LL4 - Model Car - Documentation

Mauracher, Griesbeck, Mauerer

August 6, 2017

Contents

1 Introduction

This document describes the work of team *model_car* for the LL4 lab course at *Technische Universität München* in summer semester 2017.

The overall goal of the project is to set up a hardware-in-the-loop (HIL) scenario based on Genode with Fiasco.OC. An image of the scenario is shown in figure 1. As one can see, the software part of the HIL is a racing game called *SpeedDreams* which sends control commands to a physical model of a car. The concrete commands sent to the model are braking-, steering-, and acceleration requests. The model car performs the requests in hardware and returns sensor values to the simulation.

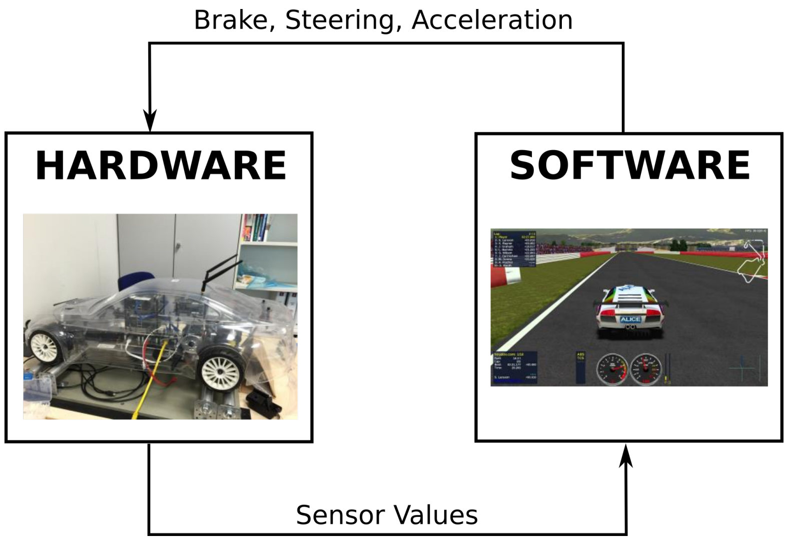


Figure 1: HIL scenario with SpeedDreams and a physical model of a car

This document describes the hardware part of the HIL, whereas the software part is implemented by another team of the lab course.

The project team consists of three master students at the *Technische Universität München*. Florian Mauracher (FM) and Christoph Griesbeck (CG) study *Computer Science* in the fourth and second semester, respectively. Thomas Mauerer (TM), on the other hand, studies *Automotive Software Engineering* in his third semester.

The complete project has always been done at meetings of the whole team which is why there has been no clear separation of tasks. Nevertheless, chapter 5 of the documentation is split into a servo part written by Christoph Griesbeck (5.1), one part about the Raspberry Pi written by Thomas Mauerer (5.2) and a PandaBoard part written by Florian Mauracher (5.3). Apart from that, chapter 6 is split accordingly.

2 Project Setup

The model car consists of the following components:

- Servos: In total there are three servos for the brakes, one servo for steering and an engine for acceleration placed on the model car. All of these components can be controlled via a servo controller board.
- Servo Controller Board: The servo controller board is a *pololu maestro mini servo controller board* which is attached to a Raspberry Pi via a usb-to-uart bridge. The braking servos and the steering servo, respectively are connected to the board, whereas the engine can not be controlled with it for several reasons.
- Raspberry Pi: The Raspberry Pi is responsible for sending control commands to the servo controller board. It receives its input data from a PandaBoard via a mqtt topic to which it is subscribed. The Raspberry Pi runs Genode with Fiasco.OC.
- PandaBoard: Besides the Raspberry Pi, the PandaBoard is the second ECU in the model car. It also runs Genode with Fiasco.OC. The task is to receive data from a simulation, transform the data into concrete servo values and send them via mqtt to the Raspberry Pi.

An image of the model car is shown in figure 2. The servos can be seen in box a. Box b shows the Raspberry Pi. The PandaBoard is placed on top of the car and can be seen in box c. The servo controller board is placed behind the Raspberry Pi which can not be seen in the image. All of the mentioned components are described in more detail in chapter 5.

