

Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Measurement

Firmware for Control Module of an Intelligent Vehicle

Bc. Jaroslav Beran

Supervisor: doc. Ing. Jiří Novák, Ph.D.

Field of study: Open Informatics

Subfield: Computer Engineering

August 2020

Acknowledgements

I would like to thank my supervisor Doc. Ing. Jiří Novák, Ph.D. for the consultations, valuable advises and support during the work on this thesis. I would also like to thank Ing. Ondrej Ille and Ing. Pavel Píša, Ph.D. for valuable support and advices during integration stage of CTU CAN FD IP Core. Finally, I would like to thank my family and friends for support during my entire studies.

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

I hereby declare that I developed the submitted work independently and that I listed all the used information sources in accordance with the Methodical Instruction on Adherence to Ethical Principles in the Preparation of University Theses.

Prague, August 14, 2020

Abstract

This work deals with the design and implementation of the control module firmware for the purpose of controlling selected functions of an intelligent vehicle. The introductory part outlines the principles of automotive CAN network and then deals with the analysis of the car functions for controlling the air conditioning, speed limiter, adaptive cruise control and infotainment. Based on the analysis, the requirements for the solution are determined, and the design part is approached. The design part deals with the selection of components that constitute the control module. After the Terasic DE0-Nano-SoC development kit, the CAN FD IP Core CTU controller and the independent CAN FD Gateway modules are selected for the solution, the overall system structure is designed. Then the communication protocol towards the superior artificial intelligence system is designed, as well as the algorithms for control of CAN Gateway modules. In the following section, the control software is designed with respect to extensibility of the system in the future. Then, the Linux OS is deployed on the development kit, the CTU CAN FD controller is integrated into its FPGA, the driver of this controller is integrated into the OS, and finally the control software, which is implemented in C++, is integrated too. The resulting prototype was integrated into a Škoda Kodiaq vehicle and several test drives were performed with success.

Keywords: CAN, CAN FD, automotive, control, Linux, kernel, programming, software design, C++, system on chip, embedded systems, FPGA, Avalon bus

Supervisor: doc. Ing. Jiří Novák, Ph.D.

Abstrakt

Tato práce se zabývá návrhem a implementací firmware řídicího modulu pro účely ovládání vybraných funkcí inteligentního vozu. Úvodní část nastiňuje principy automobilové sítě CAN a následně se věnuje analýze funkcí vozu pro ovládání klimatizace, omezovače rychlosti, adaptivního tempomatu a infotainmentu. Na základě analýzy jsou stanoveny požadavky na řešení, a je přistoupeno k návrhové části. Tato se zabývá výběrem komponent, ze kterých je složen řídicí modul. Poté co je pro řešení vybrán vývojový kit Terasic DE0-Nano-SoC, radič CTU CAN FD IP Core a nezávislé moduly CAN FD Gateway, je navržena celková struktura systému, poté komunikační protokol s nadřazeným systémem umělé inteligence a dále metody řízení modulů CAN Gateway. V následující části je navržen řídicí software s ohledem na rozšiřitelnost systému do budoucna. Pak je na vývojovém kitu zprovozněn OS Linux, do FPGA na kitu je začleněn radič CTU CAN FD, do OS Linux ovladač tohoto radiče, a nakonec je do systému zakomponován řídicí software, který byl naimplementován v jazyce C++. Výsledný prototyp byl integrován do vozu Škoda Kodiaq, kde bylo úspěšně provedeno několik testovacích jízd.

Klíčová slova: CAN, CAN FD, automotive, řízení, Linux, jádro, programování, návrh software, C++, systémy na čipu, vestavné systémy, FPGA, sběrnice Avalon

Překlad názvu: Programové vybavení modulu pro řízení funkcí inteligentního vozu

Contents

Project Specification	1	3.5 Command Manager	34
1 Introduction	3	3.6 Command processors	34
1.1 Motivation	3	3.6.1 General processing procedure	35
1.1.1 Control elements in the vehicle	3	3.6.2 Commands for the RPC	
1.2 Outline of the system	5	protocol	35
1.3 Goals	6	3.6.3 Commands for ACC and Speed	
1.4 Controller Area Network	6	Limiter	37
1.4.1 Physical layer	6	3.7 Runtime configuration	41
1.4.2 Data link layer	7	3.7.1 System schema	41
1.4.3 Frame format	9	3.7.2 Commands schema	42
1.4.4 CAN with Flexible Data-Rate	9	3.7.3 Processing schema	43
1.5 Network in a vehicle	10	3.8 Logging	43
1.5.1 Topology	10	3.8.1 Logger	44
1.5.2 Messages	11	3.8.2 Log Server	44
1.6 Analysis	12	3.8.3 Log Writer	44
1.6.1 Vehicle control functions	12	4 Integration of components to	
1.6.2 Requirements for the solution	12	Terasic DE0–Nano–SoC board	47
1.6.3 List of tasks	13	4.1 Preparation of the development	
2 System design	15	board	47
2.1 Selection of HW platform	15	4.1.1 Boot process description	47
2.1.1 CAN Gateway	15	4.1.2 SD card partitioning and	
2.1.2 Controller Module	16	contents	48
2.1.3 CAN FD controller	17	4.1.3 Making the artifacts	48
2.2 Top level system design	18	4.2 Integration of CTU CAN FD IP	
2.3 Communication protocol between		Core	53
CAN Activator and Master Control		4.2.1 IP block with Avalon interface	53
System	19	4.2.2 Device tree node	55
2.3.1 Transport layer discussion . . .	19	4.2.3 Linux driver	56
2.3.2 Data exchange	21	4.3 Integration of software artifacts .	56
2.3.3 Messages format	22	4.3.1 Build process	56
2.4 Control of CAN Gateway modules	25	4.3.2 Configuration of the target	
2.4.1 General operation	25	system	57
2.4.2 Communication with Gateway	25	4.3.3 SW deployment to the target	57
2.4.3 Gateway functions	25	5 Testing and Conclusion	59
2.4.4 Design of the interface	26	5.1 Final Conclusion	60
3 Software design	29	Bibliography	63
3.1 Control application architecture	29	A Abbreviations	67
3.2 Event Loop	30	B List of implemented commands	69
3.2.1 EventedFd	31		
3.3 CAN Components	31		
3.3.1 CAN Socket and Interface . . .	31		
3.3.2 CAN Gateway	32		
3.4 Master Control System Interface	33		
3.4.1 TCP communication	33		
3.4.2 Request and response handling	33		

Figures

1.1 Control panel of Climate unit . . .	4	5.2 Control module (on the left) connected to one of the CAN Gateways (Gateway data interfaces are disconnected)	62
1.2 Control lever of ACC and Speed Limiter	4		
1.3 Interface of Infotainment	5		
1.4 Block schema of the car network controlled by an artificial intelligence	6		
1.5 CAN node error states transitions. Taken from [40, p. 44]	9		
1.6 Frames in FD format can be transmitted with dual bit-rate. Taken from [9]	10		
1.7 A typical car network (from [5])	11		
1.8 Photo of the interior of the vehicle during analysis	13		
2.1 CAN FD Gateway module	16		
2.2 Terasic DE0-Nano-SoC board (Taken from: [4])	17		
2.3 Terasic DE0-Nano-SoC board with CTU IO Extension board attached	18		
2.4 Top-level system architecture with physical connection lines and logical data flow (grey, dashed)	19		
3.1 SW architecture of CAN Activator application	30		
3.2 Execution of one MCS request . .	31		
3.3 State diagram of the processing of RPC-protocol commands	36		
3.4 State diagram of ACC and Limiter states. Within each state are listed commands that are applicable in that state	38		
3.5 Processing of Limiter enable and ACC enable commands. Labels in parentheses apply for ACC enable	39		
3.6 Process of setting a target value by modification of a control signal . . .	40		
3.7 Logging solution	44		
4.1 Correct waveforms on CTU CAN FD Avalon interface (taken from Platform Designer IDE)	54		
5.1 Tester client window	61		

Tables

3.1 List of commands for ACC and Speed Limiter control	37
4.1 CAN signals to pins assignment	55
4.2 Build artifacts of the can_activator project	57

I. Personal and study details

Student's name: **Beran Jaroslav** Personal ID number: **364455**
Faculty / Institute: **Faculty of Electrical Engineering**
Department / Institute: **Department of Measurement**
Study program: **Open Informatics**
Specialisation: **Computer Engineering**

II. Master's thesis details

Master's thesis title in English:

Firmware for Control Module of an Intelligent Vehicle

Master's thesis title in Czech:

Programové vybavení modulu pro řízení funkcí inteligentního vozu

Guidelines:

Design and implement software for CMV module intended to control functionalities of an intelligent vehicle. Follow these steps:

1. Implement support for CTU CAN FD IP function into Linux OS.
2. Implement the software library supporting communication with CAN FD Gateway modules..
3. Design and implement communication protocols to master control system.
4. Design and validate algorithms for independent control of selected vehicle functions.
5. Implement these algorithms into the CMV module and present demonstrate their functionality in Škoda car.

Bibliography / sources:

- [1] Ille, O.: CTU CAN FD Manual, 2019
- [2] Schwank, F.: Programové vybavení pro CAN Gateway. Diplomová práce ČVUT FEL 2018
- [3] Jeřábek, M.: Open-source a Open-hardware podpora pro CAN FD. Diplomová práce ČVUT FEL 2018

Name and workplace of master's thesis supervisor:

doc. Ing. Jiří Novák, Ph.D., K 13138 - katedra měření

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **30.01.2020** Deadline for master's thesis submission: **14.08.2020**

Assignment valid until:

by the end of winter semester 2021/2022

doc. Ing. Jiří Novák, Ph.D.
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt

Student's signature

Chapter 1

Introduction

1.1 Motivation

Nowaday road vehicles are increasingly taking advantage of information and communication technologies. Demands on travel safety and comfort of passengers are growing at the same time. To meet these requirements, car manufacturers introduce new improvements and innovations into their products so they can reach customers.

The current trend is to use artificial intelligence (AI) for many interesting and computable complex tasks and one car manufacturer came up with an idea he would like to implement into one of its own vehicles.

The idea is to facilitate passengers a travelling in a vehicle by means of introducing a system that would monitor and learn which actions the passengers regularly take and under which circumstances. Such system would then use the gained knowledge and try to anticipate certain passengers' actions that it would perform automatically.

1.1.1 Control elements in the vehicle

Here comes an overview of the control elements that could be subject to that AI control. Information in this section comes from a source of the car manufacturer that cannot be referenced.

Air Conditioning, Ventilation and Heating

The control panel on Fig. 1.1 forms the user interface to Climate unit functions.

Functions that can be controlled by this panel include:

- Controlling the volume and distribution of air blowing out of air vents.
- Setting the desired temperature of the interior
- Choose between manual and automatic modes for reaching the target temperature
- Enabling or disabling the interior cooling system



Figure 1.1: Control panel of Climate unit

- Control of seat heating and ventilation
- Steering wheel heating
- Front and rear windows heating
- Enabling air recirculation, and more...

■ Adaptive Cruise Control and Speed Limiter

The control lever on Fig. 1.2 has control elements common for both Speed Limiter and ACC.



Figure 1.2: Control lever of ACC and Speed Limiter

Limiters just sets the limit on maximum speed. ACC regulates the speed of the vehicle to the desired value and also keeps distance from the preceding vehicle.

The lever has two stable positions OFF and ON. When the lever is OFF, both ACC and Limiter are disabled. In position ON, either ACC or Limiter is selected, but it is inactive. The MODE button switches between these two

modes. From position ON, the lever can be triggered to one of sprung positions RESUME or CANCEL. RESUME has the function of both activation of the selected mode and increasing the speed by 1 km/h. CANCEL deactivates the currently active mode.

The lever can be triggered up and down to increase (decrease) speed by 10 km/h. SET button decreases speed by 1 km/h.

DISTANCE switch regulates ACC distance of the preceding vehicle.

Information about current state, speed and distance is shown on the Instrument cluster display.

■ Infotainment

Infotainment interface is provided by a big touchscreen in the middle of the front panel (Fig. 1.3).

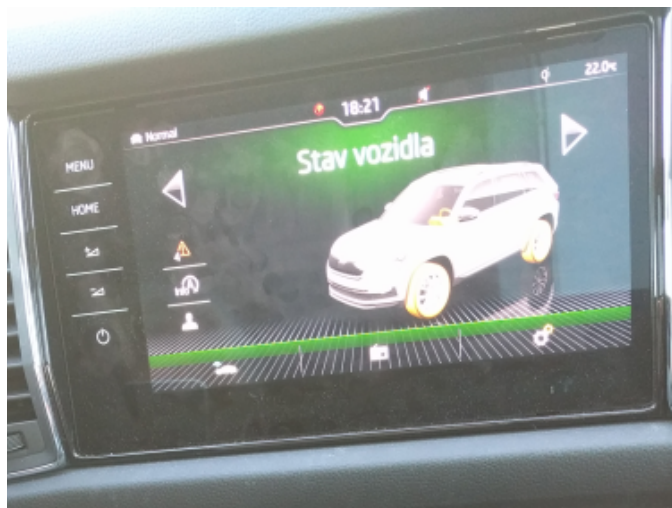


Figure 1.3: Interface of Infotainment

There are many menus and mostly static configuration is stored here. Only the interactive ones, which an user sometimes changes, are interesting for purposes of controlling by AI.

These may be e.g. changing the driving mode, setting the audio source to some value,

■ 1.2 Outline of the system

Schema on Fig. 1.4 describes the system in basics.

Block MCS (Master Control System) is the AI containing the decision logic. It commands the Control module block, whose task is to trigger actions on the vehicle control units to emulate user actions on the control elements of these units. The Control module block is the main concern of this thesis.

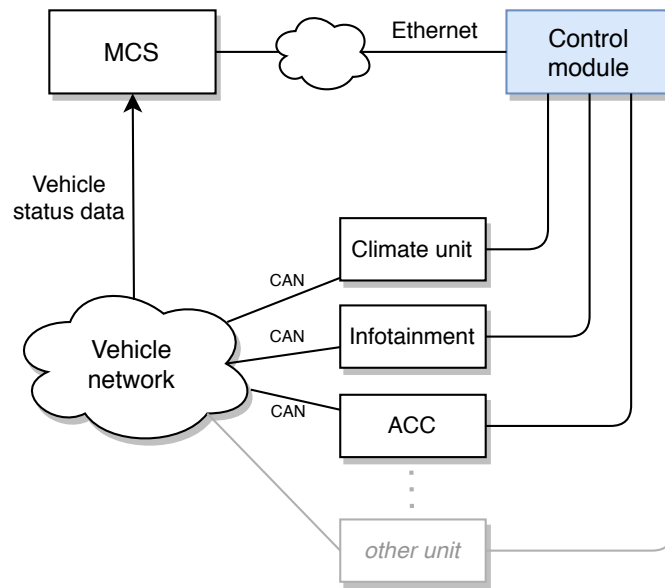


Figure 1.4: Block schema of the car network controlled by an artificial intelligence

1.3 Goals

Goals of this thesis are to implement the Control module block on Fig. 1.4, design and implement its interface towards MCS and implement a set of commands for controlling selected vehicle functions.

1.4 Controller Area Network

Since this thesis deals with a road vehicle internals, this section provides an overview of the fundamental network technology used for communication of units inside a vehicle.

