

## ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Ondřej Kulatý**

Studijní program: Otevřená informatika (magisterský)  
Obor: Počítačové inženýrství

Název tématu: **Autentizace zpráv na sběrnici CAN pro softwarovou architekturu AUTOSAR**

Pokyny pro vypracování:

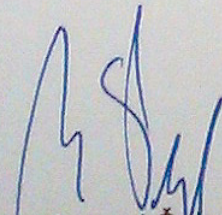
1. Seznamte se protokolem MaCAN [1] a jeho implementacemi pro Linux od firmy Volkswagen (VW) a pro procesor TriCore AUDO MAX vyvinuté na katedře.
2. Najděte nekompatibility mezi zmiňovanými implementacemi a upravte naši implementaci tak, aby byla kompatibilní s VW implementací.
3. Analyzujte o kolik je MaCAN náročnější oproti tradiční CAN komunikaci se zabezpečením pomocí CRC v parametrech jako např. výkon CPU, spotřeba paměti či počet řádků kódu.
4. Navrhněte jak integrovat protokol MaCAN do softwarové architektury AUTOSAR [2] a integrujte naši implementaci s open-source implementací AUTOSARu Arctic Core. Pokuste se, aby byl váš výsledek dostupný v repozitáři projektu Arctic Core.
5. Vše důkladně otestujte a zdokumentujte.

Seznam odborné literatury:

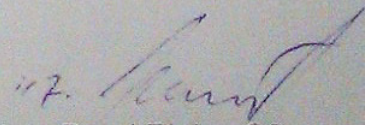
- [1] O. Hartkopp, C. Reuber, R. Schilling: MaCAN - Message Authenticated CAN, ESCAR 2012.  
[2] AUTOSAR specification, release 4.1, Available online: <http://www.autosar.org>

Vedoucí: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

  
prof. Ing. Michael Šepek, DrSc.  
vedoucí katedry

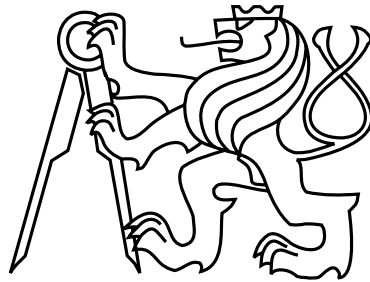


  
prof. Ing. Pavel Ripka, CSc.  
děkan

V Praze dne 22. 1. 2014



Czech Technical University in Prague  
Faculty of Electrical Engineering  
Department of Control Engineering



Master's Thesis

**Message authentication for CAN bus and AUTOSAR software  
architecture**

*Bc. Ondřej Kulatý*

Supervisor: Ing. Michal Sojka, PhD.

Study Programme: Open Informatics

Field of Study: Computer engineering

January 5, 2015



## Acknowledgements

I would like to thank my supervisor, Ing. Michal Sojka, PhD. for his guidance and feedback during the creation of this work and also Ing. Pavel Píša, PhD. for his help. Our thanks also belong to ArcCore for their support.



## Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Litovel on January 5, 2015

.....



# Abstract

The aim of this master's thesis is to integrate an implementation of the MaCAN protocol developed by Department of Control Engineering FEE CTU with the implementation of AUTOSAR architecture developed by ArcCore company. The thesis deals with security on the CAN bus and with the MaCAN protocol, which serves for message authentication on this bus. The implementation of the MaCAN protocol mentioned above and another implementation of the MaCAN protocol from Volkswagen are compared, their incompatibilities are described and resource usage analysis is carried out. Finally, the integration of our MaCAN implementation with AUTOSAR architecture is described and demonstrated with a demo application running on an STM32-based embedded system.

# Abstrakt

Cílem této práce je integrovat implementaci protokolu MaCAN vyvinutou na Katedře řídicí techniky FEL ČVUT do implementace architektury AUTOSAR od firmy ArcCore. Práce se zabývá bezpečností na sběrnici CAN a popisem protokolu MaCAN, který slouží k autentizaci zpráv na této sběrnici. Výše zmíněná implementace protokolu MaCAN a další implementace od firmy Volkswagen jsou porovnány s uvedením popisu jejich vzájemných nekompatibilit a analýzou využití zdrojů. Nakonec je popsán způsob integrace protokolu MaCAN do architektury AUTOSAR a výsledek je otestován pomocí demonstrační aplikace běžící na vestavném systému založeném na platformě STM32.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Aims of this thesis . . . . .	2
<b>2</b>	<b>Related technologies</b>	<b>3</b>
2.1	CAN bus . . . . .	3
2.2	Security problems on CAN bus . . . . .	5
2.3	CAN bus extensions . . . . .	6
2.4	AUTOSAR . . . . .	9
<b>3</b>	<b>MaCAN protocol</b>	<b>11</b>
3.1	Introduction . . . . .	11
3.2	Restrictions on CAN bus . . . . .	11
3.3	Requirements . . . . .	12
3.4	Concept . . . . .	13
3.5	Prerequisites . . . . .	14
3.6	Protocols . . . . .	14
<b>4</b>	<b>MaCAN implementation</b>	<b>21</b>
4.1	CTU MaCAN implementation . . . . .	21
4.2	Volkswagen implementation . . . . .	27
4.3	Testing . . . . .	28
4.4	Compatibility with Volkswagen implementation . . . . .	33
4.5	Porting to STM32 platform . . . . .	35
4.6	Resource usage analysis . . . . .	39
<b>5</b>	<b>Integration into AUTOSAR architecture</b>	<b>41</b>
5.1	AUTOSAR architecture . . . . .	41
5.2	AUTOSAR Communication stack . . . . .	46
5.3	ArcCore and their products . . . . .	51
5.4	Integration of MaCAN into AUTOSAR . . . . .	52
5.5	Implementation of CDD_Macan in Arctic Core . . . . .	55
5.6	Demo application . . . . .	63
<b>6</b>	<b>Conclusion</b>	<b>71</b>
<b>7</b>	<b>List of Abbreviations</b>	<b>75</b>

<b>8 Enclosed CD table of contents</b>	<b>77</b>
<b>Appendices</b>	<b>79</b>
<b>A Guide – Creating demo project in Arctic Studio</b>	<b>81</b>
A.1 Downloading Arctic Core source code . . . . .	81
A.2 Preparing Arctic Studio . . . . .	81
A.3 Creating application model . . . . .	82
A.4 Creating an Ecu Configuration Project . . . . .	85
A.5 Configuring the ECU . . . . .	86
A.6 Generate C files from the configuration . . . . .	113
A.7 Implementation C files . . . . .	114
A.8 Compilation . . . . .	117
A.9 Downloading binary file to target board . . . . .	118

# List of Figures

1.1	Vehicular network in modern a car . . . . .	1
2.1	CAN Network . . . . .	4
2.2	CAN Frame . . . . .	5
2.3	Transmission of data bit for normal CAN and CAN+ protocol [23] . . . . .	7
2.4	CANAuth frame . . . . .	8
3.1	Crypt frame . . . . .	14
3.2	Signal authentication . . . . .	15
3.3	Authenticated signal request message (SIG_AUTH_REQ frame) . . . . .	15
3.4	Crypt frame with 32 bit signature (AUTH_SIG frame) . . . . .	16
3.5	Standard CAN frame with 32 bit signature . . . . .	16
3.6	Session key establishment . . . . .	16
3.7	Challenge message . . . . .	17
3.8	Session key message format (SESS_KEY frame) . . . . .	17
3.9	Session key message format . . . . .	18
3.10	Challenge message . . . . .	19
3.11	Plain time message . . . . .	19
3.12	Signal authentication . . . . .	20
3.13	Authenticated time message . . . . .	20
4.1	CAN network in the 4signal demo . . . . .	30
4.2	CAN network in the volkswagen demo . . . . .	32
4.3	Connection of the STM3210C-EVAL to the host computer . . . . .	35
5.1	AUTOSAR top-level architecture overview . . . . .	42
5.2	Software Components interconnected with the VFB[5] . . . . .	42
5.3	Basic Software (BSW) layer hierarchy . . . . .	44
5.4	AUTOSAR Methodology overview . . . . .	45
5.5	AUTOSAR CAN Communication stack [3] . . . . .	46
5.6	PCI and SDU inside PDU [3] . . . . .	47
5.7	PDU types [3] . . . . .	48
5.8	Arctic Studio IDE . . . . .	52
5.9	Simplified CAN communication stack with CDD_Macan module . . . . .	54
5.10	Header file hierarchy . . . . .	55
5.11	Nodes in the demo . . . . .	64



# List of Tables

2.1	CAN-FD message formats . . . . .	7
4.1	Keyserver command line options . . . . .	23
4.2	Timeserver command line options . . . . .	23
4.3	Main configuration structure . . . . .	24
4.4	Members of the <code>macan_sig_spec</code> structure . . . . .	25
4.5	Frame types for signal transmission . . . . .	25
4.6	Members of the <code>macan_can_ids</code> structure . . . . .	26
4.7	Members of the <code>macan_ecu</code> structure . . . . .	26
4.8	Available options for a JSON object describing a node . . . . .	27
4.9	Available options for a JSON object describing a node . . . . .	28
4.10	VW keyserver command line options . . . . .	28
4.11	VW timeserver command line options . . . . .	28
4.12	VW gateway command line options . . . . .	29
4.13	VW gateway command line options . . . . .	29
4.14	Helper scripts available in demos . . . . .	29
4.15	Nodes in the 4signals demo . . . . .	30
4.16	Signals in the 4signals demo . . . . .	30
4.17	Nodes in the volkswagen demo . . . . .	32
4.18	Signals in the volkswagen demo . . . . .	32
4.19	Fixed memory requirements . . . . .	39
4.20	Memory requirements per node . . . . .	40
4.21	Memory requirements per signal . . . . .	40
4.22	Times measured on TriCore TC1798 and on STM32F107VCT . . . . .	40
5.1	Members of the <code>CDD_Macan_ConfigType</code> structure . . . . .	60
5.2	Members of the <code>CDD_Macan_PduConfigType</code> structure . . . . .	60
5.3	Nodes in the demo . . . . .	64
5.4	Signals in the demo . . . . .	64





# List of Listings

4.1	Example of a LTK definition . . . . .	25
4.2	4signals demo configuration . . . . .	30
4.3	ks.json file . . . . .	32
4.4	ts.json file . . . . .	32
4.5	ecuvw.json file . . . . .	33
4.6	signals_ecuvw.json file . . . . .	33
4.7	Custom fputc function . . . . .	37
4.8	Part of the config.target file . . . . .	38
5.1	Function that translates CAN-ID to PDU-ID . . . . .	57
5.2	MySRInterface port-interface . . . . .	65
5.3	Description of <b>TheProducer</b> component . . . . .	65
5.4	Description of <b>TheConsumer</b> component . . . . .	65
5.5	Description of the composition . . . . .	66
5.6	Implementation of a runnable in <b>TheProducer</b> . . . . .	67
5.7	Implementation of a runnable in <b>TheConsumer</b> . . . . .	67
5.8	<b>Os_TaskPeriodic</b> task . . . . .	69
5.9	<b>RteTask</b> task . . . . .	70



# Chapter 1

## Introduction

Automotive industry has made a giant leap ahead over the past few decades. This is also true for electric and electronics subsystems in cars, which used to be as simple as a few wires with lightbulbs and switches. Later, as a new systems and technologies were developed, electronic subsystems became more complex. Even systems which used to be operated with mechanical linkages only, like throttle pedals, are now equipped with electronics and communicate over the network with other systems (this type of control is also called *drive-by-wire*).

A typical modern car includes dozens of computer units called ECU (Electronic Control Unit). These embedded devices are connected to sensors and actuators and are used to operate a broad range of systems from seat heating and mirrors to brakes and engine fuel injection. This increased count of devices in cars implied higher wiring requirements, so it was necessary to develop vehicular bus which would efficiently and reliably interconnect various electric and electronic subsystems in vehicles, which need to communicate in real-time.

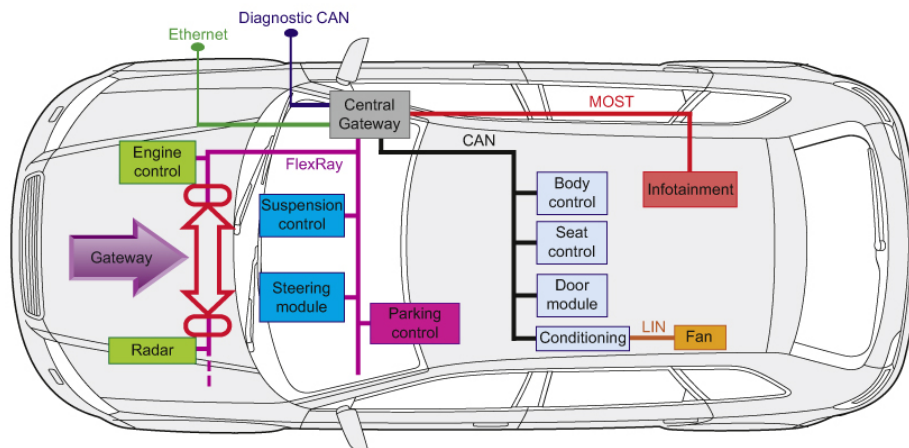


Figure 1.1: Vehicular network in a modern car<sup>1</sup>

<sup>1</sup>Source: <http://goo.gl/O4XOSx>

A high level view of a modern car is illustrated in Fig. 1.1. As can be seen, there are several such a networks which connect related systems together, each of them is based on different bus technology. The most common types of buses in today's car include CAN, LIN, MOST and FlexRay. In this thesis, we focus on the CAN bus and on existing implementations of the MaCAN protocol which is used to increase security of the CAN bus.

In order to reduce cost of the ECU software development and increase its scalability, vehicle manufacturers and suppliers decided to standardize the ECU software functionality by creating an open standard AUTOSAR. In this thesis, our efforts will be directed towards integration of the MaCAN protocol into an open source implementation of the AUTOSAR architecture.

## 1.1 Aims of this thesis

The main focus of this thesis is to:

1. Improve MaCAN documentation by clarifying ambiguities and adding description of message formats, which are missing in the original paper [14] (see Chapter 3)
2. Test implementations of MaCAN protocol from different vendors together and documenting the differences among them (see Chapter 4)
3. Port our MaCAN implementation to STM32 architecture (see Chapter 4)
4. Integrate our MaCAN protocol implementation into AUTOSAR architecture (see Chapter 5)

# Chapter 2

## Related technologies

In this chapter, we describe technologies relevant to this work. First we focus on description of the CAN bus, then we present its security extensions and finally we briefly describe the AUTOSAR architecture.

### 2.1 CAN bus

One of the first buses used in cars was the CAN bus, developed in early 1980s by Robert Bosch, GmbH. Despite of its long history and development of a new buses with higher speeds and better real-time properties, like FlexRay, it is still used frequently today and remains de facto standard in automotive industry.

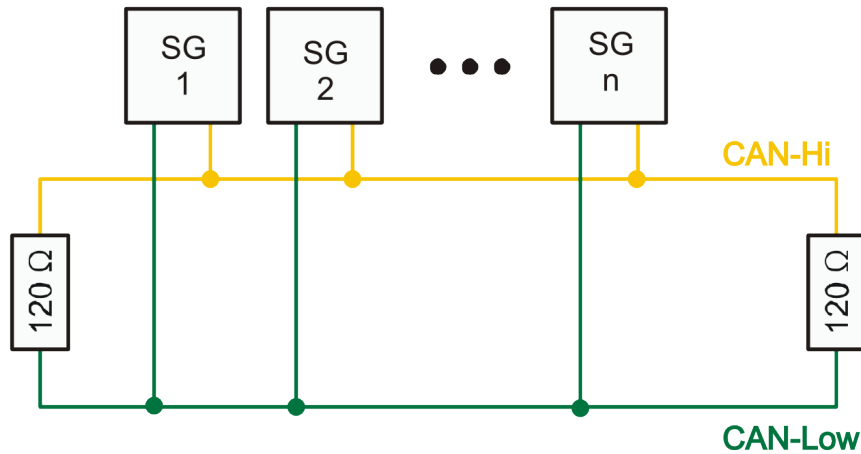
CAN is a relatively simple bus and its specification covers only physical and link layer of the standard ISO/OSI network model, making it a low level protocol. Latest specification of the CAN bus, published in 1991, is CAN 2.0.

#### 2.1.1 Physical layer

CAN network is usually arranged in line topology (as depicted in Fig. 2.1), which simplifies overall design and allows easy addition or removal of ECUs. Logical values are encoded into differential signals, which are defined as voltage difference between *CAN-Lo* and *CAN-Hi*. Log. 1 is recessive (zero voltage difference) and log. 0 is dominant (non-zero voltage difference).

A twisted two-wire line with terminating resistors (with resistance of  $120\ \Omega$ ) is used as a communication medium. These resistors do not only serve to limit wave reflection, but are essential components of the signaling system [21], since they maintain voltage difference between both wires. If two different nodes apply different logical levels on the bus, the dominant level will rule out the recessive one. This is used in bus arbitration described later. Maximum speed defined by the standard is 1 Mbit/s. In practice, both low speed and high speed buses are used together to interconnect subsystems based on their importance and real-time demands.



Figure 2.1: CAN Network<sup>1</sup>

### 2.1.2 Link layer and communication concept

Nodes on the CAN network communicate using CAN frames (format illustrated in Fig. 2.2), which can carry up to 8 bytes of data and are identified by their ID field (Arbitration field). A typical data payload of the frame is a value of some signal and therefore IDs are associated with signals rather than nodes. Value of the ID field also determines priority of the message. There are two frame formats defined in the CAN 2.0 specification — standard frame format with 11 bit CAN-ID (defined in CAN 2.0A) and extended frame format with 29 bit CAN-ID (defined in CAN 2.0B).

Before transmitting, a node must assure, that the bus is idle (in recessive log. level). Then it starts transmitting the frame and senses the bus state simultaneously. If two nodes start transmitting at the same time, the one sending a frame with lower ID will win. The reason is because lower ID always contains a bit which is set to 0, while higher ID will have set this bit to 1. So the node sending frame with higher ID will sense log. 0, while sending this bit, and must stop transmission process and wait until it can try again. This is called the arbitration of the bus and guarantees, that high priority messages are send earlier than low priority ones.

CAN also offers some basic error checking (CRC checksum, disconnection of faulty nodes) and acknowledgement mechanisms, but does not support any security features by itself. This gives rise to security problems and threats associated with networks based on CAN bus. As more electronics systems are responsible for controlling critical systems in today's cars, security of communication between these systems became an important topic recently.

<sup>1</sup>Source: [http://de.wikipedia.org/wiki/Controller\\_Area\\_Network](http://de.wikipedia.org/wiki/Controller_Area_Network)



of every single node on the network, thanks to error tracking mechanism of the CAN bus [22]. This problem is however not as severe, because due to safety concerns, all ECUs must have a fail-safe mode that gets activated when communication fails.

#### 2.2.4 Summary of CAN bus security problems

Putting together, CAN bus is not secure and is prone to various types of attacks. In an extreme case, this may allow hijacking of the car, which might result even in death of passengers. It is not hard and was proven it is possible to take control over steering, brakes and other critical systems in a car by simply sending a high amount of messages (which get accepted by the nodes because they arrive earlier than the legitimate ones) on the CAN bus as described by Miller et al. [16]. In the paper they describe successful attacks on ECUs connected to the CAN bus in Toyota Prius and Ford Explorer.

### 2.3 CAN bus extensions

For security problems mentioned in the previous section, there are standard solutions (like MAC – Message Authentication Code) available. However, this solutions cannot be implemented on the CAN bus directly, because it has many limitations, like maximum message length (8 Bytes). Therefore in this section, we describe existing solutions for bandwidth increase and for message authentication on the CAN bus.

#### 2.3.1 Bandwidth increase extensions

With ever increasing complexity of a systems with many nodes connected together, developers started to hit the limits of the CAN bus. These limits do not only prevent adding further nodes and signals, but also restrict use of a higher protocols, which present an additional overhead.

Main restrictions of the CAN bus are number of identifiers limited to 2048 (applies only for CAN 2.0A with 11 bit CAN-ID), maximum data payload length of 8 bytes and maximum bitrate of 1 Mbit/s.

The bitrate cannot be easily increased, because propagation delay and oscillator inaccuracy would cause errors in the arbitration phase [23]. However when the arbitration phase is over, only one node is transmitting. At this moment, increase of the bitrate is possible. Several extensions for CAN bus take the advantage of this fact to increase the overall bandwidth.

##### 2.3.1.1 CAN-FD

CAN-FD stands for *CAN with Flexible Data-Rate*. It is a new protocol introduced in 2012 by Bosch [11], designed to increase message throughput of the CAN bus. CAN-FD shares physical layer with standard CAN, but introduces new four types of frame formats (see Table 2.1). CAN Base format is compatible with the standard CAN format.

CAN FD formats can carry up to 64 bytes of payload data and use different DLC (Data Length Code) encoding scheme, since its length is only 4 bits, giving 16 possible values.

Frame	CAN-ID length	Data length	Bitrate
CAN Base Format	11 bits	up to 8 bytes	constant
CAN Extended Format	29 bits	up to 8 bytes	constant
CAN FD Base Format	11 bits	up to 64 bytes	dual
CAN FD Extended Format	29 bits	up to 64 bytes	dual

Table 2.1: CAN-FD message formats

Standard CAN uses only the first 9 values (0-8) to encode data length. This remains same in CAN FD, however remaining 7 values are used to encode payload lengths up to 64 bytes. This adds 7 further possible data lengths of 12, 16, 20, 24, 32, 48 and 64 bytes. DLC encoding is controlled by *extended data length* (EDL) bit.

CAN-FD also supports faster bitrates controlled by *bit rate switch* (BRS) bit. Increased alternate bitrate is used only when transmitting data payload. Arbitration phase is always transmitted at the standard bitrate.

### 2.3.1.2 CAN+

CAN+ is another protocol designed to increase bandwidth of the CAN bus described by Ziermann et al. [23]. It uses communication scheme similar to time slots and can increase maximum bitrate of 1 Mbit/s up to 16 times.

This protocol allows sending an additional 15 bits of data per standard data bit in the CAN frame by inserting *overclocked bits* between synchronization zone and sampling zone of the standard CAN bit (see Fig. 2.3). In this *gray zone*, bus can take any value without disturbing standard CAN communication. This guarantees backward compatibility with standard CAN, so it can be used in existing CAN networks.

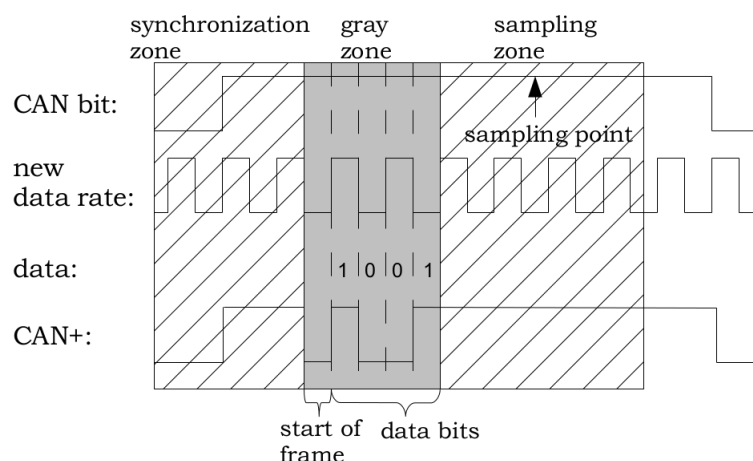


Figure 2.3: Transmission of data bit for normal CAN and CAN+ protocol [23]

### 2.3.2 Security extensions

Security extensions for the CAN bus were developed recently to mitigate some of the risks described in the previous chapter. Most of them focus on message authentication. Although this does not encrypt contents of the CAN frames, it should guarantee the origin of the message and prevent attackers from sending malicious frames. There are already some authentication protocols used in other industry domains, but CAN is a very specific bus with limited resources, so these protocols are often too heavyweight for this purpose.

Messages on the CAN bus have often hard real-time constraints. Introducing an authentication protocol might influence real-time response of the system, since it adds a significant overhead to the message transmission and processing. In the ideal case, the message should be transmitted with its signature in the single CAN frame [18]. Since maximum payload length of the CAN frame is 8 bytes, this presents significant restriction for the protocol designers. Additionally, if the signatures were not transmitted along with the messages, new CAN-IDs would have to be taken. This is not always possible, because there might not be enough free CAN-IDs available in systems with many signals. Use of bi-directional protocols would also rise the number of additional needed CAN-IDs, so most of the already proposed protocols are unidirectional.

If message authentication protocol has to be as backward compatible as possible, it should be compatible with the standard 8 byte frames and standard CAN specification, since new standards like CAN-FD are not widely adopted in the industry yet.

#### 2.3.2.1 CANAuth

CANAuth is a backward compatible, lightweight message authentication protocol for the CAN bus described in the paper by A.V. Herrewége et al. [18]. It is designed to meet basic requirements for message authentication protocol like message authentication, replay attack resistance, group keys and backward compatibility, but should also meet restrictions of the CAN bus. This includes the real-time constraints, message length, limited amount of message IDs and unidirectional communication.

It uses CAN+ protocol to send the authentication data. Since data frames can be as short as 1 byte, maximum length of the authentication message is limited to 15 bytes (120 bits). The CANAuth frame in Fig. 2.4 is used to carry the message authentication data. It is divided into 8 status bits and 112 data payload bits.

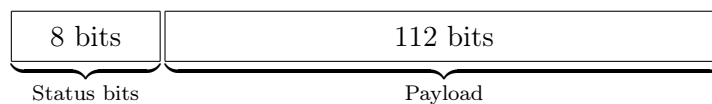


Figure 2.4: CANAuth frame

Each node in a communication group containing one sender and one or more receivers, must store a pre-shared symmetric 128 bit key  $\mathcal{K}_{p_i}$  for each group of related messages in a tamper-proof storage. To communicate, the nodes need to establish a session key  $\mathcal{K}_{s_i}$  in two phases. In the first phase, the sender node sends a 24 bit counter value  $ctr_{A_i}$  and an 88 bit

random number  $r_i$ , which nodes use to generate a session key using the HMAC (Keyed-hash Message Authentication Code ) algorithm:

$$\mathcal{K}s_i = \text{HMAC}(\mathcal{K}p_i, \text{ctr}A_i \parallel r_i) \text{ mod } 2^{128}$$

Before the session key is accepted, the sender node is required to send the second message with a 112 bit session key signature  $\text{sig}A_i$  to prove, that the previous message originated at the trusted sender node:

$$\text{sig}A_i = \text{HMAC}(\mathcal{K}s_i, \text{ctr}A_i \parallel r_i) \text{ mod } 2^{112}$$

Once the session key was established, the messages can be authenticated by sending a CANAuth frame with the 32 bit counter value  $\text{ctr}M_i$  and the 80 bit signature  $\text{sig}M_i$  obtained using HMAC:

$$\text{sig}M_i = \text{HMAC}(\mathcal{K}s_i, \text{ctr}M_i \parallel \text{msg}_i) \text{ mod } 2^{80}$$

The counter value  $\text{ctr}M_i$  is increased with every message being sent and serves as a prevention against replay attacks. Once it overflows, a new session key should be established. Security of the CANAuth protocol relies on the security of the used HMAC algorithm.

The main drawback of this protocol is that it needs a CAN+ capable controllers, which prevents using it in nodes with standard CAN hardware without modification.

### 2.3.3 MaCAN

MaCAN (*Message authenticated CAN*) is a protocol proposed by Volkswagen group researchers Oliver Hartkopp, Cornel Reuber and Roland Schilling in their paper [14]. Unlike CANAuth, it is fully backward compatible with the existing technology. It allows to send authenticated messages to the CAN bus, which can be verified by either a single node or a group of nodes. Messages are signed using CMAC signature which is sent together with data in the same CAN frame. Two new entities are introduced to the system – *keyserver* (KS) and *timeserver* (TS). The keyserver is used to distribute the session keys to other nodes, while the timeserver is used to maintain precise reference clock, which is used when signing the signals to prevent replay attacks. Detailed description of this protocol can be found in Chapter 3 and our implementation is described in Chapter 4.