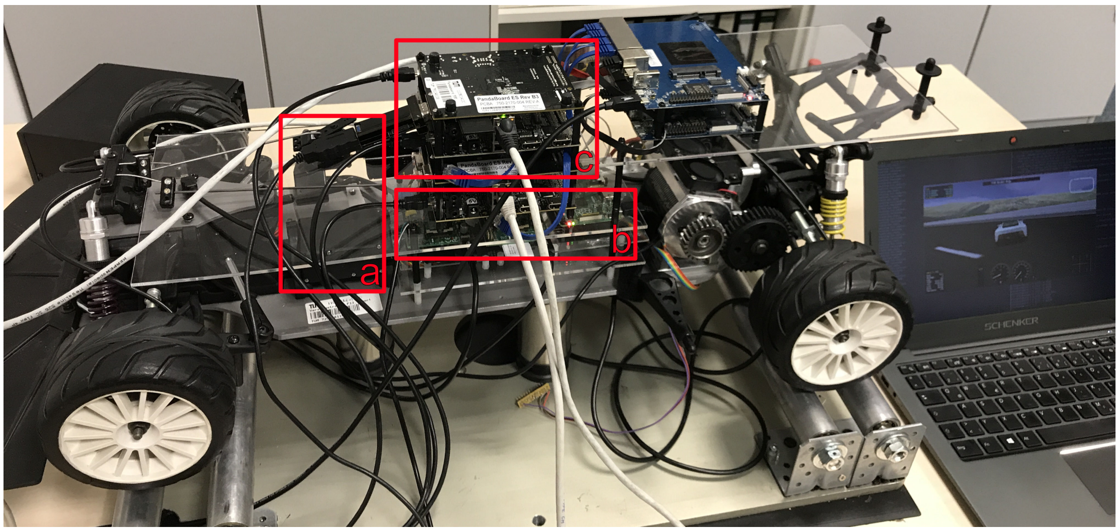


Figure 2: Model Car

3 Source Code Structure

The complete structure of the project directory is illustrated in figure 3. All of the subdirectories are described in more detail in the following chapter. For integration in the argos-research build environment it is expected that this repository is placed in the `genode/repos/` directory of the `argos-research/operating-system` repository.

```
car_project
├── run
├── include
│   ├── servo_session
│   └── controller_session
├── src
│   ├── mqtt
│   ├── rpi_component
│   │   ├── rpi
│   │   └── servo
│   └── panda_component
│       ├── panda
│       └── controller
└── old
    ├── linux
    └── pololu
```

Figure 3: Structure of the project directory

run The run folder contains the run files for the PandaBoard (`car_panda.run`) and the Raspberry Pi (`car_rpi.run`). These files are responsible for defining the project build components and the boot modules. Additionally, they contain the configuration for the started modules.

include In the include directory the header files for the servo component (`servo_session`) and the controller component (`controller_session`) are split up into their respective folders.

include/servo_session Contains all the header files for the servo component that is part of the Genode application running on the Raspberry Pi. Ihe interface is described in section 4.2.

include/controller_session Contains all the header files for the controller component that is part of the Genode application running on the PandaBoard. The interface is described in section 4.3.

src The source directory contains all the source files of the project. The different parts of the source code are split up into its own subfolders which are `mqtt`, `rpi_component` and `panda_component`.

src/mqtt Contains the header and the source file of the mqtt client. The client implements the mosquitto interface and is used in both, the `panda_component` and the `rpi_component`.

src/rpi_component Contains the code for the Genode application running on the Raspberry Pi. It consists of two components which are placed in different subfolders. The functionality is described in detail in section 5.2.4.

src/rpi_component/rpi The rpi component is responsible for handling the commands received from the mqtt server and forwarding them to the servo component.

src/rpi_component/servo The servo component controls the servos by sending commands over the serial connection to the pololu servo controller board.

src/panda_component Contains the code for the Genode application running on the PandaBoard. It consists of two components which are placed in different subfolders. The functionality is described in detail in section 5.3.3.

src/panda_component/panda The panda component sets up the network and uses the mqtt class to connect to the mqtt server during initialization. Afterwards, it passes on the `car-control` topic received commands to the controller component and publishes the result on the `car-servo` topic.

src/panda_component/controller The controller component transforms abstract commands from the simulation to pwm values.

old The old folder contains programs and scripts used during development.

old/linux An implementation of the rpi and panda components as linux user programs can be found in the `linux` folder. These were used for development during the first phase of the project when the rpi Genode image has not been available yet.

old/pololu In the subdirectory pololu resides a shell script and a c-program, which are originally from the pololu homepage and can be used for hardware testing.

4 Application Programming Interfaces

4.1 Mqtt client

We use mqtt for both, the communication between the simulation and the PandaBoard and the communication between the PandaBoard and the Raspberry Pi. The implementation of the mqtt client is based on the mosquitto library which is already ported for Genode. The client is able to publish messages on a specified topic or receive messages from a topic to which it is subscribed.

4.1.1 car-control

`car-control` is the topic name of the mqtt topic to which the PandaBoard is subscribed to. Commands to this topic are sent from the simulation. All commands need to have the following format:

Format: (command,value)
Command describes the type of request, i.e. braking, steering or acceleration. Value describes the strength of the command. All commands and values are listed in table 1.
Example: (1,0.5)
This is a brake request with half braking force.

Command	Value Range	Meaning
0	[-1.0 ; 1.0]	Steering
1	[0 ; 1.0]	Brake
2	[0 ; 1.0]	Acceleration

Table 1: Allowed values for the car-control topic

4.1.2 car-servo

`car-servo` is the topic name of the mqtt topic to which the Raspberry Pi is subscribed to. Commands to this topic are sent from the PandaBoard. All commands need to have the following format:

Format: (channel,value)
Channel is a number ranging from 0 to 11 and describes the channel number on the servo controller board. The braking servos are connected to channels 0, 1 and 2, whereas the steering servo is connected to channel 6. Values are pwm signals ranging from 4500 to 7500. The neutral value for the servo is 6000. The main engine can not be controlled with the pololu maestro servo controller board.
Example: (0,7500)
This command sends a pwm signal with value 7500 to channel 0. This means that the brake connected to channel 0 gets activated with full force.

4.2 Servo component

The servo component provides five methods. All methods return -1 in case of an error, else 0. Obviously, the getter functions return the requested value instead. Listing 1 shows the corresponding function declarations exported to Genode.

- The function **setTarget** receives the channel of the connected servo and the target position as parameters.
- The function **setSpeed** receives the channel of the connected servo and the maximum speed of the servo as parameters. A value of 0 means unlimited.
- The function **setAcceleration** receives the channel of the connected servo and the maximum acceleration of the servo as parameters. A value of 0 means unlimited.
- The function **getPosition** receives the channel of the connected servo and returns its current position.
- The function **getMovingState** returns 0 if no servo is currently moving, else 1.

Listing 1: Genode interface for servo component

```
GENODE_RPC(Rpc_setTarget, int, setTarget, unsigned char, unsigned short);
GENODE_RPC(Rpc_setSpeed, int, setSpeed, unsigned char, unsigned short);
GENODE_RPC(Rpc_setAcceleration, int, setAcceleration, unsigned char,
                                                        unsigned short);
GENODE_RPC(Rpc_getPosition, int, getPosition, unsigned char);
GENODE_RPC(Rpc_getMovingState, int, getMovingState);
```

4.3 Controller component

The controller component provides two methods each expecting a double as input and returning an integer value. Listing 2 shows the corresponding function declarations exported to Genode.

