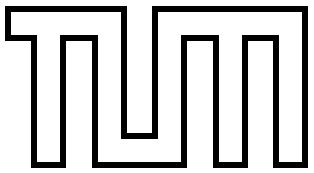


FAKULTÄT FÜR INFORMATIK

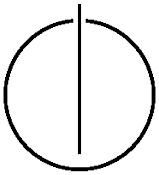
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Practical Course Report

HW/SW co-design with a LEGO car

Car2X Communication

Hagen Schmidtchen, Paul Bergmann,
Johannes Windelen, Florian Janssen



Contents

Chapter 1

Introduction

1.1 Motivation

In modern road traffic, methods guaranteeing the safety of the driver and increasing traffic efficiency in general are an important issue of research. A possible approach is to establish communication between cars (Car2Car) or between a car and a base station (Car2Infastructure), exchanging information about the status of the car or traffic in general. Cars could for example signal each other sudden events such as emergency breaks and base stations could give information about congested roads. Car2Car and Car2Infrastructure are commonly generalized with the term Car2X.

1.2 Problem description

Our task was to implement a Car2X communication system on the metal car of last years group. External devices such as other cars or base stations should be able to wireless connect to a server and communicate with it, requesting information about the state of the car or sending certain commands, such as the immediate performance of an emergency break.

1.3 Approach

The core of the system is implemented on an Altera FPGA board which is running two cores in parallel. One of them is responsible for communication, establishing a TCP server to which external devices can connect to through a wireless network set up by a WiPort. Necessary calculations are then performed on a second core. All messages follow our own Car2X protocol, which extends the CarProtocol introduced by the last years group.

Chapter 2

Concept

2.1 Overview

Johannes

2.2 Hardware structure

The hardware is design to allow an easy exchange of components and extensions. A FPGA is used as main computation unit. All components used by the car are connected to this central unit. The communication between the car's components uses ethernet. This is also used for car2x communication through a wiport. Every wheel is controlled by a single nano FPGA, which uses an UART-to-ethernet converter to communicate with the main FPGA. A schematic structure can be seen in figure 2.1.

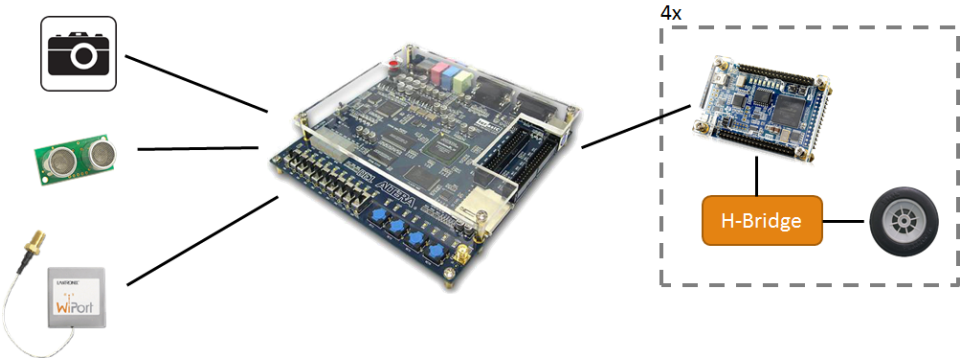


Figure 2.1: Schematic hardware structure

2.3 Car2X Protocol

The CARP protocol"(developped by a previous lab-course group) was extended by a set of messages to let the NIOS2 processor system fit into the role of the communication central of the car. One NIOS2 core is handling the internal state of the car and the other one is in charge of all the communication between internal car parts and external communication partners like other cars or car2x stations.
The internal communication protocol is based on the work of Florian Hisch and has nearly not been modified. The only major change was a role switch in terms of server-client relationship. Unlike before, the central unit now is a server to which external clients can connect to. Keeping that in mind, there now is an external communication interface featuring the so called "Car2X-Messages" as an extension of the CARP protocol. Each message of this type, which is sent to the car, will be answered by the car after being processed or getting outdated.
All this behaviour is handled on the "communication-core" of the NIOS2 system. By parsing in the messages in the socketserver.cpp file directly from the incoming TCP/IP byte stream, there is a new message object created for every new message. After that, the "sss exec command()" function handles the received message by reading out the message type an then taking various steps depending on the specific message.

Chapter 3

Hardware

This chapter describes the basic concept and configuration of the car-system's hardware. A basic knowledge of Altera Quartus II is assumed. This chapter only describes the hardware configuration of the central FPGA, since the DE0-nano boards have been configured by the previous years group.