## 2.4 AUTOSAR

*AUTOSAR* (AUTomotive Open System ARchitecture) is a worldwide development partnership of car manufacturers, suppliers and other companies from the electronics, semiconductor and software industry [2]. Its goal is to develop an open standard of the software architecture and methodology for electronic control units used in automotive industry. It was established in 2002 by BMW, Bosch, Continental, Daimler Chrysler and Volkswagen.

Organization structure is divided into *Core Partners*, who decide about the administration and organization control, *Premium Partners* who define the standard and *Associative*



*members*, who has access to the early development and can utilize AUTOSAR on royalty free basis. In 2011, about 21 million ECUs were produced by AUTOSAR Core partners, with 300 million planned for 2016 [1].

### 2.4.1 Motivation and goals

Complexity of the automotive electronics architecture reached level which requires definition of a new software architecture, which will help to fulfil various requirements imposed on the car manufacturers. These include legal enforcement requirements, passenger convenience and driver assistance and dynamic drive aspects.

Leading OEMs and Tier-1 suppliers decided to cooperate together on the standard, which addresses these requirements and has following goals in mind:

- Standardization of the basic software functionality of automotive ECUs
- Scalability to different vehicle and platform variants
- Transferability of the software between various types of ECU with as little hassle as possible
- Modularity of the software components to enable adapting of the software according to individual needs of ECUs
- Reusability of the software modules

The result of this cooperation is the AUTOSAR software architecture and methodology, described in greater detail in Chapter 5.

Although partners cooperate on standards, they compete on implementation. There are several vendors of AUTOSAR solutions, both commercial and open sourced. One of the main vendors of the open source AUTOSAR solutions is Swedish company ArcCore. It offers AUTOSAR embedded platform called *Arctic Core* and an integrated development environment called *Arctic Studio*.

## Chapter 3

# MaCAN protocol

*MaCAN* – *Message authenticated CAN* is a cryptographic protocol proposed by O. Hartkopp et al. [14] to increase security on the CAN bus. In this chapter, a detailed description of this protocol is provided.

### 3.1 Introduction

MaCAN was created as reaction to continuous growth of interfaces in modern cars, while still using a relatively basic networks based on the CAN bus. These networks are reliable and suitable for real-time systems, but they offer no security features, like message authentication. The protocol is designed to preserve real-time capabilities of the CAN bus and to be easily implemented in a real CAN environment – it is fully backward compatible with existing and deployed technology.

The basic idea is to provide a possibility to send an authenticated messages in a single CAN frame, which can be verified by either a single receiving node or a group of receiving nodes. Either permanent signal authentication or on demand signal authentication is possible.

### 3.2 Restrictions on CAN bus

Several restrictions apply when trying to add a message authentication capabilities to the CAN bus, due to it's properties.

#### 3.2.1 Message length

Standard CAN frame has a payload length of 8 bytes. In order to send an authenticated 32 bit signal, only 4 bytes remain for the signature. Although CAN FD offers payload length up to 64 bytes, MaCAN focuses on the standard CAN specification. The reasons are that CAN FD is not as widespread as the standard CAN is.

### 3.2.2 Available resources

The number of CAN identifiers as well as the bandwidth is limited. Use of a bidirectional protocols on the CAN bus would require  $m \times n$  CAN-IDs, where  $m$  is the number of nodes and  $n$  the number of messages. As number of CAN-IDs is limited, this is a significant restriction for protocol designers.

Bidirectional protocols based on challenge-response authentication also bring significant overhead, that would limit available bandwidth and therefore also real-time capabilities of the bus.

## 3.3 Requirements

Following requirements apply for a security protocol to meet the restrictions outlined above.

### 3.3.1 Message authentication

As the number of ECUs responsible for critical functions in today's cars is increasing, it is necessary to be able to verify the origin of the messages. That means ECUs should not react on forged messages sent by an attacker. Since most messages are highly time dependent, the concept should also provide mechanism which will prevent an attacker from replaying intercepted frames and guarantee freshness of the authenticated messages.

### 3.3.2 Real-time capabilities

Car network is highly real-time system with signals, which need to be delivered within their hard deadlines. The security concept should preserve this real-time properties of the bus. Some ECUs need to respond within milliseconds. If the signal signature would be spread across multiple messages, it is not guaranteed that the ECU would be able to verify origin of the signal within this time frame.

### 3.3.3 Flexibility

It should be possible to apply a message authentication to certain signals, while others may need no protection. Additionally, it should be possible to authenticate individual signals on demand, while preserving their real-time properties.

### 3.3.4 Backward compatibility

The concept should allow concurrent use of non-security equipped nodes with those able to authenticate messages. This is important for implementation in the existing vehicular networks. This means that signals that were previously defined on the network should be accessible to all nodes, even after an authentication element was added to them.

## 3.4 Concept

Due to the limited size of the payload in the CAN frame, Message Authentication Codes (MACs) are used to sign messages. To better utilize hardware implementation, CMAC mode is used. This mode uses a block cipher as a cryptographic element in MAC. It is based on similar idea as symmetric cryptography, therefore all nodes willing to communicate together must share the same group key. MaCAN allows authenticated communication between two nodes as well as a group of nodes. In the group however, all nodes must agree on common trust level, since all share the same group key.

### 3.4.1 MAC truncation

Due to limitations of the maximum payload in the CAN frame and the need to preserve real-time capabilities, MAC signature is truncated to 32 bits. This makes it easier to guess, but the number of guesses an attacker can perform is limited by a relatively low speed of the CAN bus. Short lived session keys are used to additionally increase security level.

CMAC length depends on the block size of the used cipher. In case of AES, a quick block cipher widely implemented in the hardware, it is 128 bits. Level of truncation depends on how many guesses can an attacker perform during the lifespan of the session key. According to [13] CMAC length should satisfy following inequality:

$$T_{len} = lg \left( \frac{MaxInvalids}{Risk} \right) \quad (3.1)$$

$T_{len}$  is the output length of CMAC and  $MaxInvalids$  is the number of attempts an attacker can perform before validity of the session key expires. According to [13] the minimum length of CMAC should be 64 bits. However, thanks to the low speed of the bus and short-lived session keys, it can be truncated to 32 bits. One authentication attempt every 20 ms results in 8640000 attempts during a maximum session key lifetime of 48 hours. According to equation 3.1, 32 bit CMAC results in risk of 1 in 500 of guessing a single signature during the lifetime of the key, yet still it is not guaranteed, that this forged message will be accepted. Attempts at such a high frequency can be recognized by comparing the actual and the expected message frequency on the bus and also by restricting the number of responses to false signatures.

### 3.4.2 Authentication process

Since CAN frames with the same CAN-ID may be used carry more signals, two different types of frames are presented. The first type is a dedicated frame containing only one signal, its ID and its signature as the payload while preserving the original CAN-ID of the frame. This can be used for a limited amount of signals, since more frames need to be send.

The Second type uses a dedicated CAN-ID in addition to an unauthenticated CAN frame and contains a signal and its signature only. This type can be used for the on-demand authentication of usually unauthenticated signals.

### 3.4.3 Message freshness

As described above, use of bidirectional protocols as a way to protect the messages against replay attacks is not an option. Use of a counter values as a source of freshness can also be problematic, since ECUs might be unavailable for a number of reasons. Therefore, in this concept, nodes use synchronized time to protect messages. To maintain synchronized time, a timeserver (TS) is added to the system. It broadcasts current timestamp in a regular intervals and can authenticate them upon request.

## 3.5 Prerequisites

A new identification token for all ECUs participating in a secure communication is defined. This allows to addresses and identify individual nodes. It is defined as a 6 bit identifier called ECU-ID with range from 0 to 63. Every node also shares a symmetric long-term key (LTK) with the keyserver to allow a safe distribution of the session key. Authors of the protocol recommend that the lifespan of the session key is 48 hours.

## 3.6 Protocols

In this section we describe the message format and three protocols used to sign signals, distribute session keys and synchronize time on the CAN bus.

### 3.6.1 Message format

Since the standard CAN frame does not allow direct addressing, a new type of format, called *crypt frame* is introduced (see Fig. 3.1). The CAN frame is not altered, crypt frame only introduces new partitioning scheme of the CAN frame payload.

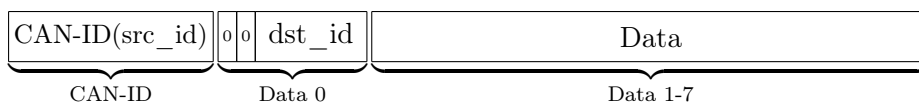


Figure 3.1: Crypt frame

Every node uses a dedicated CAN-ID for the crypt frame, which is derived from its ECU-ID using a lookup table. This allows the receiving node to identify the sender's ECU-ID (src\_id) from the CAN-ID in the received crypt frame. Two flag bits are mapped to the first two most significant bits in the first byte in the payload and are used to distinguish the different message types in the newly introduced protocols. The rest 6 bits are used for the destination node's ECU-ID (dst\_id).





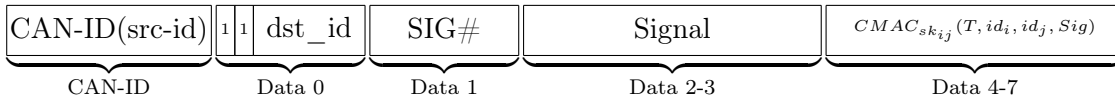


Figure 3.4: Crypt frame with 32 bit signature (AUTH\_SIG frame)



Figure 3.5: Standard CAN frame with 32 bit signature

The key distribution protocol is used to safely distribute session keys along with a new entity called *keyserver* (KS). The key distribution protocol is illustrated in Fig. 3.6. Challenge-response authentication is used to guarantee the authenticity of the keyserver.

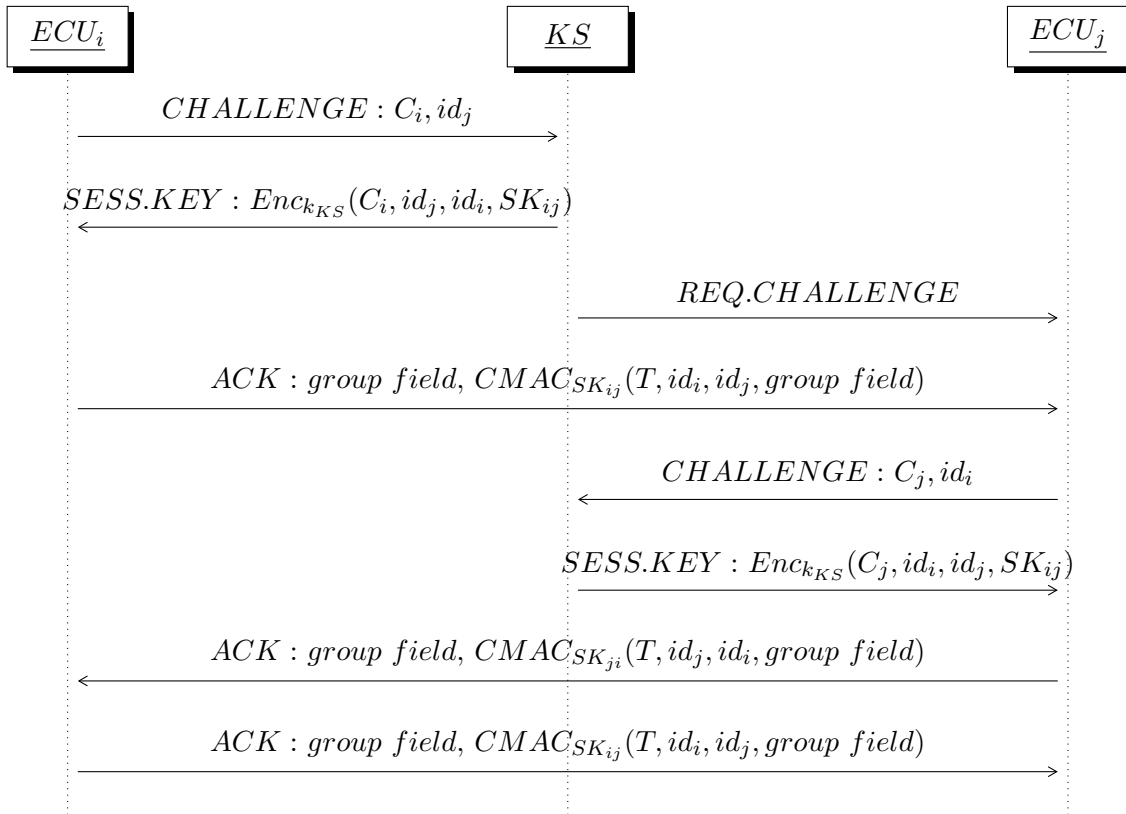


Figure 3.6: Session key establishment

In the original MaCAN protocol specification, CMAC functions inside all ACK messages in the session key establishment procedure are defined as  $CMAC_{SK_{ij}}(T, id_j, group\_field)$ .

This is however a security flaw, which might lead to situation, where  $ECU_i$  believes that the session key has been established, but  $ECU_j$  has not received the session key [12]. We have modified CMAC functions in the ACK messages according to corrections proposed in [12].

### 3.6.3.1 CHALLENGE message

Typically at startup, each node requests a session key from the keyserver for all communication partners or groups it wishes to communicate with. The process of obtaining the key is started by sending a CHALLENGE message, which contains a 6 bit ECU-ID of the communication partner and a 6 byte challenge (see Fig. 3.7).

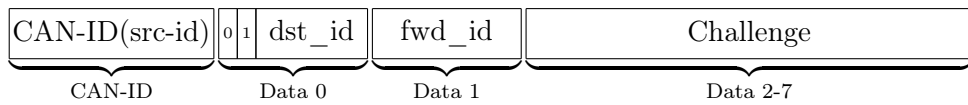


Figure 3.7: Challenge message

The keyserver will then generate the session key, encrypt it (using the long-term key) along with the received challenge and ECU-IDs of both participants in the communication pair and send it back to the requesting node. Keyserver maintains the session keys and generates a new one only if there is no previous session key, or when the current one has expired. If this is not the case, current valid session key is sent upon key requests.

### 3.6.3.2 SESSION KEY message

An encryption key function is used to encrypt a 24 byte long plain text (challenge occupies 6 bytes, both ECU-IDs 2 bytes, session key 16 bytes). However when using AES-Wrap algorithm to encrypt the session key, resulting cypher text will be 32 bytes long, because AES-Wrap divides input text into  $n$  64 bit blocks and the resulting cypher text is  $n + 1$  64 bit blocks long, as defined in [17]. Due to the limited length of the payload, this message cannot be sent in a single CAN frame. Therefore, the message is split into several parts and sent using format depicted in Fig. 3.8. Basically it is a crypt frame discussed earlier with added byte consisting of 4 bit  $SEQ\#$  value, which is a sequence number of the frame and a 4 bit  $MSG\_LEN$  value, which corresponds to the payload length. Given that the maximum payload length of this frame is 6 bytes, six frames must be sent by the keyserver to deliver the wrapped key to the requesting node.

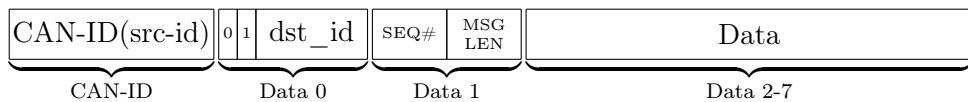


Figure 3.8: Session key message format (SESS\_KEY frame)



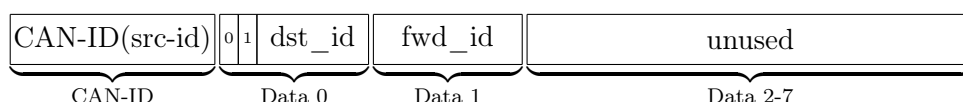


Figure 3.10: Challenge message

their members. This can prevent some less trustworthy device to forge signals of some well protected ECU.

Each group is assigned a 6 bit Group-ID (similar to ECU-ID) and the keyserver as well as every member of the group has the list of the group members as well as their ECU-IDs. If a node requests a session key for the communication with other node, it first checks if this node is a member of the same group. If this is the case, it requests key for the common Group-ID. As the keyserver is aware of the members in all groups, it will send the REQ\_CHALLENGE messages to every member of this group, so all members can request the session key.

Each node in the group still uses its ECU-ID as its `src_id`. Group-ID is used only in `dst_id` and `fwd_id` fields, so nodes need to filter messages on the bus addressed to their own ECU-ID or to the Group-ID.

### 3.6.4 Authenticated time distribution protocol

In order to assure freshness of the authenticated messages and prevent replay attacks, each node is equipped with an internal counter, which maintains current time. This time is then used as one of the inputs to the CMAC function. The problem of counter inaccuracy and synchronization (e.g. after ECU restart or wake up) is solved by adding a new entity called *timeserver* (TS). Its purpose is to maintain and provide a reliable, precise, monotonically increasing time signal to all nodes on the network and provide authenticated time upon request.

Timeserver broadcasts the current 32 bit timestamp to bus in periodic intervals. Every node is able to receive this time signal and compare the timestamp value with its internal counter. If the two values differ more than a certain limit, resynchronization is necessary. This can happen after ECU reboot or just because the internal timers of the ECUs drifted away from each other. The format of the *plain* time message is illustrated in Fig. 3.11.

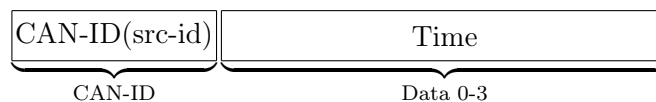


Figure 3.11: Plain time message

Authenticated time distribution protocol is used to request authenticated time from the timeserver and is illustrated in Fig. 3.12.

First, the requesting node sends the challenge message to the timeserver. The format of this message is the same as in the key distribution protocol (see Fig. 3.6) and `fwd_id` is set to 0. Time server then replies with the last broadcasted timestamp together with the

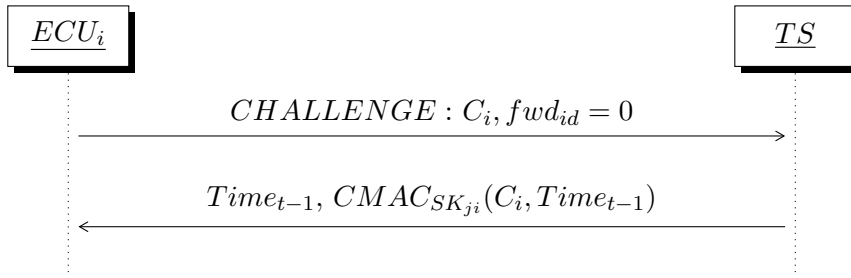


Figure 3.12: Signal authentication

signature (a challenge-response authentication is used since we cannot use time as input for this signature). The format of this authenticated time message is illustrated in Fig. 3.13. Plain time frame and authenticated time frame can be distinguished by different data length code of the message. If an attacker tries to forge a plain time message to alter time of the ECU, he will not be able to reply with the authenticated time message, so ECUs are able to detect this attack and ignore attacker's message.

Group keys should not be used in an authenticated time distribution protocol, since the integrity of the CHALLENGE message cannot be verified by all members of the group. Every node must request signed time individually, using its ECU-ID, when synchronization is necessary.

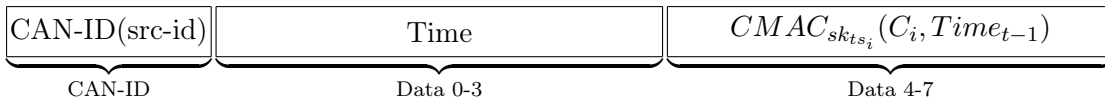


Figure 3.13: Authenticated time message

If there were only one recipient of the time signal or one group with common session key, it would not be necessary to use challenge-response protocol. After the initial challenge and response step, time signal could be broadcasted together with its signature. However, if there were more ECUs, the timeserver would have to broadcast  $n$  different authenticated time messages for every point in time, since every nodes requires its own signature. This would decrease precious available bandwidth on the bus, so authors decided to use the approach described above.

## Chapter 4

# MaCAN implementation

In this chapter, we describe two implementations of the MaCAN protocol: one developed at CTU and another one created by Volkswagen. We also describe how we tested interoperability of these implementations and present a list of found incompatibilities. Later in the chapter, we describe the process of porting of CTU implementation to the STM32 platform. In the end, a memory and CPU usage analysis of the CTU implementation is presented.

During creation of this work, we have contributed to development of the CTU implementation. We have found and fixed several minor bugs, added new features (like support for 32 bit signals) and cleaned up the code.

### 4.1 CTU MaCAN implementation

Our implementation of the MaCAN protocol was created by people from the Department of control engineering at Czech Technical University in Prague and is implemented according to a paper by Oliver Hartkopp and Roland Schilling [14]. The implementation is written in the C language and currently supported platforms include GNU/Linux and Infineon TriCore TC1798. The project is divided into several, relatively independent parts: MaCAN library, keyserver, timeserver and MaCAN monitor. The source code is available in a Git repository at <https://github.com/CTU-IIG/macan>.

#### 4.1.1 License

The main part of the project is incensed under the GNU General Public License, v2.0. There is however some code for the TriCore TC1798 platform, which has a proprietary license and is not included in the repository mentioned above. This code contains a low level routines for handling CAN controllers and hardware cryptographic accelerator/coprocessor SHE (Secure Hardware Extension) on the TriCore TC1798 processor.

#### 4.1.2 MaCAN library

The MaCAN library is a core part of the project and is statically linked to any node (incl. the keyserver and the timeserver) participating in the MaCAN communication. Library's

public API is defined in the `macan.h` file, which is included in every node's source code. The node then calls library functions from its `main` function (or any other functions). The implementation of the node's source code is responsibility of the programmer, who may design it as a command line program or a GUI application etc. The MaCAN library is available on both GNU/Linux and TriCore TC1798 platform.

#### 4.1.2.1 Usage of the library

Before any communication with other nodes can happen, the node must first initialize its hardware (mainly the CAN controller). In order to simplify this task, the MaCAN library contains a low level initialization routines wrapped in the `helper_init` function. This function is platform dependent. On GNU/Linux, it takes as the only argument the name of the CAN interface and returns a file descriptor of a SocketCAN socket. In case of TriCore TC1798, it simply initializes the CAN controller and the SHE (used for AES encryption/description), without taking any input parameter.

Once the hardware is initialized, the application must initialize the library by calling the `macan_init` function. This function requires four input parameters:

- `struct macan_ctx *ctx` – a pointer to a variable holding the “context” structure. This structure is used to hold the configuration and the runtime information. During initialization, this structure is populated with a data needed by the internal functions. The application is responsible for allocating the variable to hold the structure and passing its pointer when calling functions from the library.
- `const struct macan_config *config` – a pointer to a variable holding the static configuration structure. The configuration is described in a greater detail later in section 4.1.6.1.
- `macan_ev_loop *loop` – an event loop used for periodic calls of the functions. The library uses this concept to poll CAN FIFO and to call the internal housekeeping function, which checks if the current session keys are still valid. The application usually uses a predefined macro `MACAN_EV_DEFAULT` as the value of this parameter. Additionally, it may use the event loop for periodic calls of its own functions. The functions can be registered by calling `macan_ev_timer_setup` function.
- `int sockfd` – a file descriptor of the opened SocketCAN socket. Applies only to GNU/Linux, it is ignored on the TriCore TC1798.

After the initialization of the library, the application must register a signal callback functions by calling the `macan_reg_callback` function, in order to receive signals. This function requires a pointer to the context variable, ID of the signal and pointers to two functions. The first one is called upon successful reception of the signal, the second is called when the calculated CMAC differs from the one received with the signal.

To send signals, the application simply calls the function `macan_sen_sig`, which requires a pointer to the context variable, ID of the signal, and the value of the signal. The node may use the event loop to send signals periodically.

Once all previous configuration steps are done, the node starts the event loop by calling the `macan_ev_start` function. The library now automatically handles all tasks needed in the MaCAN communication (like session key exchange, sending and processing of ACK messages, signed time requests, signals signing and verification etc.) without any intervention from the application. The application now interacts with the library only if it wishes to send a signal or the library receives the signal and notifies the application via a callback function.

### 4.1.3 Keyserver

The project contains implementation of the keyserver, which is similar to a normal node, but uses a slightly different initialization and is used to maintain the session keys. The keyserver must know long-term keys (LTK) of all nodes participating in the communication. Under GNU/Linux, configuration and LTK keys are loaded from a shared objects, whose names are passed as a command line parameters (see Table 4.1). This approach allows using the same keyserver binary in multiple projects, without the need to recompile it when the configuration or a LTK key is changed.

Parameter	Description
<code>-c</code>	Path to a shared object with configuration
<code>-k</code>	Path to a shared object with LTK keys of all nodes

Table 4.1: Keyserver command line options

### 4.1.4 Timeserver

The timeserver is very similar to the keyserver described above. It periodically sends plain time and responds to a signed time requests. It also needs the configuration and its LTK key to run (command line parameters are listed in Table 4.2).

Parameter	Description
<code>-c</code>	Path to a shared object with configuration
<code>-k</code>	Path to a shared object with timeserver's LTK key

Table 4.2: Timeserver command line options

### 4.1.5 MaCAN monitor

MaCAN monitor is a command line debugging tool used during development of our implementation. It reads CAN traffic on a specified interface and is able to interpret MaCAN specific frames. Besides printing the contents of a CAN frame, it also colorizes different groups of frames. Because it has access to the configuration (via a `-c` command line option), it is also able to print additional information like sending or receiving node's name. The MaCAN monitor is available on the GNU/Linux platform only.



### 4.1.6 Configuration

Configuration is used to describe configuration of a MaCAN network. It contains a description of nodes participating in the communication, signals being sent between the nodes, LTK keys and configuration options used to initialize the library. The configuration is written directly in C code in a separate file, which is linked into MaCAN applications and also compiled to a shared object for use by keyserver and timeserver (see above). The configuration can be divided into node configuration, signal configuration and protocol configuration.

#### 4.1.6.1 Protocol configuration

Protocol configuration is used to set various communication options of the library and to hold pointers to the previously defined node and signal configurations. Everything is stored in the `macan_config` structure. Members of the structure are summarized in Table 4.3. Most of the members are common to all nodes in MaCAN network, but some are node specific, like `node_id` or `ltk`. These node specific options are set during runtime by each node. The configuration is stored in a separate file and compiled together with the node. Additionally it is also compiled to a shared object, which is needed by the keyserver, the timeserver and the MaCAN monitor. In order to load the configuration from a shared object, the variable holding the main configuration structure must be always named as `config`.

Data type	Struct member	Description
<code>macan_ecuid</code>	<code>node_id</code>	ECU-ID of the node
<code>struct macan_key *</code>	<code>ltk</code>	Pointer to LTK key
<code>uint32_t</code>	<code>sig_count</code>	Number of signals in <code>sig_spec</code>
<code>struct macan_sigspec</code>	<code>sig_spec</code>	Signal specification (see Table 4.4)
<code>uint8_t</code>	<code>node_count</code>	Number of nodes (ECUs) in the network
<code>struct macan_can_ids *</code>	<code>canid</code>	CAN-IDs used by MaCAN (see Table 4.6)
<code>macan_ecuid</code>	<code>key_server_id</code>	ECU-ID of the keyserver
<code>macan_ecuid</code>	<code>time_server_id</code>	ECU-ID of the timeserver
<code>uint32_t</code>	<code>time_div</code>	Number of $\mu s$ in one MaCAN time unit
<code>uint64_t</code>	<code>skey_validity</code>	Session key expiration time [ $\mu s$ ]
<code>uint32_t</code>	<code>skey_chg_timeout</code>	Session key request timeout [ $\mu s$ ]
<code>uint32_t</code>	<code>time_timeout</code>	Authenticated time timeout [ $\mu s$ ]
<code>uint32_t</code>	<code>time_delta</code>	Max. time difference from TS time [ $\mu s$ ]

Table 4.3: Main configuration structure

#### 4.1.6.2 LTK keys

Long term keys are used by nodes to request a session key from the keyserver. They are stored in the `macan_key` structure, which contains a 16 byte data section for holding the key. Each key is defined in a separate file and compiled together only with the node it belongs to. This is done for a security reasons, so other nodes cannot get access to session keys, which

do not belong to them. All LTK keys are also compiled to a shared object, which is needed by the keyserver. A LTK key for the timeserver must also be compiled to a shared object and passed via a command line parameter as described earlier. In order to load key from the shared object, its name must be in the form of `macan_ltk_nodeX` where `X` is ECU-ID of the node. An example of an LTK key definition is illustrated in Listing 4.1.

