Documentation - Safety Car

HW/SW-Co-design with (Lego)Cars

Lab course WS 2013/14

Christoph Weinisch, Florian Hisch, Lukas Kunzemann
Technische Universität München

March 23, 2014

Contents

1 Introduction

1.1 Outline

We got the task to design and constructed a car model during our lab course HW/SW-Co-design. The main aim on this was to design a distributed software architecture. This architecture consists of four FPGAs and a central Linux-PC.

The car itself has a tower-like structure with different sized levels. These levels are made out of wooden planks which are robust enough but not too heavy in order to the motor power. As pillars between the levels we took metal and wooden sticks.
We started with the biggest plank (plywood) and step by step we fixed all components on it. We wanted to assemble the car as fast as possible, but the components shipped in only in small portions. Hence we also developed the software accordingly.

Our first aim was to make the car drive, controlled by a keyboard. Then we wanted to make the control more comfortable and start to play with Wi-Fi. At this point we want to say 'Thank You' to our partner team, which helped us with our Linux-PC and the Wi-Fi very well!

To get some intelligence into the car control we plan to make our car stopping if something is crossing the car's way. So ultrasonic sensors were fixed at the front and at the end of the car.

We also played with a simple 'exploration mode'. The car should be driving in one direction. Is there an obstacle it turns left until the obstacle is not in it's way anymore. Then the car drives further in this direction.

1.2 Prelude

Our team exists of Florian Hisch, Christoph Weinisch and Lukas Kunzemann. We all are studying computer science in the fifth semester. The lab course in WS 2013/14 started in the end of October. So we got about three months to work on this project. Even though the course was labeled 'on LEGO-Cars' there was no 'brick-made car' left. Our assignment was instead assembling a 'self-made-car' and to implement all the sensors which are needed to navigate.

We got almost all components from our advisers. Only some trifles we brought by ourselves.

2 Hardware Setup

2.1 List of components

Here is a (maybe incomplete) list of our used components:

- wooden chassis , 40 cm x 35 cm
- 1 small wooden board (approx. 20 cm x 25 cm)
- 11.1 V battery with 5000 mAh
- 11.1 V battery with 3500 mAh
- 4 soft-wheels(diameter: aprox. 12 cm)each soft-wheel can take max. 3 kg.
- 4 Ethernet-UART connected to the switch
- 4 Pololu motors (max. power: 60 W @ 12,5 V and 5 A)
- 1 Board (IMX6 Sabre Lite)
- 4 H-bridges (dual channel but we only use one channel)
- 4 DE0-Nano-Boards (FPGA's)
- 1 LAN switch
- 2 DC-DC-Converters
- 2 ultrasonic sensors

2.2 Overview

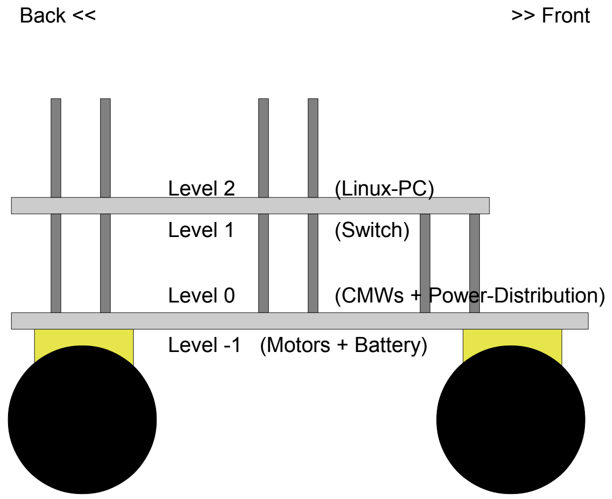


Figure 1: The car is divided into four levels. Each level has a different motto.

2.3 Level -1

The lowest level -1 (see 2) contains the four motors and the batteries. Each motor is fixed on a small wooden block so the batteries are in a more safe place.

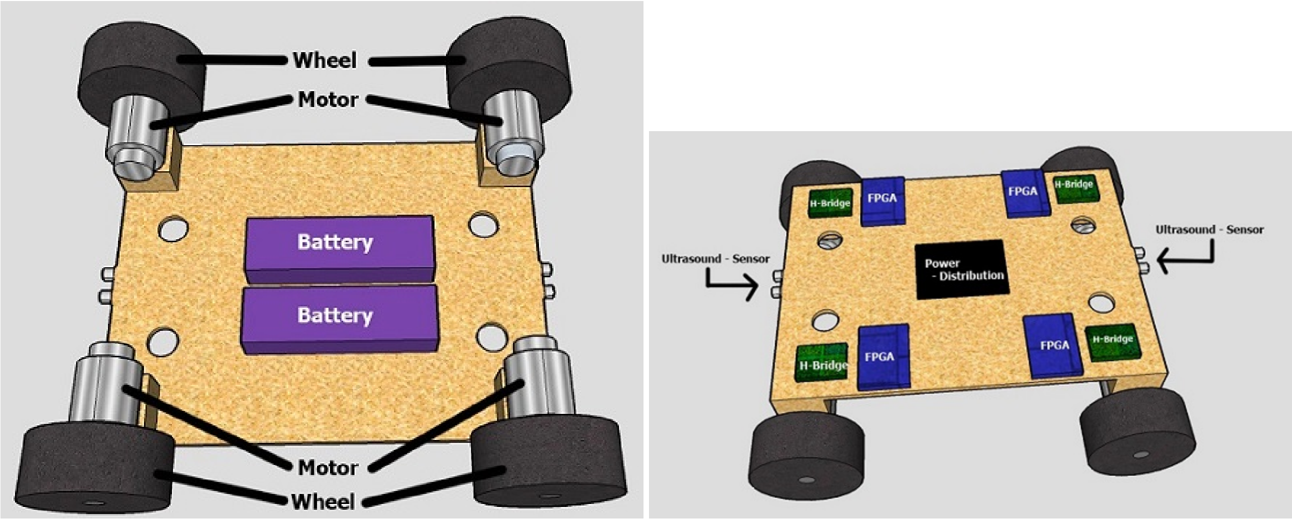


Figure 2: The lowest level -1 (see left figure) contains the four motors and the batteries. Level 0 (right figure) contains the four CMW-Units and the Voltage-Distribution.

