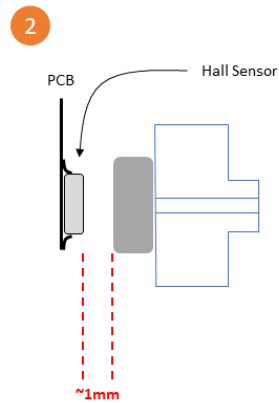
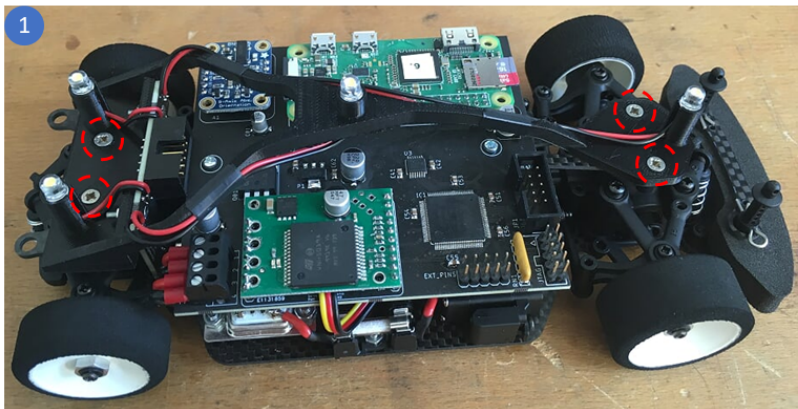
10.75cm

Step	Instructions
1	Cut IDC Cable to 10.75cm + 0.25cm
2	Attach connectors using vice Note: connector orientation important

3. LED Holder Sub Assembly Attachment

Part	Quantity
2.5x10 SPAX	4

Tool
M18 Assembly Manual

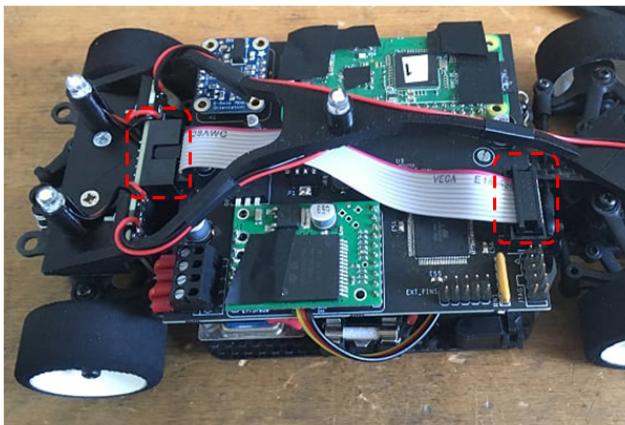
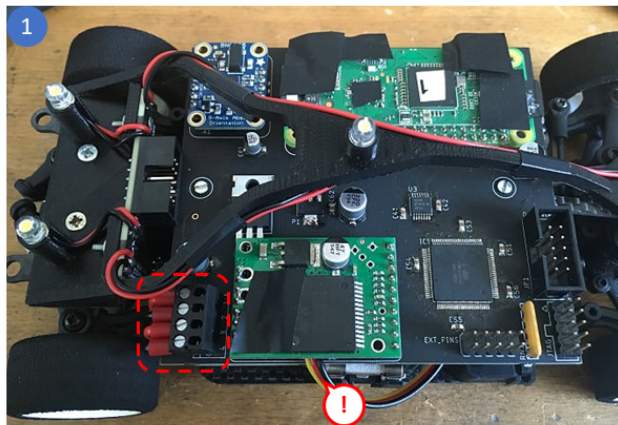


Step	Instructions
1	Attach LED Holder using 2.5x10 SPAX screws
2	Check that distance between motor cog magnet and hall sensors ~1mm
3	If necessary, loosen motor cog fixation screw to adjust distance Note: may need to remove LED Holder, PCB and Flex Plate to adjust distance

4. Connect PCB

Part	Quantity
IDC Ribbon Cable	1

Tool



Step	Instructions
1	Connect battery connector and servo connector to PCB Note: take note of servo connector direction: yellow left -> black right
2	Connect ribbon cable to PCB and LED Holder

Raspberry Pi Setup

- Install [Raspbian Lite](#). We currently use Debian 10 (Buster).
- Change password by using

```
passwd pi
```

- Activate SSH

```
sudo systemctl enable ssh  
sudo systemctl start ssh
```

- Install and configure [NTP](#)
- Install tmux

```
sudo apt install tmux
```

- "Install" the RTI Connex libraries under /usr/local/lib/ by copying the arm-folder from the Main PC under opt/rti_connex_dds-6.0.0/lib/armv6vfphLinux3.xgcc4.7.2/. This can of course only be done after the ARM libraries have been installed on the [Main Computer](#).

It should look like this afterward:

```
pi@raspberrypi-06:~ $ find /usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscppd.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscd.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndstransporttcpzd.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscpp2z.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlcppz.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtimonitoringd.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndstransporttcp.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscpp2.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscpp2d.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtimonitoringz.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndstransporttcpd.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscz.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscppzd.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndsc.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndstransporttcpz.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlcpp.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscpp.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscore.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtimonitoringzd.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscorez.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscppz.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscored.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlcppd.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndsjava.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscpp2zd.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndsczd.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndscorezd.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/libnndsjavad.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtimonitoring.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlcz.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlcppzd.a  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlc.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlcd.so  
/usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2/librtidlczd.a
```

'Register' the Libraries

Make the libraries available. Insert /usr/local/lib/rti_connex_dds-6.0.0/armv6vfphLinux3.xgcc4.7.2 into /etc/ld.so.conf. Then run

```
sudo ldconfig
```

Configure Bootup Behaviour

Copy the file `git:software/vehicle_raspberry_firmware/bootloader_raspberry.bash` to `/root/bootloader_raspberry.bash` on the Raspberry.

Enable autostart, using `sudo nano /etc/rc.local` and insert

```
tmux new-session -d -s "bootloader_raspberry" "bash /root/bootloader_raspberry.bash"
```

NTP configuration

See the NUC [NTP-Configuration](#).

ID-Related Settings

Edit the `/etc/hostname` file to give the Raspberry a unique name. The name should correspond to the vehicle ID, for example `raspberrypi-06` or `raspberrypi-42`.

Create a DHCP reservation for the Raspberry on the router. The vehicle ID is derived from the IP. The IP is `192.168.1.1XX` where `XX` are the digits of the vehicle ID.

Read-Only Filesystem

One crucial aspect regarding the Raspberry Pi is, that the vehicle will be turned off by using the switch on the bottom, mostly. That will directly cut off the power supply of the RPi. In order to avoid damage to the OS, it is important to configure the filesystem to be read-only. For that, follow the instructions on <https://kofler.info/raspbian-lite-fuer-den-read-only-betrieb/>. (As alternative <https://www.mehr4u.de/raspberry-pi-mit-readonly-filesystem.html> may also be used. Vehicle 20 is created by following the first link but using `fastboot noswap ro` instead of `fastboot noswap`.)

In order to make NTP working properly with a read-only filesystem, it is also necessary to copy the following file

```
cp /lib/systemd/system/ntp.service /etc/systemd/system/ntp.service
```

and to comment out `PrivateTmp=true` in

```
/etc/systemd/system/ntp.service
```

If you need *writing access* to the file system, you can use (dependent on the folder you need access to)



```
sudo mount -o remount,rw /
sudo mount -o remount,rw /boot
```

to remount the system.

Applying RT-Patch

Because the vehicle needs to be real-time capable, a real-time patch (RT-Patch) should be applied to the underlying Linux Kernel.

Follow [these](#) instructions to build the kernel. It is recommended to use the section *Cross-Compiling* instead of *Local Building* to speed up the process (e.g. ~15 minutes vs. ~2 hours). Additionally, there has to be done several annotations:

- **Get Sources:**
 - Make sure that there exists a real-time patch for the current kernel version. You can see this version by looking at the first lines of the makefile residing in the cloned directory, e.g., by using `head Makefile -n 4`. The currently existing real-time patches can be found [here](#).
- **Before Build sources**
 - Download the RT-Patch matching your kernel version and patch the kernel according to [these instructions](#). (Execute the patch-command in the root directory of the cloned git repository.)
- **Build sources**
 - Run the commands which depend on the RPi version.
 - Run `make menuconfig`, navigate to `General Setup > Preemption Model`, choose `Fully Preemptible Kernel (RT)` and save.

- Run the lastly given make command.
 - Make sure to use the parameter `-j n` to configure how many threads are used for this command in order to speed up this process as explained.
 - It might happen that you are **asked to set up** the `.config` file again which you already configured in the last step. (The console will show something like `*Restart config...` and asks you questions.) Then press `strg+c` and replace the command by `make ARCH=arm menuconfig CROSS_COMPILE=arm-linux-gnueabihf- zImage modules dtbs`
Now, the menu will open again, but you can quit that by only **?saving?** and leaving the menu. Afterwards, the process should start building as usual.

Troubleshooting

It might happen that the system clock is not automatically synchronizing via NTP if it differs too much from the real-time. Firstly, take care that your timezone is set correctly:

```
sudo raspi-config
```

In the opening UI go to: 4 Localisation Options I2 Change Time Zone None of the above UTC

Furthermore, you can force NTP to set the time once. Therefore, stop the service, force it, and restart it again :

```
sudo /etc/init.d/ntp stop
sudo ntpd -qq
sudo /etc/init.d/ntp start
```

Afterwards, NTP should synchronize automatically.

Repeating the process for several vehicles

In order to create identical vehicles it might be useful to clone the SD-cards by using [Clonezilla](#). Afterwards, only the [ID-Related Settings](#) have to be adapted at the cloned SD-card.

Cross Compilation and Software Distribution

The vehicle Raspberry software is cross-compiled from the master PC. The build script `git:software/vehicle_raspberry_firmware/build.bash` creates an archive of the compiled software and other required files and publishes it via [Apache](#). The `boot_loader_raspberry.bash` autostart script on the Raspberry downloads and runs this software.

ARP Table Generation

Because unsuccessful ARP requests sometimes block communication a static ARP table is sometimes useful. To generate one one needs to switch on all participants, ping each one from the main PC and then output the arp table on the main pc with e.g. <https://man7.org/linux/man-pages/man8/ip-neighbour.8.html> (newer) or `arp -a` (old).

The info can then be used to feed a script that sets up the arp entries statically for the participants. Currently this is done in https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/feature/fastdds/mid_level_controller/package/start.bash and the static arp table is part of the package.

ATmega Flashing

Please follow steps to flash ATmega with current CPM ATmega Firmware.

- Current Firmware: https://github.com/embedded-software-laboratory/cpm_lab/tree/master/low_level_controller
- For questions, please contact Patrick Scheffe

Steps require Main Vehicle PCB to be connected to vehicle to provide power

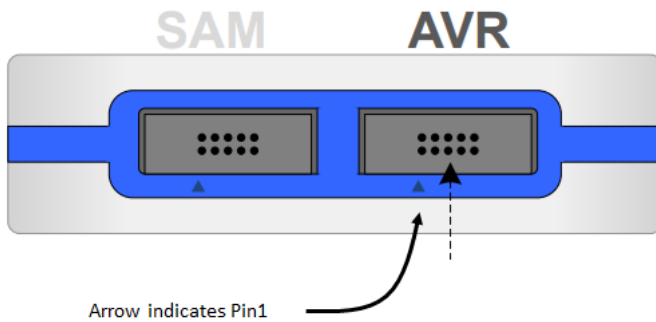
1. Atmel Studio 7.0

1. Start Atmel Studio 7.0
2. Load current vehicle firmware solution "vehicle_atmega2560_firmware.atsln"

2. ICE Programmer

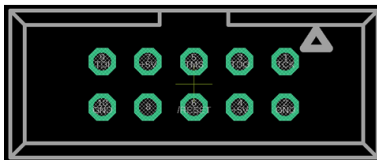
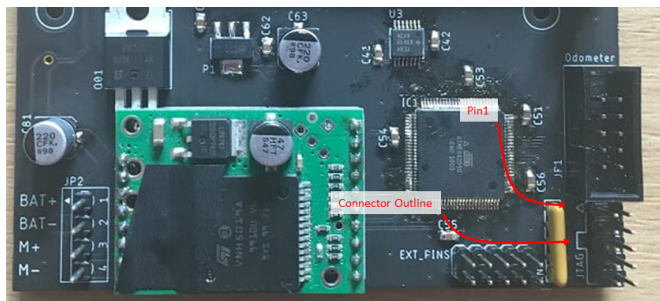
Make sure vehicle is powered off before connecting the programmer!

1. Check pinout of Programmer, Vehicle and Connector Cable:



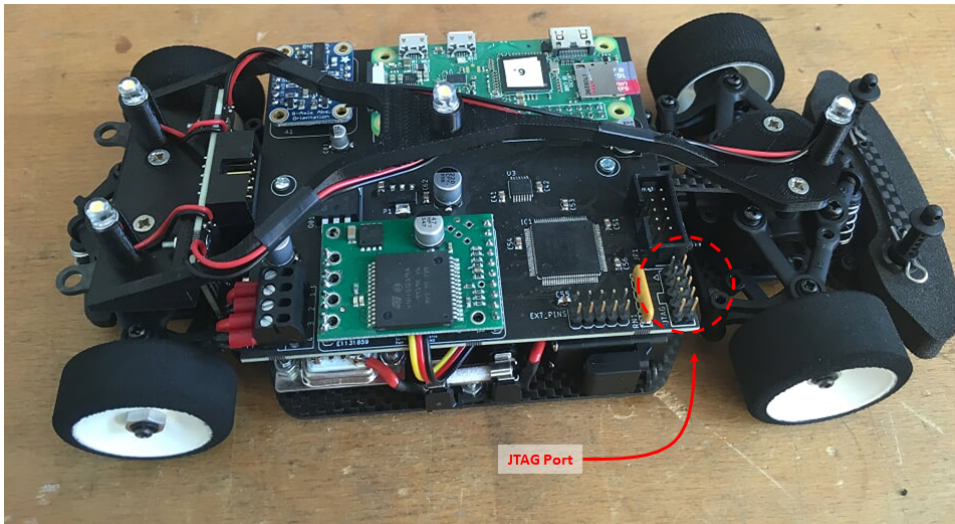
Pinout:

2	4	6	8	10
1	3	5	7	9



2. With vehicle power OFF, connect ICE programmer AVR Port to JTAG header.

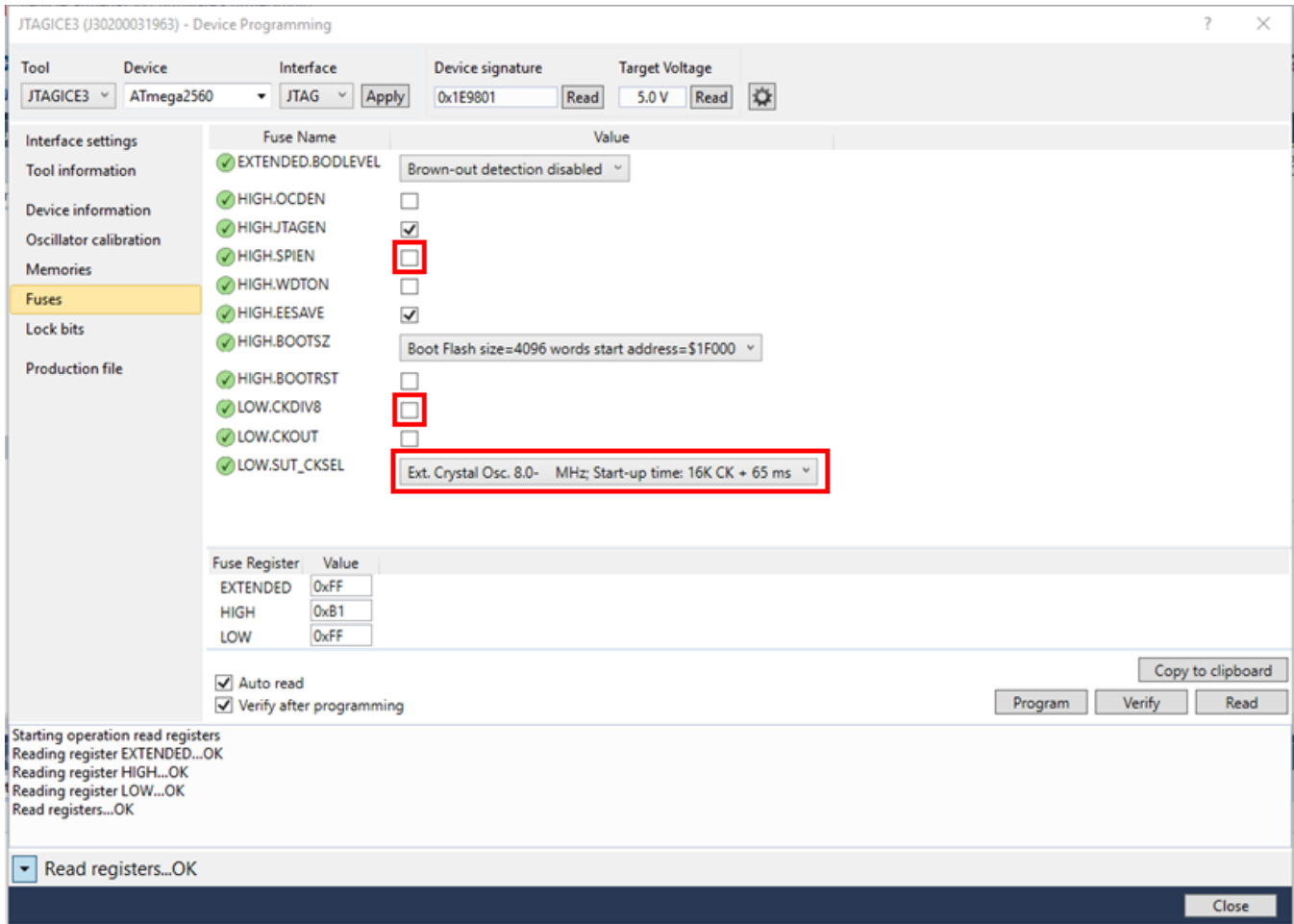
3. Connect ICE programmer with USB cable to PC.



3. Fuses

1. Turn ON vehicle
2. Connect to ATmega Chip with *ToolsDevice Programming*
 - a. Click *Apply*
 - b. Click *Read*
3. Set the correct fuses (see table and screenshot below)
4. Click *Program*

Fuse	Correct Setting
HIGH.SPIEN	Unchecked
HIGH.EESAVE	Optional i.e. checked or unchecked
LOW.CKDIV8	Unchecked
LOW.SUT_CKSEL	Ext. Crystal Osc. 8.0- MHz; Start-up time: 16 CK + 65ms



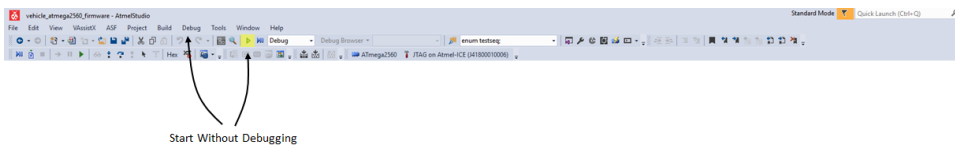
If vehicle is behaving irregularly i.e. servo steering is acting erratic, LEDs timing appears off, this is most likely due to an issue with the clock.



1. Check the above settings again.
2. Check external crystal oscillator on bottom side of board using oscilloscope (should be 16MHz)

4. Flash ATmega

1. Flash chip using Start without Debugging



2. Turn OFF vehicle
3. Disconnect programmer

5. Tests

By setting a jumper, a test mode is activated. This allows testing the inputs and outputs without the MLC.

Your browser does not support the HTML5 video element

Current Servo Setup

As described in [Vehicle Assembly](#) each servo should be configured such that the mechanical limits of its vehicle are utilised fully. Since it is technically infeasible to build the vehicles exactly the same, each vehicle will slightly differ in terms of at which servo position it moves completely straight or steers as far as possible to the left or the right. In order to assure that the vehicles have nearly the same moving behaviour, we put up with configuring the vehicle's servos differently. By doing this with the built-in end-point-adjustment (EPA) via the DPC-11 we facilitate that our software does not have to manage the vehicles differently. Instead, the minimal/central/maximal PWM signal corresponds to the left-most/straight/right-most steering command which has been set up via EPA. As a consequence, it is possible to have a different resolution in one servo from left to center and from center to right. Since the servo arm moves on a circular path and the relation between the PWM signal and the steering angle is thus non-linear (even without different resolutions in the settings), we assume the EPA with different resolutions to the left and right to be compensating the non-linearity implicitly. Consequently, we don't manage the non-linearity and the different resolutions explicitly.

The current EPA configurations are listed below:

Vehicle ID	LEFT	CENTER	RIGHT
01*	1480	2500	3700
02	1435	2550	3705
03	1415	2520	3800
04	1333	2470	3668
05*	1510	2650	3780
06*	1625	2625	3785
07	1440	2550	3720
08	1465	2520	3655
09*	1500	2550	3675
10	1340	2415	3640
11	1570	2600	3780
12	1225	2200	3445
13	1470	2500	3645
14	1485	2535	3735
15	1500	2600	3820
16	1240	2330	3420
17	1380	2445	3570
18	1555	2585	3750
19	1495	2500	3690
20	1480	2520	3700

*Due to possible mismatches between the main board, the chassis, and thus the original ID of a chassis when reassembling the vehicles, the values for these vehicles might be incorrect/permuted.

Cloning Disks with Clonezilla

In case that there are several devices which have to be set up identically (e.g. RPis, NucS, ...), it might be helpful to just clone memory disks, once the first one is set up completely. Clonezilla is a tool, addressing exactly this. The following describes three ways of using Clonezilla.

Clonezilla is an operating system and can thus be used via a bootable USB-stick. In the CPM Lab you will find a black one which contains this already. (PICTURE)

Start Clonezilla at the OptiPlex 3070

To start Clonezilla you might have to change some properties in the BIOS. Plug in the USB-Stick containing Clonezilla and start the computer. You will get into the BIOS by pressing F2.

1. Move to *Secure Boot* > *Secure Boot Enable* and deselect *Secure Boot Enable*.
2. Move to *General* > *Boot Sequence* and click on *Add Boot Option*. Choose a name, select the file */EFI/boot/grubx64.efi* and confirm it. (Depending on the used Clonezilla version and the underlying BIOS the needed file and its path differ. For more information see [here](#).) Move Clonezilla to the highest priority, apply the changes and exit. After rebooting, Clonezilla should start.

Possible Procedures

- [Device Device](#)
- [Device Image](#)
- [Image Device](#)

Device Device

Identify names of memory disks

In this explanation, one SD-card will be cloned to another one directly. Consequently, both SD-cards have to be plugged in at the same time. Because you will need to specify which of them is the source and which the target, you need to identify them clearly. For that purpose, plug them into the computer and open *Disks* (Ubuntu). Their names will be the same in Linux and Clonezilla.

Note: Something like *sdc* is NOT the name of the device. This one changes dependent on the order in which the devices are plugged in.

Navigate through Clonezilla

If you want to exit Clonezilla in the start menu, switch to console by entering "c" and then enter "exit". (The next OS specified in the Boot Sequence of the BIOS will start.)

After Clonezilla has been started successfully, you have to navigate through the "submenus" it consists of. Choose

- the first option *Clonezilla live (Default settings, VGA 800x600)*. If you don't do anything this will be chosen automatically after plenty of seconds.
- the language and the keyboard layout as you want (default recommended).
- *start Clonezilla*
- *device-device*
- *expert*
- *disk to local disk*
- the **source** as identified in the beginning
- the **target** as identified in the beginning
- toggle the options as follows by using Space. Deactivate everything except the next two options which should be set:
 - *sector-by-sector copy (-q1)*
 - *inspect checksums (-cmf)*
 - *do not update initramfs on restored GNU/Linux (-iui)*
- keep default (*skip checking/repairing source file system*)
- keep default (*use the partition table from the source disk*)
- *poweroff*

All data, which was saved at the target device, will be lost afterwards!

Finally, confirm everything (warnings, and clone of boot loader) with *y*.

Depending on the size of the devices the cloning process will take up to a few hours.

Device Image

This paragraph addresses the case, in which you want to create an image of an already existing device.

Navigate through Clonezilla

If you want to exit Clonezilla in the start menu, switch to console by entering "c" and then enter "exit". (The next OS specified in the Boot Sequence of the BIOS will start.)

After Clonezilla has been started successfully, you have to navigate through the "submenus" it consists of. Choose

- the first option *Clonezilla live (Default settings, VGA 800x600)*. If you don't do anything this will be chosen automatically after plenty of seconds.
- the language and the keyboard layout as you want (default recommended).
- *start Clonezilla*
- *device-image*
- *local dev*
- if not already done, insert the device with your image into the computer, wait five seconds and press enter
- wait until you see the devices you expect and quit with STRG-C
- choose the device you want to read the image from
- select the folder from which you want to read the image
- *expert*
- *savedisk*
- choose a name (e.g. keep default)
- choose disk as source (by Space)
- choose default (-q2 partclone > partimage > dd)
- select "Generate image MD5 checksums" (-gm) and "Image not only accessible by owner" (-noabo) (and let -c and -j2 also selected)
- select default (-z1p Use parallel gzip compression, for multicore/CPU)
- select default split size
- select default (skip checking/repairing source file system)
- select default (yes, check the saved image)
- select default (not to encrypt the image)
- *poweroff*

Finally, confirm everything by enter if you are sure.

Image Device

In case that you already have an image, you can directly clone this image onto your device.

Navigate through Clonezilla

If you want to exit Clonezilla in the start menu, switch to console by entering "c" and then enter "exit". (The next OS specified in the Boot Sequence of the BIOS will start.)

After Clonezilla has been started successfully, you have to navigate through the "submenus" it consists of. Choose

- the first option *Clonezilla live (Default settings, VGA 800x600)*. If you don't do anything this will be chosen automatically after plenty of seconds.
- the language and the keyboard layout as you want (default recommended).
- *start Clonezilla*
- *device-image*
- *local dev*
- if not already done, insert the device with your image into the computer, wait five seconds and press enter
- wait until you see the devices you expect and quit with STRG-C
- choose the device you want to read the image from
- select the folder from which you want to read the image
- *expert*
- *restoredisk*
- choose image file
- select target device
- select default options (-g auto, -e1 auto, -e2, -c, -r, -j2)
- keep default (use the partition table from the image)
- keep default (Yes, check the image before restoring)
- the **source** as identified in the beginning
- the **target** as identified in the beginning
- toggle the options as follows by using Space. Deactivate everything except the next two options which should be set:
 - *sector-by-sector copy* (-q1)
 - *inspect checksums* (-cmf)
- keep default (skip *checking/repairing source file system*)
- keep default (*use the partition table from the source disk*)

- *poweroff*

All data, which was saved at the target device, will be lost afterwards!

Finally, confirm everything by enter if you are sure.

Doxygen Setup

Why Doxygen?

The Confluence Documentation you are looking at right now only serves the purpose of a "high-level" doc - it does provide you with code snippets, tutorials, troubleshooting etc., but it does not include detailed code-specific information. We decided to use Doxygen for that instead.

Most of our code should be documented properly using Doxygen commands. You only need to create the Doxygen page from the already existing Doxygen configuration file. The following sections tell you how to do that, and where to find that documentation.

Prerequisites

Please install the Doxygen and Graphviz (for graphs drawn in Doxygen pages) packages, e.g. on Ubuntu:

```
apt install doxygen graphviz
```

The file output is set to `../doxygen`, so maybe the folder must be generated first. If the software repository of this project is in `~/software`, create a folder in `~/doxygen`.

Creating your Documentation

We do not host our Doxygen documentation on a separate server. As it is very easy to generate the Doxygen sites yourself, you need to create them locally instead.

First, enter the directory of this project, which usually is `~/dev/software`:

```
cd ~/dev/software
```

The `doxygen_config` file should be present in this folder. All you need to do is run the following command to create the doxygen output in the `../doxygen` folder (relative from your current location):

```
doxygen ../doxygen_config
```

As soon as the command has finished, head over to the doxygen folder:

```
cd ../doxygen
```

There you can find different `.html` as well as other files. Just open `~/dev/doxygen/html/index.html` in your browser to view and navigate through the generated documentation.

Settings

Some notes on settings in the Doxygen config file can be found below. All mentioned keywords have been used. For some, additional information on why you might want to change them are added.

INPUT (currently redundant, as `./` is given in the end; only `./LICENSE` is relevant additionally, because else the file without ending would not be included due to the set file patterns)

RECURSIVE

FILE_PATTERNS

GENERATE_TREEVIEW

EXTRACT_STATIC (set to YES)

PROJECT_NAME

PROJECT_BRIEF

PROJECT_LOGO

OUTPUT_DIRECTORY

For struct definitions with attribute in the `low_level_controller`, to recognize the struct name I had to enable:

ENABLE_PREPROCESSING = YES

MACRO_EXPANSION = YES

EXPAND_ONLY_PREDEF = YES

PREDEFINED = __attribute__(x)=

Set to yes:

EXTRACT_PRIVATE

EXTRACT_PRIV_VIRTUAL

Set `EXTRACT_ALL` to YES if you want undocumented files to appear in the documentation as well - currently: NO

`USE_MDFILE_AS_MAINPAGE` can also be used instead of the current `DoxygenMainpage.hpp` - currently: NO

`EXCLUDE` to exclude `bcm2835.h`, as it contains its own mainpage, as well as the `catch.hpp/cpp` files, and most `.md` files, and most package files

`SORT_BRIEF_DOCS` was set to YES by me - even though this also changes the class list, it is irritating to see functions in a group overview in a non-alphabetical order, so I changed that

System Test

CPM Lab

Simulated Setup

1. [Start the LCC](#).
2. On the right, go to the *Setup* tab
3. Simulate some vehicles (e.g., vehicles 1, 4, 7, and 10).
4. In the script path, select the HLC <repo>/high_level_controller/examples/cpp/central_routing/build/central_routing (substitute <repo> with the path to the CPM Lab repository).
5. [Run the HLC](#)

Physical Setup

This test is designed to test most of the lab's infrastructure, including the Indoor Positioning System, the Lab Control Center and the Trajectory Tracking Controller on the Raspberry Pi. Repeat the test above but with real vehicles. Consult the guide for doing [Physical Experiments](#).