3.1 Topology

To keep the system as modular as possible, every task needed to control the car[1] has it's own nios2-core. The cores communicate with each other using a share memory area of the onchip memory. The memory access is controlled by a hardware-mutex. Beside the share memory, every core has it's own memory. The control-core uses onchip memory for this, while the communication core has to use the SDRAM on the board, because the code will not fit on the limited space of the onchip memory. For communication with the other components of the car as well as the car2x communication, the responsible communication core uses one of the board's ethernet controllers. For programming and debugging purposes, both cores are accessable through a JTAG-UART. The topology is shown in figure 3.1.

3.2 Configuration QSYS

The major part of the hardware configuration is done in the QSYS tool from altera. This projects configuration has been created using couple of altera tutorials and combining them. The part used for the ethernet configuration has been placed in a seperated QSYS file to clean up the configuration slightly.

[1] here communication and car-control, not every thread

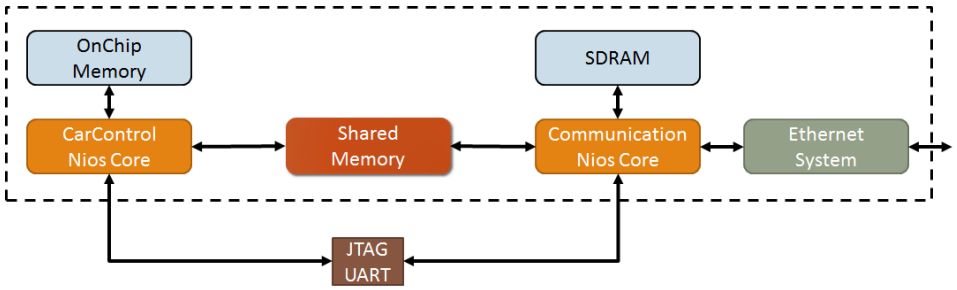


Figure 3.1: Schematic hardware structure

3.3 Configuration Top Level File

The top level files combines all subcomponents need to compile the system. This also connects the components, e.g. the QSYS-generated file with the in and output ports of the FPGA as well as additional components used by the ethernet of SDRAM controller.

3.4 Problems

Probably the greatest problem where the altera tutorials, since they are designed to fulfil only the one task and combination or integration into a bigger system is not supported or described. For example the system clock needed for the ethernet controller to work is higher then the clock used in the sdram tutorial. And since the tutorials should be kept easy, a prebuild component is used with a hardcoded clock in it, which leads to timing problems and system crashes. Also the explanations in the tutorial, especially regarding the top-level file are kept to a minimum, by providing code to be copy-pasted.
Also due to timing problems it seems not to be possible to use one block of sdram for both cores. For that reason the control core still uses onchip memory.
Another problem was the limited debugging. The debugging function of eclipse is not working at all and the default console also sometimes produces errors. To bypass this, disabling the eclipse default console output is a good idea and using the nios2-terminal instead. This also allows debug outputs from both cores simultaneously. Because of this limited debug possibilities, it was quite hard to find if the error came from hardware or software, or even from a broken board/malfunctioning hardware.

3.5 WiPort

3.5.1 Configure the WiPort

The first step to configure the WiPort is either to connect to either its predefined wireless network or directly plug in a cable directly. Then access the IP 10.10.100.254 in a web browser (standard username: admin, password: admin). Here the WiPorts security settings, static IP, etc. can be configured.

3.5.2 Our projects WiPort configuration

In our setup the WiPort is given a static IP of 10.10.100.100 (this IP is used to access the WiPort from a wireless network) and 10.10.100.101 (this IP is used to access it via cable). In fact those IPs don't really matter in our project as they are never directly used in our code. The WiPort is configured in client mode, so it is invisible in the network (as it is never addressed directly) and merely serves as a tool to establish communication between the server (the board) to the external clients. The port number used for communication is throughly set to 23.

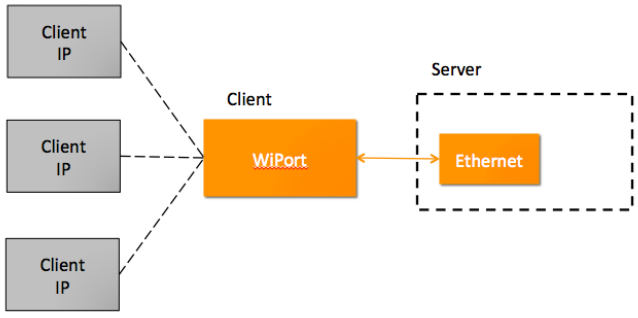


Figure 3.2: Integration of the WiPort

3.6 Circuit diagrams

Johannes

Chapter 4

Software

4.1 Communication Core

4.1.1 Basic Functionality

The main task of the communication core is to establish a TCP server on the DE2-115 FPGA board. This server should be able to accept multiple incoming client connections from the car such as the wheels nano boards, ultra sound sensors, and connections from the WiPort as for example the camera. Messages from those connections must be received, parsed and executed. This section will give a basic overview of how this functionality is implemented and structured in our code.