2.4 Level 0

The main level (see 2) is the top of the wooden chassis. The power-distribution is fixed in the middle of the board and provides power to the H-bridges, the four FPGAs and the Linux-PC. The four H-bridges are placed in the corners. Next to each H-bridge there is the corresponding FPGA board.

At the back and front side of the car are ultrasonic sensors. They are fixed in an angle that the sensor measures the floor in 3.5 meters. Doing so gives us a maximum distance for reliable sensor values.

The next higher level sits down on 4 metal pillars fixed on the main level.

2.5 CMWUnit - Control-Motor-Wheel-Unit

Each Control-Motor-Wheel(CMW)-Unit consists of:

- One *Ethernet-UART* connected to the central FPGA
- One *DE0Nano-Boards* (FPGA)
- One *H-Bridge* (dual-channel but we only use one channel)
- One *Pololu Motor* (max. power: 60 W @ 12 V, 5 A).
  Problem: Many components can not take over 2 A!
- One *Soft-Wheel* (diameter: aprox. 12 cm)
  Problem: Each Soft-Wheel can take max. 3 kg

2.6 Level 1

The small and light wooden board (see 4) is pillared by four metal sticks. From the button up there are 4 Ethernet-UART's which are glued on the wood and connected by LAN-cables to the switch.

The switch is also fixed by glue on the wooden panel from the bottom up. To make it more solid the switch is first glued on a very small piece of wood and this is glued on the wooden panel.

2.7 Level 2

On the highest board (see 4) is the evaluation board (Linux-PC). Through a cut-out the evaluation board is connected with the switch on the other side. On the board are also pillars if someone would like to upgrade the car and build one more floor. For example fixing a camera or something similar on the top plank.

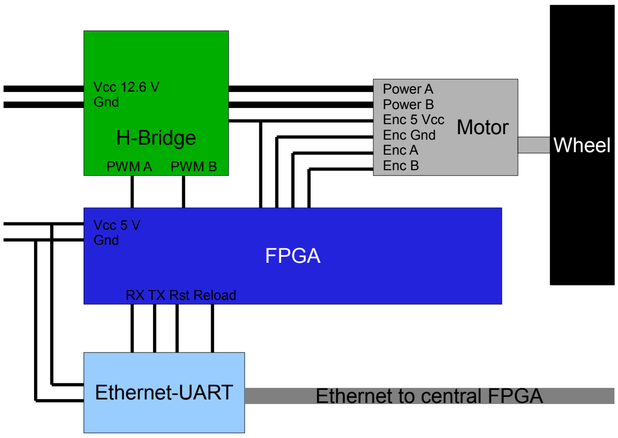


Figure 3: Interconnections in the Control-Motor-Wheel-Unit

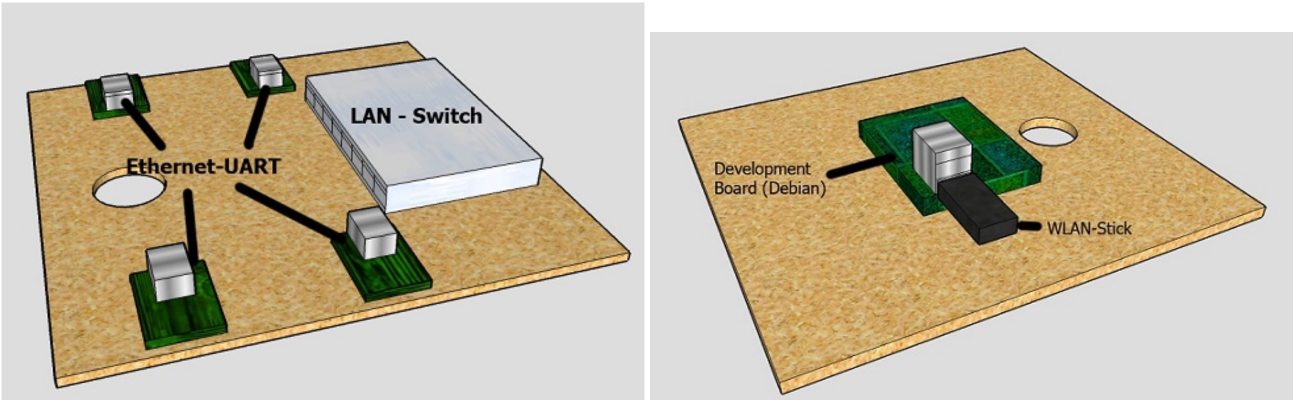


Figure 4: Level 1 (see left figure) contains the Ethernet-Switch. Level 2 (see right figure) contains the (big) Sabre-light i.mx6 (Linux-PC).

3 Construction of the car

First we didn't know which dimensions to take for the car. We had to decide between a big but heavy plank and a smaller one. We played around with different sizes and chose a wooden board with about 50 cm x 40 cm first. But afterwards we thought this is too big, so we optimized the size.
Now this wooden panel is about 40 cm x 35 cm. This dimension has barely enough space for many sensors (camera, ...) but is sufficiently light to give more dynamic driving behavior to the car. Even though the car is not as dynamic as a little racing model and behaves more like a little tank.

After that we wanted to fix the motors on this wooden panel. We took four cubic blocks of wood as spacer between the plank and the motor. Doing this gives us enough space to fix the batteries on the bottom side of the plank. We screwed the said blocks in each corner of the panel and on each block the motors. An additional advantage is the possibility to make the motors steerable.

Every motor was measured about his dynamic and static behavior. We distributed the motors int a way that a 'slower' can be compensated by a stronger one. Especially the difference in maximum speed is clearly visible in a case of a wrong distribution. Fixing the four wheels on the motors was in contrast very easy. Therefor we only had to fix one screw.

Then we considered the distribution of the remaining components. We kept an eye on not loosing too much stability with the many bore holes. We placed the H-bridges in the corners due to the needed air cooling. Next to each H-bridge is the corresponding FPGA board. During the development process the interconnection between those boards changed several times. We are still not happy about the stability of the used cables. Maybe someone can make the connectors of the cables stronger.