MATLAB with the CPM Lab

MATLAB can be tested with an example HLC.

1. [Start the LCC](#).
2. On the right, go to the *Setup* tab
3. Simulate a vehicle (e.g., vehicle 7).
4. In the script path, select the HLC <repo>/high_level_controller/examples/matlab/01_direct_control (substitute <repo> with the path to the CPM Lab repository).
5. [Run the HLC](#)

Getting Started

- [Lab Software Preparation](#)
- [Lab Control Center Introduction](#)
- [Vehicle Commands](#)
- [Physical Experiments](#)
- [Running an HLC](#)

Lab Software Preparation

In case you are unsure that the most recent build process performed on your PC built your desired version of the CPM Lab software, follow these steps.

Open a terminal, navigate to the software repository and checkout your desired branch or tag. If unsure, checkout the master branch. If there are updates available, download them, clean the repo and build the software again. Substitute `**<BRANCH\_OR\_TAG>**` in the following code!

```
cd ~/dev/software
git fetch
git checkout <BRANCH_OR_TAG>
rm -rf cpm_lib/dds_idl_cpp
rm -rf cpm_lib/dds_idl_matlab
```

If you are working on the Main PC of the lab, build the software without the `--simulation` switch. Otherwise, build all the necessary files to simulate the CPM Lab:

```
bash build_all.bash --simulation
```

What to Consider After an Update

Some common problems arise after an update. The .idl files for the communication might have changed, and the files for the NUCs or vehicles might be different as well. Thus, if you pull a new version (of the master), make sure that you do the following:

- **Delete** the auto-generated files created from the .idl files. You can find them in `cpm_lib/dds_idl_cpp` and `cpm_lib/dds_idl_matlab`. We do not provide a mechanism that looks for updates in the .idl files, and thus, if they have been updated, do not re-generate them if old versions are still present. If they do not get updated, the build or communication might fail.
- **Re-build** the `cpm_lib`, Lab Control Center, Mid Level Controller, Middleware etc. The `build_all` script should cover all these steps.
- For the NUCs: Make sure that the NUC packages were actually updated properly. You can find out more [here](#) in *Provide all required packages*.

Lab Control Center Introduction

Starting the Lab Control Center

You can start the Lab Control Center (LCC) from ubuntu's application menu, or from the terminal:

```
cd dev/software/lab_control_center/  
./run.bash
```

After you successfully started the LCC software, you can get familiar with the UI looking [here](#).

The [LCC](#) is a GUI that allows monitoring and control of various aspects of the lab. It has two main purposes,

1. to start and stop an experiment, and
2. to visualize the current state of the experiment.

[Read more about the LCC on its dedicated page](#).

Vehicle Commands

There are three ways to control CPM vehicles. They take precedence in the following order:

1. Direct Commands
2. Path Tracking Commands
3. Trajectory Commands

That means if a vehicle receives both Direct Commands and Trajectory Commands, it will perform the Direct Commands.

Direct Command

See https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/dds_idl/VehicleCommandDirect.idl

The direct command uses dimensionless, uncalibrated inputs and applies them directly to the motor and servo.

	motor_throttle	steering_servo
-1.0	Maximum reverse acceleration	Full right
0.0	Brake, stop	Steering roughly centered, but may have an offset
+1.0	Maximum forward acceleration	Full left

Path Tracking Command

The vehicles are controlled in path tracking mode with [PathTracking-Messages](#).

The inputs in Path Tracking Mode are a path consisting of poses and distances. The distance describes the distance along the path from the first path point to the current path point. Between path points, an interpolation with cubic splines is done on the basis of the poses and the distances. If you understood the Trajectory Command structure, a path is basically a trajectory with the speed of 1.

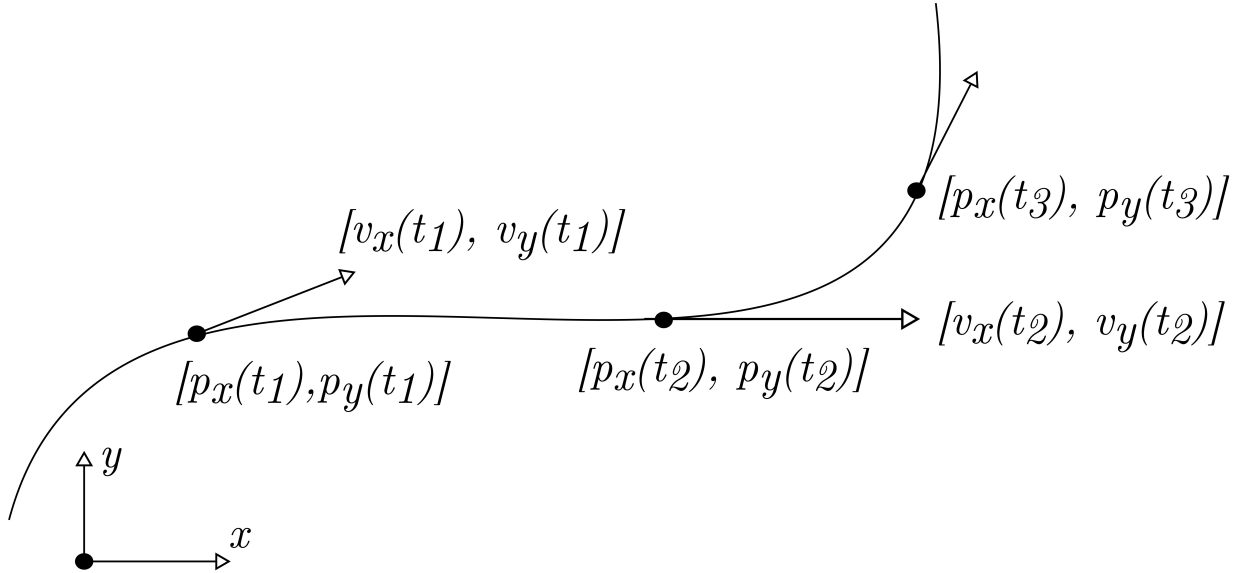
Trajectory Command

See https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/dds_idl/VehicleCommandTrajectory.idl

The reference trajectory is defined as a **Cubic Hermite spline**. A spline is a function that is defined **piecewise** in time. Thus it can be extended in real time, which (metaphorically) looks like this:



The points defining the spline must be given as a set of five numbers $[t_i, p_x(t_i), p_y(t_i), v_x(t_i), v_y(t_i)]$, where t_i is the **absolute time** in nanoseconds of the trajectory point, $[p_x, p_y]$ is the position vector (meters) and $[v_x, v_y]$ is the velocity vector (m/s).



The time between trajectory points $t(i+1) - t(i)$ is not fixed, but should be between 100 ms and 400 ms. Due to network latency and the controller structure, the trajectory points need to be created and sent ahead of time. The exact required lead time is TBD, but 1000 ms should be enough.

Troubleshooting

 Check the [Logs \(Tab\)](#).

The vehicle does not start driving.

When first taking control of the vehicle using a reference trajectory, the reference trajectory must be consistent with the current position and orientation of the vehicle. Small deviations ($\sim 0.5\text{m}$, ~ 30 degrees) are allowed. For large deviations the trajectory controller will deactivate and stop the vehicle.

The vehicle stops driving.

The reference trajectory is a function of continuous time. Thus it also implicitly defines other kinematic quantities, such as yaw, yaw rate, curvature, acceleration and jerk, for any point in time. The user must ensure that the reference trajectory is within the limits of the vehicle's capabilities. The curvature should be less than 3/m ("three per meter"), and the acceleration should be less than 5 m/s². If these limits are significantly exceeded, the trajectory controller will deactivate and stop the vehicle.

Physical Experiments

Prepare the CPM Lab

1. Start the NUCs and switch with the white remote control by turning on the wall plug number 1.
2. Start the router with the white remote control by turning on the wall plug number 2.
3. Start the [Computation Hardware](#).
4. Log in as cpm-stud.
5. [Build your desired version of the CPM Lab](#).
6. [Start the LCC](#).

Experiment

Simulated

Make sure that your HLC works in simulation. Simulate your desired vehicles, choose your desired script and follow [this guide](#) to start the experiment. All real vehicles need to be turned off for this, as they share the DDS interface with simulated vehicles.

Real

If you want to record a video of your experiment

1. Toggle the *Record* switch in the *Lab Camera* section of the *Setup* tab of the LCC.
2. After you stop the experiment, you find the video in `/tmp/cpm_lab_recordings/`

Continue to

1. Turn the vehicles on and position them in the driving area.
2. Turn on *lab mode* in the *Setup* tab of the LCC to activate the IPS.
3. Wait until all vehicles are shown at their correct poses in the LCC.
4. Choose your desired script and follow [this guide](#) to start the experiment.

Always ensure that the vehicles' batteries are loaded. If the battery percentage drops below 40%, [charge the vehicle](#) after your experiment.

Troubleshooting



- If the Main PC does not start, make sure the power bar to which the PC is connected is switched on.
- If you do not see a login screen this most likely because the login screen is shown either on the big screen on the wall or the screen on your left. You can either unplug the second screen or turn the corresponding screen on (worst case, turn on both).
- If you encounter any problems during the experiment, refer to the [diagnosis-tab](#). It can give a first idea where to look for errors.

Find errors at RTI DDS transmission

Use the RTI Tool (you find it on the main PC in applications):

1. Select "admin console"
2. Go to domain and View Preferences
3. Change the DDS Domain (on the bottom of the admin console) if necessary (the main PCs ID is 21, if you are not on the main PC you find it in `/etc/profile`)
4. Subscribe to the topic you want to listen to (keep the "advanced settings" unchanged) now you can choose the selected field if it is not already displayed from the dropdown menu.

Using NUCs for distributed calculation

The idea of the NUCs is to simulate a potent computer on a real vehicle for each vehicle. If you do not use the "NUC"-mode, all calculation will be done centralized on the main PC.

Turn on NUCs

Switch on the wallplug of the server rack by using the white remote control usually close to the main PC. The NUCs start automatically and each NUC will signal the LCC individually that it is online. You can find the online NUCs in the window on the bottom next to the button "reset view" at "HLC online". When you open the menu each NUC that is online is displayed. For each vehicle there is one corresponding NUC. The IDs have to match (e.g. vehicle 6 to NUC 6).

Nice to know



When you start the NUCs there is a program that is called "[autostart](#)". This program requests a password or, if you connect to the main PC, is connected via a ssh connection. The program will ask the main PC if there is a new software version for the NUCs and after a potential update it forces the NUCs to

give a feedback every second to the main PC in the way of " I am online (ID)", where it identifies its ID via its own [IP](#). if this alive-signal is not sent for a specified period of time, the LCC will mark this NUC as offline.

Nice to know



The NUC is supposed to simulate a potent server on a real life vehicle. This server would be usually connected via LAN. In the lab the vehicle and the NUC are separate and therefore connected via WLAN. They use the DDS structure to exchange data.

Use the NUCs

1. You can use the NUCs if you choose the "deploy remote" option. The script will be sent to the NUCs for calculation instead of being calculated on the main PC centrally.
2. Start your program choosing "Deploy" . The script will be loaded to the NUC and started.
3. Switch to the [timer-tab](#) in the LCC. When all NUC are ready ("WAITING") select "Start".
4. To finish your simulation hit either "Stop" in the timer-tab or "Kill" in the [setup-tab](#).

Fun Fact



You cannot use the "Central Routing" example for the distributed calculation, as its purpose is to be calculated centralized (that why it is called CENTRAL routing:)).

If you upload it on the NUC, each NUC would calculate the trajectory and collisions for only local IDs and would have no knowledge about the neighbours. So collisions would be quite possible.

Nice to know



The Matlab version on the NUCs can be selected from the installed ones using the startup script. See [Specifying the matlab version for HLCs on the NUCs](#).

Running an HLC

Prerequisite is that the [Experiment is set up](#).

Start HLC from the LCC

1. On the right, go to the [Setup](#) tab
2. In the script path, select the desired HLC.
3. Click *Deploy*.
4. On the right, go to the [Timer](#) tab
5. Wait for the timer with the ID *middleware* to show up.
6. Click *Start*.
7. Let your experiment run as desired.
8. Click *Stop* and *Reset* in the [Timer](#) tab.
9. On the right, go to the [Setup](#) tab
10. Click *Kill* in the [Setup](#) tab.

Start HLC externally

There are multiple reasons you might want to run an HLC from outside the LCC, e.g., for easier debugging.

Repeat the steps as above, but in step 3, you leave the HLC path blank and [run your HLC externally](#) (e.g. in MATLAB or in a terminal for a C++ script).

Read More

- [Indoor Positioning System](#)
- [Lab Control Center](#)
 - [Map View](#)
 - [Vehicle & NUC Monitoring](#)
 - [Setup \(Tab\) \(WIP\)](#)
 - [Manual Control \(Tab\)](#)
 - [Parameters \(Tab, Menu, XML\)](#)
 - [Timer \(Tab\)](#)
 - [Logs \(Tab\)](#)
 - [Commonroad \(Tab\)](#)
 - [Distributed Deployment \(NUCs\)](#)
 - [Integrated Debugging Tools](#)
- [Modules](#)
 - [CPM Library](#)
 - [Command Line Parser](#)
 - [Parameter Service](#)
 - [Timer - Periodicity and Synchronization](#)
 - [Visualization - Used in the LCC](#)
 - [Commonroad Information for the HLC](#)
 - [DDS Reader and Writer](#)
 - [DDS Participants and Topics](#)
 - [RoundTripTime](#)
 - [Logging](#)
 - [Logging Structure](#)
 - [Logging to DDS](#)
- [Monitoring \(eProsima\)](#)
- [Middleware Usage](#)
- [Matlab + RTI DDS](#)
- [Vehicle](#)
 - [Vehicle Overview](#)
 - [Equations](#)
 - [LiPo Charging](#)
- [Software structure](#)
 - [High Level Controller](#)
 - [Middleware Structure](#)
 - [Mid Level Controller](#)
 - [Low Level Controller](#)
- [HLC Examples](#)
 - [Basic Circle Example](#)
 - [Central Routing Example](#)
- [Computation Hardware](#)

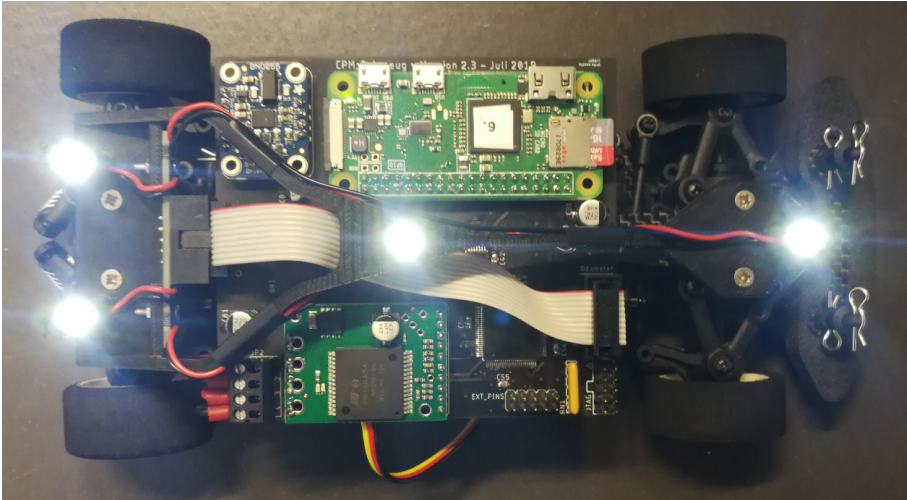
Indoor Positioning System

The Indoor Positioning System (IPS) detects the position and orientation (pose) and identity of multiple vehicles simultaneously. The poses are updated at 50 Hz.

Optics

The [Indoor Positioning System Setup](#) works with active light sources (LEDs) on the vehicles and a camera looking down from the ceiling. The LEDs are detected based on their high brightness. The camera is set to a very short exposure (~100 microseconds). Thus, the ambient light creates a very small signal (almost black), while the LEDs still appear as white dots. The short exposure time also eliminates the problem of motion blur. At a top speed of 4 m/s the vehicle travels 0.4 mm during the exposure.

Other light sources and reflective surfaces on the vehicle can create false signals and must be covered with tape. This includes the connectors on the Raspberry Pi and the status LEDs on the motor speed controller.



Vehicle Pose

The outer three LEDs indicate the vehicle pose and are permanently illuminated. The section [Pose Calibration](#) describes how the LED positions are related to the reference pose.

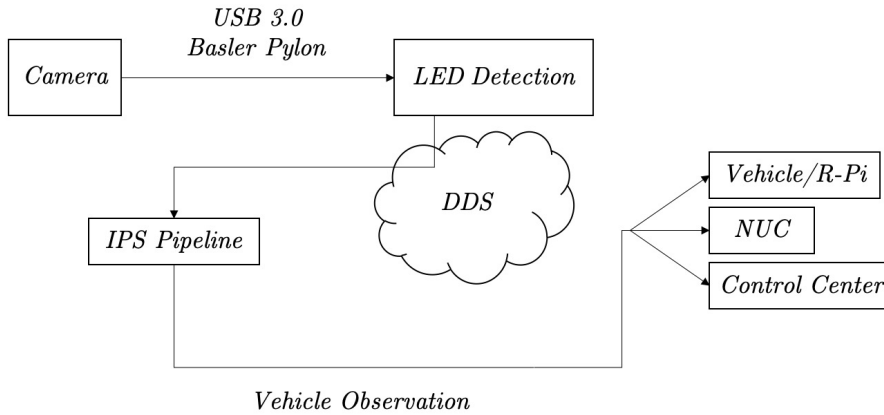
Vehicle Identification

The central LED flashes in a pattern that is different for each vehicle. The patterns are chosen such that sampling effects do not create ambiguous signals. See our paper [Vision-Based Real-Time Indoor Positioning System for Multiple Vehicles](#) for more information.

Vehicle ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
Period of flashing, in number of frames	4	7	10	13	16	7	10	13	16	19	10	13	16	19	22	13	16	19	22	25	16	19	22	25	28
LED illumination duration, in number of frames	2	2	2	2	2	5	5	5	5	5	8	8	8	8	8	11	11	11	11	11	14	14	14	14	14

Software Architecture

The IPS software has two major components, the LED detection and the IPS pipeline. The LED detection reads images from the camera at 50 Hz and extracts the list of image coordinates for all visible LEDs. These `LedPoints` are published via DDS. The raw images are not saved or published, as this would create impractically large data volumes. The IPS pipeline processes the LED points and publishes `VehicleObservations`, which include the vehicle's position, orientation, identity and a timestamp.

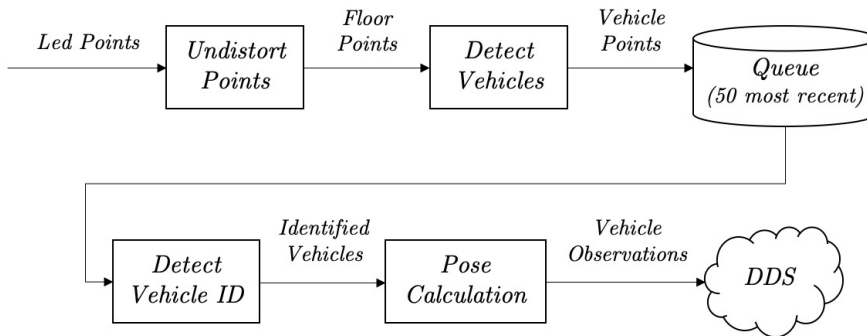


LED Detection

The LED detection uses the OpenCV functions `cv::threshold`, `cv::findContours` and `cv::moments` to find the centers of the LEDs.

IPS Pipeline

The IPS pipeline performs a relatively complex data processing task. To make the software easier to understand, it is broken down into independent, stateless processing steps.



The **UndistortPoints** function transforms the image/pixel coordinates to metric coordinates on the driving surface (floor). The calculation is based on a calibration image. See https://github.com/embedded-software-laboratory/cpm-lab/tree/master/matlab_scripts/camera_calibration

The **DetectVehicles** function groups and classifies points as front, center, back_left or back_right. Points that appear to not belong to a vehicle are discarded.

The **Queue** collects the 50 most recent samples of the vehicle points. This is necessary, as the vehicle ID can not be determined from a single image.

The **DetectVehicleID** function extracts the ID for each vehicle from the last second (50 frames) of vehicle points.

The **PoseCalculation** function calculates the vehicle reference pose based on the front, back_left and back_right points. The calculation is based on manually collected calibration data. See https://github.com/embedded-software-laboratory/cpm-lab/tree/master/matlab_scripts/ips_pose_calibration

The implementation can be found at <https://github.com/embedded-software-laboratory/cpm-lab/tree/master/ips2/src>

Calibration

Camera Calibration

First, other PVC canvases are removed from the floor to reveal the checkerboard canvas. Then an image of the checkerboard pattern is taken using the Py lonViewerApp. The exposure is adjusted such that the checkerboard corners are well resolved. If the exposure is too short or long, the corners are "rounded off", which degrades the quality of the calibration.

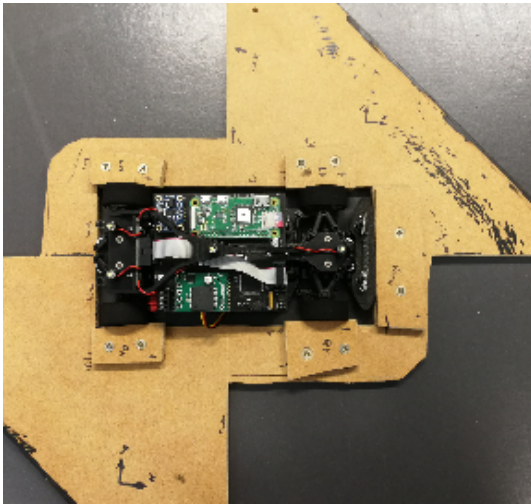
The Matlab function `detectCheckerboardPoints` is used to determine a list of matching points (`x_floor`, `y_floor`, `x_image`, `y_image`). Then a 5th order 2D polynomial is fitted to the data using linear least squares.

See https://github.com/embedded-software-laboratory/cpm-lab/tree/master/matlab_scripts/camera_calibration

Pose Calibration

The vehicle pose has the format (`x`, `y`, `yaw`). The yaw is the rotation angle in radians of the vehicle around the vertical axis in the counter-clockwise direction. When `yaw=0`, the vehicle points in the x-direction. The coordinates (`x`, `y`) give the location of the vehicle's *reference point*. The reference point is defined as the geometric center between the front and rear axle.

To accurately implement this definition of the pose, another measurement and calibration procedure is performed. First we need a means of accurately placing the vehicle on the floor, such that the true pose is known. The vehicle is clamped into a piece of wood, which extends the vehicle's local x/y coordinate system. This makes it simple to manually align the vehicle with the checkerboard pattern.



While the IPS is running, the vehicle is placed in various poses on the floor, following a particular [calibration sequence](#). The calibration features (`back_x`, `back_y`, `direction_x`, `direction_y`) are [recorded from the running IPS](#). A linear calibration is then fitted using least squares.

Coordinate Systems

IPS Image Coordinates

The image coordinates correspond to the IPS image sensor pixels. They are only relevant for the IPS LED detection.

Floor Coordinates

$(x \in [0, 4.5], y \in [0, 4])$

X/Y units: Meter

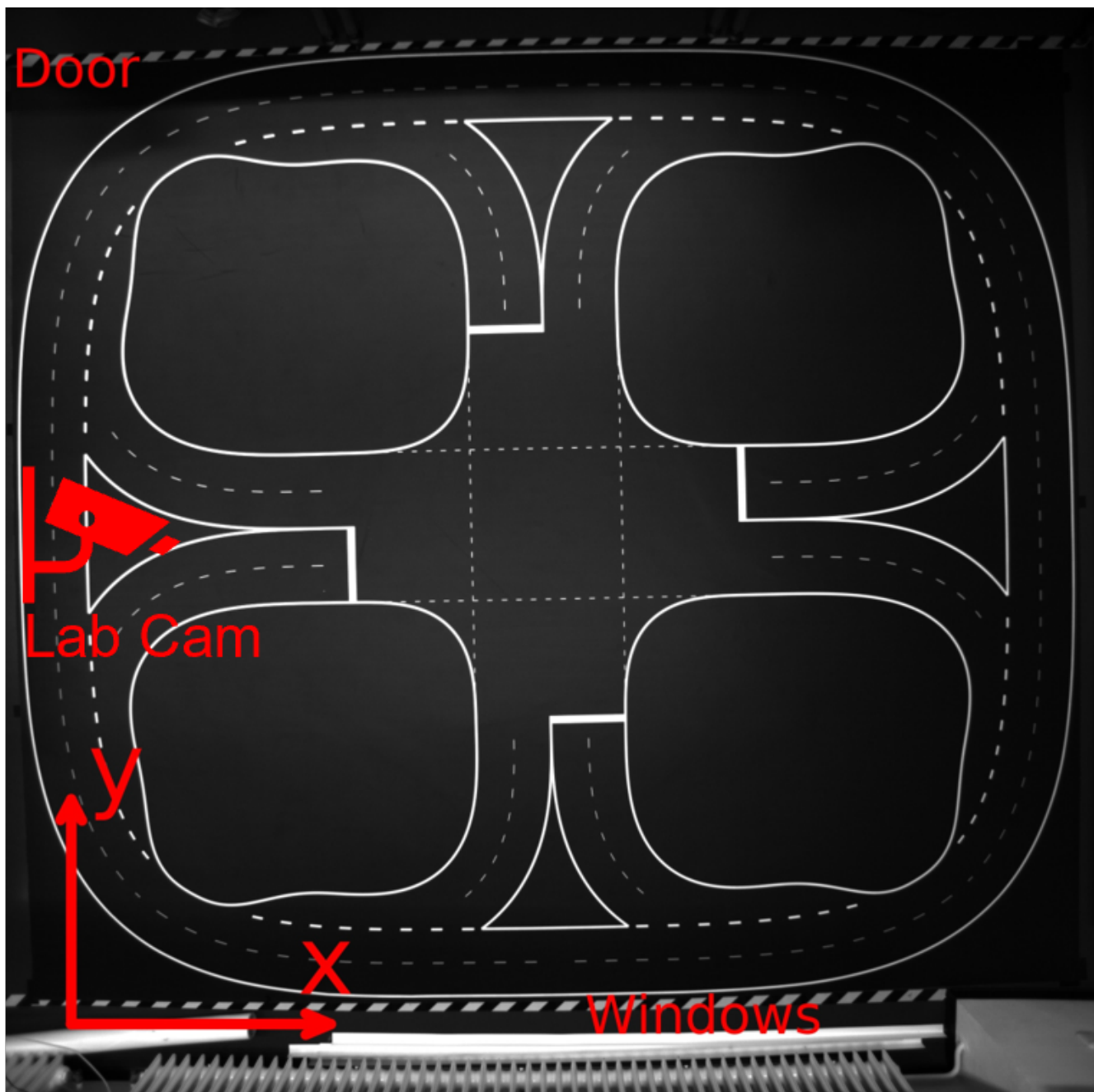
Yaw unit: Radian

Origin: corner of the PVC canvas, towards the windows on the right.

Yaw angle: Measured from the x-axis, counterclockwise.

Useful yaw equations:

```
direction_x = cos(yaw)
direction_y = sin(yaw)
yaw = atan2(direction_y, direction_x)
```



Vehicle Coordinates

X-axis: forwards

Y-axis: left

Origin: center point between both axles.

In practice, this coordinate system is [realized with a calibration jig](#).

