Institutionen för systemteknik
Department of Electrical Engineering

Examensarbete

Implementation of CAN Communication Stack in AUTOSAR

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Johan Alexandersson och Olle Nordin

LiTH-ISY-EX-ET–15/0440–SE

Linköping 2015

Department of Electrical Engineering
Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings tekniska högskola
Linköpings universitet
581 83 Linköping

Implementation of CAN Communication Stack in AUTOSAR

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Johan Alexandersson och Olle Nordin

LiTH-ISY-EX-ET–15/0440–SE

Handledare: **Anders Nilsson**
ISY, Linköpings universitet
Simon Tegelid
ÅF Consulting

Examinator: **Anders Nilsson**
ISY, Linköpings universitet

Linköping, 15 juni 2015

Abstract

In the automotive industry today, embedded systems have reached a level of complexity which is not maintainable with the traditional approach of designing automotive embedded systems. For this purpose, many of the world's leading automotive manufacturers have formed an alliance to apprehend this problem. This has resulted in AUTOSAR, an open standardized architecture for automotive embedded systems, which strives for increased flexibility and safety regulations. This thesis will explore the possibilities of implementing a CAN Communication stack using the AUTOSAR architecture and its corresponding methodology. As a result of this thesis, a complete AUTOSAR CAN communication stack has been implemented, as well has a simulator application with the purpose of testing its functionality.

Acknowledgments

First and foremost, we would like to thank Simon Tegelid and Magnus Carlsson at ÅF, for endless support whenever needed. We would also like to thank Anders Nilsson at ISY for consistently being available with good advice. Additional accolades go out to friends and family.

Linköping, June 2015
Johan Alexandersson and Olle Nordin

Contents

Notation

Abbrevations

Abbrevation	Description
ACK	Acknowledgement
API	Application Peripheral Interface
$AUTOSAR$	Automotive Open System Architecture
BSW	Basic Software Layer
CAN	Control Area Network
$CANIf$	Can Interface
CCS	Code Composer Studio
COM	Communication
CRC	Cyclic Redundancy Check
DIO	Digital Input/Output
DLC	Data Length Code
ECU	Electronic Control Unit
$ECUC$	ECU Configuration
$ECUM$	ECU Manager
E/E	Electrical/Electronic
GIO	General-Purpose Input/Output
GUI	Graphical User Interface
HW	Hardware
ID	Identifier
IDE	Integrated Development Environment
$IPDU$	Internal Layer Protocol Data Unit
$LPDU$	Data Link Layer Protocol Data Unit
$MCAL$	Microcontroller Abstraction Layer
MCU	Microcontroller Unit
OS	Operating System
PDU	Protocol Data Unit
$PDUR$	PDU Router
REC	Receive Error Counter
RTE	Runtime Environment
RTR	Remote Transmission Request
SW	Software
SWC	Software Component
TEC	Transmit Error Counter

1

Introduction

1.1 The Company

This thesis will be conducted at ÅF Consulting in Linköping. ÅF Consulting is a Swedish consulting company with over 7000 employees all over the world [3]. The company covers many different areas including for example industry, infrastructure and technology. Our thesis will be carried out at ÅF's technology section.

1.2 Background

In the automotive industry there has always been trouble with integrating software into different hardware components. In order to solve this problem, an international open source standard was developed in collaboration between many of the world's leading automotive companies. This standard was to be named AUTOSAR and its purpose is to improve the compatibility between different ECUs, regardless of hardware and manufacturer. At ÅF there has been a project with a small electrical car model type Sinclair C5. This project has consisted of upgrading the hardware and developing some software for controlling the car. This has been done in a traditional manner with no regard to the AUTOSAR specifications. One of the components of this vehicle is a control unit which controls the different units of the vehicle for example the motor driver. The control unit basically serves as a CAN-hub, sending and receiving messages to and from other ECUs in the vehicle. This thesis will explore the possibilities of implementing this control unit in a system based on the AUTOSAR system architecture.

1.3 Purpose

The purpose with this thesis is to implement a platform for CAN-communication using the AUTOSAR system architecture, going through all software layers to a graphical interface on the PC. The main focus with this thesis will be the configuration of the BSW modules within AUTOSAR, as well as the implementation and modelling of the SWCs. The platform needs to be compatible with the CAN-network of the Sinclair, which means it needs to be able to process 36 different types of CAN IDs, and also to be able to send and receive messages with a data length of 5 bytes. In specific, the following needs to be investigated:

- How to configure the Basic Software Modules in order to achieve CAN communication.
- How to create software components with the task of reading, interpreting, and writing CAN signals.
- How to set up the AUTOSAR Runtime Environment, which connects the software components with the Basic Software Modules
- How to send and receive signals exceeding limits on internal signal lengths.
- How to create a GUI with the task of sending and receiving CAN signals and displaying these signal values on a plot.

1.4 Limitations

The following limitations are set for this project:

- The actual control unit software for the Sinclair will not be integrated in AUTOSAR software. Instead, a CAN communication stack will be implemented which will leave room for future integration of the control unit software.
- Ethical aspects of any kind will not be discussed in this thesis.

1.5 Equipment

In this thesis, the following equipment and software will be used:

- ArcCore Arctic Studio
  - An IDE based on Eclipse where AUTOSAR embedded applications are developed.
- ArcCore Arctic Core
  - An embedded platform based on the AUTOSAR specification.

- Kvaser Leaf Light HS v2
  - USB Can Adapter
- Texas Instruments TMDX570LS20SMDK
  - A development kit for Cortex ARM-R MCUs

2

Theory

2.1 AUTOSAR

Given the increased complexity in automotive embedded systems today, the automotive industry has seen a need of an open industry standard for automotive electronic architecture. This has resulted in many of the world-leading automotive companies forming a partnership, which goal is to establish this standard. The standard, as well as the partnership, is called AUTOSAR, which stands for Automotive Open System Architecture. The partnership was founded in 2002, initially by BMW, Bosch, Continental, DamienChrysler, and Volkswagen, with Siemens joining the partnership shortly thereafter. [6]

According to the AUTOSAR website, the motivation for establishing the standard are the following points:

- *"Management of E/E complexity associated with growth in functional scope"*
- *"Flexibility for product modification, upgrade and update"*
- *"Scalability of solutions within and across product lines"*
- *"Improved quality and reliability of E/E systems"*[5]

The goals with AUTOSAR are the following:

- *"Fulfillment of future vehicle requirements, such as, availability and safety, SW upgrades/ updates and maintainability"*
- *"Increased scalability and flexibility to integrate and transfer functions"*
- *"Higher penetration of "Commercial off the Shelf" SW and HW components across product lines"*

- *"Improved containment of product and process complexity and risk"*
- *"Cost optimization of scalable systems"*[5]

2.1.1 Layered Software Architecture

Since one of the main goals of AUTOSAR is to provide flexibility, a layered software architecture has been conceived. It is mainly divided up in three layers, shown in Fig. 2.1 below.

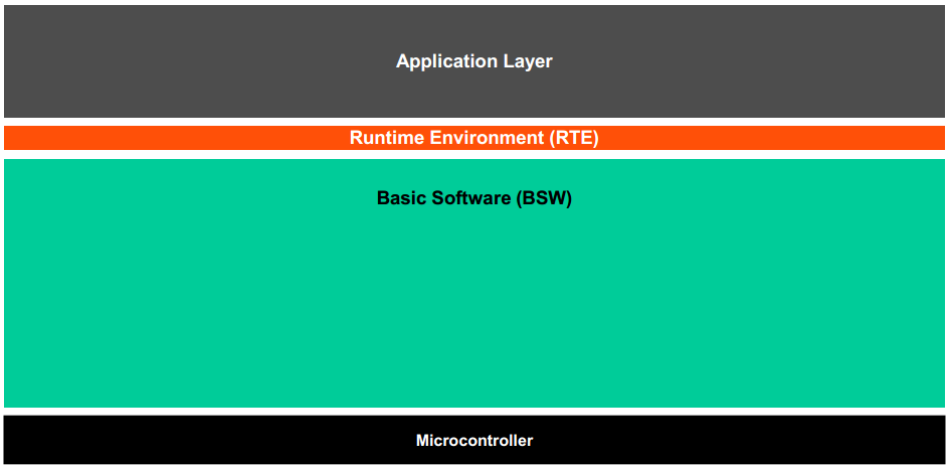

Figure 2.1: *Layered Software Architecture*

2.1.1.1 Basic Software Layer

The first layer, starting from the bottom, is called the Basic Software Layer (BSW). The purpose of this layer is to provide an hardware independent abstraction to other layers. This means that the BSW layer needs to interact with the microcontroller itself, making it hardware dependent. Due to this, the BSW layer needs to be implemented depending on what kind of hardware is used. The BSW layer contains standardized infrastructure for example the communication stack, the memory stack, diagnostic services, and the operating system. [7]

2.1.1.2 Runtime Environment

The next layer in the architecture is the Runtime Environment, commonly abbreviated RTE. The purpose of this layer is to abstract the application layer from the BSW layer. In practice, this basically means that the RTE provides the Application Layer and the Basic Software Layer with a common interface, providing interaction between the two layers. [7]

2.1.1.3 Application Layer

The final layer is the Application Layer. This layer contains application modules which are called Software Components (SWC). These contain software for the system which is completely hardware independent. This means for example that an SWC of an ECU with one kind of hardware can interact with an SWC of an ECU with another kind of hardware with great ease. [17]

2.1.2 Basic Software Layer

The Basic Software Layer is furtherly divided into sublayers, as shown in Fig. 2.2 below. Explanations for layers not used in this thesis are excluded.

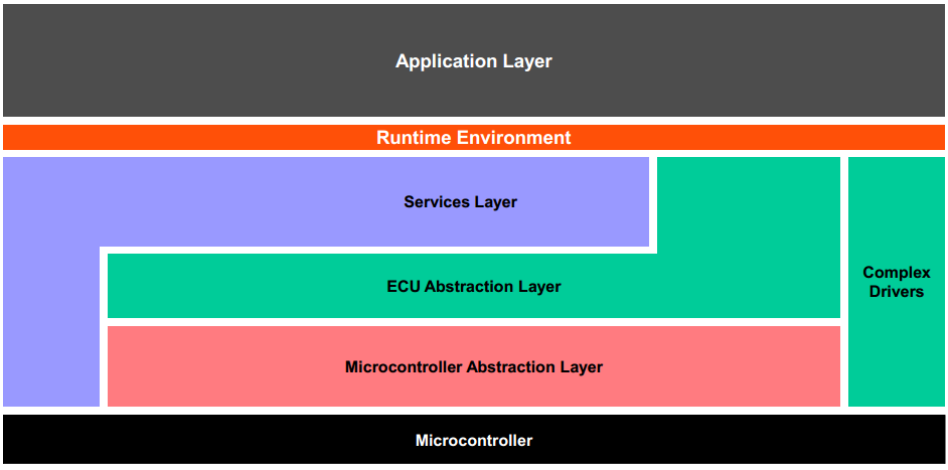

Figure 2.2: *Basic Software Layer*

2.1.2.1 Microcontroller Abstraction Layer

The Microcontroller Abstraction Layer (MCAL) is the layer at the bottom of the BSW layer. The modules in this layer interact directly with the MCU. The purpose of this layer is to make layers above this layer hardware independent, as this layer provides functions defined by the AUTOSAR specification. Given the fact that this layer interacts directly with the MCU, this layer is hardware dependent. [7]

2.1.2.2 ECU Abstraction Layer

The next layer in the BSW layer is called the ECU Abstraction Layer (ECUAL). This layer provides an interface of the drivers found in the MCAL layer to the layers situated above the ECU Abstraction Layer. Drivers for external devices such as CAN transceivers are included here as well. [7]

2.1.2.3 System Services Layer

The upper layer in the BSW layer is called the System Services Layer. This layer provides functions for the Application Layer, hence its position in the hierarchy. It contains modules for the operating system, communication services, memory services, ECU state management, and so on. [7]

2.1.3 BSW modules

This section explains some of the modules in the BSW layer, mainly the ones used in this thesis. These modules are divided into functional groups, which are shown in Fig. 2.3 below.

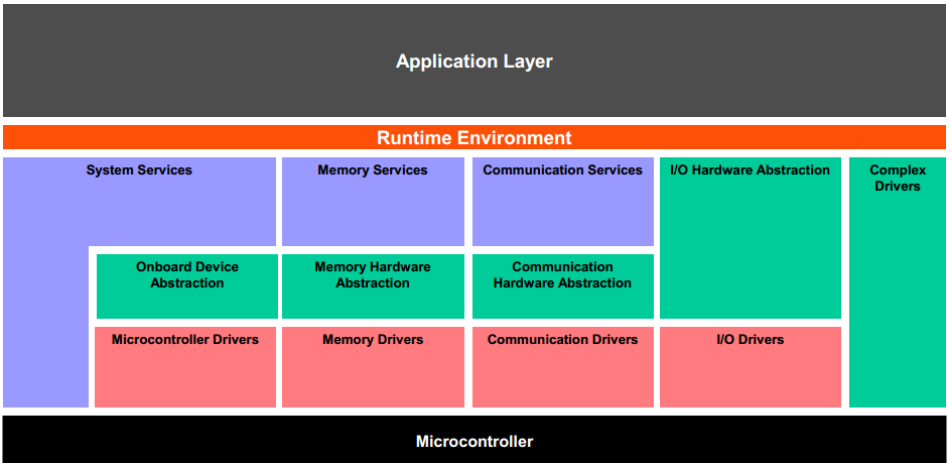

Figure 2.3: *BSW Functional Groups*

2.1.4 CAN Communication Stack

The purpose of the CAN communication stack is to accommodate a complete CAN communication interface for the Application Layer. This means that SWC's need to pay no regard to identifiers, data lengths, bit timing and so on, this is all handled by the communication stack. It is comprised by several modules fulfilling this purpose, which are shown in Fig. 2.4 below. [7]

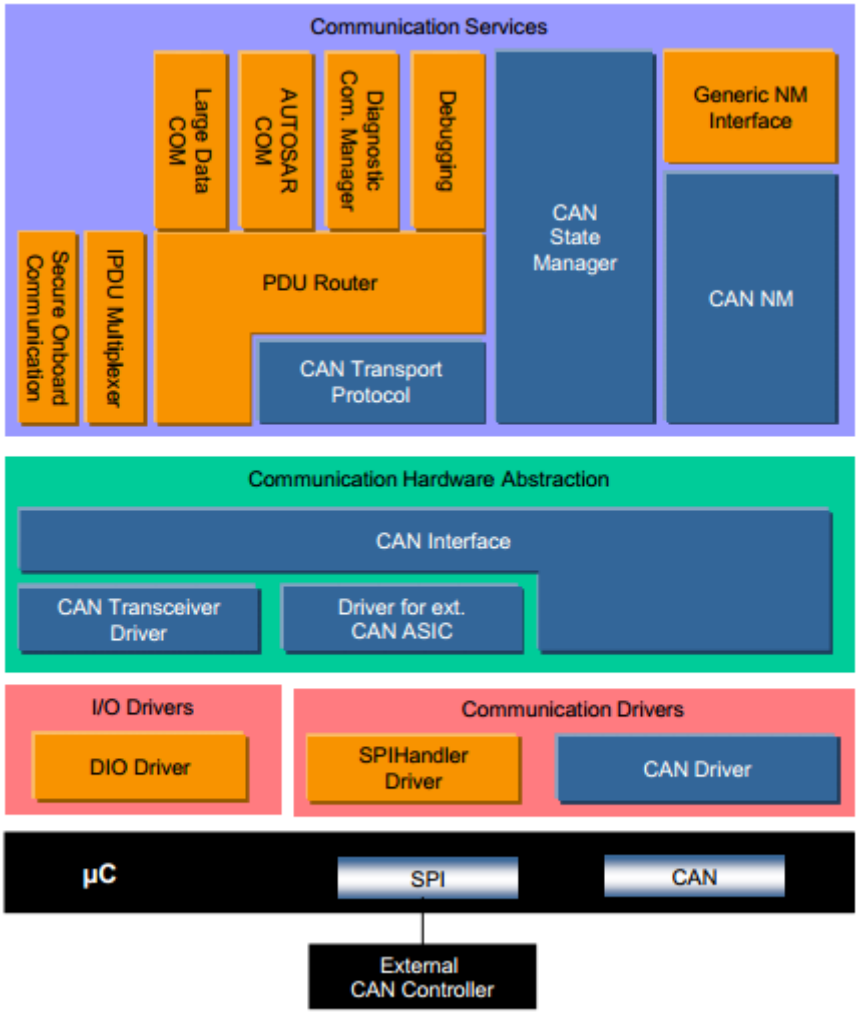

Figure 2.4: *CAN communication stack*

2.1.4.1 Protocol Data Unit

Protocol Data Unit (PDU) is a term which describes the data of a specific communication protocol. Signals sent in a CAN communication stack are grouped in CAN PDUs which in turn are mapped to CAN frames. A CAN frame is in fact also a PDU but on the link layer, and might be referred to as an L-PDU while the CAN PDUs operating on the higher layers in the stack are referred to as I-PDUs. These signal paths of the PDUs are routed between different modules of the CAN communication stack. The signal paths are routed from the COM module, through the PDU router, to the CAN interface, providing the COM module with means to initiate CAN data transmission, as well as receiving data at such

an event. All communication above the PDU router is passed through I-PDUS as well as the communication to and from the CANIf module. However, below the CANIf module, the communication is via L-PDUs. The connection of the PDU paths is shown in Fig. 2.5 below. [18]

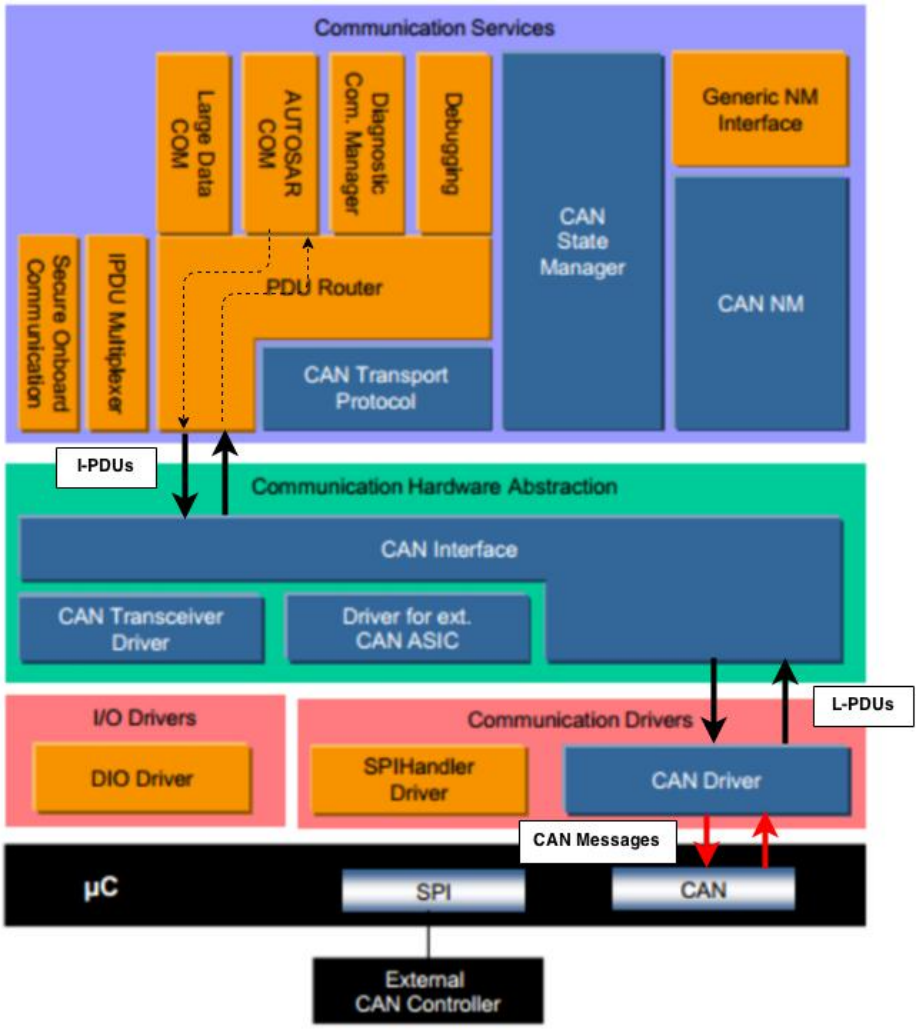

Figure 2.5: *PDU connections within the CAN communication stack*

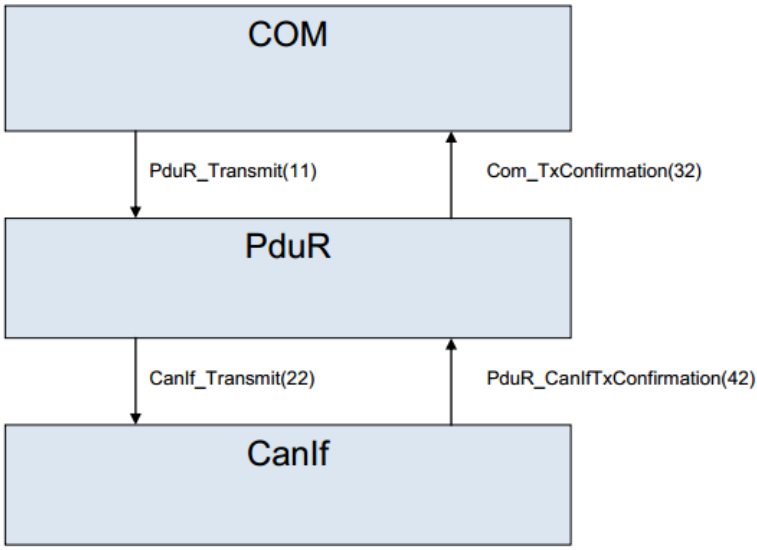

Figure 2.6: *Example of communication with PDU*

In the Fig. 2.6 above, an example of the whole process of communication through I-PDUs is shown. In this case we have an example of CAN data being sent. Initially, the COM module initiates the communication process by calling the PduR_TRANSMIT() function. Due to the routing tables of the PDU router that have been previously configured, it is known that this data's destination is the CAN bus. Thus, the CanIf_Transmit() function is called. Once the CAN interface has received an acknowledgement bit from the CAN receiver it will call the PduR_CanIfTxConfirmation() function, and the PDU router will call the Com_TxConfirmation().