---

```
const struct macan_key macan_ltk_node1 = {
    .data = { 0x28, 0xC9, 0xBF, 0x10, 0x3C, 0x70, 0xAA, 0xE0,
              0x32, 0x08, 0xEA, 0x9B, 0x45, 0x1D, 0x11, 0x6C}
};
```

---

Listing 4.1: Example of a LTK definition

#### 4.1.6.3 Signal configuration

Every signal in a MaCAN network must have an ID assigned. IDs of the signals are assigned in a similar way as ECU-IDs to nodes, i.e. using an `enum` block. Signal properties are specified in an array of `macan_sig_spec` structures (members listed in Table 4.4). The key of each entry in this array is ID of the signal to which the specification belongs. Each signal must have defined ECU-ID of both sender and receiver. The type of the signal and frames used for its transmission are determined according to values of the `can_nsid` and `can_sid` members. The value of 0 specifies that the particular type of the frame is not used for this signal. All possible types of signals are listed in Table 4.5.

Data type	Struct member	Description
<code>uint16_t</code>	<code>can_nsid</code>	Non-secure CAN-ID
<code>uint16_t</code>	<code>can_sid</code>	Secure CAN-ID
<code>uint8_t</code>	<code>src_id</code>	ECU-ID of node dispatching this signal
<code>uint8_t</code>	<code>dst_id</code>	ECU-ID of node receiving this signal
<code>uint8_t</code>	<code>presc</code>	Prescaler

Table 4.4: Members of the `macan_sig_spec` structure

<code>can_nsid</code>	<code>can_sid</code>	Max. signal size	Frame type used to transmit the signal
0	0	16 bit	crypt frame with 32 bit signature (see Fig. 3.4)
0	non-zero	32 bit	standard CAN frame with 32 bit signature (see Fig. 3.5)
non-zero	0	16 bit	standard CAN frame without MAC or crypt frame with 32 bit signature
non-zero	non-zero	32 bit	standard CAN frame without signature or standard frame with 32 bit signature

Table 4.5: Frame types for signal transmission

The first two types of signals in this table are sent always signed, so the `presc` value has no effect. However the last two types use a standard CAN frames without signature. A signed version of the signal is sent along with the unsigned one only every  $x$ -th call of the `macan_send_sig` function, where  $x$  is the value of the `presc` or when requested with `SIG_AUTH_FRAME` (see Fig. 3.3).

#### 4.1.6.4 Node configuration

Every node in the MaCAN network must have an ECU-ID assigned. This is for convenience done with a simple `enum` block, so nodes get a gradually increasing ECU-IDs by default and also the total count of nodes can be easily determined. The `macan_can_ids` structure (members listed in Table 4.6) is used to hold an array of `macan_ecu` structures (members listed in Table 4.7), which map node ECU-IDs to CAN-IDs. ECU-ID is used as index in the array. The structure also contains a member called `time` to specify the CAN-ID of the plain time message, which is different from the CAN-ID used by the timeserver to request a session key from the keyserver.

Data type	Struct member	Description
<code>uint32_t</code>	<code>time</code>	CAN-ID of the plain time message
<code>struct macan_ecu *</code>	<code>ecu</code>	Pointer to ECU-ID to CAN-ID map

Table 4.6: Members of the `macan_can_ids` structure

Data type	Struct member	Description
<code>uint32_t</code>	<code>canid</code>	CAN-ID of the crypt frames sent by the node
<code>const char *</code>	<code>name</code>	Name of the node (used by the MaCAN monitor)

Table 4.7: Members of the `macan_ecu` structure

#### 4.1.7 Compilation on GNU/Linux

On GNU/Linux, MaCAN uses Ocera Make System (<http://rttime.felk.cvut.cz/omk/>). This system is written in GNU Make and uses multiple simple makefiles spread across directories. The root directory for compilation on GNU/Linux is `build/linux`. It contains the top level `Makefile.omk`, which specifies subdirectories to be traversed. These directories contain another `Makefile.omk` files, which specify “programs” and their source files and dependencies. Common settings, like compilation flags are specified in the `config.target` file. Compilation is started by running `make` command. Intermediate results of the compilation (object files) are stored in the `_build` directory and linked binaries are stored in the `_compiled` directory. Libraries, that must be installed on the system include Nettle<sup>1</sup> and libev<sup>2</sup>.

<sup>1</sup>Available at: <http://www.lysator.liu.se/~nisse/nettle/>

<sup>2</sup>Available at: <http://software.schmorp.de/pkg/libev.html>

### 4.1.8 Compilation for TriCore TC1798

Compilation for TriCore TC1798 is done using TASKING VX-Toolset for TriCore, a proprietary IDE based on Eclipse from Altium. This IDE contains everything needed for compilation for the TriCore platform. Our implementation uses a low level CAN drivers from AUTOSAR MCAL implementation from Infineon. The configuration of those drivers is done using EB tresos tool from Elektrotbit. Configuration process results in automatic generation of source files that are included together with MaCAN sources in a project in TASKING VX-Toolset. The IDE produces a single ELF binary which is downloaded to the board via USB. The TASKING VX-Toolset project file is located in the `infineon-proprietary` directory. To import the project, select *File – Import – Existing Projects into Workspace* and navigate to the directory mentioned above.

## 4.2 Volkswagen implementation

Volkswagen provided us with a proof-of-concept implementation that runs on GNU/Linux and contains a keyserver, a timeserver and a gateway. The gateway is used to protect a legacy ECU, that is not capable of MaCAN communication. This ECU shares a private CAN bus with the gateway. The gateway adds MACs to frames intercepted on the private CAN bus and forwards them to a public CAN bus, shared with other nodes. Conversely, authentic frames (with valid CMAC) are intercepted on the public CAN bus are forwarded to the private CAN bus as plain messages.

### 4.2.1 Configuration

The configuration is done using a combination of plain text configuration files and a command line parameters. There are two kinds of configuration files, one describes nodes and the other describes signals. Configuration files are written in the JSON format and each node and signal is represented by a JSON object (an unordered collection of name/value pairs). Available options for node description are listed in Table 4.8 and for signal description in Table 4.9. Each node must have its own configuration file, but most of its content is the same, except the `ltk` field. For example configuration, see our volkswagen demo described later.

Option	Description
<code>can_id</code>	CAN-ID of the messages sent by this node
<code>bus</code>	Name of CAN bus, where MaCAN communication takes place
<code>crypt_id</code>	ECU-ID of the node
<code>name</code>	Name of the node
<code>group_id</code>	Group-ID of the group
<code>key</code>	LTK shared with this node

Table 4.8: Available options for a JSON object describing a node

Option	Description
<code>owner</code>	ECU-ID of the sending node
<code>src</code>	CAN-ID of the signal on a private CAN bus
<code>dst</code>	CAN-ID of the signal on a public CAN bus
<code>sigid</code>	ID of the signal
<code>startbyte</code>	First byte of bytes to be signed
<code>length</code>	Length of bytes to be signed
<code>orig_length</code>	Length of the signal in an unsigned CAN frame

Table 4.9: Available options for a JSON object describing a node

### 4.2.2 Command line tools

The keyserver, the timeserver and the gateway are command line tools, that accept command line parameters listed in Tables 4.10, 4.11 and 4.12. The gateway is controlled with commands from keyboard (listed in Table 4.13).

Parameter	Description
<code>-l</code>	Time before the session key expires in sec.
<code>-f</code>	Path to a node configuration file
<code>-k</code>	ECU-ID of the keyserver

Table 4.10: VW keyserver command line options

Parameter	Description
<code>-l</code>	Time before the session key expires in sec.
<code>-t</code>	ECU-ID of the timeserver
<code>-f</code>	Period of the time signal in sec.
<code>-c</code>	Path to a node configuration file

Table 4.11: VW timeserver command line options

## 4.3 Testing

We have created a set of demos to test the functionality of our MaCAN implementation and its compatibility with the Volkswagen implementation. Demos are located in the `demos` directory and each demo contains implementation of node(s) which use the MaCAN library, a configuration and helper scripts.

On GNU/Linux, all demos will be compiled by default by the `make` command invoked from the `build/linux` directory. The compilation produces multiple binaries for each communication node in the demo and common keyserver and timeserver binaries. Each demo

Parameter	Description
-t	Time before the session key expires in sec.
-o	Own ECU-ID of the node
-b	Public CAN interface
-e	Private CAN interface
-k	Key exchange CAN interface
-c	Path to a node configuration file
-s	Path to a signal configuration file
-r	Request signals (sig_id:prsc)

Table 4.12: VW gateway command line options

Command	Description
key:01	Request session key for communication with node, whose ECU-ID = 1
time	Request signed time
req_auth:0c:1	Authorized request for signal ID = 0c with prescaler = 1

Table 4.13: VW gateway command line options

contains the `test` subdirectory with a set of helper scripts written in BASH (see Table 4.14), which can be used to launch the nodes, the keyserver and the timeserver without worrying about command line parameters or location of the binary files. All nodes can run on one computer even without presence of a physical CAN bus. This is achieved by using a virtual CAN interface.

For TriCore TC1798, only one demo can be compiled at a time, so files from others demos must be manually excluded from the build. Since there is no operating system implemented for this platform, only one node can be running on the board.

Script	Description
init-vcn.sh	Initializes a virtual CAN interface
init-can.sh	Initializes a physical CAN interface
ks.sh	Launches the keyserver
ts.sh	Launches the timeserver
nodeX.sh	Launches node with ECU-ID = X

Table 4.14: Helper scripts available in demos

### 4.3.1 4signals demo

The 4signals demo was created to test all possible signal types listed in Table 4.5. This demo contains nodes listed in Table 4.15 and signals listed in Table 4.16. The CAN network is illustrated in Fig. 4.1. There are two 32 bit and two 16 bit signals with different prescalers being sent from NODE1 to NODE2. Configuration of the demo is specified in the

macan\_config.c file (see Listing 4.2).

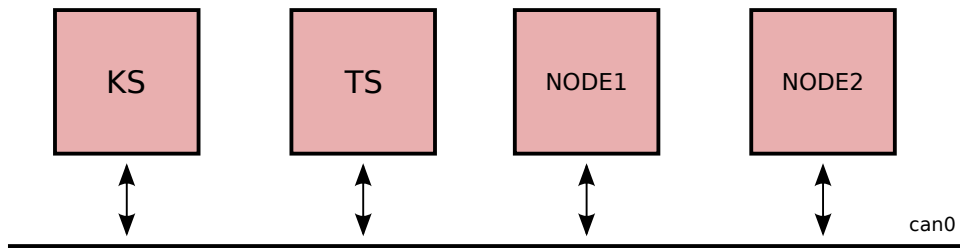


Figure 4.1: CAN network in the 4signal demo

Node name	Crypt-frame CAN-ID	ECU-ID
KS	0x101	0x0
TS	0x102	0x1
NODE1	0x103	0x2
NODE2	0x104	0x3

Table 4.15: Nodes in the 4signals demo

Signal	ID	Source	Destination	Non-secure CAN-ID	Secure CAN-ID
SIGNAL_A	0	NODE1	NODE2	none	none
SIGNAL_B	1	NODE1	NODE2	none	0x202
SIGNAL_C	2	NODE1	NODE2	0x113	none
SIGNAL_D	3	NODE1	NODE2	0x114	0x204

Table 4.16: Signals in the 4signals demo

---

```

enum node_id {
    KEY_SERVER, TIME_SERVER, NODE1, NODE2, NODE_COUNT
};
enum sig_id {
    SIGNAL_A, SIGNAL_B, SIGNAL_C, SIGNAL_D, SIG_COUNT
};

const struct macan_can_ids demo_can_ids = {
    .time = 0x000,
    .ecu = (struct macan_ecu[]){
        [KEY_SERVER] = {0x100, "KS"},
        [TIME_SERVER] = {0x101, "TS"},
        [NODE1] = {0x102, "N1"},
        [NODE2] = {0x103, "N2"},
    },
};

```

```

struct macan_sig_spec demo_sig_spec[] = {
    [SIGNAL_A] = {.can_nsid = 0, .can_sid = 0,
                 .src_id = NODE1, .dst_id = NODE2, .presc = 0},
    [SIGNAL_B] = {.can_nsid = 0, .can_sid = 202,
                 .src_id = NODE1, .dst_id = NODE2, .presc = 0},
    [SIGNAL_C] = {.can_nsid = 113, .can_sid = 0,
                 .src_id = NODE1, .dst_id = NODE2, .presc = 2},
    [SIGNAL_D] = {.can_nsid = 114, .can_sid = 204,
                 .src_id = NODE1, .dst_id = NODE2, .presc = 5}
};
struct macan_config config = {
    .sig_count      = SIG_COUNT,
    .sigspec        = demo_sig_spec,
    .node_count     = NODE_COUNT,
    .canid          = &demo_can_ids,
    .key_server_id  = KEY_SERVER,
    .time_server_id = TIME_SERVER,
    .time_div       = 1000000,
    .skey_validity  = 60000000,
    .skey_chg_timeout = 5000000,
    .time_timeout   = 1000000,
    .time_delta     = 1000000,
};

```

---

Listing 4.2: 4signals demo configuration

### 4.3.2 1signal demo

The 1signal demo is based on 4signals demo described above, but contains only one 32 bit signal (SIGNAL\_B). It was created as a simplest possible configuration, used mainly in early stages of porting to a new platforms. Later, to test both directions of the communication, we have added a second signal of the same type, but sent from NODE2 to NODE1.

### 4.3.3 Volkswagen demo

The Volkswagen demo was created to test our implementation of the MaCAN protocol with the prototype implementation from Volkswagen. We have created a test communication scenario (see Fig. 4.2), with nodes listed in Table 4.17 and signals listed in Table 4.18. A blue colored nodes are from the Volkswagen implementation and the only red colored node is from our implementation. As a substitution for an additional node representing the legacy ECU connected to a private `vw_can` bus, we have used tools from the `can-utils` package. The `candump` tool is used to read CAN frames and the `cangen` is used to generate CAN frames.



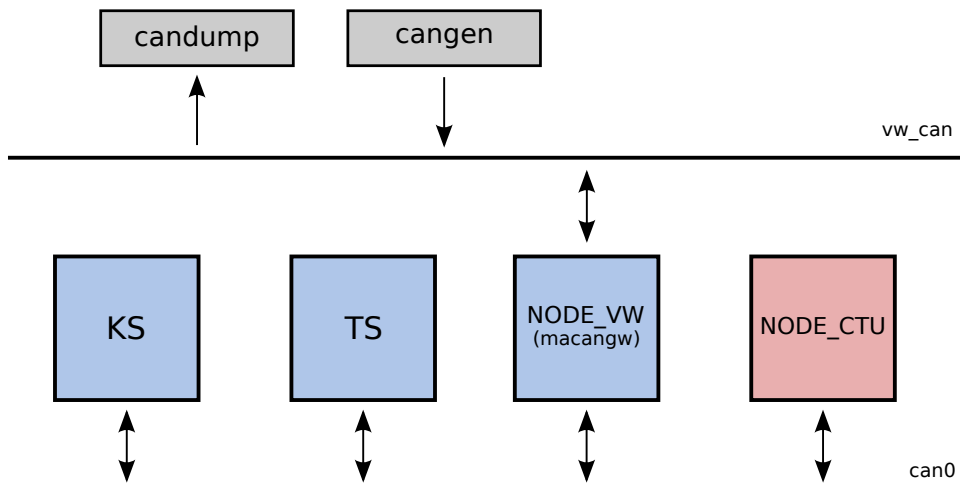


Figure 4.2: CAN network in the volkswagen demo

Node name	Vendor	CAN-ID	ECU-ID	Config files
KS	VW	0x101	0x1	ks.json
TS	VW	0x102	0x2	ts.json
NODE_VW	VW	0x103	0x3	ecuvw.json and signals_ecuvw.json
NODE_CTU	CTU	0x104	0x4	macan_config.c

Table 4.17: Nodes in the volkswagen demo

Signal	ID	Source	Destination	Non-secure CAN-ID	Secure CAN-ID
SIGNAL_VW	0	NODE_VW	NODE_CTU	none	0x2C1
SIGNAL_CTU	1	NODE_CTU	NODE_VW	none	0x2D1

Table 4.18: Signals in the volkswagen demo

```

{"can_id": "101", "bus": "any", "crypt_id": "01", "name": "ks", "group_id": "00", "key": ""}
{"can_id": "102", "bus": "any", "crypt_id": "02", "name": "ts", "group_id": "00",
 "key": "F16F3D70A3125F9D72929218C230F37B"}
{"can_id": "103", "bus": "can0", "crypt_id": "03", "name": "ecuvw", "group_id": "00",
 "key": "0C15D394CBCD4518411B6368AF5060D1"}
{"can_id": "104", "bus": "can0", "crypt_id": "04", "name": "ecuctu", "group_id": "00",
 "key": "000102030405060708090A0B0C0D0E0F"}

```

Listing 4.3: ks.json file

```

{"can_id": "101", "bus": "any", "crypt_id": "01", "name": "ks", "group_id": "00",
 "key": "0C15D394CBCD4518411B6368AF5060D1"}
{"can_id": "102", "bus": "any", "crypt_id": "02", "name": "ts", "group_id": "00", "key": ""}
{"can_id": "103", "bus": "can0", "crypt_id": "03", "name": "ecuvw", "group_id": "00", "key": ""}
{"can_id": "104", "bus": "can0", "crypt_id": "04", "name": "ecuctu", "group_id": "00", "key": ""}

```

Listing 4.4: ts.json file

---

```

{"can_id":"101","bus":"any","crypt_id":"01","name":"ks","group_id":"00",
 "key":"0C15D394CBCD4518411B6368AF5060D1"}
{"can_id":"102","bus":"any","crypt_id":"02","name":"ts","group_id":"00","key":""}
{"can_id":"103","bus":"can0","crypt_id":"03","name":"ecuvw","group_id":"00","key":""}
{"can_id":"104","bus":"can0","crypt_id":"04","name":"ecuctu","group_id":"00","key":""}

```

---

Listing 4.5: ecuvw.json file

---

```

{"owner":"03","src":"2C1","dst":"2C1","sigid":"00","startbyte":"0","length":"4",
 "orig_length":"4"}
{"owner":"04","src":"2D1","dst":"2D1","sigid":"01","startbyte":"0","length":"4",
 "orig_length":"4"}

```

---

Listing 4.6: signals\_ecuvw.json file

#### 4.3.3.1 Running the volkswagen demo

To run the demo, we have to start the keyserver, the timeserver and both nodes. Additionally we also need to generate frames with the `cangen` tool and read frames with the `candump` tool. The demo is run with the following commands:

```

./macanks -l 5000 -f ks.json -k 01
./macants -l 5000 -t 02 -k 01 -f 1 -c ts.json
./macangw -o 3 -t 5000 -b can0 -e vw_can -k can0 -c ecuvw.json \
    -s signals_ecuvw.json
./ctu-node
./candump vw_can
./cangen vw_can 2c1 -Di -L4 -g 1000

```

Unlike our node, which automatically requires session keys and sends authorized signal requests, the gateway is controlled via keyboard. To request a session for communication with the timeserver and our node and to request a signal, we need to type (or use input redirection) following commands on a standard input:

```

key:2
key:4
time
req_auth:1:1

```

## 4.4 Compatibility with Volkswagen implementation

During testing, we have found, that both implementations contain mutual incompatibilities, which prevent using these two implementations together without modifications. Some incompatibilities were caused by bugs in our implementation, but some were also caused because the Volkswagen implementation is not implemented exactly according to specification in [14]. In order to make these two implementations operate together, we had to find

the incompatibilities and modify the code of our implementation. We have introduced a new macro `VW_COMPATIBLE` which, when defined, puts our implementation into compatibility mode with the Volkswagen implementation. Incompatibilities found during the testing are described in the following subsections.

#### 4.4.1 Incorrect value of `MSG_LEN` field

In the `SESS_KEY` message frame sent by the keyserver, there is a field named `MSG_LEN` which contains length of `DATA` payload in this frame. The keyserver sends incorrect value of this field in the last (sixth) frame. In this last frame, there are only 2 bytes of data, so the value of `MSG_LEN` field should equal to 2, however the VW keyserver sets it to 6.

#### 4.4.2 Different order of fields in wrapped session key

The keyserver wraps session keys using AES-Wrap algorithm and sends it in multiple `SESS_KEY` frames to nodes. The actual input to the keywrap function is not only the session key, but also other data. The paper specifies the order of fields to be the following:  $(C_i, id_j, id_i, SK_{i_j})$ . Volkswagen implementation wraps all these fields, but in different order:  $(SK_{i_j}, C_i, id_j, id_i)$ .

#### 4.4.3 Different order of fields when computing CMAC

CMAC is used to sign frames containing data which are required to be secure. The CMAC algorithm expects the data and a session key on input and outputs the message authentication code, which is stored into the CAN frame. Volkswagen implementation uses different input data for the CMAC function, when sending these two types of frames:

- **Signed time frame** (see Fig. 3.13)  
Paper:  $CMAC(C, Time)$ , implementation:  $CMAC(Time, C, CanID_{ts})$
- **Secure 32bit signal** (see Fig. 3.5)  
Paper:  $CMAC(Time, id_i, id_j, Sig)$ , implementation:  $CMAC(Sig, Time, CanID_{sig})$

$CanID_{ts}$  is the CAN-ID of the timeserver and  $CanID_{sig}$  is the secure CAN-ID of the signal.

#### 4.4.4 Missing CMAC field in signal request message

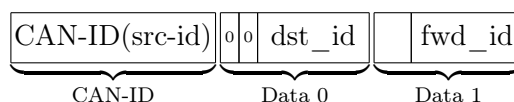
The CMAC is completely missing in the signal requests from `macangw`.

#### 4.4.5 Absence of ACK messages

Node, which received the session key, should ensure that all communication partners already have session keys. According to [14] this should be accomplished by ACK messages. Volkswagen implementation sends no ACK messages after receiving the session key.

#### 4.4.6 REQ\_CHALLENGE frame format

REQ\_CHALLENGE is a message sent by the keyserver. However format of this message is not specified in [14]. Volkswagen implementation of MaCAN sends this message in the following format:



### 4.5 Porting to STM32 platform

Porting of our implementation to the STM32 platform was necessary before its integration to Arctic Core. Initially, we planned to integrate MaCAN to Arctic Core running on GNU/Linux, because our MaCAN implementation is already ported to GNU/Linux. After exploration of Arctic Core source code it however turned out, that we won't be able to run it on GNU/Linux. Officially supported platforms are Power PC and ARM only and porting to GNU/Linux is probably in a very early development stage. This was a problem since we had no hardware to run Arctic Core on. Luckily, we managed to obtain STM3210C-EVAL development board from STMicroelectronics, which is a complete development platform for STMicroelectronic's ARM Cortex-M3 core-based STM32F107VCT microcontroller. It features RS-232 serial line, USB OTG, Two CAN 2.0A/B controllers, JTAG interface, 4 buttons, 4 LEDs and a 3.2" 320x240 TFT color LCD with touchscreen. This board is directly supported in Arctic Studio, i.e. there are configuration templates for the driver modules available, which respect the peripherals layout on the development board.

The development board is connected to the host computer through three distinct connections, each serving a different purpose (see fig. 4.3).

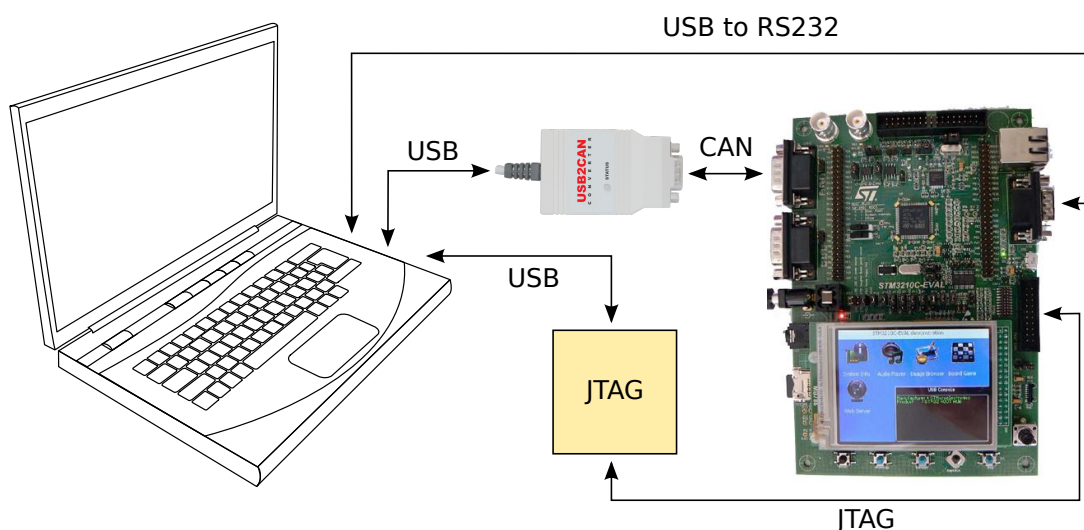


Figure 4.3: Connection of the STM3210C-EVAL to the host computer

First of all, there is a JTAG emulator (with well known FTDI FT2232D IC). This device is connected to a 20-pin JTAG connector on the development board and to the USB on the computer. It allows binary executables to be downloaded to the target board using OpenOCD and also debugging with the GDB. We had to write our own configuration script for OpenOCD, which can be found in the end of Appendix A.

CAN communication is directed through a CAN to USB converter. Once specific drivers in the Linux kernel are enabled, a new CAN interface appears. This interface can be accessed through standard PF\_CAN socket interface. There is also available a set of tools in the `can-utils` package, which allows to view and send CAN frames on the interface.

Finally, we used a standard serial line routed through RS232 to USB converter. This link is used print normal and debugging messages on the serial console. The most common serial consoles are `minicom` and `ckermit`.

Since our MaCAN implementation was not ported to the STM32 architecture, we have decided to port it and run it on the STM3210C-EVAL development board alone, prior to its integration to Arctic Core. Although most of the low level tasks like CAN controller management etc. will be handled after the integration a by low level drivers in Arctic Core, we wanted to make sure, that this board is able to run MaCAN and avoid future problems during the integration process. Thanks to a good organization of the MaCAN source code (platform dependent code separated from platform independent code), porting to a new architecture was not so hard.

### 4.5.1 Directory organization

Platform dependent code is located in `macan/src/XXX` directory, where `XXX` is the name of the platform. We have created a new subdirectory `stm32` to hold the platform dependent code. Additionally, we added a few needed libraries used in the implementation to the root directory of the MaCAN project.

### 4.5.2 Used libraries

To make porting simpler, we have used STM32F10x Standard Peripherals Library from STMicroelectronics (located in `stm32/lib`). This library allows easy access and configuration of onboard peripherals and is also used in Arctic Core, although some parts of the library are modified there.

Cryptographic functions are part of the Nettle library, which is dynamically linked to MaCAN applications on the GNU/Linux system. As dynamic linker is not available on the STM32 platform, it was necessary to compile the library from the source code and link it statically. The Nettle library also had to be stripped to the bare minimum so that the whole binary would fit into limited flash memory of 256 kB on the microcontroller. The stripped down version of the Nettle library is in the `nettle` directory.