The batteries provide around 11.5 Volts but most of the electronic runs on 5 Volts. In the first version there were four linear voltage converters one for each FPGA. Those small components are barely strong enough to handle currents about one Ampere. Since one FPGA board needs at least one Ampere and we haven't talked about the H-bridges by now, the voltage converters became very hot. This made our decision to place the power supply in the middle of the panel not very well thought-out. The DC/DC converters, which shipped in in February, solved that problem.

So there was not any more space on the panel. That's why we needed to make one more floor. For that we decided to use steel sticks as spacers. For the second floor we also took a wooden panel. But this one is much smaller. It isn't as thick as the panel of the ground floor and is so lighter. The size is about 20 cm x 25 cm.

From the bottom up we fixed the LAN switch on the second wooden panel. There was no possibility to screw it so we had to glue it. The LAN switch is connected to the four Ethernet-UART-bridges via LAN cables. These given cables were too long so we contracted them. The four Ethernet-UART-bridges are also glued at the smaller wooden plank from the bottom up.

On the top of the smaller panel is the central ECU board (Linux-PC). It is connected with the LAN switch via one LAN cable. Therefor we also had to bore a hole into this panel.

At the end we got two ultrasonic sensors. We fixed them at the ledges of the big wooden board. One ultrasonic sensor is in front of the car and one ultrasonic sensor is at the backside of the car. Each is connected with one FPGA board.

4 Software Architecture

4.1 Our Aims

We want to reach these architectural aims:

1. hierarchical and distributed system
   (e.g. separated Motor-Control)

2. self-maintaining car
   (PID calibration, no hardcoded constants, ...)

3. simple programming of the master-controller (Linux-PC)

4.1.1 Hierarchical and distributed system

The system has to be distributed because of the given hardware structure. The system functionality is distributed over the four Nios2 embedded processors running inside the FPGA boars and the central processor board. All in all we have five operators participating in a star like architecture.

The functionality is not only distributed horizontally (over the hardware) but also vertically. This means that there are at least five abstraction layers:

- user
- central processor board
- Nios2 layer
- programmable hardware IP-cores in the FPGAs
- hardware

The program running on the Nios2 processor can only interact with the IP-(Intellectual Property) cores and can exchange some data with the central processor board. The communication between Nios2 processors and the central processor must be limited because of the small data rate of RS232.
This limitation leads to a higher responsibility of the lower layers. The Nios2 is the only one who can control speed and sensors which are directly connected to the FPGA. The central processor can at least set the desired speed but has no influence on the controlling itself. Same counts for all layers.

To sum up, the distributed system gets the best out of all operators and the hierarchical structure gives us the possibility to limit communication traffic on the network.

4.1.2 Self-maintaining car

The car should be able to 'explore' itself. Therefor no constants except for the hardware required are coded. One example is the PID calibration. The user has the possibility to calibrate the PID controller or use the PID values from the last measurement (not implemented yet). This is especially interesting if the car weight has changed (more/less sensors).

4.1.3 Simple programming

It would be great if the central processor is simply programmable. For this issue, we use Debian as a well known operation system. You may be able to use more abstract languages for path finding and planing such as Prolog.

We also started a 'Wizard' to generate the needed c files for networking. It haven't been completed by now.

4.2 CMW-Unit SW design

Processing Unit: *Nios II* embedded core.
Main tasks:

1. Controlling the motor-speed (PI-Controller)
2. Communicate with Central-Linux-PC
3. Polling the sensors

Doing this tasks in a **hard timed cycle** as required for a real-time system (see 5).

4.2.1 Start sequence

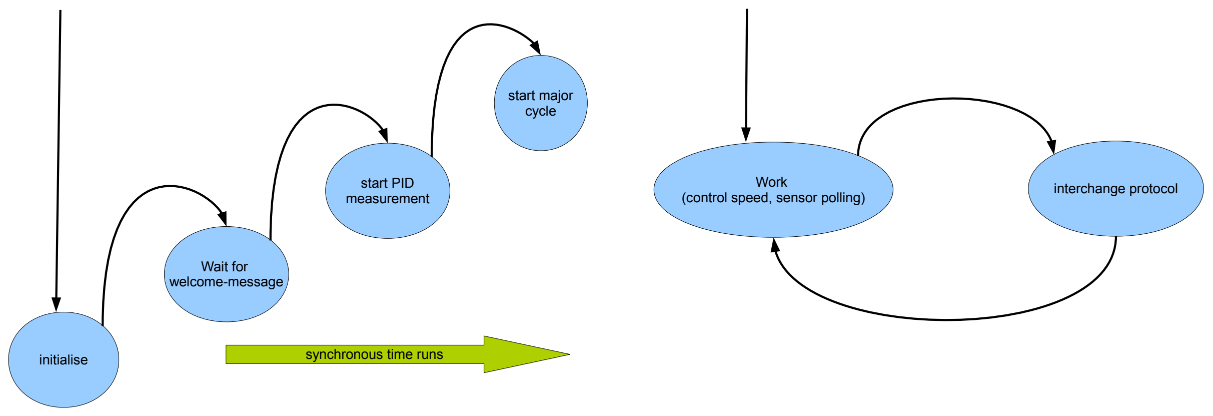


Figure 5: Left: Start sequence. Right: Major (or big) cycle.

States described in detail:

- Initialise: Set global fields to zero.
- Wait for welcome-message:
  Wait for the first WelcomeMessage. This is necessary to synchronize the Motor-ECUs. After this point every Motor-ECU should be nearly (theoretical clk) synchronous.
  Note: Due to the nature of TCP over Eth and the Eth2UART there is no guarantee that the ECUs work synchronous but the probability to do so is very high.
- Start PID measurement:
  Gets the P-, I- and D-Values for the PID controller by measurement or from the central ECU (Linux-PC) (second option not implemented yet).
  Note: The motors have a PT1 behavior so there is only a PI controller needed. The D-Value will be ignored (see also 13).
- Exit on fail:
  On every fail the speed is set to 0.
  Important Note: The central ECU is only informed about the failure because it gets no response from the Motor-ECU. Worst case (if you want to stop also the other motors) is

  1. CENTRAL to FAILED MOTOR: regular messages
  2. after 100 ms no response
  3. CENTRAL to all: stop
  4. after TCP / UART delay: motors try to stop
  5. after braking distance the car will be stopped

  The whole process can take about 500 ms!