2.1.4.2 CAN Driver

The CAN Driver is situated at the bottom of the CAN communication stack, which is depicted in Fig. 2.4 above. The CAN driver module provides hardware access to the upper layers within the communication stack. To do this, it provides the Communication Hardware Abstraction layer with a standardized API.[18] The CAN Driver includes several hardware objects, which are used to control the transmission and reception of L-PDUs. [9]

2.1.4.3 CAN Interface

Next in order in the CAN communication stack is the CAN interface module, CANIf. The CANIf module is located between the Communication Services Layer and the Communication Drivers Layer in the CAN communication stack, as is seen in Fig. 2.4. As well as controlling the initialization of the CAN driver module, it also provides the CAN driver module with notification services, such as

the function CanIf_RxIndication for reception, as well as CanIf_TxConfirmation for transmission. The CANIf module acts as a user of the CAN driver module's hardware objects for reception and transmission. This makes the CANIf module independent of hardware, since the CANIf module is not accessing hardware functionality directly, but instead using the CAN Driver module. [10]

2.1.4.4 PDU Router

Further up in the CAN communication stack is the PDU router. The PDUR module provides routing paths within the CAN communication stack. This way, the destination and the source of a PDU is specified. Receiving I-PDUs are routed with the CANIf module as the source, and to an upper module in the Communication Services layer of choice as the destination. Transmitting I-PDUs are done in the reverse approach, and are routed with a Communication Services Layer module of choice as the source, and the CAN Interface Layer as the destination. [13]

2.1.4.5 COM

The COM module is placed at the top of the CAN communication stack. The COM module handles all of the signals going to and from the RTE, as sender and receiver signals. These signals are each referring to an individual I-PDU for the communication throughout the BSW layer. [11]

2.1.4.6 Operating system

The OS module is part of the system services layer. The OS provides an interface for startup as well as the shutdown of the system. Moreover, the OS also manages the activation/termination of tasks and the execution of these in a scheduled order based on priority. There are two types of tasks namely extended task as well as basic task. As can be seen in Fig. 2.7 below the extended task consists of four different states, Wait state, Running state, Ready state and Suspended state. The task changes state depending on state transitions. The state transitions are the following: Preempt, Start, Terminate, Wait, Release, and Activate. Preempt is a state transition where the task makes a transition from Running to Ready, it occurs when the OS task scheduler starts another task with higher priority. The Start transition occurs when a task is in the Ready state and is being selected by the task scheduler. The Termination transition occurs when a termination of task function is called, for example TerminateTask(), the task makes a transition into the suspended state. The Wait transition occurs in extended task when the task is in the Running state but needs to wait for an event to occur. As for the activation of an extended task, it can be done for example through the usage of alarms, this will cause the task to enter the Ready state. These alarms could for example be set to trigger a specific event for example an event telling that a specific resource needed by an extended task is available. That event will trigger the extended task to go from the waiting state into the ready state and causes the task to be executed, this step transition is called Release. [22]

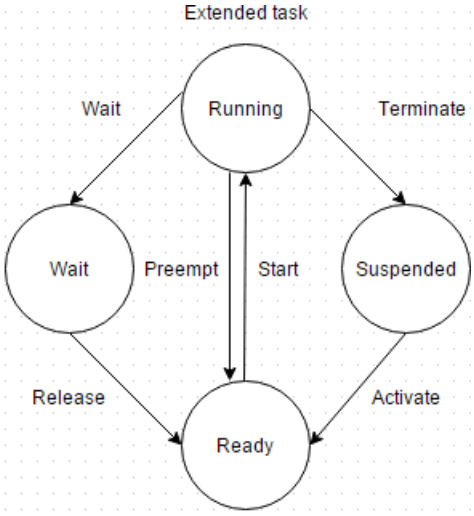

Figure 2.7: *States of an extended task*

There are also basic tasks which are not using events to be triggered, they are therefore lacking the "Wait" state. This can be seen in the Fig. 2.8 below. These tasks will instead be activated via the triggering of an alarm or by being activated via other tasks. Similar to the transitions in the extended task, the basic task uses most of them except for the ones regarding the Wait state. [22]

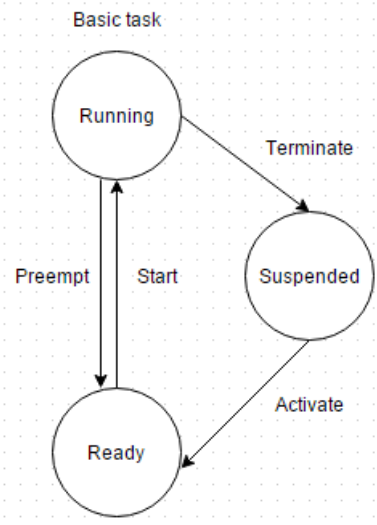

Figure 2.8: *Illustration of a basic task*

In the Fig. 2.9 below, there is an example of the execution of tasks based on priority.

TaskID	Priority	Execution Time	Deadline (=Period)
A	High	1	5
B	Medium	3	10
C	Low	5	15

Figure 2.9: *Task priorities*

Given these specifications the tasks will be executed in the following order: A -> B -> C. However, since A will occur once again during the execution of C, A will be executed and interrupt C. This is due to the priority between the tasks, which can be seen in the Fig. 2.10 below

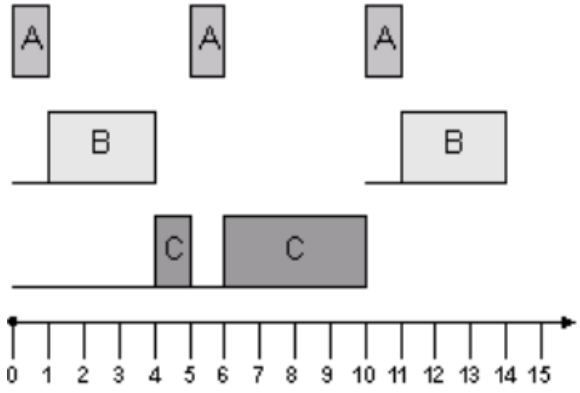

Figure 2.10: *Example of Task Execution*

2.1.4.7 ECU manager

The ECU manager controls the initialization of the OS, as well as the initialization of the MCU-drivers for example the DIO-drivers, CAN-drivers, and the Port-drivers. First and foremost all the drivers necessary for the OS functionality are initialized and there on after, a callout for the initialization of additional drivers is used, an example of this is shown below. Moreover, the ECUM also controls the shutdown of the OS as well. The startup sequence of the ECUM is shown in Fig. 2.11 below.[12]

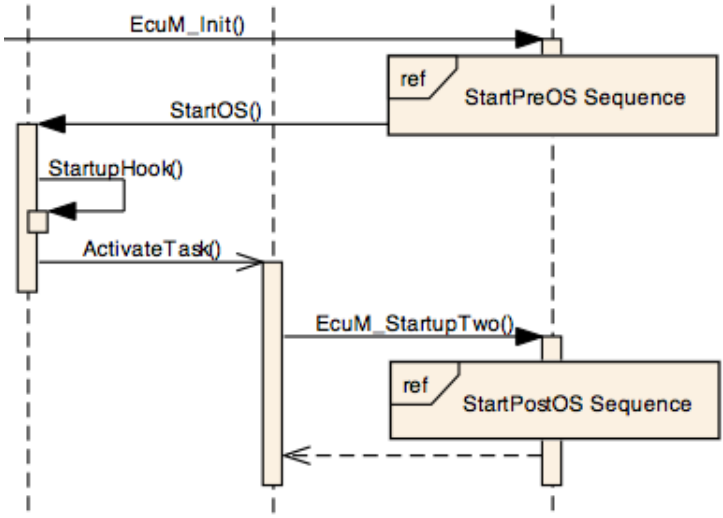

Figure 2.11: *EcuM startup sequence*

2.1.5 Application Layer

The purpose of the application layer is to provide the actual functionality of the system. This is done through the usage of SWCs, which are components containing software. An example of such a component is displayed in Fig. 2.12 below. [14]

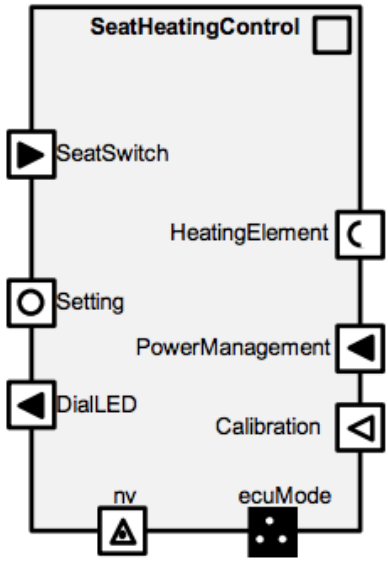

Figure 2.12: *Example of software component*

2.1.5.1 Ports

Fig. 2.12 shows an example of the port setup of an SWC. These ports are used to connect the SWC to other instances. The most basic type of ports are the sender and receiver ports. These are used to provide means of data transmission between different SWCs, or between the SWC and the RTE. Sender/Receiver ports are depicted as squares with black arrows. For example, in Fig. 2.12 the port "SeatSwitch" is a receiver port, and the port "DialLED" is a sender port. A receiver port must have a corresponding sender port and vice versa. This is shown in Fig. 2.13 below.

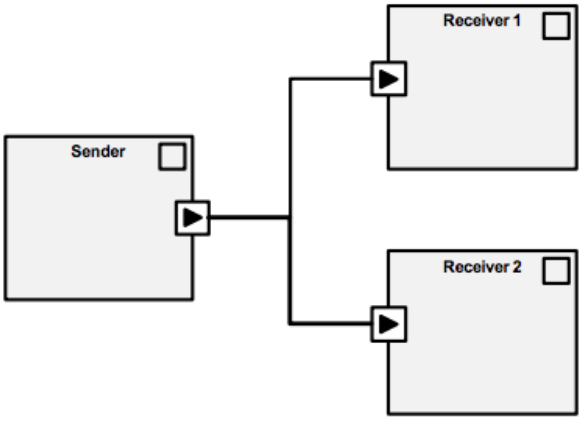

Figure 2.13: *Example of sender/receiver interface*

Another common type of port interface is the Client/Server. These ports are used when the SWC needs to use a function (in the application layer these are known as *runnables*) contained within another SWC or a function of a BSW module. The client port (HeatingElement in the Fig. 2.12) is the port requesting a function to be called, and the server port (Setting in the Fig. 2.12) is the port belonging to the entity who provides the function. Below is a Fig. 2.14 which shows Client/Server connections between different SWCs.[14]

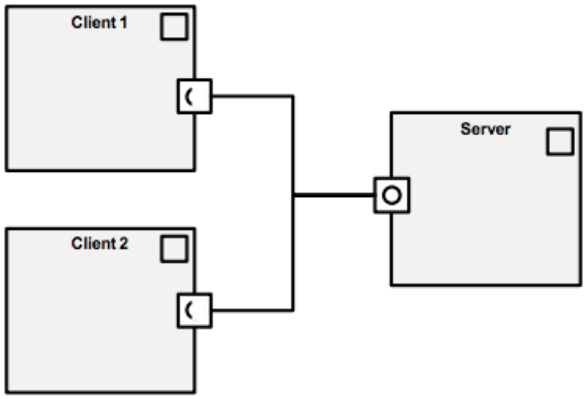

Figure 2.14: *Example of client/server interface*

2.1.5.2 Runnables

The functional implementation of an SWC is known as a "runnable entity", or simply a runnable. An SWC may contain a single runnable, as well as a larger number of runnables. A runnable contains a sequence of instructions in a coding language of choice.[15] The activation of a runnable is done by setting an event, and there are several different types of these events. If the runnable is invoked by a *timingEvent*, the runnable will be activated periodically with a given time period. A runnable can also be invoked by a *dataReceivedEvent*, which means that the runnable will be activated when data has been received on a given receiver port of the SWC. As described in the previous section, the runnable can also be invoked by a Client/Server interface, in this case the server port will produce an *operationInvokedEvent* which will activate a runnable set to a client port. These are only a handful of the events available in AUTOSAR.[14]

2.1.6 Runtime Environment

In order for an SWC in the application layer to communicate with other SWCs and BSW modules, a common interface for the Application Layer and the Basic Software Layer is required. The RTE was conceived for this purpose.[21] An example of how the RTE is used with the other modules is presented in Fig. 2.15 below.

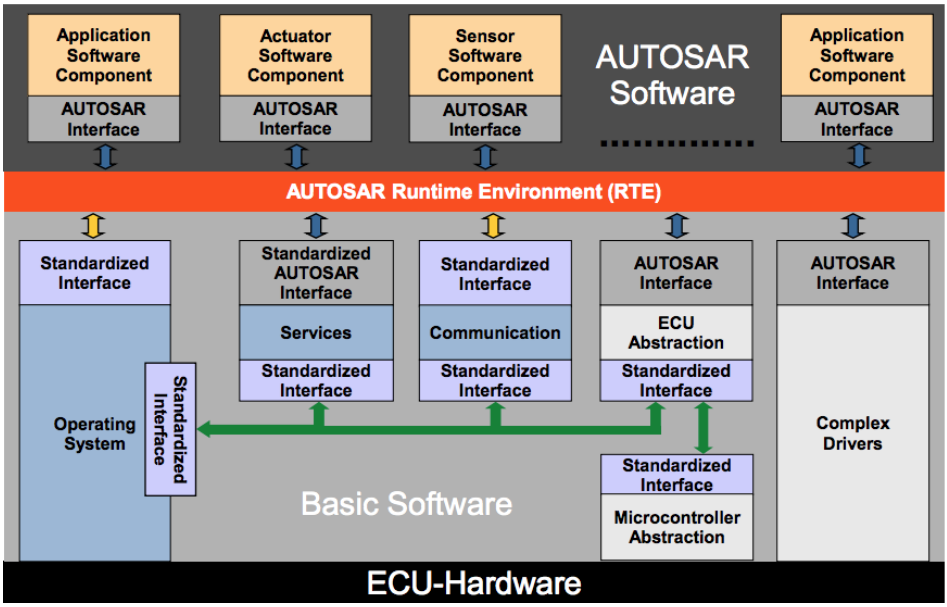

Figure 2.15: *Runtime Environment interaction with other modules*

In the Fig. 2.15 above, it is shown how the RTE is connected to different modules. It is implemented in the form of a bus, so each SWC can access any software module in the top layer of the BSW layer. This also enables the communication between the SWCs. In order to implement the RTE, a tool called the "RTE Generator" is used. This is a configuration tool which generates code corresponding to configurations set by the user. This configuration step involves mapping SWC runnables to OS tasks and events. The code generated includes functions used by the SWC runnables to call functions of modules in the BSW layer. To illustrate this, a code example is shown below.

```
uint32 message = 0xFFFF;

void Runnable {
    Rte_IWrite_Runnable_SenderPort_data(message);
}
```

The runnable in the code example uses a function generated by the RTE to write a hexadecimal value 0xFFFF to a port of the SWC. This port is affiliated with an internal signal specified in the communication stack. For example, if the signal is specified to be connected to the CAN interface, this function will send out a CAN message on the CAN bus

2.2 Controller Area Network

The controller area network is a communication protocol for serial transfer of data. It has been used ever since its introduction 1985. Today with the increasing amount of electronic devices in cars, one of the biggest concerns is establishing security and to be able to as easily as possible integrate electronics into the car. Today when integrating electronic circuits into the car, the CAN Network becomes attractive as a solution to minimize the wiring required for the devices to be able to function correctly. This saves a lot of space, and makes it cost-effective.[16]

CAN is divided into three different layers, the object layer, transfer layer and the physical layer which will be explained a little more thoroughly below.[16]

2.2.1 Layers

2.2.1.1 Object layer

The CAN object layer's task is to decide which messages are to be transmitted as well as which messages received are to be used. This is also called message filtering. [16]

2.2.1.2 Transfer layer

The transfer layer's task is to perform arbitration to check priority of messages, as well as the fault confinement with the three different states of a node (Active, Passive or bus off). Moreover it also controls error detection and message validation. Furthermore the transfer layer also controls the transfer rate and the bit timing settings. It also checks if the CAN bus is free to start a new transmission of a message from the object layer or whether a reception of a message to the object layer has just been started.[16]

2.2.1.3 Physical layer

The physical layer is defined as how the communication between two devices or more should be handled with regard to the physical medium used as connection between these devices. [23]

2.2.2 CAN Messages

There are four different types of CAN messages: remote frames, data frames, error frames and overload frames. These frames will be described below.[16]

2.2.2.1 Data frame

The data frame consists of seven different fields with the objective to carry data from a transmitter to a receiver. The first field of a data frame is named "start of frame" which contains a single logical bit set to '0' (dominant bit). This indicates

that either a new data frame or a new remote frame is on the bus.[16] Fig. 2.16 below shows the fields of the Data frame.

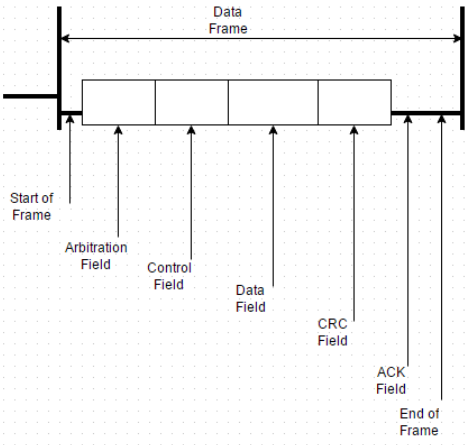

Figure 2.16: *Fields within the Data frame*

Next, the data frame consists of an arbitration field, which is made up of an identifier field as well as an RTR-bit. The identifier field in the arbitration field serves as either an 11 or 29 bit identifier for the content of the message. This is known as either a standard identifier or an extended identifier, respectively. Commonly, the standard identifier format is used. The identifier is then used by the nodes in the network to decide if they should use the information or not. The identifier also contains the necessary information on how to prioritize messages which are simultaneously sent out on the CAN bus. This is done by comparing the messages bitwise and the one with the lowest identifier gains arbitration. The RTR-bit is set to dominant to indicate that it's a data frame, if it's set to '1' (recessive) it instead indicates that it's a remote frame. [16] A Fig. 2.17 illustrating the arbitration field is shown below.

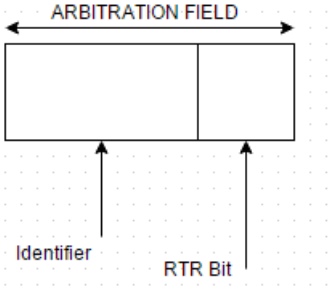

Figure 2.17: *Illustration of the arbitration field*

After the arbitration field, the data frame consists of a control field which is made up of six bits. Two reserved bits are set to dominant. Thereafter there are

four bits regarding the length in bytes of the data to be transmitted, 0 up to 8 bytes. [16] Fig. 2.18 located below is illustrating the control field.

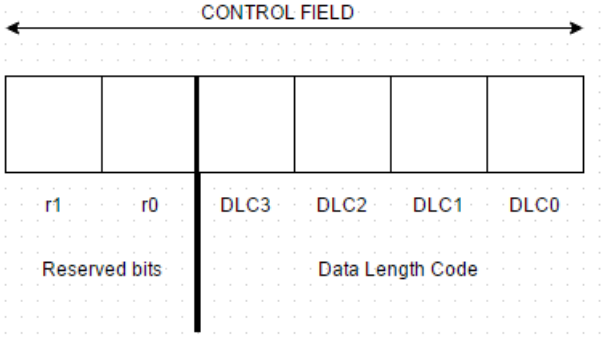

Figure 2.18: *Illustration of the control field*

Thereafter the data frame consists of a field called data field. It is made up of 8 bits, with the eight bit being the most significant, bit and can contain up to 64 bytes of data. [16]

Further on we have the CRC field which consists of a CRC sequence as well as a CRC delimiter. The CRC sequence is used for error detection. It performs a CRC on the data sent or received and compares the received CRC-value with an expected value. If these values differ, the CRC has detected a data error and will either reread the data or request a new transmission. The last bit of the CRC field is called the CRC delimiter, which is a single recessive bit. [16] Fig. 2.19 below depicts the CRC field.

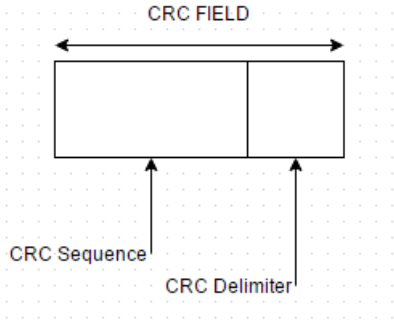

Figure 2.19: *Illustration of the CRC field*

Thereafter the data frame is made up of an ACK field which consists of two bits. The ACK slot can either be dominant or recessive. If it's a node transmitting data, the ACK slot will consist of a recessive bit. When a node receives the sent frame, the receiver verifies the integrity of the frame and sends a dominant bit during the ACK part of the frame if it was received correctly. The last bit of the

ACK field is called the ACK delimiter, which is a single recessive bit. [16] Fig. 2.20 below illustrates the ACK field.