CAN is a standard for multi-master serial communication. It was designed by Bosch company as a reliable bus intended for use in automotive industry. Since then it has been widely used for implementation of in-vehicle networks ([28]). Properties of Data Link and Physical layers are specified in ISO 11898-1 standard [40]. A general and higher-level view is provided in [10]

Communication on CAN bus happens in a broadcast manner. There is no direct addressing of target nodes, but instead various nodes transmit frames that are distinguished by a unique identifier. Any node attached to the bus receives all frames that appear.

1.4.1 Physical layer

Physical layer defines electrical properties of the bus, transceiver characteristics, signal levels, bit timing and mutual synchronization of nodes.

Implementation of the physical layer is covered in [11]. And even more details of the electrical aspects are covered in [49]

Bus signals. The physical bus is realized by two signal wires, CANH and CANL. A differential voltage between these two signals defines either dominant or recessive level of the bus. Differential signalling is one of the features that makes this bus robust with respect to crosstalks and EMI [36].

Dominant level represents logical '0', recessive logical '1'. Dominant level is actively driven by a sending node. If at least one node sends dominant, the bus level is dominant. Recessive is the default level when no node transmits dominant level. The bus is pulled to the recessive level by termination resistors. Termination also prevents reflections on the line.

Bit synchronization. Due to lack of a common clock, a proper synchronization between nodes must be ensured so they all sample the correct bit even the signal from one node can get to different nodes with different delays. Synchronization is carried out by dividing a bit-time into segments, which define the sample point of a single bit. The sample point on each node gets continuously fixed according to actually detected delay.

1.4.2 Data link layer

Data link layer handles encapsulation of data into frames with well defined format, error signaling and detection, bit-stuffing, acknowledgement of transmitted frame and serialization/deserialization. In addition to the specification [40], provided information is taken from [8].

Frame transmission. Information on the bus is exchanged in terms of frames with defined format. A node can start transmission if the bus is idle, which is the default state on the bus after certain time since the last transmission occurred.

Transmission starts by SOF (start of frame) bit, which is dominant. Then comes the arbitration phase, during which it is settled which node acquires the bus. In arbitration phase transmitting nodes send CAN identifiers, MSb first. Each of the nodes transmits its bit and simultaneously observes the actual state of the bus. If it differs, which condition happens when a node sends recessive and reads dominant, it retreats from further transmission. As a consequence of the definition of dominant and recessive states, the node that transmits CAN ID with the lowest value, wins the arbitration.

After the arbitration phase, there is only one transmitting node, which continues with the transmission of its frame. It now transmits the rest of the frame consisting of control field and data. In the end, CRC field is sent to provide some protection of the data on this layer.

At this point, error check is performed in ACK slot, during which transmitter sends recessive and any other node acknowledges reception of the frame by driving the bus to dominant.

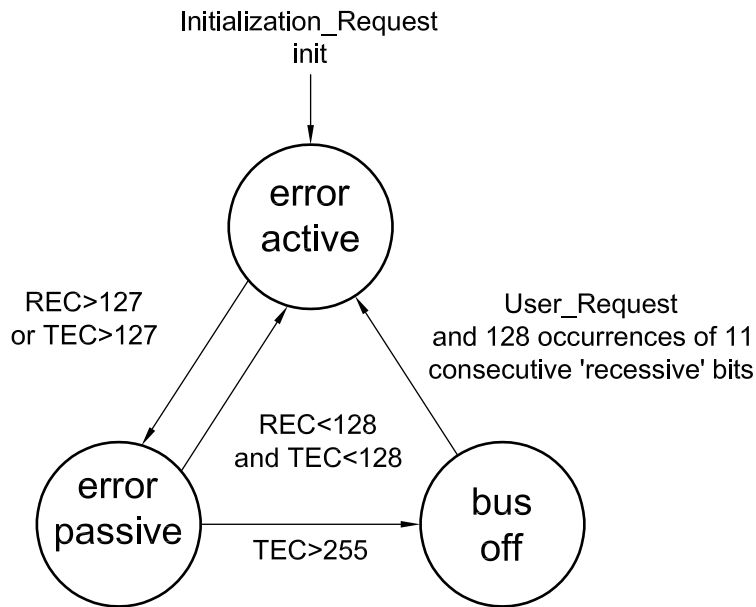


Figure 1.5: CAN node error states transitions. Taken from [40, p. 44]

- When a node detects an error during transmission (reception), it increases its TEC (REC respectively) by 8.
- When a node successfully transmits (receives) a frame, it decreases its TEC (REC respectively) by 1.

■ 1.4.3 Frame format

According to the specification ([40, p. 10.4.2.3]) CAN supports base frame format with 11-bit identifiers and extended format with 29-bit identifiers. It is achieved by dedicating defining Identifier Extension (IDE) bit in the arbitration field to distinguish between them and extending arbitration field in extended format frames.

There is also Remote Request type of frame, which unlike data frames does not contain data fields, but is rather used to request a transmission of a data frame with that identifier from a remote node.

■ 1.4.4 CAN with Flexible Data-Rate

CAN FD is an extension of CAN protocol, which is specified by Bosch [12].

It extends the classical CAN frames by redefining and introducing some new control bits in the frame control field. The changes are forward-compatible, so a CAN FD controller also supports classical CAN frames. Basic ideas are summarized in articles [9, 31].

Data length extension. Maximum length of data field in frames is extended by utilizing all possible values of 4-bit DLC field. So in addition to values

from 0 to 8 bytes, data length can be 12, 16, 20, 24, 32, 48 or 64 bytes as well. To ensure transmission reliability of longer data frames, an extension of the CRC field was also defined by this standard.

Dual bit-rate. The standard also introduces possibility to use a second bit-rate for the data phase to increase transmission speed. The idea is shown on Fig. 1.6.

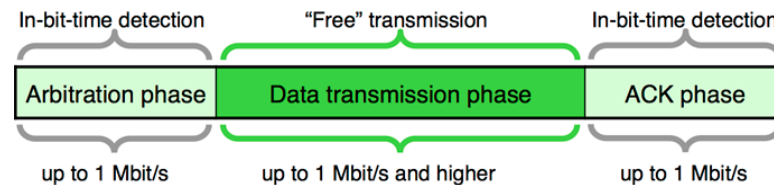


Figure 1.6: Frames in FD format can be transmitted with dual bit-rate. Taken from [9]

One bit-rate is used for arbitration and ACK phases, which is up to 1 Mbit/s that the classical CAN supports. For data phase a new control bit BRS (bit-rate switch) is introduced. If recessive, all controllers switches their clocks to a higher rate for duration of data phase. No maximal value of the second bit-rate is specified, but it is constrained by physical properties of used transceivers and of the bus such the actual topology, lengths of signal wires, number of nodes on the network and other properties that affect propagation of a signal.

Both standard 11-bit and extended 29-bit identifiers are supported for FD frames.

1.5 Network in a vehicle

Once the fundamental vehicles network protocol was introduced, it is now time for closer look on the network topology, nodes, and messages being transferred.

1.5.1 Topology

A typical car network is shown on Fig. 1.7

There are various Electronic Control Units (ECU), each is responsible for controlling a certain part of the vehicle, like engine, transmission, anti-braking system, doors, radars, etc. Units can be organized into topologies as needed to fulfill requirements on safety, performance and other properties that come to mind when dealing with a network architecture.

ECUs that communicate mostly between themselves can be grouped on a dedicated network segment and connected to the rest of the network through a central gateway.

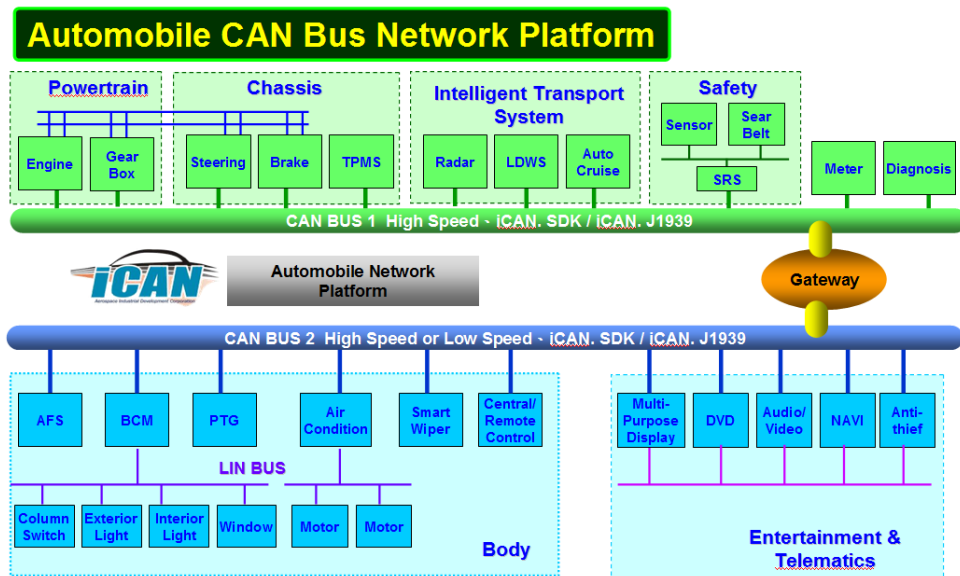


Figure 1.7: A typical car network (from [5])

1.5.2 Messages

DBC database. In order to provide ECUs on a particular network segment the ability to read and decode the messages that appear on the bus, a definition is provided in a database file. Commonly used database format is DBC, which is an ASCII-based proprietary format of the Vector company [16].

Besides other information, it defines what nodes exist on that network segment and which messages are being sent there. Message content is defined on a bit level by signals definition. For each signal, there is start bit offset, length, data type it represents, endianness and transformation options can be specified.

There is also defined which units send which messages. For each message signal, there can be defined which units are receivers of this particular signal.

Message types. There are two main classes of messages in the network of the vehicle to which this thesis relates.

- Cyclic messages are sent by an ECU with a defined period (usually tens of milliseconds). These messages contain an additional CRC field on a fixed position of the message data.
- One-shot messages that carries data of an proprietary application protocol. This protocol is similar to an RPC. It defines messages with some header, operations (read, write, response) and parameters. Details are not disclosed here. This protocol is hereinafter referred to as the proprietary RPC protocol or just RPC if the context is clear.

1.6 Analysis

1.6.1 Vehicle control functions

This part concerns analysis of CAN messages belonging to control units of the car functions that are in the scope of this thesis.

For the purposes of analysis and testing was engaged a vehicle equipped with connections to certain segments of the CAN network.

Computers with diagnostic software were connected to the network with use of the following tools:

- Kvaser Memorator Pro 2xHS v2¹ USB to CAN converter.
- Vector CANoe for capturing, decoding and sending messages to the network.
- Utilities from `can--utils` project ² for capturing and sending messages.
- A Python package `cantools` ³ for decoding messages according to the definitions in DBC databases.

Illustration from this phase of work is on Fig. 1.8.

It was necessary to determine which messages in which direction are being sent while triggering the control elements and to clarify some data fields, ranges of some parameters and to observe unexpected behaviour.

Outcomes

The analysis showed that the Air conditioning and Infotainment functions can be controlled by one-shot messages with the proprietary RPC protocol payload.

The RPC protocol cannot be utilized for control of ACC and Limiter functions. Instead, it was identified that the actual position of the control lever is sent in one cyclic message. There are two more status messages. One carries the information about current status and setpoint speed for Limiter. The second message carries the same information for ACC plus a signal indicating the distance setting.

1.6.2 Requirements for the solution

General requirements. Automotive industry puts great emphasis on the safety. It is obvious that the equipment must not put the crew in danger in any way.

The solution should be simple, because complex things tends to fail easily and it is more difficult to debug complex system than simple one. Acceptable

¹<https://www.kvaser.com/product/kvaser-memorator-pro-2xhs-v2/>

²<https://github.com/linux-can/can-utils>

³<https://github.com/eerimoq/cantools>



Figure 1.8: Photo of the interior of the vehicle during analysis

cost, accessibility of HW components, reliability and modularity are also important requirements.

Functional requirements. The outcomes of functions analysis showed that the solution must be able to send, read and modify CAN messages in some way and be able to recalculate CRC field of cyclic messages, if it will modify these messages.

The solution also has to be able to receive requests and send responses to the Master Control System over Ethernet.

■ 1.6.3 List of tasks

These are the tasks that has to be done to fulfill the goals:

- Choice of suitable platform for the solution
- Design of communication protocol between the selected platform and MCS
- Design and implementation of actions in the car network

- Testing the solution inside a vehicle, carry out several test drives

Chapter 2

System design

Based on the analysis of the control functions and the requirements for the solution, the design part can be approached.

2.1 Selection of HW platform

One of the possible approaches would be to integrate the entire HW on one board together with the control application. However, this decision would be difficult to extend if more CAN network segments were introduced in the future.

With regard to modularity and availability of the HW components listed below, the following proposal was selected:

- One central module carrying the control logic for actions on the vehicle elements and interface to the MCS.
- One or more independent modules carrying out the technical implementation of the actions.
- Connect the low-level modules to the central module in a suitable way.

2.1.1 CAN Gateway

CAN FD Gateway is a module developed on the Department of Measurement at CTU [18].

The board, which is shown on Fig. 2.1), has two CAN data interfaces and one CAN FD control interface.

By default, its main activity is to forward frames from one data interface to another. The control interface allows the gateway to be controlled using commands in the form of CAN FD frames.

Control commands allow an user to set rules for blocking or modifying some frames, based on their IDs. The important feature of the Modify command is the ability to calculate (after modification) the correct CRC field of cyclic messages, which were analysed in section 1.6.

The Gateway also has function for sending and reading of messages to or from specified target data interface.

Thanks to a configurable address of the control interface, more of these modules can be present on one network segment.

These features make this module perfect for utilizing it in this work.

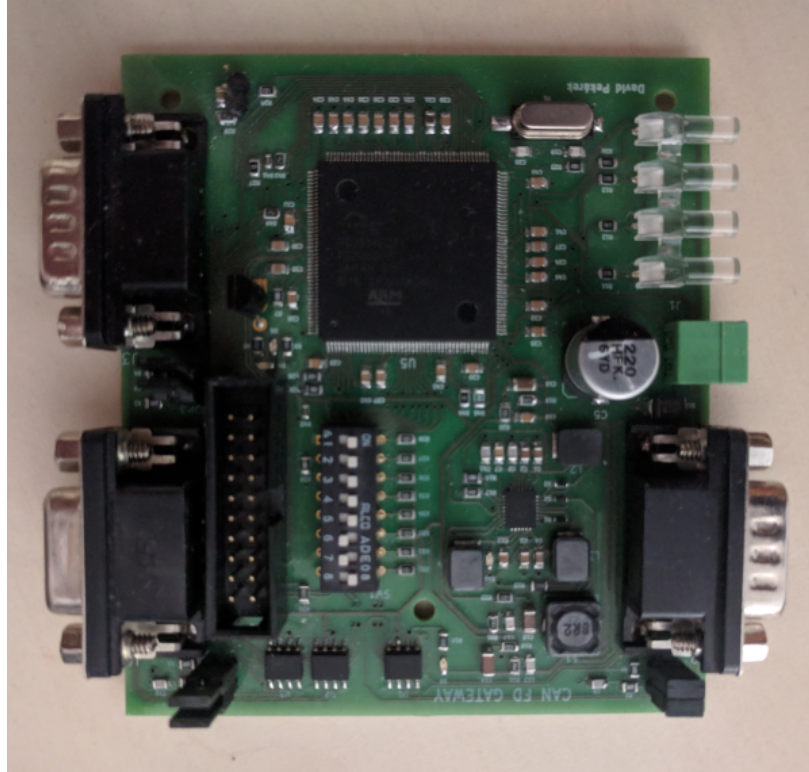


Figure 2.1: CAN FD Gateway module

■ 2.1.2 Controller Module

The CAN Gateway itself is a simple module with no clue about the meaning of frames that go through its data interfaces. It just routes the frames according to set of rules. So a control logic for such rules has to be provided by an external module with CAN FD interface.