4.2.2 Major cycle

Cycle of exact 100 ms (+- a few clock cylces). Every run of this big cycle consists ten smaller cycle runs (see 6). Task of the big cycle is the network communication which is hard timed to the 100 ms. This means that every 100 ms there should be a new request from the central ECU and the last request is answered. A message round-trip will take aprox. 100 ms + 2TCP/UART delay. Don't forget that TCP/UART is not real-timed!

Each of the small cycle runs controls the speed and handles one message (doAction) or waits for a new packet. Note: The big cycle does nothing but merges the small ones!

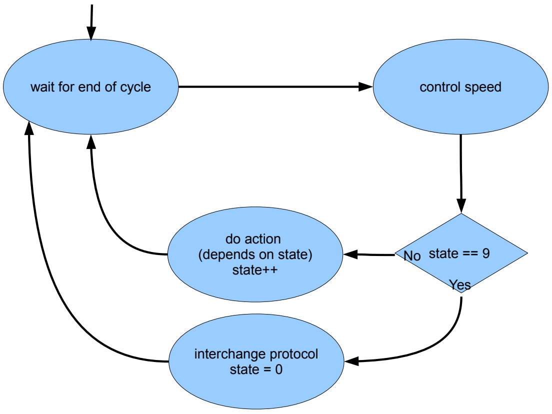


Figure 6: Small (or minor) cycle.

4.2.3 Small cycle

States described in detail:

- wait for end of cycle:
  The big cycle takes always 100ms, so the ten small cycles take each 10ms. At the begin of a small cycle run a timer is set to this 10ms. WaitForEndOfCycle will block until this timer reached 0. After that the timer is automatically set back to 10ms.

  Note: The hard timing only works if the rest of the small cycle (including the doAction() method) doesn't take longer than 10 ms. There should be even a security window of 1-2ms!
  Don't forget that 'control speed' itself takes some of the 10 ms. So said there are only 4-5 ms remaining for your doAction()!

- control speed:
  The speed measurement takes place during the whole small cycle run. To get reliable measurement results this state has to be every 10 ms! It finishes the current running measurement, starts a new one and calls the PID controller to decide what the next speed should be. This new speed is given to the motor driver.

- do Action:
  Calls the doAction() method of the message with position [getMessageCount()-state-1]. This is the reverse order. The last message in the packet is the first one handled.

  Use this ordering according to this:

Message position	Handled at	Commands in the message	Results of the commands
First message	last	done after some delay	the most current
...	...	...	...
Last message	first	done immediate	out-dated because of delay

  The only exception of this rule is the VelocityMessage. If this message is the first in the packet (and it has to be the first!) it will be handled a number of times. The first time immediately after receiving the new packet and every time when 'control speed'. This functionality guarantees that the desired speed is set immediately and the current speed is up-to-date!

- interchange protocol:
  Receives a new packet and sends the answer of the last packet. Here is the highest possibility for an error which leads to the exit of the main function.

4.3 Networking

Every data communication uses telnet over TCP/IP. This is fixed by the use of Ethernet-to-UART-chips we got from our advisers. Please note that this networking structure is not real-time ready and has a really long round-trip-time!

The physical architecture of our network is a star and looks like 7.

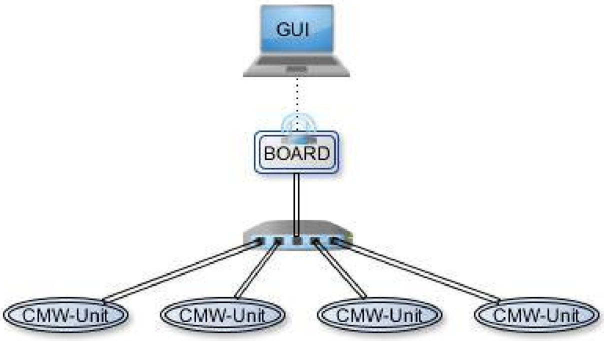


Figure 7: Star architecture with four CMW-units and one central processor board connected via LAN switch. The central board is itself also connected via Wi-Fi to a possible user.

4.3.1 CarProtocol - Packet of messages

We developed and use our own communication protocol named CarProtocol to recognize data structures in the telnet stream. CarProtocol is organized as packet of messages. The total structure is shown in 8.

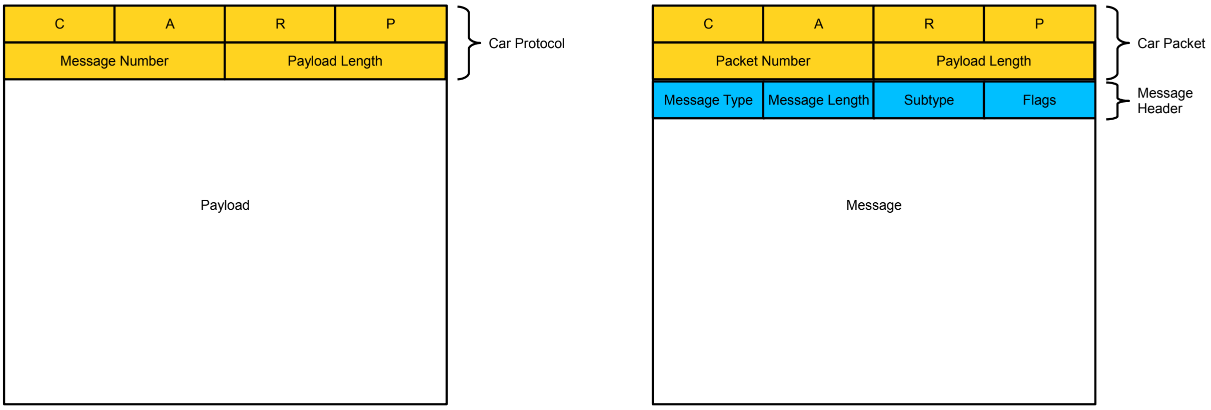


Figure 8: Left: CarProtocol header. Right: CarMessage header.

Packet fields:

- 'C' 'A' 'R' 'P':
  Start sequence of the packet to detect a packet in the incoming byte-stream.
- PacketNumber:
  Consecutive increasing 16 bit number which works as an ID. The number is never changed by the Nios2, only by the central ECU (Linux-PC). Thereby the central ECU can detect a protocol fail if the response packet has not the same PacketNumber as the request packet.
  Please remember the wrap around after the PacketNumber 65535!
- PayloadLength:
  Total length of the payload in Bytes. This length information contains not the packet-header length!

