Bachelor Thesis

Low-level software for automotive electronic control units

Leoš Mikulka

May 2013

Supervisor: Ing. Michal Sojka, Ph.D.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Cybernetics

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Leoš Mikulka
Studijní program:	Kybernetika a robotika (bakalářský)
Obor:	Robotika
Název tématu:	Nízkoúrovňový software pro řídicí jednotky do aut

Pokyny pro vypracování:

1. Seznamte se se standardem AUTOSAR a řídicí jednotkou do aut vyvinutou na katedře.
2. Prostudujte otevřený software Arctic Core implementující AUTOSAR specifikaci.
3. Zprovozněte Arctic Core na dané řídicí jednotce, pokuste se zprovoznit překlad softwaru pod OS Linux.
4. Připravte sadu ukázkových aplikací jako např. blikání LED diodou a řízení motoru, komunikaci přes sběrnici CAN.
5. Vše důkladně zdokumentujte.

Seznam odborné literatury:

[1] ARCCORE - Arctic Core and Arctic Studio wiki - available online at http://arccore.com/wiki/.
[2] AUTOSAR - AUTOSAR R4.0 specification - available online at http://www.autosar.org.
[3] RPP ECU dokumentace - dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Michal Sojka, Ph.D.

Platnost zadání: do konce zimního semestru 2013/2014

[illegible]

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta elektrotechnická -4-

L.S.

[illegible]

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2013

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student:	Leoš Mikulka
Study programme:	Cybernetics and Robotics
Specialisation:	Robotics
Title of Bachelor Project:	Low-Level Software for Automotive Electronic Control Units

Guidelines:

1. Make yourself familiar with AUTOSAR and an automotive electronic control unit (ECU) developed at our department.
2. Study the open source AUTOSAR implementation called Arctic Core.
3. Make Arctic Core running on the ECU and try to compile the software under Linux OS.
4. Prepare a set of demo applications such as LED blinking, motor control and CAN-bus communication.
5. Document everything thoroughly.

Bibliography/Sources:
[1] ARCCORE - Arctic Core and Arctic Studio wiki - available online at http://arccore.com/wiki/.
[2] AUTOSAR - AUTOSAR R4.0 specification - available online at http://www.autosar.org.
[3] RPP ECU dokumentace - dodá vedoucí práce.

Bachelor Project Supervisor: Ing. Michal Sojka, Ph.D.

Valid until: the end of the winter semester of academic year 2013/2014

[illegible]
prof. Ing. Vladimír Mařík, DrSc.
Head of Department

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.

[illegible]
prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2013

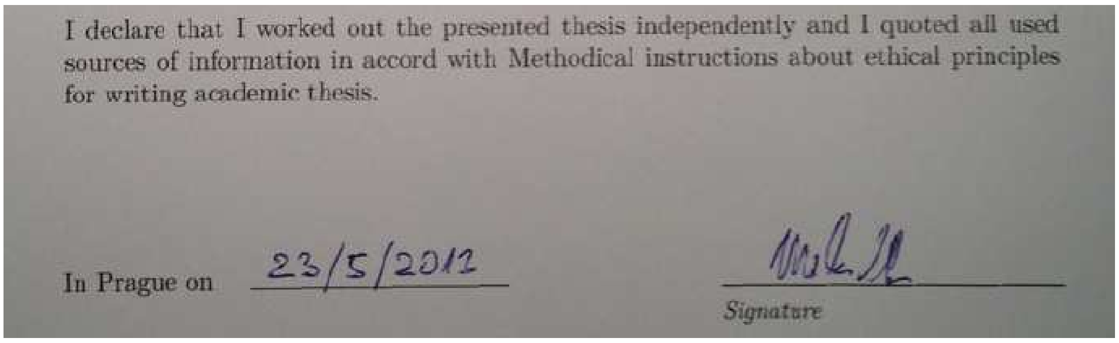
Acknowledgement

Let me thank my supervisor Ing. Michal Sojka, Ph.D. for his professional guidance and much useful suggestions during the course of this work.

Furthermore, I would like to thank to other people from Real-Time Systems group at Department of Control Engineering for their helpful hints.

Besides, I would like to send my last thanks to all other who contributed with even a little piece of advice and supported me through all the time.

Declaration



I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague on 23/5/2012

[illegible]

Signature

Abstract

Tato bakalářská práce se zabývá vývojem nízkoúrovňového softwaru podle standardu AUTOSAR pod platformou Arctic Core. V dokumentu je v krátkosti popsána softwarová architektura a metodologie standardu AUTOSAR. Kromě platformy Artic Core jsou také popsány komerční pluginy sloužící ke generování zdrojového kódu of firmy ArcCore. Dále jsou vysvětleny základní moduly nízkoúrovňového softwaru, které byly použity při vývoji ukázkových úloh, stejně jako sekvence volaných funkcí těchto modulů sloužící ke správné funkčnosti. Vyvinuty byly dvě ukázkové aplikace: blikání LED diodou a komunikace přes sběrnici CAN.

Klíčová slova

AUTOSAR; řídící jednotky; nízkoúrovňový software; open-source

Abstract

This bachelor thesis deals with the development of low-level software according to the AUTOSAR standard using the Arctic Core platform. The software architecture and methodology of the AUTOSAR is described in this document. Besides the Arctic Core platform, commercial plugins used for generating the source code are shortly described. Furthermore, the basic software modules which were used during the development of demo applications are explained, as well as the sequence of calling functions of these modules used for proper functionality. Two applications have been developed: LED blinker and CAN communication.

Keywords

AUTOSAR; electronic control units; low level software; open-source

Contents

List of Figures

List of Tables

Abbreviations

ADC	Analog/Digital Converter
AJSM	Advanced JTAG Security Module
BSW	Basic Software
CAN	Controller Area Network
COM	Communication
DAP	Debug Access Point
DIO	Digital Input/Output
DMM	Data Modification Module
ECU	Electronic Control Unit
EcuM	ECU Manager
GIO	General Input/Output
GPT	General Purpose Timer
HDK	Hardware Development Kit
HOH	Hardware Object Handle
HRH	Hardware Receive Handle
HTH	Hardware Transmit Handle
ICC	Implementation Conformance Class
IDE	Integrated Development Environment
I-PDU	Interaction Layer Protocol Data Unit
IoHwAb	Input/Output Hardware Abstraction
	LED] Light-Emitting Diode
LIN	Local Interconnected Network
L-PDU	Data Link Layer Protocol Data Unit
MCAL	Microcontroller Abstraction Layer
MCU	Microcontroller Unit
N2HET	High-End Timer Module
N-PDU	Network Layer Protocol Data Unit
NVRAM	Non-volatile Random Access Memory
OEM	Original Equipment Manufacturer
OS	Operating System
PDU	Protocol Data Unit
PDUR	PDU Router
PLL	Phase Lock Loop
PWM	Pulse-Width Modulation
RTE	Runtime Environment
RX	Receive
SDU	Service Data Unit
SPI	Serial Peripheral Interface
SWC	Software Component
TX	Transmit
VFB	Virtual Function Bus
XML	Extensible Markup Language

1 Introduction

The increasing Electrics/Electronics (E/E) complexity within the automotive domain and an increasing quantity of electronic control units[1] leads to the need of a standard that would help to accomplish this rapidly increasing complexity. The established open standards are key to manage the growing E/E complexity and improve development efficiency.
AUTOSAR – AUTomotive Open System ARchitecture is a development partnership of automobile manufacturers (OEMs), suppliers, and tool vendors and developers [5] that have been working on the development of an open, standardized software architecture for automotive control units (ECUs). The AUTOSAR architecture uses a layered architecture which includes a complete basic and environmental software stack for electronic control units. This stack, so called the AUTOSAR Basic Software is being developed as an integration platform for hardware independent software applications [2].
The main focus of this thesis lies in getting familiarized with the AUTOSAR and the modules of its basic software stack. For this purpose, demo applications were developed under the open source implementation of AUTOSAR, embedded platform called Arctic Core by Swedish company ARCCORE AB. These demos include two applications: a LED blinker and CAN-bus communication. The next goal was to make these applications running on a hardware development kit (HDK) by Texas Instruments, called TMS570LS3137 HDK to be precise, and a RPP control unit, both available at the Department of Control Engineering at CTU. The boards are shown in Figure 1 and 2. More about the RPP can be found in [36].
The thesis is structured as follows. Chapter 2 provides an introduction to AUTOSAR. The used basic software (BSW) modules are described in Chapter 3. The open-source implementation Arctic Core is described in Chapter 4. The configuration and implementation of demo applications is described in Chapter 5. Conclusion is given in Chapter 6.