When selecting the main module, more options were considered (A STM32 platform based, Xilinx Zynq based). One of the important guidelines was the idea that for prototyping it is better to have a universal platform and some reserve in resources (CPU, HW, memory, I/O).

Other important aspects are the peripherals actually present on board, connectivity options, extendability (both SW and HW), which software development tools are supported for that platform, and previous experience with a platform.

The platform has to provide at least an Ethernet interface for the connection to MCS and CAN FD controller to interface the Gateway modules.

■ Terasic DE0-Nano-SoC Development board

The Terasic DE0-Nano-SoC Kit [17] is a development board with an Altera System-on-Chip (SoC) as the main processing part. The SoC integrates an FPGA (based on Altera Cyclone V) and hard-processing system (HPS) with dual-core ARM Cortex-A9 on a single chip. The HPS also contains various controllers (Ethernet, CAN, USB, SPI, SD, I2C, ...). The board provides 1 GiB of RAM and a rich set of I/O interfaces (Fig. 2.2).

Linux OS, which runs there, renders a very nice environment for SW development.

These features make this platform perfect for prototyping.

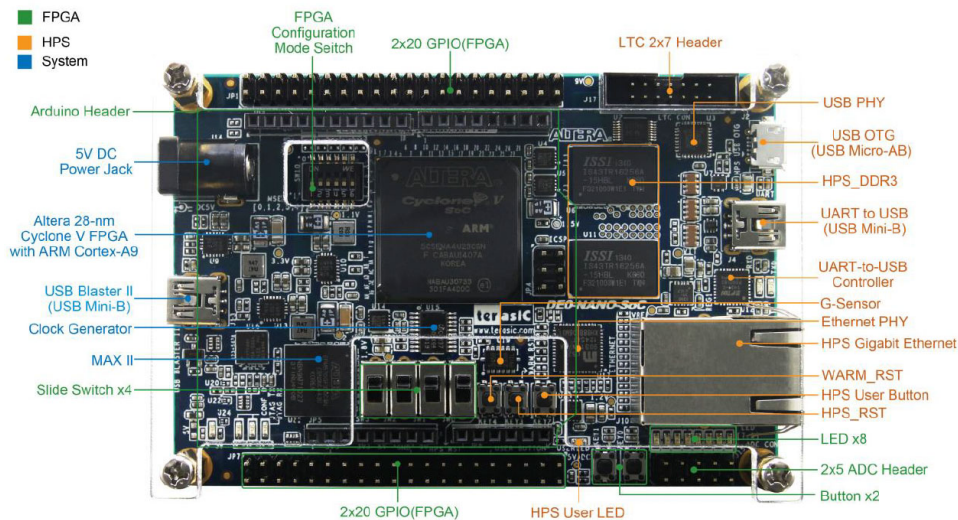


Figure 2.2: Terasic DE0-Nano-SoC board (Taken from: [4])

Connection to CAN bus. An I/O extension board was used for connecting the CAN controller on the Terasic DE0-Nano-SoC board to the CAN bus. This board had been developed by Jan Nejték in his bachelor thesis [35, Appendix C: 100Base-T1 board for Terasic DE0-NANO-SoC, rev. 2].

It was equipped with TJA1051-T high-speed CAN transceivers [46] and DB9 connectors. Both modules connected together are on Fig. 2.3.

■ 2.1.3 CAN FD controller

The SoC on the Terasic board integrates a CAN controller within its HPS, but unfortunately this controller does not support the CAN FD standard. Therefore, an external CAN FD controller has to be added.

One option that was considered was to use the MCP2517FD controller with SPI interface from Microchip [32]. However, there was no working Linux driver at that time, and there were concerns about the reliability of this HW with respect to the SPI connection, so this option was denied. Other options were some proprietary IP cores. But due to unclear pricing, licensing and

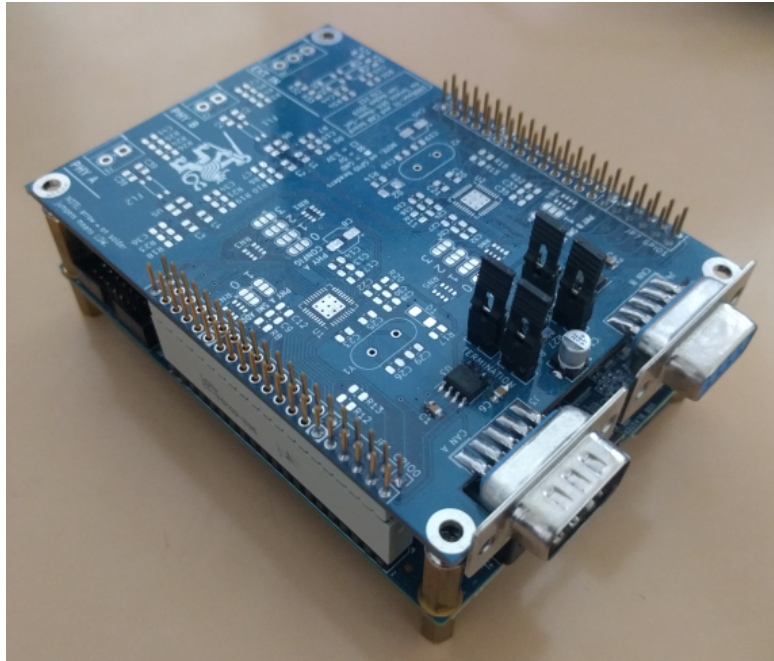


Figure 2.3: Terasic DE0-Nano-SoC board with CTU IO Extension board attached

uncertainty about the synthesis process into the Intel/Altera FPGA, these were not chosen either.

■ CTU CAN FD IP Core

CTU CAN FD IP Core [2] is an open-source HW project originated at FEE CTU by Ondrej Ille.

The core is still under development, but in stable and usable state. The project also includes a Linux SocketCAN[1] driver[30], which is also stable and tested. The bus interface for accessing its registers block is compatible with Avalon interface used for on-chip interconnections on Altera/Intel platform[29, 14]. It can be synthesized into Xilinx Zynq¹, which is similar platform as Altera SoC used in this thesis.

Based on these features, this IP core was chosen for implementation of CAN FD controller on the Terasic board.

■ 2.2 Top level system design

Just after all the HW components are selected, it is time to put them together to form a working system. A top-level structure of the system is on Fig. 2.4.

The dashed connections represent data flows of protocols and algorithms that will be designed in the following sections.

¹<https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>

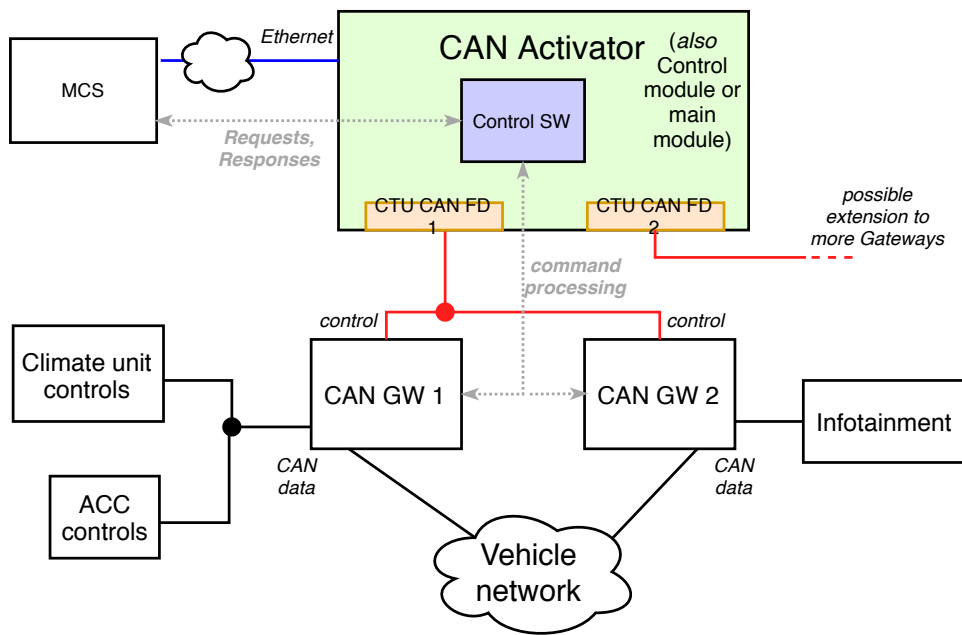


Figure 2.4: Top-level system architecture with physical connection lines and logical data flow (grey, dashed)

Due to disambiguity in terms when referring to the Control module as to main module or CAN Activator in later parts of the text, here is at least a remark on this issue.

2.3 Communication protocol between CAN Activator and Master Control System

In the section Top level system design (2.2) it was decided that CAN Activator and MCS will be connected over Ethernet. Naturally, IP, which is the most common protocol on the internet layer, is the only reasonable option. Then it is needed to make decisions on protocols of the transport and application layers.

2.3.1 Transport layer discussion

Regarding the transport layer, there are three usable options that can be considered on an IP network: UDP, TCP, SCTP.

UDP

User Datagram Protocol (specified in [48]) is the simplest among those three protocols.

Its characteristics are:

- Message based. Datagrams are transferred either whole or not at all. This is nice because transmitter does not need to care about messages framing. It just receives a message or handles an error.
- Stateless. There is no connection and state associated with it. This makes the protocol simple, but unreliable, since there is no guaranteed delivery and ordering of the messages. It has to be solved on another layer.
- Symmetry. The communication is carried out between two (or more) equal peers. No one of them acts as a server, they just transfer messages.

If UDP is used, the protocol will have to ensure proper ordering and delivery of request and response messages on an upper layer. On the other hand, framing of the messages will be ensured natively, if each message entirely fits in a UDP datagram.

■ TCP

Another widely used transport protocol is the Transmission Control Protocol (specified in [47]), whose main features are:

- Connection oriented. Prior to data exchange, a connection between client and server has to be established by a handshake procedure. When a communication is over, the connection is terminated in a similar handshake process.
- Reliability. The protocol guarantees proper ordering of packets by using sequence numbers. Retransmission of lost packets is also guaranteed.
- Stream orientation. TCP does not operate with messages but rather a stream of bytes. So there is no notion of message boundaries, which has to be handled in upper layers.

If the protocol is built upon TCP, there will be an unnecessary burden with the server side, listening, and accepting a new connection, even if the system is designed in such way there is only one client and one connection at a time.

The advantage is that the upper layers of the protocol would not care about proper data delivery. It would only need to ensure messages framing.

■ SCTP

The Stream Control Transmission Protocol (specified in [44]) provides advantages of both TCP and UDP. It is a message oriented like UDP and stateful, connection oriented and reliable as TCP. In addition it provides features like multihoming and it is more robust than TCP in terms of attacks like SYN flooding.

utility like telnet or netcat ([34]) without need of developing a client tester application at first.

Once a text oriented protocol is selected, there comes a choice about messages framing. This can be performed by roughly two ways ([19, p. 5.2.1]):

1. Prepend an information about the message length at the beginning of each message.

A downside is that the length of the message needs to be known in advance. This can be inconvenient e.g. in the development or debugging phase when one wants to send a test message with an arbitrary contents, so he needs to calculate length at first somehow.

An advantage is that it may be easier to segment such messages on the receiver side.

With this approach it is also possible to use all the byte values. However this is pointless if the protocol utilizes only a subset of byte range.

2. Use a dedicated delimiter character (or maybe a sequence of characters) at the end of each message.

Picking the right delimiter character is important, because this character cannot be used

Because the possible length of a message is conceptually unconstrained, the protocol should specify some maximum length.

With a text oriented protocol, it seems better to use the delimiter for framing. Then each message is a string terminated with a zero byte (ASCII 0x0). A message can be multiline and can contain any whitespaces. The receiver just scans for the delimiter. A message contains only ASCII characters (no multi-byte Unicode characters). Maximum length of a message on the receiving side is constrained by a sufficiently high constant (e.g. 512 bytes).

2.3.3 Messages format

On the decisions that have been made was chosen JSON format (specified in [6]) to represent messages in communication between the control module and MCS.

JSON format was chosen because it is universal, simple, lightweight and language-independent. It is also widespread, which is nice, because there are many open-source parsers available for various languages.

Possible drawbacks of JSON are that the format itself does not support comments, only 10 base format of numbers is supported and that the representation of numbers in the specification is defined quite vague. At least this can be mitigated by using a string to represent a precise value.

The concrete message format is inspired by JSON-RPC⁴. Each message is a JSON object (it is encapsulated in “{ }” at the highest level). There are following types of messages: Request, Response, Error Response, Heartbeat, and Heartbeat Response.

⁴<https://www.jsonrpc.org/specification>

Request

Requests are always initiated on the client side.

A Request message contains following fields:

- **command** – name of the command (string). This field is mandatory.
- **params** – JSON object containing key–value pairs of parameters for given command. A parameter values may be a number (integer, floating point), a string or a boolean. For a command without parameters the value shall be an empty JSON object.
- **id** – a string with an identifier used for correlation purposes between the Request and the associated Response. This field is recommended, but not mandatory. It should be omitted e.g. in the Heartbeat command.

A sample Request message is shown in Listing 2.1

Listing 2.1: Request message sample

```
{
  "command": "command_name",
  "id": "id_value",
  "params": {
    "integer_param": 1,
    "float_param": 23.5,
    "bool_param": true,
    "string_param": "normal"
  }
}
```

Response

Responses are always sent by the server side.

Response message contains following fields:

- **command** and **id** – that of the associated request
- **result** – either “ACCEPTED”, “DONE” or “ERROR”
- **error** – error field is present only in the error messages. Messages with “ACCEPTED” or “DONE” result does not contain this field

A sample Response message is shown in Listing 2.2.

Listing 2.2: Response message sample

```
{
  "command": "command_name",
  "id": "id_value",
  "result": "ACCEPTED"
}
```


When a Request is successfully accepted, the server sends a response message with the “result” field set to ACCEPTED. When a Request is successfully executed, the server sends a response message with the “result” field set to DONE.

ACCEPTED and DONE are consequent responses, so the client first gets ACCEPTED meaning the message has been validated and accepted for processing, then it receives a DONE result indicating that a command has been processed successfully.

At each of these two stages the server sends an error message if a problem occurs. An error message means that handling of the current command has been terminated and no further response for this request will come.

Error Response. If the server encounters an error during handling of a request, it sends a response with the **error** field.

The value of the error field is an object containing following fields:

- **code** – a string constant identifying the error. It may be used by the client to take further action
- **message** – a string with human readable description, intended mainly for logging purposes
- **command** and **id** fields are the same as in the corresponding request, or empty, if not specified in the request

It contains command and id fields as well if they’re provided in the Request causing the error.