The payload contains at least one message and max. 8 messages. All messages start at a multiple of 4 assuming that all messages have a multiple of 4 as length. There must be no gap between two messages!!

The order of the messages in a packet is important:
Contains the packet a WelcomeMessage this message as to be the first. All other messages in this packet will be ignored! Otherwise the first message is the most important followed by the second and so on... The meaning of 'important' was already described in section 4.2.3.

Note: As the velocity-message is the most important in normal mode it will be assumed to be the first message!

4.3.2 CarMessage - Requests and answers

A CarMessage is the atomic communication part. The central ECU writes some (request) messages and packs them into a CarProtocol packet. The packet is transferred and read by the Nios2 processor. Every message is handled in one of the small cycle runs. Handled means that the message activates a sensor. This sensor fills the empty fields of the message. We called this filled message answer although it looks exactly like the request.

Every sensor understands at least one message type. So we send only those messages to a specific FPGA which can be understand by a connected sensor. For example: Only one FPGA has an A/D-Converter so only this FPGA gets a packet with an ADCMessage.

A message consists of a message-header and a message-body. The message-header is implemented in a superclass. Due to different body-types the body must be implemented in different subclasses.
The Structure of the message header is shown in 8 and described in the following:

- Type is the ID of the message class. The type follows this rule:
  - Types between 0 and 3 are for NETWORKING (such as WelcomeMessage)
  - Types between 4 and 7 are BASIC messages and have to be understood by every networking client
  - Types between 8 and 255 are GENERAL purpose messages
- Length = Length of header (always 4 bytes) + Length of PAYLOAD
  The Length of request and answer should always be identical!
- SubType:
  If a sensor requires more than one message type, SubType distinguishes between this messages. Is there just one message type this field should be 0.
- Flags: Bits are numbered from 0 (least significant bit) to 7:
  Bit 0: Request(0) / Answer(1)

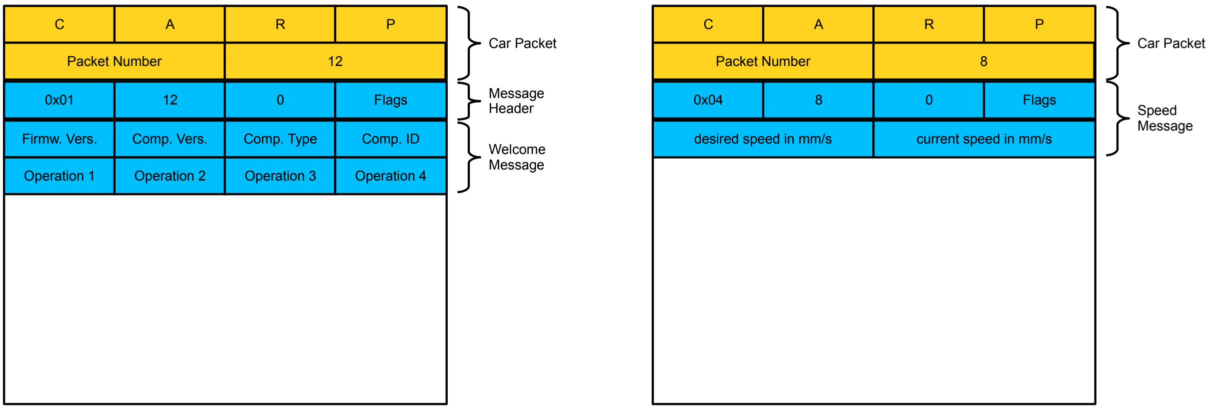
4.3.3 WelcomeMessage

One example of a CarMessage is the WelcomeMessage. This message has two aims:

1. Synchronize the 4 Motor-ECU with the central-ECU and
2. inform the central-ECU about the messages which can be understood by this Motor-ECU.

The central-ECU sends (at the same time) a WelcomeMessage to each Motor-ECU. The Motor-ECUs answer with the same message but additional with a list of those messages which are available at this ECU. The list is marked with Operation 1, Operation 2, ...

The message has the structure given in 9.

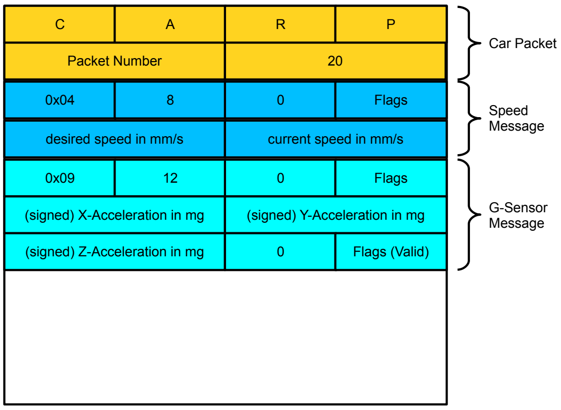


C	A	R	P
Packet Number		12	
0x01	12	0	Flags
Firmw. Vers.	Comp. Vers.	Comp. Type	Comp. ID
Operation 1	Operation 2	Operation 3	Operation 4

C	A	R	P
Packet Number		8	
0x04	8	0	Flags
desired speed in mm/s		current speed in mm/s	

Figure 9: Left: WelcomeMessage. Right: CarVelocityMessage.

Only those MessageTypes are necessary to list in the WelcomeMessage with a type >= 8. The list has to be filled up with 0x00 if less then 4 additional messages are understood.



C	A	R	P
Packet Number		20	
0x04	8	0	Flags
desired speed in mm/s		current speed in mm/s	
0x09	12	0	Flags
(signed) X-Acceleration in mg		(signed) Y-Acceleration in mg	
(signed) Z-Acceleration in mg		0	Flags (Valid)

Figure 10: Two messages in one packet.

4.3.4 Another example: CarVelocityMessage

For another example see 9 right and 10.

4.4 Central ECU: Networking interface

4.4.1 Overview

We decided to separate path-planning and communication. Therefore we developed the system structure given in 11.
We are using two threads in the process. The first one is the main thread driving the user- or agent-program. The second one drives the networking. The aim is that the user can use a struct or class which represents the current state of the car.