[1]A term for an embedded system that controls some of the electrical systems in a motor vehicle

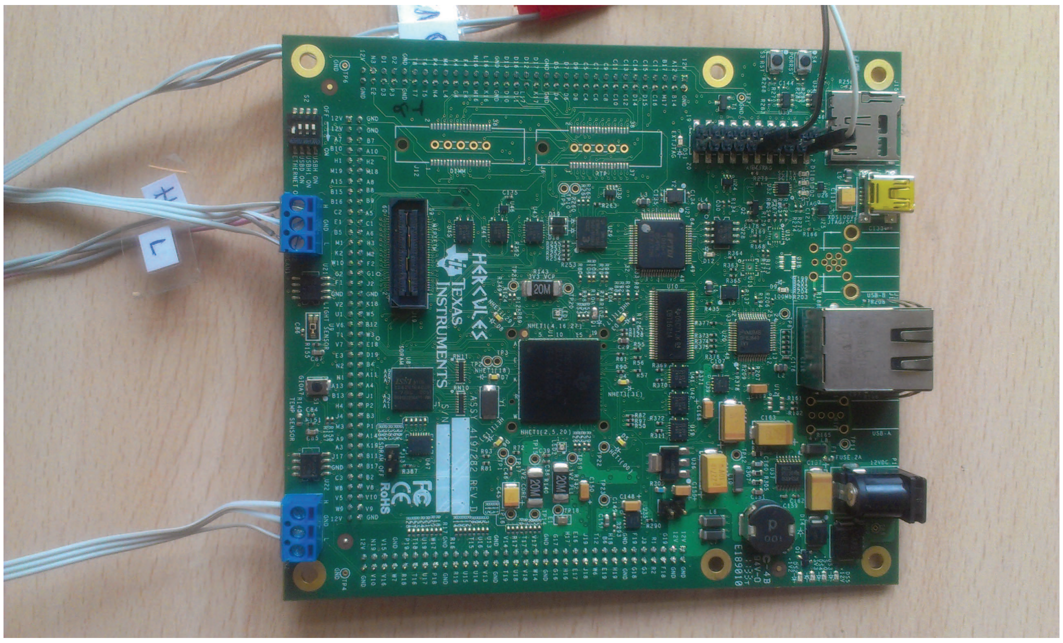


Figure 1 TMS570LS3137 HDK [1]

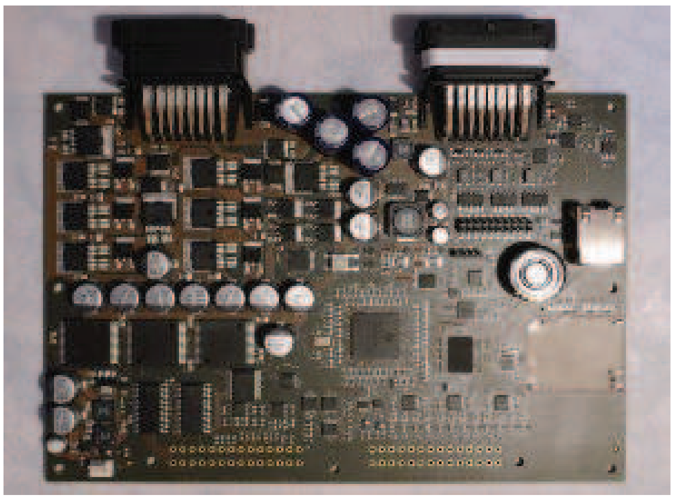


Figure 2 RPP Board [1]

2 AUTOSAR

This chapter describes the AUTOSAR as a whole. A reader should get familiarized with the AUTOSAR partnership and created software architecture and methodology. The foundation and goals and plans of the AUTOSAR partnership are described in Sections 2.0.2 and 2.0.3. After that, the software architecture is explained in Section 2.1. The methodology created is described in Section 2.2.

2.0.1 Why AUTOSAR?

As mentioned in Chapter 1, the established open standards are key to manage the growing E/E complexity and improve development efficiency. Moreover, Tier 1 suppliers[1] are often faced with requirements from legal enforcement, passengers and drivers requirements. These requirements include e.g. environmental aspects, safety requirements, entertainment domains, driver assistance, and much more. Due to all the reasons mentioned above, the AUTOSAR partnership was founded [3].

2.0.2 Foundation

First discussions about common objectives were held in August 2002 by BMW Group, DaimlerChrysler AG (now Daimler AG), Volkswagen and automotive supplier Bosch, Continental AG. They were soon joined by Siemens VDO (now part of Continental AG). Few months later, a technical team was set up to establish the technical implementation strategy. The Core Partners formally signed the partnership in July 2003. The Core Partners were afterwards joined by Ford Motor Company, Peugeot Citroën Automobiles S.A., Toyota Motor Company and General Motors. To the date of October 2012 there are a total of 146 corporate members [3].

2.0.3 Goals and Plans

AUTOSAR leading OEMs and Tier 1 suppliers look upon this foundation as an industry-wide challenge [3]. First of all, it must be ensured that current and future vehicle requirements, e.g. safety, availability, maintainability, or even software updates and upgrades will be fulfilled

[1]Companies that are direct suppliers to Original Equipment Manufacturers (OEMs)

[4]. Among others, an open standardized architecture should lead to improved quality and reliability, as well as improved development and thus optimized cost.
The one of the most important goals of the AUTOSAR are [14]:

- Standardization of basic software functions of automotive ECUs
- Scalability to different vehicle and platform variants
- Transferability of functional software modules within a particular system
- Integration of software modules by different suppliers
- Development of highly dependable systems
- Sustainable utilization of natural resources
- Maintainability throughout the Product Life Cycle and software update and upgrades throughout a vehicle lifetime

The AUTOSAR follows the principle "Cooperate on standards, compete on implementation". The main requirement is that implementation of basic software and tooling must be enabled and supported worldwide. Even though this global partnership creates one common standard, free competition is expected on implementation level [5].

2.0.3.1 Future Plans

Since 2003, there have been several releases, with the newest Release 4.1 in March 2013. According to data statistics for 2011, there was about 25 million ECUs produced based on the AUTOSAR architecture. This number is expected to grow to 300 million ECUs for the year 2016 [5]. The majority of the Core Partners will have finished their migration to fully operational AUTOSAR Basic Software in 2015.