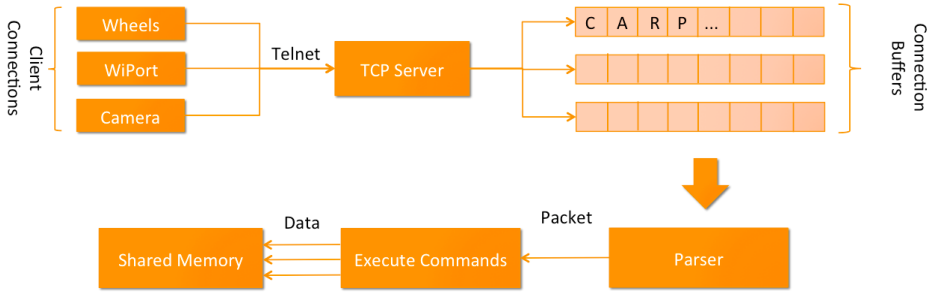


Figure 4.1: Communication flow overview

4.1.2 The main file

The main file starts up the threads of the core. As threads are not natively supported we are running the MicroC operation system. Also we are using it to run the Nichestack TCP/IP stack used for communication (via Telnet). Threads are implemented using the MicroC OS. Threads started up are the following:

- Socket Server (implements our functionality of the control core, setting up the server)
- Inet Main (OS setting up Nichstack TCP/IP stack)
- Clock Tick (OS specific task)

4.1.3 The socket server files

Main loop of the socket server

On startup the socketserver task does some initialization and sets up a TCP server using the specific commands of the Nichestack TCP/IP stack. Then it enters its main loop where each cycle it performs the following actions:

- Block for specified timeout: Check weather an incoming connection is detected or data on an already existing connection is ready to be read
- In case of a new incoming connection we create a new connection struct, initialize it and append it to the list of existing connections
- In case of incoming data on a connection we receive the data into the connections buffer and check if a whole message has been received (if this is the case, the message is directly parsed and executed)
- Also for certain incoming messages it is checked if they have been processed successfully by the communication core and an answer message needs to be generated or data needs to be forwarded to the nano boards
- Additionally every cycle it is checked if a connection has been closed and needs to be removed from the list of connections
- The main loop also checks wether all four nano boards have registered themselves (see WelcomeMessage - note all further incoming data is ignored until the wheels have successfully registered)

Receiving data on a connection

Firstly all incoming bytes on the connection are written in the connections buffer. Then step by step it is checked if a full car protocol message is existing in the buffer (see definition of the message header). If a full message was received, we use the message parser of the CarProtocol class to turn the plain buffer into the specific CarProtocol object. This object is then sent to the method that executes all messages.

Execute a command

Executing a command means establish communication with the car control core which is running in parallel on the FPGA. So the first thing to invoke is a command to access the shared memory and read the current car state (this command is blocking so the other core is denied access to the shared memory). After that all the car messages contained in the received car protocol are handled one by one, reading and manipulating the car state and write the results back to the shared memory (for more detailed information what is read / written by the different messages see the message definitions in this document). After the execution of the message an answer message will always be generated to inform the client weather the command has been performed successfully or not. In case the shared memory only has to be read the answer messages with the requested information can be generated right away. If the control core has to do some work before though, some delay is involved. Therefore a mechanism is implemented storing information about the messages that are currently being processed by the car control core. For further information on this see the next section.

Answering Messages

An answer message will also be generated if the incoming message requests the car control core to change the cars behavior according to the car state stored in the shared memory. Answers for these kind of messages cannot be generated immediately after execution because it needs some time for the car control core to process the new commands. Affected are the following message types:

- Emergency Message
- Remote Control Message
- Car Control Message

In order to check wether a command has already been processed by the car control core, information about the latest incoming message for each of those types is stored as well as two counters in the car state, one for the communication core and one for the car control core. The following mechanism checks weather a message has been processed by the car control core:

- Every time a message of the above type is received, it increments the current counter of the communication core and stores this value
- Every time the car control core adjusted itself to the new car state, it sets its counter to be the same as the counter of the communication core

- If the stored counter of a received message is smaller or equal to the current communication core counter, this message has been processed

When it is known that a message has been processed by the control core, it needs to be checked if the message has been processed correctly and also if some data needs to be forwarded to the nano boards. So the requested values of the incoming message are compared with the values in the shared memory set by the car control core. If they match, an answer indicating success is generated, otherwise a fail is reported. It can also happen that a new message of a type is received before the old one has been processed. Then the old message is deleted and answered with a flag indicating the message has been overwritten with a newer version and therefore has not been executed. In case of certain message types (see corresponding section about the specific messages) some data such as wheel speeds need to be forwarded to the nano boards.