■ Heartbeating

To track living connection between server and client, hearbeating mechanism is used since the connection is established.

The following thoughts and proposed mechanism are based on [13].

In case the link is somehow interrupted or in cases when the connection dies on one side without letting know the other side and other side does not detect that on its own, there would be a dead connection on either side.

A problem could happen after the link is reestablished after some time. If e.g. the server kept such dead connection, the client (which closed the previous connection on his side, but server didn’t notice) would not be able to reconnect to the server after a while, because the server allows only one connection at a time.

The client periodically sends request with the command set to “heartbeat” value and the server sends response (possibly with some information useful for the client).

Server measures a time since the last request was received. If the server does not receive any request within a certain time, it will close the connection.

Similarly, if the client does not receive any response within certain time, it will close the connection.

■ 2.4 Control of CAN Gateway modules

The hardware and features of CAN Gateway were described earlier 2.1.1.

This section deals with the design of how the control application on the main module communicates with the CAN Gateway modules connected to it. The design of controlling the CAN Gateways follows from its specification in the document [18].

■ 2.4.1 General operation

The Gateway manages two configuration stacks inside its firmware. One configuration is always active and another one is inactive. That one that is currently inactive is configured with the configuration functions. Toggling between these configurations is done with the Trig command. Each configuration stores filtering rules and default (blocking or passing) behavior. When the Gateway is turned on, it starts with an empty list of filtering rules and by default passes all frames through its data interfaces.

■ 2.4.2 Communication with Gateway

Each CAN Gateway module is identified by a 5-bit address, which is set by a DIP switch on the Gateway board.

The gateway address then defines CAN identifiers for request (configuration) and response (positive or negative acknowledgement) frames for communication on the control interface.

All frames on gateway's control interface are FD frames with Extended Identifier.

Request frame CAN ID has the following format:

0000 1000 0000 0000 0000 000X XXXX 0000

Response frame CAN ID is similar:

0000 1000 0000 0000 0000 000X XXXX 0001

The part marked with Xs indicates the address bits.

■ 2.4.3 Gateway functions

A gateway provides six functions:

The first byte of a configuration request frame contains a function identifier in its upper nibble and the number of segment of the request in the lower nibble in case the request is segmented into more consecutive frames. Further content of a request frame depends on the particular function.

■ Reset

This command clears filtering rules in the inactive configuration and sets the default behaviour to block or pass frames going through the Gateway's data interfaces.

■ Block/Pass

This command adds a filtering rule to the inactive configuration. It allows to filter frames with a particular CAN identifier.

■ Modify

This command adds a rule for modification of frames with a particular CAN identifier. It has “mask” and “data” parameters that determine which parts of the frame will be replaced by which data.

The command also has an option to automatically recalculate the CRC field, which is placed at the first byte of the data in messages that use this type of security, to the correct value after data modification. For this functionality a CRC seed has to be supplied in a parameter.

Due to length of data and mask fields, which is 64 bytes each, the command has to be segmented into three consecutive messages.

■ Trig

This command toggles between active and inactive configuration. That is, the configuration being currently active becomes inactive and vice versa.

■ Stat

This command is used to monitor the Gateway status on the control interface. It periodically sends a frame with a dedicated identifier and status data.

■ Send

This command allows to send a CAN frame from the control interface to one of the data interfaces. The options provided for this functions are whether the frame is in normal or FD format and if the Bit Rate Switch bit should be set.

■ Read

This command allows to forward messages with a given identifier from a selected data interface to the control interface.

■ 2.4.4 Design of the interface

The controlling software should provide such an interface to its users that they can control a gateway in a straightforward manner. The software API should provide a method to reset the gateway to a defined state, a group of methods to add, remove and apply the filtering rules and a method for sending a single frame.

The send method is quite simple as this is an one-shot action, which is carried out by a single function.

■ Hard Reset

An important functionality is to have a way to bring the Gateway to the default state at any point of its operation.

The Gateway does not offer a single function for that. The Reset command only clears the filter rules list.

Clearing all rules by the Reset command and then switching to this clear configuration by the Trig function can achieve the default state.

■ Filter rules management

Gateway does not provide any functions to read back current configurations, nor it supports removal of single rules once stored in the current configuration. Rules removal can be achieved only by Reset command, but this removes all the rules from current (inactive) configuration.

Due to these properties the most convenient way the controlling software would handle filtering is to store the configuration within its internal data structures. Every time a rule is added or removed, the software has to first clean the configuration in the gateway by the Reset function, then apply all the rules stored in its memory and finally enable the configuration by the Trig function.

For setting up the filtering rules, the API exposes add and remove methods for each of block/pass, modify and read filter types.

Then a commit method applies all the rules to the Gateway and enables the configuration.

Chapter 3

Software design

In the previous chapter, the general architecture of the system and its main parts, which are the communication protocol with the master control system and a method of controlling the CAN Gateway, were proposed.

This chapter deals with design of control software that will implement those functions.

3.1 Control application architecture

The first topic to consider in this section is the overall architecture of the control application. The control software fulfills the task to mediate functions for controlling the vehicle to the Master Control System. These functions are carried out by sending and reading messages on the CAN interface.

From an algorithmic point of view, there are apparently no computationally complex tasks required. The module will be idle most of the time. Requests from MCS will come only once in a while. Rather than effective algorithms, reliability and proper logical decomposition of software components is important. The design should also focus on configurability and extendability of the software. Of course, it is also important to provide clear logging to allow observing the software operation for easier debugging.

Another important matter to think about is the flow of control, that is, how the code will be executed.

One method would be to make the application multi-threaded and run each component in its own thread. It would be then necessary to ensure correct synchronization between threads by means of thread-safe FIFOs and carefully lock shared data structures with mutexes. This approach seems too cumbersome for this application. It is rather suitable for software that has higher demands on performance and scalability.

Event-driven approach. For a control software, which has to be reactive to possible many events from various sources, it is more suitable to choose an event-driven approach and design the application around an event loop and use just a single thread.

An event-driven software ([50]) has inverted flow of control, which is defined by occurring events rather than by a sequence of statements in the code. A

significant consequence of using event-driven architecture is non-blocking and asynchronous nature of the program. Rather than calling a function, waiting for a result (possibly for an undefined amount of time), and then returning back, the caller at first installs a handler, then he calls a function returning immediately. After a while, when the result is ready, the installed handler is called informing the caller about completion.

Based on these considerations, the application architecture was designed. A diagram of main components with sketched flow of data is on Fig.3.1. The components in the diagram are described in the following sections. An event loop is not depicted on the diagram as it has no influence on how the components communicate with each other.

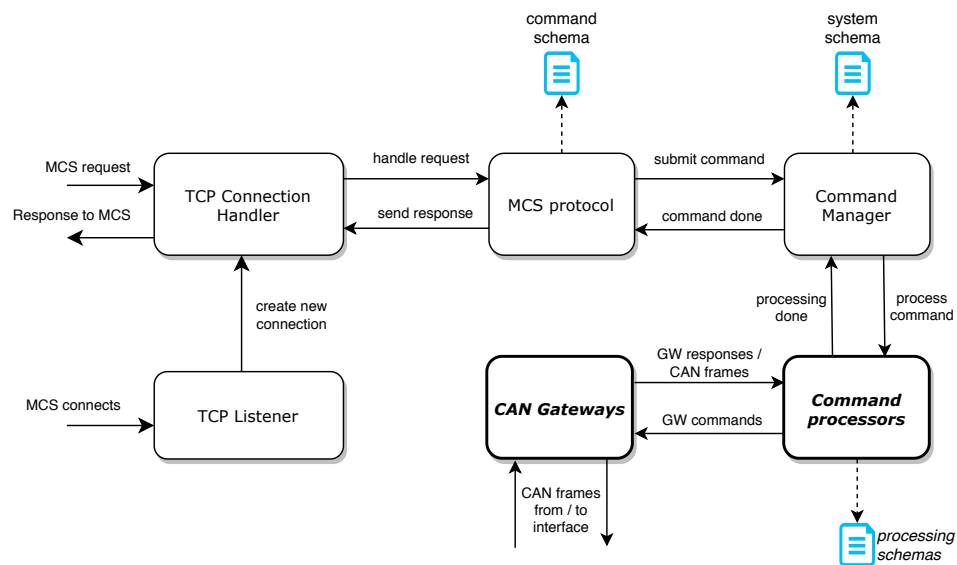


Figure 3.1: SW architecture of CAN Activator application

A sequential diagram for the proposed architecture describing execution of a single request from the Master Control System, is shown on Fig. 3.2.

3.2 Event Loop

Event Loop is a central component. It provides its users with an API to register and unregister custom event handlers. An event handler is a function, which is associated with a source of events and a list of events of interest. The source of events can be a network socket, a timer, or another system object represented by a file descriptor.

Event loop waits in a blocking system call. If an event occurs on any file descriptor, the blocking call returns with the information about on which file descriptors and which events occurred. Every time an interesting event or an error occurs on the file descriptor, the associated event handler is called.

In Linux, there are generally three options around which an event loop can be designed. Widely used on POSIX-compatible systems are “select”

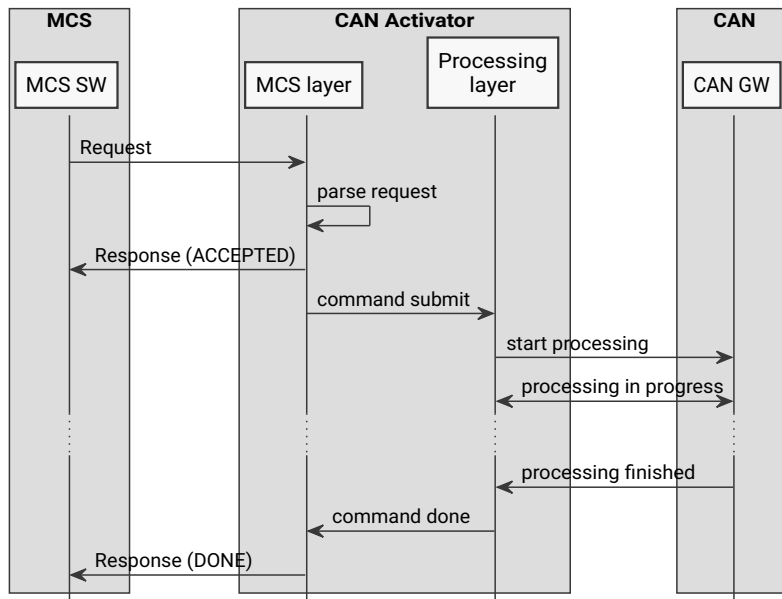


Figure 3.2: Execution of one MCS request

or “poll” system calls (described in [41, 38]). Their advantage is portability, but they have performance drawbacks when using with a large number of file descriptors ([21]). Third option is “epoll” (described in [20]), which is available only on Linux.

EventLoop component is designed around “epoll” just because it provides nicer API than “poll” or “select”. Performance reasons are not important at all in such application. If portability was an issue, it can be easily reimplemented using e.g. “poll” and without any change of EventLoop API.

■ 3.2.1 EventedFd

To allow registering components into the Event Loop, a base type “EventedFd” is provided. It encapsulates the underlying file descriptor and carries out resource management, i.e. automatic deregistering from Event Loop when the parent object goes out of scope. It can be inherited by any component that represents a source of events.

■ 3.3 CAN Components

■ 3.3.1 CAN Socket and Interface

CANSocket class is an EventedFd (3.2.1) wrapper around a “socketcan” socket. It provides basic I/O functions for reading and writing both CAN classic and FD frames. It is not intended to be used directly.

CANInterface class is the main component for accessing a single CAN

interface. It encapsulates CANSocket and carries out much of event handling on that socket. This component allows its users to send CAN frames and register a receive handlers for CAN frames with a specific ID. It also logs all frames going from and to this CAN interface.

■ 3.3.2 CAN Gateway

The control of CAN gateways is provided through two classes.

■ CANGatewayModule

This component implements low-level functions of a single CAN Gateway module.

Its main responsibilities are:

- to perform serialization of the configuration commands,
- segmentation of multisegment commands,
- providing each message with a header,
- sending messages to the CAN interface
- receiving responses from the CAN interface

This class is not intended to be used directly but rather composed into CANGateway component, which is discussed below.

■ CANGateway

A component that provides an API to a CAN Gateway. It wraps around the CANGatewayModule and maintains all the data structures needed to track the gateway state, it remembers applied configuration and creates friendly and more natural interface to users.

It stores all the filtering rules and provides methods to add, remove, or update them. These methods just manipulate with the internal state of the component, not the gateway itself. Only when the commit of rules is called, the component applies them to the gateway.

This component also provides a way to reset the gateway to a defined state, since the gateway itself does not support a single command for that. This is achieved by erasing all the rules both internally and in the gateway by issuing the Reset command. Then the Trig command switches the gateway to that empty configuration.

Committing of filter rules. The Reset command is issued at the beginning, so the process starts with a clean configuration. Then all the stored Block/Pass, Modify, and Read rules are applied. The whole process is carried out in asynchronous manner, as demands the event-driven architecture discussed earlier (3.1). That is, after application of each single rule, the control transfers

back to the EventLoop and CANGateway component continues operation when it receives a response from the gateway.

When all the rules are applied, Trigger command is sent to activate current configuration. Upon receipt of the final ACK response from the gateway, the completion handler is called to inform the user that the commit has been completed.

■ 3.4 Master Control System Interface

MCS communication layer comprises of a network part and a part for parsing requests and generating responses.

■ 3.4.1 TCP communication

The network part is made by two parts. TCPListener fulfills a function of a TCP server. Its task is to accept a new connection and pass its socket to TCPConnectionHandler. The protocol is designed in such way that at most one connection is possible. Whenever there exists a living connection, any new connection is closed right after it is accepted.

TCPConnectionHandler manages the connection socket. It carries out any event handling on the socket. On a *readable* event, it ensures correct reading and splitting the data into frames divided by the delimiter and passing them one by one to the parser.

This component also maintains a connection timer, which is refreshed with each incoming request. If it expires, the connection is considered to be dead and closed.

Writes of responses are handled directly without waiting for a write event. This method is not generally correct, because the “write” system call ([51]) on a nonblocking socket can actually write less than the required size to be written if the TCP send buffer is full.

However, it should be okay to use this method here, because the maximum size of a response is constrained by a smaller value than the TCP send buffer size (which has default size 16 KB and minimum default size is 4 KB according to the manual [45]) and the response data are sent synchronously with respect to requests, so there is no chance to fill up the send buffer.

A generally correct approach for sending a data on a nonblocking socket would most probably be (based on experiments and comprehension how the poll or epoll works) to have an internal buffer for data to send and register the socket for *writable* events. On each such event, as much data as possible would be written to the socket. If all the data from the buffer is sent, the *writable* are removed from the events of interest until there is any new data in the buffer.

■ 3.4.2 Request and response handling

This task is carried out by the ClientProtocol component.

accepts a reference completion handler, which it calls when the processing is over.

Such general design allows one to implement a number of processing algorithms. A particular algorithm may be possibly implemented for a group of commands if their processing has common features. There can be more processors for some commands. That one which to use is chosen in *command schema* configuration.

Each processor can have its own *processing schema* from which it loads a configuration (3.7.3).

All the processors are instantiated in Command Manager. Within the configuration files, they are referred to through string identifiers hardcoded in the Command Manager. Introducing a new processor is also easy.

