Bachelor's thesis

Degree Programme in Information Technology

Internet Technology

2015

Daria Shevchenko

DEFINITION AND IMPLEMENTATION OF AN ARCHITECTURAL CONCEPT FOR CONFIGURING A CAN NETWORK

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR´S THESIS | ABSTRACT
TURKU UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Information Technology | Internet Technology

2015 | 45 pages

Instructors: Andreas Kraus, Stephan Huebner, Tero Virtanen

Daria Shevchenko

DEFINITION AND IMPLEMENTATION OF THE ARCHITECTURAL CONCEPT FOR CONFIGURING A CAN NETWORK

This Bachelor’s thesis is a part of the microLabCar test system project developed by Bosch. The microLabCar test systems are used by different customers, as well as within the company for testing Engine Control Units. The system contains two parts: the hardware device and the operating software. The project has been operational for a few years and with every new step in hardware development, new features on the software level are required. This thesis is related to the Controller Area Network (CAN), an automotive bus standard, that provides communication between microcontrollers and devices.

The purpose of this thesis was to implement the loading of a Field Bus Exchange Format (FIBEX) file for configuring a CAN network. The task included developing a concept of a FIBEX loader for CAN and integrating it into the test system software. Additional to the implementation of a FIBEX loader, some architectural and GUI extension needed to be implemented. A FIBEX interface in a test system software should be reusable for other automotive protocols such as FlexRay and LIN (Local Interconnect Network). The CAN protocol should be able to work with CANDBC standard, another standard used for configuring CAN networks, as well as with FIBEX.

Since this thesis project is a part of the production process, the majority of the coding cannot be published. Only the publicly available structures can be presented in this work.

KEYWORDS:

FIBEX, CAN, CAN network, OOP, interface, protocol, FlexRay, CANDBC

CONTENTS

APPENDICES

PICTURES

FIGURES

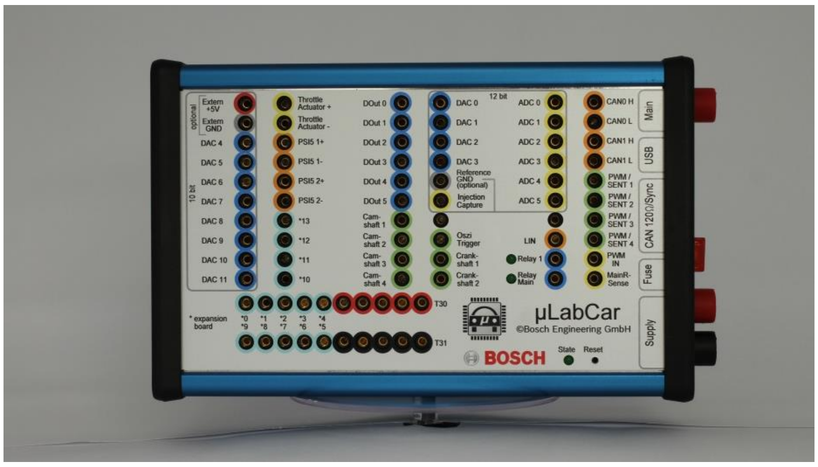
LIST OF ABBREVIATIONS (OR) SYMBOLS

CAN	A Controller Area Network
TDMA	A Time Division Multiple Access
FIBEX	The Field Bus Exchange Format
SOF	Start of Frame
RTR	Remote Transmission Request
DLC	Data Length Code
CRC	Cyclic Redundant Check
ACK	Acknowledge
EOF	End of Frame
IFS	Intermission Frame Space
CANDBC	A file standard used for a CAN network
PDU	Protocol Data Unit
GUI	Graphical User Interface
DLL	Dynamic-link Library
LIN	Local Interconnect Network
ECU	Electronic Control Unit

1 INTRODUCTION

The modern automotive industry is a rapidly growing and fast developing field. The size of the production and constantly growing demand require new approaches and methods to optimize the manufacturing process. The electronic systems in automobiles are tightly connected to bus protocols or so-called automobile bus protocols. Examples of the latter ones are the Control Area Network (CAN) and FlexRay bus protocols. The protocols are supposed to ensure safe and reliable data transmission, otherwise a car will not respond as it is intended to. In order to ensure correct functionality, multiple tests are performed. For this purpose Bosch uses LabCar produced by ETAS company. This device simulates technical environment of an Engine Control Unit. It is a complex system and due to its costs only a few of them are usually available for the employees. As a solution to this problem, a microLC test system was developed by Bosch Engineering GmbH. It is a more compact and cheaper test system used for performing tests of certain sensors of an Engine Control Unit.

The device consists of two parts – the hardware and the operating software for Windows PC. The MicroLC test system allows for performing tests right at a work place and is enabled to work with other devices. The connection between a microLC and a PC is a USB connection. An example of microLC device can be seen in Picture 1.



Picture 1. MicroLC3 device

2 BACKGROUND

In order to obtain a clear idea about the tasks that were implemented in this thesis work, a reader needs to understand the core terms that were used. The work was based on the knowledge of FlexRay and CAN protocols as well as the FIBEX standard, all used for configuring FlexRay and CAN networks. In the chapters 2.1 -2.5 the reader has the opportunity to become familiar with these terms in order to understand the scope of work described in the next chapters.

2.1 FlexRay bus

The FlexRay communications bus (National Instruments, 2009, p.1) is a deterministic, fault-tolerant, and high-speed bus system developed in collaboration with automobile manufacturers and leading suppliers. The FlexRay bus was designed as low cost bus system that ensures high performance even in a rough environment. The FlexRay uses unshielded twisted pair cabling to connect nodes together. It supports single and dual-channel configurations consisting of one or two pair of wires. Differential signaling on each pair of wires is an effective substitute for expensive shielding that decreases the effect of external noise on the network. Most FlexRay nodes also have power and ground wires to power transceivers and microprocessors.

Dual-channel configurations offer enhanced fault-tolerance and/or increased bandwidth, though first-generation FlexRay networks use only one channel to save costs.

FlexRay buses have a resistor connected between the pair of signal wires that ensure termination at the ends. A multi-drop bus requires termination only at the end nodes. Not enough or too much termination can break a FlexRay network. Therefore, the end nodes are terminated to match typical FlexRay cabling impedance between 80 and 110 ohms. To avoid frustration when connecting a FlexRay node to a test setup, modern PC-based FlexRay interfaces have on-board termination resistors.

FlexRay supports simple multi-drop passive connections as well as active star connections for more complex networks. The right selection of network topology helps to optimize costs, performance, and reliability for a given design. There are three types of network topology that FlexRay supports (National Instruments, 2009, p.2):

- The multi-drop bus is a simple topology that features a single network cable run that connects multiple ECUs together. The network has termination resistors installed on its ends to solve problems with signal reflections. Correct termination and network layout are extremely important for FlexRay networks, since they operate at high frequencies which might cause signal integrity problems
- The star network consists of individual links that connect to a central node. This type of network topology allows running FlexRay networks over long distances or segmenting it in order to ensure network reliability in case a portion of the network fails. Branches of the FlexRay network are independent. Star network topology also gives a positive environmental effect since it reduces the amount of exposed wire decreasing environmental noise that it causes
- The hybrid network combines the previous two types of network topology. Future FlexRay networks are likely to consist of hybrid networks, using the advantages of both

FlexRay protocol

FlexRay is an automotive network communication protocol developed by the FlexRay Consortium. It is a unique time-triggered protocol that provides options for both deterministic data, such that arrives in a predictable time frame and dynamic event-driven data. FlexRay succeeds in combining static and dynamic frames due to the preset communication cycle that pre-defines space for static and dynamic data. This space is configured along with the network by a network designer. To ensure communication between nodes in a FlexRay network designers should know how all pieces of network are configured.

As with any multi-drop, multiple nodes trying to write data cause data corruption and contention on the bus. In case of FlexRay a Time Division Multiple Access or TDMA scheme is the one handling multiple nodes. Every node in FlexRay implements a TDMA scheme that guarantees the determinism of data delivery in the network.

The FlexRay standard can be adapted easily to different types of network, giving a great amount of flexibility to network designers. In order to support maintaining network configurations between nodes, a standard format to store and transfer the parameters is used. The Field Bus Exchange Format, or FIBEX file – an XML-based standardized format for configuring the software of automotive networks (National Instruments, 2009, p.2).

Communication cycle

The FlexRay communication cycle is a basic element of the TDMA scheme. The duration of a communication cycle takes a fixed time which is set when a network is designed. A communication cycle consists of a dynamic segment, a static segment, and two protocol segments called symbol window and network idle time.

Static segment

A static segment is pre-allocated into slices which are reserved for data that arrives at a fixed period of time. The duration of slots and their number are determined by a FlexRay configuration. These settings must match in all controllers in the network. Each slot is exclusively owned by one FlexRay communication controller for transmission of a frame on a certain channel. On other channels in the same slot, either the same or another controller can transmit a frame. The transmitting controllers in one slot are identified using configuration parameters of the FlexRay controller. The piece of information about the transmitter of a frame and its content is local and and it is not available for the receiving controllers. The message ID in the payload section of the frame is what is used to uniquely identify the content of a frame. A controller is able to search for a specific data using a filter mechanism. A static segment guarantees strong communication latency since it exactly known when a frame is being transmitted.

Dynamic segment

A dynamic segment is subdivided into so-called minislots which are substantially shorter than a static slot. A minislot has just enough space to accommodate a potential start time of a frame. Each slot is exclusively owned by one FlexRay communication controller for transmission of a frame on a certain channel. All controllers in the network have information about a current minislot. So when a controller wants to transmit in a minislot, this action is detected by all other controllers and the counting of minislots is interrupted. Thus, a minislot becomes a real slot that has enough space for a frame transmission. The expansion of a minislot reduces the total number of minislots and minislots counting is renewed only after the end of the frame transmission. In case when there is no data to transmit by the owner of a minislot, the latter is not expanded and the slot counting is not interrupted. So a minislot does not use any additional bandwidth, saving the latter for other lower-priority minislots. A scheme used by a dynamic media access control ensures optimal use of reserved bandwidth. The minimum duration of a minislot mainly depends on a network parameter (delay) and by the maximum deviation of the clock frequency in the controllers.

Frame format

All the communication is carried out in the form of the frames. The message, consisting of bytes, is packed as it is shown in Figure 1.

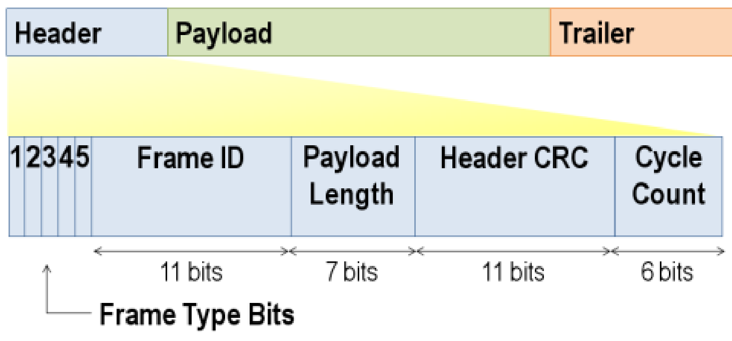


Figure 1. Flexray frame format (National Intruments, 2009, p.5)

Each of FlexRay messages consists of three parts: header, payload and trailer. The header has 40 bits, 11 out of which belong to the identifier (ID). Any ID can identify a message except ID=0x00, which is used to identify invalid messages.

As can be seen from Figure 1, the ID is preceded by four indicator bits, which in turn are preceded by a reserved bit. The payload preamble indicator shows whether a network management vector is being transmitted in the payload of a static message or whether a message identifier is being transmitted in the payload of a dynamic message.

The null frame indicator is served to indicate whether the payload is regular or invalid.

The sync frame indicator indicates whether the messages transmitted in the static segment are being used as sync frames in the context of synchronization.

The startup frame indicator indicates whether the message in the static segment is a part of a startup frame in the context of startup.

The payload length shows the size of the payload in words and consists of 7 bits. The identifier is protected by the CRC method, an error-detecting code used to detect accidental changes to raw data.
The header CRC sequence consists of 11 bits and is calculated basing on the identifier, payload length, sync frame indicator, startup frame indicator and a generator polynomial defined by the FlexRay specification.

The cycle counter consists of 6 bits and shows the number of the cycle in which the message is sent.