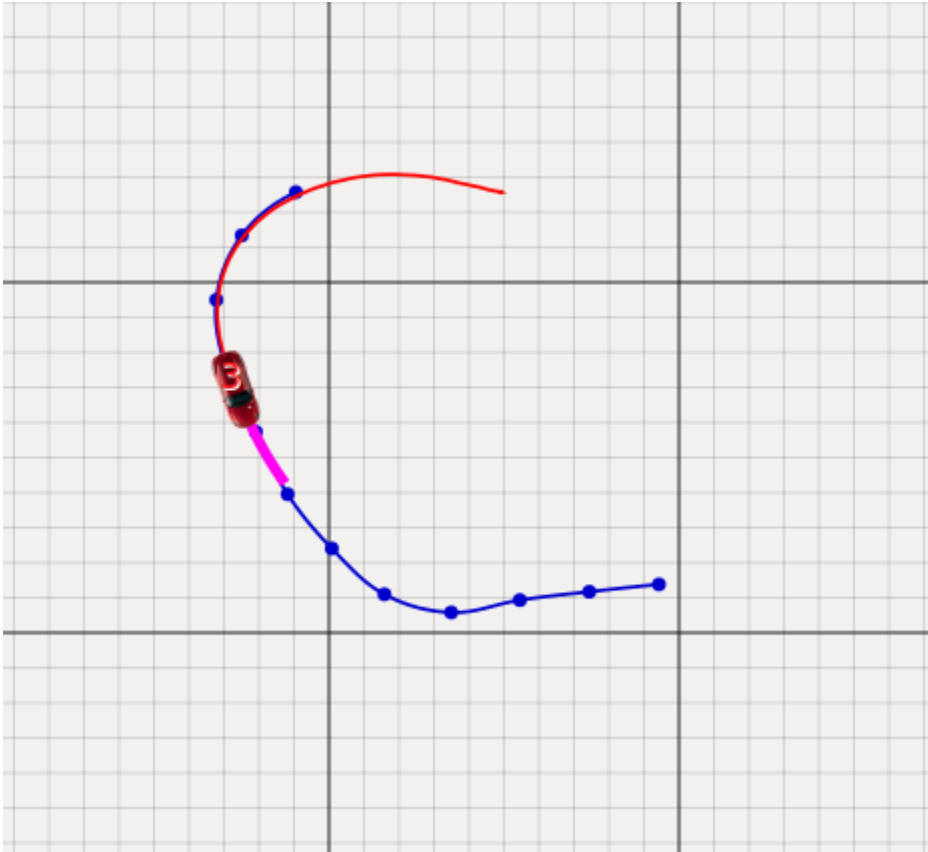
Lab Control Center

- [Map View](#)
- [Vehicle & NUC Monitoring](#)
- [Setup \(Tab\) \(WIP\)](#)
- [Manual Control \(Tab\)](#)
- [Parameters \(Tab, Menu, XML\)](#)
- [Timer \(Tab\)](#)
- [Logs \(Tab\)](#)
- [Commonroad \(Tab\)](#)
- [Distributed Deployment \(NUCs\)](#)
- [Integrated Debugging Tools](#)

Map View

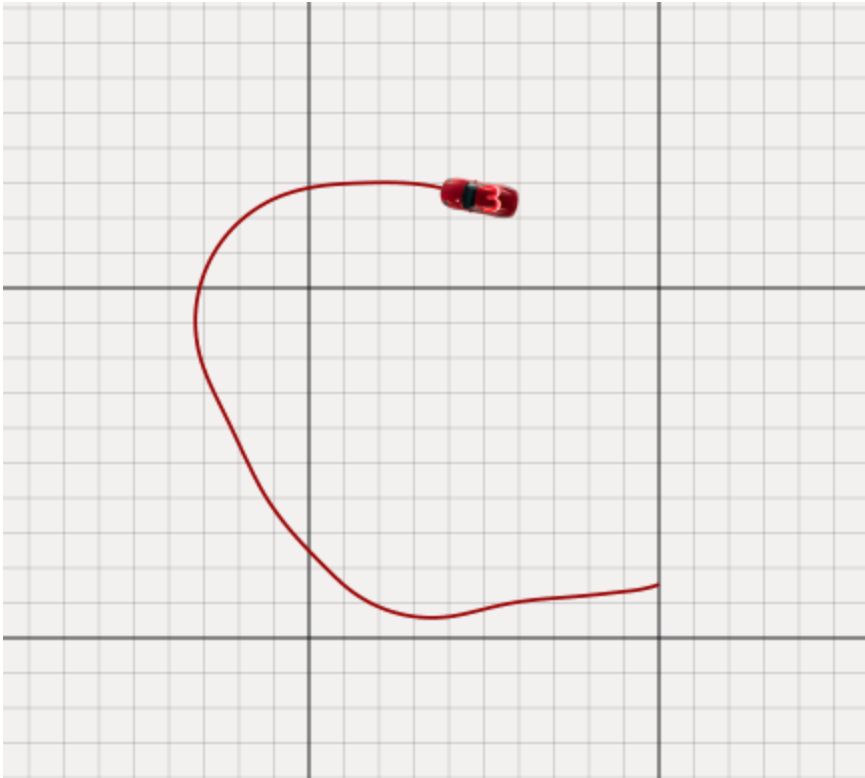
Vehicle

The map on the upper left view depicts the vehicle's positions and orientations, a "trail" of past positions (red), the commanded reference trajectory (blue), and the MPC's predicted trajectory (pink).



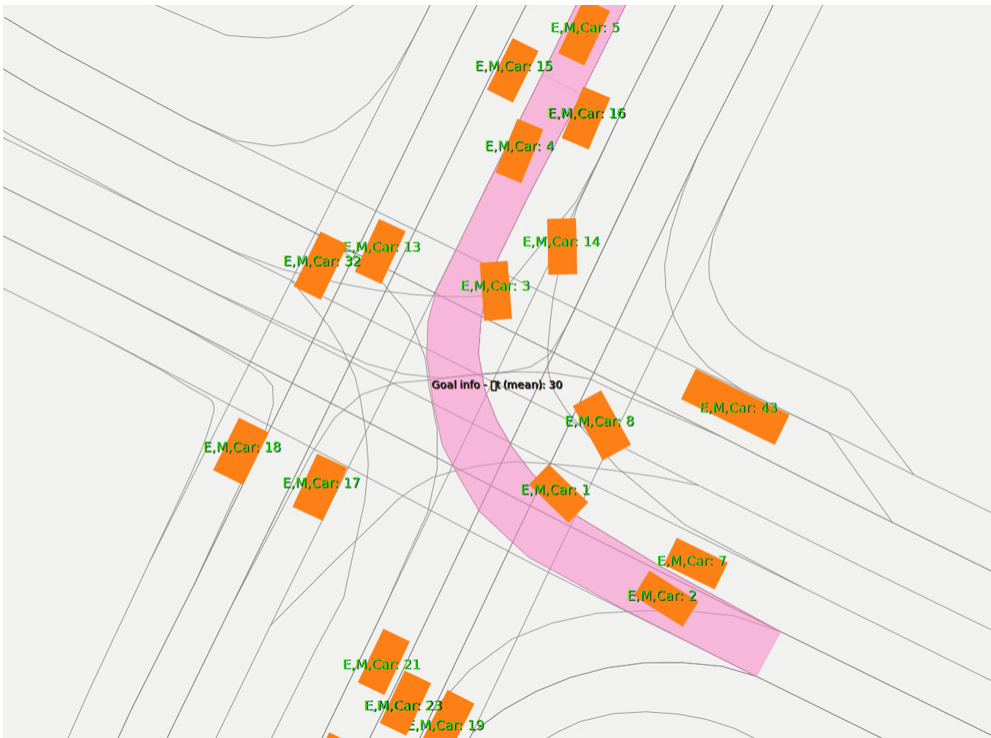
By sending [DDS visualization messages](#), elements can be dynamically added to the map view.

A vehicle can be controlled in a semi-automated fashion, by clicking and dragging to draw a reference trajectory. This only works when the vehicle is within the view of the IPS.



Commonroad Scenario

The map can also show data of a loaded commonroad scenario - lanelets, obstacles etc. Some of the information are also shown in the [Commonroad Tab](#), which can also be used to load a specific scenario.



You can see an example of a simulation in the Gif shown above. Vehicle 4 was chosen to be represented by a real vehicle, so upon starting the scenario, the vehicle is replaced by a trajectory that is sent within the network. You can find out more about this in [Commonroad Tab](#).

Vehicle & NUC Monitoring

On the bottom, you see variables related to the vehicles (vehicle state data and the IPS poses) that are exchanged via DDS. Values are color coded green, yellow or red according to expected values. This makes it possible to quickly check if all vehicles are operating normally. You can reset this view anytime using the *Reset View* button.

Reset view	Exp time: ---	▶ HLCs online: 0	▶ Reboot HLCs	▶ HLC RTT (ms): ---	▶ Vehicle RTT (ms): ---
	Vehicle 01	Vehicle 09	Vehicle 11		
Battery Level [%]	71	71	71		
IPS age [ms]	120	80	120		
VehicleState age [ms]	15.8	15.93	16.02		
Clock Delta [ms]	0.5	0.5	0.5		
Reference Deviation [m]	--	--	--		
Speed [m/s]	0.00	0.00	0.00		
Remote HLC [y/n]	no	no	no		

You also find information regarding the runtime of the currently running experiment, HLCs / NUCs that are currently online and RoundTripTime values (computed when no computation is performed) to the HLCs / NUCs and vehicles.

Round Trip Time

More information can be found [here](#).

HLC/NUC Monitoring

More information can be found [here](#).

VehicleState age

This is roughly the duration since the last VehicleState Message was received.

The when the LCC receives a new message it saves the current time `last_state_msg = t_now` and in the UI it displays `last_state_msg - t_now`. Thus the shown VehicleState age does not necessarily have to correspond with the actual age of the information that the LCC has about the vehicle this would be `vehicleState.header().create_stamp() - t_now`.

Setup (Tab) (WIP)

Setup

Commonroad

Manual Control

Parameters

Timer

Logs

LCC Errors

Lab mode

Diagnosis

Script Path

/home/leon/dev/software/high_level_controller/examples/cpp/central_routing/build/central_ro

Open

Script Command Line Parameters

Simulated Vehicles

Vehicle 1

Reboot

Vehicle 2

Simulate

Vehicle 3

Turn Off

Vehicle 4

Simulate

Vehicle 5

Simulate

Vehicle 6

Simulate

Vehicle 7

Simulate

Vehicle 8

Simulate

Vehicle 9

Simulate

Vehicle 10

Simulate

Vehicle 11

Simulate

Vehicle 12

Simulate

Vehicle 13

Simulate

Vehicle 14

Simulate

Vehicle 15

Simulate

Vehicle 16

Simulate

Vehicle 17

Simulate

Vehicle 18

Simulate

Vehicle 19

Simulate

Vehicle 20

Simulate

Select none

All simulated

Simulated Time

Deploy Remote

Lab Camera

Record

Camera Status

Not Initialized

Kill

Deploy

The setup view allows to launch all components required to perform a simulation from within the LCC's GUI. It can also be used for real vehicles, and to deploy your selected program or script remotely on the NUCs (HLCs).

Deploy Remote is now called **Deploy Distributed**

Selecting a Script

A single script can be selected by editing the path text field or selecting a file using the *Open* button. In the file chooser, you can select whether you want to use an executable or a Matlab script in a drop down menu at the bottom right of the window.

The file chooser remembers your previously selected file.

If no script is selected (empty string as script path), the LCC will only start the Middleware. You can run your script yourself outside of the LCC, connect to the Middleware and then start the simulation.

Required Script Properties

Your script should take the following arguments:

C++	Matlab
--vehicle_ids=	RTI DDS: (your_params, varargin)
--dds_domain=	eProsima: (your_params, middleware_domain_id, varargin)
--middleware_domain= (nur im eProsima Branch)	
--node_id=	

```
--simulated_time=  
--dds_initial_peer=
```

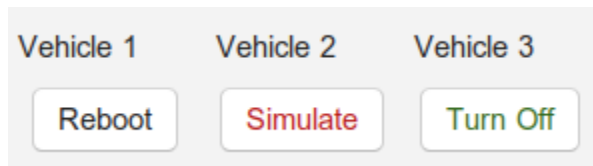
Most of the required C++ parameters are used to set values for the cpm library. Of these parameters, you are probably only interested in the list of vehicle IDs your program is responsible for (--vehicle-ids). The same is true for the matlab script, where varargin gives you a list of vehicle IDs - any other parameters, which come beforehand, are custom.

The Middleware domain ID can currently only be set consistently in the **eProsima** Branch, the functionality might not be added to support **RTI** w.r.t. Matlab, although the change would be rather small (you can adapt your Matlab scripts accordingly), as the Lab will change to using eProsima only in the future.

Further parameters can be specified in the text field below the *Open* button. If you need to access these command line parameters in your C++ program, simply use the [Command Line Parser](#) and --your_param= in the text field. Custom arguments can be set by yourself this way during testing / for your simulations. Matlab custom arguments just need to be passed in a comma-separated fashion (e.g.: 1, 155, 3).

Selecting Vehicles

You can select, for each vehicle individually, whether they should be simulated, or control their current state if they are real vehicles.



Simulated Vehicle	• <i>Simulate</i> The vehicle is currently not simulated, but can be started with a button press • <i>Turn Off</i> The vehicle is currently simulated and can be turned off by pushing the button
Real Vehicle	• <i>Reboot</i> The vehicle was detected as real vehicle. By using this button, you can send a reboot command to it to reboot the Raspberry Pi.

The buttons below the scrollable view allow you to set the state for all (non-real) vehicles at once.

Vehicles that are set to be off will be ignored by your script, as their IDs are not passed to it. You do not need to start the simulated vehicles - the LCC does that for you immediately after selection. They are killed when the LCC is closed or when another mode (real or off) is chosen.

Warning

 If you have not started the LCC with the *correct parameters* (e.g. use the [main server DDS domain](#)), you might not be able to connect to / see the real vehicles.

Simulated Time

This switch should be self-explanatory - if you want to use [simulated time](#), turn it on, else turn it off.

Lab mode

The [IPS](#) tracks the real vehicles, so it should be turned on if real vehicles are used. In any other case, it should stay turned off.

Deploy Distributed

This option includes a lot of features, thus we created an [own page for it](#). If this switch is set, your program gets uploaded to the NUCs / HLCs for deployment, instead of being deployed locally on your machine. One script instance handles one vehicle (ID) per NUC / HLC.

Lab Camera

Turn the "Record" switch on before clicking "Deploy" to record your experiment.

After stopping the experiment with a click on "Kill", you will find the video file in /tmp/cpm_lab_recordings .

If you are planning to play the video on windows without installing codecs, convert the video e.g. with

```
ffmpeg -i $INPUT_NAME_WITH_FILE_EXTENSION$ -ss $START_IN_SECONDS$ -t $DURATION_IN_SECONDS$ -c:v libx264 -strict -2 -preset slow -pix_fmt yuv420p -f mp4 output.mp4
```

Deploy / Kill

When you have finished your setup, you can launch all the components / scripts by clicking on *Deploy*. Not all software starts immediately then. Some, e.g. those that rely on the Middleware, wait until you start the simulation in the [Timer Tab](#). (Wait for the program to show up there before you start the timer). Due to initialization and synchronization, it makes more sense to wait for all participants before starting the timer.

Local: This starts several tmux sessions that run the required scripts. These sessions are terminated when you press on *Kill*. A stop signal is sent to all timers when the kill button gets pressed.

Remote: The folder in which the script is located gets uploaded to the NUCs if you choose to deploy the software remotely. Each HLC/NUC script gets assigned a single vehicle ID. IDs are given in ascending order regarding the selected vehicle IDs and the available HLC IDs. Not all vehicles might be controlled, if not enough NUCs are online. Middleware and Script are deployed on the HLCs, simulated vehicles, IPS and CloudDiscoveryService are deployed on the local machine.

Important



To execute your software, you often need to Deploy Switch to Timer Tab Wait for the Middleware to show up Start the Timer

Important



Always use Kill after you have finished, or else some software might end up running on a system (like the NUC / HLC) until it is restarted.

Manual Control (Tab)

Each vehicle can be controlled manually with a joystick or gamepad. First, make sure your gamepad works properly using `jstest-gtk`. Enter the ID of the desired vehicle and click start.

The two progress bars in the UI indicate the commanded throttle and steering position.

The manual control overrides the reference trajectory based control. Thus it is important to click "stop" before sending reference trajectories.

Manual Control

Parameters

Timer

Joystick Device

/dev/input/js0

Vehicle ID

3

(re)start

stop

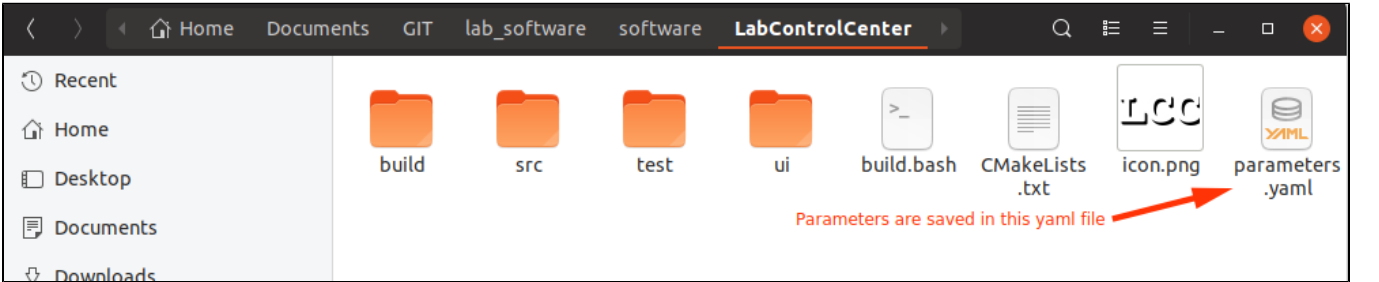
Throttle

Steering

Parameters (Tab, Menu, XML)

Defining Initial Parameters for the Participants

Initial parameters can be defined using the parameters.yaml - file. For most parameters, there is yet no convention regarding their name and types, so please contact someone responsible for the LCC if this part of the documentation has not been updated yet.



The parameters are stored in the YAML format in the file parameters.yaml in the directory of the LCC. Some parameters are distributed on program startup, these must be either edited in the LCC before restarting it, or they can be edited manually by changing the content of the YAML file:

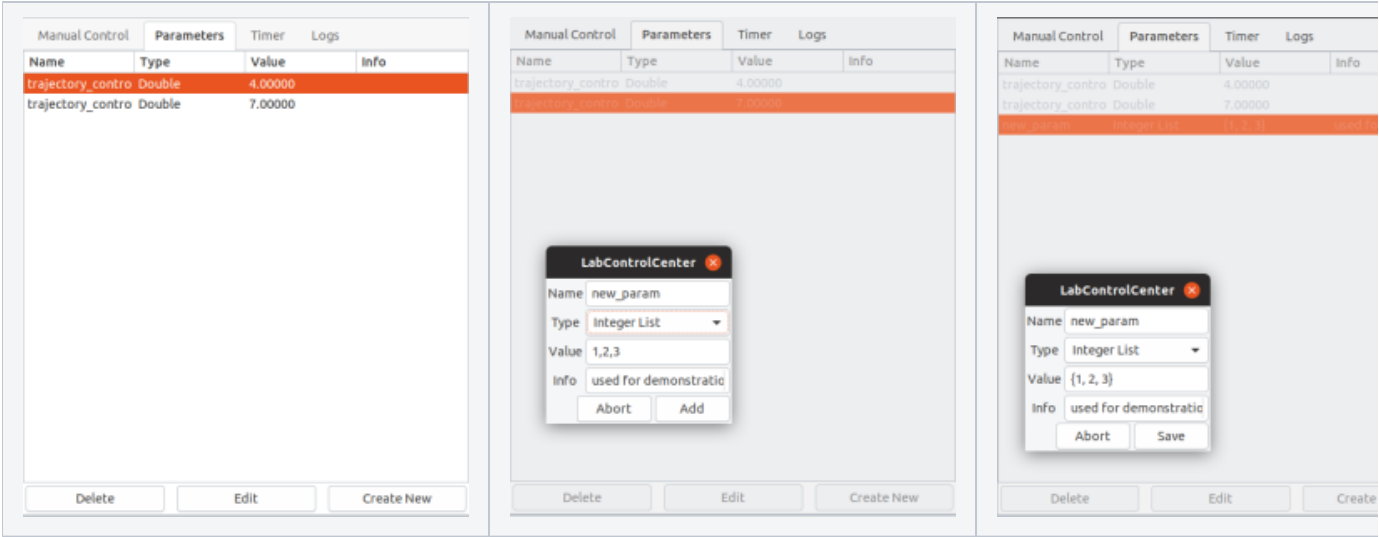


Six different parameter types can be defined - bool, integer, double and string as well as comma-separated lists of integers and doubles. The parameters consist of a name (e.g. trajectory_controller/lateral_D_gain), a value (or set of values) and an information string that can be used to describe the purpose of the parameter.

Precision of Doubles

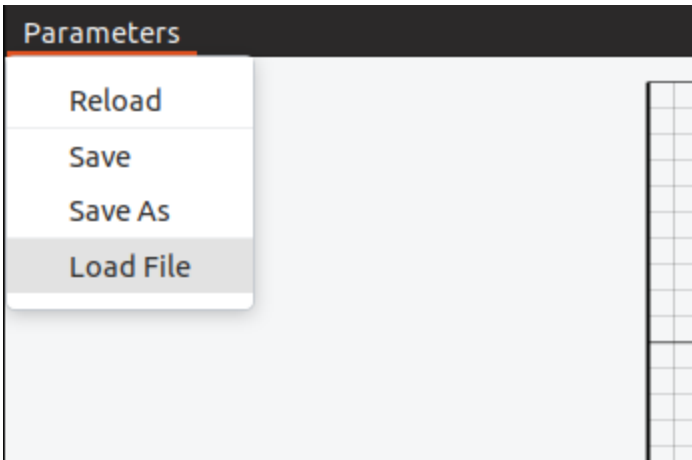
Doubles are currently stored and loaded with a precision of up to 3 decimal places.

Setting Parameters in the GUI



Parameters are shown in the LCC UI in the parameters tab. This UI simplifies the editing process of the available parameters - they can be deleted or changed, and new parameters can be created as well using the UI tools. Thus, if desired, the YAML file does not need to be used explicitly.

The LCC can use different YAML files as well. A file chooser has been implemented to load and save parameter configurations (to / from YAML files).



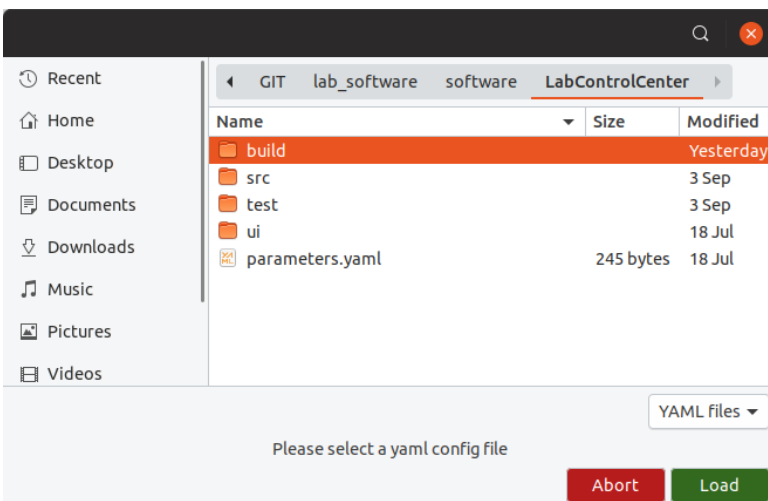
Save can be used to store the current configuration in the current file.

Save As can be used to store the current configuration in a new file.

Reload can be used to reload the currently selected file - this overwrites any unsaved changes made in between.

```
parameters:
  bool:
    {}
  int:
    {}
  double:
    trajectory_controller/lateral_D_gain:
      value: 4
      info: ""
    trajectory_controller/lateral_P_gain:
      value: 7
      info: ""
  string:
    {}
  ints:
    {}
  doubles:
    {}
```

Load can be used to load a new file.



Timer (Tab)

General Timer Functionalities

The LCC can be used to start / stop the execution of all applications that use the cpm timer (see [synchronization](#)) centrally. This functionality is available both in the real-time and in the simulated time case.

Manual ControlParametersTimerLogs

Currently active timers

ID	Last message	Participant status	Next timestep
controller_test_loop	1s	READY	1000000000
vehicle_raspberry_1	1s	READY	0

Timing information of active participants

Start the execution on all systems

Stop the execution on all systems

Current time step:0

Start

Stop

Restart system

The timer UI shows all participants that use the timer and whose ReadySignal messages have been received by the LCC. The participant's ID, the time passed since its last message was received, its current status and (in simulated time) its next active time step are shown in the UI.

Participant Status

READY	The participant sent its first ReadySignal and is now ready to receive a start signal
(realtime)	(real time) After pressing start, there are no additional information available about the participants
WAITING	(simulated time) The participant is waiting for the start of its next registered period
WORKING	(simulated time) The participant received a SystemTrigger that matches its next registered period and has not yet sent a ReadySignal that indicates that its work has been finished
OUT OF SYNC	(simulated time) The participant's next period request is lower than the current simulated time / invalid

Simulated Time

In the simulated time case, the columns Participant status and Next timestep can be a useful tool for debugging and monitoring purposes. The latter indicates at which simulated point in time the participant is woken up again to perform its periodic task.

Current time step below the table indicates the current simulated time.

The timing itself is managed by the LCC itself internally. It manages the received ReadySignals of all participants and progresses to the next smallest timestep whenever all participants of the current timestep have signaled that they have finished their current computation. For more information, have a look at the [page about synchronization](#).

Logs (Tab)

The cpm library provides a [logging functionality](#) that includes writing to stdout, to a log file and to a RTI DDS DataWriter. The logs sent via the latter over the network are listened to by the LCC and shown in the Logs UI. (The LCC only show the newest ~10000 logs to keep the UI responsive).

Log structure

The log messages include a unique identifier of the sender (*ID*), the content of the log message (*Content*) and the point in time when the log message was created (*At time*). They appear in a table in the middle of the logs view.

Manual ControlParametersTimerLogs

All

Log messages

ID	Content	At time
vehicle_raspberry_8	Vehicle Controller Tracking Errors:lo	1568115038225717501
vehicle_raspberry_4	Vehicle Controller Tracking Errors:lo	1568115039425670524
vehicle_raspberry_6	Vehicle Controller Tracking Errors:lo	1568115040744530755
vehicle_raspberry_5	Vehicle Controller Tracking Errors:lo	1568115041949255556
vehicle_raspberry_7	Vehicle Controller Tracking Errors:lo	1568115043467010015
vehicle_raspberry_3	Vehicle Controller Tracking Errors:lo	1568115054048176379
vehicle_raspberry_1	Vehicle Controller Tracking Errors:lo	1568115055236135467
vehicle_raspberry_2	Vehicle Controller Tracking Errors:lo	1568115056663164493
vehicle_raspberry_8	Vehicle Controller Tracking Errors:lo	1568115057898117590
vehicle_raspberry_4	Vehicle Controller Tracking Errors:lo	1568115059257341420
vehicle_raspberry_6	Vehicle Controller Tracking Errors:lo	1568115060466753072
vehicle_raspberry_5	Vehicle Controller Tracking Errors:lo	1568115061858257801
vehicle_raspberry_7	Vehicle Controller Tracking Errors:lo	1568115063111390697

☐ Autoscroll

The *Autoscroll* check button can be used to toggle automatic scrolling to the newest log entry on or off. The newest log entry is always at the bottom of the list. Entries are sorted w.r.t. their timestamp.

If the content of a cell is not fully visible, the user can change the width of each column using the mouse (by dragging the vertical lines between each of the cells at the head of the table). If that is not sufficient, the user can also hover over the region of interest - the full text then appears below the pointer.

Filtering Logs

Manual ControlParametersTimer**Logs**

All

Q

Log messages

filter column

filter string

ID	Content	At time
log messages appear in this scrollable list		

☒ Autoscroll

automatically scroll down to the latest log message

Manual ControlParametersTimer**Logs**

Timestamp

Q

Log messages

ID	Content
vehicle_raspberry_8	Vehicle Controller Trac
vehicle_raspberry_4	Vehicle Controller Trac
vehicle_raspberry_6	Vehicle Controller Trac
vehicle_raspberry_5	Vehicle Controller Trac
vehicle_raspberry_7	Vehicle Controller Trac
vehicle_raspberry_3	Vehicle Controller Trac
vehicle_raspberry_1	Vehicle Controller Trac
vehicle_raspberry_2	Vehicle Controller Trac
vehicle_raspberry_4	Vehicle Controller Tracking Errors:long,n long,max: 0.033614 lat,mean: 0.004125
vehicle_raspberry_6	Vehicle Controller Trac
vehicle_raspberry_5	Vehicle Controller Trac
vehicle_raspberry_7	Vehicle Controller Trac

☐ Autoscroll

The logs can be filtered using a regex search. The search can either be performed over all columns or only in one of them - this can be selected using the *filter column* combo box. The regex search string can be put in the *filter string* search entry. The table below either shows the result of the applied filter, or all logs if no filter is applied (if the search entry is empty).

Warning: A filter is only applied to all messages that have been received up to that point in time. New logs are received in the background, but are not added to the result of the search unless the search string is changed (which starts a new search).

Manual ControlParametersTimer**Logs**

Timestamp

Q 156811506

Log messages

ID	Content	At time
vehicle_raspberry_6	Vehicle Controller Tracking Errors:lo	1568115060466753072
vehicle_raspberry_5	Vehicle Controller Tracking Errors:lo	1568115061858257801
vehicle_raspberry_7	Vehicle Controller Tracking Errors:lo	1568115063111390697

Setting a Log Level

☐ Autoscroll

Current log level:

1

In the Logs tab, you can also set a log level for the domain of the cpm library DomainParticipant (the *dds_domain* you started the LCC with).

This log level is used to set the *verbosity of the participants* when logging information.

Log level	Description
0	Do not log anything (e.g. for performance reasons)
1 (default)	Only log critical failures
2	Also log less relevant error messages
3	Also log any other message of lesser relevance

For each logged message, the log level can be set programatically (in C++, using the cpm library). The message is only logged if its own log level is higher than or equal to the system-wide log level.