The said state consists of:

- General information, such as version, connected sensors, ...
- Motor information, such as current speed, max. speed, PID-values
- current state of the connected sensors

The current state of the connected sensors is represented in an own class (see below).

4.4.2 Network Thread

This thread initializes the communication (see WelcomeMessage) and polls information from the car. With the received answers it updates an thread-internal state. But how does this thread generate the necessary messages?

The answer of the WelcomeMessage contains a list of the available sensor types (message types). The network thread generates a sensor representation of each sensor type. This representation derives from the super-class called SensorState. So we know the sensor types. How about the messages?

SensorState has two important methods:

virtual CCarMessage *getCarMessage()

virtual bool updateSensorState(CCarMessage * p_message)

These methods are used to generate the necessary messages and to update the sensor state with the received answer.

So the main task of the networking thread is collecting the messages from the SensorState objects, packing them into a packet and sending it to the motor-ECUs. After receiving a packet, it is spitted back and every sensor is updated with the answer. See 12 for the complete communication chain.

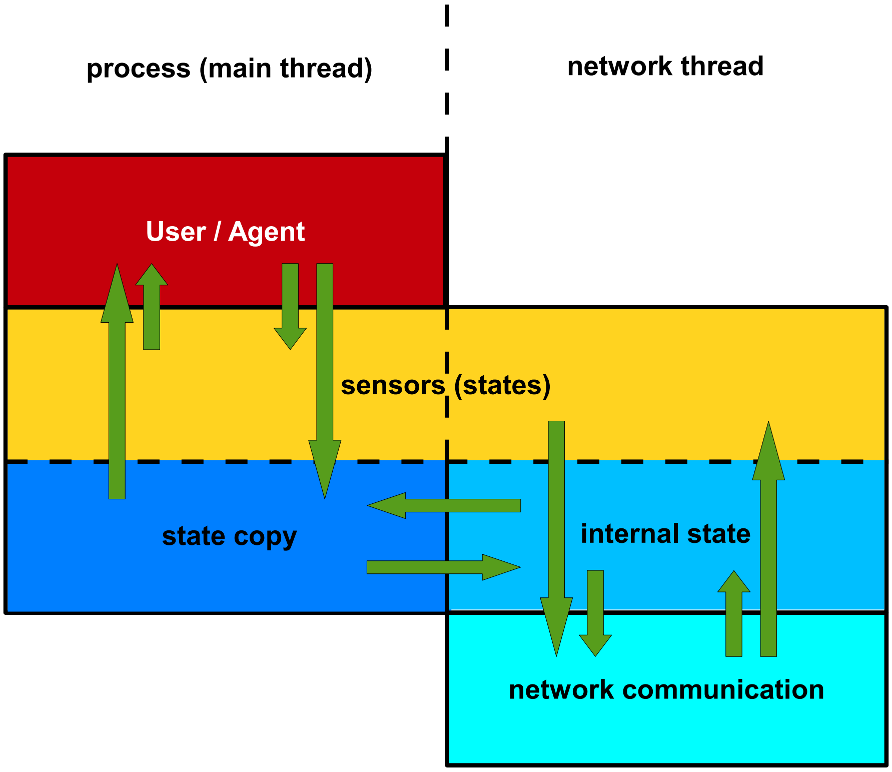


Figure 11: The software structure on the Linux-PC.

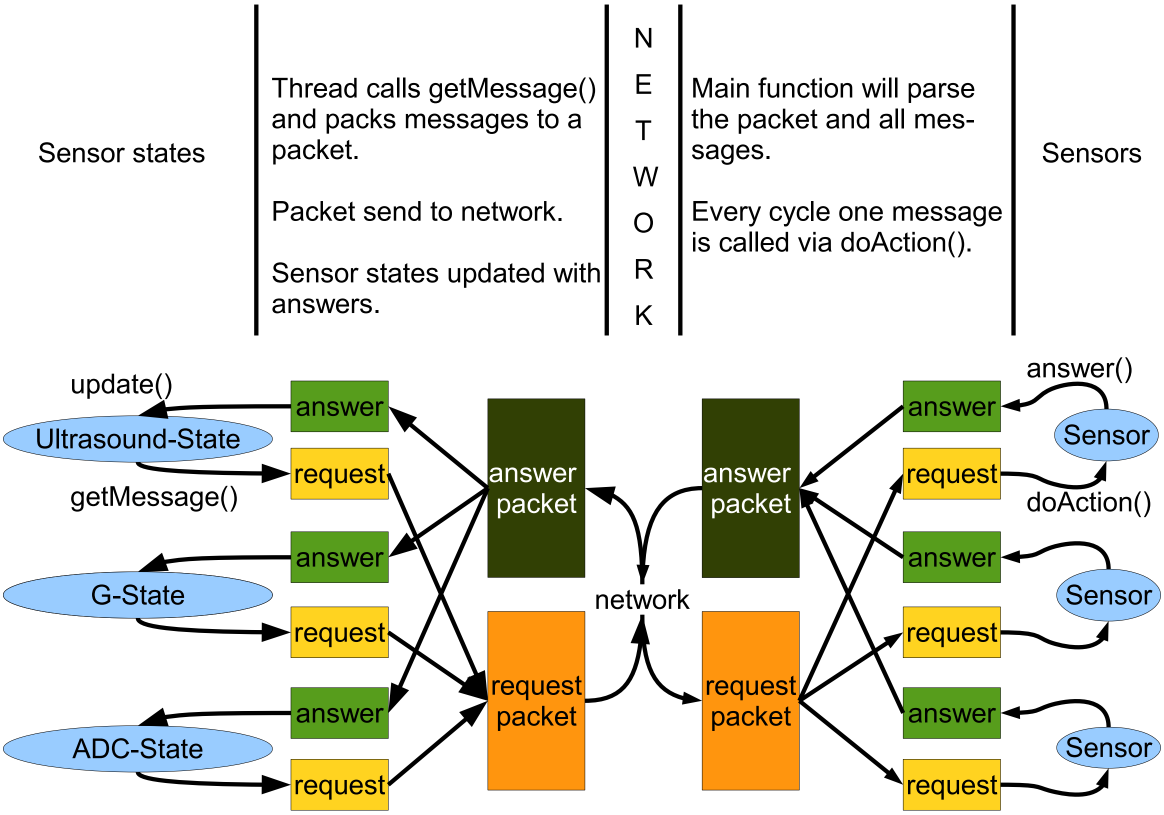


Figure 12: The sensor structure and message chain in the whole project.