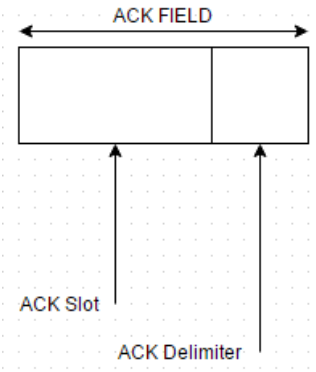

Figure 2.20: *Illustration of the ACK field*

Finally, the data frame consists of a field named "end of frame" which is made up of seven recessive bits. This will indicate the end of the data frame.[16]

2.2.2.2 Remote frame

The remote frame is, in contrast to the data frame, only composed of six fields, and is lacking the data field which can be found in the data frame. A remote frame is sent by a receiver node when it requires data from a transmitter node. The RTR-bit in the arbitration field will decide whether the frame is of the type "data frame" or of the type "remote frame". If the RTR-bit is set to recessive, the frame is of the type remote frame and vice versa. [16]

2.2.2.3 Error frame

The error frame is made up of two fields, the ERROR flag field and the ERROR delimiter field. The ERROR flag can be of two types, namely an active ERROR flag or a passive ERROR flag. The difference between these flags is that with the active flag case the information sent is six successive dominant bits, and for the passive case six successive recessive bits. The nodes connected to the bus can be in three different states, error active, error passive, or bus off. The state of the node is depending on their internal receive and transmit error counters. Moreover, the ERROR delimiter field is made up of eight succeeding recessive bits succeeding the error flag used. This will allow the bus to return to its normal mode. [16]

2.2.2.4 Overload frame

The overload frame is composed of two fields, the overload flag field and the overload delimiter. Basically, the overload frame acts as a delay for the next data or remote frame. [16]

3

Method

3.1 AUTOSAR Methodology

As well as specifications for system architecture, AUTOSAR also specifies a methodology, which will be used in this project. This can be seen as a guiding framework for implementing a system based on the AUTOSAR architecture. A basic picture of this methodology in the form of a workflow is shown Fig. 3.1 below. [17]

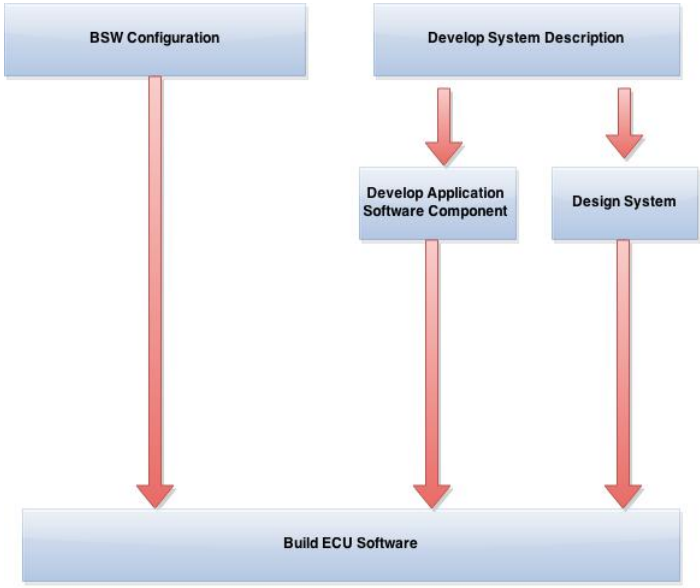

Figure 3.1: *AUTOSAR methodology workflow*

A more detailed description of each phase is explained below.

3.1.1 BSW configuration

In this phase, the BSW modules are configured with a code generation tool. In our case this is done with the Arctic Studio BSW Editor. This phase includes, among other things:

- Setting configuration parameter values for different modules
- Describing the system signals which are relevant for the system
- Creating OS Tasks & Events

Since this step includes the BSW modules in the MCAL layer, these configuration parameters need to be set in regard to which MCU hardware is being used. [17]

3.1.2 Develop System Description

In this phase, the outline for the SWCs of the system is created. There are several steps of this development phase. In this thesis, the following will be executed:

- Data Model Development
- Component Model Development

First, the Data Model Development phase is executed. During this phase, data types and interfaces are defined. Data types describe a kind of data, for example uint8 or uint32. Interfaces describe which of these data types are to be transmitted between different ports in the system. When this is completed, the Component Model Development step can be initiated. This is done by defining one or several SWCs, creating an SWC description for each component. In practice, this means specifying which ports each component should feature, and linking these ports with interfaces. The component type is also defined here. The last step of the component model development is to define a top-level-composition, which includes instantiations and internal connections between the components. [8]

3.1.3 Design System

What is actually produced in this phase can vary depending on the situation. For example, if it is desired to have several sub-systems, a system extract is generated and this is through further activities divided into several Ecu Extracts. An Ecu Extract contains a description of SWCs which is specific for a single Ecu.[8]In this thesis, only one system and one Ecu is needed, thus the Ecu Extract is generated already in this phase. The Ecu Extract in this thesis contains the following information:

- Assignment of top-level-composition

- Mapping implementation to SWC instances included in the composition
- Defining system signals, and mapping these to the outer ports of the composition

3.1.4 Develop Application Software Components

In this step, the actual implementation of each software component is done. This means defining runnables for each SWC, and also defining the event that is to be set for each runnable to be activated. This step also includes writing the actual software for the component, providing it with the desired functionality.[17]

3.1.5 Build ECU Software

This phase is where basically all comes together. The generated code as well as the manually implemented code is compiled in order to create an executable file which later can be programmed on a MCU. In order to do this, the SWCs need to be mapped to the rest of the system through the RTE. This is done by importing the ECU extract into the RTE generator and mapping SWC runnables to tasks and events.

3.1.6 Modeling approach

For AUTOSAR SWC components, there are mainly two different approaches of SWC modeling, graphical or text based. Using the graphical approach, SWCs are created in a graphical development environment with a "click and drag" type of approach. In the text based approach, modeling is done using a text-based modeling language called ARtext. An example for how this can be used is shown in the code below.

```
interface  senderReceiver MySRInterface {
    data uint8

component application MyComponent {
    receiver  MyReceiverPort requires MySRInterface
    sender My SenderPort provides MySRInterface
    }

internalBehaviour MyInternalBehavior for MyComponent{
    runnable Runnable A[1 0] {
        timingEvent [1.0]
    }
}

implementation MyImplementation for MyInternalBehavior {
    language c
    codeDescriptor "src"
}
```

This example shows how an SWC with a sender and a receiver port can be modeled using ARtext language. First, the interface is defined, in this case an eight bit integer. Then the application is defined, with a sender/receiver port specified to *provide* and *require* the interface with data, using the sender and receiver port respectively. Next, a runnable is defined in the internal behavior, and it is set to be activated on a *timingEvent*, meaning it will be activated periodically on a given time period. Lastly, the implementation is described, specifying that the implementation of the SWC will be written in C language. The implementation code itself will be linked together with the SWC during the RTE generation, which will generate a function call which will, in this case, call the function periodically.

3.2 Development Tools

3.2.1 Arctic Studio

Arctic Studio from the developer ArcCore, is a development environment based on the IDE Eclipse. The purpose of Arctic Studio is the development of SWCs, as well as the configuration of the BSW layer and the RTE layer. It is shipped with the embedded platform Arctic Core which provides actual implementation of the AUTOSAR BSW modules. The software components are designed in a text editor using the ARtext languages SYSD and SWCD. The task of the BSW editor is to configure all of the modules used within the project. The BSW editor is depicted in Fig. 3.2 below.[4]

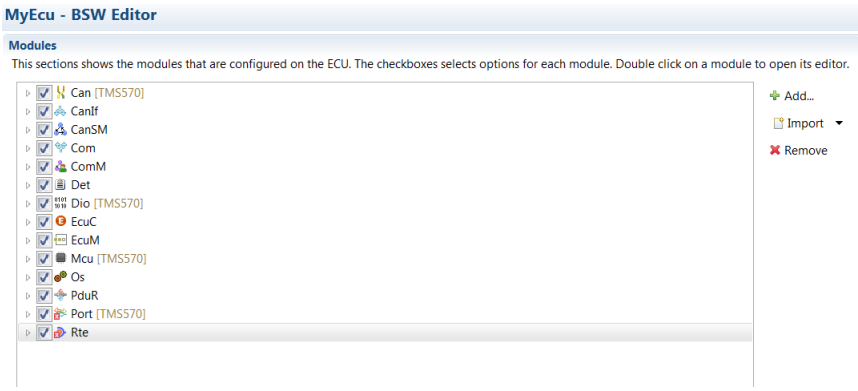

Figure 3.2: *BSW Editor in Arctic Studio*

3.2.2 HALCoGen

HALCoGen by Texas Instruments is a code generation tool intended for usage with the Hercules family of MCUs. We decided to make use of this program since it has a very simple code generating approach. Thus, we can easily verify the functionality of the development board. The program consists of a graphical user interface which enables the user to configure for example peripherals, such

as the DIO ports on the MCU. When the configuration part is done the user can then use HALCoGen to generate the drivers of the configured hardware components. When all necessary drivers for the hardware have been generated the user can then import the drivers to a development environment of choice, for example Code Composer Studio (CCS). This makes it possible for the user to use the hardware components by invoking certain functions.[20] Fig. 3.3 below illustrates a configuration window in HALCoGen.

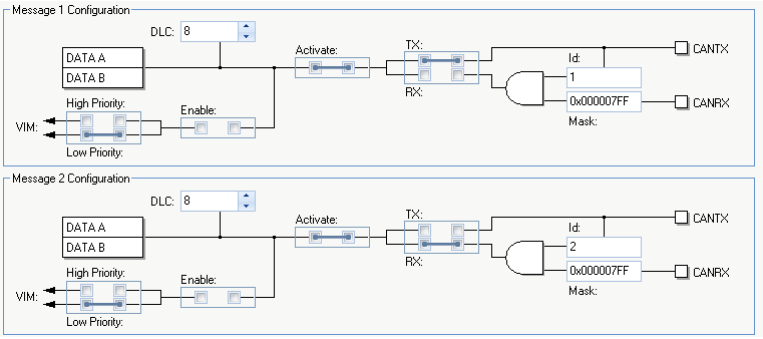

Figure 3.3: *Example of configuration window in HALCoGen*

3.2.3 Code Composer Studio

As mentioned above the program by the name of Code Composer Studio is a development environment for Texas Instrument development boards and is also compatible with the generator tool HALCoGen. One of many features with CCS is the debugger, which can be of great importance when troubleshooting your projects for errors, as well as providing knowledge about the structure of the program debugged, for example the order of initializations. [19]

3.2.4 PyQtGraph

PyQtGraph is a library for Python providing graphic and user interface functionality. This library includes functions such as plotting several curves at once with different colors, and to be able to adjust the size of the plot window. Moreover it also includes functions such as putting labels and units on the x-axis and the y-axis, as well as adjusting the range of the x-axis/y-axis. Furthermore it makes use of the QtGui platform via PyQt, which makes it easy to apply once one knows how to use PyQt. [1]

3.2.5 PyQt

PyQt is a Python Version of the Qt framework for C++. PyQt is a package with a huge variety of functionality. Some features that have been necessary for our project have been the graphical user interface part as well as the threading. [2]

3.3 Project Method

In order to reach the goals we set out to achieve with this thesis, we realized it was first necessary to achieve a greater understanding regarding the concept about AUTOSAR, as well as getting more hands-on experience with the development tools that were to be used before we started our actual implementation. In order to achieve this, we decided to conduct a combined theoretical and practical pre-study. The concept here is to study AUTOSAR theory while also doing some example projects. In this way, we would get enough understanding on the AUTOSAR concept and the development tools in order to do our project. The basic project outline is shown in Fig. 3.4 below.

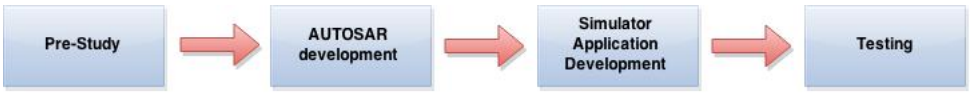

Figure 3.4: *Project Workflow*

At the point where we felt we had enough understanding of the AUTOSAR concepts and our development tools, we started the AUTOSAR development phase, which corresponds to the AUTOSAR methodology described in section 3.1. When we and our supervisor at ÅF were satisfied with the implemented CAN Communication stack, the next phase was to design the Can Simulator. Concurrently, there is also a testing phase, where simulation results are obtained for the purpose of usage in the results section. It should be noted that testing is done consistently in all steps to verify functionality, the last step simply refers to producing simulation results for the report.

3.3.1 Pre-Study

For the sake of achieving a CAN network with 36 different ID's with the data length of 5 bytes it was of great importance to understand how the modules in the AUTOSAR system interact with the processor as well as each other. We had two different approaches of trying to achieve the stated functionality, our first approach was that we created a simple program at the top of the hierarchy within AUTOSAR, the application layer. However when trying to verify the functionality of this simple program, we realized that it was very difficult to know where in the chain of hierarchy the program was at fault. That realization made us approach the problem at the bottom level of hierarchy where we were as close to the processor as possible to verify functionality of the program.

We decided to design a simple program which would light a LED on the development board. This was first done using HALCoGen to generate drivers for the Port and DIO module.

The program was based on an example included in the Arctic Studio program called OsSimple. OsSimple consisted of a configured OS module as well as the modules necessary for the MCU such as the ECUM as well as the AUTOSAR MCU

module. The example had a couple of tasks configured in the OS module, which were triggered by alarms at different time cycles. By combining this example with the generated code from HALCoGen, we were then able to light some LEDs, in the example shown below the code to light a LED is displayed.

```
gioInit ();

gioSetDirection(hetPORT,0xFFFFFFFF);

gioSetBit(hetPORT,1,1);
```

First of all, the necessary registers for the Port and Gio hardware modules are initialized by the function gioInit(). Later on the direction of the entire hetPort is specified to output, and at last a specific pin of the hetPort (pin1) is selected and the state of the pin is set to a logical '1'.

This verification of the LEDs on the development board made us strive for the same functionality but with a program completely programmed and generated by just using Arctic Studio. We then configured the Dio and Port AUTOSAR modules in the BSW editor and therefore reached the same functionality. In the example below this is illustrated.

```
Dio_WriteChannel(DioConf_DioChannel_N2HET1, 1);
```

With this approach we were able to implement these two modules and contain the same functionality with the AUTOSAR approach as if we were to program directly on the microcontroller.

One of the biggest parts with this thesis was the functionality of the CAN network. For the sake of achieving this functionality we had to grasp the concept behind CAN and the structure of a CAN message. First and foremost we decided to research the theory behind CAN communication. With the theory behind CAN backing us up we decided to try this new knowledge out. To establish a CAN connection between a PC and the development board, we used a USB CAN adapter called the Kvaser Leaflight 2.0.

We proceeded with our earlier way of approach, by generating the drivers in HALCoGen for the CAN network to function properly. Simple messages were sent and properly received by using two python functions which could transmit and display CAN traffic on the PC. In the example below, the transmitting/receiving of data on CAN channel1 is handled by two different message boxes, with one handling messages with the CAN ID 1 and the other handling messages with the CAN ID 2.

```
canTransmit(canREG1,canMESSAGEBOX1,tx_data)

while(!canIsRxMessageArrived(canREG1, canMESSAGE_BOX2));
        canGetData(canREG1, canMESSAGE_BOX2, rx_data);
```

3.3.2 AUTOSAR Development

This was the point where we were to implement our actual application, which purposes and requirements are defined in section 1.3 in this report. According to the AUTOSAR methodology the initial phase can either be "BSW Configuration" or "Develop System Description", since these are independent of each other. In our case it made more sense to start with the "BSW Configuration Phase", since this was by far the largest aspect of the project.

3.3.3 BSW Configuration

Since modules are dependent on each other depending on position in the hierarchy, this needs to be considered when configuring BSW modules. There is a pretty good example of this is in Fig. 2.4 in section 2.1.4. If it was desired, for example, to configure the modules to provide CAN communication to the system, the Can Driver would be needed to be configured first, as this provides the drivers for the MCU. Next, the CAN Interface would be needed to be configured in order to provide means of communication for upper modules. Going up in the hierarchy, the PDU router would need to be configured next, followed by the COM Modules. Following hierarchy is key in deciding the order of BSW configuration. There is, however, one module that is always needed to be configured first in order for the ECU to actually do anything at all, this is the ECUM Module, since it handles the initialization process of the processor, as well as handling sleep and wake-up modes. In the following sections, it is described how we configured each module individually.

3.3.3.1 Ecu Manager

The ECU Manager didn't, in our case, require that much thought. In the scope of our thesis it wasn't required for any specific sleep or wake-up conditions, so in this case we just went with the standard settings.

3.3.3.2 Operating System

When configuring the OS, there are mainly three things to consider, namely:

- Alarms
- Tasks
- Events

The first task to be implemented was MainTask, which is shown below.

```
void MainTask(void) {
    // Initialization function that needs to be called manually to initialize some modules
    EcuM_StartupTwo();
        //Function for setting the MCU's CAN ports to the correct mode of operation
        init_canports ();
    Com_IpduGroupVector groupVector;
    // Start the IPDU group
    Com_ClearIpduGroupVector(groupVector);
```

```
    Com_SetIpduGroup(groupVector, ComConf_ComIPduGroup_CANIPDUs, TRUE);
    Com_IpduGroupControl(groupVector, FALSE);

    // Activate  Can InterFace
    CanIf_SetControllerMode(CanIfConf_CanIfCtrlCfg_CanIfCtrlCfg, CANIF_CS_STARTED);
        CanIf_SetPduMode(CanIfConf_CanIfCtrlCfg_CanIfCtrlCfg, CANIF_SET_ONLINE);
        Can_MainFunction_Mode();

}
```

This task is used as an initialization task. The first function called here is the EcuM_StartupTwo(), which does some additional initialization to those that are done by default. In our case, the initializations sought after from this phase are the initialization functions of the CAN modules. The init_canports() is a function we created in order to configure the CAN controller correctly, this was needed since the AUTOSAR Port Driver for the MCU in this project was incomplete and lacked the ability to properly set up the CAN_RX and CAN_TX pins of the microcontroller. This is more thoroughly explained in the "Port" section 3.3.3.3. Next, the I-PDU Group for our internal signals is activated. Successively, the Can Interface is activated. Next, the function Can_MainFunction_Mode() is activated, which sets the Can Driver to the correct mode.

There is also one more task in the system, namely the RteTask, which is shown below.

```
void RteTask(void) { /** @req SWS_Rte_02251 */
    EventMaskType Event;
    do {
        SYS_CALL_WaitEvent(EVENT_MASK_DataReceivedEvent | EVENT_MASK_DataReceivedEvent2);
        SYS_CALL_GetEvent(TASK_ID_RteTask, &Event);

        if  (Event & EVENT_MASK_DataReceivedEvent) {
            SYS_CALL_ClearEvent (EVENT_MASK_DataReceivedEvent);
            Rte_CanReader_1_CanReaderRunnable();
        }

    } while (RTE_EXTENDED_TASK_LOOP_CONDITION);
}
```

This task is declared in the OS module to set activation conditions. The purpose of this task is simply to activate the runnables of the SWCs, on a given type of event. In contrast to the MainTask, which was implemented by us, the code for RteTask is generated by the RTE Editor, which will be described in section 3.3.7.1.

3.3.3.3 Port Driver

The purpose of the Port Driver in our system is to initialize port directions and port modes. In our case, we needed to set up the following ports:

- CAN_RX
- CAN_TX
- GIO

Initialization of the CAN ports was obviously needed for the CAN communication to work. We also decided to configure the GIO port. The GIO port is basically a general purpose port where you can for example set and read bits. On the development card we were using, some of the GIO pins were connected to LEDs on the card. It was very useful for us to be able to write to the LEDs for troubleshooting and verification purposes. The port configuration is supposed to be quite simple in Arctic Studio. Instead of configuring an entire port, you configure each pin individually. You simply choose a pin from a list and then configure it as you desire. The different parameters are shown in Fig. 3.5 below.

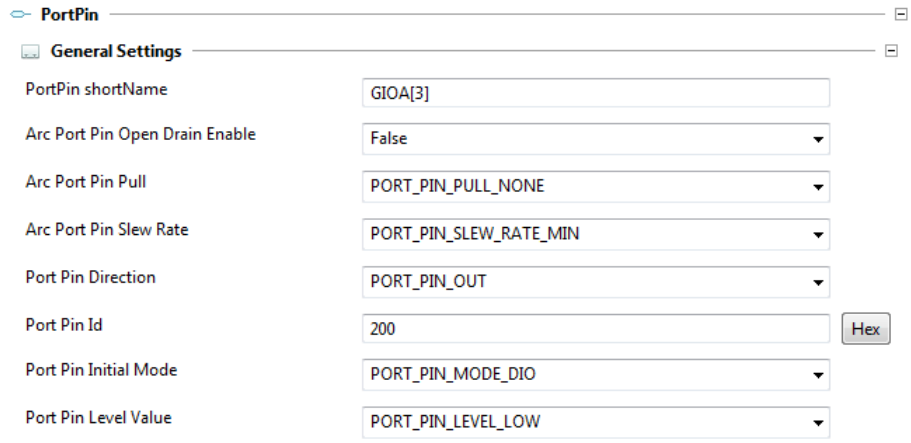

Figure 3.5: *Port configuration in Arctic Core BSW Editor*

This example shows how to configure a simple out pin which can be set to high or low. The pull is set to none, the slew rate is set to minimum (for quicker level changes), the direction is set to out, and the port also gets a symbolic Pin ID. The mode is set to DIO for digital in/out and the initial level value is set to low. What actually corresponds to what pin is being initialized on the MCU is the PortPin shortName. This name will appear when the pin is chosen in the list, so as long as you don't change the shortName, it should be working.