Commonroad (Tab)

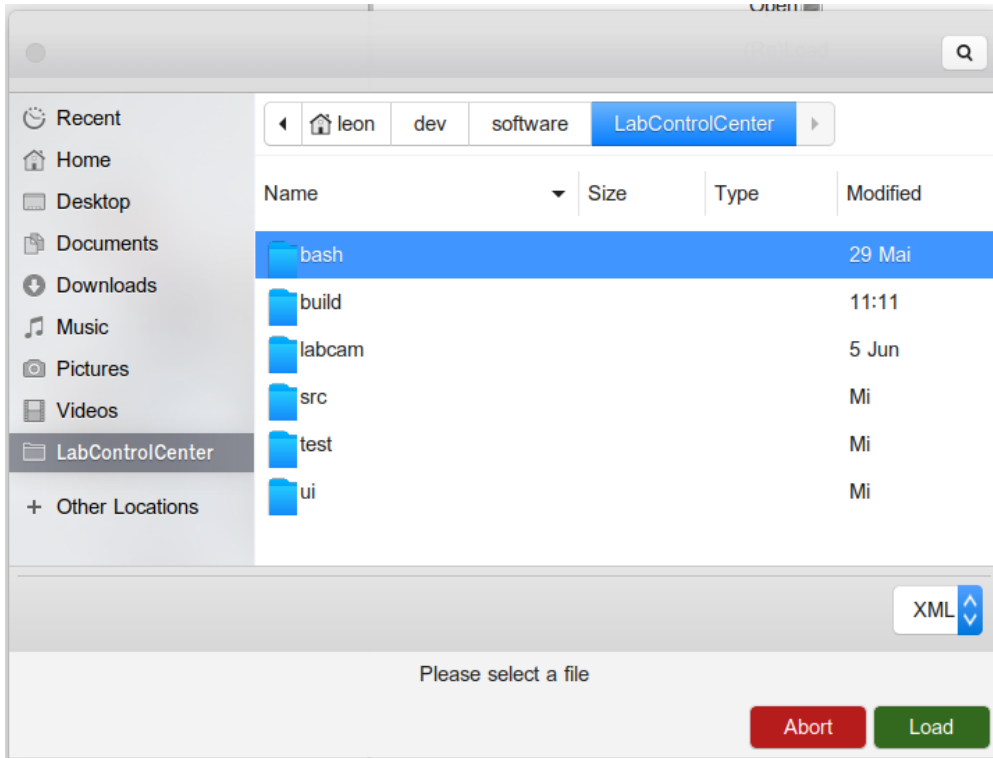
Elements in this tab allow the user to load and transform Commonroad scenarios which follow the 2018 or 2020 specifications. The scenario will then be drawn in the MapView of the LCC and obstacles will either be simulated or used to control real vehicles with VehicleCommandTrajectory messages. **(TODO** :) Planning problems, obstacles etc. defined in the scenario will be used for the simulation and will be communicated to the participants when the simulation is started in [Setup](#).



The tab also shows some basic information about the set planning problems, allows to transform the problem (and to store that transformation) and gives options on what elements of the problem to show.

Choosing a Scenario

A scenario can be chosen similarly to the script selection in [Setup](#). Either use the *Open* button or edit the path text field and hit *Enter*.



Commonroad Specification

Some unsupported features and minor problems during parsing lead to warnings visible in the [LCC Errors Tab](#). More severe warnings will lead to an error message (see below).

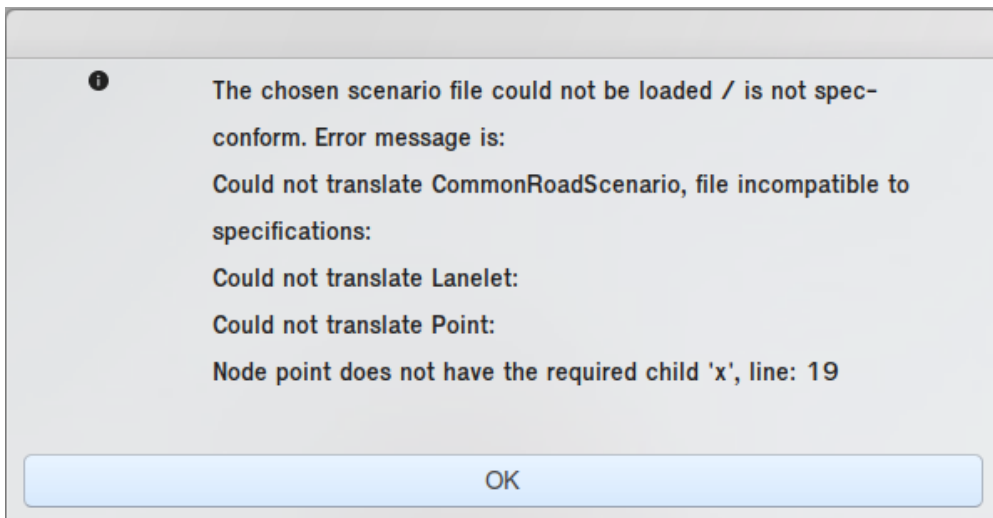
Traffic lights are currently not being simulated, but their position is shown, if possible.

We also do not (yet) have a consistent handling of inexact positional values (we currently use the average) and rotations for shapes. The assumed default value for positions is (0,0) (origin).

Your chosen scenario gets **centered** when it is loaded, w.r.t the middle of the Lab's IPS boundaries. It is not automatically resized, though, so you might not see the whole scenario when you load it.

Error Messages

Upon loading a scenario, you might receive an error message. Parts of the implemented loader check your scenario for consistency with the Commonroad specification. If it is malformed, values are missing or wrong types are used, you will find that the scenario will not load. Instead, a window will appear, which refers you to the line (in your scenario file) that raised the error.



IMPORTANT: There are also less relevant errors, e.g. related to drawing, which might be caused by currently missing implementation or decisions made on which Commonroad types to support. These error messages are shown in the [LCC Errors Tab](#) and will not prevent your scenario from loading. Keep in mind that these parts of the scenario might then not be supported in the simulation and also might not show up on the screen.

Transforming the Scenario

You can also transform your scenario:

Time step size

0.1

Transform Settings

Min. lane width

0.0

Translate x

0.0

Translate y

0.0

Rotate (origin)

0.0

Apply transformation

- *Resize* it by setting a minimum lane width
- *Translate* its x or y coordinates
- *Rotate* it around its center
- *Change* the *time steps*

Just enter the desired values in the according text boxes and either hit *Enter* if you just want to change one value, or set values for multiple fields first before applying them by clicking on *Apply transformation*. The values in the fields are set to *0.0* again after the transformation was applied.

When you **change the time steps**, you change the time between each time step set in commonroad. Larger timesteps mean a slower simulation and vice-versa. Speed and acceleration values are adjusted accordingly. (Example: Changing from 0.1seconds to 0.2seconds means half the speed)

IMPORTANT: The scenario gets centered when it is first loaded, as the set coordinates might be out of scope. This means, if your scenario only uses coordinates e.g. larger than 200 for x and y values, it should still show up in the MapView. The current centering is not perfect, so you may want to translate it further.

Transformation Profile

You might be interested in keeping the transformation that you set for your scenario. We use a YAML profile to accomplish this. You do not need to access it directly:

(Re)Load

Load saved transformation

Save transformation

Reset saved transformation

Time step size

0.1

- *Load saved transformation* This button applies transformations that you have stored before to the currently loaded scenario. **Important:** It does not reset changes made before, so you might want to *(Re)Load* before pressing this button.
- *Save transformation* Save the current transformation in the profile for your current file. **Important:** Changes in x / y are added, not overwritten.
- *Reset saved transformation* Resets the transformation profile for the current file. **Important:** This also includes the initial centering that takes place automatically, and every transformation you have done so far. It is thus advisable to *(Re)Load* after performing a reset.

Rotation is not being stored. It is the only operation where the order truly matters, and it is also non-linear. It was thus left out for now.

Scenario Preview

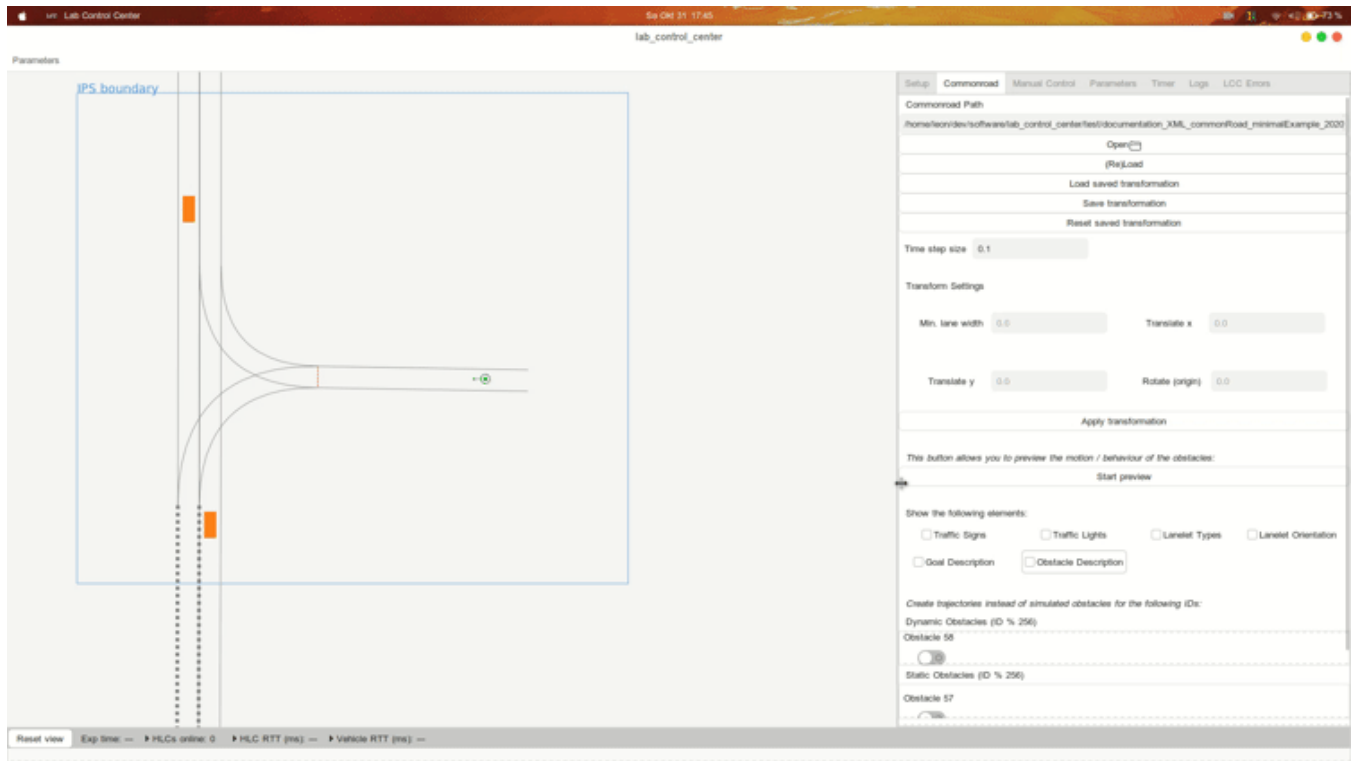
If you want to see what the scenario looks like during the simulation, you can preview the movement of the obstacles from within the Commonroad Tab:

This button allows you to preview the motion / behaviour of the obstacles:

Start preview

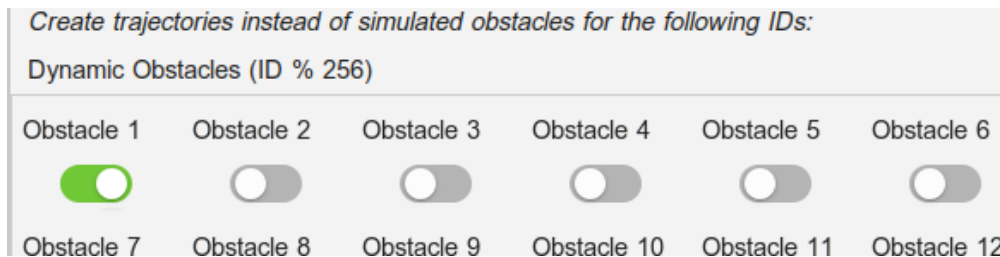
Toggle Additional Information

We did not want the view to become too cluttered. Thus, parts of the simulation like traffic signs or vehicle information are only drawn on demand. They can be toggled in this part. The explanation of the toggled descriptions can be found on the bottom of this page. Please note that the view below might be outdated (more toggles might exist), but it still works the same way.



Obstacles and IDs

The loaded Commonroad scenario may define obstacles. The **obstacle ID** set in Commonroad is interpreted as the **ID of the vehicle** in the scenario. This is especially important if you choose to use the movement defined for the obstacle to be used as trajectory for a real vehicle. In this case, the real vehicle with the same ID will respond to the message.



In the image above, the path of obstacle 1 is communicated in form of trajectory messages for vehicle 1 - which can also be received and followed by a real vehicle - while the other obstacles are simulated without controlling real vehicles.

We only allow for obstacle IDs up to 256 due to the message types in use (uint8_t for ID) (**TODO?**), thus the ID is taken mod 256 if it is higher than that - we do not expect that you need to use more than 256 obstacles within one scenario (this might also have bad consequences for the performance of a simulation).

Obstacle Classes

Commonroad defines DynamicObstacle, StaticObstacle and EnvironmentObstacle as different obstacle classes. We only allow for trajectory simulation for the first two, as environment obstacles should not be on the lanelet (and don't move as well). Static obstacles may include static vehicles, so we decided to allow trajectory simulation for them, although they also never move (as their name already implies).

Goal Information

Additional information about the goal states are shown at the bottom of the Commonroad-Tab. Please note that goals are only drawn in the MapView if a position for the goal was set.

Planning Problems		
Problem ID	Goal Speed	Goal Time (sec.)
15	[5, 15]	[2, 3.1]

Lanelet Information

Additional information about the lanelets are shown below the goal information. The lanelet IDs can be toggled to know which lanelet the information belongs to.

Lanelets				
Lanelet ID	Lanelet Type	User OneWay	User Bidirectional	Speed Limit (2018 spec)
6610	Urban			
6711	Urban, BusLane, Shoulder		Vehicle, Taxi	
6712	Urban		Vehicle	

Most information that are not visible in this table are visible implicitly or explicitly on the MapView (and / or can be toggled as additional information).

Using Commonroad as Part of the Simulation

To actually see / use the obstacle simulation, you need to:

1. Click *Deploy* in the Setup Tab
2. Switch to the Timer Tab and press *Start*

This should be your usual workflow for starting your own script via the LCC as well.

Obstacles disappear



As soon as you press *Deploy*, the obstacles disappear. This is done on purpose! The obstacle simulation, as all other simulation, is performed as soon as the lab simulation is started in the Timer Tab. As soon as the timer is started, the obstacles will reappear.

Further information

Descriptions for obstacles, lanelets etc. use few letters to save space. Their meaning is explained here.

Descriptions for Obstacles

Symbol	Meaning
E	Pose is exact
I	Pose is not exact
M	Obstacle is moving
S	Obstacle is static
Unk	Unknown obstacle
Car	Obstacle is a car
Truck	Obstacle is a truck
Bus	Obstacle is a bus
MCycle	Obstacle is a motorcycle
BCycle	Obstacle is a bicycle
Ped	Obstacle is a pedestrian
Prio	Obstacle is a priority vehicle
Train	Obstacle is a train
Taxi	Obstacle is a taxi
Constr	Obstacle is a construction zone

Parked	Obstacle is a parked vehicle
Boundary	Obstacle is a road boundary
Building	Obstacle is a building (not on the road)
Pillar	Obstacle is a pillar (should not be on the road)
Median	Obstacle is a median strip in between lanes

Descriptions for Traffic Signs

The text drawn for **traffic signs** is taken from its definition in the Commonroad file:

Traffic Sign ID (list of additional values) Traffic Sign ID (list of additional values) Traffic Sign ID (list of additional values) ...

Descriptions for Goal States

ID (ID of the planning problem) - **time** when the goal should be reached (**mean**), **v** speed within the goal (**mean**)

Color Code for Obstacles

Color	Meaning
Filled Orange	Dynamic obstacle with exact position
Filled Violet	Dynamic obstacle with inexact position
Orange Border, not filled	Non-Dynamic obstacle (static or environment) with exact position
Violet Border, not filled	Non-Dynamic obstacle (static or environment) with inexact position

Lanelet Orientation Arrows

Arrowhead	Meaning
Arrow / Triangle	Gives the lanelet orientation. No user is allowed to use the lanelet in the other direction.
Circle / Point	At least one user is allowed to use the lanelet in both directions. The Lanelet information table, given the ID of the Lanelet (if shown, which can be toggled), shows which user is allowed to use the lanelet in both directions (e.g. a bicycle).

Lanelet Border Colors

Color	Meaning
Blue	Lanelets are adjacent and have the same driving direction Crossing this line should be allowed
Black	All other lanelets
Light grey	Lanelets for which no_marking was defined. Is overwritten by blue in case of adjacency to another lanelet.

Lanelet Border Style



The Lanelet Border Style can be defined in the Commonroad file (e.g. dashed or broad). Unknown and no_marking are not easy to draw. Thus, unknown is drawn like any solid line, and no_marking is drawn in a transparent grey color. Both colors can be overwritten by blue in case of adjacency (this information is more important).

Distributed Deployment (NUCs)

If you select this option in the [Setup Tab of the LCC](#) (Deploy Distributed), the behavior of the whole deployment changes fundamentally. Your software is no longer deployed once locally on your device. If no NUCs are online or set up, **one instance of your selected program is started for each vehicle that is online** at that time, so make sure that your software can deal with getting one vehicle ID each and that the started programs can communicate with each other (if necessary). The same also holds if NUCs are set up - your software is started on the [NUCs](#) (HLCs). (Your system must have installed the [full setup](#)). Each NUC takes care of one vehicle only. If more vehicles than NUCs are online, the remaining programs will run locally on your PC.

Requirements

Setting up the NUCs correctly

The [NUCs need to be set up](#). They also need an automatically starting task running, which looks for new versions of the required files. It is called [crontab task](#) and relies on `lab_autostart.bash`.

Info on `lab_autostart.bash`



This file can also be found in the software repository (`./high_level_controller/autostart`). It is the only file that currently is **not automatically updated** on the NUC. Thus, if you change it, you need to re-upload it on the devices for the guest user. The path is described [here](#).

The script is responsible for setting up the environment on the NUCs (e.g. downloading current versions of the cpm library, middleware etc.) so that programs can be deployed on the machines similarly to local deployment.

Provide all required packages

Before turning the NUCs on

1. Start the main PC (IP: 192.168.1.249)
2. Build the lab software (or make sure your desired version was the most recent build)

The *middleware*, *cpm library* and *autostart* software must be built - the build script puts data required for the software to work on the NUCs in folders that allow the NUCs to download it from the Apache Server (`/var/www/html/nuc`).

Within the folder for the autostart software (located in `~/software/high_level_controller/autostart`), you also find the script `create_nuc_package.bash`. Call this script as well do create the final part of the NUC package (for matlab scripts).

Check again if all four scripts can be found within `var/www/html/nuc` and if they are up to date. Only then you should start the NUCs.

Reload after changes



The HLCs use a startup script that takes the cpm library, Middleware etc. from an Apache server located on the Lab's main PC. The according files are updated every single time that these components are built. If you changed them during the lab run / if they are missing, please rebuild them and restart the NUCs/HLCs, so that they get the new software.

This decision was made to reduce upload traffic when files are uploaded to the HLCs during a lab run.

Make sure that all your files are uploaded correctly

Your folder structure must match the [recommended folder structure](#) for both the cpm library and the software repository. If your script/program relies on this data, then you should always refer to other programs using paths relative from `~` (so do not use `/home/username` explicitly). Cpm library and middleware can be found in the same folders on the NUCs. Required environment variables, e.g. for DDS, are set by default, so you do not need to take care of that.

All required data must lie in the same folder as your executable or script. The whole folder containing it gets uploaded to each NUC. If you depend on other data (except for the Middleware or cpm library, which are uploaded separately), put it in that folder, or your program might not work correctly.

Relative script paths are currently **not supported**. Also, please do not enter folders that may contain a lot of data (e.g. `./`), because all that data is then  added to all NUCs (which, although it gets deleted after they are restarted, might not be a good idea)

Usage of MATLAB

If you use Matlab you need to use the **init** script. Load required data (e.g. XML configuration files), setup the reader and writer etc. Alternatively, use a script that is inspired by the init script. `QOS_LOCAL_COMMUNICATION.xml` is always located in `~/software/middleware/build`, and precompiled DDS IDL Matlab files can always be found in `~/software/cpm_lib/dds_idl_matlab`. Other files cannot be expected to be found on the NUC, unless you put them in the same folder as your own script.

MATLAB example for init script

```

clc
script_directoy = fileparts([mfilename('fullpath') '.m']);
cd(script_directoy)

% Initialize data readers/writers...
init_script_path = fullfile('..', '/init_script.m');
assert(isfile(init_script_path), 'Missing file "%s".', init_script_path);
addpath(fileparts(init_script_path));
[matlabParticipant, stateReader, trajectoryWriter, systemTriggerReader, readyStatusWriter, trigger_stop] =
init_script(matlabDomainID);
cd(script_directoy)

```

General information

Mapping of vehicle IDs

The available NUCs are mapped to vehicles in an ascending order (NUC 11 vehicle 1, NUC 15 vehicle 4, ...).

Each NUC is always responsible for one vehicle only.

Using IDL-Datatypes and the cpm library

The location of the cpm library is always set using LD_LIBRARY_PATH in the local terminal - you do not need to take care of this in your program, as long as you have linked the library properly in your build script.

The Bash Folder

There is a folder called *bash* in the source code of the LCC. It contains scripts that are relevant for (remote) deployment of scripts, uploading scripts etc. You do not need to understand these scripts, but they are also used to load environment variables. You might encounter an error if you e.g. use another version of DDS than specified e.g. in *environment_variables.bash*. Your software might, for example, simply not start if the required variables could not be set. Make sure that you use the same package versions or change the entries accordingly.

Specifying the matlab version for HLCs on the NUCs

In the aforementioned *bash* folder there is a file *lab_control_center/bash/tmux_script.bash* in which the path to the Matlab version that the NUC uses to execute the HLC scripts can be specified. For example to select MATLAB 2022a:

```

...
#Evaluate the matlab script
SCRIPT_NAME="${SCRIPT_NAME%.*}" #remove .m
/usr/local/MATLAB/R2022a/bin/matlab -logfile matlab.log -sd "${PATH_TO_SCRIPT}" -batch "${SCRIPT_NAME}"
(${SCRIPT_ARGS})"
...

```

or MATLAB 2020a:

```

/opt/MATLAB/R2020a/bin/matlab -logfile matlab.log -sd "${PATH_TO_SCRIPT}" -batch "${SCRIPT_NAME}"
(${SCRIPT_ARGS})"

```

Integrated error checking

IDL-Datatype for error checking

The following data type is used to check for NUC-crashes.

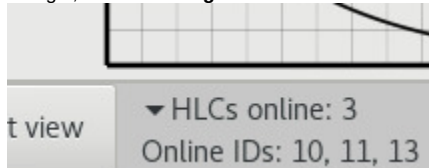
https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/dds_idl/HLCHello.idl

Explanation: Hello messages are sent regularly by each NUC. Missing messages or unexpected entries can be used to detect errors.

- **source_id**: This field can be used to obtain the ID of the NUC that sent the message
- **script_running**: True if a tmux session for the script is currently running. These sessions are started on the NUC when Deploy Remote is used, after the upload has finished. Usually, when a program crashes, the tmux session is stopped as well, thus this value can be used as an indicator to check if the script crashed.
- **middleware_running**: True if a tmux session for the middleware is running. Similar to script_running.

Online-indicators for NUCs

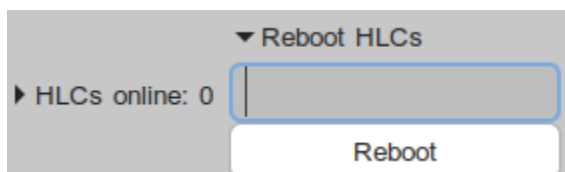
The UI indicates how many NUCs are online. The UI shows you how many NUCs are currently online (matching *might take up to a minute* after the NUCs have been booted and your program has started.) This can be useful for debugging or to find out on how many NUCs you can distributedly deploy your program before, for the rest of your selected vehicles, the computation for those takes place locally. You can expand the section by clicking on the little triangle, thus **revealing the IDs of the HLCs** which are currently online:



If the entry stays empty, you need to check what went wrong. Potential errors are:

- You are using the wrong domain ID (DDS) for the LCC. If LCC and NUCs are not within the same DDS domain, they cannot discover each other. Please check which ID you need to use.
- Packages are missing on the NUCs. In this case, upload packages are missing on the main PC or the NUCs have not been set up correctly (e.g. the autostart task, see requirements)
- The NUCs or your PC are not connected to the same network
- You are not using the main Lab PC (which is recommended for distributed deployment)
- The NUCs crashed

The following section *Reboot HLCs* allows you to **reboot HLCs of your choice**. You again need to click on the little triangle. An input will appear:



You can find out what to enter here if you hover over the input. You can either enter a comma-separated list that the LCC is supposed to reboot (you can check if that worked by examining the online HLCs simultaneously), or you can enter a '*' to tell the LCC to reboot all currently connected NUCs.

Troubleshooting



The NUCs do not show up as online in the LCC despite running?

- Set GCC 10 and G++10 as default on the main PC and all NUCs using ``update-alternatives``. It is important that the same version is used on all devices (main PC + NUCs).
- Rebuild the lab software
 - Run `git clean -xdf` and rebuild the lab software on the main PC with `./build_all.bash`
 - Start the LCC
 - Restart NUCs
- Update NUC scripts
 - ssh onto the NUCs, run the "lab_update" script in the "autostart" folder. The "autostart" and "lab_update" scripts should not be run as root, but as "guest" user.
 - The script may freeze, but the NUCs should still appear in the LCC. Do not terminate the "lab_update" script, but restart the NUCs (via the LCC or via the ssh connection).
- If problems occur when running the "autostart" scripts on the NUCs, check and adjust the file/folder permissions. If necessary, you can also examine the scripts by inserting debug output.
- Copying to the NUCs requires free memory on the main PC. The more NUCs are used, the more free memory is required. In general, problems are to be expected with less than 5GB of free memory.

The upload of your HLC-Files crashes due to denied permission?

- The ssh-connection is established using an ssh-key without a passphrase. This ssh-key belongs to the user `cpm` on the Main-PC. Thus switch to this user or adapt the scripts in `lab_control_center/bash` temporarily to establish the ssh-connection using the `guest` users password.

Crash checks for your programs

As soon as you actually deploy your programs, and after the scripts have been uploaded, the LCC also starts checking if their corresponding tmux sessions (middleware and script) are still running.

Errors are reported in the UI if that it not the case anymore (e.g. with a popup similar to [local crash checks](#), and in the [monitoring window](#) for each vehicle).

Integrated Debugging Tools

Script Output Folder

Scripts that you have started from within the LCC (as well as the middleware) are managed using [tmux](#) sessions (which is similar to screen).

Tmux is used to create, manage and kill virtual console sessions, which can be used for any purpose a console can take care of - in this case e.g. for starting (and stopping) your selected script. Tmux sessions always have a unique name by which they can be identified, e.g. *middleware* for the Middleware session. This is especially useful in case a tmux session could not be destroyed after a program crash: If a new session with the same name gets started from the LCC, the old session gets overridden.

The script that takes care of these sessions also writes command line outputs of the running programs into `.../software/.../lcc_script_logs`. For example, the script to start and run the middleware is called `tmux_middleware.bash` and can be found in `lab_control_center/bash`. If you want to find out why your program crashed, the contents of this folder might be useful, in addition to log messages in the system and other custom error reports. You can find this folder for remote deployment as well (on the NUCs).

`lcc_script_logs`

 This folder only appears after you have started a simulation. It is created by the LCC - relative to the LCCs location - to store logs of the running script and Middleware. Some of its parts are deleted with each simulation - recording, Labcam, Middleware, HLC and remote_script logs. These change / become invalid with each new run.

Remote

 On HLCs, the same folder exists (for the guest user), also at `.../software/.../lcc_script_logs`. It gets deleted and recreated (including its contents) every time a new remote simulation is started.

Further Log-Files

Additionally to the output mentioned above, the CPM Logger creates log files with the name `Log_#Date_#Time.csv` for each execution. These are created in the folder from which the LCC was called.

You can also find Logs of the LCC child there, which takes care of calling external programs. These logs are called `Log_child_process_#Date_#Time.csv`. They only include serious error messages, which can also be seen in the command line. In case of a crash of the LCC (e.g. due to errors occurring in the LCC child during runtime), you may only see these errors in the log files though.

Crash Checker

Most scripts started by the LCC run in tmux sessions (for more information, see Script Output Folder). These sessions have distinct names.

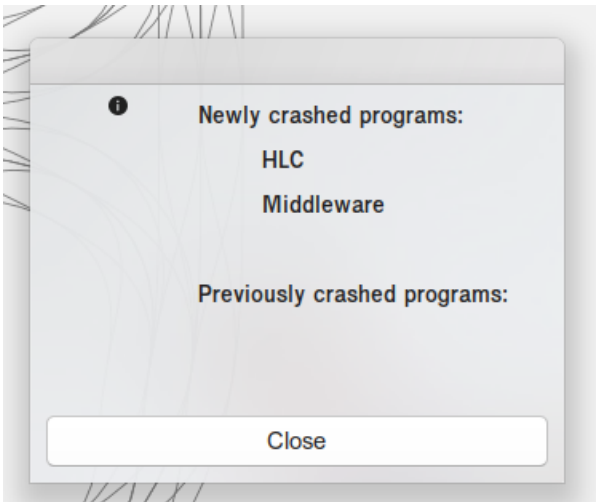
Other tasks

 Tasks with a limited runtime are managed using fork and the resulting PID, e.g. for sending reboot commands to the vehicles. These tasks are not observed by the crash checker, as they are not strictly part of a running simulation and are not supposed to be always online. Tmux offers a more convenient way to manage tasks, to detach them from the LCC and to observe if they are still online, and is thus used for any simulation-related externally started program (User-provided script, Middleware, IPS...)