4.4.3 Main Thread

The main thread can get a copy from the said internal state. The used function is (of course) thread safe. The copy will be done with memcpy(..).

Please Note: The pointers to the sensor states are always the same as in the internal state. So the sensor states are updated automatically, but the static information (current speed, ...) not. The sensor states must be thread safe as well!
Maybe someone can introduce copy-constructors on the sensor states to solve this issue.

So the proper way of controlling the car is the following:

1. Call startConnection() // Enables networking and starts new Thread
2. Allocate space for one Car_State object
3. Call getState(..) with the said object. You will get a copy of the current state
4. Call the sensors (Car_State.motorStates[i].p_sensors[j]-¿get....()) for additional state information (distance, ADC values, ...)
5. Use the information to do something.
6. Get a more actual copy of the internal state by calling getState() once again.
7. Set something in this copy to a new value (e.g. set a new desired_speed).
8. Call setState(..) to transfer your changes.
   It takes some time that the car adopts the new values (see message delay).
9. Return to step 3. or call stopConnection() before program exit.

4.5 Central ECU: User-/Agent-View

Processing Unit: *Sabre-light i.mx6* development board (4 cores).

Main tasks:

1. Control the speed-controller
2. (Polling the sensors)
3. Calculate next behavior

4.5.1 Software Architecture

- Multi-Process / Multi-Thread
- seperated Behavior-Planning and Network-Communication

4.5.2 Current setup

- Main application:
  - 1. Thread: network-communication
  - Main Thread: behavior-planning (exploration mode)
- dhcp-server, os, much more ;)

4.5.3 Current behavior

- The car should be driving in one direction.
- Is there an obstacle it turns left until the obstacle is not in it's way anymore.
- Then the car drives further in this direction.
- If car is in danger (see wall) then the car will speed down.
- If the distance between car and obstacle is lower 20 cm then the car will stop.

4.5.4 Possible behavior

Remote control mode:

- Human user controls movements via GUI
- GUI sends data to a hidden server (not NSA)
- If car is in danger (see wall ;)) then the car will speed down.
- If the distance between car and obstacle is lower 20 cm then the car will stop.

Other possible behavior:

- ABS, ESP via G-Sensors (data is already available)
- Web-Cam for little NSA-agents ;)
- ...

5 Speed Controller - PID

The motor and the hall-sensor together show PT1 behavior. An example for the curve is given in 13.

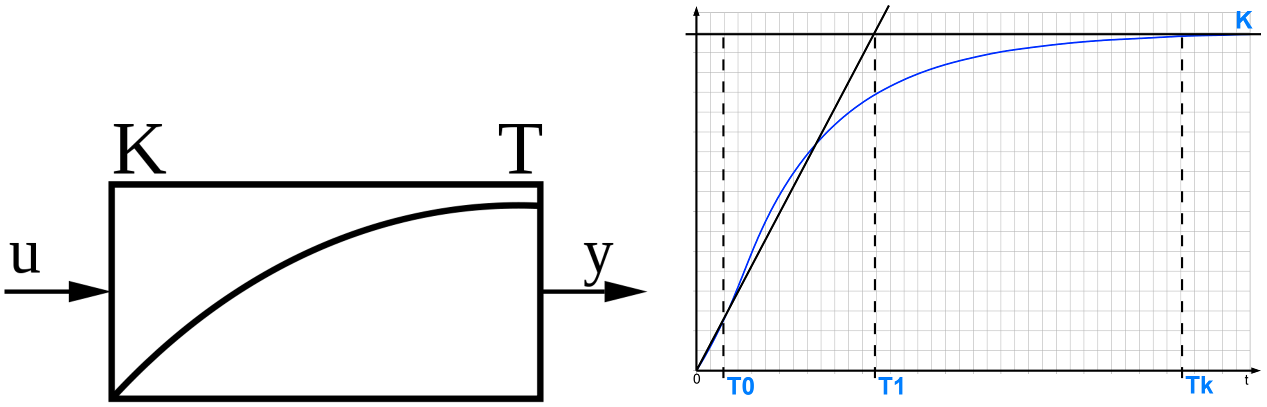


Figure 13: Left: PT1 behavior in time domain. Right: Example for a measurement of motor + hall-sensor.

To control a PT1 system you only need a PI controller. A D-value is not necessary or will have no impact on the control value.
The control-circle will be like 14.

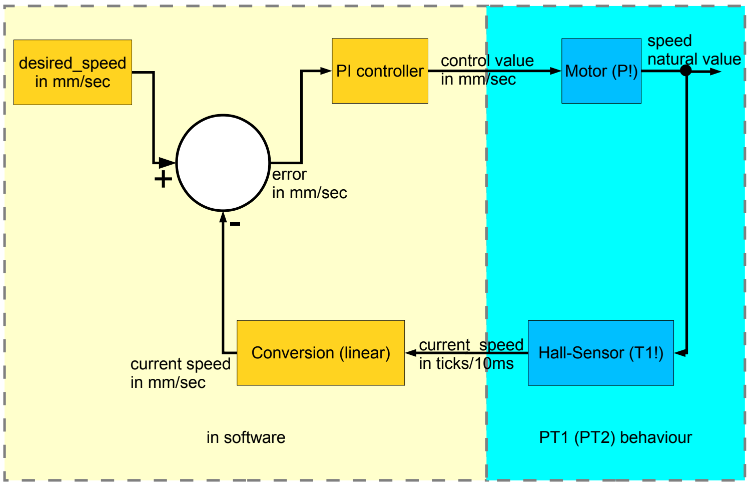


Figure 14: Control-circle with PI controller.

The exact transfer-functions for the motor and the hall-sensor are unknown. We assume that both have PT1 behavior. The motor dominates with his P and the sensor dominates with his T. Together they have a dominating PT1 behavior but in the strict sense total a PT2 behavior. You can see the PT2 behavior in 13 (right). The tangent at the origin hits the curve two times. In an PT1 system this should never happen, so we infer from that PT2 behavior.