However, in our case, the port configuration turned out to be quite troublesome. The MCAL available in the version of Arctic Core we were using was made for the TMS570LS12 MCU, while our development board featured the newer TMS570LS20. The way port initialization is done is somewhat different on the newer version, so they are not entirely compatible. On the TMS570LS12, two registers called KICK0 and KICK1 need to be written to in order to change the port configuration register, so the port initialization function in Arctic Core writes to those before writing to the port configuration register. However, the TMS570LS20 doesn't feature these registers, which resulted in the port initialization function trying to write to non-existing registers, which resulted in the MCU freezing. The problem was a bit difficult to locate, but once found it was easily solved. The instructions written to the non-existing registers simply needed to

be removed. There was also one more issue with the port driver configuration. When choosing pins in the list, the CAN_TX and CAN_RX pins did not exist. After some e-mail correspondence with the ArcCore staff, it was revealed that the TMS570 port driver was actually incomplete. To solve this, we created a function which wrote to the necessary registers to set up the CAN pins, the code is shown below.

```
void init_canports(){

        canREG1->CTL = 0x00000000U | 0x00000000U | 0x00021443U;
        /** - Setup TX pin to functional output */
    canREG1->TIOC = 0x0000004CU;
    /** - Setup RX pin to functional input */
    canREG1->RIOC = 0x00000048U;
    canREG1->CTL &= ~0x00000041U;
return;
}
```

canREG1 is a collection of registers which are related to the CAN module of the MCU. The CTL register enables modifications to other registers in canREG1, and the TIOC and RIOC enables the CAN_TX and the CAN_RX pins, respectively.

3.3.3.4 EcuC

The purpose of the EcuC module is to provide the communication stack with PDU objects. When configuring the EcuC module, it is necessary to know how many different CAN messages there are to be distributed, as every CAN message needs a corresponding PDU to carry the data in both directions throughout the communication stack. The configurations are basically the creation of one PDU for each message as well as specifying the DLC. These configurations are shown below in Fig. 3.6.

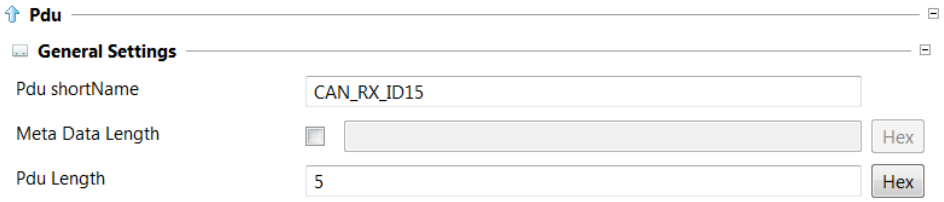

Figure 3.6: *Example of PDU object configuration in EcuC*

3.3.3.5 CAN Driver

At the bottom of the CAN hierarchy is the CAN driver module. The module itself doesn't require that much configuration, and the only thing needed to be configured is the configuration of the Can Hardware Objects. The configuration of one of these is shown below in Fig. 3.7.

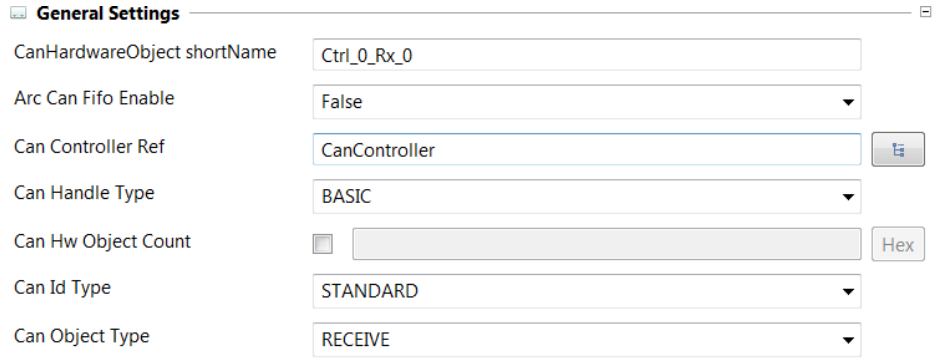

Figure 3.7: *Configuration of CAN Hardware Object*

3.3.3.6 CAN Interface

Next in order in the CAN communication hierarchy is the CAN interface module. The purpose of the CANIf module in our system is to provide services for transmission and reception of messages. In our case the configurations that needed to be done except for the configurations given when adding the module was the creation and configuration of CANIfPDUs as well as referring the CAN hardware object handle in the CANIf to the earlier configured CAN hardware objects. The configuration of the CANIfPDUs are especially important when setting up the system, considering they contain the CAN ID as well as the data length of the data that is to be transmitted, this is shown in the Fig. 3.8 below.

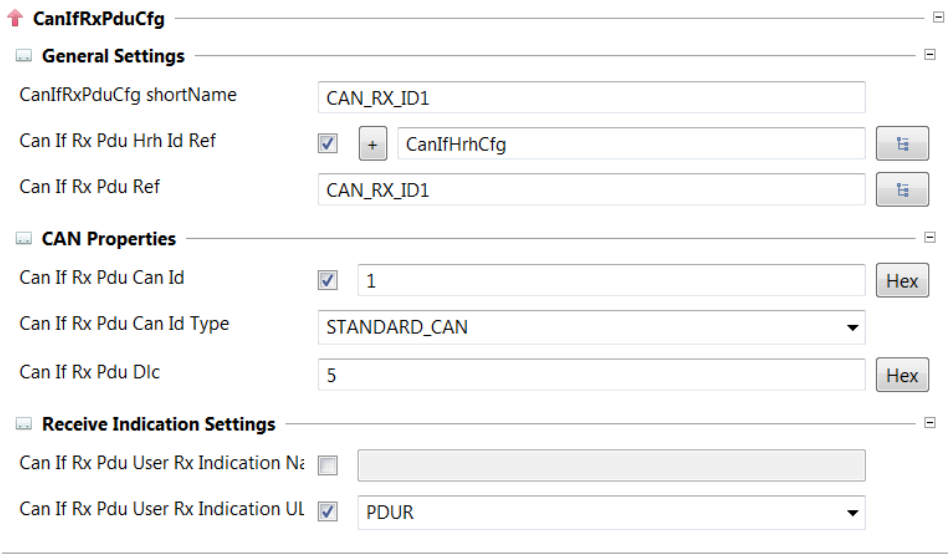

Figure 3.8: *Configuration of a CANIfPU*

3.3.3.7 PDU Router

The purpose of the PDUR module is to route I-PDUs through the CAN communication stack within AUTOSAR. This has been configured in the routing table where we have specified which modules are to be used within the communication hierarchy, in our case it consists of the COM module as well as the CANIf module. The routing table is shown below, in Fig. 3.9 and in Fig. 3.10.

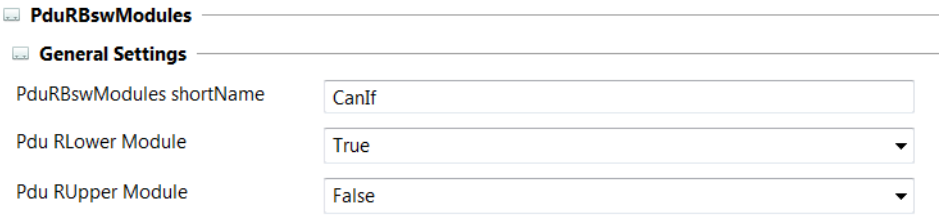

Figure 3.9: *Lower module, CAN Interface*

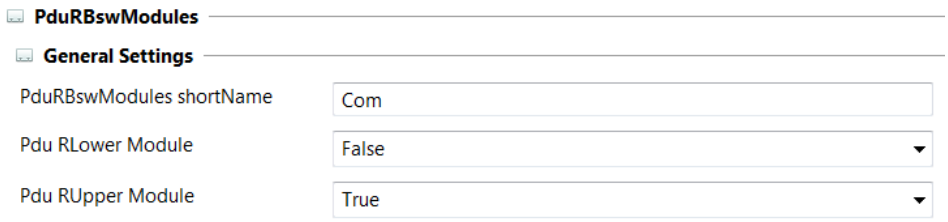

Figure 3.10: *Upper module, COM*

Moreover, perhaps the most important part when configuring the PDUR module is the configuration of the routing of individual I-PDUs, where it is of grave importance to configure which direction the specified I-PDU is heading in the CAN communication stack. These configurations depends on the individual selected I-PDU if it is either a transmitting I-PDU or a receiving I-PDU. Below is Fig. 3.11 and Fig. 3.12, showing the parameters that are needed to configure the routing of the I-PDUs.

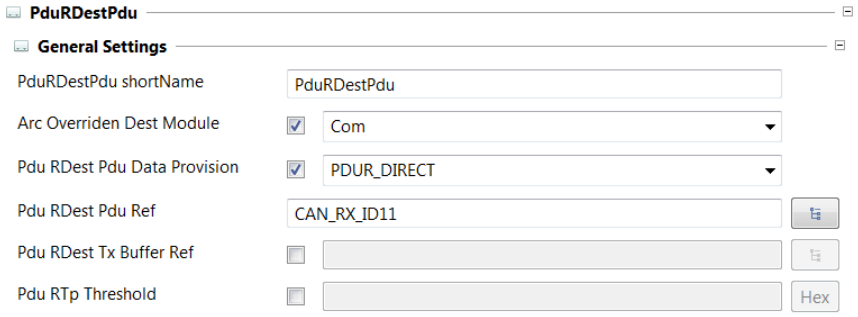

Figure 3.11: *Destination of the I-PDU*

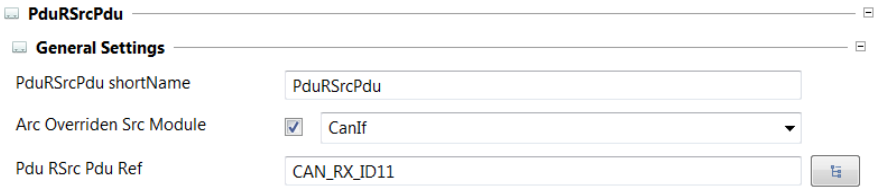

Figure 3.12: *Source of the I-PDU*

Fig. 3.11 and Fig. 3.12 show an example of the PDU routing of a receiving I-PDU. Fig. 3.11 shows the destination of where the I-PDU is headed. As for Fig. 3.12, it shows the source of the I-PDU. As for the transmitting PDUs, the destination module as well as the source module is the opposite to the receive PDU.

3.3.3.8 COM

The purpose of the COM configuration is to configure the communication of the system. This is done mainly by defining four different types of objects, namely:

- I-PDU Groups
- I-PDUs
- Signal Groups
- Signals

An I-PDU contains the data message that has been either received from one of the communication modules in the communication stack, or a data message that is to be sent to one of said modules. An I-PDU also belongs to an I-PDU group, in our case we used the same for all I-PDU's as there isn't a large amount I-PDU's in our system. The data of an I-PDU is divided into signals, depending on bit position in the I-PDU. For example if you have an I-PDU with 2 bytes of data length and desire to place the first byte to one signal and second byte to another signal, you set the bit position of each signal accordingly. Signals can be divided into signal groups as well, which is needed if it is required to send several signals to the same I-PDU simultaneously. In our system, we need to be able to send and receive 5 bytes of data in each CAN message. Since COM signals are limited to 4 bytes of data, we created a signal group for each CAN message, where one signal contains the 4 least significant bytes, and another signal contains the most significant byte. An example of this can be seen in Fig. 3.13 below.

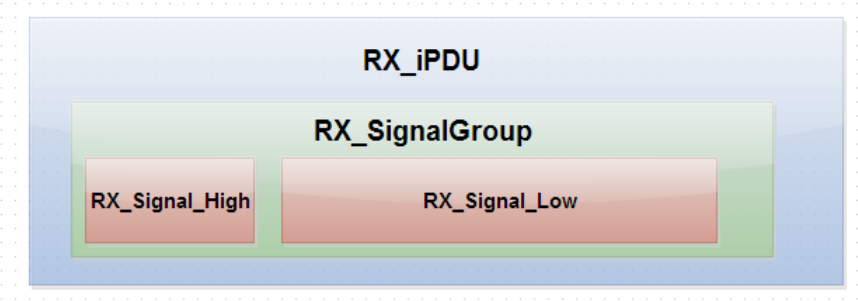

Figure 3.13: *Example of I-PDU structure*

Each of these COM objects need to be configured after they are created. An example how a receiving I-PDU was created is shown in Fig. 3.14 below.

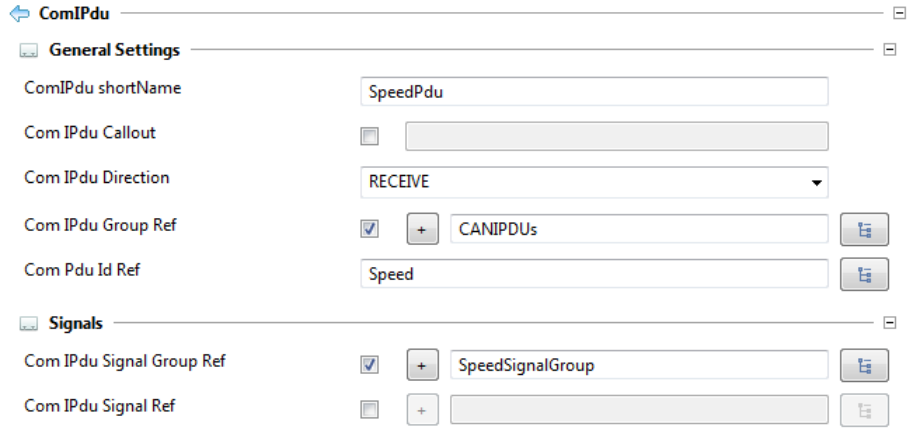

Figure 3.14: *Example of receiving I-PDU configuration*

Since this I-PDU is supposed to receive data, the direction is set to "RECEIVE". It also needs to refer to the I-PDU group, the corresponding PDU object in the EcuC PDU collection, and the signal group that it is supposed to contain. If it was to contain just a signal in contrast to a signal group, it would require a signal reference instead. The next object to be configured is the signal group. An example on how to configure a signal group is presented in Fig. 3.15 below.

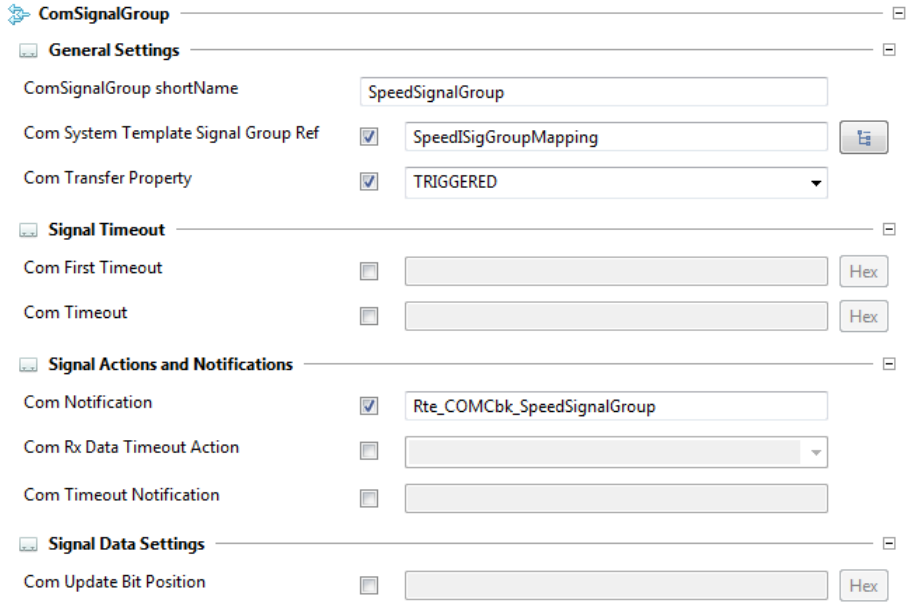

Figure 3.15: *Example of COM Signal Group configuration*

The signal group needs to be mapped to the port mapping of an SWC, this is what is referred to in Com System Template Signal Group Ref. How this is described in the SWC itself will be described in section 3.3.6.1. The transfer property is set to TRIGGERED. This means that the signal group is sent to its corresponding SWC port when new data has arrived. This is very useful for an RX signal, since there's then no need to consistently check for new data. Com Notification specifies a callback function that is activated when data is received. This callback function sets an event, which is waited upon in the RteTask in order to activate a runnable. Check example code for RteTask in section 3.3.3.2 for details. The last object to be configured in this module is the signal, which is shown in Fig. 3.16 below.

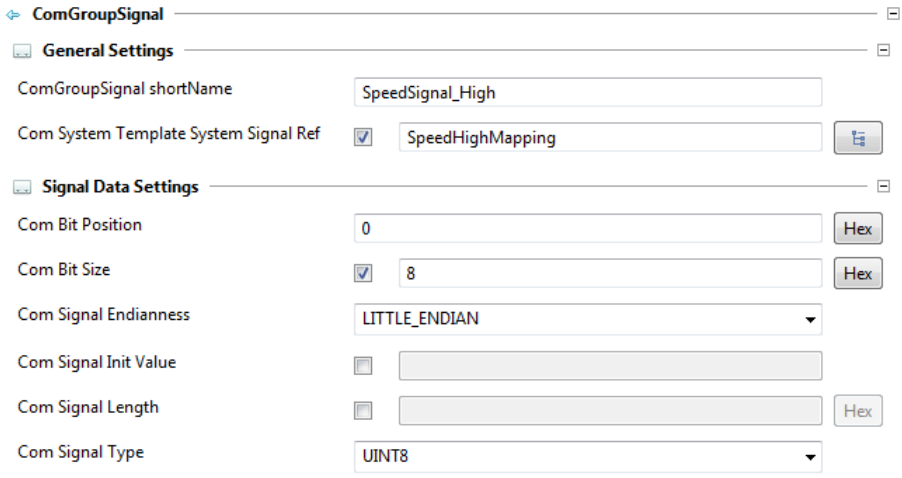

Figure 3.16: *Example of COM Signal configuration*

Similarly to the signal group, the signal itself also needs to be mapped to a port mapping of the SWC. As described before, the bit position and the bit size describes which part of the PDU is to be included in the signal. In this case it will be the highest byte of the "Speed" PDU, since the bit position is set to 0, and the size is set to 8. The signal type also needs to be defined, in this case the uint8 format is suitable to the signal's length.

3.3.3.9 Importing Communication Matrices

To create signals in Arctic Studio is time-consuming, as each signal requires many parameters set in three different modules. To solve this, there is a function in Arctic Studio to import a communication matrix from the ECU extract, which automatically adds objects in the COM, PDUR, and the CANIf modules. This would have been very useful for us, since we during the testing phase of the project method tested running the system with a large amount of signals. Unfortunately,

we could not get this feature to work, but if done correctly this would have been very useful.

3.3.4 Develop System Description

As explained before in section 3.1.2, this phase is divided into the Data Model Development and Component Model Development. These are described in the following sections.

3.3.4.1 Data Model Development

During the Data Model Development phase, there are two things to consider, defining types and interfaces. The code below shows how this was done in our system.

```
int impl Uint32 extends uint32
int impl Uint8 extends uint8

record impl CanDataFields {
        Uint8 High,
        Uint32 Low
}
interface senderReceiver SRInterface1 {
        data CanDataFields RxData
        data CanDataFields TxData
}

interface clientServer CSInterface1 {
        operation Send{
        in CanDataFields data1
        }
}
```

As was described in the previous section 3.3.3.8, the data of the CAN messages is divided into two signals, one with the most significant byte, and one with the four least significant bytes. This needs to be considered when defining the interface, as these are the very signals that are to be sent and received. Thus, a new data type is created, the CanDataFields. It is declared using the record command, which works exactly like a struct in C++. The CanDataFields includes one uint8 for the high part of the signal, and a uint32 for the lower part of the signal. This way, the interface corresponds to the signals defined and configured in the COM module. An interface is also created for the client/server ports, defining an operation called Send with a CanDataFields as an input parameter.

3.3.4.2 Component Model Development

The first step of obtaining the component models is to define the functionality of the system. In this thesis, the functionality is pretty basic, as these are the only functions required by the system:

- Send CAN messages
- Receive CAN messages

To achieve this functionality, we created one SWC for receiving messages and another SWC for sending messages. These are called CanReader and CanWriter, respectively. Fig. 3.17 and Fig. 3.18 models the SWCs.