Sessions exist for the Middleware and your script (if a locally running simulation was started and a valid script was selected). There also exist sessions for the [labcam](#), [position detection](#) and other vital features of the LCC (if they are activated).

The integrated crash checker checks regularly - during a simulation - if all these tools are still running. To simplify the process, we only look up if the according tmux sessions exists, which in most cases is enough to detect a crash of the contained program.

In case of an error, a LCC log entry is created and an additional popup window (which needs to be closed manually) warns the user about the undesired behaviour.



Local Deployment

If you deploy your script locally, the process described above can be used to check if your script and the middleware are still running.

Remote Deployment

Take a look at [this page](#). The crash checker also monitors running tmux sessions on the NUCs, if they have been deployed and the simulation is still running.

Round Trip Time (RTT) Measurements

Functionality

The round trip time is a measurement for the time (here: in milliseconds) a direct message exchange with instant answers takes between two participants within the network. It may differ with network load or may even produce no results in case of serious connection problems. Thus, regular RTT measurements provide a simple metric for the current quality of the connection to participants in the network and its stability.

The LCC regularly sends RTT measurement requests *as long as no simulation is running* (to prevent network overload). The RTT measurement gets updated at least every 2.5 seconds and at most every 0.5 s, depending on the time the longest RTT message takes to get received.

These messages are received by different participants, which can send an answer to this request. It consists of their *source_id* (e.g. *vehicle* or *hlc*), and additional information to make sure that the answer is up-to-date etc.

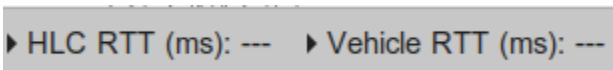
The LCC either waits for these messages until a timeout occurs, or waits for some additional time after receiving the first message to determine when - approximately - the latest answers arrive.

These messages - for HLC and vehicles - are then processed to show the user the current best and worst RTT measured for each of the two participant types, the percentage of requests with missing replies since the start of the RTT measurement (since the last simulation) and the all-time worst RTT.

If no replies were received for some time (10 seconds), then all participants of that type are assumed to be turned off. The according data structures are reset and the UI does not show any information regarding the RTT of the participants anymore.

UI

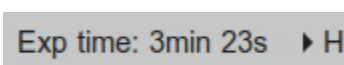
Picture of the LCC monitoring bar including information about the RTT, when no measurements could be made (because the participants are offline).



TODO: Insert image with example for working RTT measurement.

Experiment Runtime

A small indicator has been added that shows the user how long the experiment has been running. It is also part of the monitoring bar. It displays hours, minutes and seconds (if they are greater than zero).



DDS Datatype

The following datatype is used to measure the RTT:

```
struct RoundTripTime {  
    string source_id; //To find out where the msg came from and to not answer to own msg  
  
    boolean is_answer; //If true, do not respond to this msg, as it already is an answer to an RTT request  
    octet count; //Counter to not mix up old with new RTT requests  
};
```

Implementation

The implementation of the calculation of the RTT as well as answering / sending RTT requests is part of the cpm library. They can be activated using the activate function, e.g. for the HLC:

```
cpm::RTTTool::Instance().activate("hlc");
```

Here, *hlc* implies that this participant is responsible for measurements regarding the HLC, and answers to HLC RTT requests. You do not need to take care of this - HLC measurements are part of the [autostart program](#) on the NUCs.

The LCC includes some more tools to aggregate the calculated RTTs and to display them in the UI.

Modules

- [CPM Library](#)
- [Command Line Parser](#)
- [Parameter Service](#)
- [Timer - Periodicity and Synchronization](#)
- [Visualization - Used in the LCC](#)
- [Commonroad Information for the HLC](#)
- [DDS Reader and Writer](#)
- [DDS Participants and Topics](#)
- [RoundTripTime](#)
- [Logging](#)
 - [Logging Structure](#)
 - [Logging to DDS](#)

CPM Library

The cpm library is used by the vehicles, the Lab Control Center and the Middleware. It provides common functionality for DDS participants in the CPM lab.

During its build process the idl files are generated and stored in the `dds`-folder.

All functionality covered by the cpm library *should be used* by any user whose application needs to *participate in a DDS domain*. On top of that, it also provides some basic features, like a command line reader. Only rely on other methods, e.g. as described in [RTI DDS Overview](#), if the cpm library does not provide what you need for your implementation. For details on the modules provided by the cpm library (as well as other modules), see [Modules](#).

To initialize the cpm library, the following two functions are always required to be called in the main function before anything else:

```
cpm::init(argc, argv);  
cpm::Logging::Instance().set_id("desired_id");
```

Command Line Parser

The CPM lib provides a standardized way to read command line parameters.

Example

The program is executed with `binary_name --my_int=42 --simulated_time=true`

```
#include "cpm/CommandLineReader.hpp"
bool use_simulated_time = cpm::cmd_parameter_bool("simulated_time", false, argc, argv);
int my_int = cpm::cmd_parameter_int("my_int", 5, argc, argv);
```

You can read in int, uint_64, double, float and string as scalar or vector. For further information refer to our source code:

References

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/CommandLineReader.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/src/CommandLineReader.cpp

Parameter Service

The parameter service is used for initialization purposes across all participants. Several parameters can be set in the LCC and are distributed on startup / requested when other applications in the network are started.

CPM Parameter Interface

This interface allows the user to request a parameter from the LCC. The parameters that can be requested are listed below.

All requests are blocking, meaning that parameter requests are sent to the LCC periodically until an answer has been received, which is then returned. A parameter storage is implemented as well - thus, data that has been received and stored before is not requested again but returned immediately. Parameters may change during runtime, if so then these changes are handled by an internal callback function that updates the parameter storage.

Multiple parameter types are supported.

Receiving Parameters

The class `ParameterReceiver` in the `cpm` library includes a reader and a writer to request and receive parameters from the LCC. It also contains a storage that saves already received parameters and their values, using mutexes for thread-safe access, so that requests do not need to be sent multiple times. Its implementation includes the implementation of the functions for `Parameter.hpp`.

Setting Parameters

Parameters are set in the LCC. The class `ParameterStorage` is responsible for loading, storing and changing parameters, and `ParameterServer` listens for requests of other participants of parameter values and sends these values on request or whenever they have been changed in the LCC.

The usage of YAML files and the handling of parameters within the LCC's UI allows the users to conveniently set parameters for multiple participants without the need to change start-up scripts or multiple configuration files. An example of a parameter distributed using the parameter server are the lateral gain values of the vehicles.

References

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/lab_control_center/src/ParameterServer.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/lab_control_center/src/ParameterStorage.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/Parameter.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/ParameterReceiver.hpp

Timer - Periodicity and Synchronization

Important

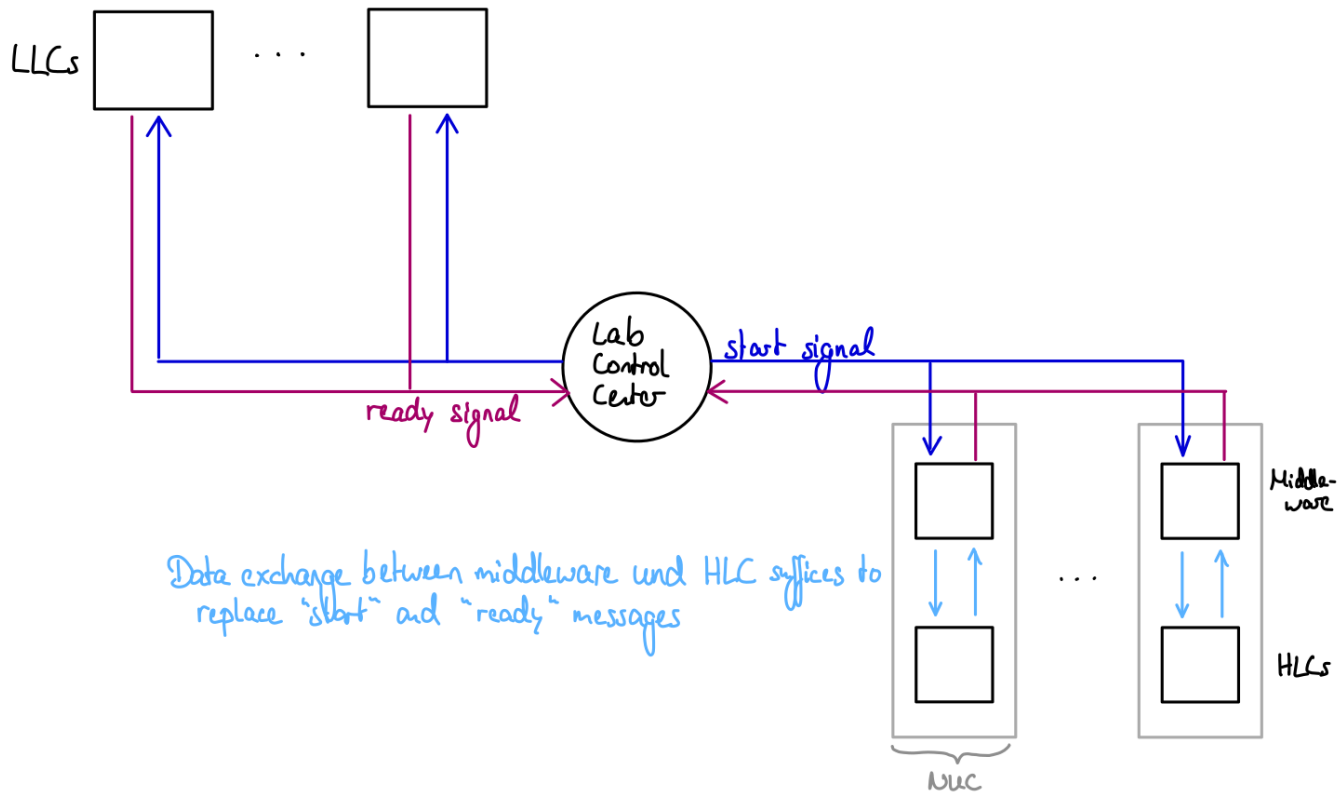
The timer can also be used to implement periodical behavior in your program, as its callback function is called every X nanoseconds after the timer has been started (X is set by the user, and the timer must not be started externally if you set it not to listen for start signals).

The CPM timer is used to achieve a **synchronized behavior across different devices** that cooperate in the network. For example, all vehicles are supposed to publish their current state (like battery voltage) **periodically** at the same time. Such a behavior can be achieved with the CPM timer. Timing is either performed using the current real time (system clock, synchronized using NTP) or a fictitious time, which is distributed by a central timing instance.

Synchronization across all participants in the RTI DDS network is very important for all tasks that

- are performed on a regular basis and
- require up-to-date information from other participants in the network to perform these tasks properly

Synchronization allows to coordinate different tasks across the network and gives some guarantees about the shared information. Some of these tasks are caused e.g. by the mutual dependency between LLCs and HLCs. LLCs require **regularly updated commands from the HLCs** to maneuver crash-free and as desired according to e.g. their planned trajectory. HLCs need up-to-date vehicle states from the LLCs to calculate commands that accommodate the current state of the overall system.



CPM Timer (Timer.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/Timer.hpp

The timer class can be used to call tasks periodically, based either on the **system clock** (**TimerFD.hpp**) or the **simulated time** (**TimerSimulated.hpp**, timing signals are given by the LCC). It **can wait for a central start signal** and can be **stopped when a stop signal is sent** (both by the LCC). Thus, timers on multiple systems can be started at the same time, and then work simultaneously if their period is set accordingly as well.

Several parameters must be set to use to cpm timer:

- `node_id` (String): The unique ID identifying the timer in the network, so that the LCC can tell different timer signals apart
- `period_nanoseconds` (uint64_t): The callback function of the timer is called every `period_nanoseconds` nanoseconds
- `offset_nanoseconds` (uint64_t): Initial offset - 1 in real-time case (offset from unix timestamp 0), first callback time in simulated time case

- `wait_for_start` (bool): If false, real-time timing is started immediately after creation without waiting for the LCC - this parameter is ignored by the timer that uses simulated time, as there it is required to listen to a central timing instance
- `simulated_time_allowed` (bool): Specify whether the task can be performed using simulated instead of real-time
- `simulated_time` (bool): Indicates if simulated or real time should be used

The callback function is registered within the `start(...)` function. The timer can be started synchronously, blocking / using the current thread until it was stopped, or asynchronously in a new thread, using `start_async(...)`. It can either be stopped manually using `stop()`, which is especially useful if the LCC is not used for real-time timing purposes, or it can be stopped with an LCC command.

The **callback function** registered in `start` is **called periodically** until the timer is stopped, according to period, offset and the usage of simulated time. The following figures illustrate the role of the timer in combination with the LCC as central timing instance.

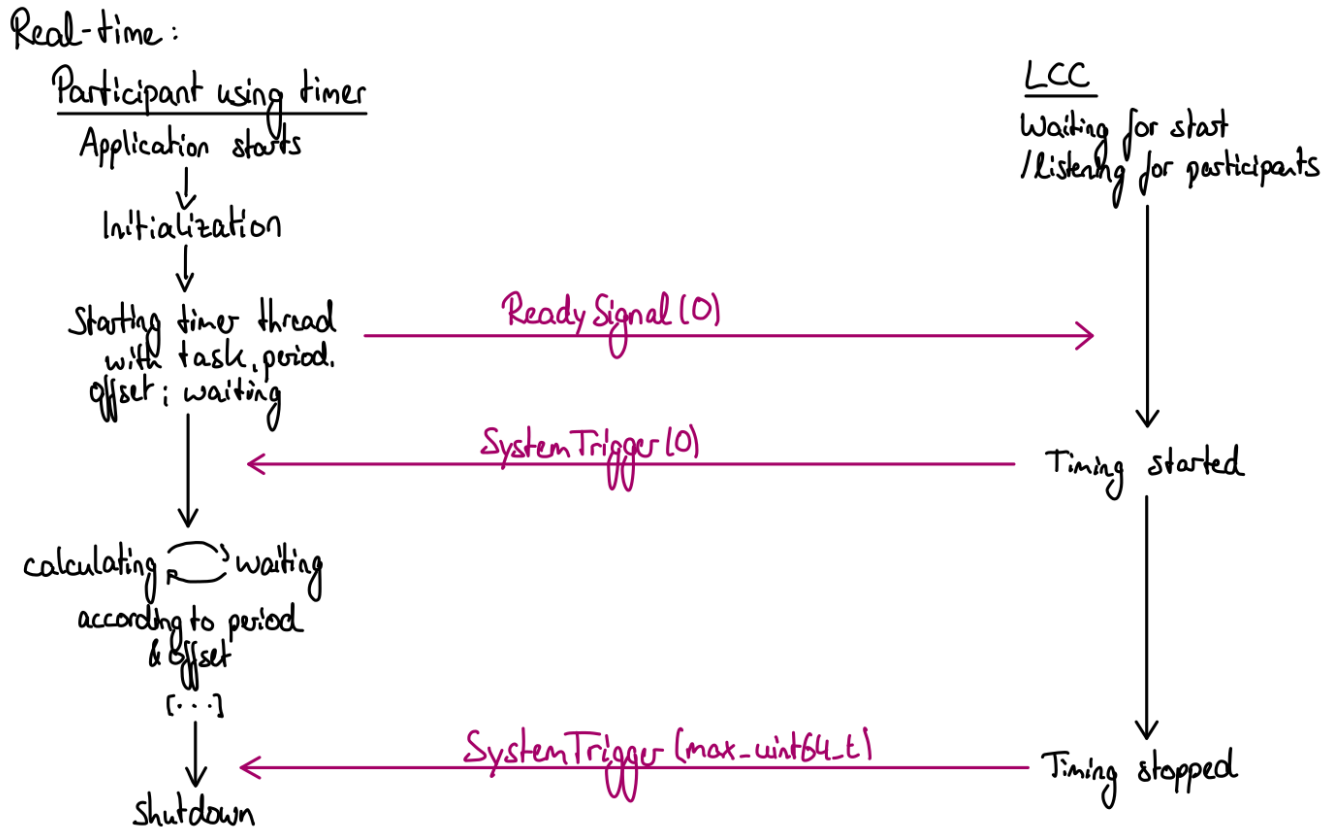
There is also an option to set a **callback for the stop signal**: If it is set, the timer is not stopped when a stop signal is received, but instead calls this function. You can set an empty callback to ignore stop signals, or define your own stopping behavior. You can call `stop()` at the end of the callback on the timer object to actually stop the timer itself afterwards.

Real-time (TimerFD.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/TimerFD.hpp

Real-time refers to the **actual current time of the system**, given by the **system's current timestamp**. To make sure that timestamps across different devices are comparable, **NTP is used for clock synchronization** on these devices. In a real-time scenario, all participants begin their computation (after being started either manually or by the LCC) at $\text{unix time} + \text{period} * n$, where n is the smallest n such that the overall term is greater than or equal to the point in time when the timer was started. This assures synchronized periodic behavior for the registered components (using callbacks) across all devices. Real-time is useful whenever the system is used for real driving scenarios, e.g. when the actual physical vehicles are used.

The following drawing shows which role the LCC plays in the real-time timing case.



This timer's constructor also takes an **optional parameter to re-define the stop-signal** to another value than `max. uint64_t`. *Do not change this value unless you know what you are doing* and only if you want your timer not to react to a global stop signal within the system, but a custom one. (If you just want to **ignore the stop signal**, define your own, potentially empty, **stop callback function** and pass that when starting the timer **instead**).

The timer also allows to get its start time. Most other details were already explained in the **CPM Timer** section.

Important

 Only use `TimerFD` specifically if you do not need your timer to be able to switch to simulated time!

"Real" Time (SimpleTimer.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/SimpleTimer.hpp

This timer is **not intended to be used for actual real-time** purposes. It can be used instead **for GUI tools etc. to achieve periodic threaded callback** of a function in **periods** which are **multiples of 50 milliseconds**. Apart from that, it works like the timer above (TimerFD).

This timer can be killed easier - while a **TimerFD** timer might take **up to period_nanoseconds** nanoseconds to be killed, the **SimpleTimer is killed** or destructed **within ~50 milliseconds** and thus does not block e.g. the UI for too long.

The support for a custom stop signal was mainly implemented to be used with this timer - if the SimpleTimer is used for other purposes than simulation, then it might be undesired that the simulation's stop signal can stop this timer. To stop multiple running SimpleTimers at once, it might still be in the user's interest to use a stop signal (differing from the one used in the simulation).

If you just need threaded periodic behaviour in your code, without any real-time requirements, then use this timer instead of an own implementation whenever possible.

Simulated time (TimerSimulated.hpp)

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/src/TimerSimulated.hpp

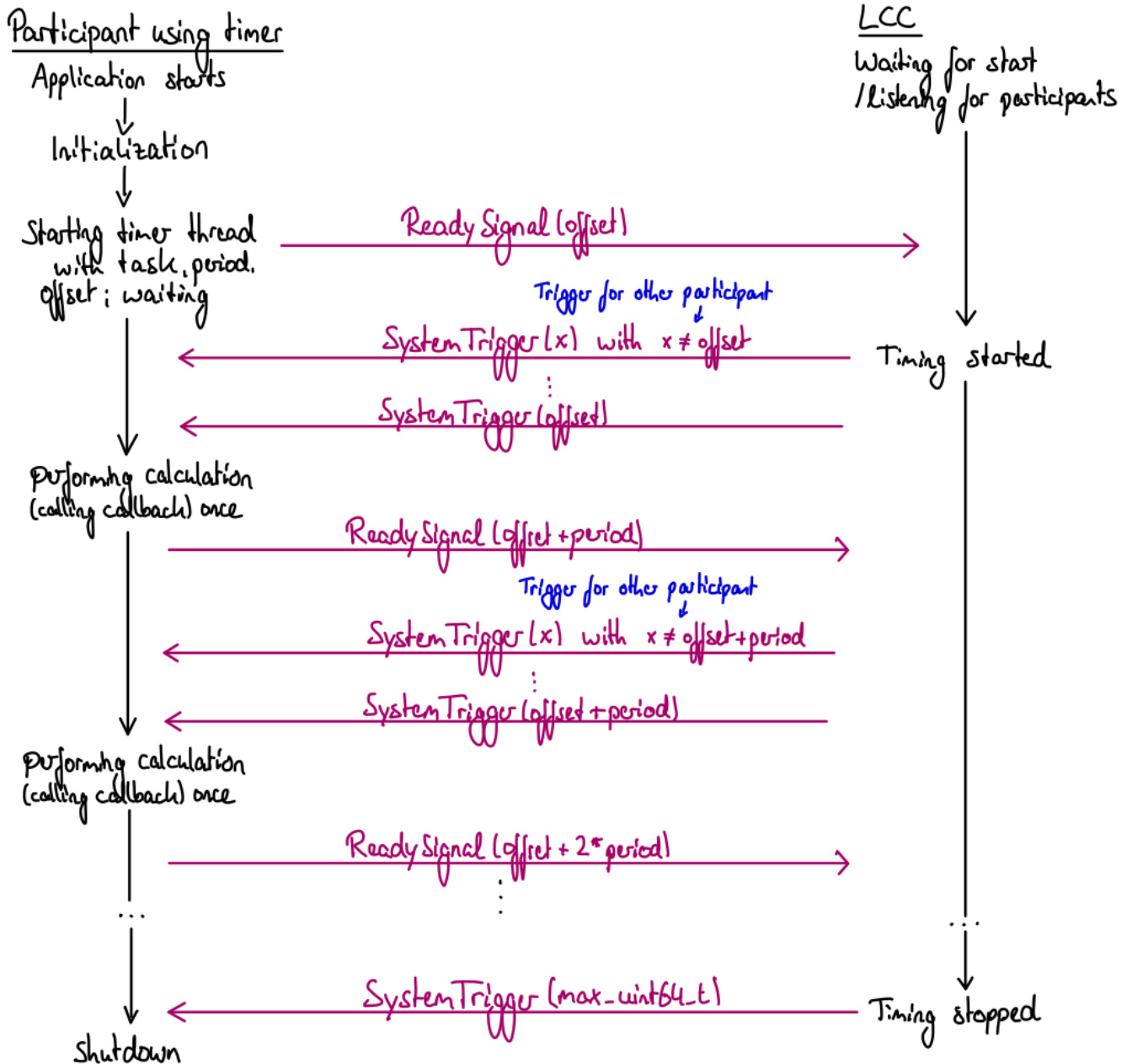
Simulated time refers to a fictional measure of time, that is not related to the unix time of the physical devices on which the different tasks operate. Instead, the current time is determined by a **central timing instance**, in this case by the LCC. Simulated time can be **used to speed up or slow down the simulation** of a scenario, which might help to detect errors, to improve the timing of the individual components or to test faster how the system operates on the long run.

In a simulated time case, it is **required that all participants are uniquely identifiable** (here by setting **node_id** differently for all participating tasks). After being invoked on their currently set time step, the participants register the next time when their callback function should be called according to the fictional time measure, which is based on the offset and period settings. Initially, the LCC collects all these ready signals for the first timer callbacks until timing is started manually by the user (or until ready signals from all pre-registered timers have been received). It then performs the following tasks in a loop:

- Select the smallest next fictional time stamp of all received timestamps (that is greater than the current simulated time) and send a system trigger to all participants containing this time stamp
- Wait for all participants that registered this timestamp to register a new, greater timestamp

The simulated time progresses this way until the timing is stopped in the LCC. The time can of course only progress if all participants send new ready signals after they have been invoked (and after the work of their callback function has been done).

The following drawing illustrates which role the LCC plays in a simulated time use case.



Important

Try not to use `TimerSimulated` specifically - use the `CPM Timer` instead, which can switch between real and simulated time based on the settings taken in the LCC.

LCC Timer

As mentioned before, the usage of **simulated time and start and stop signals** requires a central timing instance. This instance is **located in the LCC**. Timing can be controlled by the user using the LCC's **UI**.

The central class for controlling the timing in the system is `TimerTrigger`. In a real-time scenario, the class is merely used to send start and stop signals, and to keep track of which participants initially registered with a ready signal. These participants are shown in the UI, so registering them allows the user to see if they perform as expected up to the creation of the timer - if they do not show up, something must have gone wrong in the application.

The class is also responsible for handling simulated time. Here, further tasks are required. The current simulated time as well as the newest ready signals of the participants need to be stored. From the current time and the participants that rely on it, the system can also infer whether a participant is currently working or waiting for another time step, and if it is out of sync (if only old ready messages are received by it). The class receives and processes all ready signals so that the progression to the next smallest time step works properly. Of course, this also includes waiting for participants that are currently working and have not yet registered a new time step in which they want to be woken up again.

get_time_ns.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/get_time_ns.hpp

Whenever you want to **obtain the current system real-time** (as timestamp, in nanoseconds, using clock_gettime internally), use this function. This can be useful to e.g. obtain timestamps for your messages.

stamp_message.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/stamp_message.hpp

The function name is self-explanatory. If your message types includes and uses Header (https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/dds_idl/Header.idl), you need to **set a create and valid_after stamp**. Both can be set **in one line** using this function, where you pass your message object as reference so that the fields can be set on the original object. It is thus merely a **convenience function** that you are not required to use.

The create stamp usually should be the time of message creation.

The valid_after stamp tells the receiver when the message should become valid, e.g. right now (same as create stamp), in the near future (i.e. create stamp + 1000000 nanoseconds) etc.

References

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/lab_control_center/src/TimerTrigger.hpp

https://github.com/embedded-software-laboratory/cpm_lab/tree/master/lab_control_center/ui/timer

Visualization - Used in the LCC

Visualization messages can be sent to the LCC to draw basic shapes and text in the map view, where the vehicles are shown as well. An example use case for this would be to draw the desired path that a vehicle is supposed to take on command, to check if it actually follows this path. The messages can also be used to show additional information to the running program or simply to debug the sent trajectory by checking the created path visually.

Data structure

Each visualization message thus consists of seven data fields which need to be filled accordingly to draw the desired information in the LCC:

- ID
- visualization type
- time_to_live
- points
- size
- string_message
- string_message_anchor
- color.

ID

Each visualization command is identified by a **unique** ID. Choose different IDs for commands that should be displayed at the same time, or use the **same ID to override** an older visualization message.

The **highest ID is always drawn last**, so, if you want to draw symbols on top of each other, always make sure that you use the right order of IDs.

Visualization Type

Currently, four command types are supported:

- Line
- Polygon
- Filled circle
- String

Lines and polygons are defined by the points which are set in the structure data as well. In both cases, lines are drawn between subsequent points in the array. The first and the last point are also connected by a line if you want to draw a polygon.

If you choose a string, you need to specify a single point as anchor. The drawn string will be attached to this anchor as specified in the field `string_message_anchor`. A string message will only be displayed, if you set the field `_String Message_`.

A **filled circle** requires a single point which will be its center.

Time to Live

Each visualization command is **removed automatically** after a given time. Here, you can set the time to live in nanoseconds.

Points

This field defines the set of points between which lines should be drawn, or the point where the string message should be shown. The [coordinate system](#) is the same as the one used for the vehicles.

Points is a custom IDL data type which is a struct `Point2D { double x; double y}`. Points are set according to the [coordinate system](#).

Size

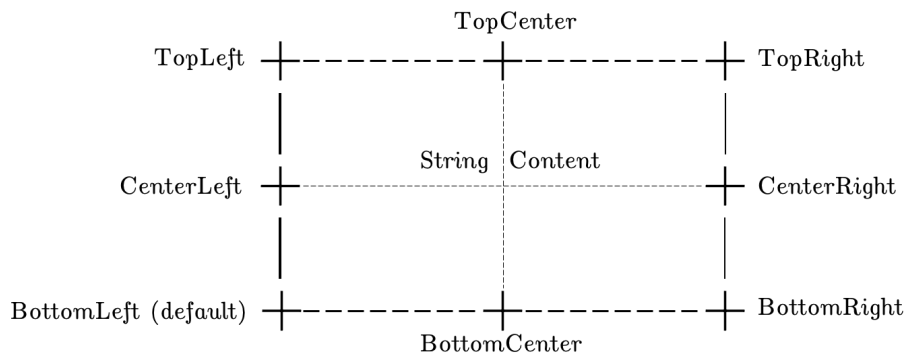
Polygons and lines interpret the size as **line width**, string messages as **font size**, and circles as **radius**.

String Message

Just set a string for this field (as RTI uses its own string format, be aware that setting this value with a `std::string` data type might lead to an error message, but give it a try before you use RTI's own implementation).

String Message Anchor

Defines where a string message will be aligned, relative to the anchor point (the first entry of `points`). You only have to specify this field manually, if it should be different from the **default: BottomLeft** corner.



Center is also a possible value.

Color

Lines, polygons and strings can be given any color within the RGB spectrum.

Color is a custom IDL data type which is a struct `Color {octet a; octet r; octet g; octet b}`. Color values are just **regular RGB** values. The **transparency** value should be **ignored**.