4.2 CarControl Core

The **CarControl Core** is responsible for processing all the data that the communications core receives from other components, and calculating the correct reactions (Motor Controller signals) to that data.

4.2.1 Location

The CarControl Core software project resides in $software/Car2X\_carControl$; the $.sopcinfo$ file used to generate the board support package here: $hardware/nios\_system.sopcinfo$. The software runs on Core 1 of the system.

4.2.2 Program flow

The control core cyclically accesses the current $CarState$ from the shared memory and performs all calculations. The main loop can be broken down into four parts.

1. Get current $CarState$ from the shared memory (blocking).
2. Check the operating mode, $state.reqOpMode$. If a new one has been requested, the transition is performed and set in $state.currOpMode$.
3. Get the requested velocities $state.reqVel$ and transform them into goal velocities for each wheel. These are then written into $state.MotorECU\_State$.
4. Write the changed $CarState$ object back into the shared memory and release the mutex.

4.2.3 Operating modes

A state machine is used to restrict or alter the car's behaviour, based on the current **Operating mode** of the car. Broadly speaking, an Operating Mode places restrictions on what actors can control the car at any given time, and the velocity at which the car may move. Some Operating Modes are used internally, while others can be requested by the user.

PreOperational Initial mode, used internally during the startup phase. The velocity is restricted to 0 on all motors.

Idle This operating mode is automatically transitioned to upon successful communication with all four Motor Control ECUs. Velocity is restricted to 0 on all motors.

AutomaticDrive This mode is to be used when the car drives autonomously. The car may only be controlled from the IP address that is registered as the autonomous driving component (the image processing component in our case). The velocity is restricted to $200mm/s$.

ManualDrive A $RemoteControlMessage$ C2X message will transition the car into this mode. Subsequently, the car may only be controlled from the IP address that sent the aforementioned message. The velocity is restricted to $400mm/s$.

EmergencyStop Triggered by an $EmergencyBrakeMessage$, this operation mode will immediately set all motor velocities to 0, effectively executing an emergency stop. To exit this operating mode, either a $RemoteControlMessage$ must be sent.

4.2.4 Motor velocity calculation

The motor velocity calculations are relatively straightforward. Each operating mode has a maximum velocity that may be requested. Any velocities above this limit are scaled down to the limit. Equations (4.1) (4.2) show how the scaling is performed.

$$v_{limit} = \frac{1}{4}\sum_{i=1}^{4} v_i \tag{4.1}$$

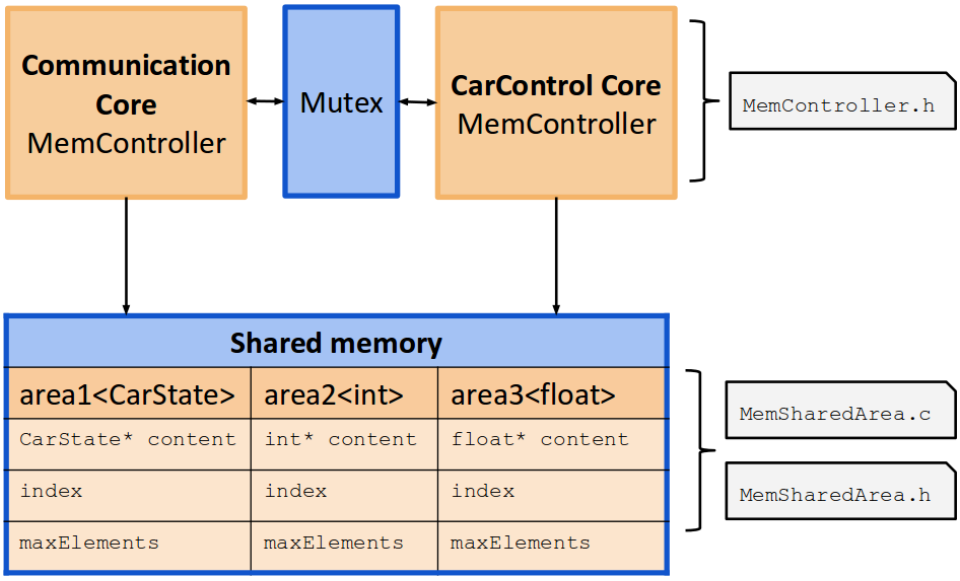
$$v_i = \begin{cases} r_i & \text{if } r_{avg} \leq v_{limit} \\ v_{limit}/r_{avg} * r_i & \text{if } r_{avg} > v_{limit} \end{cases} \tag{4.2}$$

Since the velocities are transmitted as signed 16bit integer values, integer arithmetic is used to calculate the goal velocity, at a precision of $0.01mm/s$.