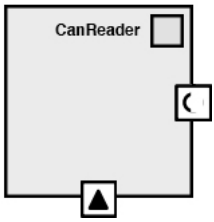

Figure 3.17: *CanReader*

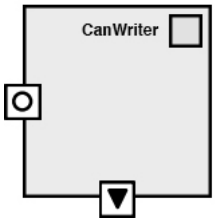

Figure 3.18: *CanWriter*

The CanReader features a receiver port, and a client port. The receiver port will later be used to obtain data, and the client port will be used to call the runnable in CanWriter. The CanWriter features a sender port and a server port. The sender port will later be used to write data, and the server port is needed in order to recognize the function call from CanReader. Since we are using Arctic Studio, which uses a text based modeling approach, we need to write some AR-text SWCD code to obtain the system description for these components. This is shown in Fig. 3.19 below.

```
component application CanReader {
        ports {
                client  CanClient requires CSInterface1
                receiver  CanReceiverPort requires SRInterface1
                }
```

Here, the component is declared as an application, which is a general-purpose type of SWC. The ports are connected to the interfaces previously defined. CanWriter is defined in the same way, but with the corresponding ports, as is shown below.

```
component application CanWriter {
        ports {
                server  CanServer provides CSInterface1
                sender CanSenderPort provides SRInterface1


        }
```

Now that the components themselves are defined, they can be instantiated and be connected to each other through ports. This is called a composition. We created the following architecture as a basis for our composition:

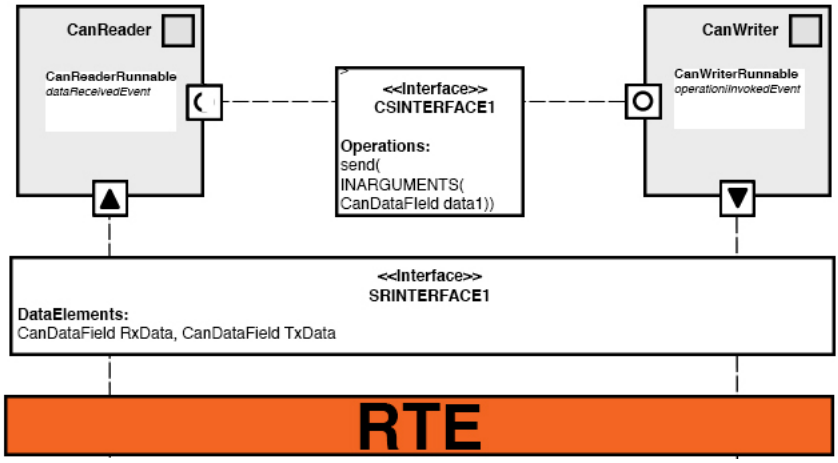

Figure 3.19: *Desired system architecture*

These instances and connections were then described in ARtext SWCD code the following way:

```
composition MyComposition{

        // Instantiation
        prototype CanWriter  CanWriter_1
        prototype CanReader  CanReader_1


        // External ports
        ports {
                provides CanWriter_1.CanSenderPort Tx01
                requires CanReader_1.CanReceiverPort Rx01

        }
connect CanWriter_1.CanServer to CanReader_1.CanClient

}
```

This code basically corresponds to Fig. 3.19 above. One instance of each SWC is created, and external ports are created for each sender/receiver port, since these are to transmit data to the COM module. Also, the connection between the client of CanReader and the server of CanWriter is established.

3.3.5 Develop Application Software Component

In this phase, the actual implementation of the SWC is conceived. The first step of doing this is to define a runnable within the SWC. Thus, a runnable for each SWC was defined. An example of this is shown in Fig. 3.20 and Fig. 3.21 below.

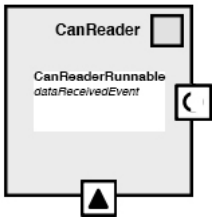

Figure 3.20: *CanReader with runnable*

Figure 3.21: *CanWriter with runnable*

The CanReaderRunnable is set to be activated on *dataReceivedEvent*, since it should read data as soon as it is available. The CanWriterRunnable is set to be activated on *operationInvokedEvent*, as it is supposed to be activated when a function call is made from CanReader. To implement this, there are two things that need to be done:

- Describe the internal behavior of the component
- Implementing the function in a programming language

Describing the internal behavior of the SWC is done in ARtext SWCD code. An example of how to describe the internal behavior for the CanReader and CanWriter SWCs is shown in the code below.

```
internalBehavior CanReaderBehavior for CanReader
{
        runnable CanReaderRunnable [1.0] {
                dataReadAccess CanReceiverPort.RxData
                dataReceivedEvent CanReceiverPort.RxData
                serverCallPoint synchronous CanClient.*
                        }
}

internalBehavior CanWriterBehavior for CanWriter {
        runnable CanWriterRunnable [1.0] {
                dataWriteAccess CanSenderPort.TxData
                operationInvokedEvent CanServer.Send
        }
}
```

The CanReaderRunnable needs dataReadAccess in order to be able to read from its receiver port. It also needs to be triggered when new data has arrived, thus a *dataReceivedEvent* to that port is declared. It also needs a serverCallPoint in order to activate CanWriterRunnable. The CanWriterRunnable needs write access to the sender port in order to be able to send data. It also needs the operationInvokedEvent to be activated on a function call. The language of the implementation also needs to be described in ARtext SWCD code, this is shown in the code below.

```
implementation CanReaderImplementation for CanReaderBehavior {
        language c
        codeDescriptor "src"
}
implementation CanWriterImplementation for CanWriterBehavior {
        language c
        codeDescriptor "src"
}
```

The next step is to actually implement the runnables in C code. Since the only functionality required in this thesis is to be able to send and receive CAN data, the actual C implementation of these SWC's is really simple. The implementation of the CanReaderRunnable is shown in the code below.

```
#include "Rte_CanReader.h"

void CanReaderRunnable(void){
                Rte_Call_CanClient_Send(Rte_IRead_CanReaderRunnable_CanReceiverPort_RxData());
}
```

The Rte_IRead_CanReaderRunnable_CanReceiverPort_RxData() function refers to a function that will later be generated by the RTE Editor. It returns a pointer to the data received on the receiver port of the CanReader SWC. The Rte_Call_CanClient_Send function is also generated by the RTE at a later stage. It is a function call to the CanWriterRunnable, and the Rte_IRead_CanReaderRunnable_CanReceiverPort_RxData() function is passed as a parameter in order for the CanWriterRunnable to send the received data. The implementation of the CanWriterRunnable is shown in the code below.

```
#include "Rte_CanWriter.h"

void CanWriterRunnable(const CanDataFields *reader) {
        Rte_IWrite_CanWriterRunnable_CanSenderPort_TxData(reader);
}
```

The Rte_IWrite_CanWriterRunnable_CanSenderPort_TxData(reader) function is, once again, another function that will be generated by the RTE generator. It takes a pointer to data of the type CanDataFields as a parameter. Since the received data from CanReader is passed in the argument *reader*, the write function will write the received data to its receiver port.

3.3.6 Design System

In this phase, the Ecu Extract is created, using a type of ARtext language called SYSD. This is divided into two steps, "Design System", and "Design Communication".

3.3.6.1 Design Communication

In this step, system signals are defined, the code for this is shown below.

```
//System signals
systemSignal MotorControlLowSSig
systemSignal MotorControlHighSsig
systemSignal SpeedLowSSig
```

```
systemSignal SpeedHighSSig

//System signal groups

systemSignalGroup MotorControlSSigGroup {
    systemSignals {
        MotorControlLowSSig, MotorControlHighSsig
}

systemSignalGroup SpeedSSigGroup {
    systemSignals {
        SpeedLowSSig, SpeedHighSsig
}

//Internal signals
iSignal for MotorControlLowSSig as MotorControlLowISig
iSignal for MotorControlHighSSig as MotorControlHighISig
iSignal for SpeedLowSSig as SpeedLowISig
iSignal SpeedHighSSig as SpeedHighISig

//Internal signal groups

iSignalGroup for MotorControlSSigGroup as MotorControlISigGroup {
    iSignals {
        MotorControlLowISig, MotorControlHighIsig
}

iSignalGroup for SpeedSSigGroup as SpeedISigGroup{
    iSignals {
        SpeedLowISig, SpeedHighISig
}

//Mapping internal signals to PDUs

iSignalPdu MotorControlIPdu{
        iSignalIPduMapping for MotorControlLowISig as MotorControlLowMapping
        iSignalIPduMapping for MotorControlHighISig as MotorControlHighMapping
        iSignalIPduMapping for group MotorControlISigGroup as MotorControlRightISigGroupMapping
}

iSignalPdu SpeedIPdu{
        iSignalIPduMapping for SpeedHighISig as SpeedHighMapping
        iSignalIPduMapping for SpeedLowISig as SpeedLowMapping
        iSignalIPduMapping for group SpeedISigGroup as SpeedISigGroupMapping
}
```

These signals describe the signals created in the COM module, and create a mapping which will later be used to connect the other ports to the COM.

3.3.6.2 Design System

Here, the actual Ecu Extract is created. The code for this is shown below:

```
system EcuExtract {
        rootComposition MyComposition

        mapping as ImplementationMappings{
                implMap CanWriterImplementation to CanWriter_1
                implMap CanReaderImplementation to CanReader_1
        }
        mapping as SignalMappings {
                signalMap record outerPort Tx01.TxData to MotorControlGroupSSig{
                    Low to MotorControlLowSSig
                    High to MotorControlHighSSig
                    }
                signalMap record outerPort Rx01.RxData to SpeedGroupSSig{
                    Low to SpeedLowSSig
```

```
                High to SpeedHighSSig
        }
}
```

First, the composition created in the previous step is included. Then, the implementation from the system description phase is mapped to each component instance. The outer ports are then mapped to the system signal groups and system signals created in the previous step.

The Ecu Extract is then exported in the AUTOSAR XML format and is in the next step combined with the basic software.

3.3.7 Build Ecu Software

This is the last step, which at the end will result in an executable file. Basically this step combines what was generated from the BSW configuration phase with the Ecu Extract. To map these together, the RTE module is used.

3.3.7.1 Runtime Environment

The whole purpose of the RTE is to provide a common interface between the application layer and the BSW layer. To achieve this, some configurations have to be done using the RTE Editor within Arctic Studio. When configuring the RTE there are several parameters to take into consideration namely

- Entity
- Runnable
- Os Task
- RTE Event
- Os Event

In the RTE Editor is the Entity column where all SWCs within your projects are listed. In order for your SWCs to do something at all, the necessary step is to instantiate them. When the instantiation has been done, you get access to a list of all RTE Events within the SWC However these RTE Events are not mapped to anything except the runnable in which they were created. At first in our thesis work we were using a special type of RTE Event called *timingEvents* which were, as mentioned in the theory chapter, periodically invoked. However as we were only interested in activating a runnable when data had been received on that particular receiver port within the SWC, we realized that the RTE Event, the *dataReceivedEvent* fulfilled our demands. Fig. 3.22 shows the configurations mentioned above in the RTE editor.

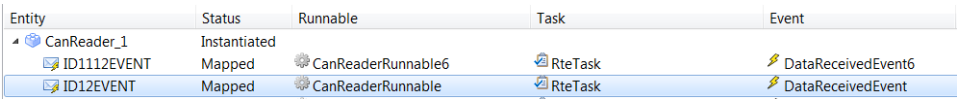

Figure 3.22: *Figure illustrating the RTE Editor*

Furthermore it was also necessary to link each runnable with a corresponding *dataReceivedEvent* to a task within the OS. The aforementioned task called RteTask was created for this purpose. However this was not everything that was needed as every specific *dataReceivedEvent* also needed to map an OS Event together with the task they were mapped with. Below is the Fig. 3.23 illustrating the connection between the task and the OS events.

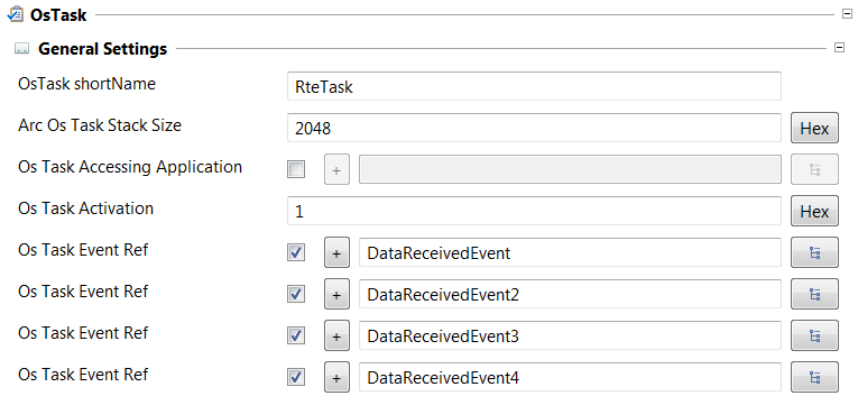

Figure 3.23: *Example of events referring to an OS task*

These configurations altogether makes up a common interface from the application layer to the BSW layer.

3.3.7.2 Building the software

When the RTE configuration is complete, code is generated from the RTE Editor. At this point, the entire project created is compiled into a single executable file of the type elf. This is done in the cmd terminal using a make command that looks like this:

```
|| make BOARDDIR=tmdx570ls12hdk CROSS_COMPILE=arm-none-eabi- BDIR=../can all
```

The parameters that are passed are the type of development board, and what compiler is to be used. When compiling, you must be inside the folder containing the Arctic Core software, and the BDIR parameter needs to refer to the path of the Arctic Studio project. When compiling is done, an executable elf file is generated which can then be programmed to the development board.

3.3.8 Simulator Application Development

To create the simulator applications, we created two separate applications which we have named CanSim and CanGraph. CanSim generates data on the CAN bus, and CanGraph plots this on a graph. Both applications use the PyQt package for Python, and also a package called PyCan, which is used for CAN communication.

3.3.8.1 CanSim

In PyQt, graphical objects are referred to as widgets. Widgets exist in the form of classes, for example if you want to create a button you create an instance of the class QPushButton. Custom widgets can be created as well. The basic approach of creating a GUI using PyQt is to declare a main window, and then add these widgets to the window. Some widgets, for example buttons, can be linked to functions executing a designated function for each button. The GUI we designed can be seen in Fig. 3.24 below.

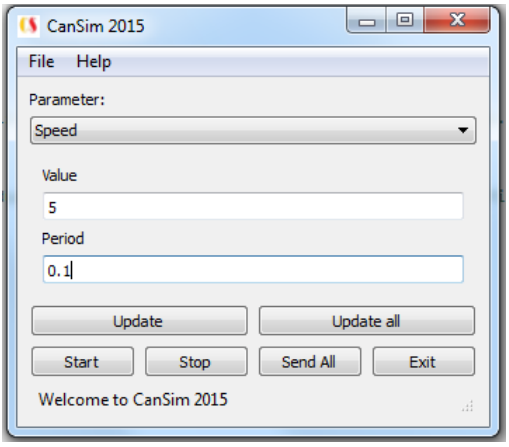

Figure 3.24: *Screenshot from CanSim 2015*

Since the application needs to be able to send messages with different IDs at different times, a custom type of widget was created, which is called the InputWidget() class. This widget contains two widgets of the type QLineEdit, which is a text edit field. It also contains the two labels "Value" and "Period". The widget also has some internal variables for Value, Period, and CAN ID. The widgets can be accessed from a combobox in the main window which is of the type QComboBox. After values have been entered in the text edit fields, the internal variables of the InputWidget need to be updated. This is done with the "Update" and the "Update All" buttons. The "Update" button updates the variables for the selected parameter in the combobox only, while the "Update All" button updates all parameters. There are two ways to add parameters to the list. One is an "Add parameter" feature, which prompts the user for a CAN ID and a parameter name. It is found in the file drop-down menu which is shown in Fig. 3.25 below. The "Add Parameter" window is shown in Fig. 3.26 below.

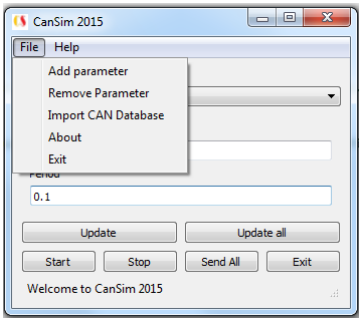

Figure 3.25: *File drop-down menu in CanSim*

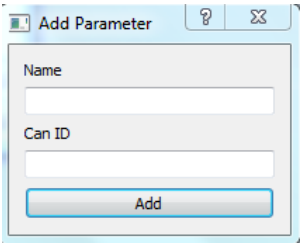

Figure 3.26: *Add Parameter window in CanSim*

We also created another way to import parameters. We defined a type of database which contains CAN ID, parameter name, and direction.

```
1;Speed;TX
2;Yaw Rate;RX
3;Temperature;TX
```

Parameter values are declared in the following order: *CANID;Parameter Name;Direction*. This database can then be imported into CanSim using the "Import CAN Database" feature, which will prompt the user to select a file. If a text file written in the database format specified above is opened, it will create input widgets for each parameter with "TX" set as direction. The simulation itself is based on a function from PyCan which is called the PeriodicCallback. The PeriodicCallback takes CAN ID, value, and period as input parameters and sends CAN messages with the specified CAN ID, name, and value. When clicking the "Start" button, PeriodicCallback objects are created for each input widget and then started. These objects are run in parallel with the GUI due to the use of threading. The simulation can then be stopped by pushing the "Stop" button. There is also a "Send All" button which sends a single CAN message of all input widgets in the list. A flowchart of the CanSim simulator can be seen in the Appendix.

3.3.8.2 CanGraph

The CanGraph application is based on widgets defined in the PyQtgraph package. By using the PyQtgraph library for Python, these widgets are able to plot live data and more. The main window is based on the class GraphicsWindow from the PyQtgraph library. The GraphicsWindow class is a plot window where different plots and curves can be added. These are added by using the same CAN database that was defined for the CanSim application. When this database is imported (using the "Import Can Database" option in the file drop-down menu), several new instances of a class we created called CanMessage() are created. This class includes internal variables for the CAN ID, parameter name, and unit. It

also includes two variables for data. The pycan function we use for CAN reception, can.receive(), returns a list of the data of the CAN message, so one of the internal variables is a hexadecimal value in the form of the list, and the other variable is the hexadecimal value converted to an integer, in order to have a value that can be displayed on the graph. At the import stage, each CanMessage() object is assigned to another object of the type plot from the PyQtGraph package, resulting in each CAN message having its own unique plot Fig. 3.27 shows what plots are generated if the same CAN database as in section 3.3.9.1 is used.

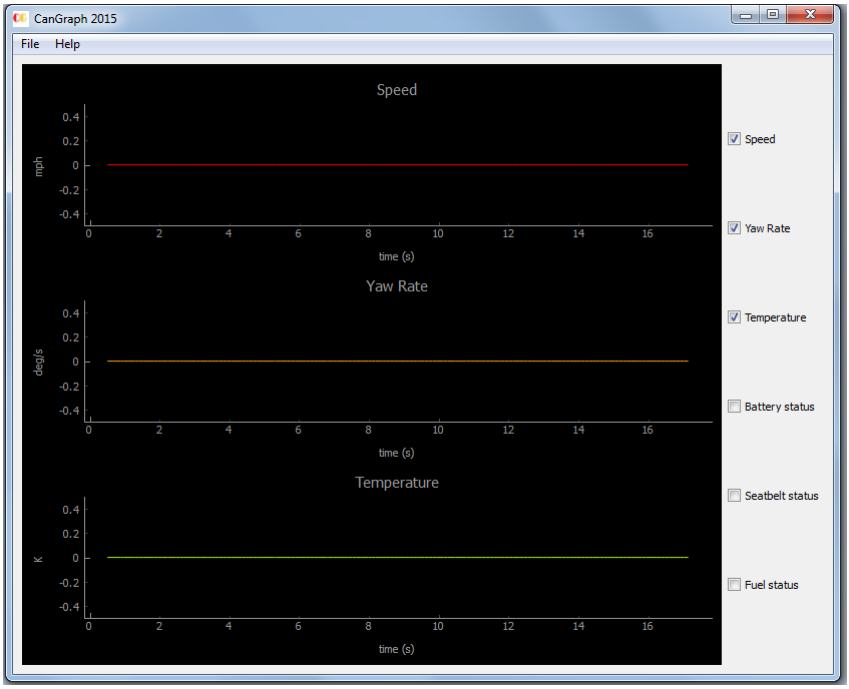

Figure 3.27: *Example of plots in CanGraph*

The can.receive() is a polling-driven function, which means it needs to be called constantly in order to receive CAN messages. However, the GUI code needs to be able to run at the same time, as well as the plot function, which is activated by a timer. We solved this by introducing a special thread class from the PyQt library, the Qthread. This thread continuously polls for data, and when data is received the CanMessage() object with the matching CAN ID gets its internal variables for data updated. As mentioned earlier, the actual plot function is activated by a timer, running with a period of 1000 ms. When this function is activated, it updates each plot with the current value stored in its assigned CanMessage object. A flowchart of the CanGraph simulator can be seen in the Appendix.

3.3.9 Testing

Given our specified thesis scope, the requirements that were stated were to be tested and proven. We needed to establish CAN communication throughout all of the layers within AUTOSAR, as well as create 36 different signals with unique IDs as well as PDUs.

To test that our system meets the given requirements, we have set up a test case. In this test case we will check that the system can process CAN messages with 36 different CAN IDs with the DLC of 5 bytes. In order to thoroughly test and validate these requirements, we have introduced 36 different signals in our system. 18 of those signals are sent from the ECU, and 18 of those are received. Each receiving signal activates a specific runnable in the system, which has the task of sending back the same message with a specific CAN ID. Since we need to verify that we can send 5 bytes of data, we will on each CAN message send the data value 4294967296. This is a value that requires 5 bytes of data, making it a suitable test value. This will validate the functionality of the system, since it will show that each CAN ID is independently recognised, and that the data received is sent back unaltered.

The test case will be tested using the two GUI applications we've developed, which will use a CAN database to send CAN messages with same CAN IDs that are defined as signals in the ECU.

3.3.9.1 Testing Environment

The testing environment consists of a PC, a USB to CAN adapter (Kvaser Leaflight 2.0) and our development board MCBTMS570, below is a Fig. 3.28 illustrating the testing environment.

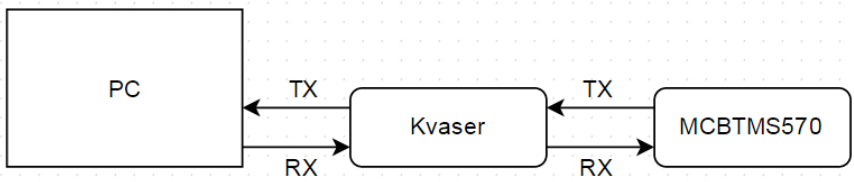

Figure 3.28: *Illustration of the physical testing environment*

The simulations start when a start button in CanSim is pressed. This causes the PC to send CAN messages on the CAN bus to the ECU. The ECU receives the messages and sends out corresponding CAN messages for each CAN message that has been received, for example if CAN message with ID1 is sent to the ECU, the ECU then receives the message and sends out CAN message with ID2 on the CAN bus. The CAN bus delivers the messages to CanGraph, which plots both the CAN message that was being sent from CanSim and the corresponding CAN messages from the ECU, a Fig. 3.29 below describes this further.