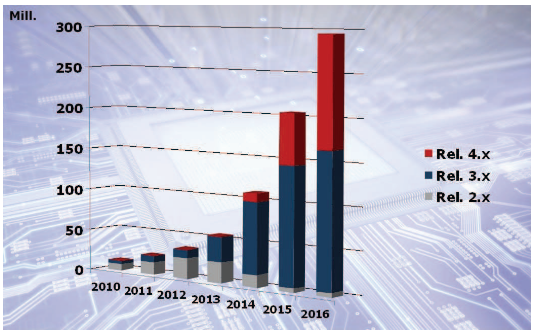


Figure 3 Volume of ECUs with AUTOSAR architecture [5]

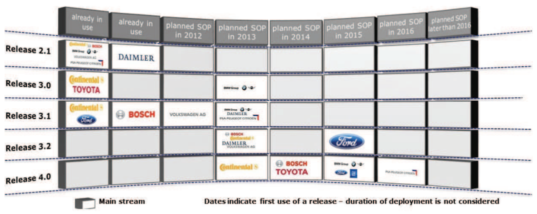


Figure 4 Usage of AUTOSAR BSW layered architecture [5]

2.1 Software Architecture

The AUTOSAR uses a layered architecture in order to separate the functionality from supporting hardware and to enable the development of independent software components. Figure 5 and 6 show a decoupling into different layers on an ECU.

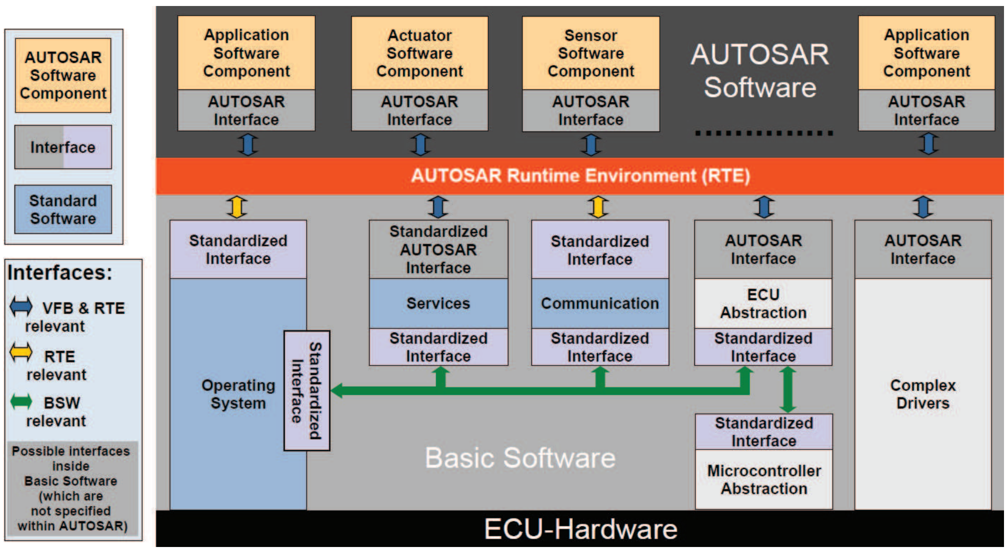

AUTOSAR Software Component
Interface
Standard Software
Interfaces:
VFB & RTE relevant
RTE relevant
BSW relevant
Possible interfaces inside Basic Software (which are not specified within AUTOSAR)
Application Software Component
Actuator Software Component
Sensor Software Component
AUTOSAR Software
Application Software Component
AUTOSAR Interface
AUTOSAR Runtime Environment (RTE)
Standardized Interface
Standardized AUTOSAR Interface
Services
Communication
ECU Abstraction
Complex Drivers
Operating System
Standardized Interface
Microcontroller Abstraction
Basic Software
ECU-Hardware

Figure 5 Simplified view of components and interfaces [8]

Set of requirements and specifications that describe a software architecture, application interfaces and a methodology can be found in documents available in the official website[2].

2.1.1 Virtual Function Bus

Applications should be implemented in form of AUTOSAR Software Components. These components have a well-defined interface that is described within the AUTOSAR [4]. More importantly, they are implemented independently from underlying hardware. The independence is achieved by providing the Virtual Function Bus (VFB).
All components are connected through the VFB. The VFB not only connects different components but also handles all communications mechanisms between them. That means that all the software components do not need to know about other components. They simply exchange information via the VFB. The Virtual Function Bus is implemented by the AUTOSAR Runtime Environment and underlying Basic Software [7].

[2]http://www.autosar.org

This ease development process of automotive software and enables better integration of AUTOSAR Software Components.

2.1.2 Basic Software

It can be deduced from the Figure 5 that the AUTOSAR architecture distinguishes on the highest abstraction level between three software layers. Those layers are Application Layer, Runtime Environment and Basic Software Layer that runs on a microcontroller.
The Basic Software contains specific components for an ECU and services, such as communication, I/O management, memory management, or an operating system. This layer is necessary to run the software. Over past years there have been three conformance classes which were supposed to ease the migration process to AUTOSAR [6]. Newest of them is Implementation Conformance Class 3 (ICC3) in which all Basic Software (BSW) modules are defined. The requirements on the basic software cover the following domains: body, powertrain, chassis and safety. The BSW layer is divided into the Services Layer, ECU Abstraction Layer, Microntroller Abstraction Layer (MCAL), and Complex Drivers. Figure 6 shows this decoupling. MCAL is indicated by red color, above that ECU Abstraction Layer by green, Services Layer by blue and Complex Drivers by green on the right side.

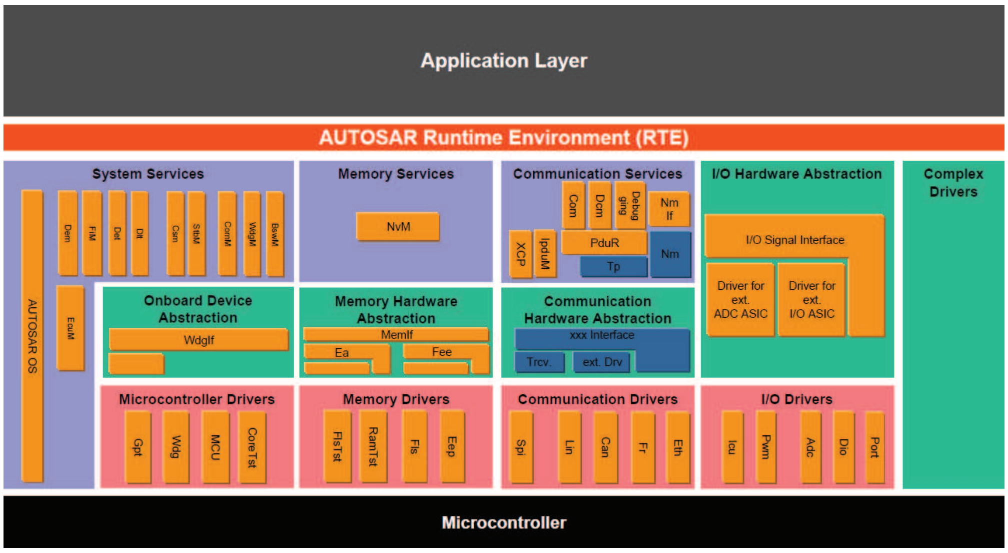


Figure 6 Overview of modules according to Implementation Conformance Class 3 [8]

2.1.3 Run-Time Environment

The Runtime Environment is the realization of interfaces of the AUTOSAR Virtual Function Bus for a particular ECU. It is a layer mediating and securing inter- (e.g. CAN, LIN, FlexRay, etc.) and intra-ECU information exchange [4]. The RTE provides communication services to the application software, that are the AUTOSAR Software Components or the AUTOSAR Sensor or Actuator components by providing interface and services [13]. The software components communicate with other components and services via the RTE. Regarding the basic software, the RTE may act as the mean by which software components access basic software modules. The RTE code is generated automatically.

2.1.4 Application Software

First of all, it is appropriate to mention that AUTOSAR enables developers designing AUTOSAR applications almost independently from hardware. Even though there is no knowledge about the network (the software architecture and the RTE hides the network from application) and about the used ECU (the software architecture abstracts from a used ECU and its microcontroller), there is knowledge about the sensors and actuators of an ECU which are used in a specific AUTOSAR application [8].
The AUTOSAR Software consists of AUTOSAR Software Components (SWC) that are mapped on the ECU. The AUTOSAR Software Components provide the core functionality, i.e. the control logic necessary for some tasks (e.g. ignition timing, exhaust gas recirculation strategy, etc.) [35]. They encapsulate an application which runs on the AUTOSAR infrastructure Interaction between components is achieved through the RTE. Those components are the AUTOSAR sensor/actuator components (ECU dependent) and the AUTOSAR application software components (ECU independent) [4].