The maximum payload is 254 user bytes. The same value of the payload length is shown for all messages transmitted in the static segment. This default value is defined by a system designer during the configuration phase. The payload length for dynamic messages may vary.

In the message transmitted in the static segment, the first 12 bytes are used to transmit the network management vector. The payload preamble indicator should be set in the header.

If the payload preamble indicator is set for a message in the dynamic segment, then the payload starts with a network management vector. The first 2 bytes of a dynamic message are the message identifier. The message identifier helps the system designer to specify a payload more precisely.

The CRC method is used to protect the payload. In the framework of the CRC method, a CRC sequence is calculated basing on a header, a payload and a generator polynomial defined by the FlexRay specification. The CRC sequence is added to the header and payload as the trailer (Vector, 2010).

Symbol window

A symbol window is a fixed-length time slot, in which special symbols can be transmitted. These symbols are used for network management purposes.

Network idle time

A network idle time allows no traffic to be scheduled on the communicational channel. It is a protocol-specific time window. A network idle time is used by communication controllers to execute the clock synchronization algorithm. The clock synchronization causes the offset correction that requires some controllers to edit their view of the time forward, and others to edit their view of the time backward. During this time, no consistent operations of the media access control can be guaranteed. Since the duration of the network idle state should be subtracted from the network bandwidth, it is kept as short as possible. The idle time is a global parameter and it has to be consistent within the network. Depending on a structure of a communication cycle, there are basically three reasonable configurations recognized:

- Static configuration, which contains only static slot for transmission. At least two slots owned by different controllers should be in a static segment in order to enable clock synchronization. In case of a fault-tolerant clock synchronization, there should be at least four static slots in a static segment.
- Mixed configuration includes both a static segment and a dynamic segment. The ratio between a static bandwidth and a dynamic bandwidth can vary.
- Dynamic configuration has all of its bandwidth only for dynamic communication. However, this type of configuration still requires a "degraded static segment" containing two static slots (National Instruments, 2009, p.5).

2.2 CAN bus

CAN bus

The CAN standard defines the network that links all the nodes connected to a bus and enables them to talk with one another. The central node is not required and new nodes can be added at any time (Corrigan, 2008, p. 7)

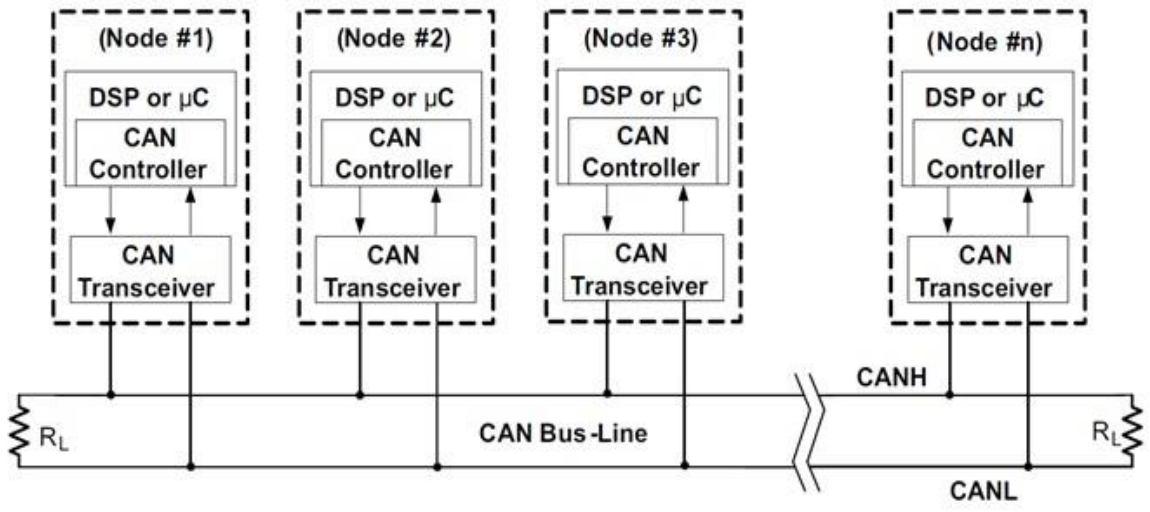


Figure 2. CAN network (Corrigan, 2008, p.7)

There is a CAN transceiver between the bus and each CAN controller. The CAN bus consists of two differential signal wires CANH and CANL. As is shown in Figure 2, the CAN transceiver is used to convert the Tx and Rx lines sent by the controller to differential signals CANH and CANL. The two signal lines of the bus, CANH and CANL, in the recessive state, are passively biased to ≈ 2,5 V. The dominant state on the bus takes CANH ≈ 1 V higher to ≈ 3,5 V, and takes CANL ≈ 1 V lower to ≈ 1,5 V, creating a typical 2-V differential signal as displayed in Figure 3.

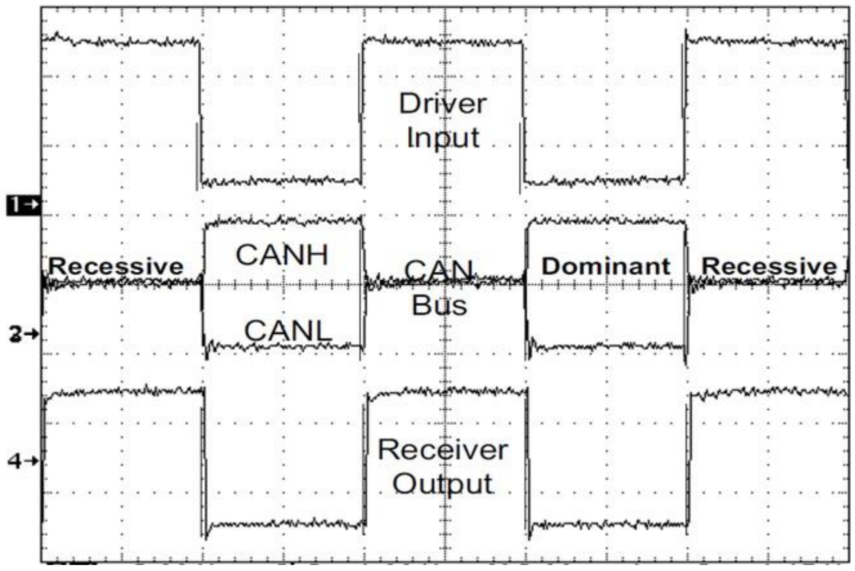


Figure 3. The CANH and CANL lines during a CAN production (Corrigan, 2008, p.8)

The differential signaling ensures noise immunity and fault tolerance of CAN. Balanced differential signaling allows high signaling rates over twisted-pair cable and reduces noise coupling. Balanced signaling assumes that the current in each signal line is the equal but opposite in directions, resulting in a field-cancelling effect.

The specification of the bus is defined by the High-Speed ISO 11898 Standard (Corrigan, 2008, p.8), according to which a maximum signaling rate is 1 Mbps with a bus length of 40 m with a maximum of 30 nodes. The cable is a shielded or unshielded twisted-pair cable with 120 Ohm characteristic impedance. The nominal specific propagation delay of the two-wire bus line is specified at 5 ns/m. In order to be physically compatible, all nodes in the network must use the same or a similar bit-timing (Corrigan, 2008, p.8).

CAN protocol

The Controller Area Network (CIA, 2013, p.4), developed by Bosch in the early 1980s, is an international standard meant for fast serial data exchange between electronic controllers in motor vehicles. In 2012, the CAN protocol was enhanced by Robert Bosch GmbH and receive the name CAN FD (flexible data-rate) (CIA, 2013, p.5).

The CAN protocol uses a “broadcast communication mechanism” which is based on a message-oriented transmission protocol. It recognizes only message content. Each message has a unique identifier which defines the priority of a message and its content. CAN provides a high degree of system and configuration flexibility due to using a content-oriented addressing scheme. The existing CAN network easily joins new stations in case they are purely receivers. Such configuration of a CAN network allows the reception of multiple data and synchronization of distributed processes. Simple servicing and upgrading of the network is possible, since data transmission does not depend on the availability of specific types of stations.

Real-time data transmission

In real-time data processing, the priority of a message defines how fast it will be processed. The priority is specified by the identifier of each message. The priorities are set during the system design in binary format and cannot be changed dynamically. The lowest binary number corresponds to the highest priority. Bus access conflicts are handled by the “wired-AND” mechanism, according to which the dominant state overwrites the recessive state. All those stations with recessive transmission and dominant observation fail to acquire a bus access. They become receivers of the message with the highest priority and renew their attempt to acquire bus access only when the bus is available again.

Such a mechanism guarantees low individual latency times in real-time systems and it proves to be especially advantageous in overload situations.

CAN frame format

The CAN protocol supports two message frame formats that differ in the length of identifier. The “CAN base frame” supports a length of 11 bits, and the “CAN extended frame” supports a length of 29 bits. The 11-bit message always has priority over the 29-bit message. Among the other benefits of 11-bit message are the shorter bus latency time. It also uses less bandwidth comparing to the 29-bit message and has higher error detection performance.

As can be seen from Figure 4, the CAN frame has the following structure:

- The “Start Of Frame (SOF)” is a start bit that begins a frame message
- The “Arbitration field” includes the identifier and the “Remote Transmission Request (RTR)” bit which is dominant for data frames. In case of CAN extended frame, the identifier contains two recessive bits: SRR and IDE
- The “Data Length Code (DLC)” indicates the number of following data bytes in the “Data field”. If the message is used as a remote frame, the DLC field contains the number of requested data bytes
- The “Cyclic Redundant Check (CRC)” guarantees the integrity of the frame
- The “Acknowledge (ACK) field” compromises the ACK slot and the ACK delimiter. The ACK bit is sent as a recessive bit and is overwritten as a dominant bit by those receivers which have at this time received data correctly.
- The “End of Frame (EOF)” indicates the end of the message
- The “Intermission Frame Space (IFS)” is the minimum number of bits separating consecutive messages.

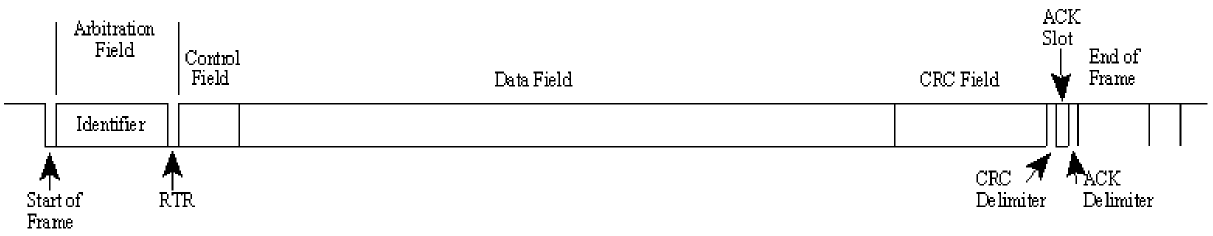


Figure 4. CAN base format (Kvazer, 2014)

Error handling

The CAN protocol signals errors immediately once they occur instead of using acknowledgement messages. The CAN protocol implements three mechanisms for error detection at the message level:

- Cyclic Redundancy Check (CRC): The CRC ensures the frame security by adding a frame check sequence (FCS) at the transmission end. The receiver compares re-computed FCS with the received FCS. When they do not match, it means that a continuity counter error (a break in the sequence numbers) has happened.
- ACK errors: Receivers acknowledge the received frames. In case the transmitter does not receive a confirmation, an ACK error is detected.
- Frame check: It is used for frame structure verification. The bit fields are compared with the fixed format and the frame size. Errors detected by frame check are “format errors”

The CAN protocol implements two mechanisms for error detection at the bit level:

- Monitoring: Each station that transmits is also able to observe the bus level and thus detects differences between the bit sent and the bit received
- Bit stuffing: After every five consecutive equal bits, the transmitter inserts a stuff bit into the bit stream. This way the synchronization edges are generated. The stuff bit is removed by the receiver.

Error detection with a help of one of the mechanisms, as described earlier, leads to the current transmission abortion by sending an “error frame” which prevents other stations from accepting the data. After the first transmission has failed, the sender automatically re-attempts transmission (CIA, 2013, p.13).