- The function **transform_steer** transforms a steering angle between -1 and 1 where -1 is completely right, and returns the corresponding pwm value in quarter microseconds for the servo.
- The function **transform_brake** transforms a braking value between 0 and 1 where 1 means fully applied, and returns the corresponding pwm value in quarter microseconds for the servo.

Listing 2: Genode interface for controller component

```
GENODE_RPC(Rpc_transform_steer, int, transform_steer, double);
GENODE_RPC(Rpc_transform_brake, int, transform_brake, double);
```

5 Architecture and components

The following chapter describes the individual components developed during this project as well as the interaction required between the components to fulfill the project goal of actuating the model car based on simulation data. A diagram that illustrates the interactions between the components is visible in Figure 4.

The data from the SpeedDreams simulation is first received by the mqtt server on the `car-control` topic, as described in subsubsection 4.1.1. The component running on the PandaBoard is subscribed to this mqtt topic and processes the information, see subsection 5.3. The resulting commands are then published on the `car-servo` mqtt topic, described in subsubsection 4.1.2. The component running on the Raspberry Pi (subsection 5.2) in turn is responsible for forwarding the commands to the servo control component (subsection 5.1) according to the rules of the commands.

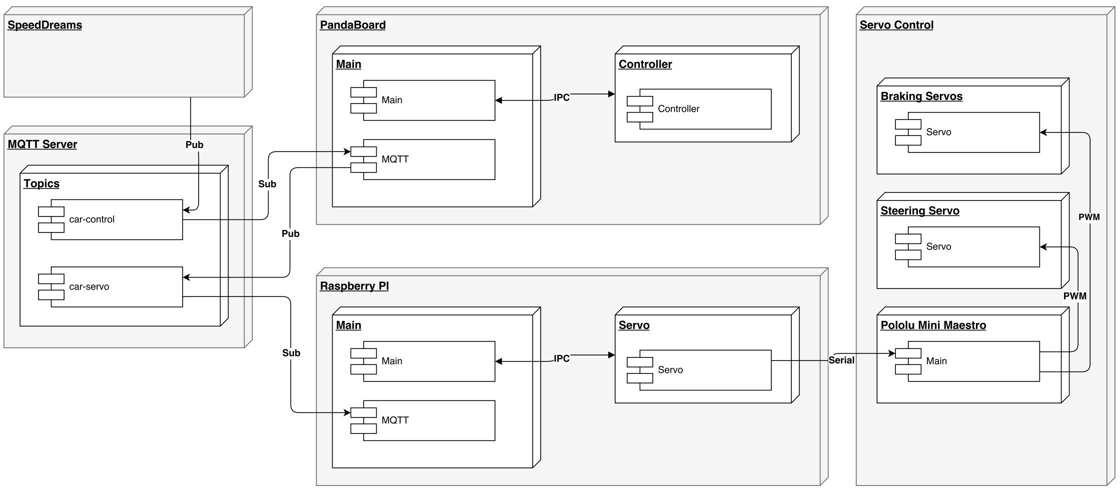


Figure 4: Components overview

5.1 Servo (CG)

5.1.1 Requirements

From the original functional requirements the following tasks related to the servos were derived:

- **TS1**: Control steering and braking servos
- **TS2**: Control main engine(optional)
- **TS3**: Connect servos to the controller board
- **TS4**: Commissioning of the servo controller board
- **TS5**: Check servo controller boards functionality

TS1 The first, most important and most obvious task is to control the one steering and three braking servos mounted on the car. This task can be seen as the overall goal, as finishing all other non optional tasks leads to accomplishing this goal.

TS2 To control the main engine was another task, though marked as optional. Accomplishing it would have been impressive to spectators, however it was not completed due to various reasons. First off all, time was running out at the end of the project due to several minor and major difficulties described in chapter 6. Furthermore the main engine can not be controlled by the Pololu Maestro Mini servo controller board, because of its complexity and power consumption. Instead another more complex controller needs to be used, which requires an additional serial connection. However Genode as an operating system only supports one serial connection currently. That is why an additional device, most likely a Raspberry Pi, would have been needed to be integrated into the setup to control it.

TS3 Another Requirement was to connect the servos to the servo controller board. However the braking servos were already connected when the project started and never disconnected and therefore introduced no work. In contrast the steering servo had to be connected several times to the servo controller board.

TS4 The next task was to commission the servo controller board. This was first done with a graphical program provided by the manufacturer of the servo controller board. Afterwards a shell script and c code were used to control it. This will be described in more detail in section 5.1.3.

TS5 At last the servo controller board had to be checked if it is working and for its capabilities. This was done while commissioning it.

5.1.2 Hardware Description

As already described in section 2 the used actuation hardware consists of three braking servos, one steering servo and a Pololu Maestro Mini servo controller board, which is displayed in figure 5, and a Raspberry Pi. Because there are only three braking servos mounted on the car, the front wheels can be controlled individually in contrast to the rear axle where both brakes are actuated by the same servo.

All four servo engines are controlled with a 50Hz pulse width modulation(PWM) signal as is standard for servo engines. The duty cycle of each PWM-signal is between 1 and 2 milliseconds. Because of the nature of the servo controller board, values for the duty cycle of a PWM signal need to be between 4000 and 8000, which corresponds to four times the value in microseconds.