2.1.4.1 Sensor/Actuator Software Components

A sensor and actuator software component is an atomic piece of software. The atomic software component cannot be divided into smaller parts and cannot be distributed over several ECUs [8]. It is a whole unit. It contains single threads of execution (Runnable Entities) and are later mapped on an ECU. A responsibility of a sensor component is to read and provide data to other components through input/output stack. A responsibility of an actuator component is to set the state of an actuator of a particular ECU. So it may provide an interface to other components for initiating state setting of an actuator.

2.1.4.2 Application (composite) software components

A software component is a composite component. That means, it is a logical interconnection of other component, either composite or atomic. The composition allows the encapsulation of several pieces of the functionality. For example, a composition that calculates the pulse width for petrol injectors may contain two atomic software components – one which calculates the base injector pulse width from the desired fuel mass flow rate [35]. These components can be distributed over more than one ECU [8].

2.2 Methodology

The AUTOSAR created a methodology that describes steps for building E/E system. This includes four steps. First step is System Configuration Description that includes all system information as well as interconnections between ECUs. It uses input information from the System Constraint Description. The System Constraint Description describes mapping of software components to one or more ECUs. The activity involves the creation of a topology, the definition of the resource available on an ECU, and the interconnection between ECU instances. Second step is System Configuration Extractor. This step is used for extracting the information from the System Configuration Description needed for one specific ECU. It provides an extract of the system description for a specific sub-system[3]. Since the content of the input description is reduced to one sub-system, it allows a start of work on a ECU even if the system is not completely described. Third step is ECU Extract that basically includes information (configuration of the BSW and RTE) for one specific ECU. The purpose of the ECU Extract is to extract information out of the system description in order to be delivered as a basis for setting up the ECU configuration and further development on ECU level. Last step is ECU Configuration Description. In this step all information for a specific ECU are known and an executable piece of software and the code can be generated [2].
In order to support this methodology the AUTOSAR meta-model was developed. The meta-model is modeled in UML. UML diagrams are used to describe the attributes of AUTOSAR systems and their interrelationships. The AUTOSAR model is an instance of the AUTOSAR meta-model. The information contained in the AUTOSAR model can be anything that is representable according to the AUTOSAR meta-model. The AUTOSAR XML schema is a schema that defines the language for exchanging AUTOSAR models. The XML schema is derived from

[3]A part of whole system

the AUTOSAR meta-model[4] by means of the rules defined in [12]. Importantly, the AUTOSAR XML schema defines the AUTOSAR data exchange format (i.e. the resulting AUTOSAR XML schema is used as the official AUTOSAR data exchange format).
The AUTOSAR methodology is illustrated in Figure 7. All of those system and ECU extracts can be generated automatically by using some of available AUTOSAR tool chains by any AUTOSAR tool providers.

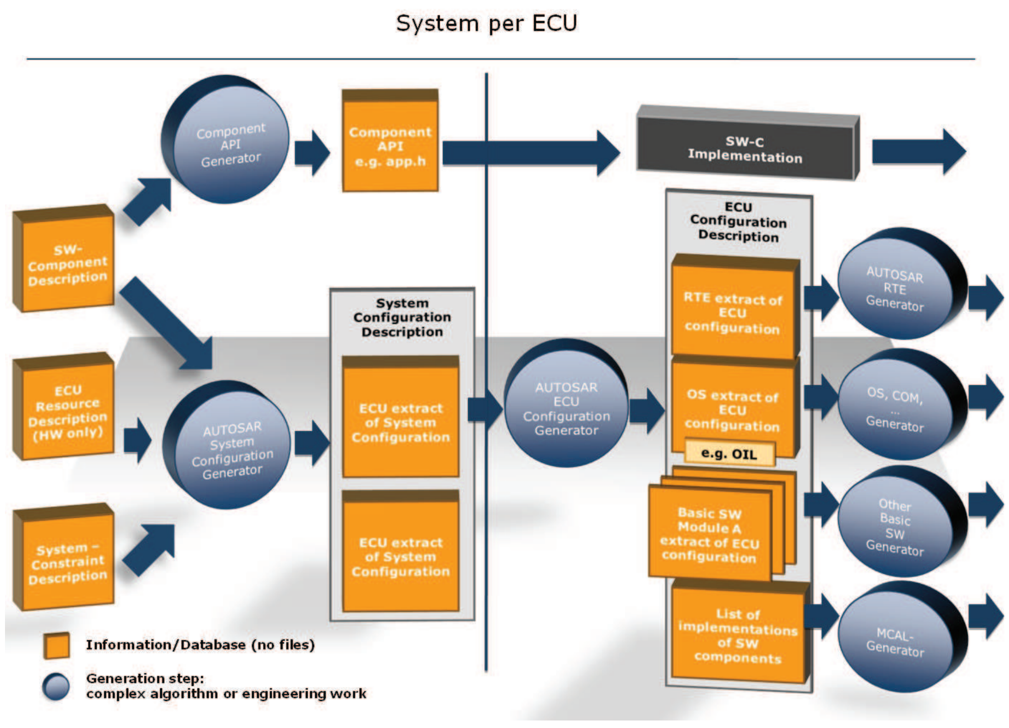

System per ECU
Component API Generator
Component API e.g. app.h
SW-C Implementation
SW-Component Description
ECU Resource Description (HW only)
System - Constraint Description
AUTOSAR System Configuration Generator
System Configuration Description
ECU extract of System Configuration
ECU extract of System Configuration
AUTOSAR ECU Configuration Generator
ECU Configuration Description
RTE extract of ECU configuration
OS extract of ECU configuration
e.g. OIL
Basic SW Module A extract of ECU configuration
List of implementations of SW components
AUTOSAR RTE Generator
OS, COM, ... Generator
Other Basic SW Generator
MCAL-Generator
Information/Database (no files)
Generation step: complex algorithm or engineering work

Figure 7 Depiction of the AUTOSAR methodology [4]

[4]Can be generated automatically out of the meta-model by using appropriate tools.

3 Used AUTOSAR BSW Modules

This chapter describes used AUTOSAR Basic Software Modules during the development of demo applications. The modules located in System Services (Section 3.1), Microcontroller Drivers (Section 3.2) and Port Driver (Section 3.3) are used in both applications. To understand difference between the Port Driver and DIO Driver, their relationship is described in Section 3.6. The layered architecture flow was explained in Section 3.7. This should be helpful in understanding used modules for CAN communication: COM module (Section 3.8), PDU Router (Section 3.9), CAN Interface (Section 3.10), CAN Driver (Section 3.11).

3.1 System Services

The System Services (Figure 6) are group of modules and functions located in the services layer. Modules of all layers can use modules and functions located in this group. Services from this group can be subdivided into three smaller groups. They differ in terms of hardware and microcontroller dependency. Some of them are microcontroller depend, e.g. the Real Time Operating System, some of them partly ECU hardware and application dependent, e.g. the ECU State Manager, and the rest are both hardware and microcontroller independent.
The purpose of these services and functions is pretty obvious. They provide basic services for application and basic software modules. The only module described in this chapter is ECU State Manager because other modules from the system services are not important for the development of our demo applications.

3.1.1 ECU State Manager

The ECU State Manager module (EcuM) is one of the crucial AUTOSAR basic software modules. It is in charge of managing initialization and de-initialization of all basic software modules (that means including the OS and the RTE). It manages conditions of an ECU states OFF, RUN, and SLEEP. These conditions are e.g. check that an ECU enters the OFF state when it is powered down or ensuring that the OS is not shut down when entering the SLEEP state. Also it handles transitions states. The transitions are called STARTUP, SHUTDOWN, and WAKEUP [16].