4.3 Shared memory

The **Shared Memory Module** is responsible for ensuring the safe exchange of data between the Communication and the CarControl cores. On the hardware level, the **Shared Memory** and **Hardware Mutex** building blocks provide the necessary foundations that the *MemController* class leverages. Figure 4.3 shows a conceptual representation of the Shared Memory Module and its components in action.

Figure 4.2: The shared memory setup. The Hardware components are shaded in blue, the components of the Shared Memory Software Module are shaded orange.



4.3.1 Location

The *MemController* Class resides in *software/shared_files/MemController.h*. The *MemSharedArea* is declared in *software/shared_files/MemSharedArea.h*, with the variable allocation occuring in *MemSharedArea.cpp*.

4.3.2 Shared Memory Areas

The *MemSharedArea* struct (see 4.1) provides a container for managing data in the shared memory. The main goal of the shared memory is to facilitate data exchange between multiple threads/cores.

```
1 template<typename T>
2 struct MemSharedArea {
3   alt_u32 maxNumElements_u32;
```

```
4   T * content_a;
5   alt_u32 index_u32;
6   enum Bufferflags flags_u32;
7 };
```

Listing 4.1: *MemSharedArea* structure, the container for shared data. Code found at: *software/shared_files/MemSharedArea.h*

All containers are declared in *MemSharedArea.c*. Here, the containers are defined and initialised to their default values. In the following snipped, Altera directives are used to instruct the linker to place the variable *AreaCarStateBuffer* into the hardware memory component titled *.shared_memory*:

```
1 CarState AreaCarStateBuffer[BUFFERSIZE_CARSTATE]
    __attribute__ ((section (".shared_memory")));
```

Note that it is necessary to define both the *MemSharedArea* container, and the array where the actual content will be stored.

Since the *MemSharedArea* structure accepts a template argument, any data structure may be stored within the *MemSharedArea* containers, as long as it can be serialised and there is enough space in memory.

4.3.3 Memory controller

The MemController class provides an interface to access these areas in a safe and ordered fashion. The shared memory area array member is treated as a ring buffer, to allow for a historical retrieval of a certain information stream.

The primary constructor of a MemController has the following signature:

```
1 MemController<T>(MemSharedArea<T> * area_p);
```

This creates a MemController object that provides access to the shared memory area *area_p*. One memory controller is always responsible for just one memory area. Note that the template argument for both the *MemController* and the *MemSharedArea* parameter must be the same.

Currently, there is only one mutex available at the hardware level, so simultaneous access to different shared memory containers is impossible, though it would be safe. A shortened version of the API is shown in snipped 4.2.

```
1 // retrieves the newest element from shared memory
2 T MemController::get();
3 // retrieves the latest element and keeps the mutex
```

```
4 T MemController::get(bool blocking);
5 // deletes all data from the shared memory
6 void MemController::clear();
7 // stores the element T in the memory area
8 void MemController::push(T element);
```

Listing 4.2: The basic *MemController* API

When using a *MemController* object, be mindful of the mutex. Most of the mutex management is abstracted away, with it being automatically locked/unlocked as required. However, it is sometimes necessary to retrieve an element from shared memory, make some changes to it and write it back into shared memory, while ensuring that no new elements are written to the buffer in the meantime.

To do so, use the T MemController::get(bool blocking); API function. When the parameter is *true*, the mutex will not be released once the element has been retrieved. The core which executed the function will retain mutex ownership until a call to void MemController::push() is made.

4.3.4 Challenges and improvements

The main challenge of designing the shared memory module primarily lay in the generic design of the data representation in memory and of the *MemController* class, providing a simple yet powerful framework for others to use. Any serialisable data can be transmitted between threads or physical cores with the current implementation. The API is simple and straightforward to use.

To provide better synchronisation between threads accessing the same shared memory, it would be great if a timestamp/flag could be added to each element in the content buffer. This would ensure that each core knows how many new elements have been added since it last accessed the area and simplify the code when more than one core needs to both read and write to shared memory. One of the challenges with this approach is to synchronise time between the cores. A hardware component may be necessary to achieve this. Currently, the cores are synchronised using indices contained in the data structure written to the shared memory.