Usage example

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/lab_control_center/test/VisualizationTest.cpp

Commonroad Information for the HLC

The HLC can receive some information from the Middleware, loaded from a Commonroad scenario, after starting the simulation.

Goal States

Please take a look at https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/dds_idl/CommonroadDDSGoalState.idl

The definition of goal states was done similarly to the specification in Commonroad. Goal states themselves are part of a planning problem, which may include several goal states, and one planning problem may consist of several planning problem "elements".

Because we could not encapsulate this definition of planning problems in DDS without causing memory problems, we send each goal state individually instead. You need the following three information to associate them with their original definition:

PlanningProblem:

- planning_problem_id
- sequence of planning problem elements:
 - > at position planning_problem_pos within that sequence: planning problem with
 - sequence of goal states:
 - > at position goal_state_pos within that sequence: goal state

So the identifiers `planning_problem_id` `planning_problems_pos` `goal_state_pos` tell you which goal state you are currently working with. All other information should be self-explanatory or are similar to the Commonroad specs.

DDS Reader and Writer

A Short Notice on (A)Synchronous Readers

You can process data that you receive in two ways:

- **Asynchronously:** The data is read immediately when it gets received (similar to interrupts, but in another thread instead of interrupting the current control flow). You can get access to this data and e.g. store it in a thread-safe manner (e.g. using mutexes) into your own data structures, which can be processed by your own threads or your main thread.
- **Synchronously:** The data is stored in a buffer. When you call a function (like *take* or *read*), the received data (all data since the last function call / only the most recent one / ... depending on your QoS settings) can be obtained. This can be done e.g. periodically in your main thread.

Reader.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/Reader.hpp

If data is read synchronously, the user might be interested in the most recent data only. For most of our own data types, we use a self-created header that contains timing information:

- `create_stamp`: Time (in nanoseconds) of data creation. The higher, the more recent is the data.
- `valid_after_stamp`: Time (in ns) when the data can be used. Data is not supposed to be used before this point in time is reached.

The Reader class can be used to create a data reader from which **only the most recent valid data can be requested**, using `get_sample`. Data is valid if its `valid_after_stamp` is lower than the current timestamp `t_now` that is passed to the function with which you can obtain the most recent data (the .hpp should contain all required information). You can use it for data types like trajectories or vehicle data.

The Reader stores all received data in a ring buffer. The order in which data arrives is not regarded when data gets overwritten. Thus, some slightly newer samples might get lost if `get_sample` is called infrequently - the Reader's buffer is not flushed regularly in between - and if data does not arrive in the correct order.

ReaderAbstract.hpp

https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/include/cpm/ReaderAbstract.hpp

If you only need a **basic DDS reader**, then you should preferably use this class. It only **offers information regarding matched publications and a `take()` function**. It does not require the user to know or understand DDS implementation details - please only use more advanced concepts if you really need to. **Setting QoS parameters** is available through the constructor.

AsyncReader.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/AsyncReader.hpp

Data received by a **DDS Reader** can either be read synchronously or **asynchronously**. The latter requires the usage of e.g. a `StatusCondition`, an `AsyncWaitSet` and a callback function. To save the users the time to look up which `StatusCondition` is (mostly) required and to write the > 10 lines of code required for every single reader that relies on a callback function, `AsyncReader` is provided as a sort of **wrapper** that performs the set-up.

The class **must be passed a callback function**. Please be aware that your callback function will be called within the **AsyncReader's own thread**, so you need to use e.g. a mutex to protect your data, and you might have to use lambda captures or `std::bind` to be able to access e.g. your class members from within the function. It also requires a domain participant (or none, the default then is [ParticipantSingleton](#)) and a topic (or a filtered topic) or topic string (for the topic name). Reliable communication can be turned on / off with a boolean parameter.

MultiVehicleReader.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/MultiVehicleReader.hpp

Like [Reader.hpp](#), but for multiple vehicles at once: Gather all received messages of type *T* by all vehicles that publish on the (filtered) topic *my_topic*, then return all newest samples when `get_samples` is used. For more information on the implementation, refer to [Reader.hpp](#). An example for a `MultiVehicleReader` is provided at the [central routing example](#).

Important

The reader in `MultiVehicleReader` uses a history of up to 2000 samples (it only buffers the most recent 2000 samples). This is due to performance reasons and because `take()` might not return all received messages immediately (it sometimes needs to be called more than once internally). The value of 2000 samples might not be enough depending on your scenario. In that case, i.e. if messages of some vehicles are sometimes missing because the buffer is not large enough, you might have to increase this value.

Writer.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/Writer.hpp

This is a basic **DDS Writer**, which only **offers information regarding matched subscriptions and a write() function**. It does not require the user to know or understand DDS implementation details and is thus recommended whenever a basic writer is required in your application. **Setting QoS parameters** is available through the constructor. You can find examples for a writer-implementation in every provided example as it is essential to transmit the trajectory to the vehicles.

DDS Participants and Topics

ParticipantSingleton.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/ParticipantSingleton.hpp

Most communication takes place within a single DDS domain. As - per application - only one participant should be created per domain, the most commonly used domain participant is provided via a Singleton. It is used in the cpm library as well. Whenever you need to communicate with the vehicles, parts of the LCC or other parts within the lab that do not use their own participant (like Middleware < - > HLC), use this participant. Most of our constructors give you an option to leave out a participant e.g. when creating a Writer - in that case, **the default choice is the ParticipantSingleton**.

Participant.hpp

https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/include/cpm/Participant.hpp

A simple participant abstraction. You can pass a domain number and an optional QoS file. Use this whenever you need to create a participant that is not part of the common DDS domain given by our ParticipantSingleton. This participant **needs to be passed explicitly** to e.g. a Writer so that it does not use the default participant.

get_topic.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/get_topic.hpp

For each domain participant, a topic can only be created if a topic with the same name has not been created before. This can lead to some trouble, e.g. if different classes depend on the same topic object but sharing it would be too complicated or would result in ugly code / a lack of data abstraction. RTI DDS provides a `find` method to find already created topics. As the user should, in our opinion, not be forced to decide which object should be created first or to find out which topics have already been created by the lib, `get_topic` uses both the create and the find method. Independent from the order in which the objects are created, `get_topic` returns the required topic object without throwing an error.

If only the topic name is passed as a parameter, a topic is created / looked up for the participant singleton instance. Else, the participant must be passed as well.

Most of our other wrappers, e.g. the Writer, do not require you to specify a topic - a topic string for the topic name suffices. The topic is then looked up internally.

VehicleIDFilteredTopic.hpp

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/include/cpm/VehicleIDFilteredTopic.hpp

If not all data that is received by a DDS Reader is of interest, a filter can be used to only process data that matches the filter expression. The most commonly used filter in this project filters by the vehicle id. The `VehicleIDFilteredTopic` thus provides a wrapper that uses such a filter. The vehicle id and the topic can be passed as a parameter, and the returned object can be used instead of the topic object. This allows e.g. the Reader to only process information that is relevant for it.

RoundTripTime

For debugging purposes, it might be interesting to determine the round trip time (RTT) within the system. We use RTT messages (https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/dds_idl/RoundTripTime.idl) to compute this time:

1. Send an RTT message to another participant
2. Wait for the participant to answer with another RTT message
3. Use the time diff to compute the round trip time in the system

Some more information on its usage can be found [here](#).

Usage

The module that abstracts this functionality is called RTTTool and can be found here:

https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/include/cpm/RTTTool.hpp

You only need to consider three functions:

Instance()

This function creates and returns an RTTTool Singleton that you can use within your program. It uses the [ParticipantSingleton](#) internally, so you can only measure the RTT within that domain.

activate(...)

Call this function to allow it to **answer to RTT measurement requests**. You are only allowed to **either activate or measure** the RTT with the same program, but do not do both at the same time - it does not make sense to answer to RTT requests sent by the same program.

You need to set an **ID** that identifies your program. It does not need to be uniquely identifiable - i.e. the vehicles in the Lab all identify as *vehicle*. But it should identify your program as what it is, so that the information makes sense in the measurement, and does not get mixed up with e.g. the information sent by the vehicles.

measure_rtt()

Call this function to **measure the RTT** in the network. After some computation time (waiting for answers etc.), it returns a map:

- Key: std::string, **ID** of the answers
- Value: **Lowest and highest RTT** measured with this ID (e.g. for multiple vehicles: Fastest and slowest RTT)

You can use this information e.g. to display it in the LCC, [which has already been implemented](#) for the vehicles and HLCs.

Logging

- [Logging Structure](#)
- [Logging to DDS](#)

Logging Structure

The Software logs various aspects during its execution. The logs can be classified into ones that are logged via network and ones that are just standard out or standard error.

- 1. cpm::Logging
- 2. std::cout

See [Logging to DDS](#) for information about cpm::Logging.

Log aggregation

Logs are being collected at /tmp/lab_software_logs/.

After an simulation experiment the structure might look as follows:

Log folder structure

lab_software_logs
|-2023-04-09--11-35-29
| |-2023-04-09--11-35-44
| | |-stderr_middleware.txt
| | |-stdout_middleware.txt
| | |-stderr_high_level_controller.txt
| | |-stdout_high_level_controller.txt
| |-stderr_vehicle3.txt
| |-stdout_vehicle3.txt
| |-stderr_vehicle2.txt
| |-stdout_vehicle2.txt
| |-stderr_vehicle1.txt
| |-stdout_vehicle1.txt
| |-stderr_tmux_kill.txt
| |-stdout_tmux_kill.txt
| |-all_received_logs.csv

After a physical experiment using the NUCs the contents could look like this:

TODO

Log deletion

 The logs in /tmp/ are usually deleted on boot. However this might depend on the Operating System.

Where are logs being written

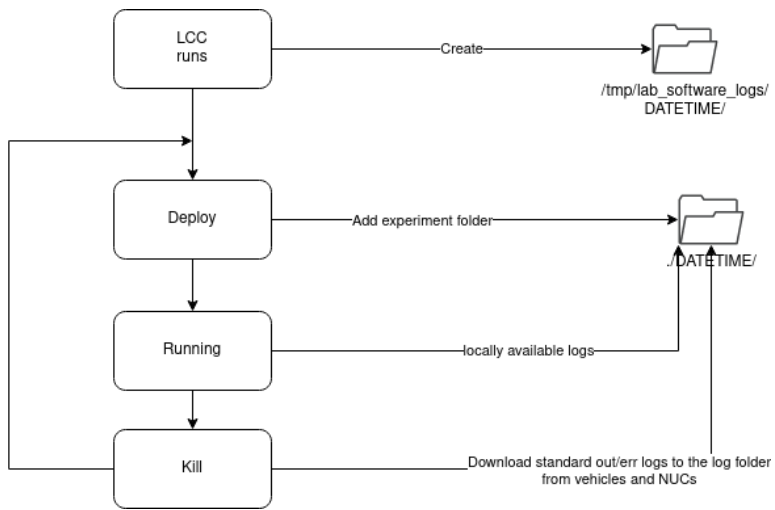
Cycle of logging

A per lcc session folder is created.

Per experiment another folder is created.

Locally generated logs are directly placed in there.

After an experiment is ended logs from the vehicle and NUCs are downloaded to the log folder automatically.



Logging to DDS

Logging.hpp

Logging provides an interface to log important messages and error messages. These messages are printed using cerr, stored in a file locally and sent to the LCC for log collection - the user does not need to think about how the log messages are stored and can simply use the .write function. A Logging Singleton is used so that the object must not be passed around between all classes that implement the feature.

Warning - use set_id: The ID of the Logging Singleton must be set at the beginning of the program whenever the cpm lib is used. If no ID would be set, the sender of the logging messages could not be identified by the LCC and the feature would become meaningless. As the cpm lib relies on the Logger, the ID must thus be set as early as possible, after the creation of the Singleton and before any other part of the cpm interface is used.

The write function can be used similarly to the C-style fprintf function. This is much easier than any cout-style function, where it is unclear when to flush and send / store / print the data, even if temporary objects are used (the user might store the object).

To uniquely identify the log files, the time of initialization of a log is used as a name for the log file. ID, timestamp and content are stored in this file and sent to the LCC. The cerr output does not include the ID, as it is used to identify the program only. The log file is realized as a csv file, which is easy to write and supported by most spreadsheet programs.

Log Level (WIP)

Logging does not always need to be verbose, and might not be required during demonstrations for performance reasons.

The log level is set for all participants using the LCC's UI, which gives options to set the log level correspondingly (see [here](#) for more information about how to set the log level in the UI of the LCC).

The name of the topic is **logLevel**. It is an unsigned short and has four different values:

Log level	Description
0	Do not log anything (e.g. for performance reasons)
1 (default)	Only log critical failures (only those that cause immediate critical system failures)
2	Also log less relevant error messages (e.g. including when a condition or specification could not be met)
3	Verbose (log even simple things, like whether a message was received)

You do not need to read the log level. The cpm library does that internally in combination with the Logging feature.

Example

```
#include "cpm/Logging.hpp"
//Always required if the cpm lib is used
cpm::Logging::Instance().set_id("my_id");
//Log a simple message (you must not use unformatted strings, or else you will get a warning)
cpm::Logging::Instance().write(3, "%s", "This is a log message");
//Usage is similar to fprintf
cpm::Logging::Instance().write(2, "The %s is %i, although that should not be the case", "answer", 42);
```

Monitoring (eProsima)

SUBJECT TO CHANGE

For eProsima Fast-DDS, an official monitoring software can be installed to check for participants in the network or to observe network statistics.

Setup

Installing the Monitor

The monitor can be installed from the eProsima FastDDS Download site: <https://www.eprosima.com/index.php/component/ars/repository/eprosima-fast-dds-monitor/eprosima-fast-dds-monitor-1-0-0>

It can also be built from Git sources: <https://github.com/eProsima/Fast-DDS-monitor>

Installation information can be found in <https://fast-dds-monitor.readthedocs.io/en/latest/index.html> for Linux and Windows.

Setting Up the Lab Software

The monitor becomes (most) useful when the eProsima statistics module is enabled. To do so, head to the place where the Fast-DDS repository can be found (should be in `cpm_lib/thirdparty/Fast-DDS`) and change the `CMakeLists.txt`: Set `FASTDDS_STATISTICS` to `ON`. Then (re)install Fast-DDS. Note: The `install.sh` script currently overwrites that setting again, so you have to install manually without it, or the Statistics Module won't be used!

After (re)building Fast-DDS, use `build_all.bash` in the software repository to rebuild the lab software as well.

Usage

Monitor

If the lab is set up properly, all you need to do is start the monitor and either:

- Enter the currently used DomainID, if no discovery server is used for eProsima or
- Register the Discovery Server (File `Initialize Discovery Server Monitor`). You need to know the GUID (can be found in `cpm_lib/src/DiscoveryServer.cpp`) the IP (127.0.0.1 or the IP of the main PC, depending on where you run your software and the discovery server) and the Port (can be found e.g. in `gdb_run.bash` for the LCC in `/lab_control_center`).

If you set these monitoring instances up properly, you should see all current DDS traffic in that part of the network, and you should be able to see statistics as well if you followed the rest of the guide properly. This allows you to inspect e.g. latency!

Lab Setup

Even though the statistics module is now installed, it also needs to be activated. Different methods are mentioned in <https://fast-dds.docs.eprosima.com/en/v2.3.2/fastdds/statistics/statistics.html>.

For our purpose, it is enough to set:

```
export FASTDDS_STATISTICS="HISTORY_LATENCY_TOPIC;ACKNACK_COUNT_TOPIC;DISCOVERY_TOPIC;PHYSICAL_DATA_TOPIC"
```

Be aware that this must be set everywhere where you need to see statistics, e.g. when you start a (simulated) vehicle or the IPS! You might need to modify some `start.bash` files by adding this line, or even dig into `Deploy.cpp` in the `LabControlCenter` to add it.

Middleware Usage

Why Do I Need To Use The Middleware?

The Middleware offers an abstraction layer for common HLC tasks and should thus be used in combination with your HLC implementation. More information can be found [here](#) (theory). Information on implementation can be found on this page.

Command Line Parameters

Parameter Usage Example

```
./middleware --vehicle_ids=1,7,8 --node_id=middleware --wait_for_start=true --dds_domain=1 --simulated_time=false
```

Important Command Line Parameters

- `node_id`: Identification string of the middleware instance in the DDS network
- `simulated_time`: True if the current setup uses simulated time, else false
- `middleware_domain`: Domain ID for communication between middleware and HLC via shared memory, default is 1
- `wait_for_start`: Whether the middleware should wait for the LCC's start signal, should be true (false only for debugging purposes)
- `vehicle_id`: Single ID of a vehicle the middleware is supposed to be responsible for
- `vehicle_ids`: IDs of the vehicles the HLC(s) that the middleware 'controls' are responsible for (usually just one ID)
- `vehicle_amount`: Set amount of vehicles instead of vehicle ID, sets IDs 1 - `vehicle_amount` (restricted to 256)
- `dds_domain`: Domain ID of the domain in which the LCC, vehicle etc. participate
- `domain_number`: Local domain ID of the middleware and the HLC
- `dds_initial_peer`: Should be explained in [Lab Control Center \(Usage\)](#)

Important ParameterServer Parameters

- `middleware_period_ms`: Length of each period in which the HLC is triggered, in milliseconds

Other Command Line parameters

- `offset_nanoseconds`: Set the offset of the communication w.r.t. ~1970 (real time) or 0 (simulated time); should usually not be changed

QOS_LOCAL_COMMUNICATION.xml.template

The communication between Middleware and HLC takes place locally and should not be shared with the other participants in the network. Thus, this DDS QoS file makes sure that messages via both of them are only exchanged locally and that other participants do not get discovered for that. .template is used as the current IP must be used for this to work, build.bash fills in the template and creates the .xml file in the build folder. This file is also relevant for the HLC scripts, which need to use the same QoS settings to communicate with the Middleware.

How Do I Use The Middleware?

The Middleware is usually started by the LCC (with the right parameters, i.e. vehicle IDs) if you [deploy it via the LCC](#). Don't forget that, after you have deployed the Middleware and after it has been initialized and communicated with the HLC script, you also need to send a start signal from within the [timer tab](#).

You can also start the Middleware manually given the parameters above (a start signal is still required though) - the most important parameters here are:

- `vehicle_ids` You need to tell the Middleware which vehicles your program is responsible for. It will then forward information about the vehicles to your HLC program, and your HLC program can send planned trajectories for these vehicles to the Middleware, which then sends these to the vehicles themselves.
- `middleware_period_ms` This parameter is set using the parameter service ([theory, usage](#)). It tells the middleware how often it should send your program new vehicle information, which you are supposed to use to start the computation of a new trajectory (and thus as timing signals).

There are some steps involved in using the Middleware after it has been started. They are mentioned below and are further elaborated in the [Matlab Tutorial](#). Even if you use another programming language, you can use that tutorial as an implementation guideline (as DDS communication should not depend on the programming language).

Communication between HLC and Middleware

Several IDL types are used for the communication between the HLC and the Middleware. In all cases, HLC and Middleware communicate using DDS over shared memory with the **DDS Domain ID 1**. Only communication between HLCs should take place on the ID that is used in the rest of the Lab. The following communication types are used in domain 1. They are:

ReadyStatus

After initialization, the Middleware waits for the initialization of the HLC. Afterwards, it tells the LCC that the HLC is ready to operate (i.e. the Middleware appears in the [Timer Tab](#), waiting for a start signal). The Middleware only continues execution (and thus only appears in the Timer Tab) if it receives a ReadyStatus by the HLC. This signal should only be send by your implementation after every possible initialization has taken place, so that, as a next step, you can start with the computation as soon as the HLC tells you to do so.

In the message, only the ID string matters, which must be of the form "hlc_" + vehicle_id, where the latter is the ID of the vehicle the HLC is responsible for. As you can see, the Middleware does not expect a single HLC script to be responsible for more than one vehicle. It can communicate with more than one script though (depending on the set vehicle IDs), in which case it waits for all HLCs to send a ReadyStatus before it continues execution.

The ReadyStatus is defined here: https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/dds_idl/ReadyStatus.idl

SystemTrigger

The LCC regularly sends timing information in the network if [simulated time](#) is used, or only start or stop signals (to start and stop the simulation) in case of [real time](#). The Middleware takes care of all timing signals for you, and also forwards timing information within the VehicleStateList, which you should use to obtain the current system time. But it is not able to stop the HLC directly in case a stop signal is received. Thus, SystemTrigger messages are forwarded to the HLC as well. Your implementation should only look for stop signals regularly, which are defined [here](#). If such a signal is received, the HLC program must be stopped.

CommonroadDDSGoalState

We use commonroad to define scenarios for the simulation. All relevant commonroad information will get passed to the HLC. Currently, this only includes information regarding the goal state. Due to the usage of TransientLocal settings and a restart of the writer for each simulation, the goal state information is always up to date and can always be received in case a commonroad scenario is in use. The structure of the datatype is similar to its Commonroad definition.

You can find out more about it [here](#).

VehicleStateList

These messages are meant as timing signals for your program, but they also include all required information about the current state of all vehicles in the system. You can find out more here:

<https://git.rwth-aachen.de/CPM/Project/Lab/software/blob/master/hlc/middleware/idl/VehicleStateList.idl>

Thus, you have to receive VehicleStateList messages, which include the current states and observations of the vehicle as well as the current time. These signals are supposed to be the start signal for the HLC, so computation should start using this data after the message was received. You thus need to:

1. Wait for the next VehicleStateList message (and look for stop signals in SystemTrigger)
2. Read the message, use the vehicle and timing information, compute a trajectory for the vehicles your program is responsible for
3. Send the trajectory to the Middleware
4. Go back to step 1

The history for this signal is set to 1, but you may still get an outdated signal here if you missed a period during your computation and read the next VehicleStateList in the middle of the next computation period. In that case, it may be better to skip that period as well and wait for the following one to start. Thus, if directly after you have sent a trajectory there is another VehicleStateList message in the reader, you should ignore that message and wait for the next one.

VehicleCommand[...]

You need to send vehicle command messages as a result of your computation - including the vehicle ID - to the Middleware, which propagates these to the vehicle. The Middleware currently supports VehicleCommandTrajectory, VehicleCommandSpeedCurvature and VehicleCommandDirect. The first of these is the message type that you are most likely to use:

https://git.rwth-aachen.de/CPM/Project/Lab/software/-/blob/master/cpm_lib/dds_idl/VehicleCommandTrajectory.idl

Matlab + RTI DDS

RTI DDS

This example is a **guideline** on how to write a Matlab script that, together with the Middleware, communicates with a vehicle. This tutorial is based on our platoon example (which is just a series of vehicles following a preset path without much logic behind that behavior), which you can find in

https://github.com/embedded-software-laboratory/cpm_lab/tree/master/high_level_controller/examples/matlab/platoon

More specifically, we will take a look at `main_vehicle_ids.m`.

As you can see in the following sections, this script uses another script, called `init_script.m`.

Init script

The **init script** is similar (in its idea) to the `cpm` library for your C++ programs: It is supposed to **take care of redundant work** for you. In this case, it sets up all relevant **DDS participants** for the communication within the network. It also loads the required **IDL files** (message types), loads **XML files** for QoS settings and sets the variable for the **stop signal**, which will be mentioned later on. It currently requires a given folder structure, which is automatically set up on the NUCs and should also be part of the software repository's structure - in an ideal case, you do not need to worry about that, as long as you import the init script from its original place (it can also be found in `~/dev/software/high_level_controller/examples/matlab/` on the NUCs).

You can find it here:

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/matlab/init_script.m

Working with RTI DDS in Matlab

Please consult [RTI DDS and Matlab](#) to get an idea of how to work with DDS in Matlab.

Your Matlab script is **not supposed to directly communicate with the vehicle** or to perform timing operations with the LCC. This is the [Middleware's](#) task, so your task is **just to properly communicate with the Middleware** (if you only want to control the vehicle).

The Middleware will wake your script up regularly to perform a computation with the currently given values, and you just need to respond in a proper way. This allows you to **focus on problem solving instead of timing and communication**. Always ensure that the middleware's period in the LCC "Parameters" tab is set according to your expectations in the HLC.

Setup

The structure of your code

Your **function head** may differ, but you **must use `varargin`** as your last parameter to pass vehicle IDs to your script. This allows you to define, for your script, which vehicle(s) it should be responsible for. Any previous parameters you define are your own 'custom' parameters and **need to be specified as additional parameters** in the Lab Control Centers' UI before starting your script (if you wish to start it using the UI).

```
function main(middleware_domain_id, varargin)
...
vehicle_ids = varargin;
```

The parameter `middleware_domain_id` only exists in the `eprosima` branch and will not be added to the RTI Branch anymore! Don't use this parameter if you use RTI DDS. Furthermore, the examples below use the RTI DDS syntax. eProsima syntax can be found [here](#).

In other words: If your function head looks like this

```
function main(some, params, middleware_domain_id, varargin)
```

then you need to pass `some, params` as additional parameters in the LCC's UI (ignore the script path in the screenshot):

Setup	Commonroad	Manual Control	Parameters	Timer	Logs
Lab mode			☐		
Diagnosis			☐		
Script Path					
../high_level_controller/examples/cpp/central_routing/build/central_routing					
Open					
Script Command Line Parameters					
some, params					
Simulated Vehicles					

Varargin covers the vehicle IDs that your script should be responsible for. The DDS Domain ID to communicate with the Middleware only via shared memory is `middleware_domain_id`, its default value is 1. To be compatible with the LCC, which expects these two parameters to be available on calling your script, you should always put them as the last two parameters in your function head as specified above.

What your HLC scripts need to include

Import the init script

You can see the sample code below. What you should do:

1. Set the Matlab domain ID to 1 in a variable of your choice. This domain is used to communicate directly with the Middleware. More information can be found [here](#).
2. Go to the right directory for the init script
3. Call the script and store the readers and writers that it sets up for you, as well as the stop signal (Which we call `trigger_stop` here).
4. Store the vehicle IDs from `varargin`. You will need them later.
5. The Middleware uses `VehicleStateList` data for timing purposes, i.e. if new data is received, you need to use it for your calculation, otherwise you should wait until you receive more data. This is further explained down below. A waitset is set to wait upon taking data from the reader if no data is yet available. The timeout is set to 10 seconds in this example, but you can also set a higher timeout.

```
function main_vehicle_ids(middleware_domain_id, varargin) % middleware_domain_id only for eProsim + Matlab
    % Set the matlab domain ID for communicating with the middleware (which is always 1)
    matlabDomainID = 1;

    % Clear command window, store path of current script, go to that path
    clc
    script_directoy = fileparts([mfilename('fullpath') '.m']);
    cd(script_directoy)

    % Set the path of the init script relative to the path of your script (or absolute), assert that it exists,
    add it to the path so that it can be found and then call it
    % Then go back to the script directory
    init_script_path = fullfile('..', '/init_script.m');
    assert(isfile(init_script_path), 'Missing file "%s".', init_script_path);
    addpath(fileparts(init_script_path));
    [matlabParticipant, stateReader, trajectoryWriter, systemTriggerReader, readyStatusWriter, trigger_stop] =
    init_script(matlabDomainID);
    cd(script_directoy)

    % Remember the set vehicle IDs in a new variable
    vehicle_ids = varargin;

    %% Use a waitset for the reader of vehicle states, which is later used to indicate if a new computation
    should be started
    % The waitset makes sure that take() or read() from the reader's storage does not return if no new data
    is available (until the timeout is reached, here 10 seconds
    % so that one can still check for a stop signal regularly)
    stateReader.WaitSet = true;
```

```

stateReader.WaitSetTimeout = 10;

%% Do not display figures
set(0, 'DefaultFigureVisible', 'off');

```

Tell the Middleware that your script is ready to operate

You must send a ReadySignal message **after initialization** - only the **ID string matters**, which must be of the form "hlc_" + vehicle_id, where the latter is the ID of the vehicle the HLC and thus your script is responsible for:

```

% (Assuming that vehicle_ids contains the IDs your script is responsible for)
% Send first ready signal to the Middleware to indicate that your program is ready to operate
% The signal needs to be sent for all assigned vehicle ids separately (usually, one script on one NUC only
manages one of the IDs)
% Also works for simulated time - period etc are set in Middleware, so you can ignore the timestamp field
for i = 1 : length(vehicle_ids)
    % Create a new msg of type ReadyStatus (idl)
    ready_msg = ReadyStatus;

    % Set the values of ready_msg
    ready_msg.source_id = strcat('hlc_', num2str(vehicle_ids{i}));
    ready_stamp = TimeStamp;
    ready_stamp.nanoseconds = uint64(0);
    ready_msg.next_start_stamp = ready_stamp;

    % Send the ready msg
    readyStatusWriter.write(ready_msg);
end

```

You use the ReadyStatus type here, which was defined in one of the IDL files that was imported for you using the init script. You can see the values of this type in the IDL file:

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/cpm_lib/dds_idl/ReadyStatus.idl

This message **tells the Middleware that your script is now ready to operate** and to receive its data. Thus, at this point, your reader for vehicle states (here stateReader) should be initialized so that it can receive the data.