Figure 8 shows the cycle of main state machine provided by the EcuM. The RUN state is entered after all basic software modules have already

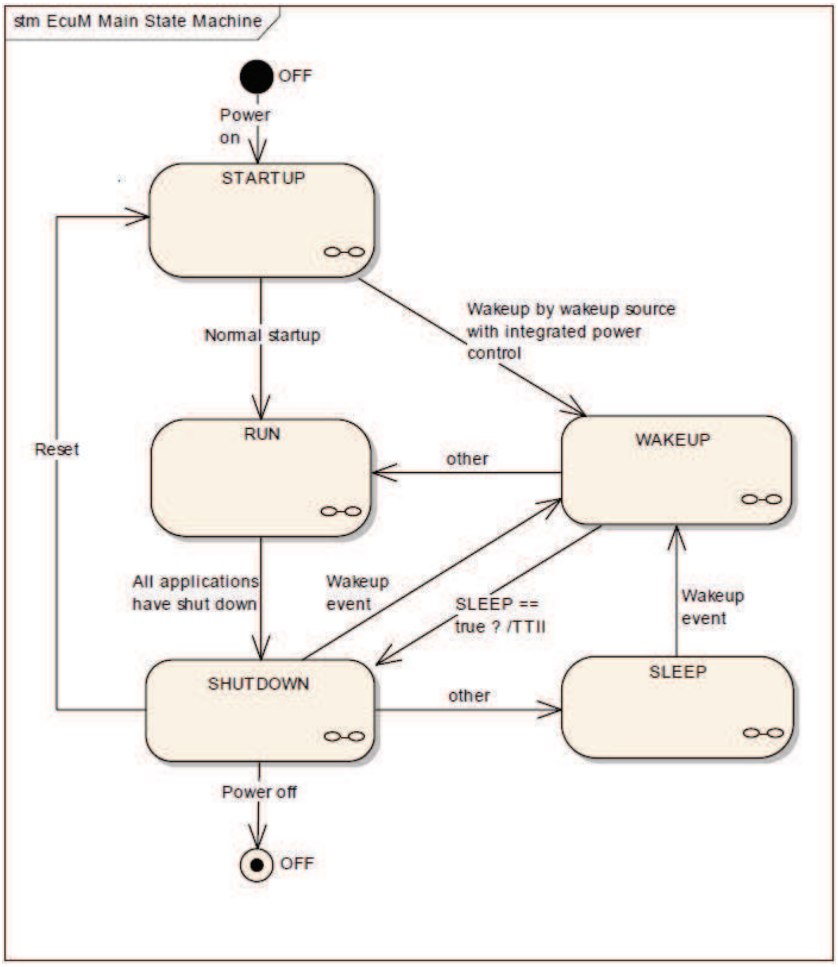


Figure 8 Main States of ECU [16]

been initialized by the EcuM. Similarly, the SHUTDOWN transition results in the SLEEP state, OFF state, or Reset transition after all basic software modules were shut down (i.e. application have shut down and no code is executed). If the Reset transition is entered, an ECU will get to the STARUP state as shown in Figure 8.

3.1.1.1 STARTUP State

STARTUP state is subdivided into several sub-states (also called phases according to [16]). This subdivision is created in order to make sure that the most important BSW modules needed for proper functionality are initialized first. The initialization is carried out through callouts. The

callouts are function stubs that the system designer can replace with code to add functionality to a module which could not be specified by the AUTOSAR. The primary goal of this state is to initialize all basic software modules. A start-up procedure of an ECU is raised by the function `EcuM_Init`. `EcuM_Init` is subdivided into two phases. After that, the RTE is started. It is assumed that a microcontroller initialization has already taken place before calling of `EcuM_Init` [16].
At the beginning of the first phase (STARTUP I) callouts[1] ensure initialization of any pre-OS driver, low level initialization code. Among initialized modules may be a development error tracer, MCU Driver, PORT and DIO, or Watchdog Driver and Manager. When this sequence of callouts is finished, the AUTOSAR OS is started [16].
During the second phase (STARTUP II) callouts[2] ensure initialization of those BSW modules that need the OS support. It may be subdivided into modules that do not need access to NvRam and modules that rely on NvRam data. Modules that may be initialized are e.g. SPI Driver, NVRAM Manager, Flash Driver, CAN Driver, CAN Interface, LIN Driver, FlexRay Driver, I-PDU Multiplexer, COM and several more [16].
Complete list of modules initialized during these phases can be found in [16].

3.1.1.2 RUN State

After all modules (including the OS and RTE) have been initialized, this state is entered. This state serves as an indication to software components above the RTE that an application is running. It is necessary that a software component request RUN state (otherwise, the ECU will launch shutdown) [16].
All actions in the RUN state are carried out by function `EcuM_MainFunction`. First phase is invoked by call of `EcuM_OnEnterRun` and this is the part in which a software component executes its tasks [16]. Similarly, callout of `EcuM_OnExitRun` is invoked when leaving this RUN state. The Figure 8 does not show the POST RUN state. This state can be requested before completely leaving the RUN state by software components to indicate the need to execute clean-up, saving data, or switching off peripherals before calling shutdown process of an ECU [16].

3.1.1.3 SHUTDOWN State

This state is responsible for the shutdown of all basic software modules and should ends up in either SLEEP, OFF, or Reset state. An important

[1]`EcuM_AL_DriverInitZero` and `EcuM_AL_DriverInitOne`
[2]`EcuM_AL_DriverInitTwo` and `EcuM_AL_DriverInitThree`

activity in the SHUTDOWN state is to write non-volatile data back to Non-volatile Random Access Memory (NvRam) [16]. When entering this state, all applications have been de-initialized [16].
SHUTDOWN can be subdivided into several transition states. These states are PREP SHUTDOWN, GO SLEEP, and GO OFF. First of all, during the transition called PREP SHUTDOWN are all handlers and managers of BSW modules shut down. This transition is a common for all shutdown targets (i.e. SLEEP, OFF, reset). Subsequently, either GO SLEEP transition state, or GO OFF transition state may be entered [16].
During the GO SLEEP transition state, persistent data should be saved to NvRam and the EcuM should set up the wake up sources for next sleep mode by calling `EcuM_EnableWakeupSources` [16]. During this state the OS is not shut down.
During GO OFF state the RTE is stopped, and modules such as Communication Manager, Basic Software Mode Manager, or Scheduler Manager are de-initialized. This state ends with call of the `ShutdownOS` service. This service ends up in the shutdown hook called `EcuM_Shutdown` which switches off an ECU (OFF state)[3] or performs the reset (reset state)[4] [16].

3.1.1.4 WAKEUP State

WAKEUP state is entered when an ECU comes out of the SLEEP state. First, during the WAKEUP phase Microcontroller Unit (MCU) is restored to normal mode and. Moreover, drivers that need re-initialization (usually drivers with wake up sources) are re-initialized by invoking the `EcuM_AL_DriverRestart`.
Because of the reason that wake up events may be unintended, e.g. the Evaluation Module (EVM) spike on CAN line, wake up events must be validated. When any wake up error occurs, SLEEP state is left, an ECU is woken up, and execution resumes by calling `MCU_SetMode` of the MCU driver [16].

3.2 Microcontroller Drivers

Microntroller Drivers are part of MCAL layer as shown in Figure 6. These drivers may have direct access to a microcontroller hardware (e.g. Core test [15]) and are intended to handle internal peripherals. Among included modules are General Purpose Timer Driver module (GPT), Watchdog Driver module and MCU Driver module.