2.3 ASAM MCD2-NET (FIBEX) standard

The ASAM MCD2-NET (FIBEX) standard is a uniform XML-based data format that provides a description for setting the software of automotive networks. FIBEX stands for “Field Bus Exchange Format”. FIBEX consists of two interfaces, one of which is generic and the other one contains technology-specific extensions for FlexRay, CAN and other protocols. The standard accepts the definitions of network topologies that consist of ECUs with network ports and gateways. Technology-specific properties are given for each network port.

FIBEX is widely used in the automotive industry for configuration, design, monitoring and simulation of communication in networks (ASAM, 2013).

The described format should comply with the following requirements:

- Design support: The format should support all steps of the design process, meaning that incomplete topologies or intermediate stages should be allowed.
- Covered areas: They should be able to model entities from different areas:
    - Functional network, consisting of functions, signals, services and data types
    - System topology that describes the logical layout of the system
    - Communication properties, describing frames, their timing settings and etc.
- Scalability: The format should be scalable, support both simple data structures and complex data buses or multi-bus networks
- Support for extensibility: The format should be extensible, meaning it should be possible to adjust it easily to manufacturer-specific requirements
- Simplicity: The format should support the description of systems without a functional level

- Communication architectures: the format should support the most common automotive data buses like LIN, FlexRay, CAN, TTCAN, MOST, Ethernet

Below is listed a description of the entities defined in FIBEX:

- Entities in system topology
  - Cluster: a cluster describes a group of ECUs, connected in an arbitrary topology. Nodes (ECUs) in a cluster use the same communication protocol.
  - Channel: a channel connects ECUs to a cluster through communication medium. To be a part of a cluster, an ECU has to have at least one controller connected to at least one channel of the cluster.
  - Connector: The connector is meant to describe the bus-interfaces of the ECUs and to define the sending/receiving behavior
  - Controller: The controller is a piece of hardware that ensures the ability of nodes to send frames to and receive from the communication medium
  - Coupling-elements: These are network devices that ensure data transmission without generating or terminating communication. Coupling-ports are used to establish the connection between the network entities and the communication medium
  - ECU: The Electronic Control Unit has one or more host processors that execute part of a distributed application. An ECU can be a gateway if it is connected to two or more different clusters by two or more of its communication controllers
  - Gateway: An ECU that is connected to two or more different clusters by two or more of its communication controllers. It ensures a frame mapping between those clusters
- Entities describing functional level
  - Function: a function is a part of a distributed application running in an ECU

  - Input-port, output-port: Exactly one function is being assigned to an input port and an output port. The signal entity ensures the link between the output port of one function and an input port of another function.
- Entities describing signals
  - Coding: A coding provides the conversion between CODED-TYPE and PHYSICAL-TYPE
  - Signal: It is a contiguous sequence of bits of a defined length. As to its “functional” purpose, it is an input or an output parameter of a function.
- Entities describing the communication level
  - PDU: A Protocol Data Unit consists of one or more SIGNAL-INSTANCES or MULTIPLEXES
  - PDU-INSTANCE: It describes the BIT-POSITION and the UPDATE_BIT_POSITION of a PDU within a FRAME
  - INCLUDED-PDU: It is a collection of INCLUDED-SIGNALS of an ECU-PORT or a PDU-REQ
  - MULTIPLEXER: It is an optional part of PDU that contains information about alternative contents of PDU
  - SWITCHED-PDU-INSTANCE: It is a PDU-INSTANCE in the DYNAMIC-PART of a MULTIPLEXER
  - STATIC-PDU-INSTANCE: A STATIC-PDU-INSTANCE is a PDU-INSTANCE in the STATIC-PART of a MULTIPLEXER
  - STATIC-PART: It is a static part of a MULTIPLEXER
  - DYNAMIC-PART: It is a dynamic part of a MULTIPLEXER
  - PDU-TRIGERRING: It describes a way the triggering of a PDU is done
  - FRAME: A frame is a datagram with a payload section of a certain length in bytes. A frame contains an arbitrary number of non-overlapping signals and /or multiplexers
  - FRAME-TRIGGERING: It describes the way the triggering of a frame is done

  - TIMING: It defines timing behavior of a FRAME or a PDU on a CHANNEL
  - SIGNAL-INSTANCE: It defines the BIT-POSITION and the UPDATE-BIT-POSITION of a SIGNAL within a PDU
  - INCLUDED-SIGNAL: It is a list of SIGNAL_INSTANCES for use in an INCLUDED-PDU
  - ORDERED-SIGNAL: An ordered signal
  - ECU-PORT: An INPUT-PORT or OUTUT-PORT in the communication oriented view
  - SWITCH: A SWITCH value specifies a data region in the same frame. It is a bit field of defined length
  - ABSOLUTELY-SCHEDULED-TIMING: It specifies a sending behavior with the guaranteed time for the frames transmission
  - RELATIVELY-SCHEDULED-TIMING: It specifies a sending behavior with a predefined order
  - EVENT-CONTROLLED-TIMING: The event has to be registered first, so that a PDU could be sent
- Entities describing requirements
  - SIGNAL_GROUP: It contains signals that must always be kept together in a common frame
  - REQUIREMENT: Functional or port requirements
  - FUNCTION_REQ: It specifies CYCLE-PERIOD, CYCLE-OFFSET, DEADLINE and WCET of a function and references all functions for which this request is valid
  - PORT-REQ: It specifies MAX-AGE, GENERATION-TYPE, CYCLE-PERIOD, CYCLE-OFFSET of a SIGNAL and references all ports for which this request is valid
  - PDU-REQ: It specifies INPUT-PORT, OUTPUT-PORT and references ECUs where the referenced PORTs are located
- Entities describing gateways:
  - GATEWAY: It is the entity that transfers signals and/or frames between channels

  - CONNECTOR-MAPPING: It provides the information about the channel connected to the ECU that is served by the gateway
  - FRAME-MAPPING: It describes the frames that are transferred between the CHANNELs
  - PDU-MAPPING: It defines the PDUs that are transferred between the FRAMEs
  - SIGNAL-MAPPING: It describes the signal transferred between the PDUs
  - OPTIMISED-MAPPING: It describes the transfer functions between frames on byte and bit level
  - GW-DIAGNOSIS-DATA: It describes the routing of diagnostic data
- Entities describing higher protocols
  - PROTOCOL: Specification of a clusters protocol
  - TP-CONFIG: Extension point for the transport layer schema
  - NM-CONFIG: Configuration of the network management protocol
- Entities describing services
  - SERVICE-INTERFACE: An IDL (Interface Description Language) describes the interface of software components. IDL allows communication between software components that do not share a common language
  - PACKAGE: It is used to build hierarchical structures
  - DATATYPE: Datatypes are used by fields, event or method arguments in services (ASAM, 2013).

2.4 FIBEX extensions for CAN

In this section an introduction to special FIBEX extensions used for configuring CAN network is given. The base is the original FIBEX standard described earlier.

2.4.1 Controller-Type

The CAN specific extension for the CONTROLLER-TYPE. The CAN specific CONTROLLER attributes are required for the CAN stack configuration in an ECU linked to the CAN cluster. The CAN-specific CONTROLLER attributes can be specified in two different ways:

- Exact values are provided by the ECU developer (CAN-BIT-TIMING)
- Requirements are provided that are checked by the ECU developer (CAN-BIT-TIMING-REQUIREMENTS)
  CAN-BIT-TIMING

In the CAN-BIT-TIMING specific extension, the following parameters can be configured:

- TIME-SEG0: It describes the duration of time 0 in time quanta (TQ). The time segment 0 includes the propagation segment and the synchronization jump width.
- TIME-SEG1: It describes the duration of the time segment 1 in time quanta (TQ).The time segment 1 and the duration of the phase-buffer-segment 2 are identical.
- SYNC-JUMP-WIDTH: It describes the possible variation range of the sample point in time quanta (TQ). The SYNC-JUMP-WIDTH should be less than the minimum of the duration of the phase-buffer-segment 1 and the duration of the phase-buffer-segment 2
- NUMBER-OF-SAMPLES: There can be 1 or 3 sample points in a CAN controller

CAN-BIT-TIMING-REQUIREMENTS

The specification of ranges for the CAN Bit Timing configuration parameters are allowed after careful consideration by the ECU developer. The required ranges are the following:

- MAX-NUMBER-OF-TIME-QUANTA-PER-BIT: It shows the maximum number of time quanta in the bit time
- MAX-SAMPLE-POINT: It shows the maximum value of the sample point in a percentage of the total bit time
- MAX-SYNC-JUMP-WIDTH: It shows the maximum Synchronization Jump Width value in a percentage of the total bit time
- MIN-NUMBER-OF-TIME-QUANTA-PER-BIT: It describes the minimum number of time quanta in the bit time
- MIN-SAMPLE-POINT: It shows the minimum value of the sample point in a percentage of the total bit time
- MIN-SYNC-JUMP-WIDTH: It shows the minimal Synchronization Jump Width value in a percentage of the total bit time

2.4.2 Identifier-Value-Type

EXTENDED-ADDRESSING

With the CAN protocol, two frame formats are possible: the standard frame with 11-bit identifiers is defined in the CAN specification 2.0 A, the extended frame with 29-bit identifiers and is defined in the CAN specification 2.0 B.

2.5 FIBEX EXTENSIONS FOR FLEXRAY

This section introduces special FIBEX extensions used for configuring a CAN network. The base is an original FIBEX standard described earlier.

2.5.1 Controller-Type

The FlexRay common extension for the CONTROLLER-TYPE includes node specific low-level parameters from the FlexRay Spec 2.1A

2.5.2 Cluster-Type

The FlexRay common extension for the CONTROLLER-TYPE includes cluster specific low-level parameters from the FlexRay Spec 2.1A

KEY-SLOT-USAGE

The FlexRay specification allows every device to send a sync, a sync frame and start-up frame or neither. When sending a sync or a start-up frame the slot ID of the KEY-SLOT should be given.

CHANNEL-TYPE

In the default configuration, there are two channels with the names A and B that can be used. Therefore, the extension of the common CHANNEL-TYPE includes the element FLEXRAY-CHANNEL-NAME.

CONNECTOR-TYPE

The device that has an ability to wake up the network needs to know which channel the device uses in order to send a wake pattern. Since every CONNECTOR belongs to exactly one ECU and references exactly one CHANNEL, the FlexRay extension of the common CONNECTOR-TYPE includes a Boolean WAKE-UP_CHANNEL flag (ASAM, 2013).

3 IMPLEMENTATION OF A FIBEX LOADER FOR A CAN

3.1 Defining use cases for FIBEX loader for CAN

The FIBEX format contains all information about network topology, configuration parameters, schedules, frames, and signals. That gives maximum information and functionality to a user to configure a CAN network. In Figure 5, a use case diagram describes the interaction between a user and a network.

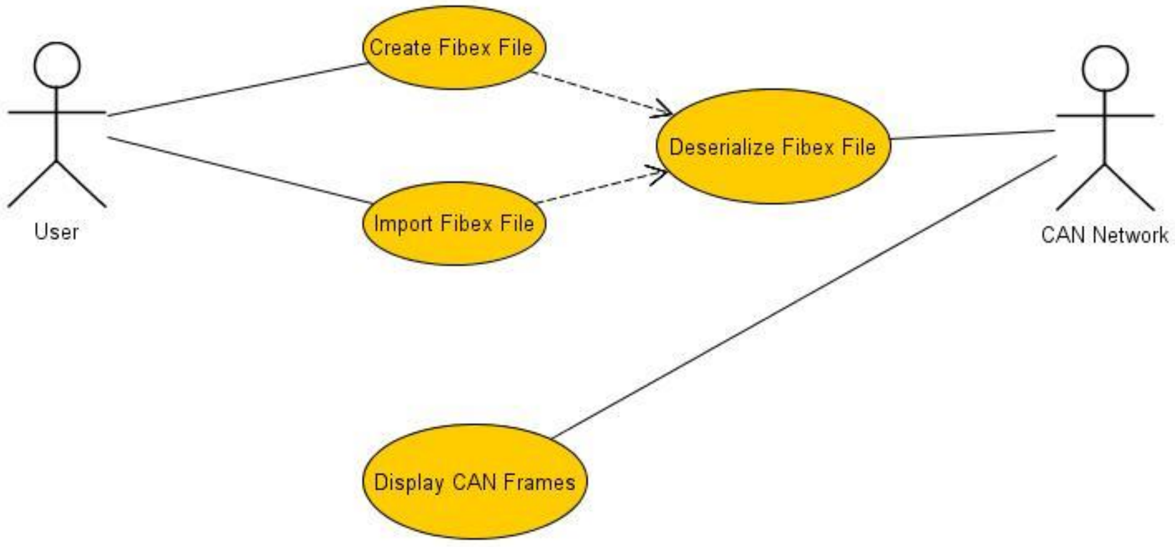


Figure 5. Use case diagram for FIBEX Loader implementation for CAN

3.2 Defining an architecture for FIBEX Loader for CAN

Since a FIBEX file has XML-based format, it can be easily translated into a class-object model. The implemented version of a FIBEX loader classes' structure for Flexray can be used as a base for a FIBEX loader for a CAN. Below an overview of a basic FIBEX configuration which is implemented by all protocols is given.