### 4.5.3 Initialization

MaCAN initialization sequence is handled by a platform dependent function `helper_init`. Its task is mainly to activate CAN communication subsystem and activate the Secure Hardware Extension (in case of Tricore). In case of our development board, at least three tasks

must be done — CAN controller initialization, timer initialization and serial port initialization.

CAN initialization involves enabling clocks for GPIO and CAN peripherals, configuring output pins and setting up CAN baud rate to 125 kbps. This was done according to an example from the STM32F10X Standard Peripherals library. There is no need to set up interrupts, since we use polling<sup>3</sup> to read received frames.

To be able to measure time, a timer must be running. We used timer TIM2 for this purpose and set its prescaler to 300 and period to 2400. Because input clock for this timer is running at 72 MHz, we get interrupt every 0.01 seconds. There is a global variable which gets incremented in the interrupt routine and is used by time reading functions.

Serial line is needed to print messages on a serial console. We have used USART2 configured to speed of 115200 b/s. We also had to implement our own `fputc` function (see Listing 4.7) to be able to print formatted messages via `printf` functions.

---

```

int fputc(char ch, void *f)
{
    USART_SendData(EVAL_COM1, (uint8_t) ch);

    /* Loop until the end of transmission */
    while (USART_GetFlagStatus(EVAL_COM1, USART_FLAG_TC) == RESET)
    {}

    /* if printing new line, add \r to print nicely on serial console */
    if (ch == '\n') {
        fputc('\r', NULL);
    }
    return ch;
}

```

---

Listing 4.7: Custom fputc function

#### 4.5.4 Transmitting and receiving CAN frames

Transmitting of the CAN frames is done using function `macan_send`. It fills a library structure `CanTxMsg` with data from the CAN frame received as a parameter and sends it via `Can_Transmit` function.

Receiving is done by periodically calling `poll_can_fifo` function, which tests if there is a pending CAN frame. If so, it retrieves the data using `CAN_Receive` function and copies them to a location given by a parameter.

#### 4.5.5 Compiler and make rules

Compilation is done using GNU Tools for ARM Embedded Processors<sup>4</sup>. To compile MaCAN with this toolchain, a new configuration for the OMK make system had to be created and

<sup>3</sup> Polling is preferred over interrupt based deliver in safety critical applications.

<sup>4</sup> Available at: <https://launchpad.net/gcc-arm-embedded>

some `Makefile.omk` files were modified. The `config.target` file for the STM32 platform is the following:

---

```
CC=arm-none-eabi-gcc
AR=arm-none-eabi-ar

CFLAGS = -ggdb
CFLAGS += -std=c99
CFLAGS += -O0
CFLAGS += -Wall -Wextra -Warray-bounds -Wconversion
CFLAGS += -mthumb -mcpu=cortex-m3
CFLAGS += -ffunction-sections -fdata-sections
CFLAGS += -DUSE_STDPERIPH_DRIVER -DSTM32F10X_CL
CFLAGS += -DUSE_STM3210C_EVAL
CFLAGS += -D__CPU_STM32F107__
CFLAGS += -DMALLOC_HEAP_SIZE=0x4000

LDFLAGS = -T$(MAKERULES_DIR)/stm32/stm32_flash.ld -Wl,-Map=application.map -
Wl,--gc-sections $(CFLAGS)
```

---

Listing 4.8: Part of the `config.target` file

The compiler is instructed use Thumb instructions for a target CPU Cortex-M3. The `function-sections` and `data-sections` options are used to reduce the size of the produced executable by discarding unreachable sections. Further, several macros need to be defined. `USE_STDPERIPH_DRIVER`, `STM32F10X_CL` and `USE_STM3210C_EVAL` are used by the STM32F10X Standard Peripheral Library. `__CPU_STM32F107__` is used in the MaCAN code to control inclusion of header files. Finally macro `MALLOC_HEAP_SIZE` is used by `_sbrk` function needed by memory allocation functions.

#### 4.5.6 Test using 1signal demo

To prove, that everything works as expected, we run the 1signal demo. Keyserver, timeserver and one of the nodes were running on the computer with GNU/Linux, while the second node was running on the development board. Later we also tried to swap the nodes to test both receiving and sending of signals.

During early testing, we encountered a strange problem. The session key request from the node running on the development board was successfully received by the keyserver, but the node was unable to receive the session key. We later found out, that this was caused by a loss of CAN frames. The CAN peripheral on the microcontroller has only 3 mailboxes, so it can hold only 3 CAN frames at a time. If reading of these received CAN frames does not happen quickly enough, some of these frames will be overwritten by other incoming frames. Since session key is split into 6 CAN frames sent one after another, these CAN frames must be read as fast as possible from the mailboxes. During runtime, the MaCAN library outputs various info messages about received frames on the console. This was the cause of the problem. Since we have decided to not use interrupts at all, reading of CAN frames is done in software event loop. If some of the functions processing the CAN frame prints a long string of text, it may take quite long to output all the characters on the serial

line. During this process, reading of CAN frames is not available and received frames waiting in mailboxes are lost.

There are several solutions for this problem, but we decided to simply turn off the problematic messages. After this step, the communication worked as expected and MaCAN was partly prepared to be integrated into Arc Core.

## 4.6 Resource usage analysis

In contrast to a standard CAN communication, the MaCAN protocol imposes additional requirements on memory and CPU resources. In this section, we analyze the usage of these resources in our implementation. This is especially important in automotive mass production, where every cent counts. It is therefore useful to know what would be the price of the increased security of on-board communication.

### 4.6.1 Memory usage

Memory requirements are based on size of the context structure `macan_ctx`, which contains a configuration and runtime information representing the state of the MaCAN library. The size of this structure is partly fixed and partly varies according to number of nodes and signals. It may vary slightly on a different platforms. This is because the C standard only requires size relations between the data types and minimum sizes for each data type. We have used 64 bit GNU/Linux platform for the memory analysis. Fixed memory requirements are listed in Table 4.19. Memory requirements per node are listed in Table 4.20 and per signal in Table 4.21. Total memory usage  $M_{req}$  can be expressed in terms of  $N_{node}$  (number of nodes) and  $N_{sig}$  (number of signals):

$$M_{req} = 331 + N_{node} * 71 + N_{sig} * 29$$

Data type	Size	Description
<code>struct macan_config *</code>	8 bytes	Pointer to configuration structure
<code>struct macan_config</code>	68 bytes	Configuration structure
<code>struct com_part **</code>	8 bytes	Pointer to communication partners vector
<code>struct sig_handle **</code>	8 bytes	Pointer to signal handles vector
<code>struct macan_timekeeping</code>	35 bytes	Time keeping structure
<code>uint8_t keywrap[32]</code>	32 bytes	Storage for wrapped session key
<code>int sockfd</code>	4 bytes	CAN socket file descriptor
<code>macan_ev_loop</code>	8 bytes	Pointer to event loop
<code>macan_ev_can</code>	48 bytes	I/O watcher
<code>macan_ev_timer</code>	48 bytes	Timer watcher
<code>union</code>	64 bytes	Used by a timeserver and a keyserver
	<b>331 bytes</b>	<b>TOTAL</b>

Table 4.19: Fixed memory requirements



Data type	Size	Description
<code>struct macan_ecu</code>	12 bytes	Node configuration
<code>struct com_part</code>	59 bytes	Communication partner structure
	<b>71 bytes</b>	<b>TOTAL</b>

Table 4.20: Memory requirements per node

Data type	Size	Description
<code>struct macan_sig_spec</code>	7 bytes	Signal configuration
<code>struct sig_handle</code>	22 bytes	Signal handle structure
	<b>29 bytes</b>	<b>TOTAL</b>

Table 4.21: Memory requirements per signal

### 4.6.2 CPU usage

MaCAN communication is more computationally demanding than standard CAN communication due to calculation and verification of the CMAC fields in the frames, which must be performed while sending/receiving signed frames. We have measured times of the functions that compute CMAC and unwrap the session key. The time was recorded before and after each function's call and the result was computed as a difference between these recorded times. Every function was measured about 100 times, minimum and maximum values were put in the tables. If the function `macan_check_cmac` fails to verify CMAC with a current time, it will try again by decreasing/increasing one time unit from/to current time. That's why it occupies three rows in the table.

Task	Time (Tricore)		Time (STM32)	
	min	max	min	max
<code>macan_unwrap_key</code>	77.43 $\mu$ s	78.50 $\mu$ s	1570 $\mu$ s	1580 $\mu$ s
<code>macan_sign</code>	4.82 $\mu$ s	4.84 $\mu$ s	130 $\mu$ s	140 $\mu$ s
<code>macan_check_cmac</code> (1 round)	12.42 $\mu$ s	12.57 $\mu$ s	130 $\mu$ s	140 $\mu$ s
<code>macan_check_cmac</code> (2 rounds)	16.96 $\mu$ s	17.15 $\mu$ s	250 $\mu$ s	260 $\mu$ s
<code>macan_check_cmac</code> (3 rounds)	21.58 $\mu$ s	21.78 $\mu$ s	380 $\mu$ s	390 $\mu$ s

Table 4.22: Times measured on TriCore TC1798 and on STM32F107VCT

Tests were done on the TriCore TC1798 board and also on the STM3210C-EVAL board (results listed in Table 4.22). Times measured on the STM32 platform were way more slower than on the TriCore TC1798 platform. The reason is that the TriCore TC1798 processor is not only faster (300 MHz compared to 72 MHz), but mainly uses dedicated Secure Hardware Extension for AES cryptographic routines.

## Chapter 5

# Integration into AUTOSAR architecture

In this chapter we provide a more detailed look on the AUTOSAR architecture and its communication stack, mainly the part dedicated to CAN communication. Then we discuss two possible ways of MaCAN integration into the AUTOSAR architecture followed by a brief overview of ArcCore products. Finally we present implementation details of the integration in Arctic Core and describe a demo used to test the implementation.

### 5.1 AUTOSAR architecture

AUTOSAR defines a layered software architecture illustrated in Fig. 5.1, which is divided into *Software Components* (SWC) representing the application layer, *Runtime Environment* (RTE) and *Basic Software* (BSW). Applications are created by connecting Software Components together. RTE implements communication mechanism and provides access to Basic Software modules (like an operating system, communication stack etc.) for Software Components. We discuss each layer in greater detail in subsequent sections.

#### 5.1.1 Software components (SWC)

Software components represent a basic unit of application software in AUTOSAR architecture and are completely independent of used hardware and infrastructure. Each component encapsulates a part of functionality of the application and communicates with other components via AUTOSAR interface using *ports*. Every port is associated with a *port-interface*, which defines the contract that must be fulfilled by the port providing or requiring that interface. Types of port-interfaces include *Client-server* (The server is provider of operations and clients invoke them), *Sender-receiver* (The sender distributes information to one or several receivers) and several other types of port-interfaces.

All communication between ports is handled by a *Virtual Function Bus* (VFB) — a communication mechanism, that virtually connects ports of SWCs together and depending on the configuration, these virtual connections between Software Components are mapped locally or through a network communication like CAN or FlexRay. This allows for strict

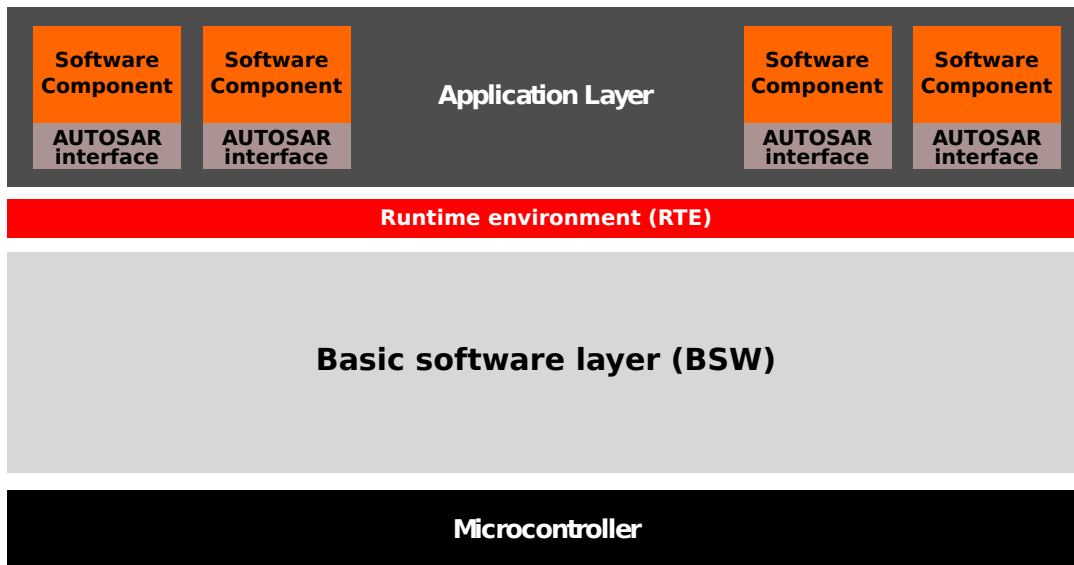


Figure 5.1: AUTOSAR top-level architecture overview

separation between the application and the infrastructure and adds a level of abstraction, which allows relocation of Software Components to different ECUs without modification. Figure 5.2 illustrates several Software Components communicating via VFB.

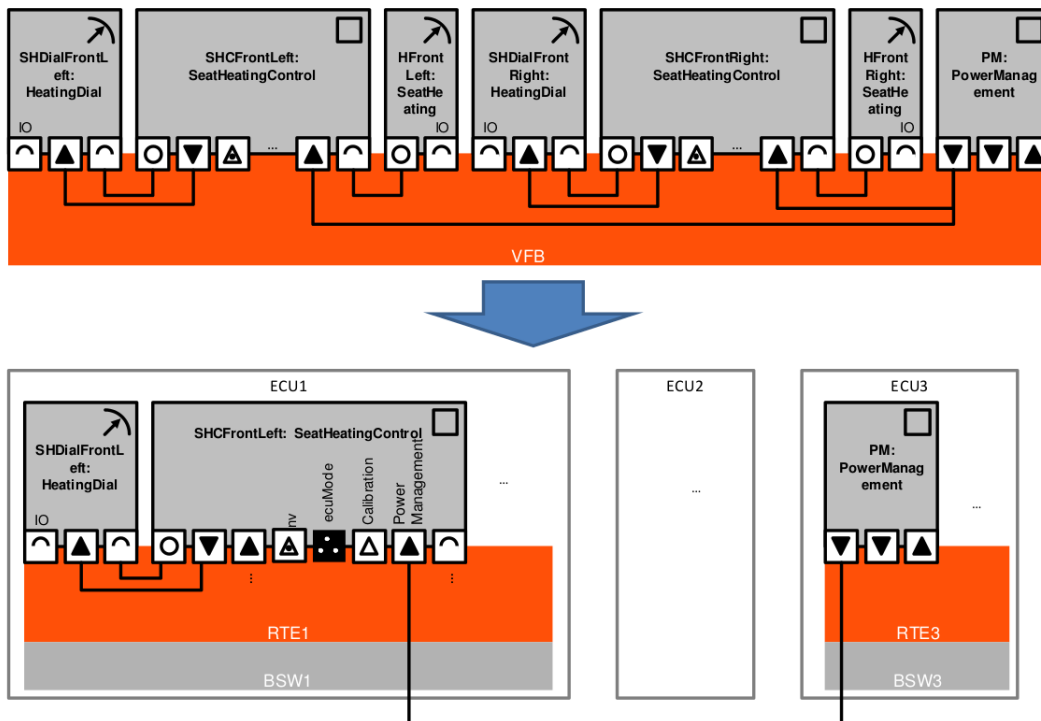


Figure 5.2: Software Components interconnected with the VFB[5]

Software Components are typically further divided into *runnables*, which are small pieces of code executed by RTE (periodically, or triggered by an event). Runnables implementation can be created manually (involves traditional C coding) or from a model, from which the code is generated automatically.

### 5.1.2 Runtime environment (RTE)

The RTE is a central part of the AUTOSAR ECU architecture. It provides a complete environment for Software Components, shields them from lower layers in the Basic Software and realizes the communication interfaces of the Virtual Function Bus. RTE code is automatically generated individually for each ECU. Runnables in Software Components are scheduled by the RTE and use generated functions of the RTE to access data on ports or to call operations of other Software Components.

### 5.1.3 Basic software (BSW)

The Basic software (BSW) is the lowest layer of the AUTOSAR software architecture and contains modules that offer services to Software Components via the RTE. It is further divided into several sublayers as described in [3] (see Fig. 5.3). Most of the Basic Software modules provide only a C-like standardized interface, but some also provide AUTOSAR interface (ports) which can be plugged into Software Components through VFB. Modules in the Basic Software support following configuration classes [3]:

- **Pre-compile time** (`*_Cfg.h`, `*_Cfg.c`) – Static configuration used for enabling/disabling optional functionality of the module (via `#define`).
- **Link time** (`*_LCfg.h`, `*_LCfg.c`) – Configuration of modules that are available only as object code. Configuration data is accessed through external constants.
- **Post-build time** (`*_PbC.h`, `*_PbCfg.c`) – Used for configuration of data where only the structure is defined, but the contents are not known during ECU-build time (e.g. calibration data).

In following section we describe each sublayer of the Basic Software.

#### 5.1.3.1 Microcontroller abstraction layer (MCAL)

The Microcontroller abstraction layer is the lowest layer of the Basic Software. It contains modules called device drivers, which have direct access to the microcontroller. The main task of modules in this layer is to make higher software layers microcontroller independent. An example might be two CAN drivers, which control a different CAN controllers, but provide the same standardized interface to higher layers (but their implementation is hardware dependent).

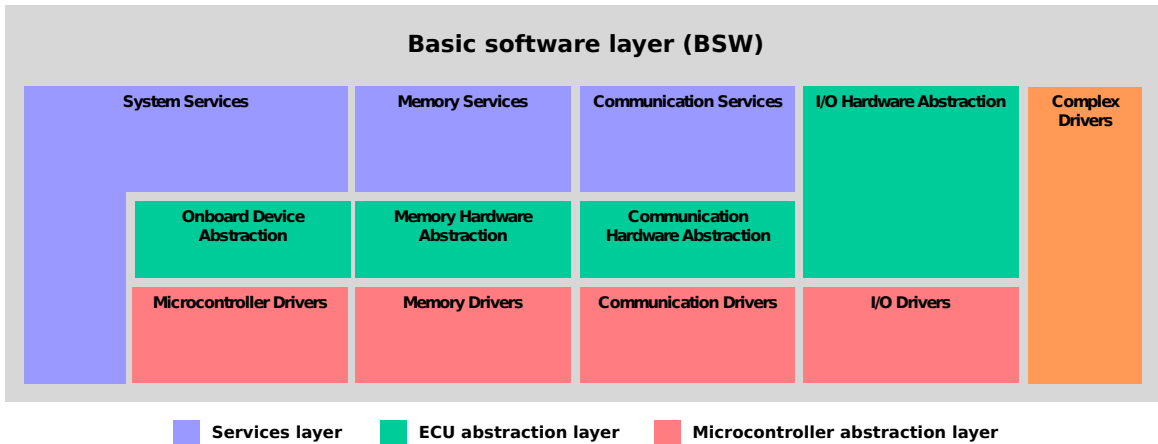


Figure 5.3: Basic Software (BSW) layer hierarchy

### 5.1.3.2 ECU abstraction layer

The ECU abstraction layer is located above the Microcontroller abstraction layer and offers access to peripherals and devices regardless of their location (internal/external to the CPU) and type of connection. Its task is to make higher layers independent of the ECU hardware layout. An example might be the CAN interface module, which provides a generic API to access CAN communications network, independent of the number of CAN controllers on the ECU and their hardware realization.

### 5.1.3.3 Complex Device Drivers (CDD)

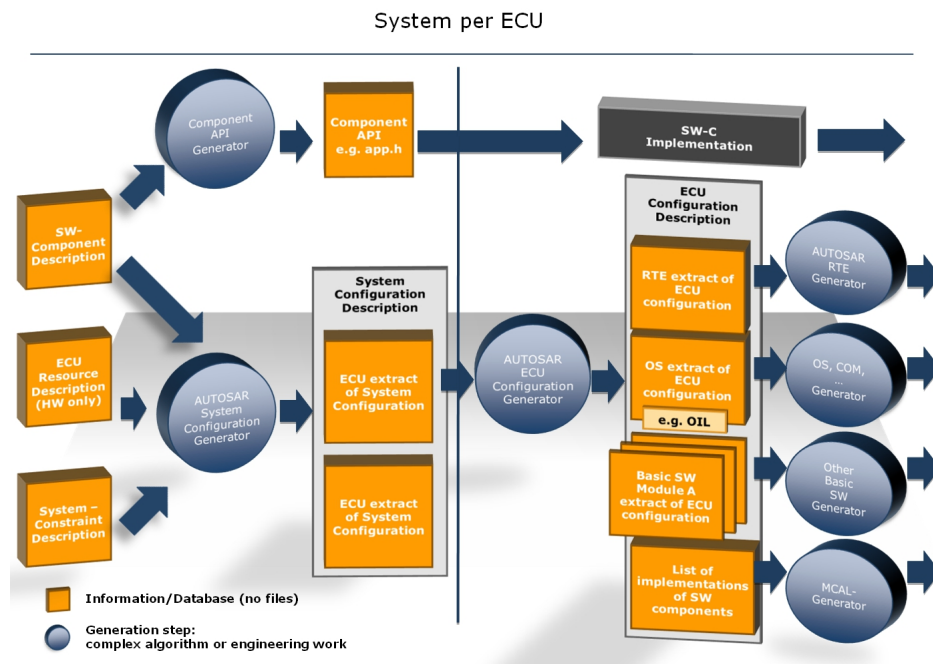
Complex Device Drivers is a special class of modules, which integrate a special purpose functionality. This may include drivers or protocols, which are not standardized in AUTOSAR. As previously seen in Fig. 5.3, CDD modules span from RTE all the way down to the microcontroller hardware and may be accessed by both Basic Software modules (via a standardized interface) and Software Components (via AUTOSAR Interface).

### 5.1.3.4 Services layer

The Services layer is the highest layer of the Basic Software. Its task is to provide services for Software Components and other modules in the Basic Software. Among other functionality it offers an operating system, network communication and management services, diagnostic services, ECU state management, mode management etc. It is mostly microcontroller and ECU independent.

## 5.1.4 Methodology

AUTOSAR defines not only a layered software architecture, but also a software development methodology. This methodology is divided into several phases (see Fig. 5.4). This simplifies the development process and allows independent development of different parts of the system. It also simplifies integration of Software Components from the OEM and Tier-1 suppliers.

Figure 5.4: AUTOSAR Methodology overview<sup>1</sup>

Information between each development phase is exchanged using AUTOSAR XML files and the steps are the following:

1. Vehicle functions are described in terms of Software Components, each described with a *SWC description*.
2. Software Components are put together to form entire functional system, this is called *System description* and it is created using a model-based development tool.
3. Parts of the system (individual Software Components) are distributed to ECUs.
4. For each ECU, an *ECU extract* is generated.
5. Basic software of each ECU is configured based on this extract and a *BSW Module Description*.
6. Finally, code generators are used to generate configurations for Basic Software modules and an ECU-specific RTE layer.

<sup>1</sup>Source: <http://www.autosar.org/index.php?p=1&up=2&uup=4&uuup=0>

## 5.2 AUTOSAR Communication stack

AUTOSAR Communication stack is a group of modules in the Basic Software, which provide communication services to other modules and Software Components. Modules of the communication stack span from the RTE all the way down to the microcontroller hardware. The uppermost communication services layer is common to any type of communication, but modules in the lower ECU hardware abstraction layer and Microcontroller abstraction layer are network protocol specific (different set of modules is used for CAN, FlexRay, Lin etc.).

As this thesis deals with integration of a CAN based protocol into AUTOSAR, we describe modules related to CAN communication only. CAN communication stack is depicted in Fig. 5.5.

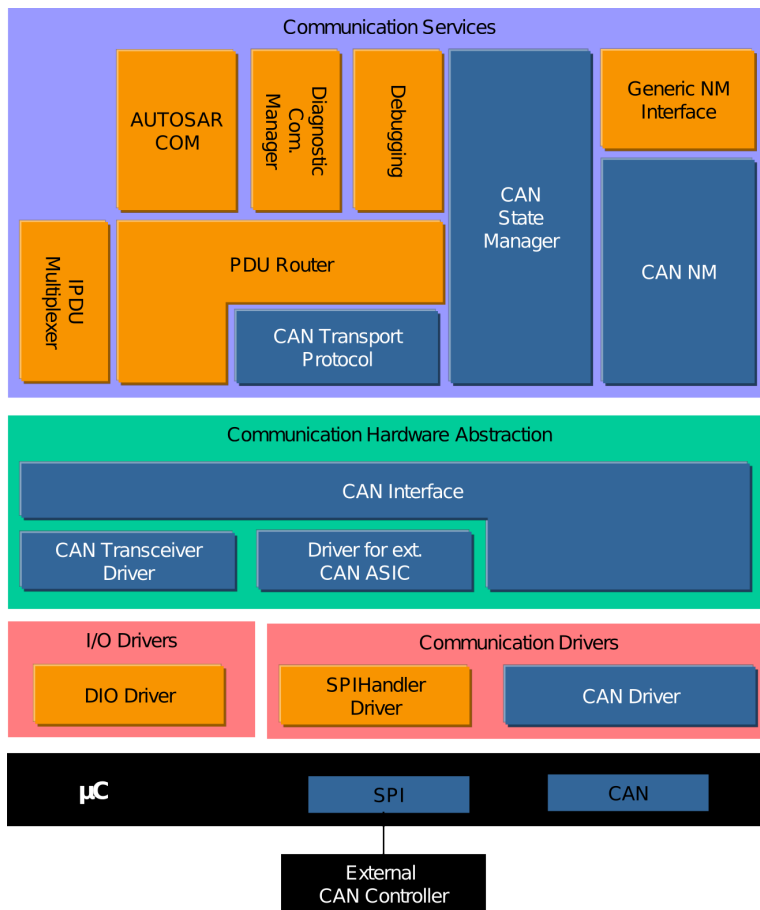


Figure 5.5: AUTOSAR CAN Communication stack [3]

Before proceeding, we explain concepts of signals and PDUs, which are used throughout the description of the CAN communication stack.

### 5.2.1 Signals

As described earlier, Software Components communicate using ports, which must have assigned a port-interface. If the port-interface is of Sender-receiver type, it contains several data elements for data exchange. When two Software Components need to communicate and run on the same ECU, RTE connects the ports locally without the need for a communication stack. But when these components run on different ECUs, communication must be routed through a network. In this case, data elements in the port interface must be mapped to *signals*. RTE sends and receives signals from/to *AUTOSAR COM module*, which will be described later.

### 5.2.2 Protocol Data Unit (PDU)

*Protocol Data Unit* (PDU) is used as a container for data exchange between modules in the communication stack. Every PDU is composed from two parts. *Service Data Unit* (SDU) contains data from the upper layer or application and *Protocol Control Information*, which is used by lower (e.g. transport) layers<sup>2</sup> (see Fig. 5.6).

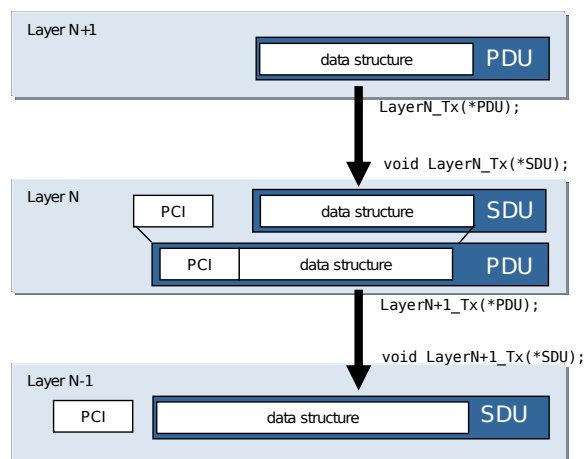


Figure 5.6: PCI and SDU inside PDU [3]

PDUs are prefixed with letters depending on the layer and module in the communication stack (see Fig. 5.7). I-PDUs (Interaction layer PDU) are used by the COM module to pack one or more signals and further by the PDU router which is used to transport them to lower layers. Every module, that handles I-PDUs and provides an API for I-PDUs must contain a list of I-PDU IDs [10]. N-PDUs (Network layer PDU) are used by transport protocol modules and L-PDUs (Data Link layer PDU) are used by lower interface and driver modules. The maximum length of SDU in L-PDUs is protocol specific. For CAN, it is 8 bytes.