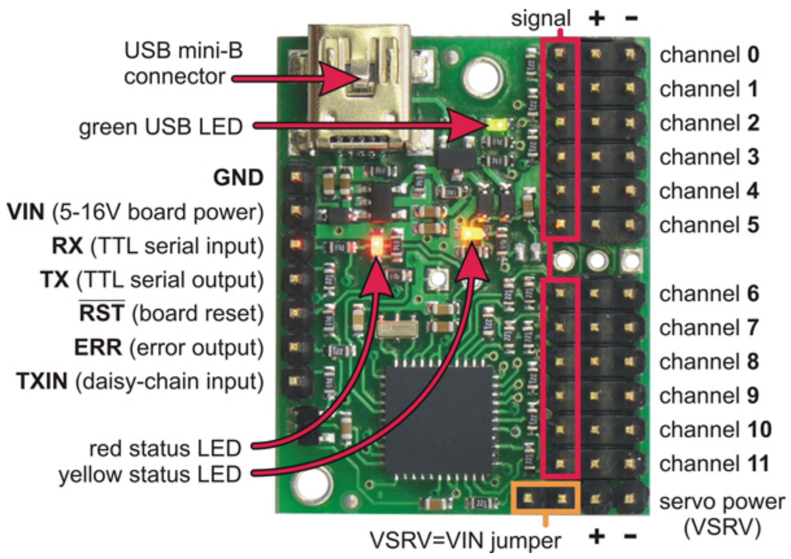


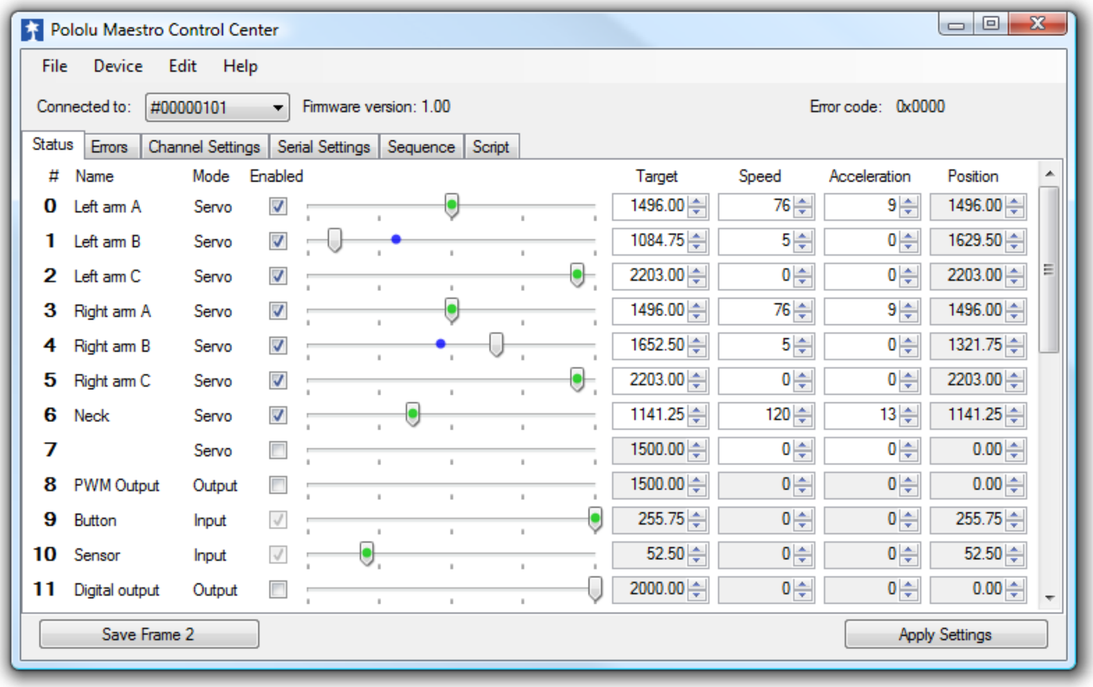
Figure 5: Pololu Maestro Mini

The installed controller board features twelve PWM channels which need to be run on the same frequency but support different duty cycles, so that up to twelve servos can be controlled individually. For communication with a computer, serial communication via USB or UART/TTL on a byte basis can be used. The baudrate for the UART connection is normally autodetected, but can be configure as well with the program. Messages always consist of eight bits without a parity bit and the last bit being a stop bit(8N1). It also has the capability to execute small script programs which though was not investigated in this project. Features like setting a maximum speed for the change of the PWM signal and setting a maximum acceleration of the PWM signal and therefore the servo allow smooth position changes of the servos. The braking servos are connected to channel 0,1 and 2 and the steering servo is connected to channel 6.

Initially the steering servo was not working and therefore was replaced by the chair. Additionally we configured it after its replacement to have a correct neutral position. Also the braking servos were tried to be configured to engage at the same time. However the made improvement was only minor.

5.1.3 Software Description

For first testing a shell script and an interactive program with a graphical user interface, depicted in figure 6, both working on all common operating systems, are provided by the supplier and were used several times. Both helped during the whole project to find errors. Moreover some small C example code for communicating with the servo controller board is provided and was used as a starting point during development. The shell script and the example code is for reference located in the old/pololu folder described in chapter 3.