[3]`EcuM_AL_SwitchOff callout`
[4]`Mcu_PerformReset` callout

3.2.1 MCU Driver

The Microcontroller Unit Driver (MCU) is located in the Microcontroller Abstraction Layer. It has the direct access to a microcontroller hardware. As standardized according to the AUTOSAR, it provides initialization services for a microcontroller initialization, power down and reset functionality, and other microcontroller specific functions required by other MCAL modules. During initialization process, it takes care of setting up MCU clock, PLL, clock pre-scalers, as well as setting up RAM sections. Furthermore, it activates microcontroller reduced power modes and reset [17].
However, before the initialization of the MCU driver takes place, a start-up code that ensures basic initialization of a microcontroller must be executed. This code is very MCU specific and is not AUTOSAR standardized. A start-up code must be executed after power up or microcontroller reset. The code should execute basic initialization steps, i.e. set up base addresses for interrupt vector tables, initialize an interrupt stack pointer and a user stack pointer, enable cache memory [5], initialize a minimum amount of RAM, or initialize features like memory protection [17].
The MCU driver features are described in [18].
In our examples, the MCU specific settings is configured in file `Mcu_Cfg.c`. The configuration set contains the clock setting, i.e. PLL setting.

3.3 Port Driver

We start with the description of terms Port Pin and Port according to AUTOSAR to understand the difference. The term Port Pin represents a single configurable input or output pin on an MCU device. The term Port represent the whole configurable port on an MCU device [20].
The Port Driver is responsible for overall configuration and initialization of a whole port structure of a microcontroller. A configuration depends on a specific microcontroller and an ECU [19].
Ports and port pins can be assigned one of many various functionalities, e.g. general purpose Input/Output, Analog/Digital Converter (ADC), CAN, Pulse-Width Modulation (PWM), Serial Peripheral Interface (SPI), and many more. Port pins must be configured in the Port driver as DIO, ADC pin, etc. because the Port driver is only connected to other input/output drivers (DIO Driver, ADC Driver, etc.) by means of configuration [19].

[5]If supported by a MCU

3.3.1 Configuration of the MCU port/port pins

The configuration data for the Port driver are included in the external data structure `Port_ConfigType`. The type of `Port_PinType` (uint8, uint16, or uint32) is based on a specific MCU. Every port pin number has assigned a symbolic name of a port pin to make a configuration clearer.
Parameters used in the structure `Port_ConfigType` pin mode, pin direction, decision whether pin direction and pin mode are changeable changeable during runtime. Configuring a pin mode, (e.g. ADC, SPI, etc.) is not mandatory if a port pin is configured as a DIO [19]. On the other hand, pin direction (input, output) is mandatory when a port pin is used as a DIO [19]. Defining whether a pin direction and a pin mode are changeable during runtime is MCU dependent [19]. Moreover, `Port_ConfigType` may contain optional parameters.
An overall initialization is done by calling the function `Port_Init`. After power-on reset, the first things to be done are bring GIO out of reset by writing 1 to Global Control Register (GIOCR0[6]), disabling the interrupts for pins GIOA[7:0] and GIOB[7:0] by writing 1 Interrupt Enable Clear Register (GIOENACLR), and setting low level interrupts by writing 1 to Interrupt Priority Clear Register (GIOLVLCLR).
Before the description of subsequent function, the `Port_RegisterType` will be described. This type is used to access Port registers that handle I/O functionality. It is (except Functionality register) same as `GIO_RegisterType` located in file `core_cr4.h` which is used to access GIO Port registers. Pointers used to access different ports are then grouped into an array to ensure their straightforward utilization in later code.
The function `Port_RefreshPin` is called after `Port_Init`. This function, and functions `Port_SetPinDirection`, `Port_RefreshPortDirection` and `Port_SetPinMode`[7] can be used to change the registers depending on a configuration.

3.4 I/O Hardware Abstraction

The I/O Hardware Abstraction (Figure 6), abbreviated as IoHwAb is located above MCAL drivers. That means, it will call drivers' APIs to manage on chip devices. The purpose of IoHwAb is to provide access to MCAL drivers by mapping IoHwAb ports to ECU signals [21]. The following picture shows all interfaces of MCAL drivers:

[6]according to [31]
[7]Not applicable to all modules

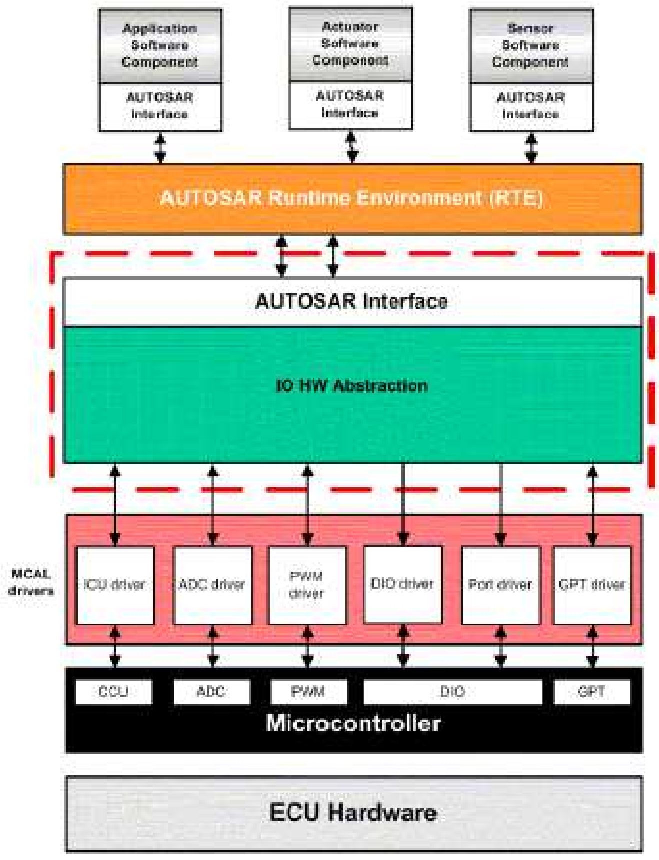


Figure 9 All interfaces with MCAL drivers [21]

If the functionality of MCAL modules' drivers is abstracted, the RTE can be independent on hardware [21]. There are currently several modules handled by IoHwAb from which three are supported by ArcCore. These modules are Pulse-Width Modulation Driver module (PWM), Analogue/Digital Converter Driver module (ADC) and Digital I/O Driver module (DIO). Due to the reason that nor PWM, neither ADC are used in our examples, we are concerned only about DIO.
The IoHwAb cannot provide Standardized AUTOSAR Interfaces to AUTOSAR software components. Instead, the IoHwAb provides interfaces (not standardized) which represent an abstraction of electrical signals coming from the ECU inputs or addressed to ECU outputs. Intention of the IoHwAb is that it also performs filtering, debouncing, or conditiong of input signals which are read through I/O drivers [21].

3.5 DIO Driver

The DIO driver module works on pins and ports configured by the PORT driver. There is no initialization or even configuration of pins and ports in the DIO driver (this is done by the PORT driver).
At the beginning, let us define the key terms. DIO channel represents a general purpose input/output pin, DIO port represents several DIO channels usually controlled by one hardware register (i.e. grouped by a hardware), DIO channel group represents several adjoining DIO channels represented by a logical group and belong to one DIO port. The purpose of the DIO driver is to provide services for reading and writing to and from DIO ports, channels, or channel groups. In other words, it provides services to setting and getting the state of microcontroller's pins [23].

3.5.1 Read & Write Services

Read and write functions of the DIO driver depend on the fact whether is dealt with the level of a single channel (i.e. a single pin), all channels of a port, or a channel group. These functions are `Dio_ReadChannel` (`Port`, `ChannelGroup`) and `Dio_WriteChannel` (`Port`, `ChannelGroup`) [22].