Every PDU has an ID represented by type `PduIdType` which may be up to 16 bits long. Content of the PDU is defined by structure `PduInfoType`, which contains `SduDataPtr` (pointer to payload data) and `SduLength` (length of SDU in bytes).

<sup>2</sup>For a simple CAN communication without the transport protocol layer, PCI is not needed.



ISO Layer	Layer Prefix	AUTOSAR Modules	PDU Name	CAN / TTCAN prefix	LIN prefix	FlexRay prefix
Layer 6: Presentation (Interaction)	I	COM, DCM	I-PDU	N/A		
	I	PDU router, PDU multiplexer	I-PDU	N/A		
Layer 3: Network Layer	N	TP Layer	N-PDU	CAN SF CAN FF CAN CF CAN FC	LIN SF LIN FF LIN CF LIN FC	FR SF FR FF FR CF FR FC
Layer 2: Data Link Layer	L	Driver, Interface	L-PDU	CAN	LIN	FR

Figure 5.7: PDU types [3]

### 5.2.3 Communication module (COM)

The COM module is located in the highest services layer of the AUTOSAR communication stack, directly under the RTE. It is responsible for provision of a signal oriented data interface for the RTE, packing and unpacking signals to/from I-PDUs and communication transmission control. Many different data types of signals (including both signed and unsigned integers, floats and booleans) are supported. For integer types, endianness conversion and sign extension might be performed. Signals can also be joined to *signal groups* (used for complex data types) and filtered (for a complete list of features, see [8]).

Initialization is done by the ECU Manager module, which calls the `Com_Init` function with a pointer to module's configuration. During initialization, the COM module initializes I-PDU buffers and sets signals to their initial values. There are scheduled functions, mainly `Com_MainFunctionRx` and `Com_MainFunctionTx` which perform processing activities that are not directly handled by functions for sending signals or functions invoked by lower layers upon I-PDU reception. These functions must be called periodically [8].

When RTE executes a request to send a specific signal, it calls COM's `Com_SendSignal` function, where the signal value is written to the appropriate I-PDU buffer as defined in the configuration. There are several *transfer properties* for signals available, which control if a write access to the signal can trigger the transmission of the corresponding I-PDU. Properties that trigger transmission of the I-PDU include `TRIGGERED` and `TRIGGERED_ON_CHANGE`, while `PENDING` property never triggers the transmission. Whether the I-PDU will be actually transmitted also depends on the *transmission mode* of the corresponding I-PDU. Available modes include `DIRECT`, `MIXED` (triggered by an event) and `PERIODIC` (not triggered, but sent periodically). I-PDUs are sent to lower layers by calling PDU Router's function `PduR_ComTransmit` which in turn confirms the transmission by calling `Com_TxConfirmation`. When the transmission of I-PDU fails, the COM module can retry the transmission request. The maximum number of the repetitions can be set in the configuration.

Incoming I-PDUs from the PDU Router are announced to the COM module by calling the `Com_RxIndication` function which unpacks signals from the I-PDU. In order to support both interrupt driven and polled systems, it can be configured when the signal indication

takes place. There are two configurable *signal indication modes* for each I-PDU – **IMMEDIATE** and **DEFERRED**. When the signal indication mode is set to **IMMEDIATE**, the COM module shall immediately call notification callbacks of the signal contained in the I-PDU (typically callback functions created during generation of RTE). The second mode, **DEFERRED**, means that callback invocation will be made asynchronously during the next call of the `Com_MainFunctionRx` [8].

#### 5.2.4 PDU Router (PduR)

The *PDU Router* is a central part of the AUTOSAR communication stack and is located in the services layer of the Basic Software. It must be present in every AUTOSAR ECU, where communication is needed. Its main purpose is to route I-PDUs (Interaction layer Protocol Data Units) between modules in both lower and upper layers. The set of modules the PDU router can route to, is not fixed and can be specified in the module's configuration. This approach allows integration of Complex Device Driver modules as sources or destinations of I-PDUs.

Routing of I-PDUs is described by static routing tables in the PDU Router configuration. Routing path is uniquely determined by I-PDU ID lookup in the routing tables. I-PDUs can be routed from a single source to a single destination (singlecast 1:1), from a single source to multiple destinations (multicast 1:n) or PDU Router can act as a bi-directional gateway between two communication interfaces. During routing, a different I-PDU ID (from a list of the destination module) is assigned to the original I-PDU, so every module, that handles I-PDUs, should maintain a list of accepted I-PDU IDs. Dynamic routing or any routing decisions dependent on payload of the I-PDU are not supported [10].

In case of simple CAN communication (when I-PDUs are not longer than 8 bytes), the PDU Router is used to route I-PDUs from the COM module to the CAN Interface module. Transmit requests from the COM module are forwarded to the CAN Interface module by calling its `CanIf_Transmit` function. The PDU Router must provide contents of the I-PDU and PDU-ID of the L-PDU to be used for transmission. In the opposite direction, CAN Interface module notifies PDU Router about an incoming PDUs by calling its `PduR_CanIfRxIndication` function.

#### 5.2.5 CAN Interface (CanIf)

The CAN Interface module is located in the intermediate layer of the communication stack (see Fig. 5.5) and represents an interface to the services of the CAN Driver modules for the upper communication layers. This interface consists of all high-level hardware independent tasks needed in CAN communication (transmit request processing, transmit confirmation, receive indication, error notification etc.). The CAN Interface module is also used for mode management of the underlying CAN controllers and stores information about their current status. State transitions are realized by upper modules, like CAN State Manager, or by external events. Initialization is done by the ECU Manager module, which calls the `CanIf_Init` function. The initialization process shall only take place, if all controllers are in stopped or uninitialized state [6].

Upper layers call `CanIf_Transmit` function to request a transmission of L-PDUs on the CAN network, without a direct interaction with the CAN Driver modules. When the function is called, it identifies target CAN Driver module (only if multiple CAN Drivers are used), determines Hardware Transmit Handle (an abstract reference to a CAN Mailbox) and calls `Can_Write` function of the CAN Driver module. The transmission request is completed, when the `CanIf_Transmit` function returns `E_OK`. However, when target CAN controller's mailbox is in use, the function will return `E_NOT_OK` and the upper layer must retry the transmission. This can be avoided by buffering the L-PDU in the CAN Interface module, which then takes care of the outstanding transmission of the L-PDU. The CAN Driver notifies the CAN Interface module about successful transmission by calling its `Can_TxConfirmation` function, which in turn notifies upper modules.

Upon reception of the CAN L-PDU, the CAN Driver module notifies the CAN Interface module by calling its `CanIf_RxIndication` function. The CAN Driver also provides CAN ID and DLC of the received frame to the CAN Interface, so the function may perform software filtering and compare DLC value of the CAN frame with value specified in the configuration. Appropriate upper layer (specified in the configuration) is then notified by calling its receive indication function.

### 5.2.6 CAN Driver module (`CanDrv`)

The CAN Driver module is part of the lowest microcontroller abstraction layer in the Basic Software. It has direct access to the hardware and offers a hardware independent API to the CAN Interface module, which is the only module in the upper layer, which communicates with it. The CAN Driver module can control several CAN controllers which may be located on-chip or as an external device, but they must belong to the same CAN Hardware Unit<sup>3</sup>. If this is not the case, several CAN Driver modules shall be implemented. As a consequence of a direct hardware access, this module is hardware dependent.

The main task of the CAN Driver module is to provide services for transmitting L-PDUs and notify the upper CAN Interface module upon reception of L-PDUs by calling its callback functions. Furthermore it offers services to control the behavior and state of the CAN controllers. If the CAN controller is on-chip, the CAN Driver module does not use services of other modules. The only exception is the digital I/O configuration, which is done by the Port module. If the CAN controller is off-chip, the CAN Driver module may use services of other MCAL drivers, like the SPI driver.

CAN Driver modules support two modes of operation — polling and interrupt. Polling mode is maintained by `Can_MainFunction_xxx` functions which must be called periodically. If the interrupt mode of operation is selected, the CAN Driver module must implement interrupt service routines of the CAN hardware interrupts.

CAN Driver module is initialized by the ECU Manager during the startup phase. It calls function `Can_Init` with a pointer to configuration. This function then initializes the CAN controller. When the CAN Driver module receives an L-PDU, it notifies the CAN Interface layer by calling its receive indication callback with ID, DLC and pointer to L-SDU as parameters. During transmission, CAN Driver module converts L-PDU contents, ID and

---

<sup>3</sup>CAN Hardware Unit consists of one or multiple CAN controllers of the same type, each controller serves exactly one physical channel

DLC to a hardware specific format, if necessary. A CAN controller has four basic states defined at the software level:

- **UNINIT** – CAN controller is not initialized and is not participating on the CAN bus.
- **STOPPED** – CAN controller is initialized, but is not participating on the CAN bus.
- **STARTED** – CAN controller is in normal operation and participates on the CAN bus.
- **SLEEP** – Same as **STOPPED**, but CAN controller can be woken up over the CAN bus.

For each CAN controller a corresponding software state machine is implemented inside the CAN Interface module, which manages the state of the CAN controller by calling functions of the CAN Driver module. These functions only encapsulate CAN hardware access and the CAN Driver module neither memorizes the state changes nor checks for validity of the state changes. States can also be changed by an external events like Bus-off event and HW wakeup event. When the CAN Driver module recognizes these external events (by polling or interrupt) it does not change the state by itself, but rather notifies the CAN Interface module which changes the state inside the callback function.

### 5.3 ArcCore and their products

ArcCore, founded in 2009 and based in Sweden, is a provider of products and services for embedded systems, including AUTOSAR solutions for automotive market. There are also several other vendors of AUTOSAR solutions, like Vektor Informatik, Elektrobit, Freescale or ETAS. Products offered by these companies range from sole implementations of Basic Software layer to complete environments including generators, configurators and system tools, but all these products are licensed under commercial licenses. In contrast, people in ArcCore are strongly focused on open standards, like AUTOSAR and dislike proprietary solutions. That is why they offer an open source licensing model, apart from standard commercial licenses. Being open source means more users can use their products and also bring better understanding of what is needed to progress forward. Products available under the open source license include Arctic Core and Arctic Studio (but excluding AUTOSAR specific tools).

#### 5.3.1 Arctic Core

Arctic Core is an implementation of the AUTOSAR embedded platform (compliant with AUTOSAR 4.1.1) and includes a full set of features required in an automotive Electronic Control Unit. Standard package among others includes AUTOSAR OS (extended OSEK OS), ECU state manager, communication services (CAN, LIN, ETH and COM), diagnostic services and memory management services. It is distributed in form of source code, including build scripts and several example projects for Arctic Studio (including demonstrations of LIN and CAN communication and examples about how to use the RTE and the real-time operating system). Currently supported microcontroller architectures are Power PC (Freescale MPC microcontrollers) and ARM (STMicroelectronics STM32F103 and STM32F107 series).

### 5.3.2 Arctic Studio

Arctic Studio (see Fig. 5.8) is a complete embedded software development environment for automotive embedded software based on the AUTOSAR standard. It is based on Eclipse IDE and provides tools for different types of tasks: application development, embedded platform development and system integration. Build system supports GNU Make, GCC (ARM) and CodeWarrior (Power PC). Currently the only supported host operating system is Microsoft Windows (Linux version is under development).

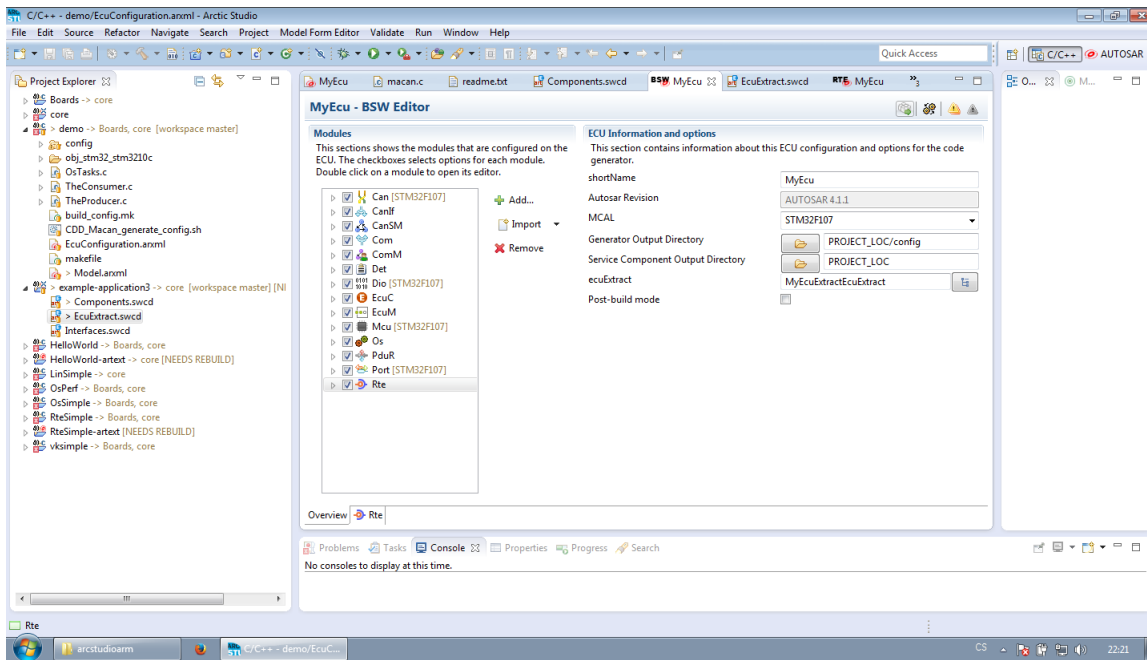


Figure 5.8: Arctic Studio IDE

Arctic Studio allows developers to create Software Components in the text editor and to define data types, ports, signal mappings and behaviour using ARText<sup>4</sup> language. These definitions, stored in several SWCD files, are then transformed (according to composition of Software components) to an arxml file known as Ecu Extract. An integrated tool called RTE Editor is used to configure the RTE (map runnables and tasks to Software Components etc.) and for actual RTE code generation. Arctic Studio also provides full set of configuration tools for configuring AUTOSAR Basic Software. These tools are able to validate and generate configurations for Basic Software modules according to the AUTOSAR standard and are distributed under a commercial license.

## 5.4 Integration of MaCAN into AUTOSAR

After getting familiar with the AUTOSAR architecture and reading the specification, it was obvious, that MaCAN cannot exist in the AUTOSAR implementation “as is”, but must be

<sup>4</sup><https://www.artop.org/artext/>

modified to be able to communicate with other subsystems of AUTOSAR. We have stated a set of goals, that the chosen way of integration should meet:

- **Easy to implement** – as stated in the KISS (Keep It Simple, Stupid) principle, most systems work best if they are kept simple rather than made complicated. Therefore simplicity should be a key goal in design and unnecessary complexity should be avoided.
- **Lowest possible impact on implementation of existing AUTOSAR modules and components** — this will increase the possibility to use MaCAN with modules from different AUTOSAR vendors.
- **Modify the original MaCAN code as little as possible** — MaCAN still exists and is maintained as a standalone implementation. This goal is a must for keeping its maintenance and future development on a manageable level.

It turned out there are basically two very distinct ways how to integrate MaCAN into AUTOSAR, which more or less conform with the goals stated. One possibility is to encapsulate MaCAN into a Software Component and the other one is to integrate it into Basic Software as a module. We have considered both options and tried to find their advantages and disadvantages.

#### 5.4.1 MaCAN as a Software Component

First, we have considered integration of MaCAN into AUTOSAR as a Software Component, that would offer services via a Client-Server port interface to other components. This idea is based on the fact that lower layers of the communication stack are safety-related, so every additional piece of code must be developed according to strict rules and certified by an independent auditor. If we were able to integrate MaCAN into a Software Component, its deployment in real system would be much easier at least from a safety point of view. However, since MaCAN is designed to have full access to the CAN interface, this would imply considerable overhead and extensive modifications to existing modules in the Basic Software. Therefore, we have decided to abandon the idea of integrating MaCAN as a Software Component.

#### 5.4.2 MaCAN as a module in the Basic Software

At first glance, it seemed that integrating MaCAN as a module into the Basic Software layer will have greater impact on existing modules than the approach described in the previous section. This was supported by the fact, that MaCAN module would need to communicate with other modules standardized in the Basic Software so some modifications of the existing modules would be necessary. In AUTOSAR, there is however a class of Basic Software modules called *Complex Device Drivers* (CDD), which allows to add a non-standard functionality.

CDD were introduced in section 5.1.3.3 as modules for complex sensor evaluation and actuator control. But in addition, CDD might also be used to implement enhanced services/protocols or encapsulate legacy functionality of a non-AUTOSAR system [7]. As CDD

span all layers of the Basic Software, they have access to both high level services and low level drivers. Additionally, other modules in the Basic software, including those in the communication stack (COM module, PDU Router and bus interface modules) can be configured to interface CDD. These are good preconditions for integrating MaCAN as a CDD module into the CAN communication stack. MaCAN functionality integrated into the CAN communication stack is illustrated in Fig. 5.9.

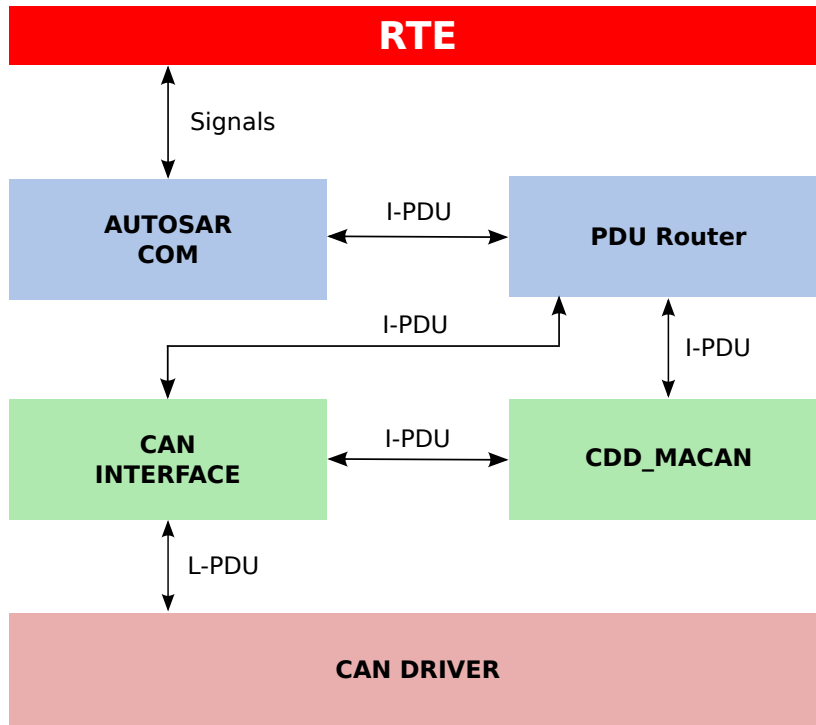


Figure 5.9: Simplified CAN communication stack with CDD\_Macan module

Proposed CDD module is in the form of a wrapper module, that encapsulates the MaCAN implementation and is used to call its functions. It is placed in the ECU abstraction layer between the PDU Router and the CAN Interface module.

Incoming signals packed in I-PDUs from the PDU Router are transformed to CAN frames according to a simple translation table (this step is necessary since MaCAN implementation works with CAN frames and not with I-PDUs). These CAN frames are then handed over to MaCAN which appends a signature. CAN frames are then again transformed to I-PDUs and forwarded to the CAN Interface module to be transmitted. The CAN Interface is made aware of I-PDUs that belong to the MaCAN module so it knows where to forward incoming I-PDUs.

This approach also meets the goals stated above:

- It is simple, because it wraps the MaCAN implementation. In general, the module will only do several translations between data formats and pass the data to the MaCAN implementation.

- The CAN Interface and the PDU Router can interface CDD modules and will possibly require only minor modifications. No modifications to the application software is necessary. Existing signals can be easily secured by configuring the PDU Router to route them through the CDD\_MaCAN module.
- The MaCAN implementation will require only a little modification, in fact only target specific functions will have to be modified.

We have finally decided to integrate MaCAN into AUTOSAR as a Basic Software module.

## 5.5 Implementation of CDD\_Macan in Arctic Core

In this section, we cover details about implementation of the CDD\_Macan module described earlier. We also describe configuration options and details about compilation together with Arctic Core.

### 5.5.1 File structure

Functionality of the CDD\_Macan module is contained within a single CDD\_Macan.c file located in the `core/communication/CDD_Macan` directory. Functions in this file have CDD\_Macan prefix to be clearly distinguished from functions that belong to the MaCAN implementation and have `macan` prefix. Header file hierarchy is illustrated in Fig. 5.10.

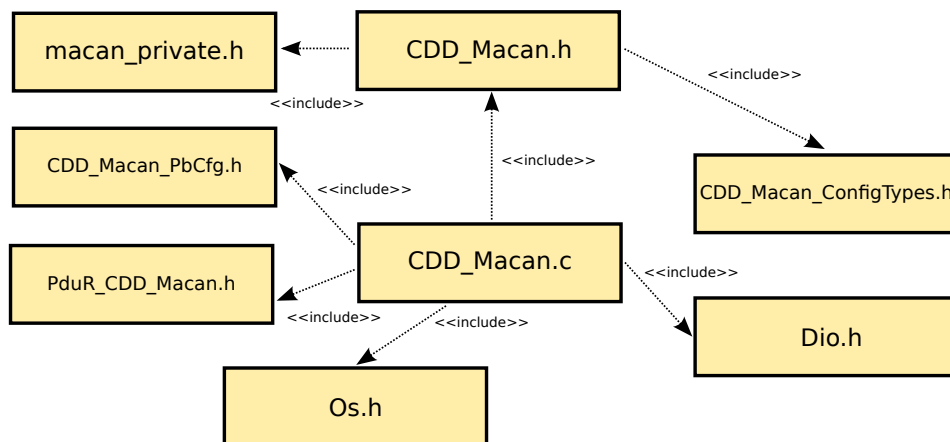


Figure 5.10: Header file hierarchy

### 5.5.2 Initialization

The module is initialized by the ECU Manager during second phase of the startup procedure (when the operating system is already running), which calls the `CDD_Macan_Init` function with a pointer to the configuration structure. To allow this, it is necessary to add an entry to the `PostBuildconfigType` structure type, which stores pointer to post-build configurations:



```
#if defined(USE_CDD_MACAN)
    const CDD_Macan_ConfigType* const CDD_Macan_ConfigPtr;
#endif
```

assign this pointer in CDD\_Macan's configuration into `Postbuild_config` structure located in `EcuM_PBHeader.c`:

```
#if defined (USE_CDD_MACAN)
    .CDD_Macan_ConfigPtr = &CDD_Macan_Config,
#endif
```

Finally, we had to modify the `EcuM_Callout_Stubs.c` file by adding following lines inside the `EcuM_AL_DriverInitTwo` function (used to initialize Basic Software modules):

```
#if defined(USE_CDD_MACAN)
    // Initialize Macan module
    NO_DRIVER(CDD_Macan_Init(ConfigPtr->PostBuildConfig->CDD_Macan_ConfigPtr));
#endif
```

The initialization function is used to initialize our MaCAN implementation, by calling its `macan_init` function and to register callback functions for every signal defined in the configuration of the MaCAN implementation.

### 5.5.3 Establishing a secure communication channel

Before any signed signals can be sent or received, the CDD\_Macan module must establish a secure communication channel with all communication partners. This involves sending a session key request, an ACK message and a signal request message. In the standalone MaCAN implementation, there is an event loop, which is used to periodically call so called "housekeeping" function, which checks validity of the session keys and requests a new session key when necessary. This event loop is not needed in AUTOSAR, since there is a real-time operating system. We have wrapped a call of the housekeeping function to the `CDD_Macan_MainFunction_Hk` function, which is called about every 500 ms by a periodic task or Basic Software Scheduler.

### 5.5.4 Transmission of a signed signal

Signal transmission starts at the application layer, when a Software Component wants to update a value of a data element in its port. The request is directed through RTE to the COM module, which wraps the signal to an I-PDU. Signals that should be signed are packed to I-PDUs and must be at most 4 bytes long. I-PDUs with signals are sent to the PDU Router, which routes them to the CDD\_Macan module. To add this routing capability to the PDU Router, we had to add the `ARC_PDUR_CDD_MACAN` entry to the `ARC_PduR_ModuleType` enumeration. This value is used in the PDU Router configuration to specify destination of the I-PDU. We also had to add following lines in a switch statement in the `PduR_ARC_RouteTransmit` function (used to send I-PDU to the destination module):

```

case ARC_PDUR_CDD_MACAN:
#if PDUR_CDD_MACAN_SUPPORT == STD_ON
    retVal = CDD_Macan_Transmit(destination->DestPduId, pduInfo);
#endif
    break;

```

If the destination path of the I-PDU is configured to `ARC_PDUR_CDD_MACAN`, the I-PDU is passed to the `CDD_Macan` by calling its `CDD_Macan_Transmit` function. The PDU Router changes the PDU-ID to a value specified in its configuration and defined in the destination module's config header file, so the destination module can recognize the I-PDU. When the `CDD_Macan_Transmit` function receives an I-PDU for transmission, it must convert the I-PDU ID to the signal ID internal to the MaCAN implementation (by reading an entry inside the translation table in the `CDD_Macan` configuration) and to copy the signal value from the I-PDU. The signal ID and its value is then passed to the `macan_send_sig` function. The MaCAN implementation then creates a CAN frame (fills up the `can_frame` structure) with the signal and appends a signature. This CAN frame is normally transmitted to the CAN network by the platform dependent `macan_send` function. In AUTOSAR, however, the CAN frame needs to be transformed back to the PDU so it can pass through the rest of the communication stack. Therefore the CAN frame is sent to the `CDD_Macan_SendCanFrame` function, which searches a translation table in the configuration (see Listing 5.1) and creates a PDU with PDU-ID defined in the configuration of the CAN Interface. The PDU is then passed to the `CanIf_Transmit` function. This translation mechanism is used also to create PDUs from CAN frames originating in the MaCAN implementation, like session key requests, ACK messages etc. The CAN Interface must know all these PDUs and their PDU-IDs must be defined in its configuration.

The function `CDD_TxCanIdToPduId` could perform better using the binary search algorithm on a translation table sorted by CAN-IDs. This would however also require modifications of the configuration tool described in section 5.5.7. As this implementation is the first prototype, its optimization was not primary goal and is planned in the future.

---

```

Std_ReturnType CDD_Macan_TxCanIdToPduId(uint32 canId, uint32 dlc, PduIdType *pduId) {
    int i;
    for(i = 0; i < CDD_Macan_ConfigPtr->NumberOfTxPduIds; i++) {
        if(CDD_Macan_ConfigPtr->TxPduConfigPtr[i].CanId == canId &&
           CDD_Macan_ConfigPtr->TxPduConfigPtr[i].Dlc == dlc) {
            *pduId = CDD_Macan_ConfigPtr->TxPduConfigPtr[i].PduId;
            return E_OK;
        }
    }
    return E_NOT_OK;
}

```

---

Listing 5.1: Function that translates CAN-ID to PDU-ID

### 5.5.5 Reception of a signed signal

When a CAN frame with a signed signal is received, it is first processed by the CAN Driver module, which notifies the CAN Interface module by passing an L-PDU to its `CanIf_RxIndication` function. Each CAN L-PDU must be specified in the CAN Interface configuration together with CAN-ID, DLC and a module from the upper layer which will be notified about reception of this L-PDU. The CAN Interface can be configured to notify a CDD module by calling its (arbitrary) receive indication function. PDUs containing signed signals and other MaCAN related data must be routed to the `CDD_Macan` module. The receive indication function `CDD_Macan_RxIndication` is used to translate received PDUs to CAN frames according to a translation table in the configuration and store them in a software queue (FIFO). This is necessary, because the receive indication function is called from the context of an interrupt service routine (polling access is not supported by Arctic Core CAN Driver) and direct processing of the received CAN frames can cause loss of CAN messages, due to reasons described in section 4.5.6. CAN frames in the FIFO are read by a scheduled function `CDD_Macan_MainFunction`, which should be called fast enough (once every 10 ms) to minimize the delay before received frames are processed. This function passes the CAN frames to the `macan_process_frame` function.