Connected to: #00000101   Firmware version: 1.00   Error code: 0x0000

#	Name	Mode	Enabled	Target	Speed	Acceleration	Position
0	Left arm A	Servo	✓	1496.00	76	9	1496.00
1	Left arm B	Servo	✓	1084.75	5	0	1629.50
2	Left arm C	Servo	✓	2203.00	0	0	2203.00
3	Right arm A	Servo	✓	1496.00	76	9	1496.00
4	Right arm B	Servo	✓	1652.50	5	0	1321.75
5	Right arm C	Servo	✓	2203.00	0	0	2203.00
6	Neck	Servo	✓	1141.25	120	13	1141.25
7		Servo	☐	1500.00	0	0	0.00
8	PWM Output	Output	☐	1500.00	0	0	0.00
9	Button	Input	✓	255.75	0	0	255.75
10	Sensor	Input	✓	52.50	0	0	52.50
11	Digital output	Output	☐	2000.00	0	0	0.00

Figure 6: Pololu Maestro Control Center

Because of delays in the installation of Genode on the Raspberry Pi that is communicating with the servo controller board, a first program was developed for controlling the servos on Linux, especially Raspbian. In section 5.2.3 the setup using this version is described in more detail. The serial port used for communication is exposed as a file in

Linux. Therefore commands can be send by writing into the correct file. The program can be found in the old/linux folder as described in chapter 3.

All functions provided by the servo class/component have a similar structure. First the input values are checked for valid range, second a command is built and sent and lastly its return value checked for errors. If the servo controller board is supposed to answer, the response is read in, checked for errors and returned. There are three possible protocols called Compact Protocol, Pololu Protocol and Mini SSC Protocol supported by the controller board. We decided to use the compact protocol, as its messages are shorter and less complex than the ones of the Pololu Protocol but doesn't reduce the resolution of PWM signals as the Mini SSC Protocol does. Each command is represented as a char array, where the 7 least significant bits represent one serial message. The first message in the Compact Protocol is always the actual command type e.g. 0x84 for setting a target PWM value. The next one is the channel that is affected and lastly a value is sent often split up into two messages in little-endian format. For example a complete command in the Compact Protocol for setting the PWM duty cycle to 1500 on channel 0 would be 0x84, 0x00, 0x70, 0x2E. Although only the command for setting the target position is used in the project, several more were implemented and are documented in chapter 4.2

Only little changes were needed to port the code to Genode. Additionally the program was made a complete component, which can be called via remote procedure calls(RPC), which lead to several difficulties. The exposed functions are documented in chapter 4.2. How the code is used on the Raspberry PI is described in more detail in the following chapter 5.2.

5.2 Raspberry Pi (TM)

5.2.1 General information

The Raspberry Pi is one of the ECUs used in the setup. There are three Raspberry Pis placed on the model car in total, whereas two of them are Raspberry Pis of the first generation and one is a Raspberry Pi of the second generation. For our purpose we only need one of them.

The main goal of the Raspberry Pi is to receive commands from the second ECU which is the PandaBoard and control the servos with the given command. Therefore, it is connected via a usb-to-uart bridge with the *pololu maestro* servo controller board which in turn is connected to the braking and the steering servos. This can be seen in figure 4.

Since the Raspberry Pi has IO pins itself, it would have been possible to directly connect the servos to the Raspberry Pi and completely give up the *pololu* board. However, there are mainly two reasons for not doing so. Firstly, because in this lab course we want to simulate a real car which usually consists of many different ECUs that have to work together. Secondly, because the *pololu* board has more power for controlling the servos than the Raspberry Pi.

5.2.2 Task description and solutions

From the initial task description we can derive the following tasks related to the Raspberry Pi component:

- **TR1**: Install Genode with Fiasco.OC
- **TR2**: Implement ProtoBuf by Google
- **TR3**: Develop mqtt client
- **TR4**: Connect with servo controller board
- **TR5**: Implement Genode servo controller application
- **TR6**: Convert control commands into concrete servo values
- **TR7**: Read sensor values (optional)

TR1 Since the complete project is based on Genode with Fiasco.OC, the first task was obviously to install the operating system on the Raspberry Pi. There have been many issues with this task which is described in detail in section 6.2. Shortly summarized, we were not able to build the operating system for the Raspberry Pi until the last two weeks of the lab course. For this reason we came up with a complete Linux-based solution at interim. The Linux-based solution is explained in section 5.2.3.

TR2 The next two tasks are related to the message exchange between the PandaBoard and the Raspberry Pi. Originally, it was planned to use protocol buffers (ProtoBuf) by Google for the communication. ProtoBuf is a language- and platform-neutral way of serializing data comparable to XML or JSON format. However, the main advantage is its binary format which allows really fast parsing and shrinks messages to a minimum size. We have already started by specifying a *.proto* file for the message type. However, afterwards we came to the conclusion that ProtoBuf is not really necessary and can be replaced by mqtt with messages in plain-format. This has mainly two reasons. Firstly, because the messages sent from the PandaBoard do only consist of a few characters and also have a really simple format. Therefore, it is absolutely fine to use plain-format. Secondly, because mqtt is already used for the communication between the simulation and the PandaBoard. Therefore, we have already a mqtt broker in our project and apart from that, we have to implement only one of the two techniques.

TR3 The mqtt client is only implemented once in the project and is used by the Raspberry Pi, as well as the PandaBoard. The mqtt client is based on the mosquitto library which is already ported for Genode. The Raspberry Pi is subscribed to the topic `car-servo` at which the PandaBoard publishes its messages. A description of the message format can be found in section 4.1.2. After receiving the messages, the Raspberry Pi only checks for valid input values and controls the servos accordingly.

TR4 The next task was to connect the Raspberry Pi with the *pololu maestro* servo controller board. Obviously, this is necessary in order to be able to control the connected servos. The *pololu* boards are quite comfortable in the way that they support a serial connection. Therefore, we only had to plug the usb-to-uart bridge into the Raspberry Pi.

TR5 The Genode servo controller application is the main task regarding the Raspberry Pi. It is completely described in section 5.2.4.