Consider the timing signals

This is very important. There are two signals you must consider: Start and stop signals. You have to **check regularly for the stop signal**, and **once for the start signal**:

- Start signal: This indicates that your script is supposed to calculate a new e.g. trajectory for the vehicles it is responsible for. Start with calculation immediately and send the results back as soon as possible. The Middleware handles the rest.
- Stop signal: trigger_stop = uint64(18446744073709551615) was already set by the init script. It is the highest number representable with a uint64_t type. This number indicates that the simulation was stopped, and thus your script should immediately terminate.
- All other numbers are irrelevant. From here on, **new messages received by the reader of VehicleStateList (stateReader), are interpreted as start signals for less redundancy.**

Waiting for the initial start signal may look like this:

```

% Wait for start signal / stop if a stop signal was received
got_stop = false;
got_start = false;
while(true)
    % Read the newest system trigger message, if one exists
    trigger = SystemTrigger;
    sampleCount = 0;
    [trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);

    % Go through all received messages since the last check, if they exist (sampleCount gives the number of
read messages)
    % Any received message would indicate "start", but you have to check if a stop signal was received as
well
    while sampleCount > 0
        % Check for stop signal
        if trigger.next_start().nanoseconds() == trigger_stop
            got_stop = true;
        elseif trigger.next_start().nanoseconds() >= 0

```

```

        got_start = true;
    end

    % Read the next sample and continue until sampleCount is zero again (all samples have been read
then)
    [trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);
    end

    % Stop the loop if a signal was received, then proceed with handling a stop signal (stop the program)
or a start signal (continue)
    if got_stop | got_start
        break;
    end
end
end

```

You need to repeat this procedure to check for the stop signal every time before you wait for the next start indicator sent by the Middleware (which is indicated by new samples in VehicleStateList). The loop, that includes calculating new data as well, may look like this:

```

% Again: Go through all samples and stop the program if you received a stop signal
% break_while is the name used in the sample program (as it breaks the enclosing while-loop of the whole
program), but you could also name it e.g. stop_received
trigger = SystemTrigger;
sampleCount = 0;
[trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);

break_while = false;

while sampleCount > 0
    current_time = trigger.next_start().nanoseconds();
    if current_time == trigger_stop
        break_while = true;
    end

    [trigger, status, sampleCount, sampleInfo] = systemTriggerReader.take(trigger);
end

if break_while
    ...

```

Important

The Middleware might already have been stopped by a stop signal while you are waiting for the next start signal in form of a VehicleStateList. It is thus important to set a timeout for stateReader that is not too high, so that you can check for stop signals in between waiting for a new start signal. On the other hand, you want to react as fast as possible to a new start signal, so the waiting time should also not be too low - only consider stop signals if you can be sure that you should have received a new start signal by that time, i.e. after 10 seconds.

Receive information about your vehicle

Information about your vehicle are contained in the VehicleStateList messages, which include the **current states and observations of all vehicles** as well as the **current time**. This signal is supposed to be the start signal for the HLC, so computation should start using this data directly after the message was received.

These signals, which, in this example, you can obtain using the stateReader, contain two fields which are vital to the correct usage of your script.

1. The messages contain current information about the **whereabouts and settings of your vehicle as well as any other vehicle** in the simulation. Use this data **for planning**.
2. The messages contain **timing information**. Use these when you send commands to your vehicle - they contain the current time you set for the messages.

This signal has a third purpose as well. It shows the script when to start the computation, similar to a timing signal. The desired computation cycle is the following:

- **Wait** for a new message
- After receiving a message, **start your computation**
- Then **send** the new **command(s)** to the vehicle
- Remove messages that were received during computation, as they are old and potentially unusable, and **wait** for the next message
- ... (repeat)

The history for this signal is set to 1, but you may still get an outdated signal here **if you missed a period during your computation** and read the next VehicleStateList in the middle of that next period. In that case, it may be **better to skip that next period** as well and wait for the following one to start.

```

...
% Read the current vehicle information from the according reader, if such data is available
% Due to the set waitset, you wait for up to 10 seconds at take(...) if no data is available
sample = VehicleStateList;
status = 0;
stateSampleCount = 0;
sampleInfo = DDS.SampleInfo;
[sample, status, stateSampleCount, sampleInfo] = stateReader.take(sample);

% Check if any new message was received, then proceed with handling the data, computing your solution and
finally sending back a command to the vehicle(s) your script controls
if stateSampleCount > 0
    ...

```

Send commands to your vehicle

You need to send vehicle command messages as a result of your computation including the vehicle ID to the Middleware, which propagates these to the vehicle. The implementation of the computation of e.g. the vehicle's trajectory is not explained here and depends on your task. You are only given an example of how to send a simple trajectory here.

We are now taking a look at In the script which we were looking at `Leader.m`, which, as you can see in the first code sample below, is called within the platoon script. We use it to compute the next trajectory segment:

Important

All the scripts that you use internally **should be in the same folder**, else they will not work when they are being deployed remotely.

```

% Call the programs which calculate the trajectories of all HLCs, then send the results back to the vehicles
if (size(vehicle_ids) > 0)
    for i = 1 : length(vehicle_ids)
        msg_leader = leader(vehicle_ids{i}, sample.t_now);

        % Send resulting trajectory to the vehicle
        trajectoryWriter.write(msg_leader);
    end
end

```

In the following, you see an example of how to set up a trajectory message, that can then be sent using the `trajectoryWriter`:

```

%Create msg of type trajectory
trajectory = VehicleCommandTrajectory;

% Set the vehicle ID
trajectory.vehicle_id = uint8(vehicle_id);

% In this example, we only send one trajectory point (which is not sufficient for interpolation)
trajectory_points = [];
point1 = TrajectoryPoint;

% This trajectory point should be considered in the future, so add some nanoseconds to the current time, then
set the time stamp for the trajectory point
% t_eval here is the time that we got from the middleware in the VehicleStateList message - you are not
supposed to use your own clock / timer (else you would have trouble working with simulated time)
time = t_eval + 4000000000;
stamp = TimeStamp;
stamp.nanoseconds = uint64(time);
point1.t = stamp;

% Set other trajectory values - position and velocity in 2D coordinates
point1.px = trajectory_point(1);
point1.py = trajectory_point(2);
point1.vx = trajectory_point(3);
point1.vy = trajectory_point(4);

% Append to the list of all trajectory points
trajectory_points = [trajectory_points [point1]];
trajectory.trajectory_points = trajectory_points;

% Send the trajectory to the vehicle
trajectoryWriter.write(trajectory);

```

Further Information

Bash script / Deploy Using the LCC

If you want to start your script using a bash script, you can do it like this:

```
/opt/MATLAB/R2019a/bin/matlab -logfile matlab.log -sd $script_dir -batch "$script_name(1, ${vehicle_id})"
```

Usually, you would want to [start it using the LCC instead](#), which takes care of these things for you.

Deploying with Matlab GUI

See [here](#): If, within the [setup tab](#), you do not select any script but leave the script field empty, only the Middleware gets started. You can then start your own script from within the Matlab GUI, it should detect the Middleware (if you set the correct number of vehicles, the Middleware will, if deployed locally, wait for HLC messages for all vehicles that are currently online).

How to Actually Start Your Script

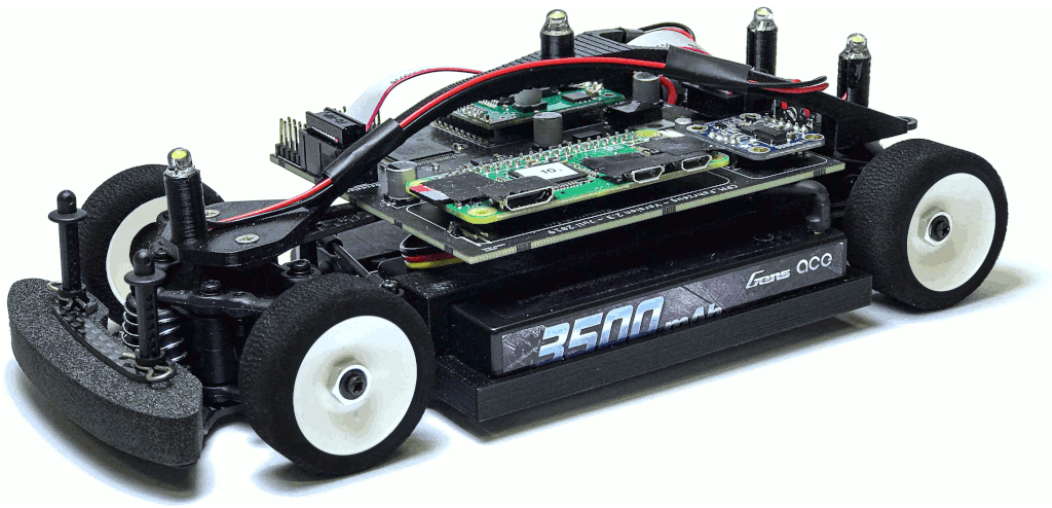
As mentioned before, the Middleware takes care of timing. As soon as your script has told it that it is ready to operate, the Middleware should appear in the [Timer Tab](#) in the LCC. Then, you can start the simulation by sending a start signal (press on the start button). Of course, you need to [deploy / start the simulation](#) to start the Middleware before you start the timer (and, if you do not use the Matlab GUI, your selected Matlab script as well).

Vehicle

- [Vehicle Overview](#)
- [Equations](#)
- [LiPo Charging](#)

Vehicle Overview

The μ Cars are connected and automated model-scale vehicles (with a scale of 1:18) with Ackermann steering geometry developed for experiments in research and education. The vehicle is open-source, moderately costed, and highly flexible, which allows for many applications. It obtains its pose (position and orientation) via WLAN from the IPS and is equipped with an inertial measurement unit and an odometer. The Car is based on the [XRAY M18 PRO LiPo Kit \(PN 380003\)](#). It is combined with other off-the-shelf components and custom electronics.

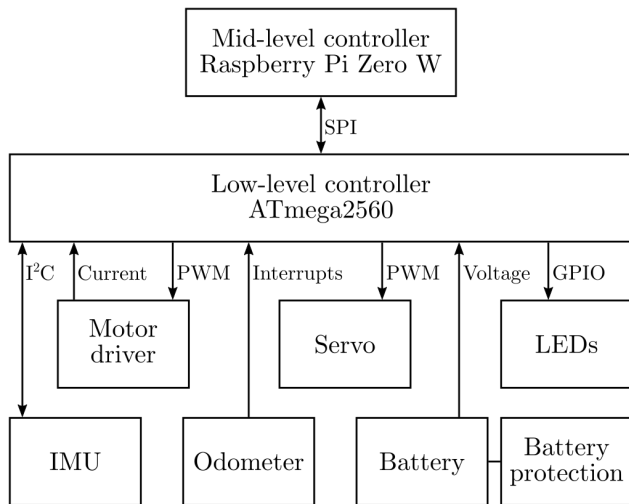


Specifications

Physical representation	Value
Vehicle length	220 mm
Vehicle width	107 mm
Vehicle height	70 mm
Wheelbase	150 mm
Min. radius of curvature for steering	0.3 m
Max. steering angle	35 deg
Max. speed (limited by software)	2.3 m/s
Weight	500 g
Weight distribution front:back	0.48:0.52
Weight distribution left:right	0.56:0.44

Hardware

Architecture



Components

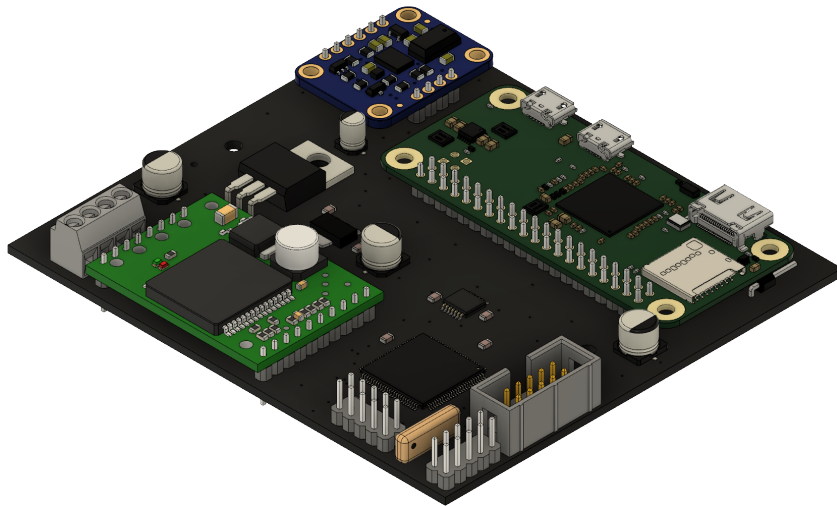
Off-the-shelf components:

- Gens Ace 3500 mAh 7.4V 2S LiPo Battery Pack
- Hitec Servo D89MW
- Motor Johnson NF113LG-011
- Pololu VNH5019 Motor Driver Carrier
- Adafruit BNO055 Absolute Orientation Sensor
- Raspberry Pi Zero W
- Microchip AVR ATmega2560

Custom electronics:

- For connecting COTS components
- Battery protection
- Magnetic odometer
- Position LEDs
- Switch, fuse, charging connector

All components are connected via a printed circuit board.



References

[Vehicle Assembly](#)

[PCB Assembly](#)

Equations

$\text{lateral_acceleration}[\text{m/s}^2] == \text{curvature}[\text{1/m}] * (\text{speed}[\text{m/s}])^2$

$\text{lateral_acceleration}[\text{m/s}^2] == \text{yaw_rate}[\text{rad/s}] * \text{speed}[\text{m/s}]$

$\text{turn_radius}[\text{m}] * \text{curvature}[\text{1/m}] == 1$

$\text{direction_x}[1] == \cos(\text{yaw}[\text{rad}])$

$\text{direction_y}[1] == \sin(\text{yaw}[\text{rad}])$

$\text{yaw}[\text{rad}] == \text{atan2}(\text{direction_y}[1], \text{direction_x}[1])$

$2 * (\text{braking_deceleration}[\text{m/s}^2]) * (\text{braking_distance}[\text{m}]) == (\text{initial_speed}[\text{m/s}])^2$

LiPo Charging

Never leave LiPo Batteries unattended during charging due to fire hazard.

The following instructions should be followed to properly charge the CPM Lab Vehicle LiPo Batteries using a VOLTcraft V-CHARGE 50 charger. If unsure, please consult following people for help:

- Maximilian Kloock
- Patrick Scheffe

Instructions can also be found in git:

- https://git.rwth-aachen.de/CPM/Project/Lab/docs/blob/master/Vehicle/Version2/production_manuals/lipo_charging.pptx

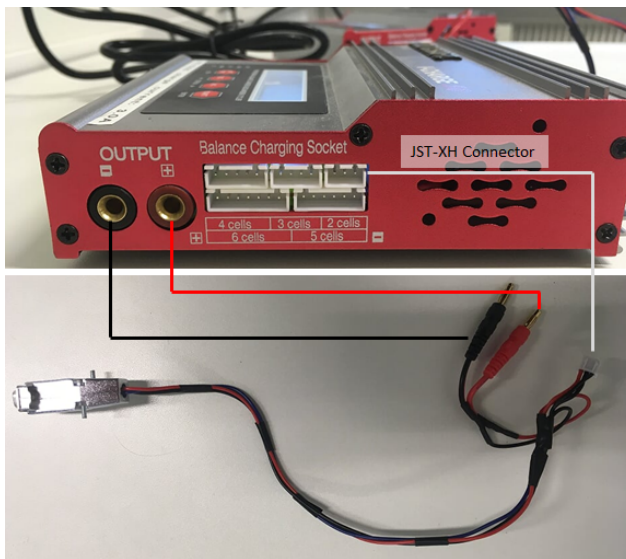
General Info



1. Charge process can be terminated at any point. LiPo Batteries should never be left unattended while charging. Charge should be terminated before leaving CPM Lab.
2. LiPo Battery Health and Lifespan is kept highest if LiPo Batteries are regularly charged. Do not hesitate to charge vehicles before battery protection circuit is triggered. The battery protection circuit is triggered at 6.0-6.4V. If possible, charge batteries regularly to maintain a charge between 7.4-8.0V.

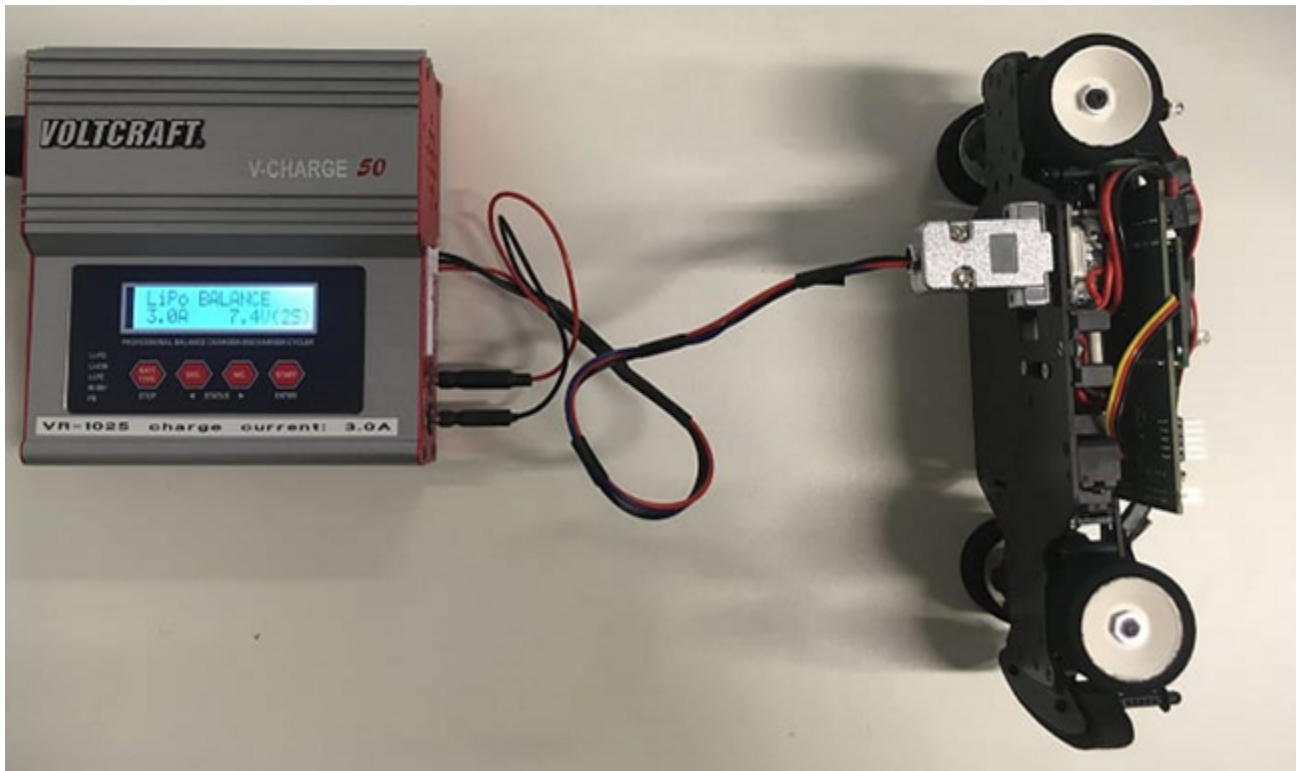
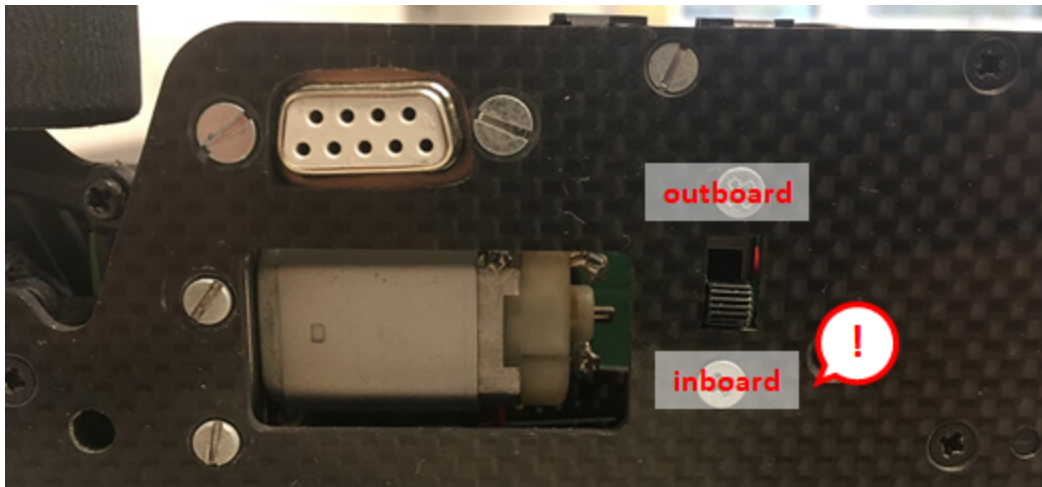
1. Charger Setup

Plug charger cable into correct ports.



2. Connect Vehicle

Turn off the vehicle (switch should be inboard toward center of vehicle) and connect DSUB connector.



Turning off the vehicle is important so that charge termination is successful. LiPo chargers work by charging in two modes, first constant current (CC) then constant voltage (CV). During CV, the current is monitored until it drops under a certain threshold at which point the charger terminates the charging process. If the vehicle is left on, this threshold current may not be reached and the charger will charge indefinitely.

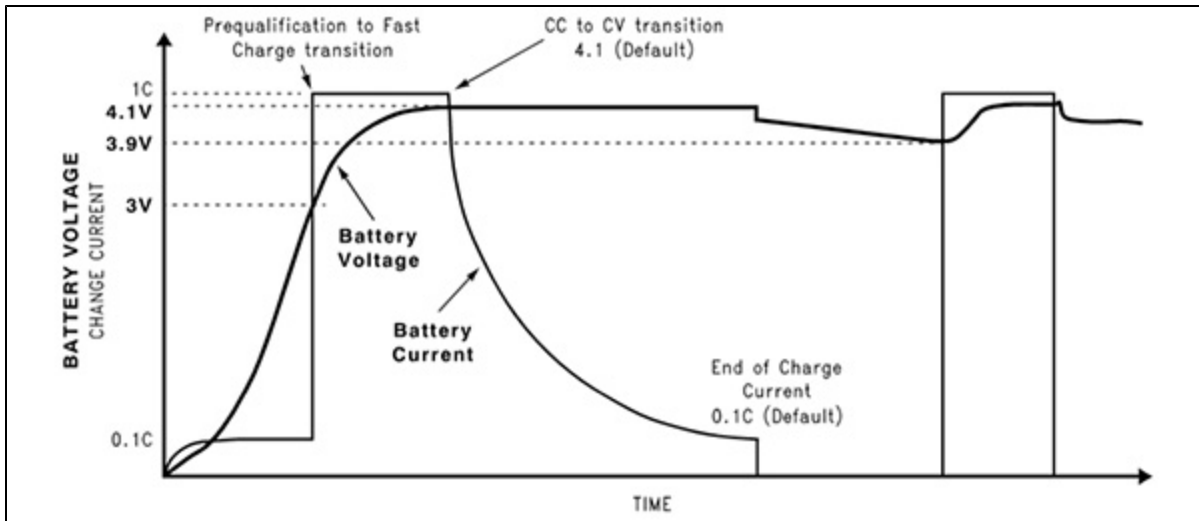


Image Courtesy: <https://www.digikey.com/en/articles/techzone/2016/sep/a-designer-guide-fast-lithium-ion-battery-charging>

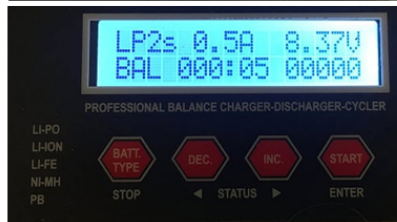
3. Charge Vehicle

It is okay to stop LiPo charging at any time if need be e.g. to leave room. To stop charging, press STOP button.

Follow slide to properly begin charging vehicle. Following settings should be observed:

Setting	LiPo Balance
Charge Current	3A
Voltage	2.7V(2S)

WARNING: NEVER LEAVE LIPO UNATTENDED WHILE CHARGING DUE TO FIRE RISK



Never leave LiPo Batteries unattended during charging due to fire hazard.

Software structure

- High Level Controller
- Middleware Structure
- Mid Level Controller
- Low Level Controller

High Level Controller

The high level controller (HLC) is based on the NUCs. Its purpose is to plan trajectories for one vehicle. In networked trajectory planning, multiple HLCs are involved. Each HLC is responsible for the trajectory of one vehicle.

The HLC is not part of the infrastructure software, but has to use the interfaces of the lab. We recommend to use [MATLAB](#), but you can use any language. We describe example HLCs in the [example](#) section.

Middleware Structure

Middleware Workflow

Shared or common functionality for the communication via DDS is bundled in the *cpm library* (*cpm_lib*). This library is written in C++ and is used by the vehicle, the LCC and the Middleware. The HLC, on the other hand, is supposed to be more or less *independent from the implementation language* of the library. Programmers of the HLC programs should not be troubled with its implementation details. Thus, it was decided to create a Middleware which performs most of the communication tasks that otherwise would have to be implemented in some form in the HLC, which might not be able to use a C++ library.

The Middleware *coordinates the communication* with the DDS domain of the vehicle, LCC and further participants with the HLC. It is responsible for *sending timed pre-aggregated vehicle information* to the HLC. These can be interpreted as a start signal for starting the computation using this data. It *sends the commands generated by the HLC* as a reaction to this start signal to the vehicles. Any *timing* task is performed by the Middleware, so period and offset for the interaction with the HLC are set here (and can be set when starting the application using command line parameters). The HLC is *informed about the current time*, in case that is necessary for its own computations.

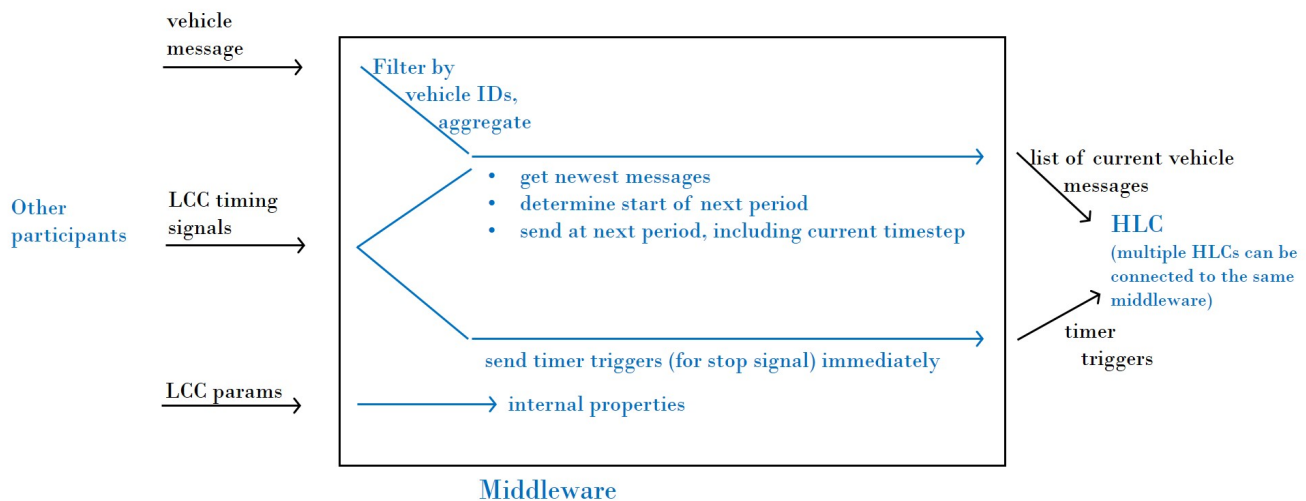


Figure: Communication from other participants to the HLC filtered by the Middleware

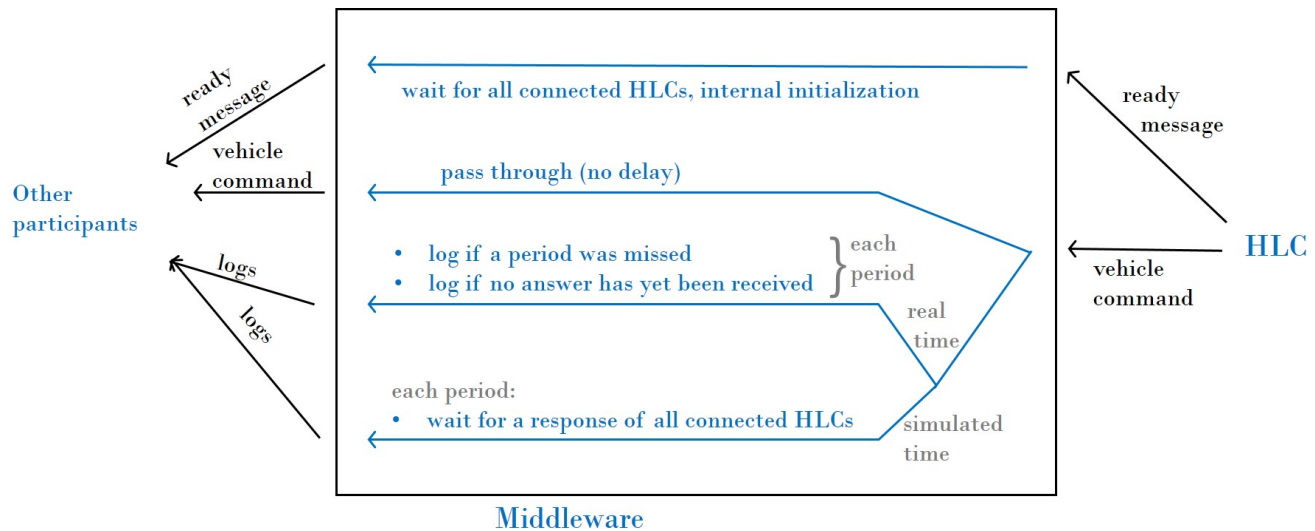


Figure: Communication from the HLC to other participants

The middleware also manages further internal timing according to the current simulated and real time, propagates the LLC's start and stop signals and waits for the HLC to finish its computation in the simulated case. It is also responsible for logging communication or timing errors.