4.4 Motor Controller

The **Motor Controllers** are aptly named. They control the motor velocities based on a *MotorVelocityMessage* sent by the central ECU. Last semester's group (WS13/14) did the vast majority of the work on the motor controller code, all credit goes to them. Please see their documentation (found in the repository at *doc/group_ws*13_14*/documentation/documentation.pdf*).

The following is a short overview of the motor controller, included for completeness' sake.

4.4.1 Location

The software project for the Motor Controllers resides in $software/Car2X\_nanoMotorCtrl$. The hardware configuration is found here: $hardware/DE0\_Nano.sof$ with the corresponding $.sopcinfo$ file $hardware/DE0\_Nano\_SOPC\_20131216.sopcinfo$

4.4.2 Program flow

The Motor Controller software consists of two logical parts: communication and control. The communications part is responsible for sending message to and receiving messages from the central ECU. These messages include among others, the $WelcomeMessage$ (used during system startup) and the $MotorVelocityMessage$ (contains the desired velocity). The control part is essentially a PID controller that translates the requested motor velocity into PWM signals, and controls the actual velocity using the measurements from an encoder mounted on the motor. The prodedure is shown in snippet 4.3.

```
1  int main {
2    bool startup = false;
3    bool newMsg = false;
4    double velocity;
5    ControlMsg msg;
6
7    while(!startup)  {
8      sendWelcomeMsg;
9    }
10   calibratePIDController;
11
12   while(true) {
13     if(newMsg) {
14       velocity = msg.velocity;
15       msg.sendResponse()
16     }
17     controlMotor(velocity);
18   }
19 }
```

Listing 4.3: The motor controller program in pseudo-code.

4.4.3 Changes from previous semester

Only one major change was made. Previously, the Central ECU would poll the Motor Controllers during system startup with $WelcomeMessages$ until all four Motor Controllers were available. To improve the synchronicity of the system, this behaviour was switched around. The Motor Controllers now periodically send a $WelcomeMessage$ to the Central ECU. Once all Motor Controller have been registered, the Central ECU sends out answers to all client Motor Controllers simultaneously.

4.4.4 Challenges and improvements

We have been unable to use the PID Controller calibration at system startup. The code fails to find correct parameters, resulting in erratic and errorneous behaviour. As a stopgap, the PID values are hardcoded into the software. Debugging was further complicated by the lack of a debugger, thus we were unable to find the true cause for the PID calibration failure.

4.5 CarProtocol

It was part of the work of the previous lab-course group to set up a communication protocol for the car. As the communication was realized using telnet streams, they decided to add another layer over the already existing TCP/IP and telnet protocol layer. This means that the actual protocol for the car is represented by the payload of telnet messages, representing just characters.

In fact, the actual message must be parsed from the incoming telnet characters. Therefore the so called „CARP“ or „Car-Protocol“ consists of packets and messages. Every packet consists of a CARP header and several CARP messages what makes it possible to search the incoming telnet stream for the CARP header and extract all the reqired information about the messages which are part of the specific packet.

For giving a rough idea on how this structure is assembled, here is a figure from Florian Hisch, showing the header-packet-message setup. For a more detailed description have a look at chapter 4.3 of the WS13/14 groups documentation.

4.6 C2X extensions

There are basically two types of CAR2X messages. „Polling messages“ and „Triggering messages“ which have nearly the same structure as CARP messages. The only difference is that these messages are intended to establish

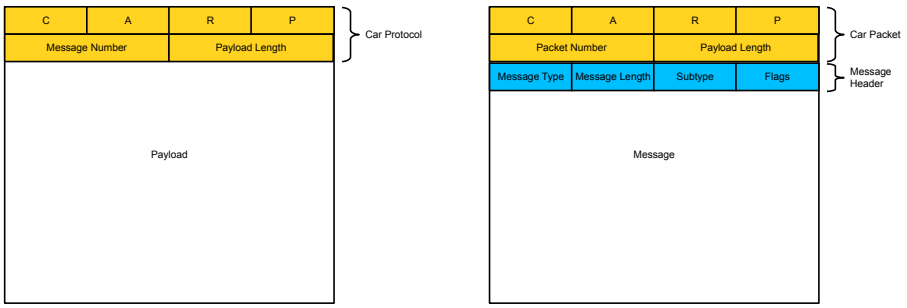


Figure 4.3: Left: CarProtocol header. Right: CarMessage header.

communication with clients other than the core-car-parts. To make it easy to identify CAR2X messages, their 8 bit type-value always has the 4 least significant bits equal to zero: 0xV0 with $0x1 \leq V \leq 0xF$.