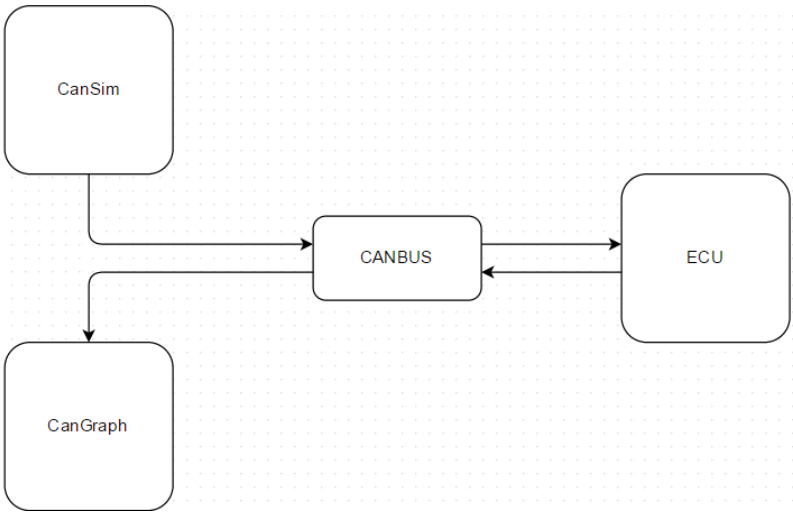

Figure 3.29: *Illustration of the relation between CanSim - ECU - CanGraph*

Furthermore, there is a photograph below, Fig. 3.30 displaying the testing environment.

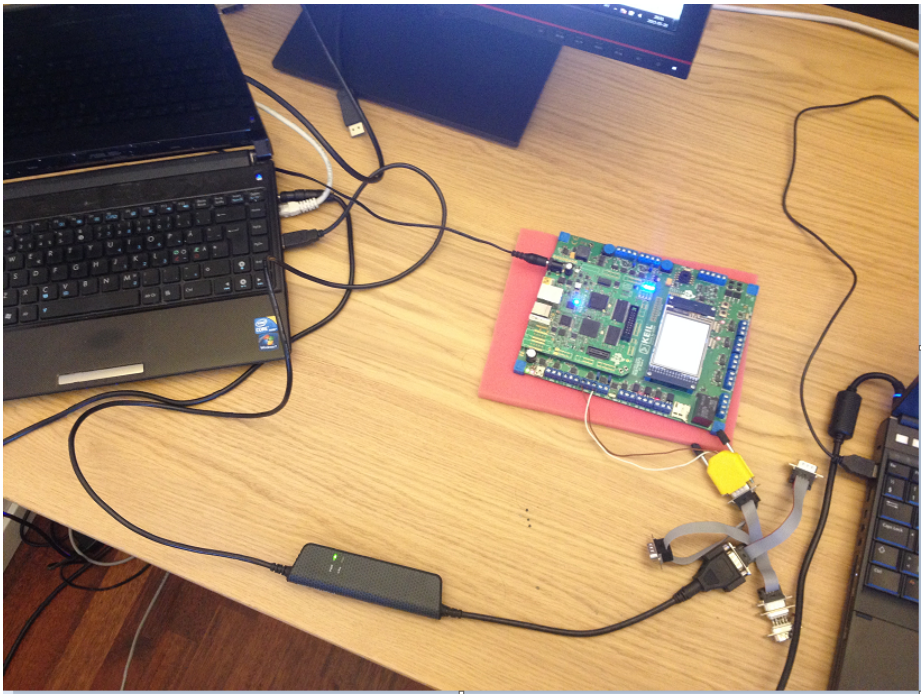

Figure 3.30: *Photograph of the testing environment*

4

Results

4.1 ECU

In this thesis, an ECU was designed with the AUTOSAR system architecture. This ECU has the ability to send and receive CAN messages with 36 different CAN IDs and DLC of 5 bytes. The BSW modules were configured and generated using Arctic Studio, with some modules requiring modifications to the source code and additional implementation, which was done in C language. The SWCs were modeled using Artext language and implemented using C language. Fig 4.1 below shows an overview of which modules the ECU includes and how they were implemented.

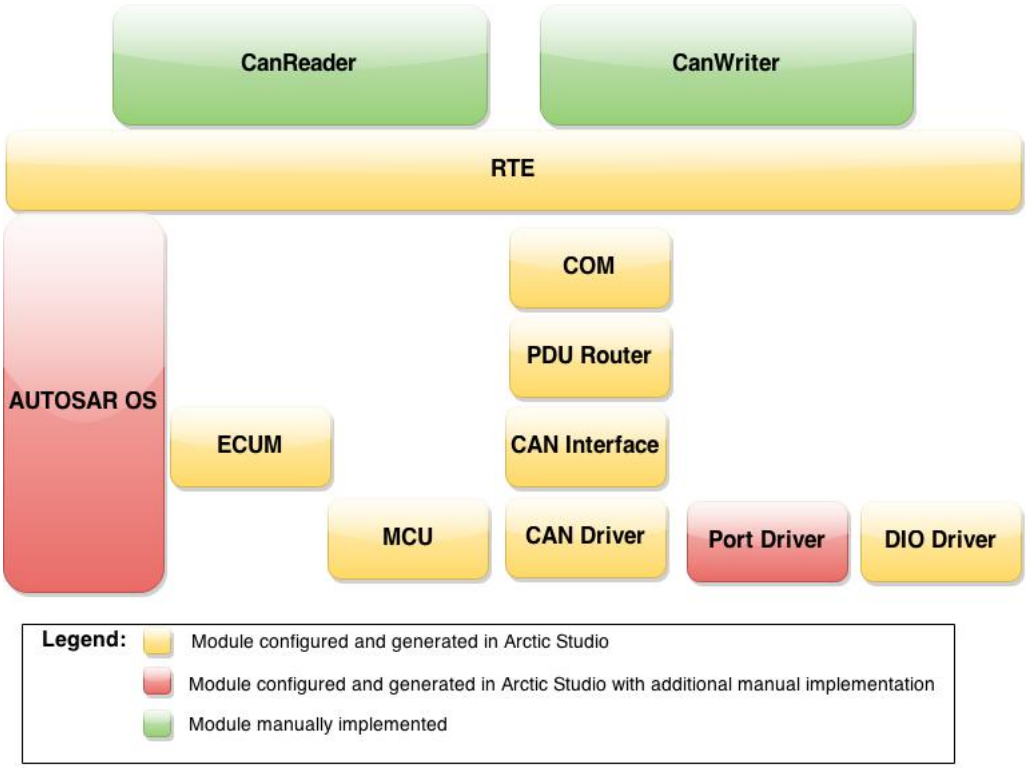

Figure 4.1: *System overview of ECU*

The AUTOSAR OS and Port Modules were generated using Arctic studio, however additional implementation was required. The Port module was not compatible with our development board, and lacked support for the CAN pins. For the AUTOSAR OS module, implementation of the tasks were needed, as the module only generates the scheduling software for the tasks and not the tasks themselves. More details about this, as well as all configuration and implementation, can be found in the method section 3.3.2 of this thesis.

In order to test that the system meets the requirements as specified in section 1.3 it was, as explained before, necessary to verify that the system can handle 36 different types of CAN IDs. For this reason, the system contains 36 different signals, each corresponding to a unique CAN ID, with 18 TX signals and 18 RX signals. In the SWC CanReader, we assigned each RX signal a unique runnable, and in the SWC CanWriter we assigned each TX signal a unique runnable. Each CanReader runnable is then assigned to call a designated runnable belonging to CanWriter. This is clarified in Fig. 4.2 below.

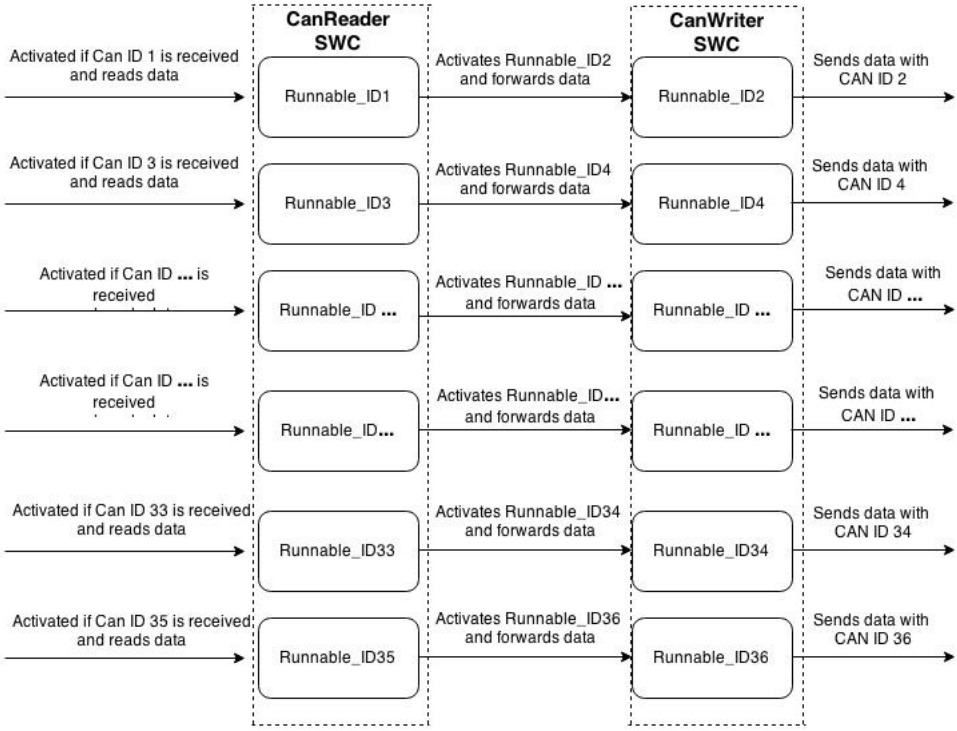

Figure 4.2: *Runnable Activation Schedule*

4.2 Simulation

The simulation was done with the applications CanSim and CanGraph which we designed ourselves. The database on the next page was created and then imported into CanSim:

```
1;Message_1;TX;
2;Message_2;RX;
3;Message_3;TX;
4;Message_4;RX;
5;Message_5;TX;
6;Message_6;RX;
7;Message_7;TX;
8;Message_8;RX;
9;Message_9;TX;
10;Message_10;RX;
11;Message_11;TX;
12;Message_12;RX;
13;Message_13;TX;
14;Message_14;RX;
15;Message_15;TX;
16;Message_16;RX;
17;Message_17;TX;
18;Message_18;RX;
19;Message_19;TX;
20;Message_20;RX;
21;Message_21;TX;
22;Message_22;RX;
23;Message_23;TX;
24;Message_24;RX;
25;Message_25;TX;
26;Message_26;RX;
27;Message_27;TX;
28;Message_28;RX;
29;Message_29;TX;
30;Message_30;RX;
31;Message_31;TX;
32;Message_32;RX;
```

Since CanSim imports the messages with the direction set to "TX", in this case it imported the messages with odd Can IDs. This was to easily distinguish between messages sent to the ECU and messages received from the ECU. The data of each CAN message was set to be 4294967296. This specific data value was chosen since it uses 5 bytes. Thus, if the same value is returned, we can verify that 5 bytes of data have successfully been received and sent by the ECU. Each message was given a unique time period as well. This way, in the graphs we could easily distinguish which, from the PC point of view, TX message triggers which RX message. The simulation values entered in CanSim are shown in the table below.

Parameter Name	ID of message sent to ECU	Data sent (Integer)	Period in seconds
Message_1	1	4294967296	1
Message_3	3	4294967296	2
Message_5	5	4294967296	3
Message_7	7	4294967296	4
Message_9	9	4294967296	5
Message_11	11	4294967296	6
Message_13	13	4294967296	7
Message_15	15	4294967296	8
Message_17	17	4294967296	9
Message_19	19	4294967296	10
Message_21	21	4294967296	11
Message_23	23	4294967296	12
Message_25	25	4294967296	13
Message_27	27	4294967296	14
Message_29	29	4294967296	15
Message_31	31	4294967296	16
Message_33	33	4294967296	17
Message_35	35	4294967296	18

Table 4.1: *Simulation values of first simulation*

The same database is then imported into CanGraph. CanGraph creates a plot for each CAN ID in the database, displaying both incoming and outgoing messages on the CAN Bus. If the ECU works as expected, each message that is sent to the ECU should, through the runnables shown in Fig. 4.2, send back the data but with another CAN ID. The CAN IDs defined as RX signals in the ECU are all odd IDs from 1 to 35. As shown in Fig. 4.2, each runnable in CanReader is assigned to activate a runnable sending the same message but with a CAN ID which is the CAN ID of the received message + 1. Thus, the expected results of the simulation are that after 1 second, since the period for Message_1 is 1, the value of Message_1 should be 4294967296. Since the ECU will immediately send back a message with the same data but with the consecutive CAN ID, the value of Message_2 should also rise to 4294967296 at the same time. Similarly, after

14 seconds Message_27 should be sent to the ECU, given its time period,and thus its value should become 4294967296. Since the ECU is designed to immediately send out a message with the same data but with the consecutive CAN ID, the value of Message_28 should at the same time rise to 4294967296. The same pattern applies to all CAN Messages. A CAN message with an odd ID is sent to the ECU, the corresponding runnables for the CAN ID are activated, and the data is then sent in a CAN message with the CAN ID that is the same of the received message + 1.

The actual simulation results with the values from Table 4.1 are presented in the figures 4.3-4.8 below. The RX/TX label on each graph denotes if the message was sent or received by the PC. Each plot is updated once every second, and the value of the plot is only updated if new data has been received, otherwise it will remain at the same level. The scale on each plot is the shown value $* 10^9$. There is no unit specified to the value, it's simply just an integer.