TR6 Another obvious task was to convert the abstract control commands coming from the simulation into concrete servo values that can be used in order to control the servos. However, in our opinion it is more meaningful to separate this task from the rest of the tasks of the Raspberry Pi. That is because the Raspberry Pi is actually only responsible for controlling the servos, but not for the core logic. Therefore, we shifted this task to the PandaBoard and expect mqtt messages that are already in the right servo format.

TR7 The last task regarding the Raspberry Pi was to read sensor values of the model car which could be steering angles or wheel speeds, for instance. This task was marked as optional in the initial task description. Actually, this task is required in order to close the loop of the HIL-testbed. However, due to the fact that the model car does currently not contain any sensors, we were not able to achieve this task.

5.2.3 Linux-based solution

One major problem, we experienced during the development, was that we were not able to build the Genode operating system for the Raspberry Pi and it was not predictable whether this would be possible until the end of the lab course, at all. For this reason we decided to implement all tasks of the Raspberry Pi on a Linux basis as a backup solution.

The used Linux distribution was a *Raspbian* image copied onto a sd-card that was plugged into the Raspberry Pi. Of course, Linux comes with a high overhead for what is actually required in order to control some servos. Indeed, this is one of the main reasons why we use a microkernel-based solution, at all. However, Linux does also simplify a lot of things during the development.

One of those simplifications is for example an ssh server. After changing something in the program code, we could simply transfer the updated code to the Raspberry Pi via ssh or even program directly on the Raspberry Pi. With Genode, this does always require to build a new image, copy the image onto the sd-card and reboot the Raspberry Pi.

A second simplification is the way how Linux handles its serial connections. After connecting the *pololu* board with the Raspberry Pi, the board appears inside the `/dev` directory and we can simply use normal file operations like `open, read` and `write` in order to control the servos that are connected to the *pololu* board.

A last advantage of Linux over Genode is the ability to easily run a mosquitto server beside of the main program. The mosquitto server is required for mqtt. The good thing about running the mosquitto server directly on the Raspberry Pi is that we do not have to struggle with correctly setting up the network since the Raspberry Pi is already in the same network as the PandaBoard.

The source code itself is similar to the Genode version with only a few differences. One is for example that there is only one task for the whole program under Linux. In Genode it is common to separate different components into different tasks which requires a few changes in the source code, as well. For the header and source files of the mqtt client it was possible to reuse them in the exact same way in the Genode version. As already mentioned, the only big difference compared to the Genode version is the way how commands via the serial connection are sent. In Genode this requires the usage of a so-called `terminal` connection which is described in detail in section 5.2.4.

5.2.4 Genode application

The Genode application is the main task regarding the Raspberry Pi. One of the main reasons for using a microkernel-based operating system like Genode is the minimal overhead of the system itself. The disk image file does basically only contain programs and libraries that are really required in order to fulfill the task and nothing more. Especially in embedded systems, where computing power and memory are limited, this is a big advantage over "normal" operating systems like Linux.

The relevant components of the Genode application running on the Raspberry Pi are illustrated in figure 7. As one can see, the application consists of two different components that are connected via IPC. This is the usual procedure in microkernel-based operating systems like Genode. The advantage of seperating tasks into different components is that a failing component usually only leads to a crash of that individual component, but not of the whole program.

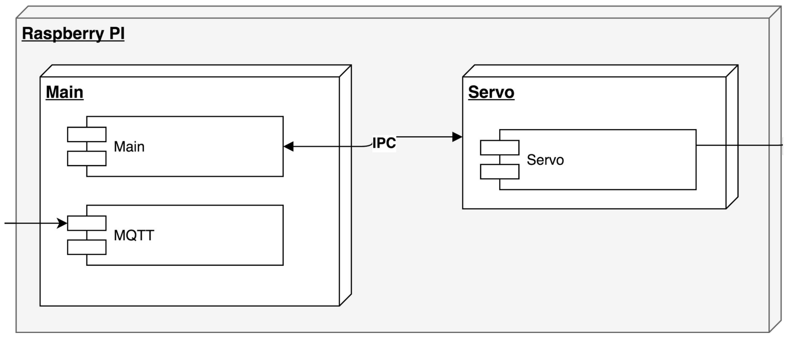


Figure 7: Genode components of the Raspberry Pi

The `Main` component fulfills the following functions:

- Setup the network
- Subscribe to `car-servo` mqtt topic
- Receive mqtt messages
- Call servo component

The relevant source code file is *main.cc* placed in `src/rpi_component/rpi/` according to the source code structure shown in figure 3.

The `Servo` component on the other hand is responsible for sending the received commands to the *pololu* board via the serial connection. The relevant files for the `Servo` component are the header files *client.h, connection.h* and *servo_session.h* placed in `include/servo_session/` and the *main.cc* file in `src/rpi_component/servo/`.

The *servo_session.h* defines all the function names and exports them to Genode. This is already described in the API description in section 4.2. The *main.cc* file on the other hand implements all of the functions inside a `Servo_component` struct that inherits from `Genode::Rpc_object`. Apart from that, it implements a `Servo_root` class that in turn inherits from `Genode::Root_component` and is responsible for creating the session of a `Servo_component` object. This is the usual procedure of implementing Genode objects.

An object of a `Terminal::Connection` is required in order to write to the serial connection. This differs from the Linux-based solution where normal file operations were possible. However, the Genode version is not really more complicated because the terminal does also provide a `write` function which behaves the same as the Linux function. Listing 3 exemplarily shows the `setTarget` function where the `write` operation can be seen. The `command` in the listing follows the *pololu* compact protocol which is documented at the *pololu* homepage[1].

```
int setTarget(unsigned char channel, unsigned short target)
{
    // check for valid input
    [...]

    // create command
    unsigned char command[] =
        {0x84, channel, (unsigned char)(target & 0x7F),
        (unsigned char)(target >> 7 & 0x7F)};

    // write command to serial connection
    if (_terminal->
        write(command, sizeof(command)) < sizeof(command)) {
            PERR("error writing");
            return -1;
    }

    return 0;
}
```

Listing 3: setTarget function of the servo component

The last important file for the Raspberry Pi is its run file (*run/car_rpi.run*) in which all build components and boot modules are defined. Apart from that, the configuration of the loaded modules is placed inside the run file and the IP address of the mosquitto servo is declared here.