In case of simple „Polling messages“ that dont interact with the car state, like the „CInfoStateMessage“ and the „CInfoSensorMessage“ which do only read some information out of the current state, there is immediately created and sent an answer message, containing the required data, after the message is executed by the car.
The other three messages have an influence on the car state and therefore have to be queued until the car state gets updated. Once the update is there, the main loop of the socketserver checks if the requested state change like for example an emergency brake process has been performed or not, deletes the message from the queue and sends an answer message. Depending on this check the produced answer message contains a different flag:

- „A“ for successful execution(„Answer“)
- „F“ for failed execution
- „O“ for the message being outdated

The main challenge and reason for this check is the fact that communication and state control are performed by different NIOS2 cores. While the state is being updated periodically, incoming messages might arrive more frequent. If for example 3 „CControlMessages“, which contain the information to set the motors to a specific speed, are received within one state update cycle, it is obvious that only the last one should be executed in the new state and the older ones get outdated as soon as a new message of the same type is received. Whenever a message gets queued, and there is already a message in the queue, the outdated one gets immediately answered with an „O“ flag. This message then is outdated and doesnt have to be executed any more, since it has been overwritten by the latest message, resulting in only one

queued message at maximum for every message type at every state update. For more information about the queueing process read the Communication Core chapter.

In the following part all types of CAR2X messages are xplained using examples. Note that there is always the protocol header included. For answer messages the protocol header consists of CARP, „ControlCoreCounter“,„CommCoreCounter“ payloadLength, success-flag and message type.

CEmergencyBrakeMessage

Example packet sent to the car:

Byte	1	2	3	4	5 + 6	7 + 8	9	10	11	12
Inhalt	C	A	R	P	PacketID	PayloadLength: 0x04	Type: 0x20	Length: 0x04	Subtype: 0x0	Flags: 0x0

After receiving this message, the control core is requested to change the car state into emergency braking and this message is queued to wait for the next state update, or being outdated.

The payload of the answer consists of the PacketNumber of the received message.

Example answer message sent by the car:

Byte	1	2	3	4	5 - 8	9 - 12	13 - 16	17	18	19 + 20
Inhalt	C	A	R	P	ControlCoreCounter	CommCoreCounter	PayloadLength	Success flag (A,F,O)	Type: 0x20	PacketID of the received message

CControlMessage

Example packet sent to the car:

Byte	1	2	3	4	5 + 6	7 + 8	9	10	11	12	13 + 14	15 + 16	17 + 18	19 + 20
Inhalt	C	A	R	P	PacketID	PayloadLength: 0x0C	Type: 0x30	Length: 0x0C	Subtype: 0x0	Flags: 0x0	V1	V2	V3	V4

After receiving this message, the control core is requested to set the specified motor velocities (V1...4 as 16-bit values), specified in the payload. It has to be mentioned that in case the sender of this message is not registered as the current source of control, the message is not queued but immediately replied as failed because the sender is not allowed to control the car. But if the control is allowed, this message is queued to wait for the next state update, or getting out of date.

Example answer message sent by the car:

Byte	1	2	3	4	5 - 8	9 - 12	13 - 16	17	18	19 + 20	21 + 22	23 + 24	25 + 26	17 + 28
Inhalt	C	A	R	P	ControlCoreCounter	CommCoreCounter	PayloadLength	Success flag (A,F,O)	Type: 0x30	PacketID of the received message	V1	V2	V3	V4

V1 to V4 are now standing for the current desired motor speed values delivered to the specific PWM motor controllers. In case of a successful

message, they are equal to the requested values of the message, sent to the car, otherwise they differ and the message is answered as „failed“, giving the controlling unit feedback about the current state of the car, to let it know what might have failed.

CRemoteControlMessage

Example packet sent to the car:

Byte	1	2	3	4	5 + 6	7 + 8	9	10	11	12	13	14	15	16
Inhalt	C	A	R	P	PacketID	PayloadLength: 0x08	Type: 0x60	Length: 0x08	Subtype: 0x0	Flags: 0x0	IP1	IP2	IP3	IP4

The payload of this message contains the IP of a source which should be allowed to control the car by using „CControlMessages“. Per default this source is the unit inside of the car featuring the ImageProcessingUnit with its IP 192.168.0.110. In the standard state of the car „AutoDrive“, the car is controlled from this IP by „CControlMessages“. By sending a „CRemoteControlMessage“ to the car containing a different IP, the car is requested to set its state to „ManualDrive“ with the new source IP locked for „CControlMessages“. Sending 0.0.0.0 as new IP will set the car back to „AutoDrive“, using the ImageProcessing IP again. As before, the answer gets queued until the state is updated for the next time.