■ 3.6.1 General processing procedure

A common actions of a comand processor taken after the start of processing a command are:

Initialization of its data structures, finding the correct CAN Gateway and Interface, setting response and error handler on the Gateway, setting a CAN frame read handler if needed, firing up a timer for detecting processing timeout and starting a communication with the Gateway.

Then the processing is carried out in an asynchronous manner. After a while the processing succesfully finishes or an error or a timeout occurs.

Command state and info string are set according to the processing outcome, the response handlers are deregistered, the Gateway is reinitialized to the default state, and the completion handler is invoked to inform the Command Manager.

■ 3.6.2 Commands for the RPC protocol

In an introductory section, there was a mention about a proprietary protocol used for a RPC-like communication of the vehicle control units. A general description of the processing procedure follows. Details of the protocol are not disclosed.

This command processor is designed for a set of commands that use CAN frames of that protocol to control in-vehicle functions. These are the air conditioning and some infotainment functions.

■ Processing algorithm

The functions of this protocol are processed in a uniform way. They only differ on parametrization. So the processing is designed such that only common procedures are hardcoded and the rest is defined in configuration files.

The procedure implements a read-modify-write operation, so only the “send” and “read” functions of CAN gateway are needed. A state diagram is on Fig. 3.3.

A brief description of the states follows:

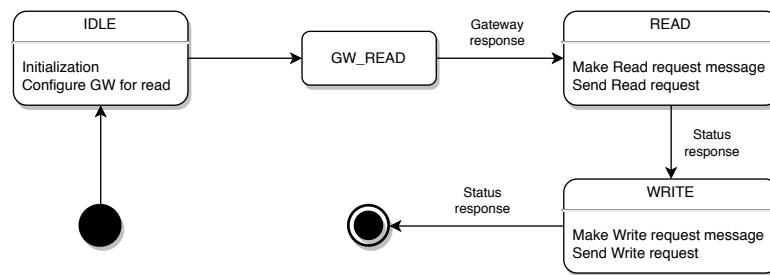


Figure 3.3: State diagram of the processing of RPC-protocol commands

IDLE. The CAN Gateway’s identifier and all other parameters for the current command are found in this command’s descriptor of the processing schema. CAN Gateway is then programmed to get status messages of the RPC protocol from a vehicle control unit.

GW_READ. This state is waiting for the response from CAN Gateway.

READ. When the Gateway acknowledges the committed rule, a message with a read operation of the RPC protocol is created and sent to the vehicle network through the Gateway. A status response is expected as an answer to this message.

WRITE. After obtaining the status response, it is transformed to a write operation message and some of its parameters are modified. Which parameters to set is defined in the processing schema.

■ Special cases

If any command needs a special handling, the processing can be implemented separately. This is e.g. case of the “air_volume” command, whose observed behaviour was irregular. A correct solution for the “air_volume” command would be to first set its value to a different value and then to the setpoint value.

■ Configuration

Runtime configuration of the processor for the RPC functions is stored in two files.

Processing schema. Here are definitions for each of the commands, which CAN Gateway to use and mappings to fields and parameters of RPC-protocol messages.

A mapping to the RPC message parameters can be defined from a constant value or a command parameter.

There are three types of mapping from command parameters:

- **identity**, which is a simple 1:1 binary mapping,

- **linear** with a scale and offset,
- **tab**, which is defined by an enumeration.

For both types of mapping a destination, which is a parameter of a RPC message defined in Functions schema (described below), can have a bitfield specifier if one needs to write only to some part of the parameter.

Functions schema. This schema contains definitions of the RPC protocol messages according to their specification together with request and response messages identifiers, which come from DBC files.

■ 3.6.3 Commands for ACC and Speed Limiter

ACC and Limiter functions are controlled by cyclic messages as follows from the analysis in section 1.6. It has been already stated that there is one control messages indicating the state of the control lever, one status message for ACC and one status message for Limiter.

The algorithms here perform their actions by modification of particular bits in the control lever control message thus simulating real clicks and switches of the lever. Algorithm decisions are made upon information in those two status messages. CAN gateway “modify” and “read” functions are utilized for these tasks.

Provided commands (listed in Tab. 3.1) are designed such that they can enable a given function, only if the current state of ACC or Limiter allows it.

Table 3.1: List of commands for ACC and Speed Limiter control

Command	Parameters
ACC enable	speed
ACC disable	–
ACC set distance	distance
Limiter enable	speed
Limiter disable	–

If the entire ACC / Limiter is off (the control lever is switched to position OFF), no control by a command is allowed. If the control lever is switched to ON, either ACC or Limiter mode is currently selected (a grey icon on the display). In this default state, any of ACC or Limiter commands can be activated except of ACC set distance, which only works in ACC mode.

If the ACC is active (indicated by a green icon on the display), only the ACC functions take effect. For switching to the Limiter mode, ACC must first be deactivated by ACC disable command. Likewise if the Limiter is active, only the Limiter commands are applicable until it is deactivated by Limiter disable command.

These rules are expressed in the state diagram 3.4.

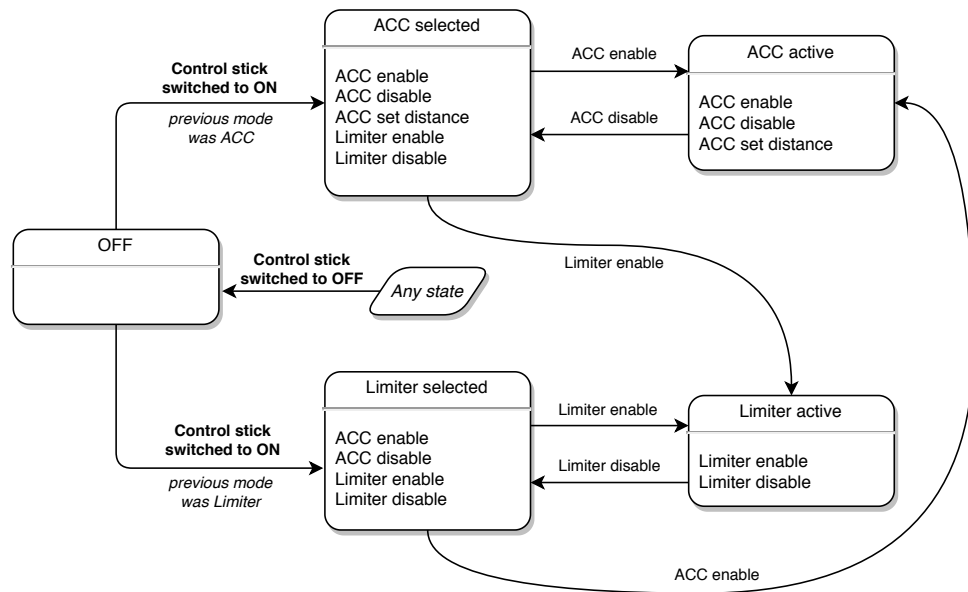


Figure 3.4: State diagram of ACC and Limiter states. Within each state are listed commands that are applicable in that state.

■ ACC and Limiter enable

These commands have the task to set target speed and to activate the function, either Limiter or ACC.

The processing of both functions is carried out by the same way. A flowchart on Fig. 3.5 describes the algorithm.

Current state and setpoint speed of a given function is indicated in cyclic status messages.

At the beginning, the status of the function (either ACC or Limiter) that is the subject of the control is read.

If the function is already active, the speed can be set directly. If the function is not active, but it is selected, the setting of target speed is performed in two steps.

Preparing the speed and activating. Firstly in the “Prepare speed” stage, the speed is set to the nearest lower or equal value of the target speed only using the UP (+10 km/h) and DOWN(−10 km/h) signals of the control lever control message. This operation is added due to safety reasons. If the current value on ACC is much higher than the setpoint value, and ACC is activated, the vehicle could start accelerating unexpectedly.

When the rounded value of the target speed is reached, the function can be activated by clicking on the RESUME of the control lever. The RESUME also has the function of increasing the speed by 1 km/h. This is why the exact target speed cannot be set before activating the function. After the function is activated, which is determined by checking the status message, the exact speed can be set by iterating by 1 km/h.

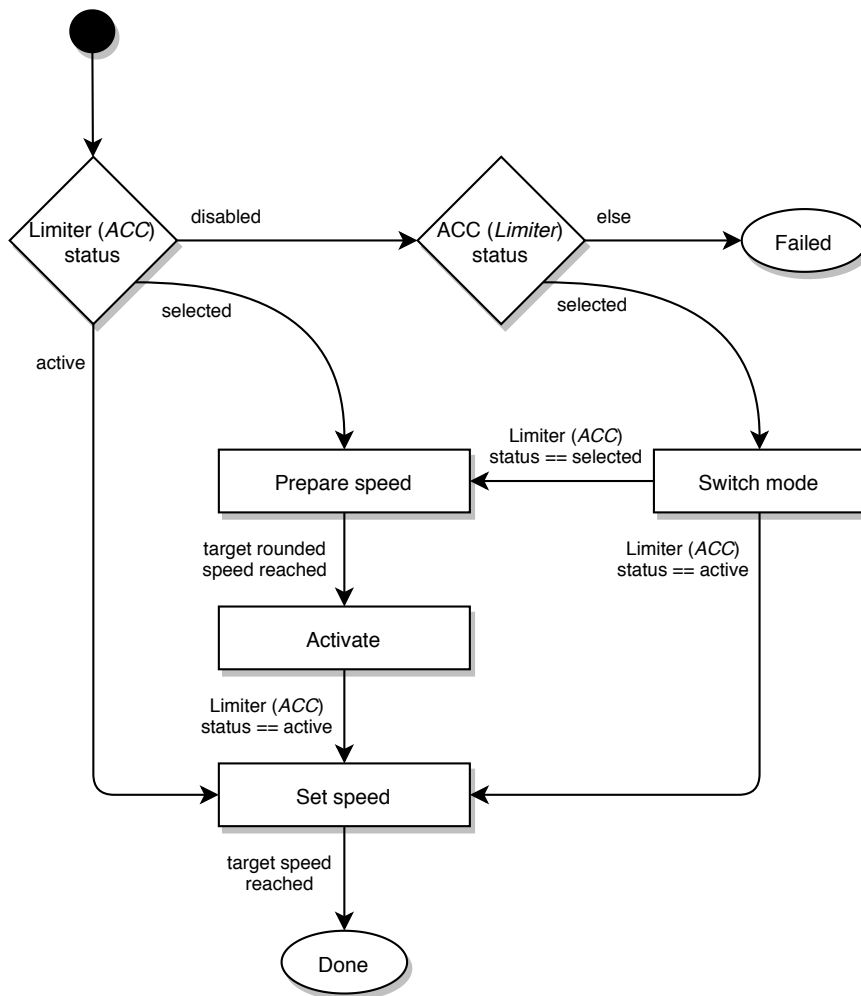


Figure 3.5: Processing of Limiter enable and ACC enable commands. Labels in parentheses apply for ACC enable

Switching the modes. If the function to be controlled is disabled at the beginning, it has to be selected before setting the speed and activation.

At first it is determined whether it is even possible to switch the mode. A status message of the second function is read. If the status is active, the mode cannot be changed due to the rules defined earlier (see Fig. 3.4). When the status is disabled, it means that both Limiter and ACC are disabled, which happens iff the control lever is in the position OFF. If the status of the other function is selected, the mode can be switched. This is performed by triggering LIMITER signal on the lever until the status changes to one of the required values.

Setting the target speed. In the active state of either functions, reaching the target speed is accomplished by first getting to the closest smaller value, by moving UP or DOWN by 10 km/h and then by using the ± 1 km/h signals of the control lever. Details of the algorithm to reach the target speed

are discussed in the next paragraph.

Process of setting a target value. There are three general ways to reach a target value by operating the control lever.

One approach is to activate and hold the control element until the target value is set and then release. This is the most simple, but unfortunately the slowest solution.

A method that is used in this application is to activate a control element until an observed value is changed, then release the control and wait for some time. The delay time was determined experimentally as the time of three consecutive status messages, which proved to work well. A flowchart of the process is on Fig. 3.6.

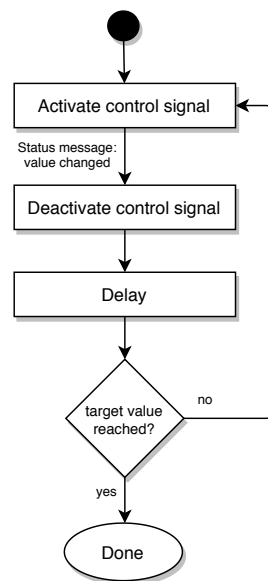


Figure 3.6: Process of setting a target value by modification of a control signal

A possible drawback of this method manifested itself while the Limiter (or ACC) was in an intermediate state. Even if a control signal for speed change was active, the speed was not changing for several seconds until the intermediate state went away.

There is a possibility that this unpleasant behavior could be evaded by using short clicks on the control element instead of activating and waiting for a change of the value. Unfortunately, this idea has not been examined yet, because it was difficult to induce that intermediate state. It happened only about twice during the whole testing.

■ ACC and Limiter disable

Disabling both of the functions is carried out by activating CANCEL element on the control lever and waiting for the state in a status message to change from active to any other.

At the beginning a check is made to ensure the actual state of the function is active. If not, the function is considered to be already disabled.

■ ACC set distance

Setting a distance of the preceding vehicle is only possible within ACC mode (cf. rules on Fig. 3.4).

It is carried out by controlling DISTANCE+ and DISTANCE− signals of the control lever and observing actual value in ACC status message. The method of reaching the target value was described in a previous paragraph (3.6.3).

■ 3.7 Runtime configuration

Much of the runtime configuration for the control application is stored in JSON schema files rather than hardcoded. This decision was proposed to facilitate high configurability and to minimize the need of rebuilding the software every time some configuration changes. JSON format seems to be suitable for storing configuration and it is already used for implementation of MCS commands protocol.

■ 3.7.1 System schema

The System schema contains a description of available CAN interfaces on the board and connection of CAN gateways. An example of System schema is exposed in Listing 3.1. The “interfaces” property has an array of CAN interface names. These names are strings that match the interface names in the system. The “gateways” property has a list of JSON objects, each of them defining each CAN gateway.

The “name” property contains a string, which serves as an identifier for a specific gateway in the control application. Every CAN gateway is referred to and accessed by means of this identifier. The “interface” and “address” fields uniquely identify each gateway in the system.

The “interface” is one of the strings from the top-level “interfaces” list in the schema and “address” is a string with “0x”-prefixed hex number. Gateway addresses are 5-bits as mentioned in section 2.4.2.

Listing 3.1: System schema example

```
{
  "interfaces": [ "can0" ],

  "gateways": [
    {
      "name": "KCAN",
      "interface": "can0",
      "address": "0x1f"
    }
  ]
}
```

```

    },
    {
      "name": "ICAN",
      "interface": "can0",
      "address": "0x1c"
    }
  ]
}

```

3.7.2 Commands schema

The schema defines properties of a request for each command, i.e. command name, parameters, its data types and valid values. The schema is used by the request parser to validate and parse a request from the Master Control System.

The structure of the schema is shown in Listing 3.2. On top level there is a JSON object containing key–value pairs of records, where each key is a command name and the value is a JSON object describing the command. This command description contains key–value pairs for each parameter.