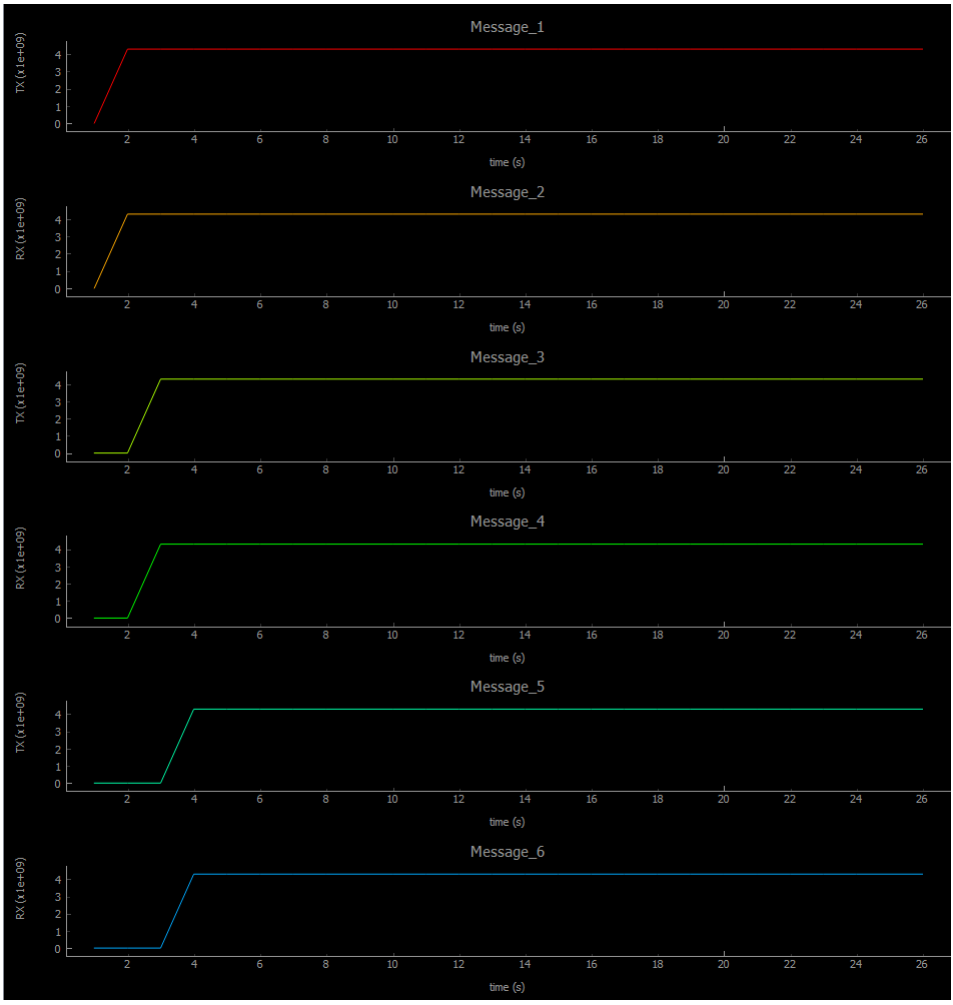

Figure 4.3: *Simulation plot of messages with CAN IDs 1-6*

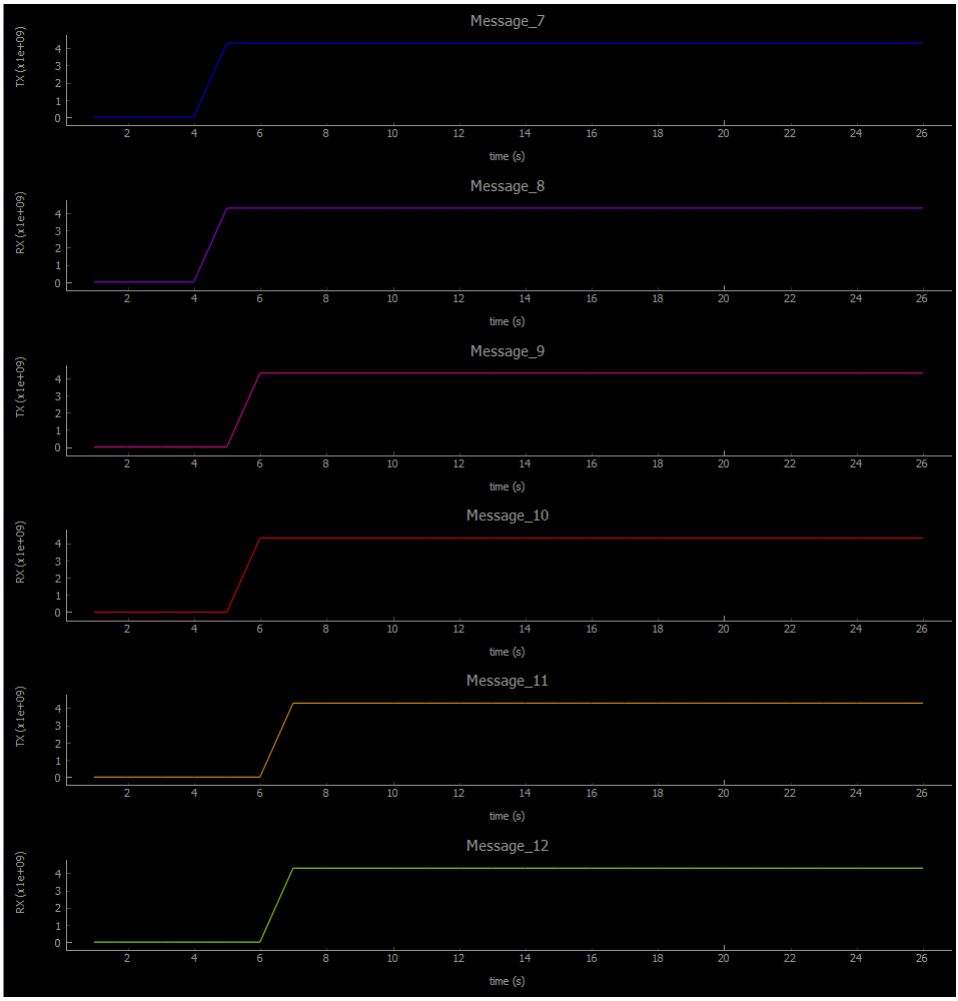

Figure 4.4: *Simulation plot of messages with CAN IDs 7-12*

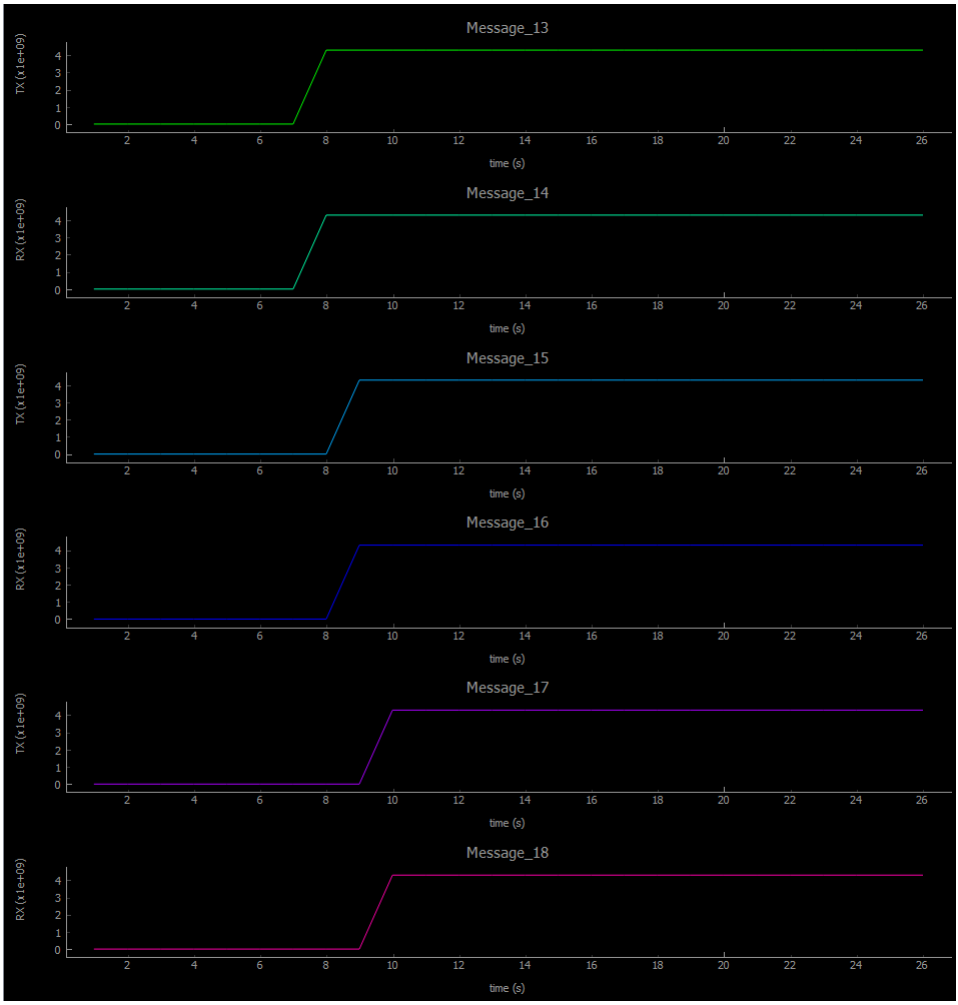

Figure 4.5: *Simulation plot of messages with CAN IDs 13-18*

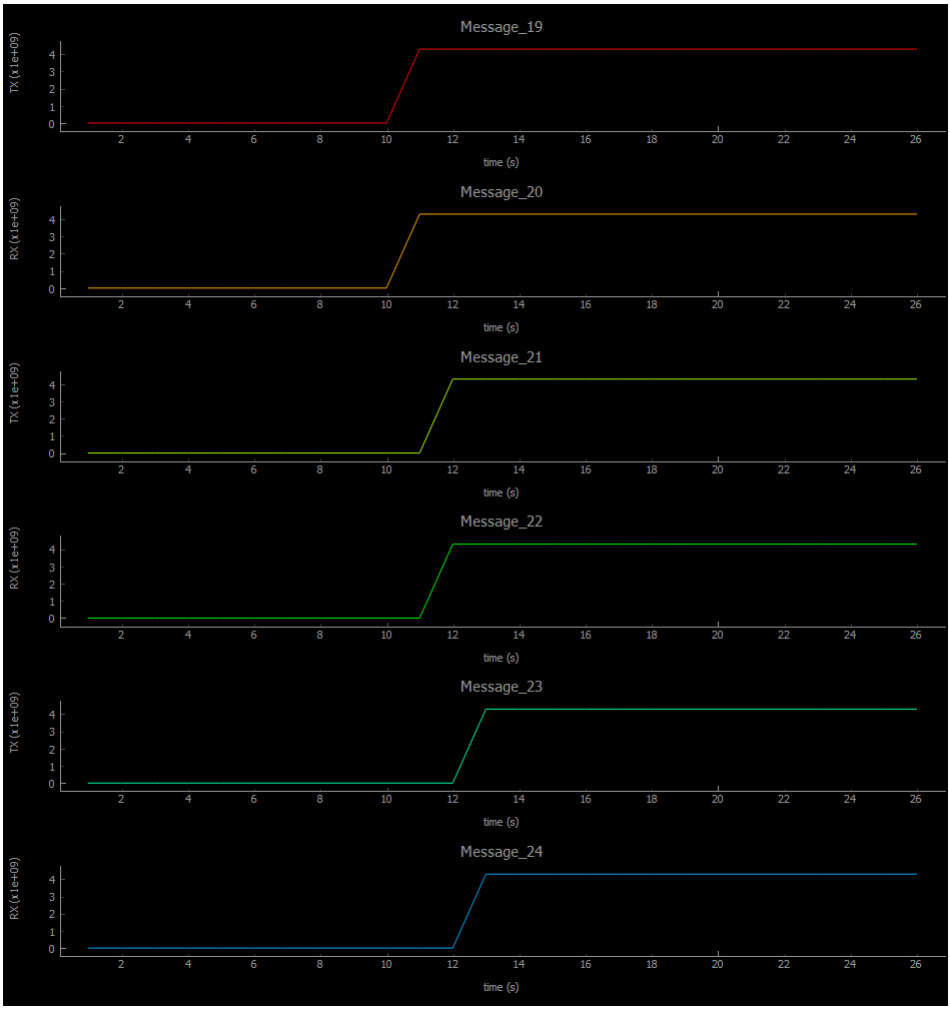

Figure 4.6: *Simulation plot of messages with CAN IDs 19-24*

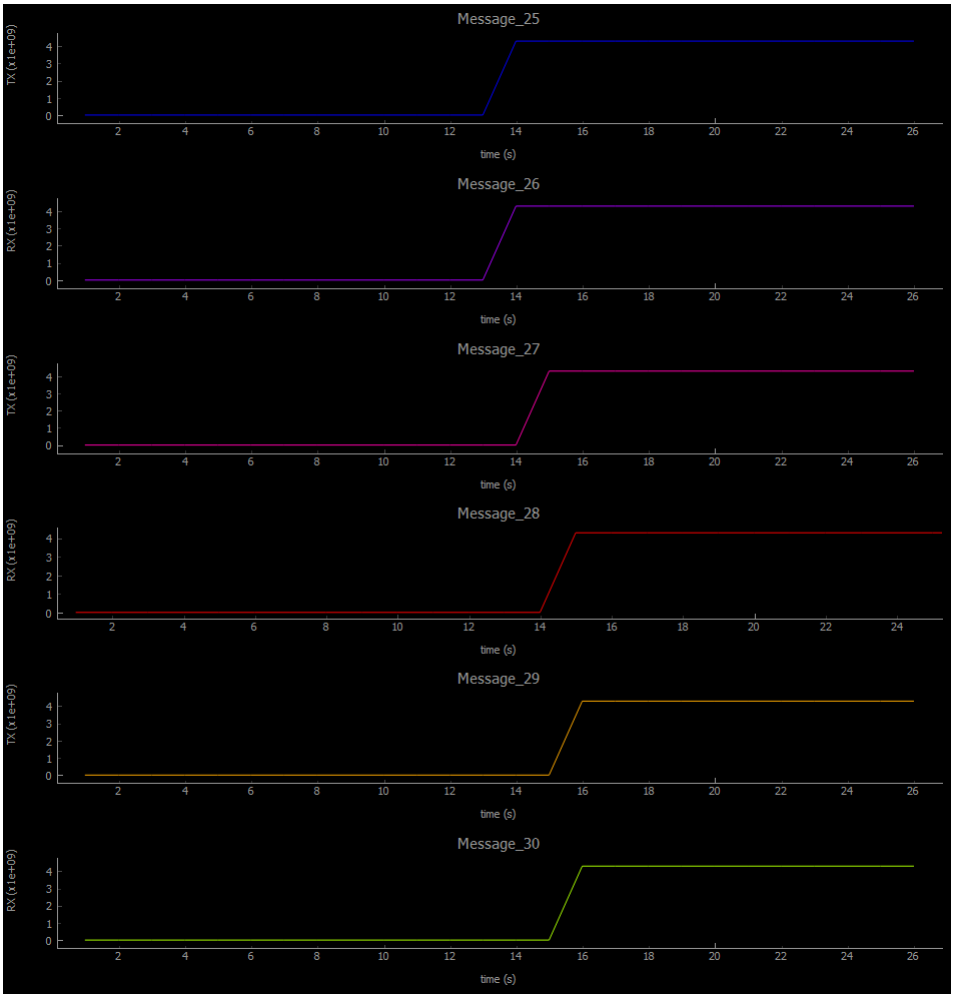

Figure 4.7: *Simulation plot of messages with CAN IDs 25-30*

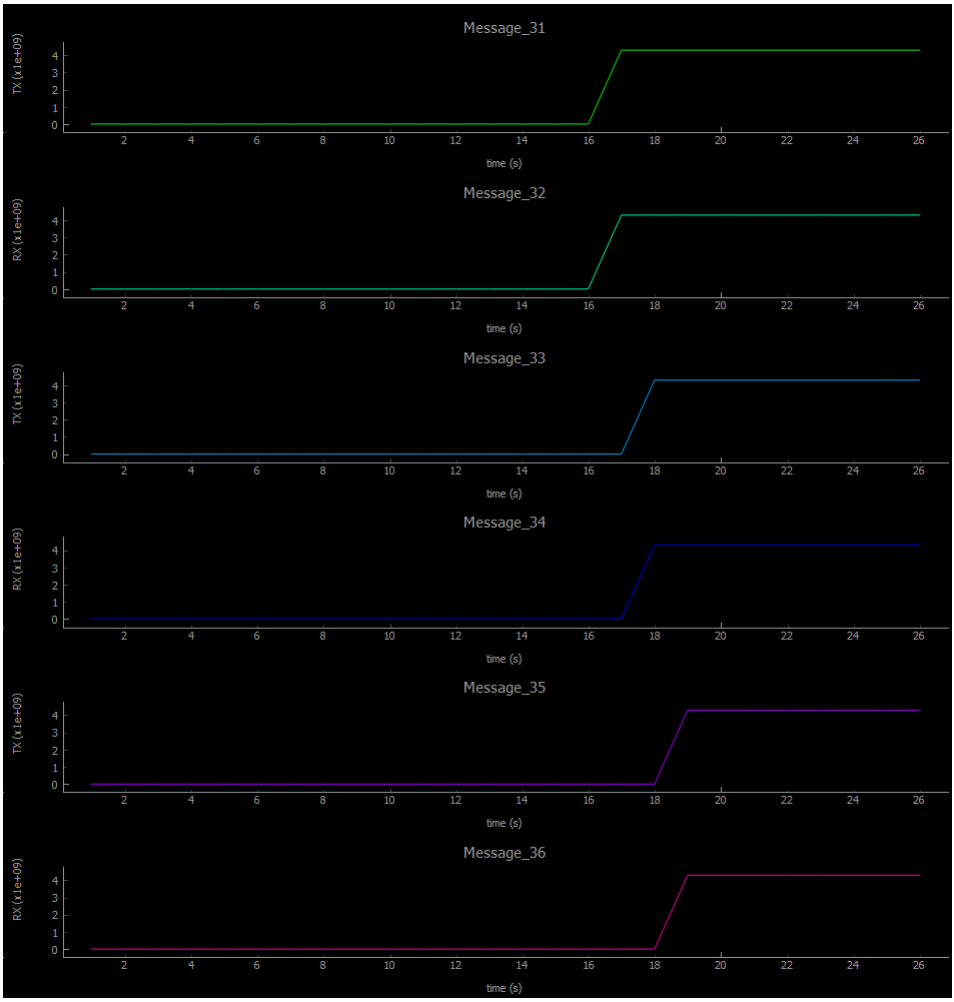

Figure 4.8: *Simulation plot of messages with CAN IDs 31-36*

To further emphasize that the ECU's message handling works as specified, an additional simulation was performed with some miscellaneous values for period and data value. These values are presented below. The signals on the ECU are still mapped in the same way as in the previous simulation, i.e. that a received message with CAN ID 7 will trigger transmission of a message with the CAN ID 8, and so on.

Parameter Name	ID of message sent to ECU	Data sent (Integer)	Period in seconds
Message_1	1	10	3
Message_3	3	20	8
Message_5	5	40	5
Message_7	7	130	9
Message_9	9	50	2
Message_11	11	170	20
Message_13	13	30	7
Message_15	15	90	4
Message_17	17	100	4
Message_19	19	33	10
Message_21	21	60	12
Message_23	23	110	17
Message_25	25	60	13
Message_27	27	80	21
Message_29	29	150	6
Message_31	31	140	14
Message_33	33	3	18
Message_35	35	160	1

Table 4.2: *Simulation values of second simulation*

The following figures 4.9-4.14 show the simulations with the values in table 4.2 above. As these data values are considerable lower than the ones in the previous simulation, no scaling is done of the data values.