If the MaCAN implementation successfully verifies signal signature, it calls the callback function `CDD_Macan_SigCallback` assigned to all signals during initialization, which is used to create I-PDU from the internal signal ID and fill it with the signal value. This I-PDU is then passed to the PDU Router, which has a dedicated receive indication function for each lower layer module. Therefore we had to create a receive indication function for the `CDD_Macan` module, which is very similar to the receive indication function called by the CAN Interface module:

```
void PduR_CDD_MacanRxIndication(PduIdType pduId, PduInfoType* pduInfoPtr) {
    PduR_LoIfRxIndication(pduId, pduInfoPtr, 0x01);
}
```

The function is only used to call a common receive indication function with different parameters. Received I-PDUs with signals are then routed to the COM module. During routing, I-PDU ID is changed to a value defined in the COM configuration header file.

When signal verification fails, MaCAN implementation calls the invalid signal callback `CDD_Macan_SigInvalidCmacCallback` (also assigned during initialization) which is used to monitor frequency of invalid signals. If it reaches a certain threshold value, the function calls a callback defined in the configuration (see section 5.5.7 for more details).

### 5.5.6 Time measurement

MaCAN implementation needs a monotonically increasing clock to keep information about current time, which is returned by the `read_time` function. In AUTOSAR, the easiest solution is to use a counter increased by the Operating System. This counter value is multiplied by constant `CDD_MACAN_TIME_MULTIPLIER` specified in the configuration, to get correct time readings in microseconds.

### 5.5.7 Configuration

Configuration of the CDD\_Macan module is divided into several containers, that are defined in the CDD\_Macan\_ConfigTypes.h file. There is a top level container structure CDD\_Macan\_ConfigType which holds main configuration parameters and other configuration containers with members listed in Table 5.1. Specification of PDUs is held in the CDD\_Macan\_PduConfigType structure with members listed in Table 5.2.

The actual configuration of the CDD\_Macan module in the ECU configuration project is stored in config/CDD\_Macan subdirectory. Inside this directory, there is the CDD\_Macan\_PbCfg.c file holding the configuration structures and the CDD\_Macan\_PbCfg.h file with PDU-IDs.

<b>Name:</b>	MacanConfig
<b>Data type:</b>	struct macan_config *
<b>Description:</b>	Pointer to configuration of the MaCAN implementation. It is passed to the macan_init function during initialization of the module. Files holding the configuration macan_config.c and macan_config.h should be placed in the config/CDD_Macan directory.

<b>Name:</b>	RxPduConfigPtr
<b>Data type:</b>	const CDD_Macan_PduConfigType *
<b>Description:</b>	Pointer to an array holding the specification of PDUs received from the CAN Interface. PDU-IDs are specified in the CanIf_PbCfg.h file and are numbered gradually up from zero. This allows using PDU-ID directly as index in the array, provided it is sorted ascending by PDU-ID.

<b>Name:</b>	TxPduConfigPtr
<b>Data type:</b>	const CDD_Macan_PduConfigType *
<b>Description:</b>	Pointer to an array holding the specification of PDUs transmitted to the CAN Interface. PDU-IDs are specified in the CanIf_PbCfg.h file. When performing a CAN-ID to PDU-ID translation, CAN-ID cannot be used as index, so the array must be searched.

<b>Name:</b>	NumberOfRxPduIds
<b>Data type:</b>	uint32
<b>Description:</b>	Size of the array holding the specification of PDUs received from the CAN Interface. Used for boundary check.

<b>Name:</b>	NumberOfTxPduIds
<b>Data type:</b>	uint32
<b>Description:</b>	Size of the array holding the specification of PDUs transmitted to the CAN Interface. Used for boundary check.

<b>Name:</b>	PduIdToInternalSignalIdMap
<b>Data type:</b>	const uint32 *

<b>Description:</b>	Pointer to an array representing a PDU-ID to Internal MaCAN signal ID translation table. It is used in the <code>CDD_Macan_Transmit</code> function to recognize signals coming packed in I-PDUs from the PDU router. PDU-IDs are defined in the <code>CDD_Macan_PbCfg.h</code> file and are numbered gradually up from zero. This allows using PDU-ID directly as index of the searched internal signal ID.
---------------------	--

<b>Name:</b>	<code>InternalSignalIdToPduIdMap</code>
<b>Data type:</b>	<code>const uint32 *</code>
<b>Description:</b>	Pointer to an array representing an Internal MaCAN signal ID to PDU-ID translation table. It is used in the <code>CDD_Macan_SigCallback</code> function to pack received signals to I-PDUs before sending them to the PDU Router. Internal signal IDs are defined in the <code>macan_config.h</code> file and must be numbered gradually up from zero. This allows using the Internal signal ID directly as an index to the array.

<b>Name:</b>	<code>maxNumberOfInvalidCMAC</code>
<b>Data type:</b>	<code>uint32</code>
<b>Description:</b>	Maximum number of invalid CMACs within <code>invalidCMACPeriod</code> .

<b>Name:</b>	<code>invalidCMACPeriod</code>
<b>Data type:</b>	<code>uint32</code>
<b>Description:</b>	How long to count invalid CMACs before resetting the counter (in $\mu$ s).

<b>Name:</b>	<code>invalidCMACCallback</code>
<b>Data type:</b>	<code>const void *(void)</code>
<b>Description:</b>	Pointer to a callback function, which is called once the <code>maxNumberOfInvalidCMAC</code> has been exceeded within the <code>invalidCMACPeriod</code> .

Table 5.1: Members of the `CDD_Macan_ConfigType` structure

<b>Name:</b>	<code>PduId</code>
<b>Data type:</b>	<code>PduIdType</code>
<b>Description:</b>	PDU-ID of the PDU being send to/received from a lower layer.

<b>Name:</b>	<code>canId</code>
<b>Data type:</b>	<code>uint32</code>
<b>Description:</b>	CAN-ID of the CAN frame associated with this PDU.

<b>Name:</b>	<code>Dlc</code>
<b>Data type:</b>	<code>const void *(void)</code>
<b>Description:</b>	Data Length Code of the CAN frame associated with this PDU. This is needed because for two frames with the same CAN-ID but different DLC we need two different PDU-IDs

Table 5.2: Members of the `CDD_Macan_PduConfigType` structure

### 5.5.8 Compilation with Arctic Core

Arctic Core code is compiled inside ECU configuration projects using makefiles generated during creation of the project. Each project has one or more target boards specified and each board has a set of Basic Software modules available. In the project's config subdirectory, there is a set of \*.mk files, which specify modules to be used with the project (one file per module). Source codes of modules are then conditionally compiled by the rules in the `board_common.mk` file located in `core/boards`.

We have created a new directory `CDD_Macan` in `core/communication`. Its purpose is to hold sources of our MaCAN implementation and also sources of the `CDD_Macan` module. The sources of our MaCAN implementation are located in the `macan` subdirectory. More precisely, this subdirectory is a GIT submodule. To compile our MaCAN implementation with Arctic Core, we had to add `CDD_MACAN` to available modules for the STM3210C-EVAL board by adding following contents to the `board_config.mk` file in `core/boards/stm32_stm3210c`:

```
# Macan CDD module
MOD_AVAIL += CDD_MACAN
```

To add `CDD_Macan` module to a project, the line `MOD_USE += CDD_MACAN` must be put into a newly created `CDD_Macan.mk` file in the project's config directory. Finally we had to add following lines to `core/boards/board_common.mk` file:

```
obj-$(USE_CDD_MACAN) += CDD_Macan.o
obj-$(USE_CDD_MACAN) += CDD_Macan_PBCfg.o
obj-$(USE_CDD_MACAN) += macan_config.o
vpath-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/CDD_Macan
inc-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/CDD_Macan

# MaCAN C files
vpath-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/ \
                          CDD_Macan/macan/macan/src/target/stm32
vpath-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/CDD_Macan/macan/macan/src

MACAN_C_FILES += stm32_cryptlib.c
MACAN_C_FILES += common.c
MACAN_C_FILES += cryptlib.c
MACAN_C_FILES += debug.c
MACAN_C_FILES += macan.c
MACAN_C_FILES += stm32_autosar.c

# Nettle C files
VPATH += $(ROOTDIR)/communication/ \
         CDD_Macan/macan/macan/src/target/stm32/nettle
MACAN_C_FILES += aes-decrypt.c
MACAN_C_FILES += aes-decrypt-internal.c
MACAN_C_FILES += aes-encrypt.c
```

```

MACAN_C_FILES += aes-encrypt-internal.c
MACAN_C_FILES += aes-encrypt-table.c
MACAN_C_FILES += aes-invert-internal.c
MACAN_C_FILES += aes-set-decrypt-key.c
MACAN_C_FILES += aes-set-encrypt-key.c
MACAN_C_FILES += aes-set-key-internal.c
MACAN_C_FILES += memxor.c

obj-$(USE_CDD_MACAN) += $(MACAN_C_FILES:%.c=%.o)
inc-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/CDD_Macan/macan/macan/include
inc-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/CDD_Macan/macan/macan/include
inc-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/CDD_Macan/macan/macan/src/ \
    target/stm32
inc-$(USE_CDD_MACAN) += $(ROOTDIR)/communication/CDD_Macan/macan/macan/src/ \
    target/stm32/nettle

CFLAGS += -D__CPU_STM32F107__
CFLAGS +=-DUSED_IN_AUTOSAR

```

In the beginning we include files of the CDD\_Macan module. Then we have set paths, where Make should look for the C files followed by a list of files belonging to MaCAN and Nettle library. Finally, we have specified compiler flags and a list of locations, where Make should look for include files. We have introduced a new macro USED\_IN\_AUTOSAR which is used to disable the event loop, as described in section 5.5.3.

### 5.5.8.1 Configuration tool

Configuration of the modules in Basic Software is typically done by a GUI tool called BSW Editor, which validates the configuration and generates configuration files inside project's `config` directory. Since we had not access to source codes of Arctic Studio, we were not able to use the BSW Editor for configuration of the CDD\_Macan module. As a substitution, we have created a command line configuration tool which generates all configuration files needed. It is written in BASH and uses common command line utilities, which can be found on most GNU/Linux systems. The script performs following actions:

- Creates a CDD\_Macan.mk file, which enables the CDD\_Macan module.
- Generates CDD\_Macan\_PbCfg.h and CDD\_Macan\_PbCfg.c files by reading configuration of the MaCAN implementation and configuration of the CAN Interface.
- Modifies configuration files of the PDU Router. Mainly it enables CDD\_Macan support in the PDU Router and corrects destination and source modules in routing paths. This is necessary, since the BSW Editor is not aware of changes we have made in PDU Router.

Note that it is necessary to specify `node_id` in the `macan_config.c` file so the tool knows which signals belong to this node. It is also required, that the PDU-IDs of L-PDUs in the CAN Interface configuration have prefix `SPECIAL_PDU_ID_MACANRXPDU` and

SPECIAL\_PDU\_ID\_MACANTXPDU. This can be achieved by using a virtual EcuC module to store PDU names with “MacanRXPdu” or “MacanTxPdu” prefix followed by name of a node or name of a signal (see Appendix A for an example demo configuration).

To generate configuration of the CDD\_Macan module, the following steps must be performed:

1. Generate a complete ECU configuration for other modules.
2. Copy the configuration tool script `CDD_Macan_generate_config.sh` from the `core/communication/CDD_Macan/` directory to project’s root directory.
3. Create or copy configuration of the MaCAN implementation into `CDD_Macan` located in project’s `config` directory.
4. Invoke the script without any parameters from project’s root directory.

The scripts outputs information about what it currently does and which nodes and signals it found in the configuration of the MaCAN implementation:

```
INFO: Exploring MaCAN config...
--> Our node_id is NODE1
--> Our CAN id is 0x102
Other participants are:
--> KEY_SERVER
--> TIME_SERVER
--> NODE2
Signals found:
--> TX signal SIGNAL_A
--> RX signal SIGNAL_B
INFO: Creating file ./config/CDD_Macan.mk
INFO: Creating file ./config/CDD_Macan/CDD_Macan_PbCfg.h
INFO: Creating file ./config/CDD_Macan/CDD_Macan_PbCfg.c
INFO: Enabling CDD_Macan support in PduR
INFO: Modifying PduR config
DONE!
```

## 5.6 Demo application

We have created a demo based on **1signal** demo described in section 4.3.2 to test functionality of the CDD\_Macan module. The `NODE1`, formerly implemented on GNU/Linux, is now replaced with an AUTOSAR implementation running on the `STM3210C-EVAL` board, while `NODE2`, `KS` and `TS` still run on GNU/Linux. Nodes participating in the communication are listed in Table 5.3, signals are listed in Table 5.4 and layout of the CAN network is illustrated in Fig. 5.11. For a more detailed guide on how to setup this demo in Arctic Studio, see Appendix A.



The AUTOSAR node sends signed **SIGNAL\_A** to **NODE2** according to state of a button on the board – when the button is pressed, the signal has value 1, when released, the signal has value 0. When signed **SIGNAL\_B** from **NODE2** with value greater than zero is received, a blue LED on the board is switched on. The LED is switched off when the signal value equals to zero.

Node name	Platform	Crypt frame CAN-ID	ECU-ID
KS	GNU/Linux	0x100	0x0
TS	GNU/Linux	0x101	0x1
NODE1	AUTOSAR	0x102	0x2
NODE2	GNU/Linux	0x103	0x3

Table 5.3: Nodes in the demo

Signal	ID	Source	Destination	Secure CAN-ID
SIGNAL_A	0	NODE1	NODE2	0x201
SIGNAL_B	1	NODE2	NODE1	0x202

Table 5.4: Signals in the demo

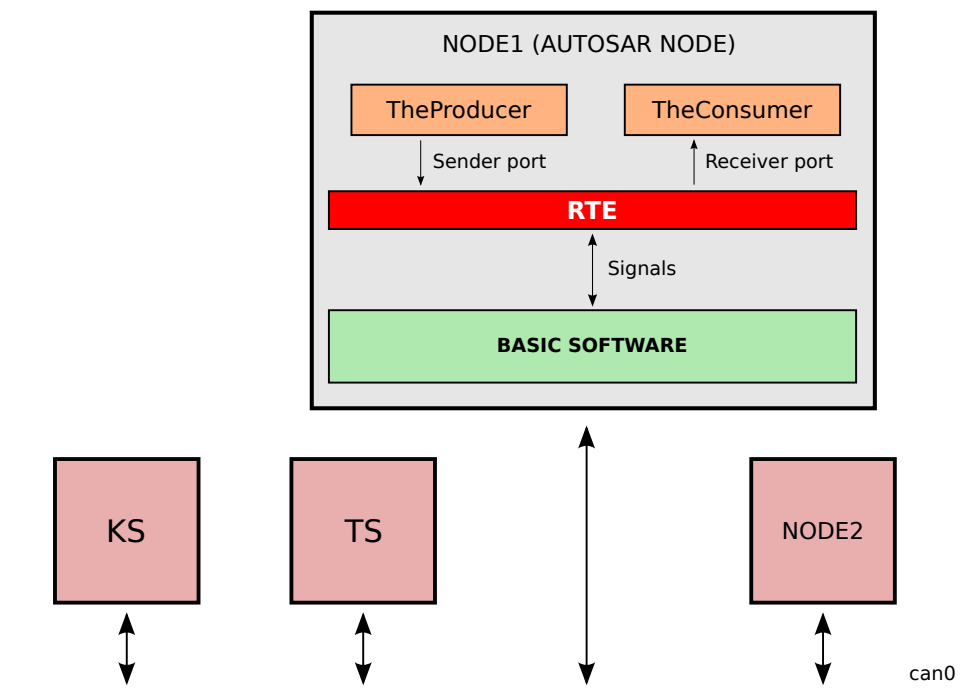


Figure 5.11: Nodes in the demo

### 5.6.1 Application model

Application model of the demo is created inside an ArcCore AUTOSAR project and consists of two Software Components – **TheProducer** and **TheConsumer**. Both communicate through a port of a **MySRInterface** type. This sender-receiver port-interface is very simple and contains only one data element (see Listing 5.2). Both **TheProducer** and **TheConsumer** contain runnables, which will be implemented later in the ECU Configuration project.

---

```
interface senderReceiver MySRInterface {
    data uint32 MyElement
}
```

---

Listing 5.2: **MySRInterface** port-interface

**TheProducer** provides **MySRInterface** and contains one runnable, which is executed periodically and has a write access to the **MyElement** data element in its port. The ARText description of **TheProducer** is given in Listing 5.3.

---

```
component application MyProducer {
    ports {
        sender MySenderPort provides MySRInterface
    }
}
internalBehavior MyProducerBehavior for MyProducer {
    runnable MyProducerRunnable [1.0] {
        timingEvent 1.0
        dataWriteAccess MySenderPort.*
    }
}
implementation MyProducerImplementation for MyProducerBehavior {
    language c
    codeDescriptor "src"
}
```

---

Listing 5.3: Description of **TheProducer** component

**TheConsumer** requires **MySRInterface** and contains one runnable, which is executed upon an event that is raised when data element in the port changes. The ARText description of **TheConsumer** is given in Listing 5.4.

---

```
component application MyConsumer {
    ports {
        receiver MyReceiverPort requires MySRInterface
    }
}
internalBehavior MyConsumerBehavior for MyConsumer {
    runnable MyConsumerRunnable [1.0] {
        dataReadAccess MyReceiverPort.*
        dataReceivedEvent MyReceiverPort.MyElement as DataReceivedEvent
    }
}
```

```

    }
  }
  implementation MyConsumerImplementation for MyConsumerBehavior {
    language c
    codeDescriptor "src"
  }
}

```

---

Listing 5.4: Description of `TheConsumer` component

Finally, we have created a system composition (see Listing 5.5) with `TheProducer` and `TheConsumer` components and connected their ports together. We also had to map the ports to signals<sup>5</sup>, otherwise the components would communicate only locally through RTE. When ports are mapped to signals, the RTE will communicate with the COM module, which sends/receives signals through the communication stack. The composition is extracted to an AUTOSAR XML file called ECU Extract and will be used in the ECU Configuration project.

```

@EcuExtract
composition MyEcuExtract {

  @ImplMapping
  prototype MyProducer TheProducer
  @ImplMapping
  prototype MyConsumer TheConsumer

  ports {
    @SignalMappings
    provides TheProducer.MySenderPort Tx1
    @SignalMappings
    requires TheConsumer.MyReceiverPort Rx1
  }
  connect TheProducer.MySenderPort to TheConsumer.MyReceiverPort
}

```

---

Listing 5.5: Description of the composition

### 5.6.2 ECU Configuration Project

ECU Configuration project is used for configuration of a particular ECU and also holds an implementation of runnables in the Software Components. Configuration files for Basic Software modules are generated using the BSW Editor. The ECU extract, obtained in the previous step, is used to generate the RTE layer with the RTE Editor. Once all necessary files are generated, the project is compiled into an executable file, which can be downloaded to the target board/ECU.

In this section, we describe an implementation of runnables from both `TheProducer` and `TheConsumer` components. We also briefly describe configuration of some Basic Software modules that are directly relevant to the demo. The list of all modules needed for running the demo, together with their configuration options is given in Appendix A.

---

<sup>5</sup>`Rx1` and `Tx1` are just names here. They must be assigned to real AUTOSAR signals in configuration of the COM module.

### 5.6.2.1 TheProducer runnable

**TheProducer** runnable (see implementation in Listing 5.6) is called periodically and reads value of a button on the board and calls a generated method of the RTE to update data element in its port. The runnable should not directly call functions of Basic Software modules, but to make the demo simpler, we wanted to use the DIO module directly. However, we were not able to successfully configure it to read the button state so we have used function from the STM32 Standard Peripheral Library. In a real world application, the Software Component should use AUTOSAR interface to communicate with a Basic Software module or use another Software Component to access hardware on the microcontroller.

---

```
void MyProducerRunnable( void ) {
    uint8 value;
    value = GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_9);
    Rte_IWrite_MyProducerRunnable_MySenderPort_MyElement(value);
}
```

---

Listing 5.6: Implementation of a runnable in **TheProducer**

### 5.6.2.2 TheConsumer runnable

**TheConsumer** runnable (see implementation in Listing 5.7) is called once the data element in the port of the component changes. The value is read using a generated method of the RTE and depending on the value, the blue LED is switched on or off. As mentioned above, the DIO module is used directly for simplicity, but it should be avoided when used in a real world application.

---

```
void MyConsumerRunnable( void ) {
    uint32 x = Rte_IRead_MyConsumerRunnable_MyReceiverPort_MyElement();
    if(x > 0) {
        Dio_WriteChannel(DIO_CHANNEL_NAME_LED_CHANNEL4, STD_HIGH);
    } else {
        Dio_WriteChannel(DIO_CHANNEL_NAME_LED_CHANNEL4, STD_LOW);
    }
}
```

---

Listing 5.7: Implementation of a runnable in **TheConsumer**

### 5.6.2.3 EcuC module

The EcuC is a virtual module used to collect ECU configuration specific or global configuration information. In our case it serves for storing a collection of PDUs, or more precisely, their names. These names can be then referenced in other modules which helps to generate more readable configuration files, because modules can use these PDU names in constants representing PDU-IDs.

#### 5.6.2.4 COM module

The COM module contains specification of signals and I-PDUs to which they are packed. We have added a specification of AUTOSAR signals named Tx1 and Rx1, which correspond to MaCAN signals SIGNAL\_A and SIGNAL\_B respectively. These signals contain a reference to signal names used in the system composition, so when a Software Component writes/reads a port mapped to a signal, the RTE knows it must use functions of the COM module to send/receive that signal. The Tx1 signal has transfer property set to TRIGGERED\_ON\_CHANGE so it is transmitted only when it's value changes. The Rx1 signal must have a notification callback specified, so the COM module can notify the RTE about signal change. In our demo, the callback is RTE's generated function Rte\_COMCbk\_COM\_SIGNAL\_ID\_RX1.

#### 5.6.2.5 PDU Router module

In our demo, the PDU Router is configured to route the Tx1 signal packed in I-PDUs to the CDD\_Macan module, which signs it and sends it to the CAN Interface module. Additionally, the signal is also routed directly to the CAN Interface module to be sent without a signature. In other direction, I-PDUs containing the Rx1 signal are accepted from the CDD\_Macan module and also from the CAN Interface module and are further routed to the COM module, which does not care from which module the I-PDU came. This is used to demonstrate, that addition of the CDD\_Macan module has no impact on layers above the PDU Router.

#### 5.6.2.6 CAN Interface module

All CAN L-PDUs sent between the CAN Interface module and the CAN Driver module must be specified in its configuration. When upper layer wants to send data, it specifies PDU-ID of the L-PDU that should be used for transmission. In our case there are L-PDUs carrying plain signals, signed signals and other L-PDUs used in MaCAN communication for session key requests, ACK messages etc. Because we have disabled DLC check for incoming L-PDUs, it is not necessary to configure separate L-PDUs for CAN frames with the same CAN-ID, but different DLC. However this is not the case for outgoing PDUs. A list of CAN L-PDUs (which are directly associated with CAN frames) used in this demo is listed in following table:

CAN-ID	DLC	Direction	Dest./Src.	Description
0x0	8	Receive	CDD_Macan	Plain time from the timeserver
0x1	8	Receive	PduR	Plain signal from NODE2
0x100	8	Receive	CDD_Macan	Crypt-frames from the keyserver
0x101	8	Receive	CDD_Macan	Crypt-frames from the timeserver
0x103	8	Receive	CDD_Macan	Crypt-frames from NODE2
0x201	8	Receive	CDD_Macan	Signed signal from NODE2
0x2	8	Transmit	PduR	Plain signal to NODE2
0x102	7	Transmit	CDD_Macan	Crypt-frames for other nodes
0x102	8	Transmit	CDD_Macan	Crypt-frames for other nodes
0x200	8	Transmit	CDD_Macan	Signed signal to NODE2

### 5.6.2.7 CAN Driver module

We have used a template configuration for the STM3210C-EVAL board shipped with Arctic Studio to configure the CAN Driver module. The only change we have made is to set CAN Hardware objects to accept standard CAN-IDs instead of extended CAN-IDs.

### 5.6.2.8 OS module

The Operating System module is an essential part of our demo. Its configuration contains specification of alarms, counters, events and tasks. Alarms are used to periodically raise events, counters can be used for time measurement and tasks are pieces of code which can be activated.

In our demo, there are 3 tasks which are activated right after the start of the operating system – `Os_TaskInit`, `Os_TaskPeriodic` and `RteTask`. The `Os_TaskInit` task is used to initialize modules and the communication stack and its implementation is taken from other demos in Arctic Core. The `Os_TaskPeriodic` waits for events raised every 10ms and 500ms and is used to call scheduled functions of the Basic Software modules. The implementation of the task is listed in Listing 5.8. The `RteTask` is generated by RTE and executes runnables when events associated with them are raised.

---

```

void OsTask_Periodic( void ) {

    EventMaskType eventMask = 0;
    while (1) {
        WaitEvent(EVENT_MASK_OsEvent_500ms | EVENT_MASK_OsEvent_10ms);
        GetResource(RES_SCHEDULER);
        GetEvent(TASK_ID_OsTask_Periodic, &eventMask);

        /** Alarm 10ms – Main functions */
        if (eventMask & EVENT_MASK_OsEvent_10ms) {
            Can_MainFunction_Mode();
            CanSM_MainFunction();
            Com_MainFunctionRx();
            Com_MainFunctionTx();
            ComM_MainFunction(COMM_NETWORK_HANDLE_ComMChannel);
            CDD_Macan_MainFunction();
        }

        /** Alarm 500ms */
        if (eventMask & EVENT_MASK_OsEvent_500ms) {
            CDD_Macan_MainFunction_Hk();
        }
        ClearEvent(EVENT_MASK_OsEvent_500ms | EVENT_MASK_OsEvent_10ms);
        ReleaseResource(RES_SCHEDULER);
    }
}

```

---

Listing 5.8: `Os_TaskPeriodic` task

We have also configured one counter, which is used in MaCAN implementation to read time. The counter is increased every 1 ms.

### 5.6.2.9 RTE

In the configuration of the RTE, we have instantiated prototypes of the Software Components and mapped events to the `RteTask` mentioned above. The generated code of the task is listed in Listing 5.9.

---

```

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

        if (Event & EVENT_MASK_DataReceivedEvent) {
            SYS_CALL_ClearEvent (EVENT_MASK_DataReceivedEvent);
            Rte_TheConsumer_MyConsumerRunnable();
        }
        if (Event & EVENT_MASK_StepEvent) {
            SYS_CALL_ClearEvent (EVENT_MASK_StepEvent);
            Rte_TheProducer_MyProducerRunnable();
        }
    } while (RTE_EXTENDED_TASK_LOOP_CONDITION);
}

```

---

Listing 5.9: RteTask task

### 5.6.3 Running the demo

The demo was compiled using a generated makefile in the ECU configuration project for the STM3210C-EVAL board and the binary file was downloaded to the board using OpenOCD. On GNU/Linux we have launched the keyserver, the timeserver and NODE2, which was configured to send 1 and 0 alternately as value of `SIGNAL_B` which caused blinking of the blue LED on the board. NODE1 (running on the board) was sending `SIGNAL_A` as reaction on the button press/release. The signal value was displayed in the console by NODE2.