A parameter descriptor contains a mandatory “type” property, which can take one of values “bool”, “int”, “float” or “string”. The next property in the parameter descriptor is an optional validation specifier, which is either “enum” or “ranges” key with a value of an array. In case of enum, this array contains an enumeration of valid values. In case of ranges, it contains a list of inclusive ranges of valid values. If the type is “string”, the only legal validation specifier is “enum”. For “int” or “float” type, it can be either “enum” or “ranges” and “bool” parameters have no validation specifier. The type of values in the validation specifier must match the parameter type.

The “proc_name” field specifies which Command processor to use to process this command. The available command processors are instantiated in the CommandManager component (discussed in section 3.5).

Listing 3.2: Commands schema structure

```

{
  "command_1": {
    "params": {
      "param_1": {
        "type": "int",
        "enum": [ 0, 1, 2 ]
      },
      "param_2": {
        "type": "float",
        "ranges": [[ 16.0, 29.5 ]]
      }
    },
    "proc_name": "command_processor"
  },
}

```

```

    "command_2": {
        ...
    },
    ...
}

```

3.7.3 Processing schema

Processing schema describes the particular way the commands are processed. Each Command processor can use its own processing schema, so the syntax is not uniform.

There can be a general part with common or default options as timeout, associated CAN gateway, target interface on the gateway where to send and receive messages, etc.

If a specific “commands” part is present, a recipe for processing each command describes function codes, CAN identifiers and conversion of Command parameters to data fields in messages of the protocol used on the CAN bus.

3.8 Logging

Proper logging is important not only in the development phase of the project, but also during its use. The role of logging layer is to provide as much information as possible to track system behavior, but only as much as is necessary. It is also important to have these information preserved in case of application or system failure.

Requirements for the logging subsystem are:

1. Log messages should not be lost when the control application is terminated.
2. It is not desirable to save possibly large and multiple log files to SD card.
3. Log content should be accessible from a remote machine.
4. Logging application cannot be blocked in case nobody is reading the log messages.

Obviously, to fulfill point 1, it is necessary to keep log messages outside the control application itself. However, following point 2, redirecting logs to files is not a good solution.

To use some system object like a named pipe[22] or a message queue[33] alone would not be enough, because messages are lost once read from these channels. Not even shared memory[43] is optimal, because it does not provide FIFO semantics.

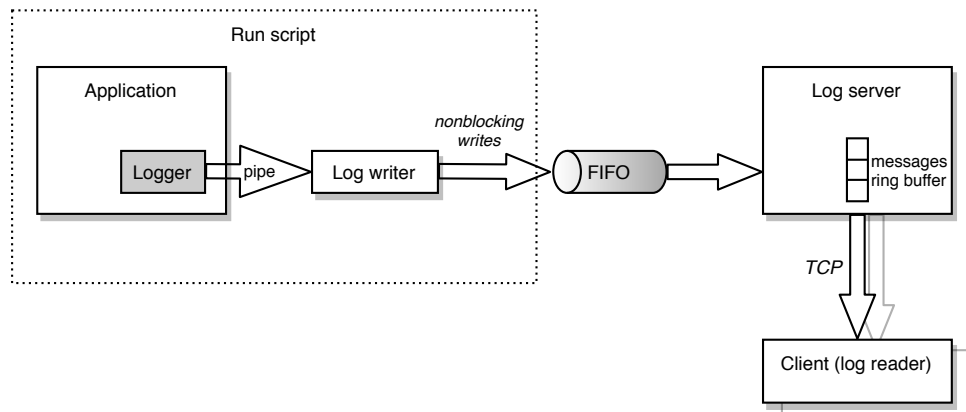


Figure 3.7: Logging solution

The most suitable approach seems to be to have a dedicated application, store log messages in its memory and provide ways to fetch all messages on demand from there.

The logging subsystem is designed with regard to these principles. It consists of three parts. All parts together are visualised on Figure 3.7.

■ 3.8.1 Logger

Logger is a component in the control application itself. It provides uniform way to emit log messages with information about system timestamp, the log level of a message in terms of severity. Supported severity levels are: INFO, WARNING, ERROR, and DEBUG. The Logger can be set up to filter messages of some severity or higher. For WARNING or higher levels, more verbose information about current function or method is also shown.

The Logger is designed in such way it doesn't take care of routing messages to (possible multiple) destinations. The messages are simply put on standard output and further processed by another component.

■ 3.8.2 Log Server

Log Server is a standalone application, which maintains a circular buffer of text lines. On one end, it reads incoming data from an external FIFO and saves it into the circular buffer. An access to the ring buffer is provided over a TCP connection. Every time a client connects to the Log server, it flushes the entire buffer to the client and as new messages arrive in the buffer, they are immediately sent to all connected clients.

■ 3.8.3 Log Writer

■ Background

Messages are passed to the Log Server via a named pipe (described in [22]). This is a common way for one-direction IPC in Unix environment ([37]). The

standard procedure to initialize such communication channel is:

1. Create a FIFO special file in a filesystem.
2. A reader process opens the FIFO for reading and a writer process opens the FIFO for writing.

The “open” system call in each process blocks until both sides of the FIFO are opened. This behavior can be avoided by opening the FIFO in *nonblocking* mode. In that case, “open” call in reader process returns successfully without blocking. In writer, “open” returns success only if the reading end has been opened, otherwise, it returns error.

3. Perform I/O.
4. If all reading ends are closed and a writer tries to write to the FIFO, it gets the SIGPIPE.

One problem that can occur is when the process on the writer side terminates and it is the only writing process to the FIFO, the reader would get failure upon reading and it would have to reopen the FIFO.

This can be solved by opening the FIFO for writing on the reader side, so there is at least one writer and one reader at a time.

The second problem that can occur is, if no reader has the FIFO opened or nobody is continuously reading from the FIFO, the writer side could block, which is forbidden. It could be avoided if *nonblocking* mode was used on the writer side, but this cannot be achieved directly. This is because the output from the Logger (the main application) has to be redirected from the standard output to the FIFO by a shell redirection (described for bash in [7]). But the shell redirection itself does not provide a way to open a destination file in nonblocking mode.

■ Solution

These problems are addressed by introducing Log Writer, an utility that opens the FIFO in nonblocking mode and forwards any input towards it. If a write to the FIFO fails, Log Writer handles the error transparently and reopens the FIFO on the fly (if possible). A logging application, which is providing the input to Log Writer, therefore does not need to care about accessing the FIFO and handling error conditions.

Chapter 4

Integration of components to Terasic DE0–Nano–SoC board

In previous chapters the HW platform for the main CAN Activator module was selected. It was also decided on which CAN controller will be used for connecting to CAN portion of the system. The architecture of the main controller software was designed as well. Now it is time to prepare the operating system and runtime environment on the development board and integrate all those parts there.

Procedures, parts of scripts and commands were partially taken and modified according to needs and experiments from the Rocketboards.org site [3] and some other information from Xillybus pages ¹.

4.1 Preparation of the development board

4.1.1 Boot process description

The boot process of the Terasic DE0–Nano–SoC is described in [39, 23]. The exact sequence of the boot can differ depending on particular system and application. Here the description corresponds to this particular system.

After the device is turned on, the CPU in the SoC starts to execute code in its boot ROM. This code performs some initialization steps and determines a boot source by reading a value from pins on the board. Usually a boot source selection is made by a jumper, which allows an user to select between SD card, integrated flash memory, or perhaps another source. This system uses SD card as a boot media.

The boot ROM code then loads the Preloader (or First Stage Bootloader) from a known location on the SD card and executes it. The Preloader contains further initialization code that is dependent on a custom HW design, like clocks configuration, setup of on–chip peripherals, watchdog, etc.

When Preloader finishes the initialization of the SoC, it runs a “Second Stage Bootloader”, which is U–Boot in this case.

Das U–Boot ² is highly customizable, advanced open–source bootloader,

¹<http://xillybus.com/tutorials>

²<https://www.denx.de/wiki/U-Boot>

which is often used on embedded platforms that boot an operating system rather than a bare-metal application. It supports many platforms and provides default configurations for them.

U-Boot proceeds according to a boot script if present on the SD card. It executes these steps:

1. Load a bitstream with the HW design and program it to the FPGA
2. Load the device tree to the memory
3. Load the Linux kernel image to the memory
4. Boot the kernel with the device tree address and boot parameters specifying root filesystem type and partition, configuration of serial console, and possibly other options

The kernel then starts a normal boot process.

4.1.2 SD card partitioning and contents

Information are taken from [27].

The SD card contains (at least) three partitions:

1. Boot partition – The first FAT32 partition on the device contains these files:
 - **u-boot.img** – U-Boot image
 - **boot.scr** – U-Boot script
 - **soc_system.dtb** – compiled device tree
 - **soc_system.rbf** – FPGA bitstream
 - **zImage** – compressed Linux kernel

Size of the partition can be in tens of megabytes.

2. A partition of type A2 (set in fdisk utility) and no filesystem. It contains binary image of the Preloader with bootROM header. Its size can be as small to contain the preloader image (which has exactly 262144 bytes).
3. An Ext3 or Ext4 partition with the root filesystem.

4.1.3 Making the artifacts

Here is described a process of making each artifact on the micro SD card.

Needed prerequisites are:

- Quartus Prime IDE with Embedded Command Shell.
- GNU Cross-compiler for ARM little-endian / gnueabi (gcc-arm-linux-gnueabi).

- Altera fork of U-Boot source at: <https://github.com/altera-opensource/u-boot-socfpga.git>
- Altera fork of Linux kernel at: <https://github.com/altera-opensource/linux-socfpga.git>

■ Bitstream file (`soc_system.rbf`)

Information are taken from [24].

The bitstream is a binary file with compiled HW design that can be loaded to FPGA. It is produced by Quartus Prime IDE.

As a start point for making any design, a default design from the “DE0-Nano-SoC CD-ROM (rev.D0 Board)”³ can be used. A design in directory (Demonstrations/SoC_FPGA/my_first_hps-fpga_base) on that CD proved to work.

After succesful compiling the HW design, the important result files are:

- `soc_system.sof` – the compiled HW design
- `soc_system.sopcinfo` – can be used for creating the device tree
- `hps_isw_handoff/` – a directory with configuration needed for building the preloader

To be able to load the bitstream from the bootloader, it is necessary to convert `soc_system.sof` to `.rbf` format at first. This can be done from Quartus Prime by following these steps:

1. In main menu select File – Convert Programming Files
2. Choose `.rbf` output
3. Choose proper mode according to MSEL pins on the board (FPPx16), see [17].
4. Generate `soc_system.rbf`

■ Preloader

Information comes from [25, 39].

Some commands in this section need to be executed within the *Embedded shell*, which is a shell environment for development tools shipped with Quartus IDE. It is located at “`intelFPGA/18.1/embedded/embedded_command_shell.sh`” provided that “`intelFPGA`” is the installation directory of Quartus IDE.

³http://download.terasic.com/downloads/cd-rom/de0-nano-soc/DE0-Nano-SoC_v.1.3.2_HWrevD0_SystemCD.zip

Generate the Preloader. Preloader has to be configured at first. In the following paragraphs it is assumed that “design_root” is the root directory of the Quartus HW design project.

Following command will generate the Preloader sources in “spl_bsp” directory. The support for the FAT filesystem has to be explicitly enabled so the Preloader can load next boot stage from SD card FAT32 partition.

```
[embedded-shell] design_root $ cd software &&
  bsp-create-settings --type spl --bsp-dir spl_bsp \
  --preloader-settings-dir ../hps_isw_handoff/soc_system_hps_0/ \
  --settings spl_bsp/settings.bsp \
  --set spl.boot.WATCHDOG_ENABLE false \
  --set spl.boot.FAT_SUPPORT true
```

Compile the Preloader. This command creates “preloader-mkimage.bin” file:

```
[embedded-shell] design_root $ cd software/spl_bsp && make
```

The final step is to binary copy the preloader image to the A2 partition on the SD card. One should be careful about writing to the correct partition!

```
# dd of=/dev/mmcblk0p3 bs=512 if=preloader-mkimage.bin ; sync
```

■ U-Boot image (u-boot.img)

This project uses a fork of U-Boot source that is provided by Altera/Intel. It contains default configuration for the Altera’s board/platform. The version used in this project is: “tag: v2019.04”

The U-Boot image was created by these commands:

```
$ git clone http://github.com/altera-opensource/u-boot-socfpga
$ cd u-boot-socfpga
$ git checkout v2019.04
$ export CROSS_COMPILE=cross-toolchain-prefix-
  Cross toolchain is a tuple in format
  'arm-vendor-linux-gnueabi-' depending
  on the concrete toolchain used by the host
$ export ARCH=arm
$ make mrproper
$ make socfpga_cyclone5_config
$ make
```

The result file u-boot.img is in the root directory.

This procedure is usually made once, or only when it is needed to update U-Boot.

■ Boot script (boot.scr)

Boot script contains boot commands for U-Boot. It is an ASCII file with the U-Boot header prepended on it.

The following bootscript is used in this work (based on information from [15]):

```

echo -- Loading bitstream
setenv fpgadata 0x2000000
fatload mmc 0:1 $fpgadata soc_system.rbf

echo -- Programming FPGA
fpga load 0 $fpgadata $filesize

echo -- Enabling HPS <=> FPGA bridges
bridge enable

echo -- Loading linux image
setenv bootimage zImage
setenv loadaddr 0x100000
fatload mmc 0:1 $loadaddr $bootimage

echo -- Loading device tree
setenv fdtimage soc_system.dtb
setenv fdtaddr 0x1000000
fatload mmc 0:1 $fdtaddr $fdtimage

# Note: bootargs can be set in device tree
setenv bootargs mem=1024M console=ttyS0,115200 root=/dev/mmcblk0p2 rw rootwait

echo -- Booting kernel
bootz $loadaddr - $fdtaddr

```

The boot script is created by the following command, given that u-boot.script is the input ASCII file and boot.scr is the output ([15]).

```

Switch to Embedded Command Shell at first
to make 'mkimage' tool available
$ intelFPGA/18.1/embedded/embedded_command_shell.sh
[embedded-shell] $ mkimage -A arm -O linux -T script -C none \
    -a 0 -e 0 -n boot_script -d u-boot.script \
    boot.scr

```

■ Linux kernel image (zImage)

This project uses Intel/Altera Linux fork, which adds some patches and additional drivers for SoCFPGA platform. A real-time version of the kernel is used, with CONFIG_PREEMPT_RT_FULL=y config option as this is recommended for CTU CAN FD IP Core in Linux (this information was provided by P. Pisa in personal communication). The RT patched kernel is included in the Altera's kernel repository. The version used in this repository is "tag: rel_socfpga-4.14.126-ltsi-rt_19.11.01_pr"

The following procedure produces the kernel image:

```

$ git clone https://github.com/altera-opensource/linux-socfpga
$ cd linux-socfpga
$ git checkout rel_socfpga-4.14.126-ltsi-rt_19.11.01_pr
$ export CROSS_COMPILE=cross-toolchain-prefix-
  Cross toolchain is a tuple in format
  'arm-vendor-linux-gnueabi-' depending
  on the concrete toolchain used by the build host
$ export ARCH=arm
$ make socfpga_defconfig
  Instead of 'make socfpga_defconfig' it is possible
  to run 'make nconfig' and load a custom
  configuration if provided
$ make zImage

```

The result artifact is created in “arch/arm/boot/zImage”.