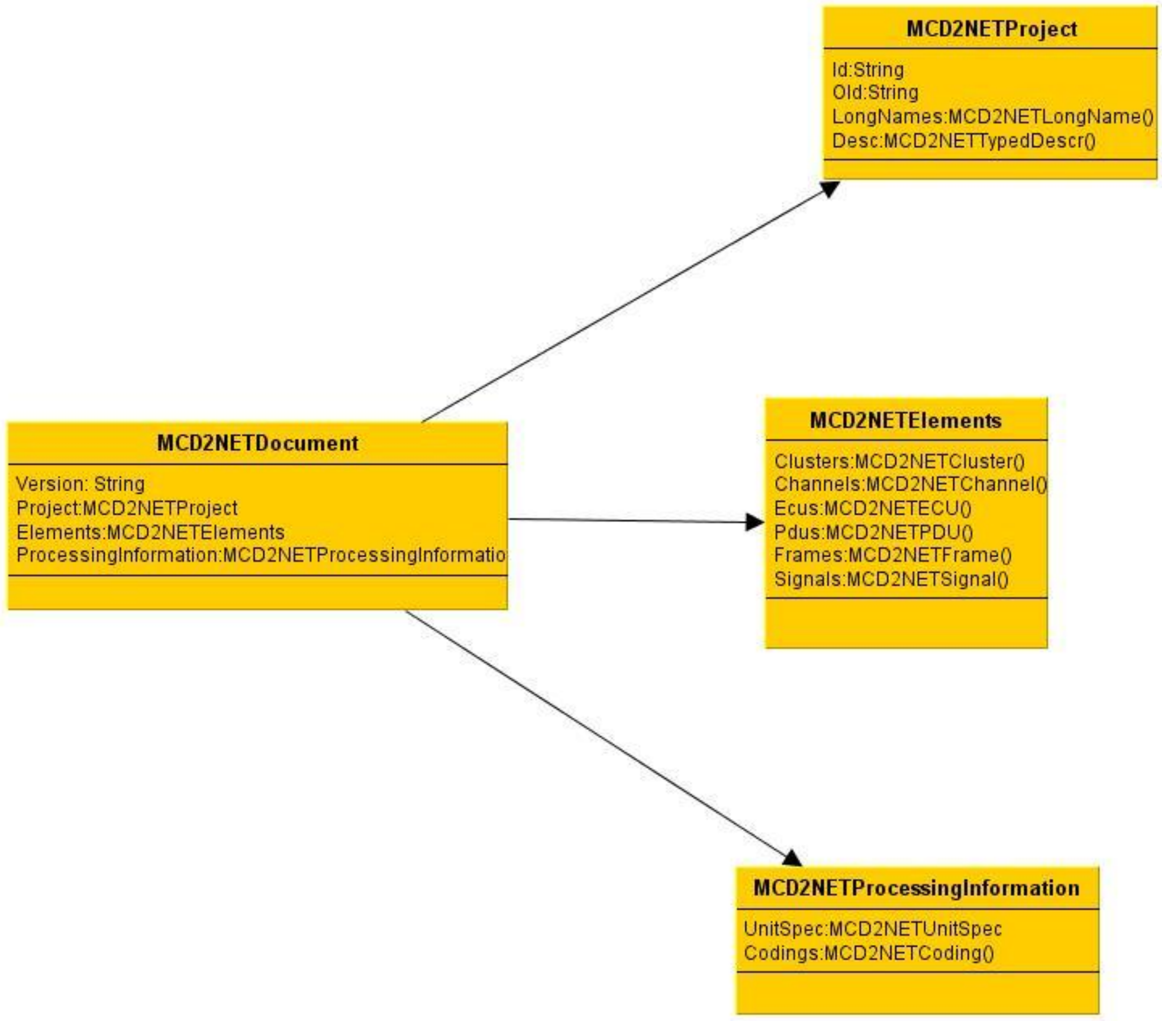


Figure 6. FIBEX loader classes' structure.

The MCD2NETDocument described in Figure 6 is the main class that includes all information from a FIBEX file. The object of this class is passed later to processing classes. The detailed description of the class MCD2NETElements is given below. In Figure 7 the key entities, that are important for understanding an automotive protocol, are described.

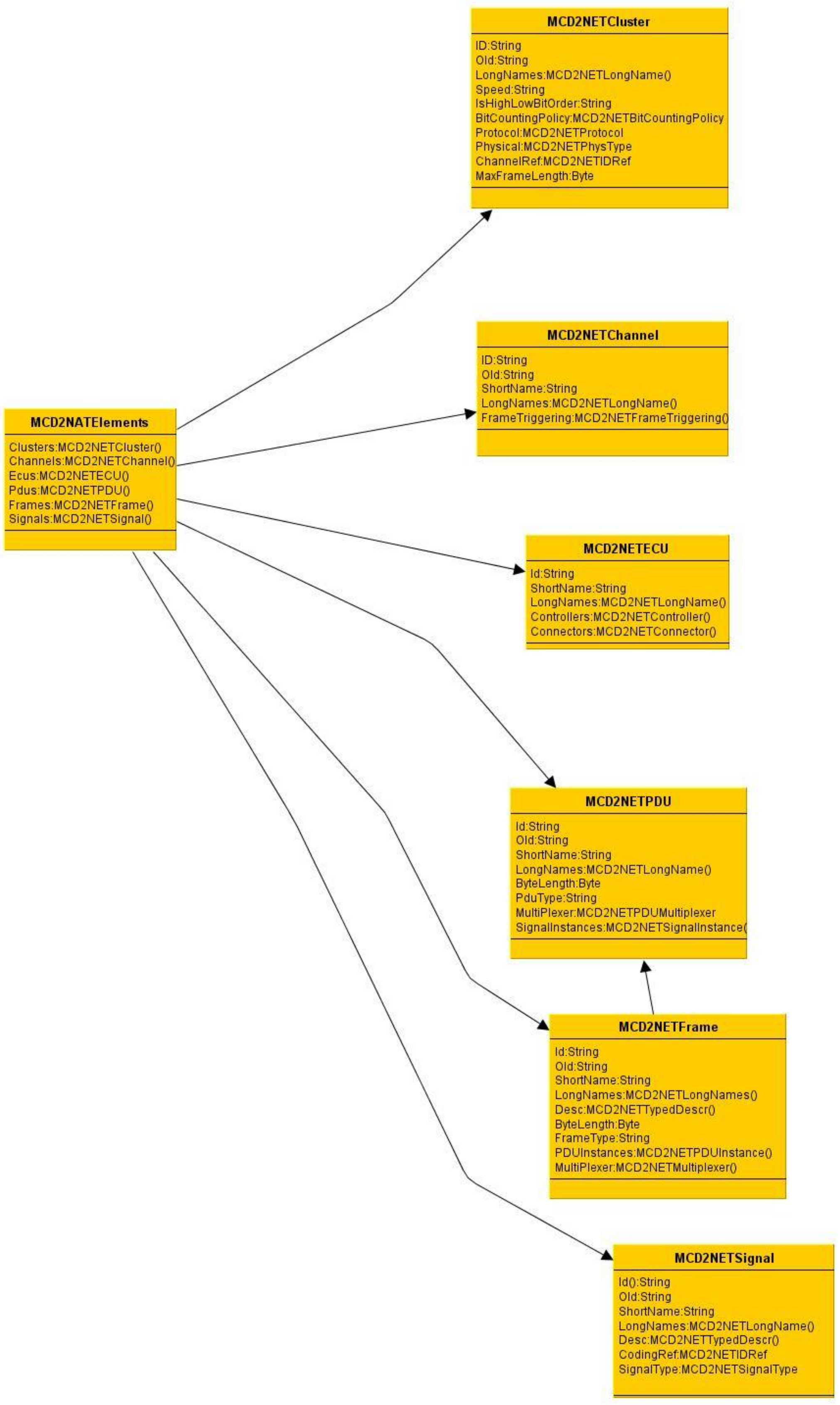


Figure 7. FIBEX loader MCD2NETElements classes' structure.

A detailed description of the main classes that are used in processing the information from xml file is given in Figures 8-13.

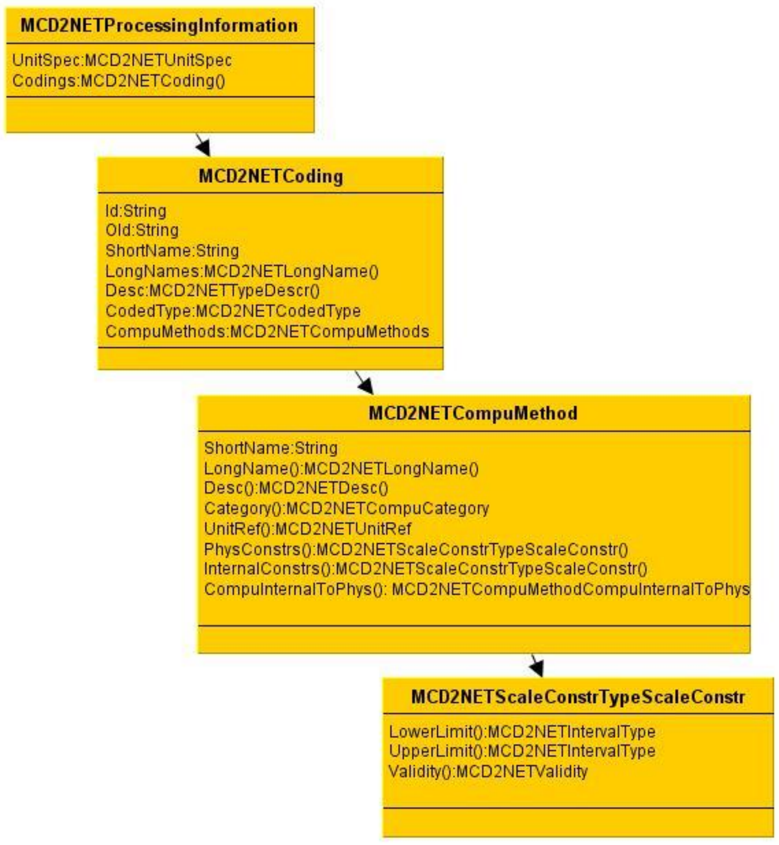

MCD2NETProcessingInformation
UnitSpec:MCD2NETUnitSpec
Codings:MCD2NETCoding()
MCD2NETCoding
Id:String
Old:String
ShortName:String
LongNames:MCD2NETLongName()
Desc:MCD2NETTypeDescr()
CodedType:MCD2NETCodedType
CompuMethods:MCD2NETCompuMethods
MCD2NETCompuMethod
ShortName:String
LongName():MCD2NETLongName()
Desc():MCD2NETDesc()
Category():MCD2NETCompuCategory
UnitRef():MCD2NETUnitRef
PhysConstrs():MCD2NETScaleConstrTypeScaleConstr()
InternalConstrs():MCD2NETScaleConstrTypeScaleConstr()
CompuInternalToPhys(): MCD2NETCompuMethodCompuInternalToPhys
MCD2NETScaleConstrTypeScaleConstr
LowerLimit():MCD2NETIntervalType
UpperLimit():MCD2NETIntervalType
Validity():MCD2NETValidity

Figure 8. FIBEX loader MCD2NETProcessingInformation class structure.

The MCD2NETCoding subclass described in Figure 8 contains all information regarding signal coding, limits, units, and computational methods.

A short overview of the MCD2NETSignal class is shown in Figure 9. It contains references to MCD2NETCoding, MCD2NETDesc and MCD2NETSignalType which contain all additional information about the signal. Every signal has a unique ID that is stored in the MCD2NETSignalInstance class as a reference. The MCD2NETSignalInstance class is a subclass of the MCD2NETPDU described in Figure 10.