Thus we can never guarantee that our PI controller produces no overshoots. The probability for this scenario may be very low and the overshoots also.

The controller is written in software. This causes a high discretization (error). The speed is not controlled continuously but every 10ms. With this errors and delays the PT2 behavior becomes more unstable. We recommend a (hardware) implementation in the FPGA. This will also not be fully continuously but will control the speed more detailed.

All together the system has PIT1 behavior. This means there is still a remaining delay for acceleration and slow-down!

6 Workflow and guidance

6.1 Nios2 workflow

The Nios2 processors are programmed similar but always a little bit different. The program depends on the corresponding CMW-Unit and especially on the connected sensors. For example the left motors have to move in the opposite direction than the right ones to let the car move forward. Another example is that only two FPGA boards are connected to the ultrasound sensors.

The differences are collected in the 'motor_control/properties.h' file. You may have to change the following constants:

INVERTED	One side has to be 1 and the the other side 2.
AVAILABLE_OPERATIONS	Array which contains the (message-)types of all messages which can be understood by this build. Only those types are included that are optional (so said with a value >= 8). Currently only max. 4 message-types are supported. All 4 array elements have to be filled with a value. Are there less then 4 messages the rest should be filled up with 0x00.

6.1.1 Programming the device

You may use this workflow to program the Nios2 processors:

1. Start the NIOS II Eclipse Plug-in
2. Connect all batteries.
3. Connect one of the DE0-nano boards with the USB cable to your laptop.
4. Change INVERTED to the necessary value.
5. What sensors are connected to this FPGA? Don't forget the sensors which are directly on the board (such as GSensor).
6. Look up the message types of the collected sensors.
7. Fill up AVAILABLE_OPERATIONS with this message types. Please note that only those types have to filled in with types >= 8.
8. Generate and load the .elf file to the FPGA.
9. Continue with the other FPGAs at step 3.
10. After you loaded all four FPGAs turn the car head down. Start the central ECU.
11. The central-ECU sends the WelcomeMessage and MotorMeasurementMessage. The wheels will start tuning with full speed (thus the head down).

Maybe the advisers can activate the flash memory to start the motor program on power-up automatically!

6.1.2 Tour through program files

The program files are organized as the following tree structure:

- *docu*: the documentation incl. the presentations
- *firmware*: all software files
  - *main*: the 'main' branch of the files
  - *test*: files and programs for testing
  - *tools*: tools such as code-generators, ...
- *hardware*
- *etc*

The main folder itself contains the following structure:

- *application*: the projects and make-files
- *car_control_linux*: main files for the central processor (Linux-PC)
- *car_control*: a little outdated version of above but for Windows
- *etc*: utility libs
- *external_drivers*: sensor and communication drivers
- *motor_control*: main files for the motor ECUs
- *networking*: CarProtocol, CarMessages and derived messages for the sensors
- *sensors*: files which represent the state of a sensor
- *terasic_lib*: library files for the DE0-Nano board
- *web_interface*: code and content files for web services

6.1.3 Guide to the ComBuilder

You can find the guide to the ComBuilder code-generator under *firmware/tools/ComBuilder/docu*.

6.2 Central ECU workflow

1. The central ECU boots the Debian OS from a SD card. This may take a minute or two.
2. After booting the system opens a Wi-Fi-Accesspoint named 'SafetyCarNet'. You are now able to connect to this network.
   Passphrase: safetycarnet
3. The dhcp service takes very long (1 - 2 minutes) to give you an IP address. Please check your system IP if you are really connected to the network!
4. Use a SSH client to connect to the terminal. A useful client for Windows is 'WinSCP'.
   Rootname: root Password: root
   Username: board Password: board
5. copy the changed code files to the corresponding location (under root/SafetyCar/firmware/main/).
6. change current directory to root/SafetyCar/firmware/main/application/car_control_linux.
7. build program with: make
8. check build result
9. start program with: ./car_control

There is also an Apache-Webserver running in background. The web-files are under the standard path for this server type in the folder www.

6.3 Maintaining the Ethernet-to-UART chips

The Ethernet-to-UART chips have the following addresses:
192.168.0.11, 192.168.0.12, 192.168.0.13, 192.168.0.14

Just connect your laptop via a LAN-cable to the switch and change your local IP(v4) address to 192.168.0.200 (subnet: 255.255.255.0). You can now enter one of the addresses above in your browser and get the config-webpage of the corresponding chip. See the manual of the chips under docu!

6.4 Additional help

If you are in need of additional help, feel free to contact me under florian.hisch@tum.de!

7 Summary

At the end of October 2013 we started with the Lab Course SW-Co-design with (Lego)Cars. We got the task to design and constructed a car model during our lab course HW/SW-Co-design. The main aim on this was to design a distributed software architecture. This architecture consists of four FPGAs and a central Linux-PC.

The car itself has a tower-like structure with different sized levels. These levels are made out of wooden planks which are robust enough but not too heavy in order to the motor power. As pillars between the levels we took metal and wooden sticks.
We started with the biggest plank (plywood) and step by step we fixed all components on it. We wanted to assemble the car as fast as possible, but the components shipped in only in small portions. Hence we also developed the software accordingly.

Our first aim was to make the car drive, controlled by a keyboard. Then we wanted to make the control more comfortable and start to play with Wi-Fi. At this point we want to say 'Thank You' to our partner team, which helped us with our Linux-PC and the Wi-Fi very well!

To get some intelligence into the car control we plan to make our car stopping if something is crossing the car's way. So ultrasonic sensors were fixed at the front and at the end of the car.

We also played with a simple 'exploration mode'. The car should be driving in one direction. Is there an obstacle it turns left until the obstacle is not in it's way anymore. Then the car drives further in this direction.

8 Conclusion

8.1 Prospects

Now the car is working. But there are many possibilities to upgrade the car. You could install a camera or something like that at the top of the car to improve the autonomy of the car. You also could make the wheels steering like in an ordinary car. So as you can see there are many things which would be additional. But which project is finally finished? So we are content with our result like it is in this moment.

8.2 Closing remarks

All in all it was a very interesting project. We had a lot of fun ;)