3.5.1.1 Dio_ReadPort

The purpose of this function is to read level (STD_HIGH, STD_LOW)[8] of all channels of a particular port. A parameter is an ID of a port and the value is obtained by reading DIN register.

3.5.1.2 Dio_WritePort

Opposite to reading level of all channels, this function sets simultaneously level of all output channels. Parameters of this function are an ID of a port and a level. The level value is written into DOUT register of a port.

3.5.1.3 Dio_ReadChannel

This function is used to get the logical value from a pin. First, `Dio_ReadPort` is called and the value of all channels is stored into `portVal` variable. Then, a value of the position of a used bit is obtained in the same manner as in the Port driver (i.e. shifting 1 to the left by logical sum of a channel ID and value 0×1F) and the values is stored into the variable called `bit`. Finally, if statement compares `portVal` and `bit`, and decides whether the value of the channel is STD_HIGH or STD_LOW.

[8]These are represented by logical 1, logical 0 respectively.

3.5.1.4 Dio_WriteChannel

This function sets the logical value of an output pin. Parameters of this function are ID of a channel and the level. As in previous function, a value of the position of a used bit is obtained first. It is necessary to check whether a channel is not defined as an input channel. In that case, function does not proceed. In other case, the current value of channels is read by calling `Dio_ReadPort`. After that if statement is used. If the level is STD_HIGH, the value of a desired bit is set. On the other hand, if the level is STD_LOW, a desired bit is cleared. The new value is stored in `portVal` and `Dio_WritePort` is called.

3.5.2 Configuration of the DIO driver

Similarly as configuration of the Port driver, the DIO driver includes types that are used to define channels and ports. `Dio_ChannelType` is the type representing a numerical ID of a channel. A numerical ID of a port is described by defining `Dio_PortType`.

3.6 Relationship between PORT Driver and DIO Driver

Since the DIO driver works on pins and ports configured by the Port driver, the Port driver should be initialized before the use of the DIO functions to avoid unexpected behaviour. The configuration of the Port driver mentioned above is used in the DIO module. Both of these modules are a part of the Microcontroller Abstraction Layer. The figure 10 shows the structure of the modules in this layer.

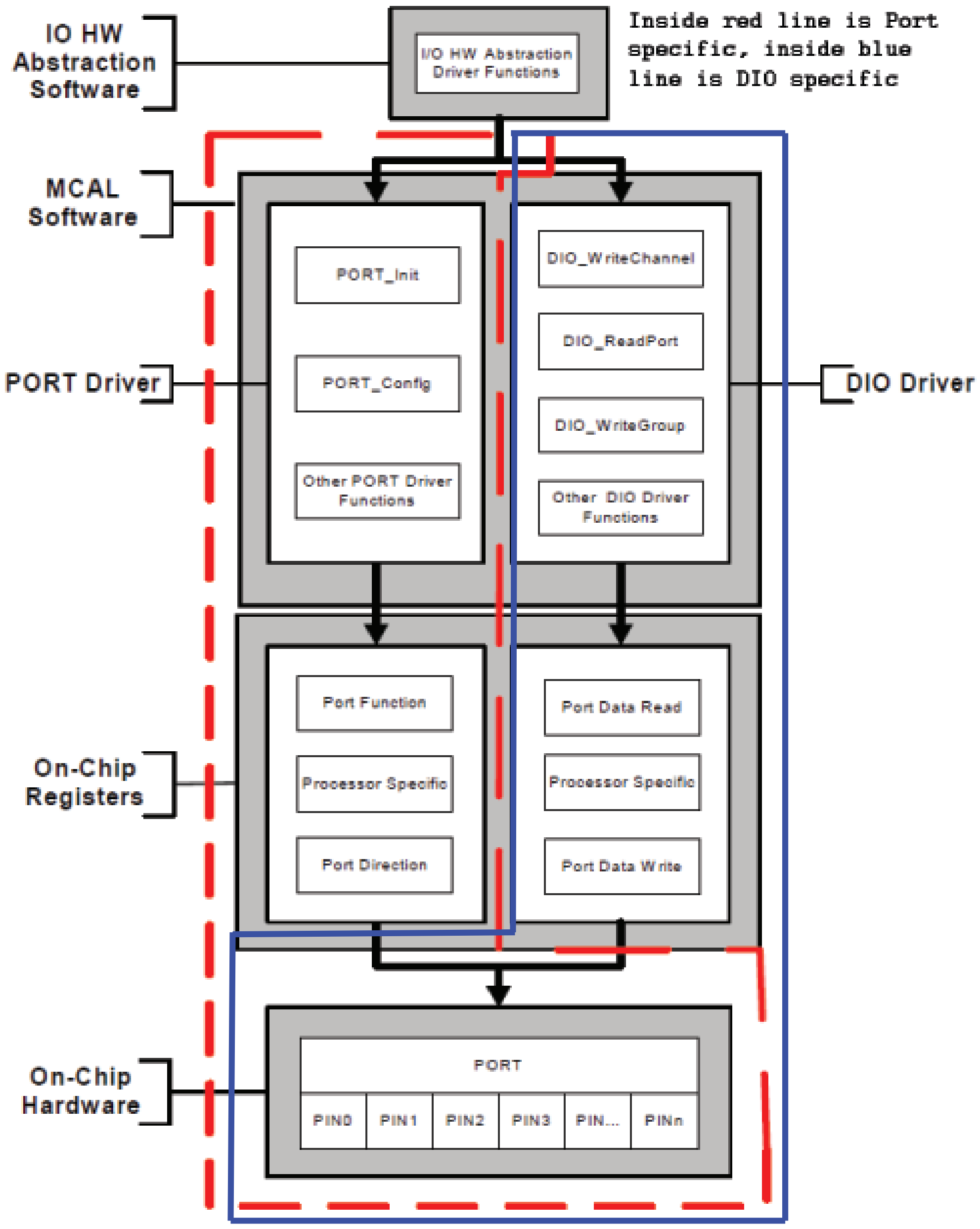


Figure 10 Port and DIO Driver structure in the MCAL software layer [19], [22]

3.7 CAN Communication – Layered Architecture Flow

Before anything else, the terms in Protocal Data Unit (PDU) flow through the layered architecture will be explained in order to avoid any confusion in later sections. Let us begin with the Figure 11,

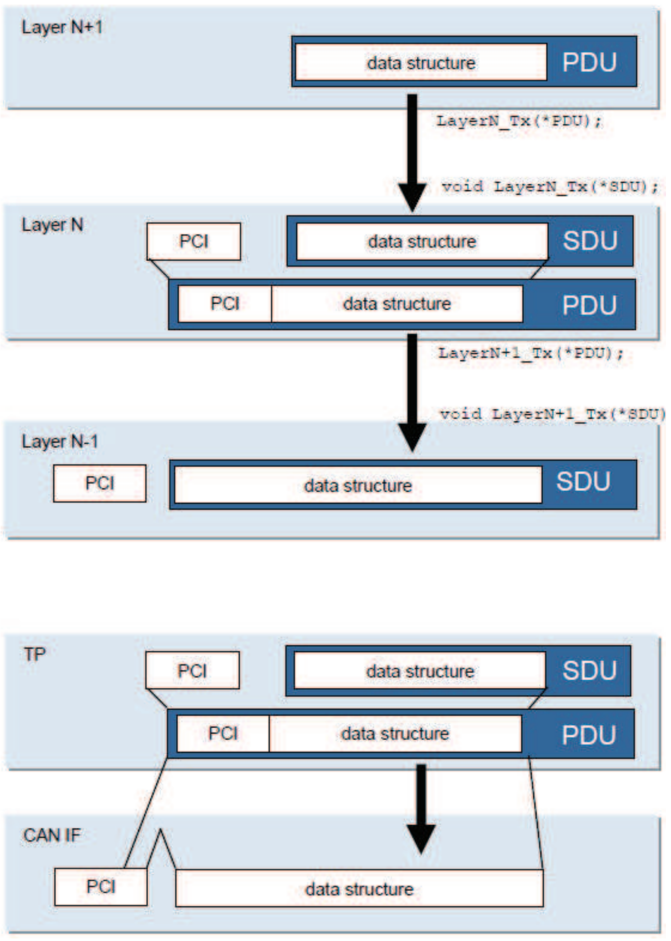


Figure 11 Flow of PDUs and SDUs through the layers [8]

As we can see, PDUs and Service Data Units (SDUs) are used in several layers. However, different prefixes are used in different layers for PDUs in order to distinguish PDUs in those various layers. This will be explained later, though.