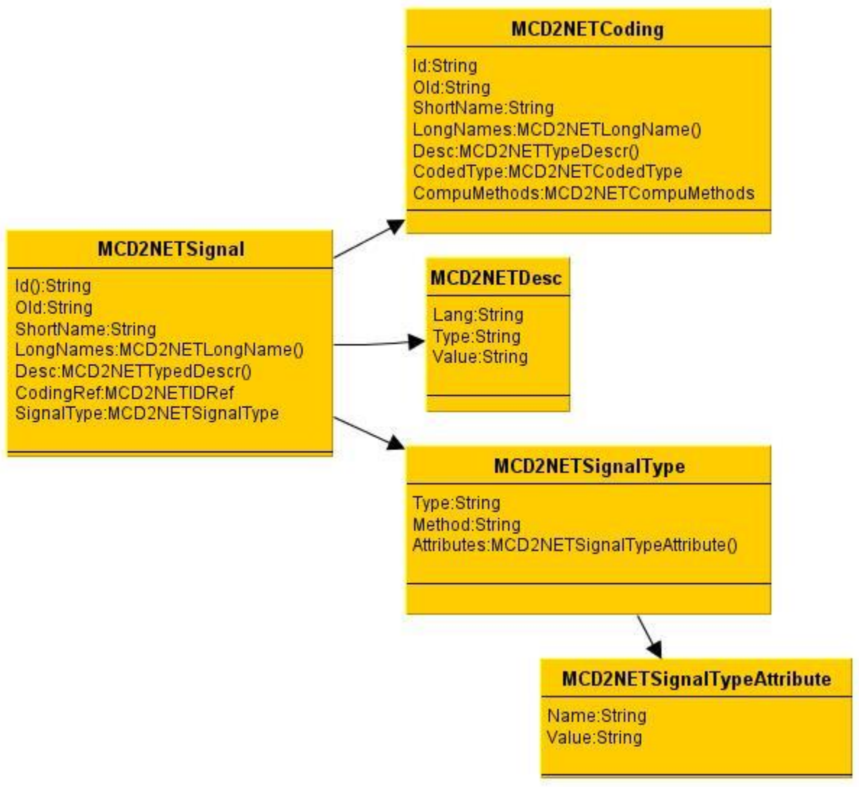


Figure 9. FIBEX loader MCD2NETSignal class structure.

The MCD2NETPDU class contains payload and control information. A PDU may or may not contain signal instances and a multiplexer. The implementation of multiplexing in more details is given in Section 3.3 of this thesis work. A PDU is the data format which is used for storing data within a frame.

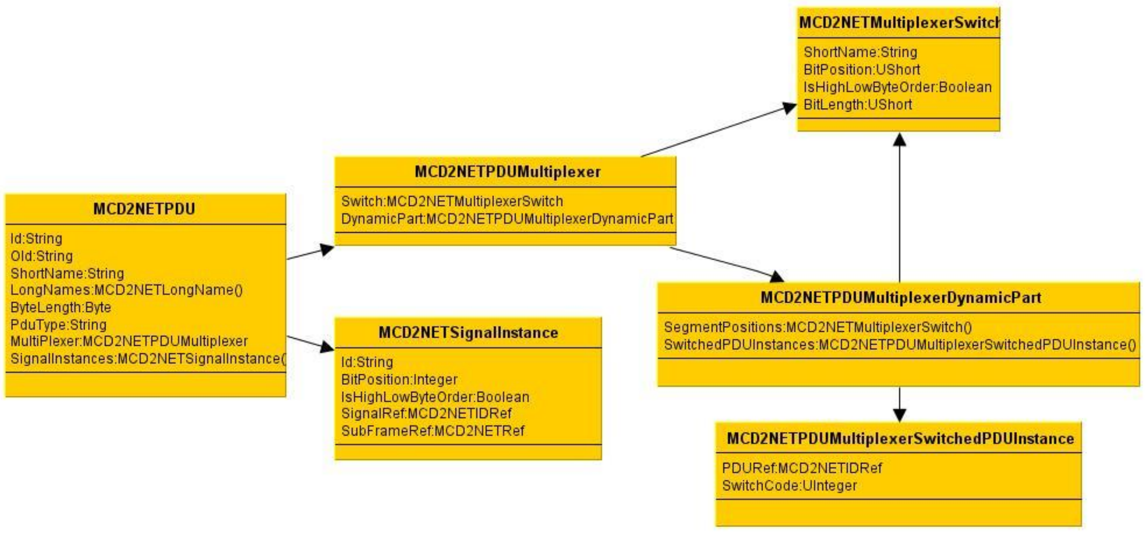

MCD2NETMultiplexerSwitch
ShortName:String
BitPosition:UShort
IsHighLowByteOrder:Boolean
BitLength:UShort
MCD2NETPDUMultiplexer
Switch:MCD2NETMultiplexerSwitch
DynamicPart:MCD2NETPDUMultiplexerDynamicPart
MCD2NETPDU
Id:String
Old:String
ShortName:String
LongNames:MCD2NETLongName()
ByteLength:Byte
PduType:String
MultiPlexer:MCD2NETPDUMultiplexer
SignalInstances:MCD2NETSignalInstance(
MCD2NETSignalInstance
Id:String
BitPosition:Integer
IsHighLowByteOrder:Boolean
SignalRef:MCD2NETIDRef
SubFrameRef:MCD2NETRef
MCD2NETPDUMultiplexerDynamicPart
SegmentPositions:MCD2NETMultiplexerSwitch()
SwitchedPDUInstances:MCD2NETPDUMultiplexerSwitchedPDUInstance()
MCD2NETPDUMultiplexerSwitchedPDUInstance
PDURef:MCD2NETIDRef
SwitchCode:UInteger

Figure 10. FIBEX loader MCD2NETPDU class structure.

The relation between a PDU and a frame is described in Figure 11.

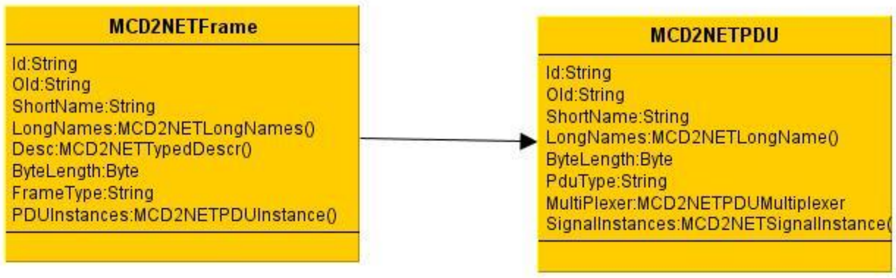


Figure 11. FIBEX loader MCD2NETFrame class structure

Information about received and sent frames can be accessed from the channel. The reference to each frame is stored in MCD2NETFrameTriggering class, the structure of which can be seen in Figure 12. The current hardware implementation has only one CAN channel.

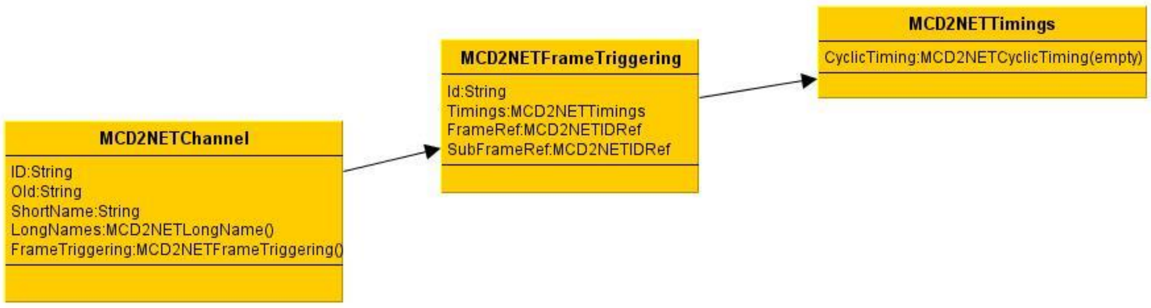


Figure 12. FIBEX loader MCD2NETChannel class structure.

All frames in the network can be either received or transmitted frames. This information is accessible from an input (receiving frame) port or an output (transmitting frame) port. Each port in the network refers to a specific ECU that it belongs to. The structure of the ECU class can be seen in Figure 13.

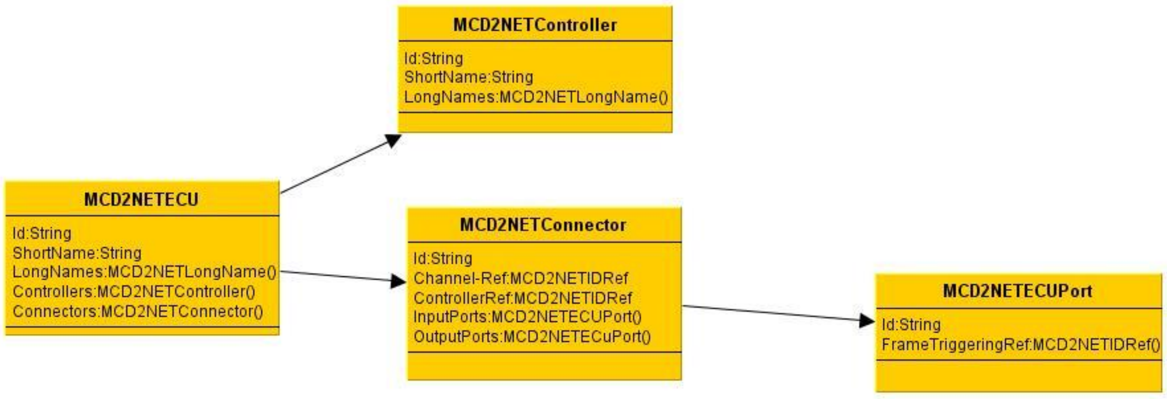

MCD2NETController
Id:String
ShortName:String
LongNames:MCD2NETLongName()
MCD2NETECU
Id:String
ShortName:String
LongNames:MCD2NETLongName()
Controllers:MCD2NETController()
Connectors:MCD2NETConnector()
MCD2NETConnector
Id:String
Channel-Ref:MCD2NETIDRef
ControllerRef:MCD2NETIDRef
InputPorts:MCD2NETECUPort()
OutputPorts:MCD2NETECuPort()
MCD2NETECUPort
Id:String
FrameTriggeringRef:MCD2NETIDRef()

Figure 13. FIBEX loader MCD2NETECU class structure.

As described in Figure 6, the architecture for MCD2NETDocument class is used for a FIBEX loader implementation for a CAN. Following specifications for CAN, the following modifications in MCD2NETDocument class were carried out: the MCD2NETController class was extended with CAN specific parameters; MCD2NETFrameTriggering holds additional values as well.

In the next step, the FIBEX configuration file is being deserialized and converted to MCD2NETDocument type. This allows us to refer to the information from FIBEX file as to a class-object model. The following step is to process the information and parse it.

The general flow of composing a FIBEX model which is passed to a CAN interface can be seen in Appendix 1.

3.3 Multiplexing implementation

The most challenging part in FIBEX Loader implementation was the implementation of multiplexing. Unlike CANDBC, another file format was used to configure CAN networks, in FIBEX files, the frame’s data has a different format. It consists of PDUs that include signals. Since the CANDBC implementation already exists, it was decided to adapt the final FIBEX model to a CANDBC model. That will simplify the integration into MicroLabCar software and will allow using common interfaces. This part is described in section 3.4.

Since the CANDBC frame contains only a list of signals, a similar list for each frame was created. The signals are being read from PDUs and depending on the type of PDU, they store the information about multiplexing if one is present. As y can be seen from Figure 10, in case of a multiplexed PDU, it contains a switch and a dynamic part. Following the data format of a CANDBC file, a switch should be converted to a signal as a multiplexer signal and added to a general payload of a frame. A constructor with a specific argument handles the converting as shown in Figure 14.