## Chapter 6

# Conclusion

In this work, we focused on the MaCAN protocol and its integration into the AUTOSAR architecture. We have presented an augmented specification of the protocol with description of frame formats, that were missing in the original paper [14].

Next, we performed tests with two implementations of the MaCAN protocol – one developed at CTU and another developed by Volkswagen. These implementations were initially not compatible. To find the incompatibilities, we have created a demo application and analyzed the communication between the nodes. Every incompatibility represented an obstacle in the communication flow, so to move forward, we had to modify our implementation progressively. In the end, we managed to make these implementations work together and presented a list of incompatibilities we have found. Then, our implementation was subject to a resource usage analysis, which determined size of an additional memory needed when using MaCAN protocol in contrast to traditional CAN communication. We have also measured running times of cryptographic functions. These tests showed that cryptographic functions run much faster on a dedicated hardware cryptographic coprocessor SHE.

Before integration into the AUTOSAR architecture we also had to port our implementation of the MaCAN protocol to the STM32 architecture. This step was not planned initially, but because Arctic Core was implemented neither for GNU/Linux nor for Tricore TC1798, we had to find a different hardware with a compatible architecture. Finally we successfully ported and tested the implementation on the STM3210C-EVAL board.

In order to understand how to integrate MaCAN into AUTOSAR we had to study the AUTOSAR architecture and its communication stack in great detail. Based on this gained knowledge, we have decided to create a Complex Device Driver module, which encapsulates the functionality of our MaCAN implementation. The module is embedded within the communication stack and it is able to secure signals without impact on higher layers of the AUTOSAR architecture. We have created and successfully tested an implementation of this module using a simple demo. Since this is the first prototype implementation, there are many opportunities for improvements including: implementation of an AUTOSAR Interface, creation of a GUI based configuration tool or implementation of all requirements imposed on Complex Device Drivers (described in [7]).





# Bibliography

- [1] AUTOSAR. *AUTOSAR - The Worldwide Automotive Standard for E/E Systems* [online]. 2014. [cit. 27.4.2014]. Available at: [http://www.autosar.org/download/papersandpresentations/AUTOSAR\\_Brochure\\_EN.pdf](http://www.autosar.org/download/papersandpresentations/AUTOSAR_Brochure_EN.pdf).
- [2] AUTOSAR. *AUTOSAR Home* [online]. 2014. [cit. 26.4.2014]. Available at: <http://www.autosar.org/index.php?p=0&up=0&uup=0&uuup=0>.
- [3] AUTOSAR. *AUTOSAR Layered software architecture*. January 2013.
- [4] AUTOSAR. *Software Component Template*. January 2013.
- [5] AUTOSAR. *Virtual Function Bus*. January 2013.
- [6] AUTOSAR. *Specification of CAN Interface*. January 2013.
- [7] AUTOSAR. *Complex driver design and integration guideline*. January 2013.
- [8] AUTOSAR. *Specification of PDU Router*. January 2013.
- [9] AUTOSAR. *Specification of Communication stack types*. January 2013.
- [10] AUTOSAR. *Specification of PDU Router*. January 2013.
- [11] BOSCH. *CAN with Flexible Data-Rate*. January 2013.
- [12] BRUNI, A. et al. *Formal Security Analysis of the MaCAN Protocol.*, s. 241–255. Lecture Notes in Computer Science. Springer, 2014. doi: 10.1007/978-3-319-10181-1\_15. ISBN 978-3-319-10180-4.
- [13] DWORKIN, M. J. SP 800-38B. Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication. 2005.
- [14] HARTKOPP, O. – SCHILLING, R. MaCAN – Message Authenticated CAN. In *Escar Conference*, Berlin, Germany, November 2012.
- [15] MIKULKA, L. *Low-level software for automotive electronic control units*. Czech Technical University in Prague, 2013.
- [16] MILLER, C. – VALASEK, C. Adventures in Automotive Networks and Control Units. *Last Accessed from [http://illmatics.com/car\\_hacking.pdf](http://illmatics.com/car_hacking.pdf) on*. 2013, 13.

- [17] SCHAAD, J. – HOUSLEY, R. RFC 3394, Advanced Encryption Standard (AES) Key Wrap Algorithm. *Internet Engineering Task Force*. 2002.
- [18] VAN HERREWEGE, A. – SINGELEEE, D. – VERBAUWHEDE, I. CANAuth-a simple, backward compatible broadcast authentication protocol for CAN bus. In *ECRYPT Workshop on Lightweight Cryptography 2011*, 2011.
- [19] Vector Informatik GmbH. *Introduction to AUTOSAR* [online]. 2014. [cit. 26. 4. 2014]. Available at: [https://elearning.vector.com/vl\\_autosar\\_introduction\\_portal\\_en.html](https://elearning.vector.com/vl_autosar_introduction_portal_en.html).
- [20] Vector Informatik GmbH. *Introduction to CAN* [online]. 2014. [cit. 20. 4. 2014]. Available at: [https://elearning.vector.com/vl\\_can\\_introduction\\_en.html](https://elearning.vector.com/vl_can_introduction_en.html).
- [21] WIKIPEDIA. Can bus, 2014. Available at: [http://en.wikipedia.org/wiki/CAN\\_bus](http://en.wikipedia.org/wiki/CAN_bus). [Online; accessed 22-April-2014].
- [22] WOLF, M. – WEIMERSKIRCH, A. – PAAR, C. Security in automotive bus systems.
- [23] ZIERMANN, T. – WILDERMANN, S. – TEICH, J. CAN+: A new backward-compatible Controller Area Network (CAN) protocol with up to 16× higher data rates. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, s. 1088–1093. IEEE, 2009.

## Chapter 7

# List of Abbreviations

**AES** Advanced encryption standard

**BASH** Bourne Again SHell

**BRS** Bit Rate Switch

**BSW** Basic Software

**CAN** Controller Area Network

**CAN-FD** CAN with Flexible Data rate

**CDD** Complex Device Driver

**CMAC** Cipher based Message Authentication Code

**CPU** Central Processing Unit

**CRC** Cyclic Redundancy Check

**CSMA/CR** Carrier Sense Multiple Access with Collision Resolution

**DLC** Data Length Code

**ECU** Electronic Control Unit

**EDL** Extended Data Length

**ELF** Executable and Linkable format

**FIFO** First In First Out

**GCC** GNU Compiler Collection

**GDB** GNU DeBugger

**GPIO** General Purpose Input Output

**GUI** Graphic User Interface

**HMAC** Keyed-hash Message Authentication Code

**IDE** Integrated Development Environment

**JSON** JavaScript Object Notation

**JTAG** Joint Test Action Group

**KS** Keyserver

**LCD** Liquid Crystal Display

**LED** Light Emitting Diode

**LIN** Local Interconnect Network

**LTK** Long-Term Key

**MAC** Message Authentication Code

**MCAL** Microcontroller Abstraction Layer

**MOST** Media Oriented Systems Transport

**OSEK** Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen

**OTG** On The Go

**PDU** Protocol Data Unit

**RTE** RealTime Environment

**SWC** Software Component

**TS** Timeserver

**USART** Universal Synchronous Asynchronous Receiver and Transmitter

**USB** Universal Serial Bus

**VFB** Virtual Function Bus

## Chapter 8

# Enclosed CD table of contents

- `arctic-core` - source code of Arctic Core with `CDD_Macan` module.
- `ctu-macan` - source codes of our macan implementation
- `thesis` - this thesis in PDF and Latex format



# Appendices





## Appendix A

# Guide – Creating demo project in Arctic Studio

This guide describes how to create a simple AUTOSAR application in Arctic Core studio. This application consists of two software components, which communicate with outer world using authenticated CAN frames. The demo is based on **1signal** demo from MaCAN repository. This AUTOSAR node takes role of NODE1 and communicates with the keyserver, the timeserver and NODE2 which run on the GNU/Linux machine.

### A.1 Downloading Arctic Core source code

1. Clone source code of Arctic Core with MaCAN module from remote GIT repository:

```
git clone git@rtime.felk.cvut.cz:arc-core-macan
cd arc-core-macan/core/communication/CDD_Macan/macan
```

2. It is also necessary to initialize Macan submodule:

```
git submodule update --init
```

### A.2 Preparing Arctic Studio

1. Download **Arctic Studio 4.1.0** for **ARM-Architecture** from Arc Core website
2. Launch **Arctic Studio.exe**
3. Create empty directory which will serve as a workspace
4. Select **File > Switch workspace > Other...** and select the directory created in the previous step and click **OK**
5. Select **File > Import > General > Existing Projects into Workspace** and click **Next**

6. Select the directory where you cloned Arctic Core source code
7. A list of projects will appear. Select only **core** and **Boards** projects
8. Click **Finish**

### A.3 Creating application model

Application model needs to be created first. It is a high level and hardware independent description of SWCs, their interfaces and how are they communicating with each other. This model will be then exported in a form of XML into ECU configuration project.

#### A.3.1 Create project

1. Click **File > New > Other...**
2. Select **ArcCore AUTOSAR Project** under **ArcCore** and click **Next**
3. Name the project "**example-application**"
4. Click **Finish**
5. The workbench will suggest opening the AUTOSAR perspective, it is recommended to accept.
6. Right-click on the project and select **Properties**
7. In **Project References** select **core** project and click **OK**

#### A.3.2 Create interfaces

1. Click **File > New > Other...**
2. Select **ARText swcd File** under **Other** and click **Next**
3. Make sure **Container** points to the **example-application** project
4. Set **File Name** to "**Interfaces**"
5. Set **Package Name** to "**Example.Interfaces**"
6. Click **Finish**
7. Open the new file by double clicking it in **AUTOSAR Navigator** or **Project Explorer**
8. If asked to add the ARText nature to the project, click **Yes**
9. Put following contents in this file:

```

package Example.Interfaces

import AUTOSAR.Platform.ImplementationDataTypes.*

interface senderReceiver MySRInterface {
  data uint32 MyElement
}
interface clientServer MyNotifyInterface {
  operation Notify_Invalid_CMAC_Limit_Reached {

  }
}

```

### A.3.3 Create Component Types

1. Follow the steps of **Create ARText swcd File** above, but this time with
  - File Name set to "**Components**"
  - Package Name set to "**Example.Components**"
2. Open the file **Components.swcd**
3. Put following contents in this file:

```

package Example.Components

import Example.Interfaces.MySRInterface
import Example.Interfaces.MyNotifyInterface
import MyProducer.MyProducerBehavior
import MyConsumer.MyConsumerBehavior

component application MyProducer {
  ports {
    sender MySenderPort provides MySRInterface
  }
}

internalBehavior MyProducerBehavior for MyProducer {
  runnable MyProducerRunnable [1.0] {
    timingEvent 1.0
    dataWriteAccess MySenderPort.*
  }
}

implementation MyProducerImplementation for MyProducerBehavior {
  language c
  codeDescriptor "src"
}

```

```

component application MyConsumer {
  ports {
    receiver MyReceiverPort requires MySRInterface
    server MyServerPort provides MyNotifyInterface
  }
}

internalBehavior MyConsumerBehavior for MyConsumer {
  runnable MyConsumerRunnable [1.0] {
    dataReadAccess MyReceiverPort.*
    dataReceivedEvent MyReceiverPort.MyElement as DataReceivedEvent
  }
  runnable LedRunnable [1.0] {
    timingEvent 1.0
  }
  runnable Notify_Invalid_CMAC_Limit_Reached [1.0] {
    operationInvokedEvent MyServerPort.Notify_Invalid_CMAC_Limit_Reached
  }
}

implementation MyConsumerImplementation for MyConsumerBehavior {
  language c
  codeDescriptor "src"
}

```

### A.3.4 Create an Ecu Extract

1. Follow the steps of **Create ARText swcd File** above, but this time with
  - File Name set to "**EcuExtract**"
  - Package Name set to "**Example.EcuExtract**"
2. Open the file **EcuExtract.swcd**
3. Put following contents in this file:

```

package Example.EcuExtract

import Example.Components.MyProducer
import Example.Components.MyConsumer
import Example.Types.*

@EcuExtract
composition MyEcuExtract {

  @ImplMapping
  prototype MyProducer TheProducer
  @ImplMapping
  prototype MyConsumer TheConsumer
}

```

```
ports {
    @SignalMappings
    provides TheProducer.MySenderPort Tx1
    @SignalMappings
    requires TheConsumer.MyReceiverPort Rx1
}

connect TheProducer.MySenderPort to TheConsumer.MyReceiverPort
}
```

## A.4 Creating an Ecu Configuration Project

Ecu configuration project is ECU specific project used to configure various components of the ECU and generate source code from application model.

### A.4.1 Create project

1. Click **File > New > C Project**
2. Set **Project name** to "example-ecu"
3. Set **Project type** to **Empty Arctic Core Project**
4. Set **Toolchain** to **Core Builder ARM Toolchain**
5. Click Next
6. Point **Arctic Core source path** to **core** dir located in cloned repository
7. Set **AUTOSAR Version** to **4.1.1**
8. Click Finish
9. When prompted, select the target board **stm32\_stm3210c**
10. Click OK
11. Right-click on the project and select **Properties**
12. In **Project References** select **Boards** project and click **OK**

### A.4.2 Import Application Model

1. Switch to **AUTOSAR perspective**
2. Select **example-application** in **Autosar Navigator**
3. Click **Project > Rebuild Project**

4. Expand **Merged Model** element and select **AUTOSAR** and **Example** packages
5. Right click on selected packages and select **Export to AUTOSAR XML**
6. Set project to **example-ecu** and File Name to **Model.arxml**
7. Click **OK**

### A.4.3 Create Ecu Configuration File

1. Click **File > New > ArcCore Autosar File**
2. Set **Parent folder** to the **example-ecu** project
3. Set File name to **"EcuConfiguration.arxml"**
4. Select **Create Ecu Configuration** and set the name to **"MyEcu"**
5. Select **With an ARPackage** and set the name to **"Example"**
6. Click **Finish**
7. Open the file with the **BSW Editor** by double clicking it
8. Set **ecuExtact** in the ECU Information and options section by clicking the tree button
9. In the dialog, select **MyEcuExtractEcuExtract** and click **OK**

## A.5 Configuring the ECU

Once the application model is imported, it is time to configure modules of the ECU in the BSW Editor.

### A.5.1 Set MCAL

1. Click on a drop down list in **MCAL** option and select **STM32F107**

### A.5.2 Can module

1. Select **Import > Module...**
2. Select **/Boards/stm32\_stm3210c.arxml > Can** and click **Finish**
3. Double click on **CAN** module to open BSW Editor
4. Change following settings:

Location	Setting name	Change to
Can > CanConfigSet > CanHardwareObject:s > CAN_HRH_A1	Can Id Type	STANDARD
Can > CanConfigSet > CanHardwareObject:s > CAN_HTH_A1	Can Id Type	STANDARD

### A.5.3 EcuC module

1. Click **Add** and select **EcuC** module
2. Click **OK**
3. Double click on **EcuC** module to open BSW Editor
4. Right click on **EcucPduCollection** and select **create Pdu**
5. Click on newly created element
6. Set **Pdu shortName** to **TxPdu**
7. Set **Pdu Length** to **8**
8. Using the same way, create PDUs with following names:


Pdu shortName	Pdu Length
MacanTxPdu7	8
MacanTxPdu8	8
MacanTxPduSIGNAL_A	8
RxPdu	8
MacanRxPduKEY_SERVER	8
MacanRxPduNODE2	8
MacanRxPduSIGNAL_B	8
MacanRxPduTIME_SERVER	8

### A.5.4 CanIf module

NOTE: When setting values using the tree icon, always choose items from **Example** package, unless specified otherwise.

1. Click **Add** and select **CanIf** module
2. Click **OK**
3. Double click on **CanIf** module to open BSW Editor
4. Click on element **CanIfCtrlCfg**
5. Adjust values of the element as in the figure below:

CanIfCtrlCfg shortName



Can If Ctrl Can Ctrl Ref  

6. Right click on element **CanIfInitCfg** and select **Create CanInitHohCfg**





7. Right click on element **CanIfInitHohCfg** and select **Create CanHrhCfg**

8. Click on newly created element and set the values as follows:

CanIfHrhCfg shortName	<input type="text" value="CanIfHrhCfg"/>
Can If Hrh Can Ctrl Id Ref	<input type="text" value="bxCAN"/> 
Can If Hrh Id Sym Ref	<input type="text" value="CAN_HRH_A_1"/> 
Can If Hrh Software Filter	<input type="text" value="True"/>


9. Right click on element **CanIfInitHohCfg** and select **Create CanHthCfg**

10. Click on newly created element and set the values as follows:

CanIfHthCfg shortName	<input type="text" value="CanIfHthCfg"/>
Can If Hth Can Ctrl Id Ref	<input type="text" value="bxCAN"/> 
Can If Hth Id Sym Ref	<input type="text" value="CAN_HTH_A_1"/> 



11. Right click on element **CanIfInitCfg** and select **Create CanIfBufferCfg**

12. Click on newly created element and set the values as follows:

CanIfBufferCfg shortName	<input type="text" value="CanIfBufferCfg"/>
Can If Buffer Hth Ref	<input checked="" type="checkbox"/> <input type="button" value="+"/> <input type="text" value="CanIfHthCfg"/> 
Can If Buffer Size	<input type="text" value="2"/>



13. Right click on element **CanIfInitCfg** and select **Create CanIfTxPduCfg**

14. Click on newly created element and set the values as follows:

<b>General Settings</b>	
CanIfTxPduCfg shortName	<input type="text" value="TxFrame"/>
Can If Tx Pdu Buffer Ref	<input type="text" value="CanIfBufferCfg"/> 
Can If Tx Pdu Ref	<input type="text" value="TxPdu"/> 
Can If Tx Pdu Type	<input type="text" value="STATIC"/>
<b>CAN Properties</b>	
Can If Tx Pdu Can Id	<input type="text" value="2"/>
Can If Tx Pdu Can Id Type	<input type="text" value="STANDARD_CAN"/>
Can If Tx Pdu Dlc	<input type="text" value="8"/>
<b>Transmit Confirmation Settings</b>	
Can If Tx Pdu User Tx Confirmation Name	<input type="checkbox"/> <input type="text"/>
Can If Tx Pdu User Tx Confirmation UL	<input checked="" type="checkbox"/> <input type="text" value="PDUR"/>

15. Right click on element **CanIfInitCfg** and select **Create CanIfTxPduCfg**

16. Click on newly created element and set the values as follows:

General Settings	
CanIfTxPduCfg shortName	MacanTxFrameSignalA
Can If Tx Pdu Buffer Ref	CanIfBufferCfg 
Can If Tx Pdu Ref	MacanTxPduSIGNAL_A 
Can If Tx Pdu Type	STATIC

CAN Properties	
Can If Tx Pdu Can Id	512
Can If Tx Pdu Can Id Type	STANDARD_CAN
Can If Tx Pdu Dlc	8

Transmit Confirmation Settings	
Can If Tx Pdu User Tx Confirmation Name <input checked="" type="checkbox"/>	CDD_Macan_CanIfTxConfirmation
Can If Tx Pdu User Tx Confirmation UL <input checked="" type="checkbox"/>	CDD



17. Right click on element **CanIfInitCfg** and select **Create CanIfTxPduCfg**

18. Click on newly created element and set the values as follows:

General Settings	
CanIfTxPduCfg shortName	MacanTxFrame8
Can If Tx Pdu Buffer Ref	CanIfBufferCfg
Can If Tx Pdu Ref	MacanTxPdu8
Can If Tx Pdu Type	STATIC
CAN Properties	
Can If Tx Pdu Can Id	258
Can If Tx Pdu Can Id Type	STANDARD_CAN
Can If Tx Pdu Dlc	8
Transmit Confirmation Settings	
Can If Tx Pdu User Tx Confirmation Name	<input checked="" type="checkbox"/> CDD_Macan_CanIfTxConfirmation
Can If Tx Pdu User Tx Confirmation UL	<input checked="" type="checkbox"/> CDD

19. Right click on element **CanIfInitCfg** and select **Create CanIfTxPduCfg**

20. Click on newly created element and set the values as follows:

General Settings	
CanIfTxPduCfg shortName	MacanTxFrame7
Can If Tx Pdu Buffer Ref	CanIfBufferCfg 
Can If Tx Pdu Ref	MacanTxPdu7 
Can If Tx Pdu Type	STATIC

CAN Properties	
Can If Tx Pdu Can Id	258
Can If Tx Pdu Can Id Type	STANDARD_CAN
Can If Tx Pdu Dlc	7

Transmit Confirmation Settings	
Can If Tx Pdu User Tx Confirmation Name	<input checked="" type="checkbox"/> CDD_Macan_CanIfTxConfirmation
Can If Tx Pdu User Tx Confirmation UL	<input checked="" type="checkbox"/> CDD

21. Right click on element **CanIfInitCfg** and select **Create CanIfRxPduCfg**

22. Click on newly created element and set the values as follows:

General Settings	
CanIfRxPduCfg shortName	RxFrame
Can If Rx Pdu Hrh Id Ref	<input checked="" type="checkbox"/> + CanIfHrhCfg
Can If Rx Pdu Ref	RxPdu
CAN Properties	
Can If Rx Pdu Can Id	<input checked="" type="checkbox"/> 1
Can If Rx Pdu Can Id Type	STANDARD_CAN
Can If Rx Pdu Dlc	8
Receive Indication Settings	
Can If Rx Pdu User Rx Indication Name	<input type="checkbox"/>
Can If Rx Pdu User Rx Indication UL	<input checked="" type="checkbox"/> PDUR

23. Right click on element **CanIfInitCfg** and select **Create CanIfRxPduCfg**

24. Click on newly created element and set the values as follows:

General Settings	
CanIfRxPduCfg shortName	MacanRxFrameTS
Can If Rx Pdu Hrh Id Ref	<input checked="" type="checkbox"/> + CanIfHrhCfg
Can If Rx Pdu Ref	MacanRxPduTIME_SERVER
CAN Properties	
Can If Rx Pdu Can Id	<input checked="" type="checkbox"/> 0
Can If Rx Pdu Can Id Type	STANDARD_CAN
Can If Rx Pdu Dlc	8
Receive Indication Settings	
Can If Rx Pdu User Rx Indication Name	<input checked="" type="checkbox"/> CDD_Macan_CanIfRxIndication
Can If Rx Pdu User Rx Indication UL	<input checked="" type="checkbox"/> CDD

25. Right click on element **CanIfInitCfg** and select **Create CanIfRxPduCfg**

26. Click on newly created element and set the values as follows:

General Settings	
CanIfRxPduCfg shortName	MacanRxFrameSignalB
Can If Rx Pdu Hrh Id Ref	<input checked="" type="checkbox"/> + CanIfHrhCfg
Can If Rx Pdu Ref	MacanRxPduSIGNAL_B
CAN Properties	
Can If Rx Pdu Can Id	<input checked="" type="checkbox"/> 513
Can If Rx Pdu Can Id Type	STANDARD_CAN
Can If Rx Pdu Dlc	8
Receive Indication Settings	
Can If Rx Pdu User Rx Indication Name	<input checked="" type="checkbox"/> CDD_Macan_CanIfRxIndication
Can If Rx Pdu User Rx Indication UL	<input checked="" type="checkbox"/> CDD

27. Right click on element **CanIfInitCfg** and select **Create CanIfRxPduCfg**

28. Click on newly created element and set the values as follows:

General Settings	
CanIfRxPduCfg shortName	MacanRxFrameN3
Can If Rx Pdu Hrh Id Ref	<input checked="" type="checkbox"/> + CanIfHrhCfg
Can If Rx Pdu Ref	MacanRxPduNODE2
CAN Properties	
Can If Rx Pdu Can Id	<input checked="" type="checkbox"/> 259
Can If Rx Pdu Can Id Type	STANDARD_CAN
Can If Rx Pdu Dlc	8
Receive Indication Settings	
Can If Rx Pdu User Rx Indication Name	<input checked="" type="checkbox"/> CDD_Macan_CanIfRxIndication
Can If Rx Pdu User Rx Indication UL	<input checked="" type="checkbox"/> CDD

29. Right click on element **CanIfInitCfg** and select **Create CanIfRxPduCfg**

30. Click on newly created element and set the values as follows:

General Settings	
CanIfRxPduCfg shortName	MacanRxFrameKS
Can If Rx Pdu Hrh Id Ref	<input checked="" type="checkbox"/> + CanIfHrhCfg
Can If Rx Pdu Ref	MacanRxPduKEY_SERVER
CAN Properties	
Can If Rx Pdu Can Id	<input checked="" type="checkbox"/> 256
Can If Rx Pdu Can Id Type	STANDARD_CAN
Can If Rx Pdu Dlc	8
Receive Indication Settings	
Can If Rx Pdu User Rx Indication Name	<input checked="" type="checkbox"/> CDD_Macan_CanIfRxIndication
Can If Rx Pdu User Rx Indication UL	<input checked="" type="checkbox"/> CDD



31. Click on element **CanIfCtrlDrvCfg** and in option **Can If Ctrl Drv Init Hoh Config Ref** select **CanIfInitHohCfg**
32. Click on element **CanIfDispatchCfg**
33. In option **Can If Dispatch User Ctrl Bus Off UL** select **CAN SM**
34. In option **Can If Dispatch User Ctrl Mode Indication UL** select **CAN SM**
35. Click on element **CanIfPublicCfg**
36. Set option **Arc Can If Public Max Number Of Tx Buffers** to **1**
37. Set option **Arc Can If Public Tx Buffer Size** to **2**
38. Set option **Can If Public Tx Buffering** to **True**
39. Click on **CanIfPrivateCfg** and set **Can If Private Dlc Check** to **False**

#### A.5.5 ComM module

1. Click **Add** and select **ComM** module
2. Click **OK**
3. Double click on **ComM** module to open BSW Editor
4. Click on element **ComMChannel**
5. Set the values as follows:

General Settings	
ComMChannel shortName	ComMChannel
Com MBus Type	COMM_BUS_TYPE_CAN
Com MMain Function Period	0.01
Com MNo Com	False

6. Right click on element **ComMChannel** and select **Create ComMUserPerChannel**
7. Click on newly created element
8. Set option **Com MUser Channel** to **ComMUser**
9. Click on **ComMNetworkManagement** element
10. Set **Com MNm Varian** to **NONE**

### A.5.6 CanSM module

1. Click **Add** and select **CanSM** module
2. Click **OK**
3. Double click on **CanSM** module to open BSW Editor
4. Click on element **CanSMConfiguration**
5. Set option **Can SM Mode Request Repetition Time** to **10**
6. Click on element **CanSMManagerNetwork**
7. Set the values as follows:

<b>CanSMManagerNetwork</b>	
<b>General Settings</b>	
CanSMManagerNetwork shortName	CanSMManagerNetwork
Can SM Com MNetwork Handle Ref	ComMChannel
<b>Bus-off Recovery</b>	
Can SM Bor Counter L1To L2	10
Can SM Bor Time L1	1
Can SM Bor Time L2	1
Can SM Bor Time Tx Ensured	2

8. Click on element **CanSMController**
9. Set option **Can SM Controller Id** to **bxCAN**
10. Click on element **CanSMGeneral**
11. Set option **Can SM Main Function Period** to **0.01**

### A.5.7 Det module

1. Click **Add** and select **Det** module
2. Click **OK**
3. There is no need to change any default settings

### A.5.8 Dio module

1. Select **Import > Module...**
2. Select **/Boards/stm32\_stm3210c.arxml > Dio** and click **Finish**
3. There is no need to change any default settings