The procedure of making the kernel image is usually done once, or when a new version or a patch is introduced.

■ Device tree (soc_system.dtb)

Firstly, there has to exist the DTS file:

```
arch/arm/boot/dts/socfpga\_cyclone5\_de0\_nano\_soc\_canfd.dts
```

Procedure of making this DTS file for CTU CAN FD IP Core is described in section 4.2.2.

Compiling the device tree source (DTS) into the device tree blob (DTB) is done simply by running this command in kernel root:

```
$ make ARCH=arm socfpga_cyclone5_de0_nano_soc_canfd.dtb
```

This creates “arch/arm/boot/dts/socfpga_cyclone5_de0_nano_soc_canfd.dtb” file, which can be transferred to the SD card and renamed to “soc_system.dtb”.

■ Root filesystem

Root filesystem contains the operating system data, settings, tools, programs and libraries, which is all necessary to have fully functional environment for running user applications.

There are several options for getting a root filesystem. It can be built from scratch, which option is provided by projects like Buildroot⁴ or Yocto⁵. This choice is more suitable for final customer embedded products, where it is desirable to have the entire process of creating the distribution fully under control.

⁴<https://buildroot.org/>

⁵<https://www.yoctoproject.org/>

Another way to obtain a root filesystem is to use some existing Linux distribution. This project uses Arch Linux ARM, port of Arch Linux for ARM architecture ⁶. It is a rolling type of release, which means upgrades are on daily basis so one does not need to care about support periods for a certain version and upgrading the whole distribution possibly breaking some configuration.

Altera boards are not on the list of supported platforms of this distro, but it is possible to use the release for ZedBoard, which has Xilinx Zynq SoC, which is the same architecture as Intel/Altera SoC (ARMv7 Cortex-A9)⁷.

The downloaded archive is just extracted to the Ext3/4 partition of the SD card.

4.2 Integration of CTU CAN FD IP Core

CTU CAN FD IP Core can be obtained from a CTU git repository ⁸.

Choose top level design (a default design from the board CD – see section 4.1.3).

4.2.1 IP block with Avalon interface

The first task is to create an IP core from HDL sources that could be integrated into the HW design in the Quartus IDE.

Preparation of sources

At first, the “update_reg_map” script in the “scripts” directory has to be run to obtain the source files that are not hardcoded, but rather generated from a register map description.

The second step is to run a script in the “scripts” directory that collects all the HDL source files in a single directory:

```
$ python3 create_release.py --output_dir "../release"
```

Creation of the IP block

Now the files in the “src” directory under the generated release can be used to create an IP block in the Platform Designer.

Instructions and a TCL script provided in “synthesis/Quartus/ctu_canfd_avalon” directory in the repository can be used for creating the IP core with the Avalon interface. ([14])

The TCL script provides all the needed interfaces (clock, timestamp, CAN, reset, interrupt, Avalon slave) to allow integration to the Platform Designer. When it is opened in the Platform Designer Component Editor, the list of

⁶<https://archlinuxarm.org/>

⁷<https://archlinuxarm.org/platforms/armv7/xilinx/zedboard>

⁸https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core

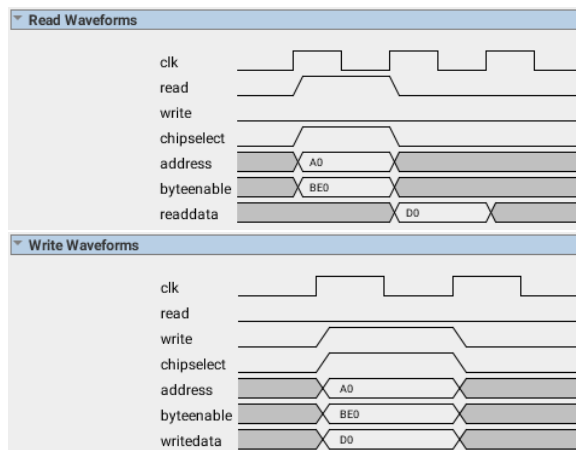


Figure 4.1: Correct waveforms on CTU CAN FD Avalon interface (taken from Platform Designer IDE)

Synthesis files can be edited (if a new file in the future is introduced or any is changed). It is just important to do not forget to set the Top-level file to `can_top_level.vhd`.

In “Signals & Interfaces” tab, it is very important to ensure the correct properties of the Avalon Memory Mapped Slave interface according to the IP Core System Architecture [29, 2.1 Memory bus]. The Address units has to be SYMBOLS, Read latency must have value 1 and Read wait must have value 0. The result waveforms can be seen on Fig. 4.1.

■ Integration to the HW design

Some information concerning this topic was provided by O. Ille and P. Pisa in personal communication.

When the IP block is created, it can be instantiated in the Platform Designer System Content. This design uses two instances of the core.

Registers interface. The Avalon interfaces of both cores are connected to the “h2f_axi_master” interface on Hard Processing System. Platform Designer creates an AXI-Avalon bridge for them. In Address mapping, a 64 KB window must be allocated for each instance, though the core itself uses only 4 KB. If the core registers of both controllers are mapped in the same 64 K window, the registers of one controller become inaccessible.

Hard Processing System configuration. In Parameters of the HPS, FPGA-to-HPS interrupts has to be enabled.

Output clock is chosen according to needed frequency. Some frequencies cannot be exactly achieved by some clock. One should pay attention to warnings in Platform Designer. In such case, it may help to use another clock that uses different PLL. In this design, HPS-to-FPGA user clock 1 with 50 MHz frequency is used. The design also runs at 100 MHz, but at this frequency, warnings from the timing analyzer appear during synthesis.

IRQ, Counters, CAN interface. A free-running synchronous 64-bit counter has to be connected to “timestamp” port of each core instance.

IRQ lines from “interrupt_sender” are connected to “f2h_irq0” port on the HPS.

The “can_interface” has to be exported for connection with top-level entity.

Top-level entity and pin mapping. After finishing setup in Platform Designer, the top-level HDL file (ghrd.v) as to be adjusted to the changes that has been made.

CAN_RX and CAN_TX signals for each CAN controller have to be added to “ghrd” module definition and then connect these signals to those in the instantiation of “soc_system”.

In Pin Planner, these signals are assigned to the correct pins (see Tab. 4.1) according to the Extension I/O board schema [35, Appendix C: 100Base-T1 board for Terasic DE0-NANO-SoC, rev. 2].

Table 4.1: CAN signals to pins assignment

Signal	Pin
CAN0_RX	PIN_AF8
CAN0_TX	PIN_AB4
CAN1_RX	PIN_AE23
CAN1_TX	PIN_AC22

4.2.2 Device tree node

Based on information from [26, 42] and [30, p. 19, Listing 1].

After the design is compiled, one of the produced files is “soc_system.sopcinfo”, which describes the specific parts of the design. There is an option to use the `sopc2dts` utility within Quartus Embedded Command Shell to generate DTS using “soc_system.sopcinfo” file and board definitions XML files, which are placed in the HW design root directory (“hps_common_board_info.xml” and “soc_system_board_info.xml”). Unfortunately, this method turned out to produce an unusable DTS (the Linux could not boot). Most probably because the board XML files provided with the default design on the board CD are broken.

However, a working solution is to use only the part of the generated device tree that describes the bus portion where the CTU CAN FD controller is connected and then to integrate this node to an existing DTS in Altera Linux kernel source tree.

The “arch/arm/boot/dts/socfpga_cyclone5_de0_sockit.dts” in the Linux kernel source can be used for this purpose. One can copy this file to the new one “arch/arm/boot/dts/socfpga_cyclone5_de0_nano_soc_canfd.dts” and add the relevant part with the CTU CAN FD controller under the existing “soc” node there.

In the added part it is needed to adjust handle references of “clocks” and “interrupt-parent” properties so they point to the right ones. It is also important to set the “compatible” property of CTU CAN FD node(s) to the correct value “ctu,ctucanfd”, so the driver can find the controller. The value of interrupt number in the “interrupts” property has to be reduced by 32, because of a mapping that the driver of the interrupt controller performs ([42]).

■ 4.2.3 Linux driver

The CTU CAN FD project comes with a Linux driver [30] for the SocketCAN Linux layer [1].

The driver sources are located in “driver/linux” directory of the CTU CAN FD repository. An ARM/GNUEABI cross toolchain is needed for building the modules. The driver is built by running this command in the driver’s source directory:

```
$ make ARCH=arm CROSS_COMPILE=toolchain-prefix- \
      KDIR=linux-socfpga-path
```

This command produces “ctucanfd.ko” and “ctucanfd_platform.ko” modules. They can be inserted into the kernel on the target system by “insmod” utility.

■ 4.3 Integration of software artifacts

■ 4.3.1 Build process

The can_activator software is a CMake-based project. It consists of the main application, configuration files, logger utility, and run script.

The only third-party library this project uses is a C++ header-only JSON library⁹. The library is provided with this project in the `third-party/` directory. The GNU C++ library (libstdc++) has to be present on the target system to be able to run the application.

To build the can_activator project a “build.sh” script is provided. It builds the project for native and ARM targets.

The following tools are required to successfully build the project:

- CMake, minimum version 3.9
- GCC native compiler, a version supporting C++17 standard (only for native build)
- GNU cross-toolchain for ARM architecture with a C++17 compiler

⁹<https://github.com/nlohmann/json>

Before building the ARM version of the project, variables “`toolchain_prefix`” and “`toolchain_tuple`” in `arm.toolchain.cmake` file have to be set to correct values.

Results of build process are listed in Tab. 4.2.

Table 4.2: Build artifacts of the `can_activator` project

Artifact	Description
<code>can_activator_run</code>	shell script for running the application
<code>BUILD_DIR/can_activator</code>	main control application
<code>BUILD_DIR/log_server/log_server</code>	logging server
<code>BUILD_DIR/log_server/log_writer</code>	auxiliary utility providing safe and nonblocking write to FIFO
<code>config/</code>	directory with JSON schemas for the main application

4.3.2 Configuration of the target system

Network address. The target board ethernet interface has to be set up for a static address. The prototype uses following configuration:

```
IP address:      192.168.1.10
Netmask:        255.255.255.0
```

Configuration of the network is provided by `systemd-networkd.service`, whose configuration is stored in “`/etc/systemd/network/`”.

Startup services. Insertion of CTU CAN FD modules to the kernel and initialization of the CAN interfaces is carried out by a custom script. It can be made to run automatically during the system startup by providing a `systemd` service that executes it. Similarly, another `systemd` service can execute the main application’s run script after the former one is executed.

These services are not provided yet with the current version of the project.

4.3.3 SW deployment to the target

When the application is built, it has to be deployed to the board together with the CTU CAN FD drivers and execution scripts.

At current state, there is no automation deployment tool or a script, and data has to be transferred over SSH to the board SD card and run manually.

The sequence of steps needed to run the CAN Activator application is following:

1. Transfer all the files listed in 4.2 to a directory on the target
2. Transfer the CTU CAN FD Linux kernel modules built in section 4.2.3 (if they are not already present on the board)

3. Within the Linux shell on the board, following steps has to be taken:

- a. Insert the modules to the kernel:

```
# insmod ctucanfd.ko
# insmod ctucanfd_platform.ko
```

- b. Bring up CAN interfaces:

```
# ip link set can0 up type can bitrate 1000000 \
    dbitrate 1000000 fd on restart-ms 100
# ip link set can1 up type can bitrate 1000000 \
    dbitrate 1000000 fd on restart-ms 100
```

- c. The CAN activator can now be run with the “config” directory path as an argument.

- d. If one wants to utilize the Log server, a FIFO has to be first created in the filesystem before the Log server is brought up:

```
# mkfifo /tmp/can_a_logger.fifo &&
# ./log_server/log_server /tmp/can_a_logger.fifo
```

- e. Then CAN activator can be run with standard output redirected to the FIFO through Log writer utility, which ensures nonblocking behavior in case no one reads the FIFO (Log server fails to start or terminates).

```
# ./can_activator | ./log_server/log_write \
    /tmp/can_a_logger.fifo
```

- f. Or the provided run script can be used. It performs redirecting output and restarting the application in a case of failure (any unhandled exception, segmentation fault, or so)

Chapter 5

Testing and Conclusion

A prototype of the Control module was developed and tested.

Note on CTU CAN FD controller. Despite the fact the core seems to be stable and is operating well, it is still under development and lacks an industrial certification.

There is still an unresolved issue with the core going to Bus-off when transmitting a CAN frame on a single-node network. However, this is probably a matter of synthesis process in Quartus Prime, not a bug in the core itself.

Regarding the Linux driver for the CAN controller, it seems to work well. During its integration, some bugs were fixed and it was split to use separate modules for PCI and platform bus implementation. Now it is in the process of integrating into the mainline kernel.

Client for testing. For testing and demonstration purposes a simple testing client GUI application was developed. The tester includes the heartbeating mechanism (described in section 2.3.3) required by the CAN Activator – MCS protocol to keep the connection alive.

It offers a simple logging window with nice formatting of messages marked with timestamps, colors and an option of hiding/showing of heartbeat messages.

The supported commands are loaded from the command schema, so commands definition can change or be extended in the future without need of changing the application.

The tester (its relevant parts) may be used as a reference implementation of the communication layer with the CAN Activator for the Master Control System.

Screenshot of the testing client is on Fig. 5.1.

Testing process. Testing was performed inside the vehicle. Air Conditioning and Infotainment functions could be tested while the vehicle was stationary. To test the ACC and Speed Limiter functions, several test drives had to be performed, because these functions only work correctly when the vehicle is traveling at least at a certain speed.

Implemented commands. The commands implemented in this thesis are listed in Appendix B.

Almost all of the implemented Air Conditioning commands worked well. The only exception is the `air_volume` command, which sometimes does not take effect in the control element. The observed behavior is probably caused by a bug in an ECU, but it can be worked around by special handling of this command. The worked around solution has not yet been implemented.

Function for setting a driving mode has not been properly tested in the final version, but it had been tested earlier with success.

Regarding functions for ACC and Speed Limiter, they work well under most circumstances. There is still an opportunity to experiment with processing of these functions to avoid delays in corner cases.

Functions for setting the Audio source on the Infotainment has not yet been implemented.

Enhancements in the integration part are still possible. Some configuration is still hardcoded and should be moved to the configuration files. There could also be an (semi)automatic process for preparing the Terasic board and deploying all the files there instead. It has to be done manually in current version.

■ 5.1 Final Conclusion

All the assignment points have been met and the implementation is working. The solution (Fig. 5.2) was integrated into a Škoda Kodiaq vehicle and several test drives were performed with success. Works on this project will continue in the future.