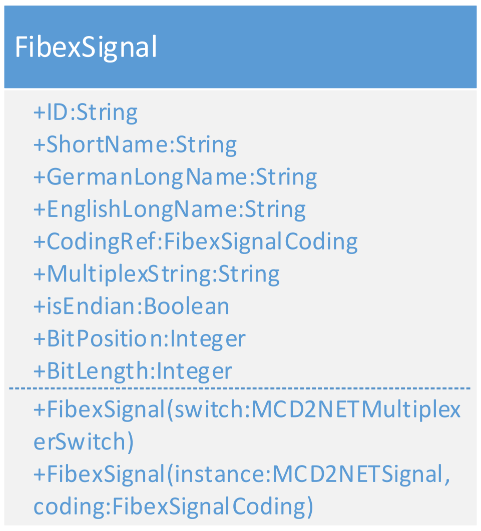


Figure 14. A FibexSignal class structure

Each signal contains information about multiplexing and a multiplexer itself if it is multiplexed. The multiplexer signals, switches in our case, have a key word “switch” in ID property.

3.4 Integration of a FIBEX loader into a CAN interface

As mentioned before, a CAN network can be configured either from a CANDBC file, which is already implemented, or a FIBEX file. Therefore in this part of my thesis work, it was important to come up with an idea of how to generalize and unify the integration of CANDBC and FIBEX files. The task was to make CAN architecture easily extendable and the changes within any of configuration files’ loaders should be easily applied. Therefore approach adopted was to use a common interface for reading both CANDBC and FIBEX files, allowing the microLC architecture to be indifferent to any changes in external DLLs.

Figure 15 describes CAN interface architecture for reading information from the FIBEX loader. Each of the classes implements an interface, which is referred to further in logic and the GUI layer. For the CANFIBEXProtocolService class, an adapter class SerialProtocolAdapter was used. The purpose of using an adapter class in this case was to overloading interface methods

ProcessEvents(variable as HardwareEvent) and ResolvePayload(variable as HardwareEvent) with methods with different signatures – ProcessEvents(event as CANRxEvent) and ResolvePayload(event as CANRxEvent). The idea behind the interfaces is to possibly use it for other protocols as well. That means they should be as general as it is required in order to be implemented by other protocols' services.

The same architecture is implemented for CANDBC loader, so the current version of the software is already able to work with both libraries.

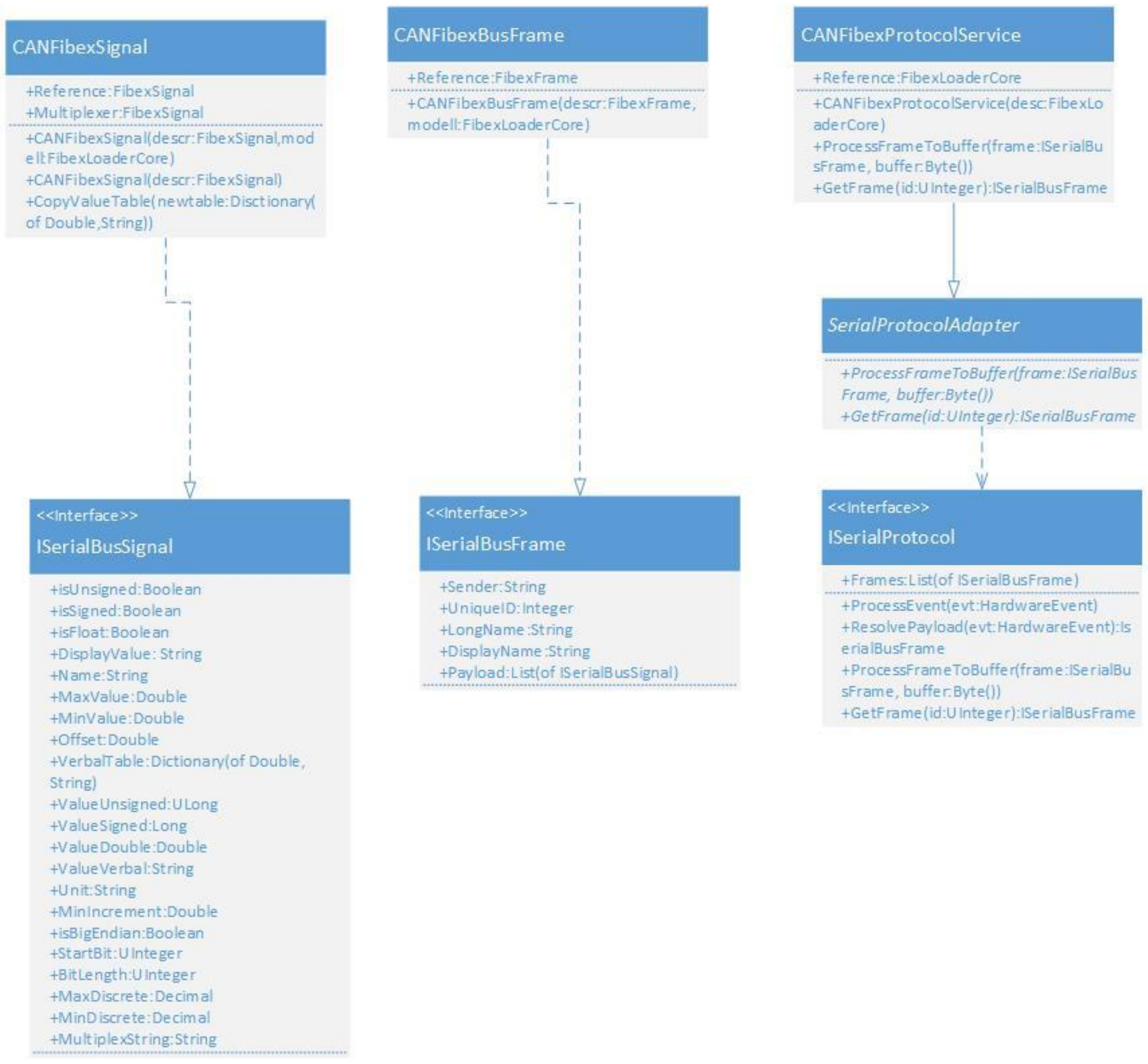

CANFibexSignal
+Reference:FibexSignal
+Multiplexer:FibexSignal
+CANFibexSignal(descr:FibexSignal,modelt:FibexLoaderCore)
+CANFibexSignal(descr:FibexSignal)
+CopyValueTable(newtable:Disctionary(of Double,String))
CANFibexBusFrame
+Reference:FibexFrame
+CANFibexBusFrame(descr:FibexFrame, modell:FibexLoaderCore)
CANFibexProtocolService
+Reference:FibexLoaderCore
+CANFibexProtocolService(desc:FibexLoaderCore)
+ProcessFrameToBuffer(frame:ISerialBusFrame, buffer:Byte())
+GetFrame(id:UInteger):ISerialBusFrame
SerialProtocolAdapter
+ProcessFrameToBuffer(frame:ISerialBusFrame, buffer:Byte())
+GetFrame(id:UInteger):ISerialBusFrame
<<Interface>>
ISerialBusSignal
+isUnsigned:Boolean
+isSigned:Boolean
+isFloat:Boolean
+DisplayValue: String
+Name:String
+MaxValue:Double
+MinValue:Double
+Offset:Double
+VerbalTable:Dictionary(of Double, String)
+ValueUnsigned:ULong
+ValueSigned:Long
+ValueDouble:Double
+ValueVerbal:String
+Unit:String
+MinIncrement:Double
+isBigEndian:Boolean
+StartBit:UInteger
+BitLength:UInteger
+MaxDiscrete:Decimal
+MinDiscrete:Decimal
+MultiplexString:String
<<Interface>>
ISerialBusFrame
+Sender:String
+UniqueID:Integer
+LongName:String
+DisplayName:String
+Payload:List(of ISerialBusSignal)
<<Interface>>
ISerialProtocol
+Frames:List(of ISerialBusFrame)
+ProcessEvent(evt:HardwareEvent)
+ResolvePayload(evt:HardwareEvent):ISerialBusFrame
+ProcessFrameToBuffer(frame:ISerialBusFrame, buffer:Byte())
+GetFrame(id:UInteger):ISerialBusFrame

Figure 15. CAN interface architecture for reading information from FIBEX loader.

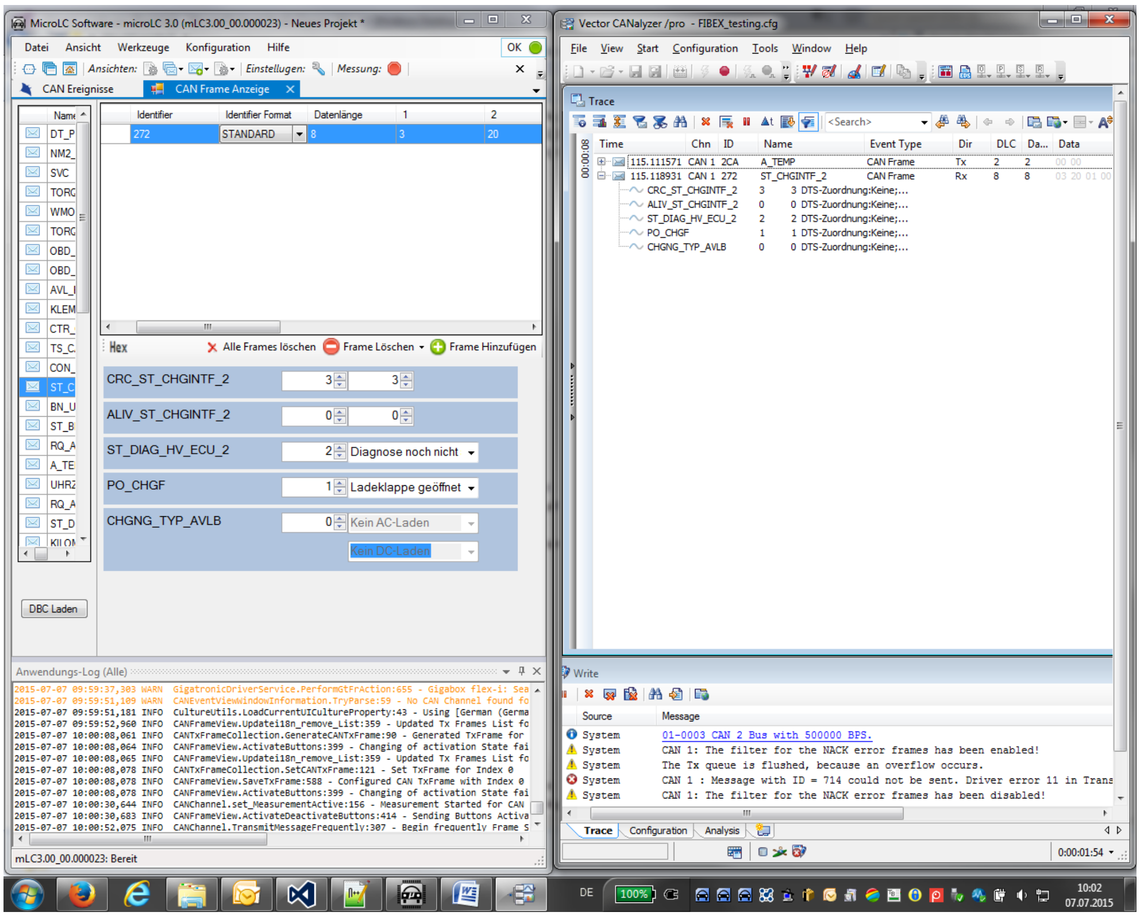
3.5 Testing

The testing of the FIBEX loader for CAN was carried out using a microLC3 device (Picture 1) and a CANcaseXL device (Picture 2), which is a professional CAN interface produced by Vector, to simulate communication in a CAN network.

Picture 2. CANcaseXL device

For the monitoring purposes, the CANalyzer tool from the same producer was used.

For both devices, FIBEX configuration files were loaded. The results of sending frames from microLC3 and CANcaseXL can be seen from the screenshot of a CANalayzer window in Picture 3 and 4.