[1]`https://www.pololu.com/docs/0J40`

5.3 PandaBoard (FM)

5.3.1 Overview

The modelcar hardware contains 4 PandaBoards intended for running the Engine Control Units (ECU) communicating with the SpeedDreams simulation, processing the simulation data and generating the servo commands. While two PandaBoards are used by the simulation team to process the simulation data from SpeedDreams, another PandaBoard is used by our team to process the hardware control commands described in subsubsection 4.1.1.

5.3.2 Requirements

The following tasks were derived from the initial project description for the component running on the PandaBoard:

- **TP1**: Install Genode with Fiasco.OC
- **TP2**: Develop MQTT client
- **TP3**: Convert control commands into concrete servo values
- **TP4**: Generate control commands

TP1 Genode running on top of the Fiasco.OS microkernel is used as operating system for the PandaBoard. While there were less issues with running Genode on the PandaBoard compared to the Raspberry PI, a Linux userland implementation of the PandaBoard component is also available in the `old/linux` folder and was used during initial development and testing.

TP2 As described in subsubsection 5.2.2 the same MQTT client based on the mosquitto library was used for the Raspberry Pi and the PandaBoard. The PandaBoard subscribes to the `car-control` topic, processes received messages and then publishes to the `car-servo` topic.

TP3 The main purpose of the PandaBoard component is the transformation of generic control commands (eg. steer left, break) to concrete servo values. The format of the input and output messages is described in section 4 while the transformation is described in subsubsection 5.3.3

TP4 To test the application independently of the SpeedDreams simulation, control commands can also be supplied directly as described in subsubsection 5.3.5

5.3.3 Genode application

The Genode application on the PandaBoard consists of two separate tasks communicating over IPC. The *Main* task handles initial setup and network communication while the *Control* task is responsible for the transformation of control commands to concrete servo commands. The interaction between these components is visible in Figure 8.

The runfile (`run/car_panda.run`) contains the build and runtime dependencies as well as configuration settings for individual components.

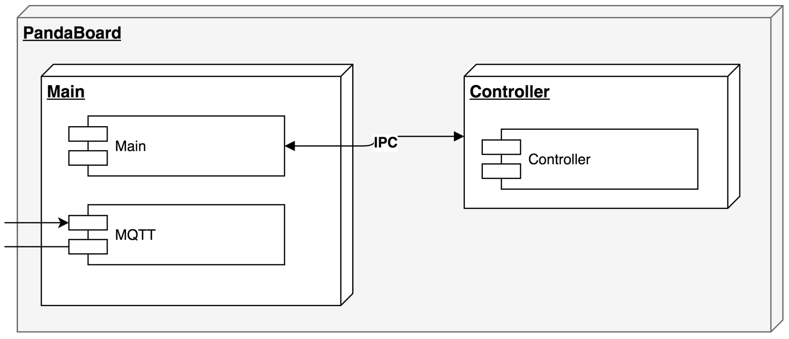


Figure 8: Genode components of the PandaBoard

After startup the *Main* task reads the network configuration from the runfile and configures the network accordingly with either a static IP address or a IP address dynamically obtained over DHCP.
Afterwards it retrieves the IP of the MQTT server from the runfile and attempts to establish a connection. The task then subscribes to the *car-control* MQTT topic which causes all messages that are published on this topic to be forwarded to the application.

A callback that is executed on reception of every message then copies the data to a buffer where it is in turn processed by the main loop of the task. After parsing the type of the message, the command is then sent to the responsible function of the *Controller* task over IPC.

As all IPC under Genode is synchronous, the *Main* task resumes execution after the function call from the *Controller* task returns with the servo adjustment value obtained from the processed command. The servo value is then published to the `car-servo` MQTT topic to which the *Raspberry PI* is subscribed and handles all further processing.

5.3.4 Command processing

The *Control* component is responsible for transforming the messages received in an hardware independent format as described in subsubsection 4.1.1 to concrete servo values suited for the servos built into the model car.

Listing 4 contains the algorithm for calculating the steering servos commands. First the input value is checked to verify it is within the acceptable input range. The value is then inverted to ensure correct mapping from the steering direction to the servo direction.

Afterwards the value is transformed by first adjusting the range to $0 - 1$ and then mapped to the servo input range by multiplying it with the input range of the servo (*SERVO_UPPER_BOUND - SERVO_LOWER_BOUND*) and then adding the minimum servo value *SERVO_LOWER_BOUND*.

```
int transform_steer(double value) {
  if (value < -1 || value > 1) {
    PERR("Invalid steering angle - range is -1 to 1");
    return -1;
  }

  // Invert value as SpeedDreams thinks -1 is right
  value = -value;

  value = (value + 1)/2;
  return (SERVO_UPPER_BOUND - SERVO_LOWER_BOUND) * value +
     SERVO_LOWER_BOUND;
}
```

Listing 4: Algorithm for calculating the steering servo commands

While the current controller only transforms the servo commands and forwards them, the design would also allow more high-level commands from the simulation to be processed where a single control `car-control` command generates a series of `car-servo` commands.

One example for this would be ABS style actuation of the servos and relying on sensor data to control the breaking strength. As no suitable sensor measurements were available during this project, this part has not been implemented.