### A.5.9 EcuM module

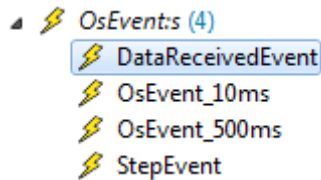
1. Click **Add** and select **EcuM** module
2. Click **OK**
3. Double click on **EcuM** module to open BSW Editor
4. Click on element **EcuMSleepMode**
5. Set option **Ecu MSleep Mode Mcu Mode Ref** to **SLEEP**, but make sure to select the one from **ArcCore > Boards > stm32\_stm3210c > Mcu > McuModule-Configuration**
6. Set option **Ecu MWakeup Source Mask** to **EcuMWakeupSource**, but make sure to select the one from **Example** package
7. Click on element **EcuMFixedConfiguration**
8. Set option **Ecu MCom MCommunication Allowed List** to **ComMChannel**
9. Set option **Ecu MNormal Mcu Mode Ref** to **NORMAL**, but make sure to select the one from **ArcCore > Boards > stm32\_stm3210c > Mcu > McuModule-Configuration**
10. Click on element **EcuMGeneral**
11. Set option **Ecu MMain Function Period** to **0.01**
12. Right click on element **EcuM** and select **Create EcuMFixedGeneral**

### A.5.10 Mcu module

1. Select **Import > Module...**
2. Select **/Boards/stm32\_stm3210c.arxml > Mcu** and click **Finish**
3. There is no need to change any default settings

### A.5.11 Os module

1. Click **Add** and select **Os** module
2. Click **OK**
3. Double click on **Os** module to open BSW Editor
4. Right click on element **Os** and select **Create OsCounter**
5. Click on newly created element
6. Set option **Os Counter Type** to **OS\_TICK**
7. Right click on element **Os** and select **Create OsEvent**
8. Following previous step, create these OsEvents (only **OsEvent shortName** needs to be changed):



9. Right click on element **Os** and select **Create OsTask**
10. Click on newly created element and set the values as follows:

General Settings	
OsTask shortName	OsTask_Init
Arc Os Task Stack Size	2048
Os Task Accessing Application	<input type="checkbox"/> + <input type="text"/> <input type="button" value="Add"/>
Os Task Activation	1
Os Task Event Ref	<input type="checkbox"/> + <input type="text"/> <input type="button" value="Add"/>
Os Task Priority	1
Os Task Resource Ref	<input type="checkbox"/> + <input type="text"/> <input type="button" value="Add"/>
Os Task Schedule	FULL <input type="button" value="v"/>

11. Right click on element **Os** and select **Create OsTask**

12. Click on newly created element and set the values as follows:

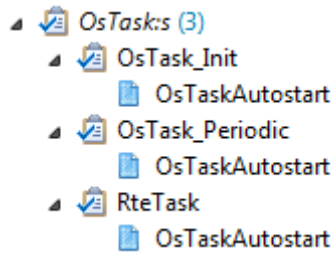
OsTask shortName	OsTask_Periodic
Arc Os Task Stack Size	2048
Os Task Accessing Application	<input type="checkbox"/> + <input type="text"/> <input type="button" value="..."/>
Os Task Activation	1
Os Task Event Ref	<input checked="" type="checkbox"/> + OsEvent_500ms <input type="button" value="..."/>
Os Task Priority	1
Os Task Resource Ref	<input type="checkbox"/> + <input type="text"/> <input type="button" value="..."/>
Os Task Schedule	FULL <input type="button" value="v"/>

13. Right click on element **Os** and select **Create OsTask**

14. Click on newly created element and set the values as follows:

<b>General Settings</b> <input type="button" value="..."/>	
OsTask shortName	RteTask
Arc Os Task Stack Size	2048
Os Task Accessing Application	<input type="checkbox"/> + <input type="text"/> <input type="button" value="..."/>
Os Task Activation	1
Os Task Event Ref	<input checked="" type="checkbox"/> + DataReceivedEvent <input type="button" value="..."/>
Os Task Event Ref	<input checked="" type="checkbox"/> + StepEvent <input type="button" value="..."/>
Os Task Priority	1
Os Task Resource Ref	<input type="checkbox"/> + <input type="text"/> <input type="button" value="..."/>
Os Task Schedule	FULL <input type="button" value="v"/>

15. Right click on each created task and select **Create OsTaskAutostart**, so the tree on the left side looks like this:



16. Right click on element **Os** and select **Create OsAlarm**

17. Click on newly created element and set the values as follows:

**General Settings**

OsAlarm shortName	<input type="text" value="OsAlarm_10ms"/>
Os Alarm Accessing Application	<input type="checkbox"/> + <input type="text"/> <input type="button" value="..."/>
Os Alarm Counter Ref	<input type="text" value="OsCounter"/> <input type="button" value="..."/>

18. Right click on **OsAlarmAction** located under this newly created alarm and select **Create OsAlarmSetEvent**

19. Click on newly created element and set the values as follows:

**General Settings**

OsAlarmSetEvent shortName	<input type="text" value="OsAlarmSetEvent"/>
Os Alarm Set Event Ref	<input type="text" value="OsEvent_10ms"/> <input type="button" value="..."/>
Os Alarm Set Event Task Ref	<input type="text" value="OsTask_Periodic"/> <input type="button" value="..."/>

20. Right click on this newly created alarm and select **Create OsAlarmAutostart**

21. Click on newly created element and set the values as follows:



**General Settings**

OsAlarmAutostart shortName	OsAlarmAutostart
Os Alarm Alarm Time	10
Os Alarm Autostart Type	ABSOLUTE
Os Alarm Cycle Time	10

22. Right click on element **Os** and select **Create OsAlarm**



23. Click on newly created element and set the values as follows:

**General Settings**

OsAlarm shortName	OsAlarm_500ms
Os Alarm Accessing Application	<input type="checkbox"/> + <input type="text"/> 
Os Alarm Counter Ref	OsCounter 

24. Right click on **OsAlarmAction** located under this newly created alarm and select **Create OsAlarmSetEvent**

25. Click on newly created element and set the values as follows:

OsAlarmSetEvent shortName	OsAlarmSetEvent
Os Alarm Set Event Ref	OsEvent_500ms 
Os Alarm Set Event Task Ref	OsTask_Periodic 


26. Right click on this newly created alarm and select **Create OsAlarmAutostart**

27. Click on newly created element and set the values as follows:

General Settings	
OsAlarmAutostart shortName	OsAlarmAutostart
Os Alarm Alarm Time	150
Os Alarm Autostart Type	ABSOLUTE
Os Alarm Cycle Time	500



28. Right click on element **Os** and select **Create OsAlarm**

29. Click on newly created element and set the values as follows:

General Settings	
OsAlarm shortName	StepAlarm
Os Alarm Accessing Application	<input type="checkbox"/> + <input type="text"/> 
Os Alarm Counter Ref	OsCounter 

30. Right click on **OsAlarmAction** located under this newly created alarm and select **Create OsAlarmSetEvent**

31. Click on newly created element and set the values as follows:

General Settings	
OsAlarmSetEvent shortName	OsAlarmSetEvent
Os Alarm Set Event Ref	StepEvent 
Os Alarm Set Event Task Ref	RteTask 

32. Right click on this newly created alarm and select **Create OsAlarmAutostart**

33. Click on newly created element and set the values as follows:



**General Settings**

OsAlarmAutostart shortName	OsAlarmAutostart
Os Alarm Alarm Time	100
Os Alarm Autostart Type	ABSOLUTE
Os Alarm Cycle Time	10

34. Right click on element **Os** and select **Create OsApplication**

35. Click on newly created element and set the values as follows:

**General Settings**

OsApplication shortName	OsApplication
Os App Alarm Ref	<input checked="" type="checkbox"/> + OsAlarm_500ms
Os App Alarm Ref	<input checked="" type="checkbox"/> + OsAlarm_10ms
Os App Alarm Ref	<input checked="" type="checkbox"/> + StepAlarm
Os App Counter Ref	<input checked="" type="checkbox"/> + OsCounter
Os App Isr Ref	<input type="checkbox"/> +
Os App Schedule Table Ref	<input type="checkbox"/> +
Os App Task Ref	<input checked="" type="checkbox"/> + OsTask_Periodic
Os App Task Ref	<input checked="" type="checkbox"/> + OsTask_Init
Os App Task Ref	<input checked="" type="checkbox"/> + RteTask
Os Application Core Assignment	<input type="checkbox"/>
Os Trusted	True

36. Click on element **OsOs**

37. Change **Arc Os Tick Frequency** to **1000**

38. Change **Os Status** to **STANDARD**

### A.5.12 PduR module

1. Click **Add** and select **PduR** module
2. Click **OK**
3. Double click on **PduR** module to open BSW Editor
4. Right click on element **PduR** and select **Create PduRBswModules**
5. Click on newly created element and set the values as follows:

General Settings	
PduRBswModules shortName	CanIf
Pdu RLower Module	True
Pdu RUpper Module	False

6. Right click on element **PduR** and select **Create PduRBswModules**
7. Click on newly created element and set the values as follows:

General Settings	
PduRBswModules shortName	CDD_Macan
Pdu RLower Module	True
Pdu RUpper Module	False



8. Right click on element **PduR** and select **Create PduRBswModules**
9. Click on newly created element and set the values as follows:

General Settings	
PduRBswModules shortName	Com
Pdu RLower Module	False
Pdu RUpper Module	True

10. Click on **PduRRoutingTables** and set **Pdu RConfiguration Id** to 1


11. Right click on **PduRRoutingTables** and select **Create PduRRoutingTable**
12. Right click on **PduRRoutingTable** and select **Create PduRRoutingPath**
13. Click on this newly created element and set **PduRRoutingPath shortName** to **RxPath**
14. Click on **PduRDestPdu** under this routing path and set the values as follows:

**General Settings**

PduRDestPdu shortName	<input type="text" value="PduRDestPdu"/>
Arc Overriden Dest Module	<input checked="" type="checkbox"/> <input type="text" value="Com"/>
Pdu RDest Pdu Data Provision	<input checked="" type="checkbox"/> <input type="text" value="PDUR_DIRECT"/>
Pdu RDest Pdu Ref	<input type="text" value="RxPdu"/> 
Pdu RDest Tx Buffer Ref	<input type="text"/> 
Pdu RTp Threshold	<input type="text"/>

15. Click on **PduRSrcPdu** under this routing path and set the values as follows:

**General Settings**

PduRSrcPdu shortName	<input type="text" value="PduRSrcPdu"/>
Arc Overriden Src Module	<input checked="" type="checkbox"/> <input type="text" value="CanIf"/>
Pdu RSrc Pdu Ref	<input type="text" value="RxPdu"/> 

16. Right click on **PduRRoutingTable** and select **Create PduRRoutingPath**
17. Click on this newly created element and set **PduRRoutingPath shortName** to **RxPathFromMacan**
18. Click on **PduRDestPdu** under this routing path and set the values as follows:

**General Settings**

PduRDestPdu shortName	<input type="text" value="PduRDestPdu"/>
Arc Overriden Dest Module	<input checked="" type="checkbox"/> <input type="text" value="Com"/>
Pdu RDest Pdu Data Provision	<input checked="" type="checkbox"/> <input type="text" value="PDUR_DIRECT"/>
Pdu RDest Pdu Ref	<input type="text" value="RxPdu"/>
Pdu RDest Tx Buffer Ref	<input type="text"/>
Pdu RTp Threshold	<input type="text"/>

19. Click on **PduRSrcPdu** under this routing path and set the values as follows:

**General Settings**

PduRSrcPdu shortName	<input type="text" value="PduRSrcPdu"/>
Arc Overriden Src Module	<input checked="" type="checkbox"/> <input type="text" value="CanIf"/>
Pdu RSrc Pdu Ref	<input type="text" value="MacanRxPduSIGNAL_B"/>

20. Right click on **PduRRoutingTable** and select **Create PduRRoutingPath**
21. Click on this newly created element and set **PduRRoutingPath** shortName to **TxPath**
22. Click on **PduRDestPdu** under this routing path and set the values as follows:

**PduRDestPdu**

**General Settings**

PduRDestPdu shortName	<input type="text" value="ToCanIf"/>
Arc Overriden Dest Module	<input checked="" type="checkbox"/> <input type="text" value="CanIf"/>
Pdu RDest Pdu Data Provision	<input checked="" type="checkbox"/> <input type="text" value="PDUR_DIRECT"/>
Pdu RDest Pdu Ref	<input type="text" value="TxPdu"/>
Pdu RDest Tx Buffer Ref	<input type="text"/>
Pdu RTp Threshold	<input type="text"/>

23. Right click on **TxPath** and click **Create PduRDestPdu**
24. Click on newly created element and set the values as follows:

General Settings	
PduRDestPdu shortName	ToCddMacan
Arc Overriden Dest Module	<input checked="" type="checkbox"/> CanIf
Pdu RDest Pdu Data Provision	<input checked="" type="checkbox"/> PDUR_DIRECT
Pdu RDest Pdu Ref	MacanTxPduSIGNAL_A
Pdu RDest Tx Buffer Ref	<input type="checkbox"/>
Pdu RTp Threshold	<input type="checkbox"/>

25. Click on **PduRSrcPdu** under this routing path and set the values as follows:

General Settings	
PduRSrcPdu shortName	PduRSrcPdu
Arc Overriden Src Module	<input checked="" type="checkbox"/> Com
Pdu RSrc Pdu Ref	TxPdu

### A.5.13 Port module

1. Select **Import > Module...**
2. Select **/Boards/stm32\_stm3210c.arxml > Port** and click **Finish**
3. There is no need to change any default settings


### A.5.14 Com module

1. Click **Add** and select **Com** module
2. Click **OK**
3. Double click on **Com** module to open BSW Editor

4. Right click on **ComConfig** element and select **Create ComSignal**
5. Click on newly created element and set the values as follows:

<b>General Settings</b>	
ComSignal shortName	<input type="text" value="Rx1"/>
Com System Template System Signal Ref	<input checked="" type="checkbox"/> <input type="text" value="Rx1MyElementMapping"/> <input type="button" value="..."/>
Com Transfer Property	<input type="checkbox"/> <input type="text"/>
<b>Signal Data Settings</b>	
Com Bit Position	<input type="text" value="0"/>
Com Bit Size	<input checked="" type="checkbox"/> <input type="text" value="32"/>
Com Signal Endianness	<input type="text" value="LITTLE_ENDIAN"/>
Com Signal Init Value	<input type="checkbox"/> <input type="text"/>
Com Signal Length	<input type="checkbox"/> <input type="text"/>
Com Signal Type	<input type="text" value="UINT32"/>
Com Update Bit Position	<input type="checkbox"/> <input type="text"/>
<b>Signal Timeout</b>	
Com First Timeout	<input type="checkbox"/> <input type="text"/>
Com Timeout	<input type="checkbox"/> <input type="text"/>
<b>Signal Actions and Notifications</b>	
Com Notification	<input checked="" type="checkbox"/> <input type="text" value="Rte_COMCbK_COM_SIGNAL_ID_RX1"/>
Com Rx Data Timeout Action	<input type="checkbox"/> <input type="text"/>
Com Timeout Notification	<input type="checkbox"/> <input type="text"/>

6. Right click on **ComConfig** element and select **Create ComSignal**
7. Click on newly created element and set the values as follows:

General Settings	
ComSignal shortName	<input type="text" value="Tx1"/>
Com System Template System Signal Ref	<input checked="" type="checkbox"/> <input type="text" value="Tx1MyElementMapping"/> 
Com Transfer Property	<input checked="" type="checkbox"/> <input type="text" value="TRIGGERED_ON_CHANGE"/>
Signal Data Settings	
Com Bit Position	<input type="text" value="0"/>
Com Bit Size	<input checked="" type="checkbox"/> <input type="text" value="32"/>
Com Signal Endianness	<input type="text" value="LITTLE_ENDIAN"/>
Com Signal Init Value	<input type="checkbox"/> <input type="text"/>
Com Signal Length	<input type="checkbox"/> <input type="text"/>
Com Signal Type	<input type="text" value="UINT32"/>
Com Update Bit Position	<input type="checkbox"/> <input type="text"/>
Signal Timeout	
Com First Timeout	<input type="checkbox"/> <input type="text"/>
Com Timeout	<input type="checkbox"/> <input type="text"/>
Signal Actions and Notifications	
Com Notification	<input type="checkbox"/> <input type="text"/>
Com Rx Data Timeout Action	<input type="checkbox"/> <input type="text"/>
Com Timeout Notification	<input type="checkbox"/> <input type="text"/>

8. Right click on **ComConfig** element and select **Create ComIPduGroup**
9. Click on newly created element and set **CanIPduGroup shortName** to **CANIP-DUs**
10. Right click on **ComConfig** element and select **Create ComIPdu**
11. Click on newly created element and set the values as follows:

General Settings	
ComIPdu shortName	ComRxIPdu
Com IPdu Callout	<input type="checkbox"/>
Com IPdu Direction	RECEIVE
Com IPdu Group Ref	<input checked="" type="checkbox"/> + CANIPDU <sub>s</sub>
Com Pdu Id Ref	RxPdu
Signals	
Com IPdu Signal Group Ref	<input type="checkbox"/> +
Com IPdu Signal Ref	<input checked="" type="checkbox"/> + Rx1
Signal Data Settings	
Com IPdu Signal Processing	IMMEDIATE

12. Right click on **ComConfig** element and select **Create ComIPdu**

13. Click on newly created element and set the values as follows:

General Settings	
ComIPdu shortName	ComTxIPdu
Com IPdu Callout	<input type="checkbox"/>
Com IPdu Direction	SEND
Com IPdu Group Ref	<input checked="" type="checkbox"/> + CANIPDU <sub>s</sub>
Com Pdu Id Ref	TxPdu
Signals	
Com IPdu Signal Group Ref	<input type="checkbox"/> +
Com IPdu Signal Ref	<input checked="" type="checkbox"/> + Tx1
Signal Data Settings	
Com IPdu Signal Processing	IMMEDIATE



14. Right click on this newly created element and select **Create ComTxIPdu**
15. Click on this newly created element and set **Com Tx Ipdu Unused Areas Default** to **0**
16. Right click on this newly created element and select **Create ComTxModeTrue**
17. Click on **ComTxMode** and set the values as follows:

General Settings	
ComTxMode shortName	ComTxMode
Com Tx Mode Mode	DIRECT
Com Tx Mode Number Of Repetitions	<input type="checkbox"/>
Com Tx Mode Repetition Period	<input type="checkbox"/>
Com Tx Mode Time Offset	<input checked="" type="checkbox"/> 0
Com Tx Mode Time Period	<input type="checkbox"/>

18. Click on **ComTimeBase** and set the values as follows:

General Settings	
ComTimeBase shortName	ComTimeBase
Com Rx Time Base	<input checked="" type="checkbox"/> 0.01
Com Tx Time Base	<input checked="" type="checkbox"/> 0.01

#### A.5.15 RTE module

1. Click **Add** and select **Rte** module
2. Click **OK**
3. Double click on **Rte**, when prompted, click on **Yes** to open RTE Editor
4. Right click on **TheConsumer** element and select **Instantiate prototypes**
5. Right click on **TheProducer** element and select **Instantiate prototypes**

6. Right click on **TheConsumer** > **DataReceivedEvent** element and select **Map events to task**
7. Select **RteTask** and click **OK**
8. In the **TheConsumer** > **DataReceivedEvent** row, select **DataReceivedEvent** in the **Event** column
9. Right click on **TheConsumer** > **timingEvent\_1\_0** element and select **Map events to task**
10. Select **RteTask** and click **OK**
11. In the **TheConsumer** > **timingEvent\_1\_0** row, select **StepEvent** in the **Event** column
12. Right click on **TheProducer** > **timingEvent\_1\_0** element and select **Map events to task**
13. Select **RteTask** and click **OK**
14. In the **TheProducer** > **timingEvent\_1\_0** row, select **StepEvent** in the **Event** column
15. Configured RTE should look like this:

Entity	Status	Runnable	Task	Event	Positi...
▲ TheConsumer	Instantiated				
DataReceivedEvent	Mapped	MyConsumerRunnable	RteTask	DataReceive...	1
MyServerPort_Notify_Inval	Unmapped	Notify_Invalid_CMAC...			
timingEvent_1_0	Mapped	LedRunnable	RteTask	StepEvent	3
▲ TheProducer	Instantiated				
timingEvent_1_0	Mapped	MyProducerRunnable	RteTask	StepEvent	2

## A.6 Generate C files from the configuration

The ECU is now configured and it is possible to generate C files from the configuration.

1. Open BSW Editor by double clicking **EcuConfiguration.arxml** in **example-ecu** project
2. Click on the gears icon in top right corner to generate the C files. This will take a while, but the process should finish with no errors.
3. Create a new directory **CDD\_Macan** the under **config** directory
4. Copy files **macan\_config.c** and **macan\_config.h** from 1signal demo

5. BASH script `CDD_Macan_generate_config.sh` from `core > communication > CDD_Macan` serves as substitution for GUI config used to configure other BSW modules. It reads Macan configuration, which was copied in previous steps and other config files to create config files for CDD\_Macan module. It also alters some already generated configuration files (of PduR, CanIf etc.) since some options, which are necessary for CDD\_Macan module to work, are not available in GUI config. There are a few parameters in the configuration file, which can be edited. You can copy the script from its original location to the project root if you wish to change the configuration parameters, but for this demo, you can stick with defaults and invoke the script from its original location. But always make sure your present working directory is project root. **Warning:** this is a BASH script and it should be launched under Linux.

```
cd example-ecu/
./../../ arc-core-macan/core/communication/CDD_Macan/
CDD_Macan_generate_config.sh
```

6. All configuration files are now generated

## A.7 Implementation C files

Last step before compilation is to implement runnables of SWCs and other tasks needed by the operating system.

### A.7.1 OsTasks.c

1. Create file `OsTasks.c` in `example-ecu` project root
2. Put following contents in the file:

```
#include "Os.h"
#include "debug.h"
#include "Can.h"
#include "CanIf.h"
#include "CanSM.h"
#include "Com.h"
#include "ComM.h"
#include "Port.h"
#include "PduR.h"
#include "Dio.h"
#include "EcuM.h"
#include "CDD_Macan.h"
#include "uart_console.h"

// needed by button init
#include "stm32f10x_rcc.h"
#include "stm32f10x_gpio.h"
```

```

static void init_button() {
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    /* Configure PD0 and PD2 in output pushpull mode */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
}

/* Initialization Task */
void OsTask_Init( void ) {
    // initialize uart
    UART_Console_Init();
    //run ini functions
    EcuM_StartupTwo();
    /** Setup Com stack with necessary parameters**/
    Com_IpduGroupVector groupVector;
    //Start the IPDU group
    Com_ClearIpduGroupVector(groupVector);
    Com_SetIpduGroup(groupVector, COM_PDU_GROUP_ID_CANIPDUS, TRUE);
    Com_IpduGroupControl(groupVector, FALSE);

    //Run CanSM & Can main function once before requesting a full communication
    CanSM_MainFunction();
    Can_MainFunction_Mode();

    //Request ComM for FULL_COMM mode
    ComM_RequestComMode(COMM_USER_HANDLE_ComMUser,
        COMM_FULL_COMMUNICATION);

    CanSM_MainFunction();

    init_button();
    TerminateTask();
}

void OsTask_Periodic( void ) {

    EventMaskType eventMask = 0;

    while (1) {
        WaitEvent(EVENT_MASK_OsEvent_500ms | EVENT_MASK_OsEvent_10ms);
        GetResource(RES_SCHEDULER);
        GetEvent(TASK_ID_OsTask_Periodic, &eventMask);

        /** Alarm 10ms - Main functions */
        if (eventMask & EVENT_MASK_OsEvent_10ms) {

            Can_MainFunction_Mode();
        }
    }
}

```

```

    CanSM_MainFunction();
    Com_MainFunctionRx();
    Com_MainFunctionTx();
    ComM_MainFunction(COMM_NETWORK_HANDLE_ComMChannel);
    CDD_Macan_MainFunction();
}

/** Alarm 500ms */
if (eventMask & EVENT_MASK_OsEvent_500ms) {

    CDD_Macan_MainFunction_Hk();

}
ClearEvent(EVENT_MASK_OsEvent_500ms | EVENT_MASK_OsEvent_10ms);
ReleaseResource(RES_SCHEDULER);
}
}

/* needed by OS */
void OsIdle( void ) { while(1){} }
void ErrorHook ( StatusType Error ) {}
void ShutdownHook ( StatusType Error ) {}
void StartupHook ( void ) {}

```

### A.7.2 TheConsumer.c

1. Create file **TheConsumer.c** in **example-ecu** project root
2. Put following contents in the file:

```

#include "Rte_MyConsumer.h"
#include "Rte_Type.h"
#include "Dio.h"

int stop_receiving = 0;
int led_mod = 50;

void MyConsumerRunnable( void ) {

    // check if we can receive data
    if(stop_receiving) {
        return;
    }

    uint32 x = Rte_IRead_MyConsumerRunnable_MyReceiverPort_MyElement();
    if(x > 0) {
        Dio_WriteChannel(DIO_CHANNEL_NAME_LED_CHANNEL4, STD_HIGH);
    } else {
        Dio_WriteChannel(DIO_CHANNEL_NAME_LED_CHANNEL4, STD_LOW);
    }
}

```

```

    }
}
void LedRunnable() {
    static uint8 cnt;
    static uint32 cnt2 ;

    if((cnt % 2) == 0) {
        Dio_WriteChannel(DIO_CHANNEL_NAME_LED_CHANNEL1, STD_HIGH);
    } else {
        Dio_WriteChannel(DIO_CHANNEL_NAME_LED_CHANNEL1, STD_LOW);
    }

    if(++cnt2 % led_mod == 0) {
        cnt++;
    }
}
void Notify_Invalid_CMAC_Limit_Reached( void ) {
    // MaCAN notified us that we should stop receiving data
    stop_receiving = 1;
    // blink green LED faster
    led_mod = 5;
    // switch off blue LED
    Dio_WriteChannel(DIO_CHANNEL_NAME_LED_CHANNEL4, STD_LOW);
}

```

### A.7.3 TheProducer.c

1. Create file **TheProducer.c** in **example-ecu** project root
2. Put following contents in the file:

```

#include "Rte_MyProducer.h"
#include "Dio.h"
#include "stm32f10x_gpio.h"

void MyProducerRunnable( void ) {

    uint8 value;

    value = GPIO_ReadInputDataBit(GPIOB,GPIO_Pin_9);

    //value = Dio_ReadChannel(DIO_CHANNEL_NAME_BUTTON_CHANNEL);
    Rte_IWrite_MyProducerRunnable_MySenderPort_MyElement(value);
}

```

## A.8 Compilation

1. Switch to C/C++ perspective

2. On the right side, select **Make target** tab
3. Double click on **all** under **example-ecu**
4. Once the compilation process finishes, the executable file **example-ecu.elf** is created in **obj\_stm32\_stm3210c** directory in the project root

## A.9 Downloading binary file to target board

Downloading of the binary file to the target board is done using OpenOCD. You can perform the following steps in any directory, you don't need to be in project root. Just make sure you have the executable file mentioned in the previous section in current directory.

1. Create file **load-jt\_usb5.cfg** and put the following contents to this file:

```
telnet_port 4444
gdb_port 3333
#set CHIPNAME STM32F107VCT6
#set WORKAREASIZE 0x10000
interface ftdi
#ftdi_device_desc "Dual_RS232"
ftdi_vid_pid 0x0403 0x6010

ftdi_layout_init 0x05f8 0x0cfb
ftdi_layout_signal nTRST -data 0x0010 -noe 0x0800
ftdi_layout_signal nSRST -ndata 0x0040 -noe 0x0400

# reset layout for ul_usb1 and jt_usb5 with ft232
#          7   TRST   NTRST   IOL0
#          6   nHRST  RST     IOL2
#          4   RTCK   SRST    IOL1

#source [find interface/ftdi/xds100v2.cfg]
source [find board/stm3210c_eval.cfg]
reset_config trst_and_srst

init
jtag arp_init--reset
#ftdi_set_signal PWR_RST 1; jtag arp_init
```

2. Create file **flash.sh**

```
openocd -c "source_[find_load-jt_usb5.cfg]" -c "stm32f1x.cpu_arp_halt" -c "wait_halt"
\
-c "program_\example-ecu.elf\_reset"
```

3. Make this script executable and invoke it:

```
./flash.sh
```