Example answer message sent by the car:

Byte	1	2	3	4	5 - 8	9 - 12	13 - 16	17	18	19 + 20	21 + 22	23 + 24	25 + 26	17 + 28
Inhalt	C	A	R	P	ControlCoreCounter	CommCoreCounter	PayloadLength	Success flag (A,F,O)	Type: 0x60	PacketID of the received message	IP1	IP2	IP3	IP4

IP1 to IP4 are the current state values of the locked IP. In case of success they equal the requested one, otherwise they give the feedback about who is currently controlling the car.

CInfoStateMessage

Example packet sent to the car:

Byte	1	2	3	4	5 + 6	7 + 8	9	10	11	12	13	14	15	16
Inhalt	C	A	R	P	PacketID	PayloadLength: 0x04	Type: 0x40	Length: 0x04	Subtype: 0x0	Flags: 0x0				

This message just polls the current state information from the car. It is answered immediately and thus has not to be queued. The answer always contains the „A“ flag for success and features the biggest part of the state object from the shared memory as its payload. It can be used for debugging purposes or by the ImageProcessing unit or a station as additional odometry feedback

Example answer message sent by the car:

Byte	1	2	3	4	5 - 8	9 - 12	13 - 16	17	18	19 + 20	21 - 24	25	26	27	28	29	30	31	32	33	34	35 - 36	37 - 78	79 - 100	101 - 122	123 - 130
Inhalt	C	A	R	P	ControlCoreCounter	CommCoreCounter	PayloadLength	Success flag (A,F,O)	Type: 0x40	PacketID of the received message	tMaxSpeed	IP1	IP2	IP3	IP4	reqIP1	reqIP2	reqIP3	reqIP4	currMode	reqMode	1st ECU state	2nd ECU state	3rd ECU state	4th ECU state	reqVelocity

As „ControlCoreCounter“ and „CommCoreCounter“ are already part of the answer header while also beeing part of the car-state, they are excluded from the payload. Its the same for the sensor values which are polled by the „CInfoSensorMessage“seperately to safe bandwidth.

CInfoSensorMessage

Example packet sent to the car:

Byte	1	2	3	4	5 + 6	7 + 8	9	10	11	12	13	14	15	16
Inhalt	C	A	R	P	PacketID	PayloadLength: 0x04	Type: 0x50	Length: 0x04	Subtype: 0x0	Flags: 0x0S				

This message just polls the current sensor information from the car state. It is answered immediately and thus has not to be queued. The answer always contains the „A“flag for success and features the whole sensor information which are part of the car state. This message can be used to access the sensor data. Note that currently there is no sensor hardware connected and there is still no implementation to get the sensor information from this missing hardware. On the other hand, the state object already contains memory to store this data which is read out by one of these messages. The polling interface by CAR2X messages is completely working for an assumed amount of 2 sensors but will always deliver no information until the internal sensor read out will be implemented.

Example answer message sent by the car:

Byte	1	2	3	4	5 - 8	9 - 12	13 - 16	17	18	19 + 20	21 - 24	25 - 28
Inhalt	C	A	R	P	ControlCoreCounter	CommCoreCounter	PayloadLength	Success flag (A,F,O)	Type: 0x50	PacketID of the received message	Sensor 1 Value	Sensor 2 Value

Chapter 5

Conclusion

As the project goals were slightly changing over the lab course semester and the possible work and amount of ideas to be realized got into dimensions that could have filled many lab course semesters and not just one, we had to realize that we wont be able to build the perfect and completely stable Car2X platform without any bugs. We learned a lot of things aboout Software/Hardware co-design what includes a lot of time spent into looking for errors, bugs and hardware faults. This lead us to the decision to define our goal as getting the system far enough to run on our multicore FPGA and provide all the Car2X fatures in a way that it was possible to successfully test the functionalities in a way the positive results could be reproduced. This means that at the end we reached a state where the basic concept was successfully implemented and working BUT our result equals something comparable to an early developement prototype, meaning that there are definitely a lot of bugs and unhandeled cases making the car itself quite unstable in terms of reliability.

Therefore we suggest to invest some decent amount of time into testing and optimizing the current state of the car because we sadly did not have the time to do that as well as we would have wanted to. Also feel free to contact our group if you need some help. We know how hard it is to get into a previous' groups work...

Appendix A

Stuff...