The HLC only needs to integrate and configure a reader and a writer to listen to the Middleware and send commands. Thus, as long as the programming language is supported by RTI DDS, the programmer does not need to know the implementation details of the communication.

Functionality

- Logging
- Waiting for the HLC to join the domain & send a ready message
- Periodically calling the HLC to request commands for the vehicle
- Forwarding vehicle state information to the HLC periodically (always the newest information is sent) + timing information s.t. the HLC does not need to care about getting the right timestamps itself
- Forwarding commands sent by the HLC to the vehicle
- Forwarding stop signals to the HLC
- Receiving (some) parameters from the parameter server, e.g. the period with which to call the HLC script

Mid Level Controller

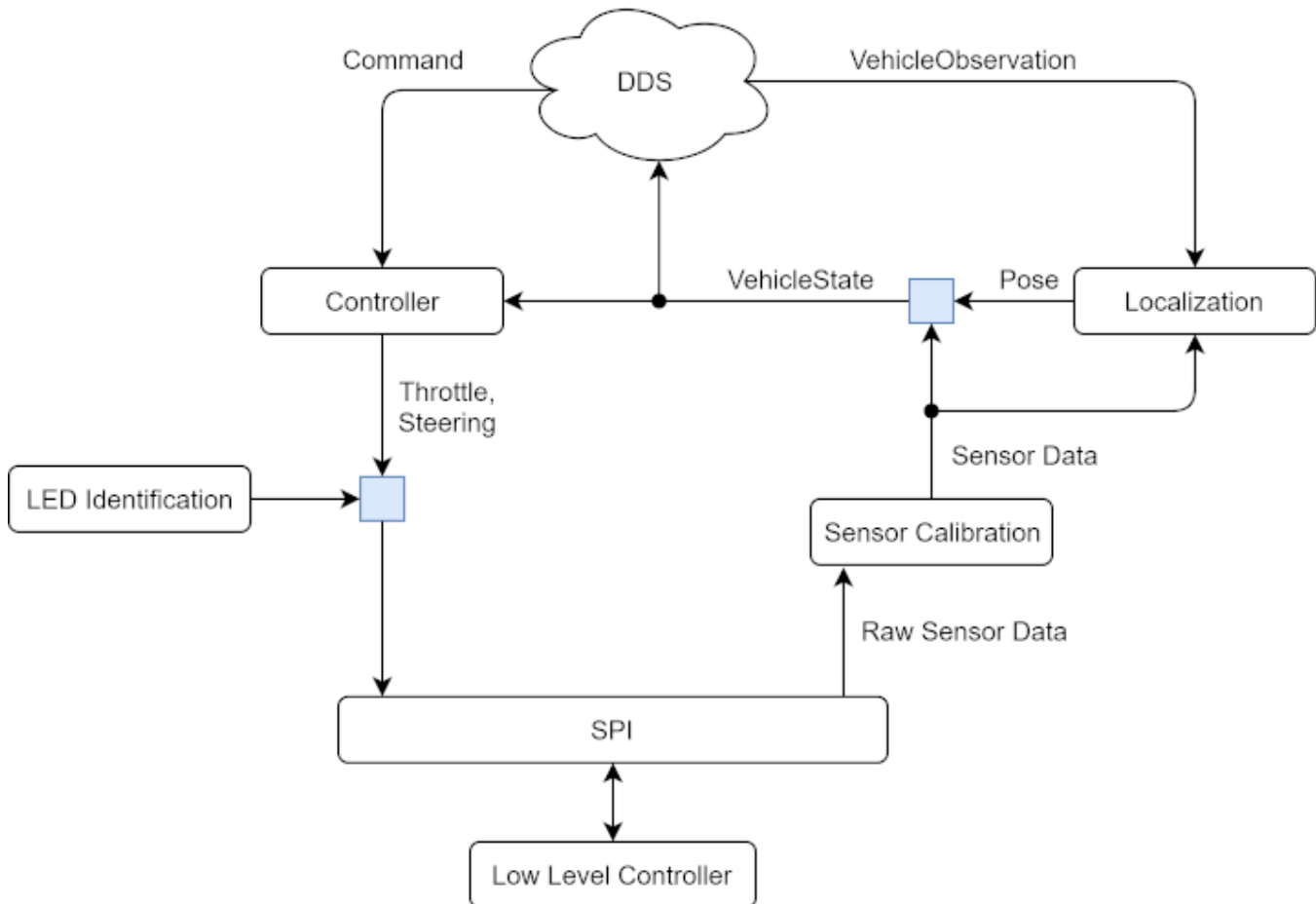
The Mid Level Controller is based on the Raspberry Pi Zero W. It is part of the vehicle on-board electronics.

Its purposes are:

- Provide a wireless network connection for the vehicle.
- Provide a synchronized clock on the vehicle, via the Linux NTP implementation.
- Provide a control system for accurate trajectory following of the vehicle. The control system runs locally on the vehicle to eliminate the network latency from the control loop.
- Send vehicle state information and receive trajectory commands via the network.

The setup and software build for the Raspberry Pi is explained under [Raspberry Pi Setup](#).

The following data-flow diagram shows the MLC architecture.



Trajectory Controller

A [model predictive controller](#) runs on-board on the Raspberry Pi. It can steer the vehicle to follow a given trajectory.

To see how to use it to control a vehicle, go to [Vehicle Commands](#).

To see how it works, go to https://github.com/embedded-software-laboratory/cpm-lab/blob/master/matlab_scripts/vehicle_dynamics_identification_and_mpc/documentation/main.pdf

Localization

The localization combines the IPS vehicle observation with local vehicle sensor data to determine the vehicle pose. If the IPS vehicle observation is delayed or temporarily unavailable, the localization can continue to give accurate poses through [dead reckoning](#). However, the dead reckoning will fail when driving aggressively, i.e. with high wheel slip.

Sensor Calibration

The Low Level Controller provides its data as low-resolution integers. The LLC units are chosen to match the required value range and quantization error. The sensor calibration converts the LLC data to floating point values with SI units.

LED Identification

See [Indoor Positioning System](#).

SPI

The MLC / Raspberry Pi is the SPI master. It exchanges data with the LLC / ATmega2560 slave. The communication is synchronous, i.e. the LLC waits for the MLC to initiate the exchange. The data exchange repeats at 50 Hz. SPI is full duplex, the input and output data are exchanged simultaneously.

In each time step (20 ms), two data packets are transmitted between the MLC and LLC, one in each direction. The data integrity is checked on both sides using the [cyclic redundancy check](#). The LLC will acknowledge receiving a correct message from the MLC through a status bit. The MLC will repeat the transmission within a time step, until a correct message from the LLC is received and the LCC's acknowledge bit is set. Thus, every transmission will be made at least twice. The MLC signals the end of the (re)transmission phase to the LLC through a separate "chip select" / "slave select" line.

https://github.com/embedded-software-laboratory/cpm-lab/blob/master/vehicle_raspberry_firmware/src/spi.c

https://github.com/embedded-software-laboratory/cpm-lab/blob/master/vehicle_atmega2560_firmware/vehicle_atmega2560_firmware/spi.c

https://github.com/embedded-software-laboratory/cpm-lab/blob/master/vehicle_atmega2560_firmware/vehicle_atmega2560_firmware/spi_packets.h

Simulation Mode

The MLC contains a simulation of the vehicle, which replaces the LLC and SPI. The switch to the simulation mode is made at compile time. The MLC build script creates multiple variants of the MLC software, for simulation and for normal operation.

Static ARP table

In some cases [ARP](#) is unsuccessful and this leads to high latencies. To mitigate this a static ARP table is provided for the vehicles at software /mid_level_controller/package/arp_table. See also [Vehicle Setup](#).

Low Level Controller

The Low Level Controller is based on the ATmega2560 microcontroller. Its sole purpose is to act as the interface between the physical and the digital world.

Communication (SPI Slave)

The LLC / ATmega2560 acts as a SPI slave to the MLC / Raspberry Pi. See [Mid Level Controller](#), [SPI](#) for details.

Inputs / Sensors

ADC

The ADC measures the battery voltage and the motor current. These are only used for monitoring, not control.

IMU

The yaw rate and acceleration vector are read from the IMU via I2C. The yaw rate is summed, to give an estimate of the yaw angle.

Odometer

The odometer has 3 digital outputs. Each signal change (rising or falling) is handled by an interrupt. The signal change indicates a motor rotation of approx. 60 degrees. The rotation count and motor speed are derived from those signals.

Outputs / Controls

Servo

The servo is controlled through a standard 50 Hz PWM signal, where the signal is encoded in the ON-time, from 1000 us to 2000 us.

Motor

The motor is controlled by a PWM H-bridge at 20 kHz.

LEDs

The four LEDs are individually controlled through GPIO pins.

Watchdog

Should the communication with the MLC fail repeatedly, the LLC will enter a safe mode. The motor is commanded to brake. The safe mode is indicated by flashing all LEDs.

HLC Examples

- [Basic Circle Example](#)
- [Central Routing Example](#)

Basic Circle Example

You find the source code for our examples in the *software*-folder:

C++

https://github.com/embedded-software-laboratory/cpm_lab/tree/master/high_level_controller/examples/cpp/basic_circle

MATLAB

https://github.com/embedded-software-laboratory/cpm_lab/tree/master/high_level_controller/examples/matlab/basic_circle

Goal

One single vehicle will drive in a basic circle with a radius of 1m. The center of the circle will be in the center of the map, but the circle will not be connected to the map. The circle shall drive continuously.

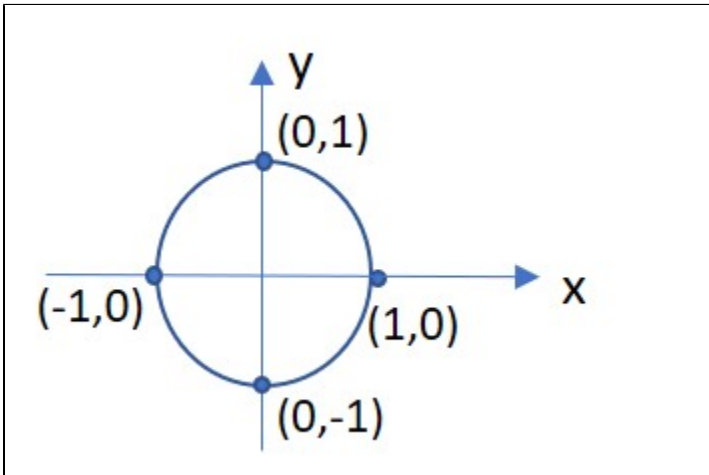
Initialisation

Define the ID of the vehicle you want to drive. Read the ID in from the LCC using the function `cmd_parameter_ints()` from our cpm library.

Initialize a DDS [Writer](#) for the RTI DDS Service to ensure that position and speed will be sent to the simulation and the vehicle. Write on the topic `VehicleCommandTrajectory` with datatype `VehicleCommandTrajectory`. The topic's name is the name of the datatype with a lowercase letter.

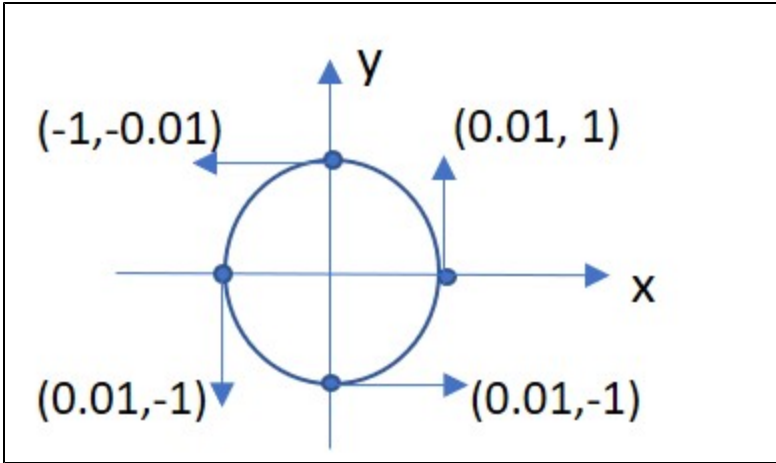
Giving direction and speed

Define four points of your circle on a coordinate system.



Define the speed vector of the vehicle for each point for the entire trajectory. When you define v_x and v_y remember that $v_{ges} = \sqrt{v_x^2 + v_y^2}$ (as it is a vector). You use this vector to define the direction for your vehicle, so where it has to steer next. Always assign at least a small value to both vectors to ensure the correct direction. Remember that you are defining a physical system, so avoid sharp edges.

Draw the vectors on each point of your trajectory:



Now implement your points and speed values using the predefined vectors `trajectory_px` and `trajectory_py` for your points of the trajectory and `trajectory_vx` and `trajectory_vy` for the speed at each point. The speed is given in m/s. Set the center of the circle to the center of the map at $x = 2$, $y = 2$ m.

Segment duration

For the segment duration keep in mind that speed, time and waylength have to fit. Verify your speed/way/duration with $v_{ges} = s_{segment} / t_{segmentduration}$. The segment duration shows the length of a timestep. Transforming it into a timestamp you need to add up all segment durations until the current timestep to the start time: $(\text{mathjax-inline}(t_i = t_0 + \sum_{n=1}^i \{n = i-1\})\text{mathjax-inline})$

Sending the trajectory information to the vehicle

All data are sent to the vehicle using the writer-function from the beginning in the form of

```
{point_x_position;
point_y_position;
velocity_x_direction;
velocity_y_direction;
timestamp;
}
```

where each value is a scalar.

Run the example

1. Compile your code.
2. [Select](#) the resulting executable as script the LCC.
3. Select vehicle 4 ([simulated mode](#)).
4. [Run](#) the HLC.
5. During the experiment, you will see a trail which builds and erases from point to point. This is your [trajectory](#). Now move the simulated vehicle close to the trajectory by [dragging it to a close point](#) to the circle trajectory. A second trail will be drawn which is the trajectory for your vehicle to get the circle trajectory.
6. Once the vehicle is on the circle trajectory it will drive continuously in endless loop.

Matlab specific information

First steps - what the HLC scripts need to include

- Send a ReadySignal message after initialization - only the ID string matters, which must be of the form "hlc_" + `vehicle_id`, where the latter is the ID of the vehicle the HLC is responsible for
- Wait for the start signal sent by the LCC and propagated to the HLC by the Middleware to start the experiment
- Receive VehicleStateList messages, which include the current states and observations of the vehicle as well as the current time. The history for this signal is set to 1, but you may still get an outdated signal here if you missed a period during your computation and read the next VehicleStateList in the middle of that next period. In that case, it may be better to skip that period as well and wait for the following one to start.
- Send vehicle command messages as a result of the communication including the vehicle ID to the Middleware, which propagates these to the vehicle
- React to stop signals sent by the LCC and propagated to the HLC by the Middleware

- **Your function head** may differ, but you **must use varargin** as your last parameter to pass vehicle IDs to your script. This allows you to define, for your script, which vehicle(s) it should be responsible for. Any previous parameters you define are your own 'custom' parameters and **need to be specified as additional parameters** in the Lab Control Centers' UI before starting your script (if you wish to start it using the UI).
- **For eProsima:** You **must use matlab_domain_id** as the parameter before varargin to determine on which domain ID the HLC should communicate with the middleware.

Central Routing Example

You find the source code for our examples in the *software*-folder:

C++

https://github.com/embedded-software-laboratory/cpm_lab/tree/master/high_level_controller/examples/cpp/central_routing

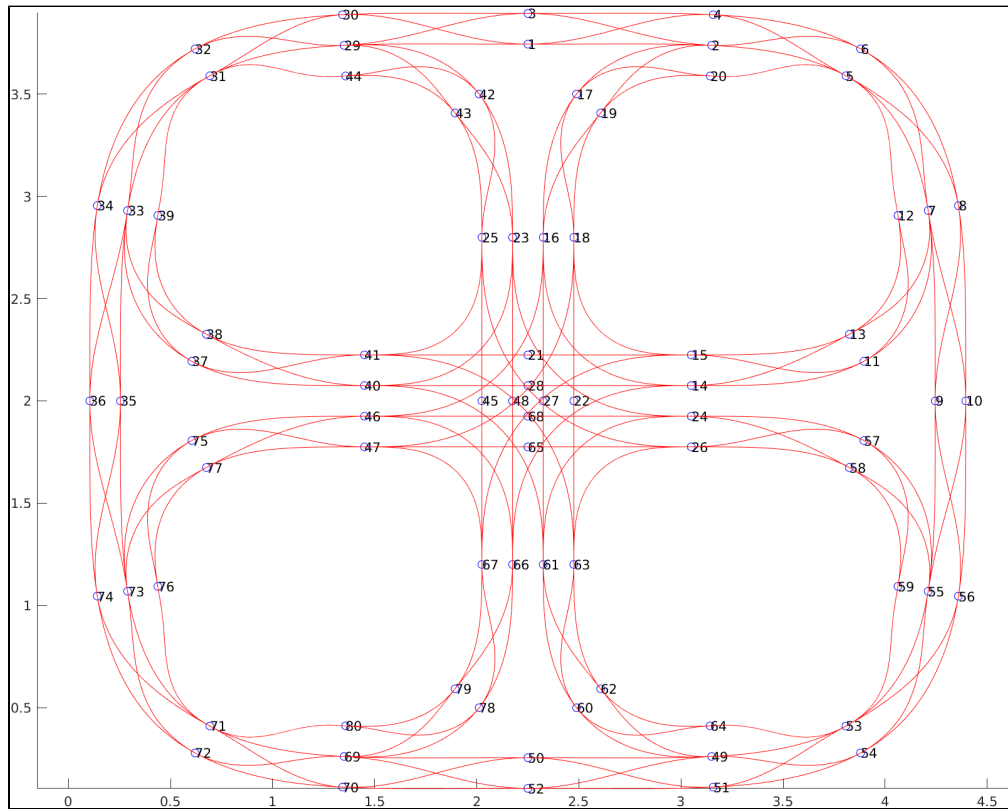
Goal

A variable number of vehicles drive randomly on the map with no collisions:

Your browser does not support the HTML5 video element

Map representation

The program is based on the routing graph for the map layout. The original lane graph is defined as followed:



Each node represents the index in the lane graph to receive the information you need for your [trajectory](#):

```
struct LaneGraph{  
    vector node_x;  
    vector node_y;  
    vector nodes_cos;  
    vector nodes_sin;  
    vector edges_start_index;  
    vector edges_end_index;  
    vector edges_x;  
    vector edges_y;  
    vector edges_sin;  
    vector edges_cos;  
}
```

The graph edges correspond to center lines, along which a vehicle may drive. Parallel lanes are not modeled, instead extra edges are added for lane changes in discrete locations. All values from your lane graph are already included in the struct which you find in:

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/cpp/central_routing/include/lane_graph_full/lane_graph.hpp

You can modify this lane graph by using the programs written in Matlab and can be found in:

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/tools/map_print/map_print2/lane_graph.m

Tools include: adding edges, deleting edges, deleting points and selecting points.

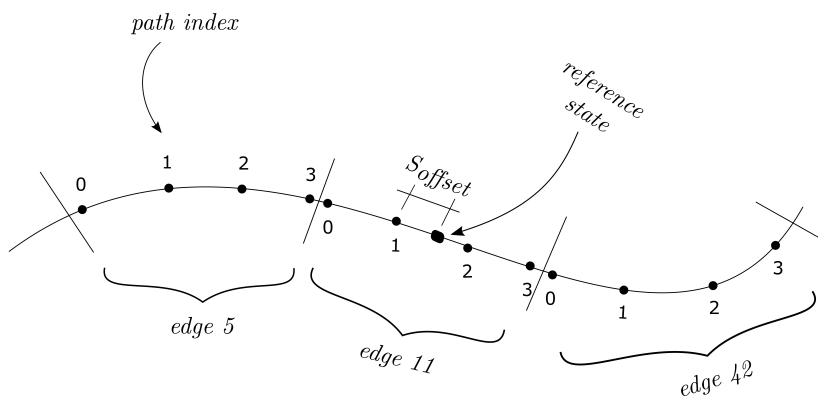
Class "LaneGraphTools"

In this class implement the following requirements:

- The constructor calculates the length of each waysegment between two neighboured points.
- The constructor defines a vector called "jobs" where all jobs are listed with the information, if the minimal distance of 0.01 between two vehicles is undergone (called collision) or not. All possible collisions, between all pairs of path nodes are checked. The result (boolean: collision yes/no?) is cached. The collision decision is based on the rectangle-rectangle distance between the vehicles.
- **map_match_pose()**: A function which defines for each path if it can be matched to an edge on the map
- **find_subsequent_edges()**: A function which finds the subsequent edges of a trajectory

The reference state of a vehicle in the graph is stored as the triple (edge_index (called current_edge_index in the sourcecode), path_index (called current_edge_path_index in the sourcecode), s_offset [in meters](called delta_s_path_node_offset in the sourcecode))

The following image shows a reference state that would be stored as (edge_index = 11, path_index = 1, s_offset = 0.01m).



move_along_route(): A function which helps you to move along the route. It moves the reference state along the given route by a given distance and takes into account the remaining distance on the current edge to next node and the changing to the next edge if necessary. For example, assume that the distance between path nodes in the above image is 0.02 m (except for the end nodes 0/3, which are in the same location). We want to move the reference state by 0.055 m.

Inputs: route = (11, 42), state = (11, 1, (0.01+0.055))

The state input is then modified to (42, 1, 0.015).

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/cpp/central_routing/src/lane_graph_tools.cpp

Class "VehicleTrajectoryPlanningState"

In this class implement the following requirements:

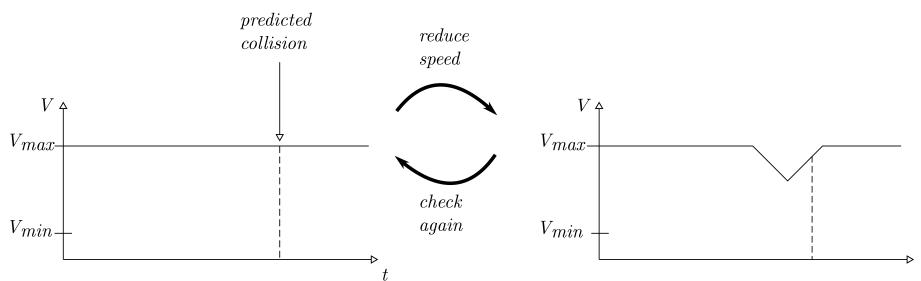
- Create a vector that contains the **speed profile** of each vehicle. The minimum of the maximum speed of the vehicle and the speed of the previous timestep is to be chosen and written in the vector.

How the vehicle moves along the path over time is defined by the speed profile. The speed profile is the function of the vehicle speed over time. When the speed profile is constructed and modified, it is done such that the given maximum speed, minimum speed and maximum acceleration are never exceeded. When both the speed profile and the path are given, they can be combined to form the trajectory. The traveled distance (arc length) is calculated by integrating (i.e. summing) the speed profile.

- **extend_random_route()**: a function that randomly adds new edges to the current trajectory for each vehicle. For each vehicle a random route is selected. This route is dynamically extended as time progresses. The route is represented as a list of edge indices. Since every edge is part of a cycle, the route can be extended indefinitely.

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/cpp/central_routing/src/geometry.hpp

- **apply_timestep()**: a function that can apply the movement of the vehicle for each timestep to the map. It will buffer the transition from the planning phase to the buffer phase. Use the function **move_along_route()** of your LaneGraphTools. Get new random route description from **extend_random_route()**.
- **get_trajectory_points()**: a function to get trajectory points for each vehicle. It contains the time elapsed in nanoseconds, the position in x and y and the speed for the direction in x and in y.
- **avoid_collisions()**: a function that avoids collisions between two vehicles by measuring the distance between two vehicles and reducing the speed of one vehicle. Use priority based decision making, when collision is predicted. In this case choose the vehicle with the smaller ID to reduce the speed. This process has to be repeated until all predicted collisions are resolved. If it is not possible to resolve all collisions, the program shall stop. As a result, vehicle 1 never slows down. Vehicle 2 slows to avoid vehicle 1. Vehicle 3 slows to avoid vehicles 1 and 2, and so on. The speed is increased and decreased in small increments over time, to maintain the maximum acceleration.



Sources

Priorities

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/cpp/central_routing/src/MultiVehicleTrajectoryPlanner.cpp#L83

Collision prediction

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/cpp/central_routing/src/VehicleTrajectoryPlanningState.cpp#L135

Speed reduction

https://github.com/embedded-software-laboratory/cpm_lab/blob/master/high_level_controller/examples/cpp/central_routing/src/VehicleTrajectoryPlanningState.cpp#L217

Class "MultiVehicleTrajectoryPlanner"

- **add_vehicle()**: a function that can add a new vehicle.
- **start()**: a function that will buffer the trajectory planning. This function includes the collision avoidance.

Main Program

Waiting Mode

The program shall start in waiting mode. It will transition into the running mode, when the following conditions are met.

- Vehicle IPS observations were received for all expected vehicles.
- All vehicles could be matched to the graph. All vehicles must be close to a center-line and oriented correctly.

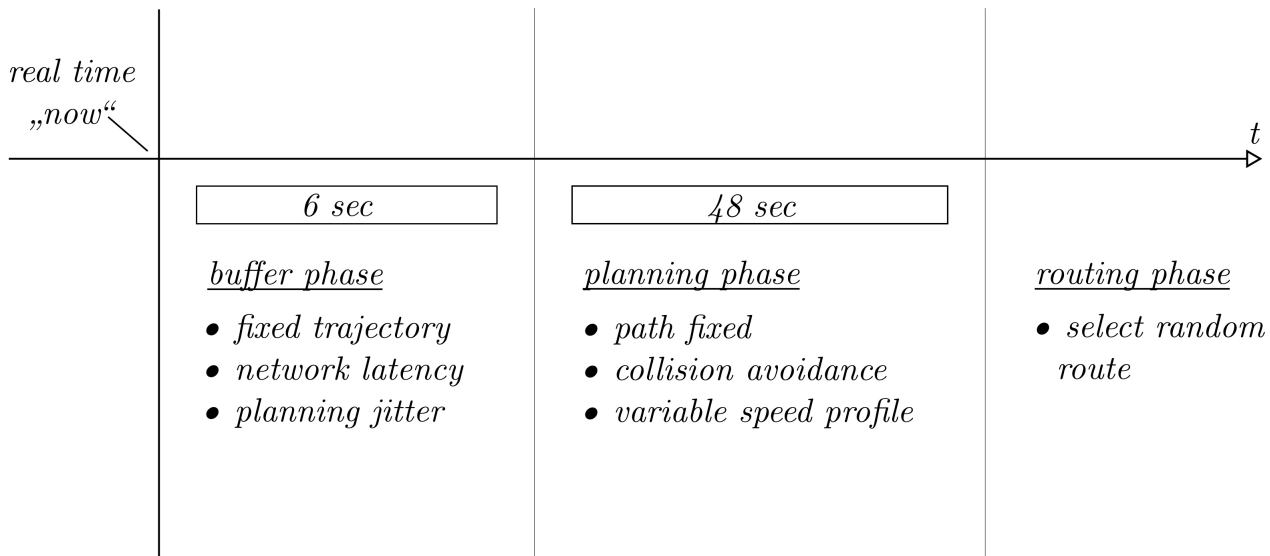
Running Mode

The route planning and collision avoidance shall run independently of the vehicle's actual behaviour. It is assumed that the vehicles follow the reference trajectories with sufficient accuracy.

Define phases for your program:

- Routing phase: Select route *sufficiently far* into the future. However the timeline is not precisely defined at this stage.
- Planning phase: Fix route and phase. Only the speed profile can be changed to avoid collisions. The planning phase considers a constant (but configurable) time span (currently 48 sec).

- Buffering phase: Fully define and commit trajectory. The purpose of the buffer phase is to allow for delays, both in sending the commands over the network and in the planning phase. For a particularly difficult collision avoidance, the planning phase may have a long computation time. The buffer phase can "absorb" this delay jitter.



Apply

Start via Lab Control Center

If you use the real lab application, make sure that the supporting software (LCC, IPS) is running and that the vehicles are online (only the ID LED is still blinking, all three other LEDs are turned on) and placed on the map. Assure that you activated Lab mode in the LCC. Then follow the steps described in [Running an HLC](#). **The example does not start, if collision cannot be avoided.** Therefore make sure not to have too many vehicles selected. Usually up to 10 vehicles will work fine.

Start via Terminal

In a new terminal, run:

```
cd ~/dev/software/central_routing_example
./run.bash 2,3,5,7,11
```

The list of integers in the run command must be the exact list of vehicle IDs for the active vehicles. The resulting trajectories will be shown in the LCC.

Computation Hardware

Main PC

The Main PC is the one to which the cables from the ceiling (IPS and lab cam) are connected. It is the third PC from the entrance door when you go straight, the one that is bigger than the others.

Specifications

Memory	16 GB
Processor	Intel Core i9-7900X CPU @ 3.3 GHz x 20
Graphics	GeForce GTX 1050 Ti/PCIe/SSE2

Intel NUCs

The NUCs are the distributed computing units for motion planning. They are located in the server rack.

Specifications

Model	NUC7i5BNK
Memory	16 GB DDR4 RAM @ 2400 MHz
Processor	Intel Core i5 7260U CPU @ 2.2 GHz
Graphics	Iris Plus Graphics 640