MicroLC Software - microLC 3.0 (mLC3.00_00.000023) - Neues Projekt *

Identifier	Identifier Format	Datenlänge	1	2
272	STANDARD	8	3	20

CRC_ST_CHGINTF_2	3	3
ALIV_ST_CHGINTF_2	0	0
ST_DIAG_HV_ECU_2	2	Diagnose noch nicht
PO_CHGF	1	Ladeklappe geöffnet
CHGNG_TYP_AVLB	0	Kein AC-Laden / Kein DC-Laden

Anwendungs-Log (Alle)

```
2015-07-07 09:59:37,303 WARN  GigatronicDriverService.PerformGtFrAction:655 - Gigabox flex-i: Sea
2015-07-07 09:59:51,109 WARN  CANEventViewWindowInformation.TryParse:59 - No CAN Channel found fo
2015-07-07 09:59:51,181 INFO  CultureUtils.LoadCurrentUICultureProperty:43 - Using [German (Germa
2015-07-07 09:59:52,960 INFO  CANFrameView.Updatei18n_remove_List:359 - Updated Tx Frames List fo
2015-07-07 10:00:08,061 INFO  CANTxFrameCollection.GenerateCANTxFrame:90 - Generated TxFrame for
2015-07-07 10:00:08,064 INFO  CANFrameView.ActivateButtons:399 - Changing of activation State fai
2015-07-07 10:00:08,065 INFO  CANFrameView.Updatei18n_remove_List:359 - Updated Tx Frames List fo
2015-07-07 10:00:08,078 INFO  CANTxFrameCollection.SetCANTxFrame:121 - Set TxFrame for Index 0
2015-07-07 10:00:08,078 INFO  CANFrameView.SaveTxFrame:588 - Configured CAN TxFrame with Index 0
2015-07-07 10:00:08,078 INFO  CANFrameView.ActivateButtons:399 - Changing of activation State fai
2015-07-07 10:00:30,644 INFO  CANChannel.set_MeasurementActive:156 - Measurement Started for CAN
2015-07-07 10:00:30,683 INFO  CANFrameView.ActivateDeactivateButtons:414 - Sending Buttons Activa
2015-07-07 10:00:52,075 INFO  CANChannel.TransmitMessageFrequently:307 - Begin frequently Frame S
```

mLC3.00_00.000023: Bereit

Vector CANalyzer /pro - FIBEX_testing.cfg

Trace

Time	Chn	ID	Name	Event Type	Dir	DLC	Da...	Data
115.111571	CAN 1	2CA	A_TEMP	CAN Frame	Tx	2	2	00 00
115.118931	CAN 1	272	ST_CHGINTF_2	CAN Frame	Rx	8	8	03 20 01 00
CRC_ST_CHGINTF_2	3	3	DTS-Zuordnung:Keine;...					
ALIV_ST_CHGINTF_2	0	0	DTS-Zuordnung:Keine;...					
ST_DIAG_HV_ECU_2	2	2	DTS-Zuordnung:Keine;...					
PO_CHGF	1	1	DTS-Zuordnung:Keine;...					
CHGNG_TYP_AVLB	0	0	DTS-Zuordnung:Keine;...					

Write

Source	Message
System	01-0003 CAN 2 Bus with 500000 BPS.
System	CAN 1: The filter for the NACK error frames has been enabled!
System	The Tx queue is flushed, because an overflow occurs.
System	CAN 1 : Message with ID = 714 could not be sent. Driver error 11 in Trans
System	CAN 1: The filter for the NACK error frames has been disabled!

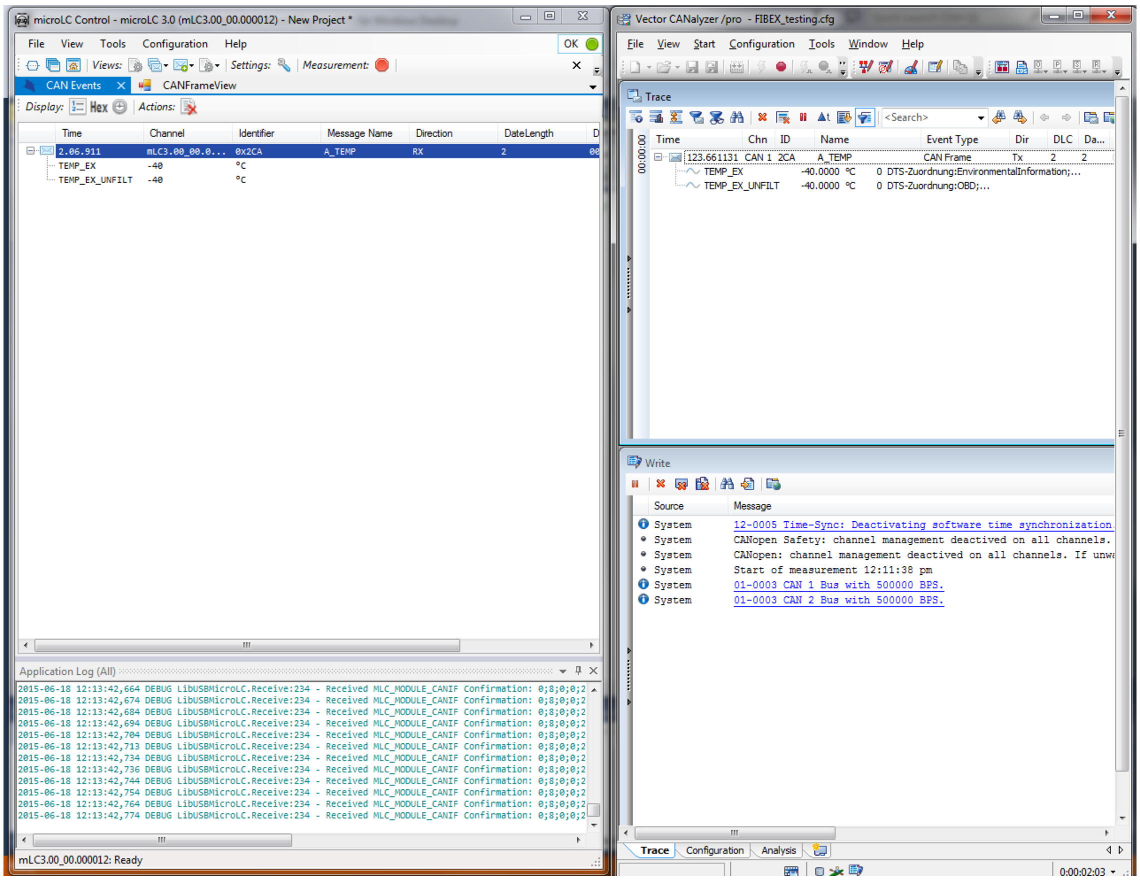
0:00:01:54

10:02 07.07.2015

Picture 3. Sending a frame from microLC3.

As can be seen from Picture 3, a frame from microLC3 is sent. The same frame is received by the CANcaseXL as it is shown on the right side of the picture 3. After looking at the detailed description of the received frame in a CANalyzer window, one can confirm the correctness of the received data by comparing it to the signal values in the microLC3 window.

Picture 4 describes sending a frame from the CANcaseXL to the microLC3. The detailed descriptions of both the transmitted frame in CANcaseXL and the received frame in a microLC3 window demonstrate the correctness of the processing of transmitted and received data on both devices.


microLC Control - microLC 3.0 (mLC3.00_00.000012) - New Project *
File View Tools Configuration Help OK
Views: Settings: Measurement:
CAN Events CANFrameView
Display: Hex Actions:

Time	Channel	Identifier	Message Name	Direction	DateLength	D
2.06.911	mLC3.00_00.0...	0x2CA	A_TEMP	RX	2	00
TEMP_EX	-40	°C				
TEMP_EX_UNFILT	-40	°C				

Application Log (All)

```
2015-06-18 12:13:42,664 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,674 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,684 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,694 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,704 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,713 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,734 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,736 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,744 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,754 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,764 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
2015-06-18 12:13:42,774 DEBUG LibUSBMicroLC.Receive:234 - Received MLC_MODULE_CANIF Confirmation: 0;8;0;0;2
```

mLC3.00_00.000012: Ready

Vector CANalyzer /pro - FIBEX_testing.cfg
File View Start Configuration Tools Window Help
Trace
<Search>

Time	Chn	ID	Name	Event Type	Dir	DLC	Da...
123.661131	CAN 1	2CA	A_TEMP	CAN Frame	Tx	2	2

```
TEMP_EX         -40.0000 °C   0 DTS-Zuordnung:EnvironmentalInformation;...
TEMP_EX_UNFILT  -40.0000 °C   0 DTS-Zuordnung:OBD;...
```

Write

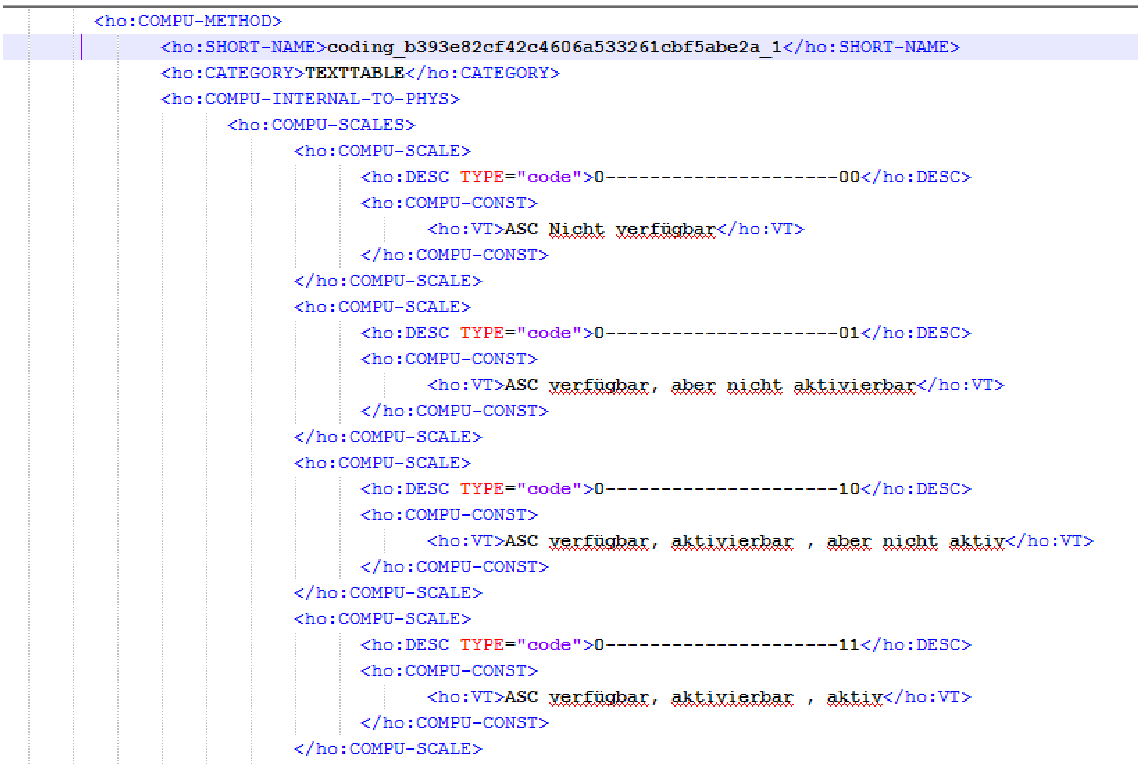
Source	Message
System	12-0005 Time-Sync: Deactivating software time synchronization.
System	CANopen Safety: channel management deactived on all channels.
System	CANopen: channel management deactived on all channels. If unwa
System	Start of measurement 12:11:38 pm
System	01-0003 CAN 1 Bus with 500000 BPS.
System	01-0003 CAN 2 Bus with 500000 BPS.

Picture 4. Sending a frame from CANcaseXL.

4 IMPLEMENTATION OF FIBEX SPECIFIC EXTENSIONS

4.1 Bit coding

Unlike a CANDBC configuration file that has textable or numerical interpretation of a signal, a FIBEX file has a ”coded” type of a signal interpretation. A signal encoding in this case has multiple bit patterns, each corresponding to a certain signal state. An example of such bit coding can be seen in Picture 5.