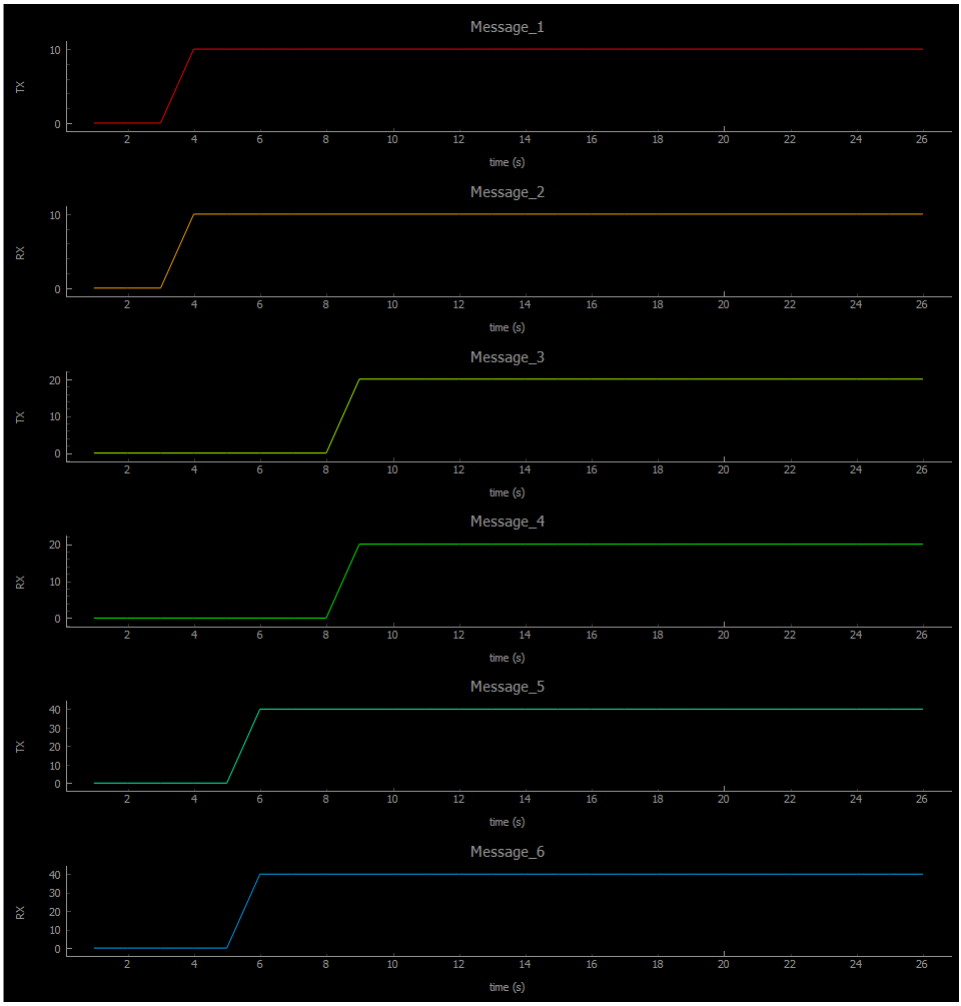

Figure 4.9: *Simulation plot of messages with CAN IDs 1-6*

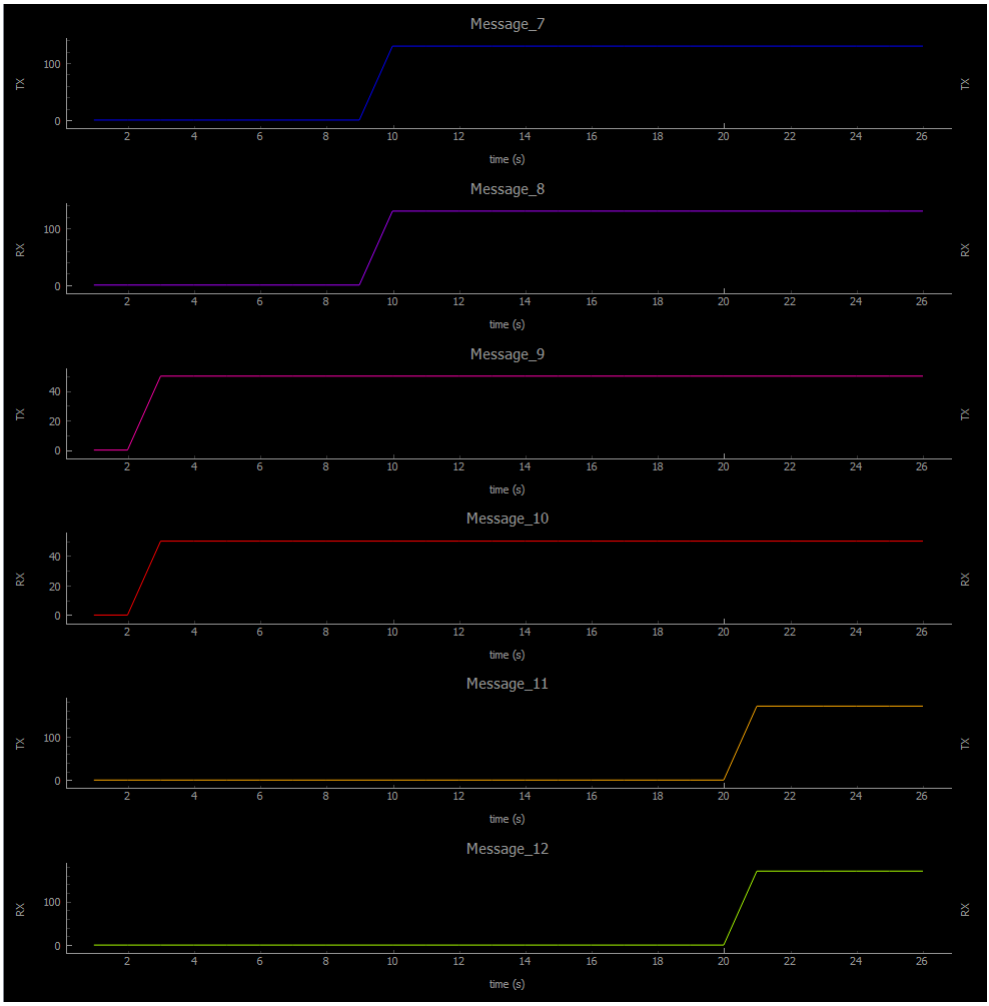

Figure 4.10: *Simulation plot of messages with CAN IDs 7-12*

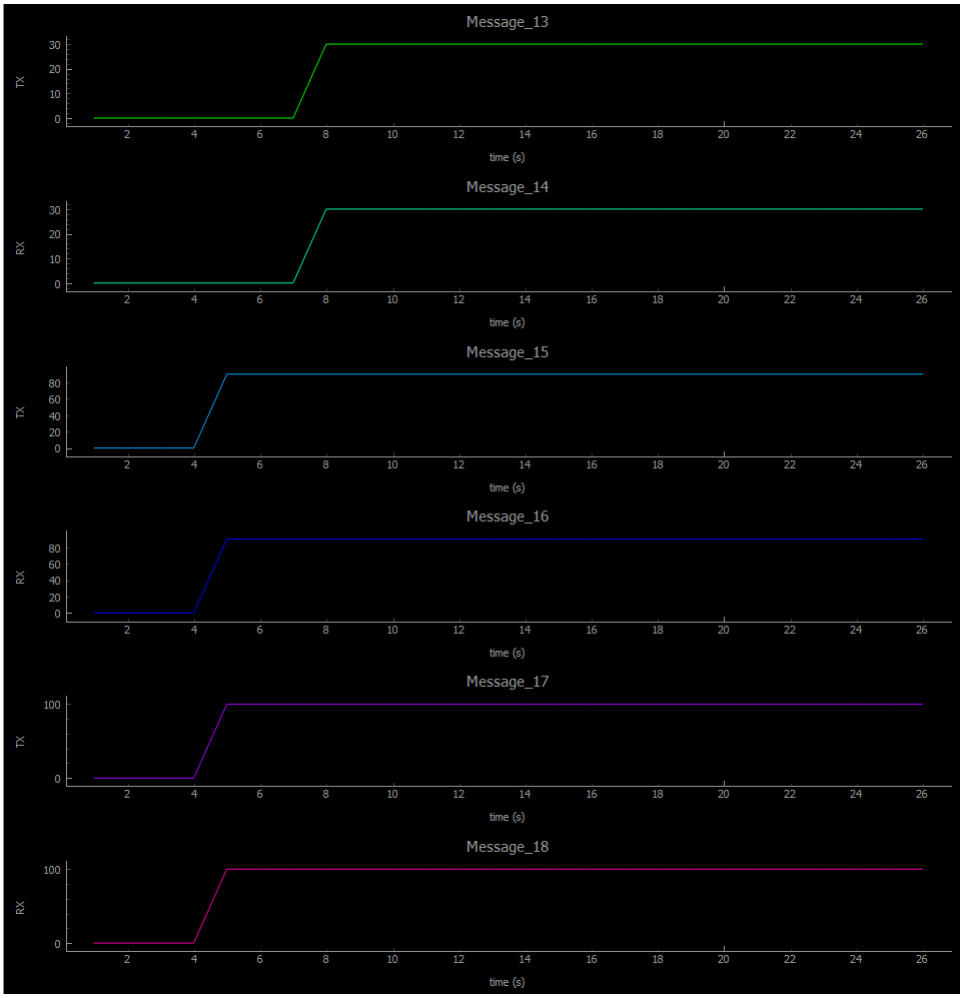

Figure 4.11: *Simulation plot of messages with CAN IDs 13-18*

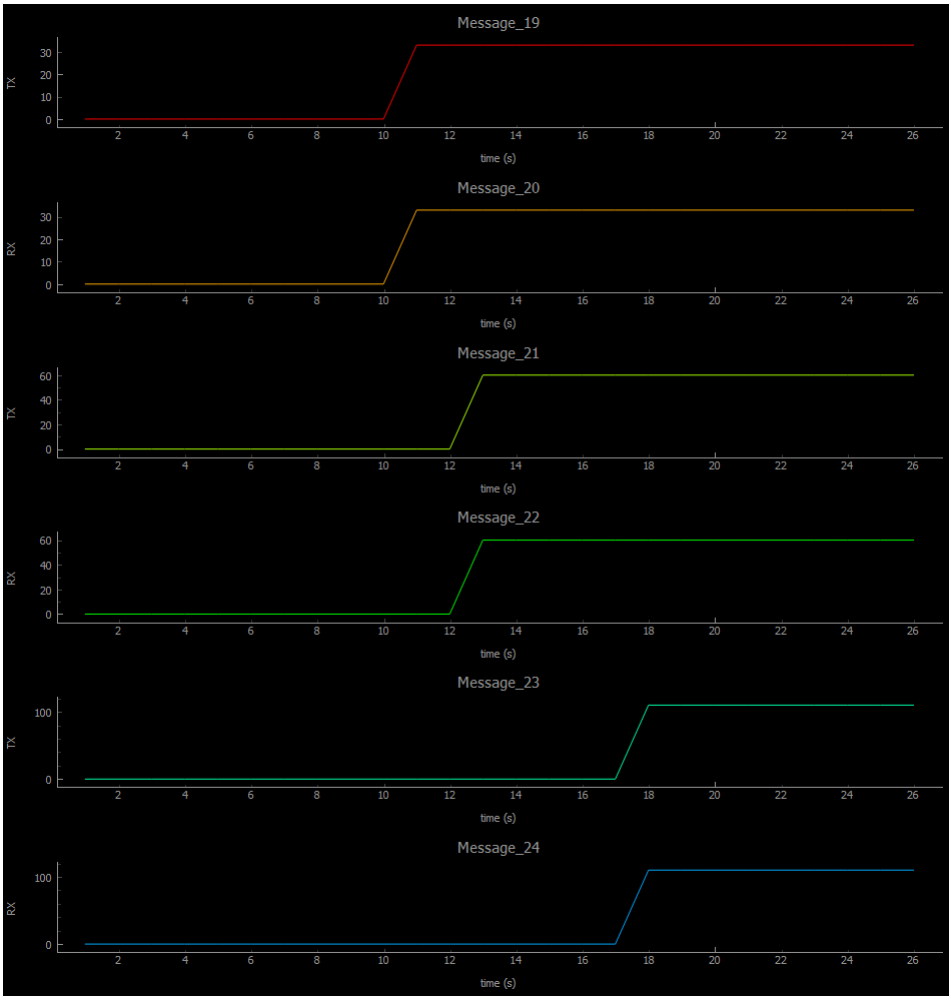

Figure 4.12: *Simulation plot of messages with CAN IDs 19-24*

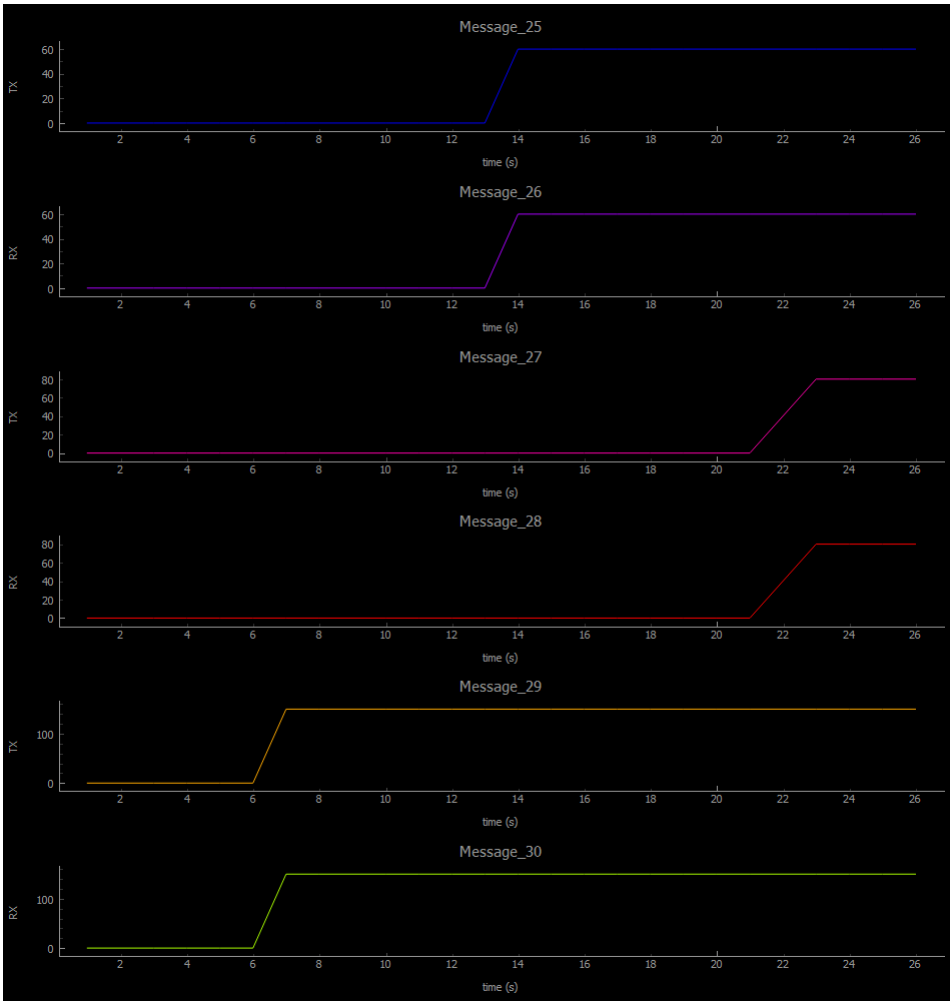

Figure 4.13: *Simulation plot of messages with CAN IDs 25-30*

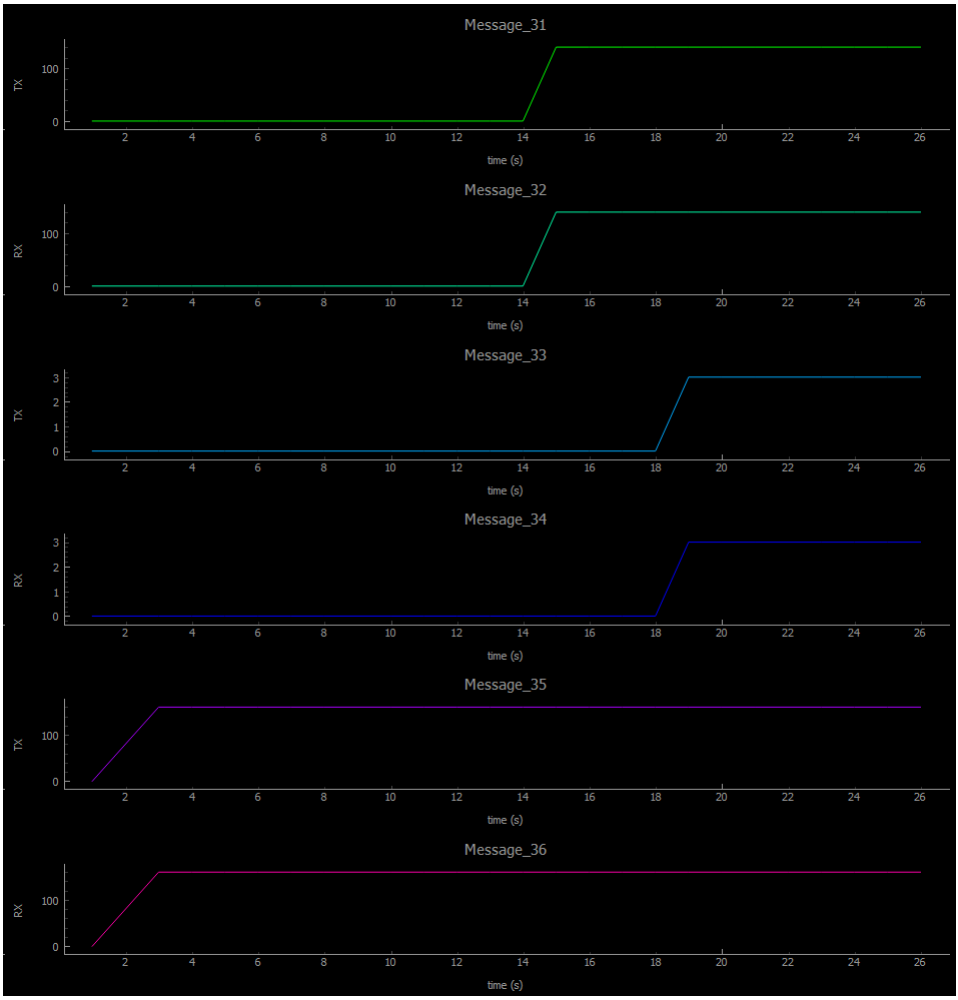

Figure 4.14: *Simulation plot of messages with CAN IDs 31-36*

5

Discussion

5.1 Analysis of the results

The plots in the results section 4.1 show the CAN traffic on the bus in the form of 36 individual plots. Each plot depicts messages of one specific CAN ID. As is stated in the purpose section in 1.3, our CAN communication platform needs to be able to process 36 signals with unique CAN IDs. To prove that this works, we designed a test case where the ECU is supposed to send out signals with a specific CAN ID depending on which CAN ID the ECU received. For clarity, we decided to use odd CAN IDs for messages that were received by the ECU and even CAN IDs for messages that were sent by the ECU. For each CAN message received, the ECU was programmed to send out a CAN message with a CAN ID adjacent to the one that was received. For example, if the ECU receives a message with the CAN ID 1 it is specified to send out a message with the CAN ID 2, and if the ECU receives a message with the CAN ID 13 it is specified to send out a message with the CAN ID 14, and so on. This is verified by the plots in the results section 4.1. Message_5, for example is set to be transmitted from the PC. As can be seen in the table 4.1, the ECU is programmed to send out Message_6 when Message_5 is received. Message_5 is also supposed to be sent with a period of 3 seconds. Thus, if the CAN communication is set up correctly on the ECU, the plot should show the value of Message_5 at 3 seconds, and also Message_6 since it is immediately sent by the ECU. Figure 4.1 clearly shows that this is the case. Each figure also shows that the same value sent to the ECU is also returned in the corresponding message. This proves that the ECU can send and receive CAN messages of the DLC 5 bytes. The same thing can be verified for the other messages, by checking the table 4.1 and comparing it to the figure.

5.2 Method

The project method was very suitable for this thesis. Since we in the beginning had very little knowledge of AUTOSAR and automotive embedded systems in general, the combined theoretical and practical pre-study served as a very pedagogic way of learning the AUTOSAR concepts and how to apply it in practice, as well serving as a tutorial for the development tools. The AUTOSAR methodology itself had to be simplified to some extent, as it usually covers projects of a larger scale than this thesis. However, to actually realize which methodology steps as well as AUTOSAR modules were needed was of some difficulty, resulting in some situations where the next step of the work flow was unclear.

There were also some inconsistencies in the pre-study phase of the project method. One example of this is that we used HALCoGen-generated code in the example applications we created in Arctic Studio. Code generated in HALCoGen turned out to be not entirely compatible with the Arctic Core embedded platform, and it cost us some extra hours of work for troubleshooting. A better approach would be to still use HALCoGen-generated code for verification of the hardware, but not in the Arctic Studio-generated ECU software.

One main drawback of the project method is that there was no specified test protocol for each project phase. This resulted in inconsistent testing of the functionality during the development phases, meaning that we in many cases had to go back in the work flow to fix errors which could have easily been found at earlier stages.

All in all, the overall outline of the project method was very suitable, but each project phase should have been more detailed, explicitly defining test protocol and similar things.

5.3 Conclusions

This thesis work has resulted in the fulfillment of the requirements specified in chapter 1. All of the requirements have been validated thoroughly and the purpose has been reached.

Furthermore, we believe that our thesis can be of use for people who are trying to learn the methodology and structure of AUTOSAR. As of a whole, the structure has been kept simple which makes it compatible for future expansion.

5.3.1 Future work

As what might be expected of 10 weeks of work, is that there is still room for improvement. Our system meets the specification of requirements in chapter 1. It does not, however, have any practical use in a car as of yet.

Since we have developed a basic platform for CAN communication, the system could in the future be expanded in a wide amount of varieties. The system could be expanded into a control unit for a car, for example the Sinclair C5 described in chapter 1, by introducing additional SWCs to the system. Basically,

our system can serve as the basis for any kind of ECU using CAN communication. The simulator programs CanSim and CanGraph which we have developed are not limited for usage with the ECU we have developed. They could be used in general to simulate and plot CAN communication in any setting. However, it should be noted that these need further development in order to be used in a professional environment. One example is that they pay no regard to error frames. Since these programs were only a small part of the thesis, not a lot of time was spent on developing them. They could probably be better optimized performance-wise, and also be more user-friendly. Thus, in order for these programs to be used in the industry further development would be needed.

List of Figures

Figures 2.1-2.14 are used with permission from the AUTOSAR alliance. The original versions can be found in the documents at http://www.autosar.org/specifications/re
42/

Appendix

A

Appendix A: Flowcharts

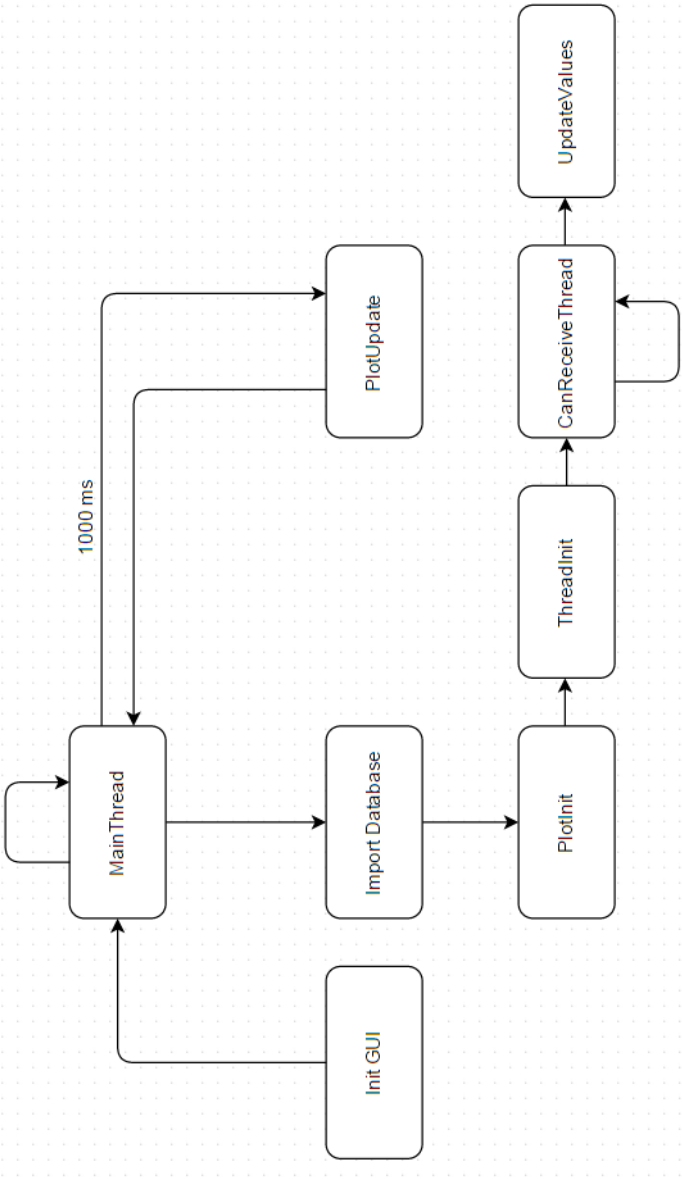

Figure A.1: *Flowchart of CanGraph*

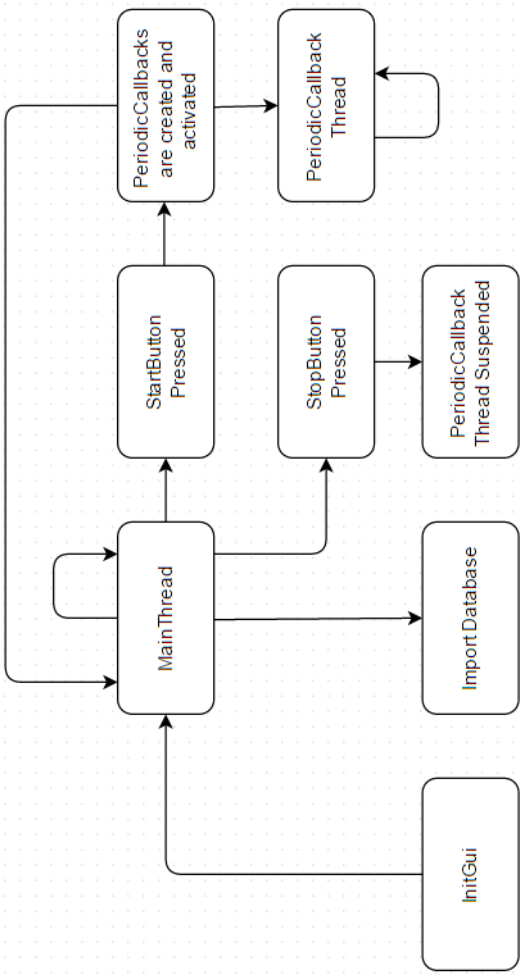

Figure A.2: *Flowchart of CanSim*

Bibliography

[1] Documentation for pyqtgraph. http://www.pyqtgraph.org/documentation/introduction.html#what-is-pyqtgraph, 2011. Cited on page 27.

[2] Documentation for pyqt. http://pyqt.sourceforge.net/Docs/PyQt4/introduction.html, 2014. Cited on page 27.

[3] *About the company ÅF*, 2015. Cited on page 1.

[4] ArcCore. Arctic studio. http://www.arccore.com/products/arctic-studio/for-application-developers, 2015. Cited on page 26.

[5] AUTOSAR, http://www.autosar.org/about/basics/. *AUTOSAR Basics*, 4.2.1 edition, 2014. Cited on pages 5 and 6.

[6] AUTOSAR, http://www.autosar.org/about/basics/background/. *Background of AUTOSAR*, 4.2.1 edition, 2014. Cited on page 5.

[7] AUTOSAR, http://goo.gl/ZSwxo5. *Layered Software Architecture*, 4.2.1 edition, 2014. Cited on pages 6, 7, and 8.

[8] AUTOSAR, http://goo.gl/8bLwPo. *Specification of AUTOSAR Methodology*, 4.2.1 edition, 2014. Cited on page 24.

[9] AUTOSAR, http://goo.gl/RmX67P. *Specification of CAN driver*, 4.2.1 edition, 2014. Cited on page 11.

[10] AUTOSAR, http://goo.gl/k9TktX. *Specification of CAN Interface*, 4.2.1 edition, 2014. Cited on page 12.

[11] AUTOSAR, http://goo.gl/Fq4FaJ. *Specification of Communication*, 4.2.1 edition, 2014. Cited on page 12.

[12] AUTOSAR, http://goo.gl/RyJjqu. *Specification of ECU State Manager*, 4.2.1 edition, 2014. Cited on page 14.

[13] AUTOSAR, http://goo.gl/vCkXiM. *Specification of PDU router*, 4.2.1 edition, 2014. Cited on page 12.

[14] AUTOSAR, http://goo.gl/GC0w6Y. *Specification of Virtual Function Bus*, 4.2.1 edition, 2014. Cited on pages 15 and 17.

[15] Huang Bo, Dong Hui, Wang Dafang, and Zhao Guifan. Basic concepts on autosar development. In *Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on*, volume 1, pages 871–873. IEEE, 2010. Cited on page 17.

[16] BOSCH, http://esd.cs.ucr.edu/webres/can20.pdf. *CAN Specification*, 2.0 edition, September 1991. Cited on pages 19, 20, 21, and 22.

[17] Stefan Bunzel. Autosar - the standardized software architecture. *Informatik-Spektrum*, 34(1):79–83, 2011. Cited on pages 7, 23, 24, and 25.

[18] Johannes Gosda. Autosar communication stack. Master's thesis, Hasso Plattner Institut, http://hpi.de/fileadmin/user_upload/fachgebiete/giese/Ausarbeitungen_AUTOSAR0809/CommunicationStack_gosda.pdf. Cited on pages 10 and 11.

[19] Texas Instruments. Code composer studio. http://www.ti.com/tool/ccstudio, 2015. Cited on page 27.

[20] Texas Instruments. Halcogen. http://www.ti.com/tool/halcogen/, 2015. Cited on page 27.

[21] Hyun Chul Jo, Shiquan Piao, Sung Rae Cho, and Woo Young Jung. Rte template structure for autosar based embedded software platform. In *Mechtronic and Embedded Systems and Applications, 2008. MESA 2008. IEEE/ASME International Conference on*, pages 233–237. IEEE, 2008. Cited on page 17.

[22] OSEK, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf. *OSEK/VDX*, 2.2.3 edition, 2005. Cited on pages 12 and 13.

[23] TexasInstruments, http://www.ti.com/lit/an/sloa101a/sloa101a.pdf. *Introduction to CAN*, July 2008. Cited on page 19.