5.3.5 Testing

To verify the data processing and the actuation of the hardware servos based on control commands sent to the `car-control` MQTT topic a test script was developed. This way the hardware side of the HIL setup described in section 1 can be tested independently of the software side developed by another project team.

The publish–subscribe pattern used by the MQTT server allows multiple clients to simultaneously connect to the server and subscribe to updates on a certain topic or publish messages to a topic. In this style of communication there are no direct communication links and the components are loosely coupled, simplifying the testing of the interfaces between components.

By using the MQTT commandline tools available under Linux in the `mosquitto-clients` package message can easily be injected into the `car-control` topic without changing the configuration of any components. Listing 5 shows a small test script that continuously sends steering control messages that cause the steering servos to alternate between actuation to the left and to the right.

```
#!/usr/bin/env bash
MQTT_IP=$1

while true; do
  mosquitto_pub -h $MQTT_IP -t 'car-control' -m '0,1.0'
  sleep 1
  mosquitto_pub -h $MQTT_IP -t 'car-control' -m '0,-1.0'
  sleep 1
done
```

Listing 5: Injecting steering commands over MQTT

In a similar manner the `mosquitto_sub` commandline tool can be used to subscribe to MQTT topics. This provides easy access to the communication between components for debugging and testing purposes.

```
mosquitto_sub -h $MQTT_IP -t 'car-servo'
```

Listing 6: Subscribing to MQTT topics

Listing 6 shows the command used to monitor the message published by the panda component. In both listings `$MQTT_IP` is a variable containing the IP address of the MQTT server.

6 Challenges

6.1 Lab computer setup (CG)

Most of our work was done on the lab computer. During this work we encountered several challenges which proved to be at least time consuming. The lab computer is attached to the lrz network and on booting it loads an image from a central server. Because of this network set up the user has very restricted rights on the computer to prevent abuse.

However this also prevented us from installing missing libraries on the computer, e.g. mosquitto for gaining experience and testing of mqtt communication. Furthermore our working directory(/var/tmp) was not mounted at some point for an unknown reason. Because the working directory was on the local harddrive, everything had to be set up again on the second lab computer and the latest changes were lost.

Additionally it was planned to use tftp boot to ease booting a newly compiled image. However, this first failed due to firewall rules and later due to missing administrator rights on the computer.

All these problems could not be fixed by ourselves, but needed either a workaround or a technician. Nevertheless trying to fix them ourselves cost us a lot of time. For example getting tftp boot to work took us a whole afternoon of three hours although we were getting help from the system administrator of the chair. Moreover it would have saved us time later, especially as the sd card reader was not reliably working.

6.2 Raspberry Pi build (TM)

Until the last two weeks of the lab course we were not able to build the Genode operating system for the Raspberry Pi. According to the documentation of the *argos-research* project[2], the only requirement in order to build the image for the Raspberry Pi is to change the `GENODE_TARGET` inside the Makefile to `foc_rpi`. However, due to problems in the kernel of Genode itself, the compilation always failed.

Since our knowledge about the internals of Genode is limited, we were not able to fix this problem by ourselves. However, trying to compile and searching for the problem still took quite a long time of our lab course.

Apparently, after some time the problem was fixed by a member of the *argos-research* team by updating the Genode version and changing from the `foc` kernel to the `focnados`. However, this way we were able to compile the image, but the image was not bootable on the hardware. Later, it turned out that a wrong base address inside the bootloader

[2]`https://argos-research.github.io/platforms.html`

was causing this problem. Again we were not able to solve this issue by ourselves.

In the meantime we decided to switch to *Raspbian* instead of Genode since we could not predict whether we would be able to compile the operating system until the end of the lab course, at all.

In the last two weeks of the course the problem was again fixed by the member of the *argos-research* team and we could finally switch back to Genode.

6.3 Build infrastructure (FM)

The *argos-research/operating-system* repository provides a Varantfile in combination with a provision script allowing the setup of the build environment in an automated and reproducible manner. Unfortunately the vagrant setup caused some issues with multiple project members due to the use of Virtualbox shared folders for the complete source code repository. Additionally the provision script could not be used in the lab environment due to missing access right for some steps that required elevated permissions and some hardcoded paths only available in vagrant.

After an update of the provision script based on feedback during the first weeks of the practical it can now also be used to setup up the project outside of the lab environment and without direct dependence on vagrant as virtualization backend.
To ensure a simple setup of the model-car project with all it's dependencies without direct integration in the *argos-research/operating-system* repository a `setup.sh` script is provided in the top level of the project repository and the required steps for the build are described in the projects `README.md`.

7 Summary and Future Work

As mentioned in section 1 the overall goal of this project is to set up a HIL scenario. Since there are no physical sensors on the model car, it is currently not possible to measure and send any of these values. Therefore, the loop is not closed at the moment.

Apart from that, it is only possible to control the braking and the steering servos. That is because the main engine can not be controlled via the *pololu maestro* servo controller board, but requires an own controller with more power. A further technical problem resulting from this is the need of a second Raspberry Pi, since Genode is currently not able to handle more than one serial connection simultaneously. In the future a second Raspberry Pi could be used for controlling the main engine.

As always there are several possibilities for further improvement of the project. The most obvious one is to add sensors to the car in order to close the loop for a real HIL test-bed. Especially in combination with a more advanced control, e.g. ABS, implemented on the PandaBoard, this would improve the whole setup.

Furthermore, the project architecture could be enhanced by exporting the mqtt class into its own component. As a result of this the main component of the Raspberry Pi and the PandaBoard, respectively, could be simplified since it would no longer be required to setup the network in these components.

Moreover, the mqtt communication could be redesigned so that there is a topic for each possible command. This way, a message would only consist of a single value.