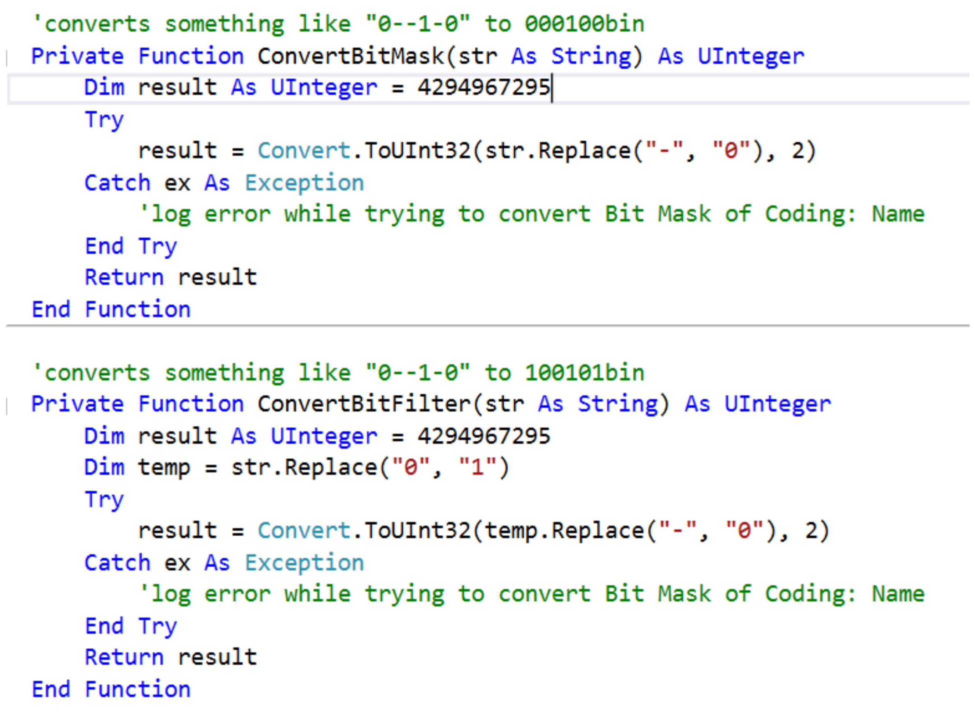


```
<ho:COMPU-METHOD>
    <ho:SHORT-NAME>coding_b393e82cf42c4606a533261cbf5abe2a_1</ho:SHORT-NAME>
    <ho:CATEGORY>TEXTTABLE</ho:CATEGORY>
    <ho:COMPU-INTERNAL-TO-PHYS>
        <ho:COMPU-SCALES>
            <ho:COMPU-SCALE>
                <ho:DESC TYPE="code">0---------------------00</ho:DESC>
                <ho:COMPU-CONST>
                    <ho:VT>ASC Nicht verfügbar</ho:VT>
                </ho:COMPU-CONST>
            </ho:COMPU-SCALE>
            <ho:COMPU-SCALE>
                <ho:DESC TYPE="code">0---------------------01</ho:DESC>
                <ho:COMPU-CONST>
                    <ho:VT>ASC verfügbar, aber nicht aktivierbar</ho:VT>
                </ho:COMPU-CONST>
            </ho:COMPU-SCALE>
            <ho:COMPU-SCALE>
                <ho:DESC TYPE="code">0---------------------10</ho:DESC>
                <ho:COMPU-CONST>
                    <ho:VT>ASC verfügbar, aktivierbar , aber nicht aktiv</ho:VT>
                </ho:COMPU-CONST>
            </ho:COMPU-SCALE>
            <ho:COMPU-SCALE>
                <ho:DESC TYPE="code">0---------------------11</ho:DESC>
                <ho:COMPU-CONST>
                    <ho:VT>ASC verfügbar, aktivierbar , aktiv</ho:VT>
                </ho:COMPU-CONST>
            </ho:COMPU-SCALE>
```

Picture 5. An example of bit encoding type in a FIBEX file.

Such a type of signal encoding serves as a filter for setting signal states.

In order to be able to work with these signal patterns, a signal mask and a signal filter are calculated. The algorithms are described in Picture 6.



```
'converts something like "0--1-0" to 000100bin
Private Function ConvertBitMask(str As String) As UInteger
    Dim result As UInteger = 4294967295
    Try
        result = Convert.ToUInt32(str.Replace("-", "0"), 2)
    Catch ex As Exception
        'log error while trying to convert Bit Mask of Coding: Name
    End Try
    Return result
End Function

'converts something like "0--1-0" to 100101bin
Private Function ConvertBitFilter(str As String) As UInteger
    Dim result As UInteger = 4294967295
    Dim temp = str.Replace("0", "1")
    Try
        result = Convert.ToUInt32(temp.Replace("-", "0"), 2)
    Catch ex As Exception
        'log error while trying to convert Bit Mask of Coding: Name
    End Try
    Return result
End Function
```

Picture 6. Code implementation of bit mask and bit filter calculations.

A raw value of signal is converted to a binary format. A logical bitwise AND operation is performed on each pair of the corresponding bits of a binary signal value and a filter value. The resulting value from this operation should match with a corresponding bit mask.

For example, we have the following bit pattern:

0 - - - - 1

The filter is: 100001

The mask is: 000001

Let us take a raw value of signal as 3, which in binary format is 000011.

000011 bitwise AND 100001=>000001 which matches to a mask 000001.

So in this case, a signal state that correspons to a matching pattern would be set.

The necessary extensions in GUI and logic layer were carried ou to implement the bit coding and were demonstrated during the demo.

5 CONCLUSION

This thesis had a practical significance for the commissioning company. The work, that was done, was a part of bus systems development for the microLC project for Bosch that has been going on already for a few years. This thesis took care of configuring one specific protocol – a CAN protocol, developed by Bosch in the early 80s. It is an international standard meant for fast serial data exchange between electronic controllers in motor vehicles.

Since the hardware implementation of CAN has been carried out earlier, a standard called CANDBC was used for configuring the CAN network. This file standard has a simpler design, thus it is suitable for configuring networks with a simpler architecture. In order to support networks with more complicated design a new standard called FIBEX was developed. The implementation of network configuration using FIBEX was the aim of this thesis work.

In order to proceed with a solution to this task, a good understanding of a complicated architecture of the microLC software and strong knowledge of CAN protocol workflow were required. After a couple months of study and research, the practical part was successfully completed. A number of tests on the hardware was performed using special monitoring tools, confirming the correctness of a data flow.

The implementation of FIBEX for configuring CAN network opens many possibilities for further development of other automotive protocols. One example of those is LIN, which has been recently released within the microLC project. The FIBEX standard structure, that unifies all common characteristics of automotive protocols, allows using the current FIBEX implementation for configuring other networks with small adjustments of protocol specific extensions.

SOURCE MATERIAL

ASAM, 2013. ASAM MCD-2 NET(FIBEX) ASAM. Available at: < http://www.asam.net/> [Accessed 06.05.2014].

CIA, 2013. CAN specification. Available at: < http://www.can-cia.org> [Accessed 19.04.2015]

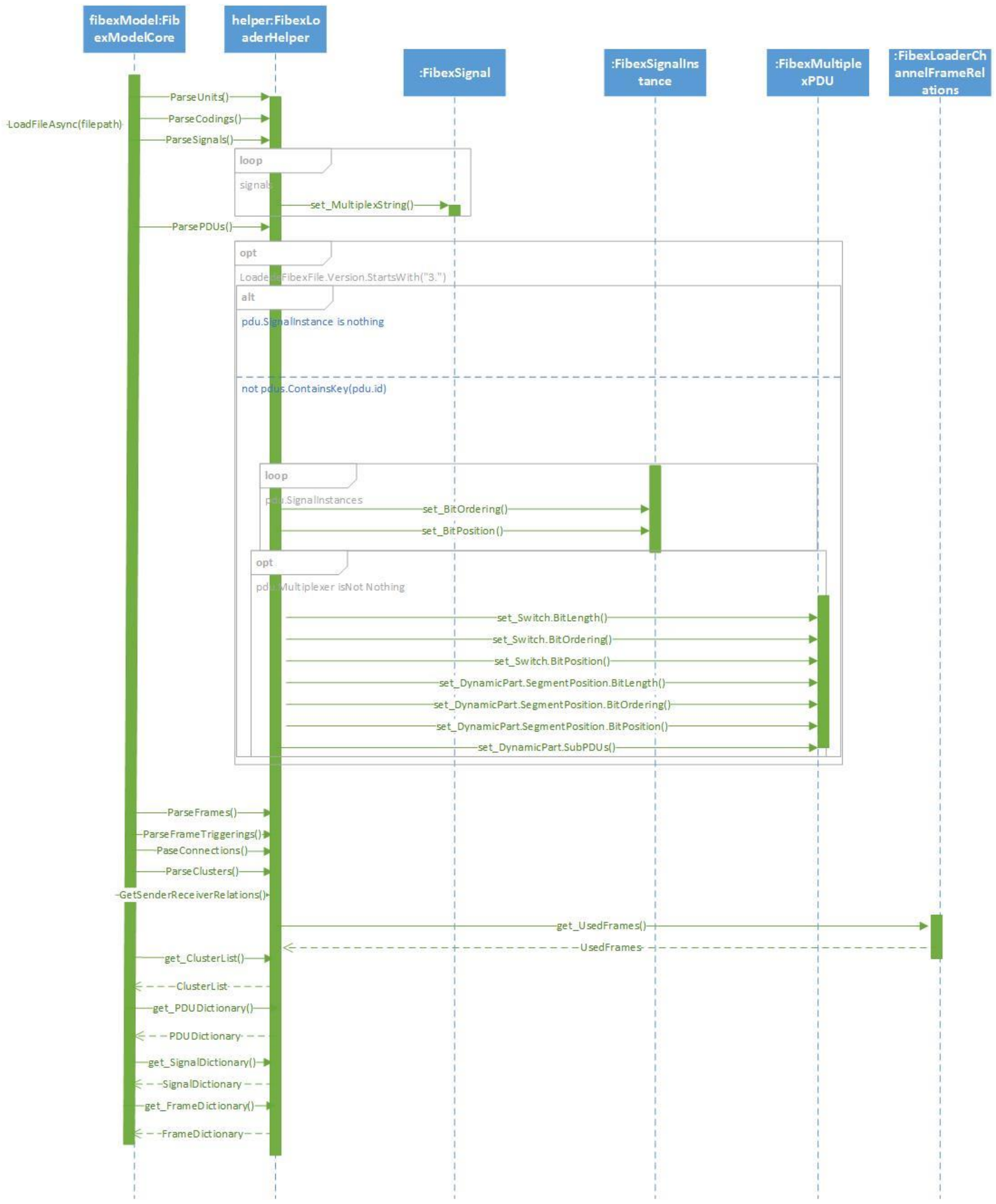
Kvaser, 2014. CAN Protocol Tutorial. Available at: <http://www.kvaser.com/can-protocol-tutorial/> [Accessed 17.04.2015].

National Instruments, 2009. FlexRay Automotive Communication Bus Overview. Available at :< http://www.ni.com/white-paper/3352/en/> [Accessed 28.03.2015].

Texas Instruments, Corrigan S., 2008. Introduction to the Controller Area Network (CAN) Texas Instruments. Available at :< http://www.ti.com/lit/an/sloa101a/sloa101a.pdf > [Accessed 02.04.2015].

Vector, 2010-2015. E-Learning module “Introduction to FlexRay”. Available at :< https://elearning.vector.com/vl_flexray_introduction_en.html > [Accessed 02.04.2015].

A sequence diagram describing a flow of composing a FIBEX model


fibexModel:FibexModelCore
helper:FibexLoaderHelper
:FibexSignal
:FibexSignalInstance
:FibexMultiplexPDU
:FibexLoaderChannelFrameRelations
LoadFileAsync(filepath)
ParseUnits()
ParseCodings()
ParseSignals()
loop
signals
set_MultiplexString()
ParsePDUs()
opt
Loader.FibexFile.Version.StartsWith("3.")
alt
pdu.SignalInstance is nothing
not pdus.ContainsKey(pdu.id)
loop
pdu.SignalInstances
set_BitOrdering()
set_BitPosition()
opt
pdu.Multiplexer isNot Nothing
set_Switch.BitLength()
set_Switch.BitOrdering()
set_Switch.BitPosition()
set_DynamicPart.SegmentPosition.BitLength()
set_DynamicPart.SegmentPosition.BitOrdering()
set_DynamicPart.SegmentPosition.BitPosition()
set_DynamicPart.SubPDUs()
ParseFrames()
ParseFrameTriggerings()
PaseConnections()
ParseClusters()
GetSenderReceiverRelations()
get_UsedFrames()
UsedFrames
get_ClusterList()
ClusterList
get_PDUDictionary()
PDUDictionary
get_SignalDictionary()
SignalDictionary
get_FrameDictionary()
FrameDictionary