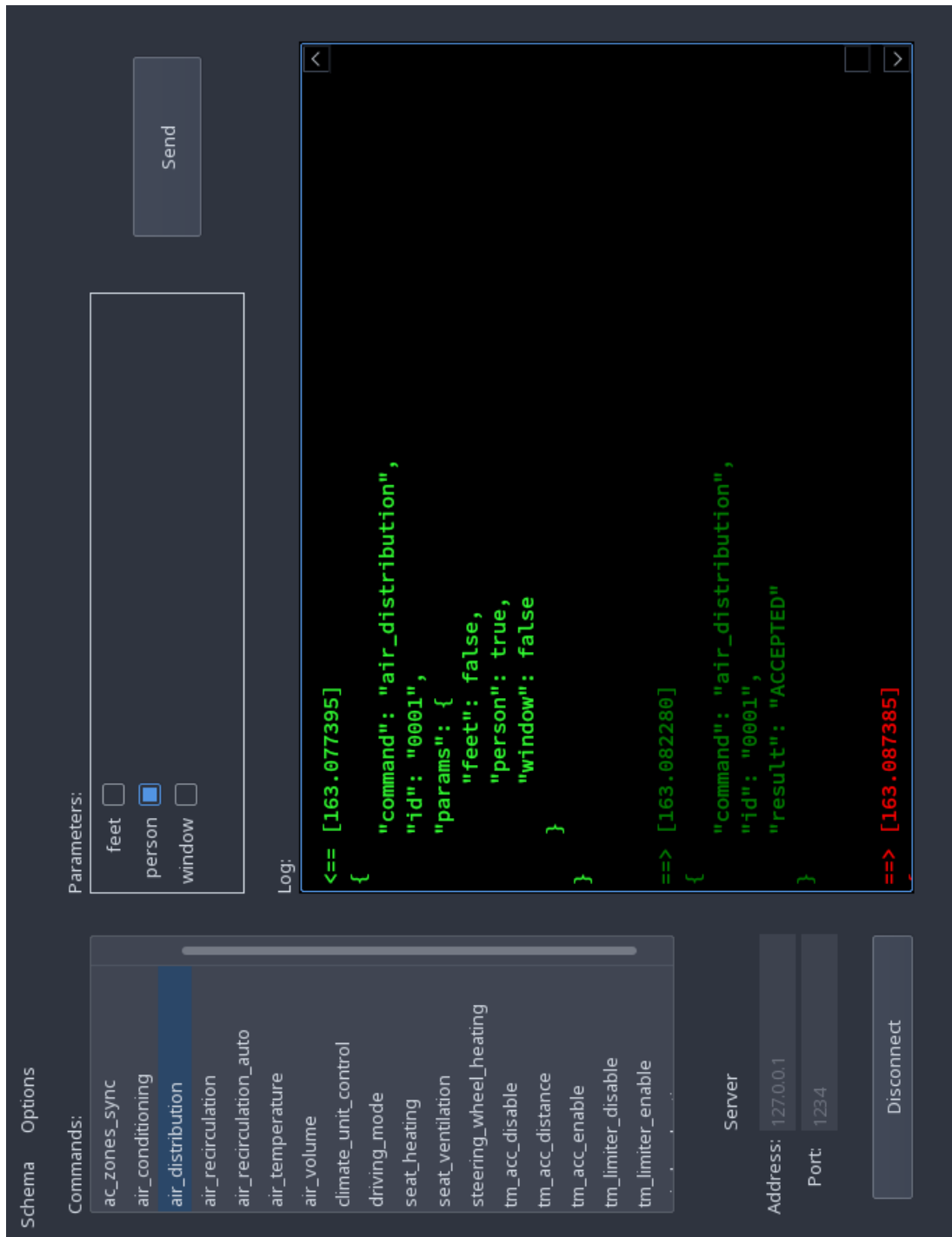


Figure 5.1: Tester client window



Figure 5.2: Control module (on the left) connected to one of the CAN Gateways (Gateway data interfaces are disconnected)




Bibliography

- [1] Oliver Hartkopp et al. *Readme file for the Controller Area Network Protocol Family (aka SocketCAN)*. URL: <https://www.kernel.org/doc/Documentation/networking/can.txt> (visited on 2020-08-11).
- [2] Ondrej Ille et al. *CTU CAN FD IP Core. CAN with Flexible Data-rate IP Core developed at Department of Measurement of FEE CTU*. git repository. URL: https://gitlab.fel.cvut.cz/canbus/ctucanfd_ip_core (visited on 2020-08-11).
- [3] *Altera Cyclone V SoC Board*. rocketboards.org. URL: <https://rocketboards.org/foswiki/Documentation/AlteraSoCDevelopmentBoard> (visited on 2020-08-11).
- [4] *Atlas-SoC Development Platform*. digital image. URL: <https://rocketboards.org/foswiki/pub/Documentation/AtlasSoCDevelopmentPlatform/Board-Top.jpg> (visited on 2020-08-11).
- [5] *Automobile CAN Bus Network Platform*. digital image. URL: <https://www.aidc.com.tw/Content/Image/caren-3-2.png> (visited on 2020-08-11).
- [6] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. 2017-12. DOI: 10.17487/RFC8259. URL: <https://rfc-editor.org/rfc/rfc8259.txt> (visited on 2020-08-11).
- [7] Chet Ramey Brian Fox. *BASH(1) General Commands Manual. bash - GNU Bourne-Again SHell*. Free Software Foundation, 2018-12-07.
- [8] *CAN data link layers in some detail*. URL: <https://www.can-cia.org/can-knowledge/can/can-data-link-layers/> (visited on 2020-08-11).
- [9] *CAN FD - The basic idea*. URL: <https://www.can-cia.org/can-knowledge/can/can-fd/> (visited on 2020-08-11).
- [10] *CAN lower- and higher-layer protocols*. URL: <https://www.can-cia.org/can-knowledge/> (visited on 2020-08-11).
- [11] *CAN physical layer*. URL: <https://www.can-cia.org/can-knowledge/can/systemdesign-can-physicallayer/> (visited on 2020-08-11).

- [12] *CAN with Flexible Data-Rate. Specification*. Version 1.0. Robert Bosch GmbH, 2012-04-17. URL: <https://can-newsletter.org/assets/files/ttmedia/raw/e5740b7b5781b8960f55efcc2b93edf8.pdf> (visited on 2020-08-11).
- [13] Stephen Cleary. *Detection of Half-Open (Dropped) Connections*. 2009-05-16. URL: <https://blog.stephencleary.com/2009/05/detection-of-half-open-dropped.html> (visited on 2020-08-11).
- [14] Intel Corp. *Avalon Interface Specifications*. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf (visited on 2020-08-11).
- [15] *Creating the U-boot script. Golden System Reference Design (GSRD) User Manuals FPGA Configuration*. rocketboards.org. URL: <https://rocketboards.org/foswiki/Documentation/S10SGMIIRDV181CreateUbootScript> (visited on 2020-08-11).
- [16] *DBC File Format Documentation*. Version 1.0. Vector Informatik GmbH, 2007-02-09. URL: http://read.pudn.com/downloads766/ebook/3041455/DBC_File_Format_Documentation.pdf (visited on 2020-08-11).
- [17] *DE0-Nano-SoC User Manual. (rev.D0 Board)*. Terasic, Inc., 2019-11. URL: https://www.terasic.com.tw/attachment/archive/941/DE0-Nano-SoC_User_manual_rev.D0.pdf (visited on 2020-08-11).
- [18] *Dokumentace modulu CAN FD Gateway*. Version 1.1. ČVUT FEL v Praze, Katedra měření, 2018.
- [19] M.J. Donahoo and K.L. Calvert. *TCP/IP Sockets in C: Practical Guide for Programmers*. TCP/IP Sockets in C Bundle. Elsevier Science, 2009. ISBN: 9780080923215. URL: https://books.google.cz/books?id=dmt%5C_mERzxV4C (visited on 2020-08-11).
- [20] *EPOLL(7) Linux Programmer's Manual. epoll - I/O event notification facility*. 2019-03-06.
- [21] Julia Evans. *Async IO on Linux: select, poll, and epoll*. URL: <https://jvns.ca/blog/2017/06/03/async-io-on-linux--select--poll--and-epoll/> (visited on 2020-08-11).
- [22] *FIFO(7) Linux Programmer's Manual. fifo - first-in first-out special file, named pipe*. 2017-11-26.
- [23] *GSRD - Boot Flow*. rocketboards.org. URL: <https://rocketboards.org/foswiki/Documentation/GSRDBootFlow> (visited on 2020-08-11).
- [24] *GSRD v13.1 - Compiling Golden Hardware Reference Design*. rocketboards.org. URL: <https://rocketboards.org/foswiki/Documentation/GSRD131CompileHardwareDesign> (visited on 2020-08-11).
- [25] *GSRD v13.1 - Generating and Compiling the Preloader*. rocketboards.org. URL: <https://rocketboards.org/foswiki/Documentation/GSRD131Preloader> (visited on 2020-08-11).

- [26] *GSRD v13.1 - Generating the Device Tree*. rocketboards.org. URL: <https://rocketboards.org/foswiki/Documentation/GSRD131DeviceTreeGenerator> (visited on 2020-08-11).
- [27] *GSRD v13.1 - SD Card*. rocketboards.org. URL: <https://rocketboards.org/foswiki/Documentation/GSRD131SdCard> (visited on 2020-08-11).
- [28] *History of CAN technology*. URL: <https://www.can-cia.org/can-knowledge/can/can-history/> (visited on 2020-08-11).
- [29] O. Ille. *CTU CAN FD IP CORE. System Architecture*. 2019.
- [30] Martin Jeřábek. “Open-source and Open-hardware CAN FD Protocol Support”. MA thesis. Czech Technical University in Prague. Department of Control Engineering, 2019-01-08.
- [31] Kent Lennartsson. *Comparing CAN FD with Classical CAN*. 2016-10. URL: <https://www.kvaser.com/wp-content/uploads/2016/10/comparing-can-fd-with-classical-can.pdf> (visited on 2020-08-11).
- [32] *MCP2517FD. External CAN FD Controller with SPI Interface*. Microchip, Inc., 2018. URL: <http://ww1.microchip.com/downloads/en/DeviceDoc/MCP2517FD-External-CAN-FD-Controller-with-SPI-Interface-20005688B.pdf> (visited on 2020-08-11).
- [33] *MQ_OVERVIEW(7) Linux Programmer’s Manual. mq_overview - overview of POSIX message queues*. 2020-06-09.
- [34] *NC(1) BSD General Commands Manual. nc — arbitrary TCP and UDP connections and listens*. 2018-12-27.
- [35] Jan Nejtěk. “Automotive Ethernet Analyzer”. bachelor thesis. Czech Technical University in Prague. Computing and Information Centre, 2019-06-12.
- [36] Carsten Pinkle. *The Why and How of Differential Signaling*. 2016-11-16. URL: <https://www.allaboutcircuits.com/technical-articles/the-why-and-how-of-differential-signaling/> (visited on 2020-08-11).
- [37] *PIPE(7) Linux Programmer’s Manual. pipe - overview of pipes and FIFOs*. 2017-09-15.
- [38] *POLL(2) Linux Programmer’s Manual. poll, ppoll - wait for some event on a file descriptor*. 2020-04-11.
- [39] *Preparing a Uboot image for Altera’s Cyclone V SoC FPGA*. Xillybus Ltd. URL: <http://xillybus.com/tutorials/u-boot-image-altera-soc> (visited on 2020-08-11).
- [40] *Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*. Standard ISO 11898-1:2003(E). Geneva, CH: International Organization for Standardization, 2003-01-12.

- [41] *SELECT(2) Linux Programmer's Manual*. *select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing*. 2020-04-11.
- [42] *Setting up a device tree entry on Altera's SoC FPGAs*. Xillybus Ltd. URL: <http://xillybus.com/tutorials/device-tree-altera-soc-cyclone> (visited on 2020-08-11).
- [43] *SHM_OVERVIEW(7) Linux Programmer's Manual*. *shm_overview - overview of POSIX shared memory*. 2016-12-12.
- [44] Randall R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. 2007-09. DOI: 10.17487/RFC4960. URL: <https://rfc-editor.org/rfc/rfc4960.txt> (visited on 2020-08-11).
- [45] *TCP(7) Linux Programmer's Manual*. 2020-06-09.
- [46] *TJA1051 High-speed CAN transceiver*. product datasheet. NXP, 2017-11-28. URL: <https://www.nxp.com/docs/en/data-sheet/TJA1051.pdf> (visited on 2020-08-11).
- [47] *Transmission Control Protocol*. RFC 793. 1981-09. DOI: 10.17487/RFC0793. URL: <https://rfc-editor.org/rfc/rfc793.txt> (visited on 2020-08-11).
- [48] *User Datagram Protocol*. RFC 768. 1980-08. DOI: 10.17487/RFC0768. URL: <https://rfc-editor.org/rfc/rfc768.txt> (visited on 2020-08-11).
- [49] Conal Watterson. *Controller Area Network (CAN) Implementation Guide*. Application Note. Rev. A. Analog Devices, 2017. URL: <https://www.analog.com/media/en/technical-documentation/application-notes/AN-1123.pdf> (visited on 2020-08-11).
- [50] Stewart Weiss. "Chapter 6 Event Driven Programming". In: *UNIX Lecture Notes* (). URL: http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes/chapter_06.pdf (visited on 2020-08-11).
- [51] *WRITE(2) Linux Programmer's Manual*. *write - write to a file descriptor*. 2019-10-10.



Appendix A

Abbreviations

CTU	Czech Technical University in Prague
FEE	Faculty of Electrical Engineering
CAN	Controller Area Network
FD	Flexible Data Rate
CRC	Cyclic Redundancy Check
EMI	Electromagnetic Interference
ACK	Acknowledgment
NACK	Negative Acknowledgment
GW	Gateway
AI	Artificial Intelligence
MCS	Master Control System
OS	Operating System
FIFO	First In First Out
IPC	Inter-Process Communication
AC	Air Conditioning
ACC	Adaptive Cruise Control
TCP	Transmission Control Protocol
IP	Internet Protocol
IP	Intellectual Property (in IP cores context)
IDE	Identifier Extension (a bit in CAN frame)
IDE	Integrated Development Environment

A. Abbreviations

RPC Remote Procedure Call

JSON Javascript Object Notation

SoC System on Chip

HPS Hard Processing System (Intel/Altera SoC)

FPGA Field-programmable Gate Array

I/O Input/Output

SD Secure Digital

Appendix B

List of implemented commands

Command	Description	Parameters
climate_unit_control	Enables or disables the climate unit compressor	enable
ac_control	Enables or disables cabin air conditioning	enable
ac_controls_rear	Locks or releases AC controls on the rear row	enable
air_recirculation	Enables or disables air recirculation in cabin	enable
air_recirculation_auto	Enables or disables automatic air recirculation in cabin	enable
ac_zones_sync	Enables or disables synchronization of air conditioning zones to the driver's zone	enable
window_heating	Enables or disables heating of the front or rear window	window, enable
air_temperature	Sets a temperature for one of three cabin zones	zone, value
air_volume	Sets a volume of air ventilation	value
air_distribution	Controls the distribution of air ventilation	window, person, feet
ac_auto	Switches air conditioning to automatic mode. In this mode the air volume is regulated automatically at the rate set by "ac_auto_style" command, to reach the setpoint temperature	
ac_auto_style	Sets a style (rate) of automatic mode of AC	style
seat_heating	Sets a heating degree for front-row seats	seat, value
seat_ventilation	Sets a ventilation degree for front-row seats	seat, value

B. List of implemented commands

steering_wheel_heating	Controls a level of steering wheel heating	level, enable
driving_mode	Selects one of six driving modes	mode
tm_acc_enable	Activates Automatic Cruise Control to maintain the vehicle at target speed	speed
tm_acc_distance	Sets the preceding vehicle distance for Automatic Cruise Control	distance
tm_acc_disable	Deactivates Automatic Cruise Control	
tm_limiter_enable	Activates Speed Limiter to prevent the vehicle from speeding above limit	speed
tm_limiter_disable	Deactivates Limiter function	