3.7.1 Signal

A signal in the communication module of the AUTOSAR (AUTOSAR COM) context is equal to a message in the communication module according to the open automotive standard Open Systems and their Interfaces for the Electronics in Motor Vehicles (OSEK COM). According to OSEK COM, a message defines a mechanism for data exchange between different entities and with other CPUs. An AUTOSAR signal is carried by one or more signals in COM. An AUTOSAR signal is transformed to a AUTOSAR COM by the RTE. AUTOSAR COM does not necessarily have to be the same as an AUTOSAR signal because in the case of complex data types, the transformation may change the syntax a little bit [24].

3.7.2 SDU

SDU is the abbreviation of Service Data Unit. It is a part of a PDU. SDU is the data passed by an upper layer, with the request to transmit data. On the other hand, it is also data extracted by a lower layer after reception and passed to an upper layer [15].

3.7.3 PCI

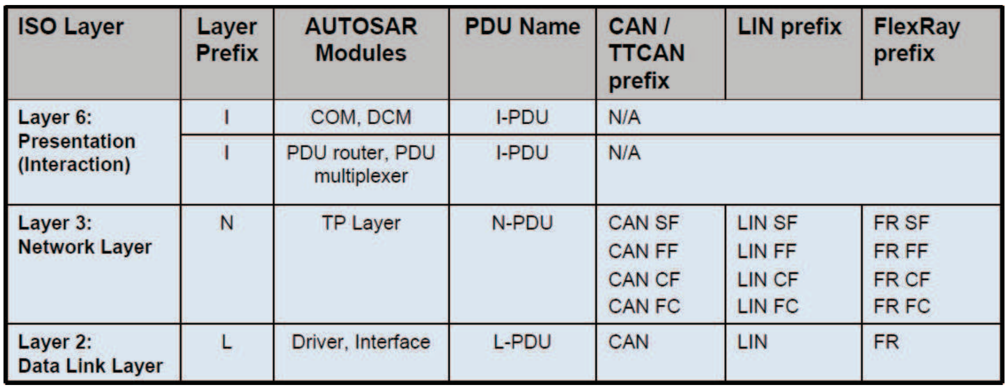
PCI is the abbreviation of Protocol Control Information. The PCI is added by a protocol layer on the transmission side and is removed on the receiving side. It is information needed to pass SDU from one instance of a specific protocol layer to another instance. For example, it contains source and target information [15].

3.7.4 PDU

PDU is the abbreviation of Protocol Data Unit. It contains SDU and PCI. On the transmission side the PDU is passed from the upper layer to the lower layer which interprets is as its SDU [15].

3.7.5 PDU & SDU Naming Conventions

As mentioned above different prefixes are used for naming PDUs in different layers. Since there is no use of LIN or FlexRay in our examples, we are not concerned about different bus prefixes. PDUs are divided into Interaction Layer Protocol Data Unit (I-PDU), Network Layer Protocol Data Unit (N-PDU), and Data Link Layer Protocol Data Unit (L-PDU) for various layers. All prefixes are described in the following figure:



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 12 Bus and layer prefixes in different layers [8]

An I-PDU is used for the data exchange of the AUTOSAR COM module. The I-PDU consists of one or more signals and is assembled and disassembled in the COM module [25]. There may be an I-PDU group which is a collection of I-PDUs. A L-PDU is assembled and disassembled in the AUTOSAR Hardware Abstraction Layer [25]. The maximum length of an I-PDU depends on the maximum length of a L-PDU because I-PDUs of the COM module are passed via the PDU router. The maximum length of CAN and LIN L-PDU is 8 bytes, of FlexRay is 254 bytes. The Communication Hardware Layer and the Microcontroller Abstraction Layer is the equivalent to the Data Link Layer [24]. The L-PDU consists of an identifier, data length code and data (i.e. SDU). It is also bus specific, e.g. CAN frame. A N-PDU is used by transport protocol modules to fragment an I-PDU [26]. Transport protocol modules are not used in our examples. However, it is appropriate to mention that it is a set of data coming from the PDU router (Section 3.9).

3.8 COM Module

The COM module is a part of communication services (Figure 6). These services are interconnected with communication drivers via communication hardware abstraction. The COM module is located between the RTE and the PDU router. The main feature of this module is the provision of signal oriented data interface for the RTE [24].
Some of other key features are following [24]:

- Packing of AUTOSAR signals to I-PDUs to be transmitted
- Unpacking of received I-PDUs and provision of received signal to the RTE
- Routing of signals from received I-PDUs into I-PDUs to become transmitted

- Communication transmission control

All of main features can be found in [24].
There are also filtering and notification mechanisms for incoming signals supported. For example, there is notification function for error notification, timeout notification function for case of any signal timeouts, or transit (TX) and receive (RX) notification. These notifications are configured in the structure `ComSignal_type` in `Com_PbCfg.c`.
Other properties when defining a signal are an ID for a signal and the numerical value which is used as an ID of a particular I-PDU (`ComIPduHandleId`). This ID is required to either by the API calls to receive an I-PDU from the PDU router, or by the PDU router to confirm the transmission of an I-PDU. Furthermore, the initial value of the signal is defined, as well as the type of the signal. From other defined properties, the most important ones are the size of a signal in bits, starting position (bit) of a signal within an I-PDU, the endianess of signal's network representation, bit position in a PDU for signal's update bit, and definition whether a signal can trigger the transmission of the corresponding I-PDU (`ComTransferProperty`).
Besides that, the configuration structure for an I-PDU is `ComIPdu_type`. Each I-PDU has a unique ID for easy PDU handling. Then the size of an I-PDU is set in bytes and the direction of an I-PDU (RECEIVE or SEND) is set as well. Signal processing mode can be configured for either unpacking received I-PDU or the transmission on an I-PDU. There are two modes configurable [24]:

- IMMEDIATE: signal indication / confirmation are performed in `Com_RxIndication` / `Com_TxConfirmation`
- DEFERRED: signal indication / confirmation are deferred for example to a cyclic task

Reference to the actual PDU data storage and reference to signals contained in an I-PDU are defined as parameters in the structure `ComIPdu_type`. Also an optional callout function can be configured for both sender and receiver side.

3.9 PDU Router

The PDU Router is a part of communication services (Figure 6). It is placed below COM module. It must be instantiated in every AUTOSAR ECU where communication is necessary. How the name indicates, its main task is to route I-PDUs between the Communication Services and the Communication Hardware Abstraction Layer modules. The PDU Router consists of two main parts, the PDU Router routing tables and the PDU Router Engine. The PDU Router routing tables are static routing tables describing the routing attributes for each I-PDU that should be routed. The PDU Router Engine is the actual code that performs routing actions based on the routing tables. No routing operation modify actual I-PDU. Detailed PDU Router structure is shown in Figure 13. Besides routing an I-PDU from source to destination, it is responsible for translating an ID of an I-PDU to destination. Moreover, the PDU Router Engine provides an additional minimum routing capability which is independent of the PDU Router. This allows access to the Diagnostic Communication Manager (used during car diagnostic) even if the routing tables are corrupted or not programmed [26].

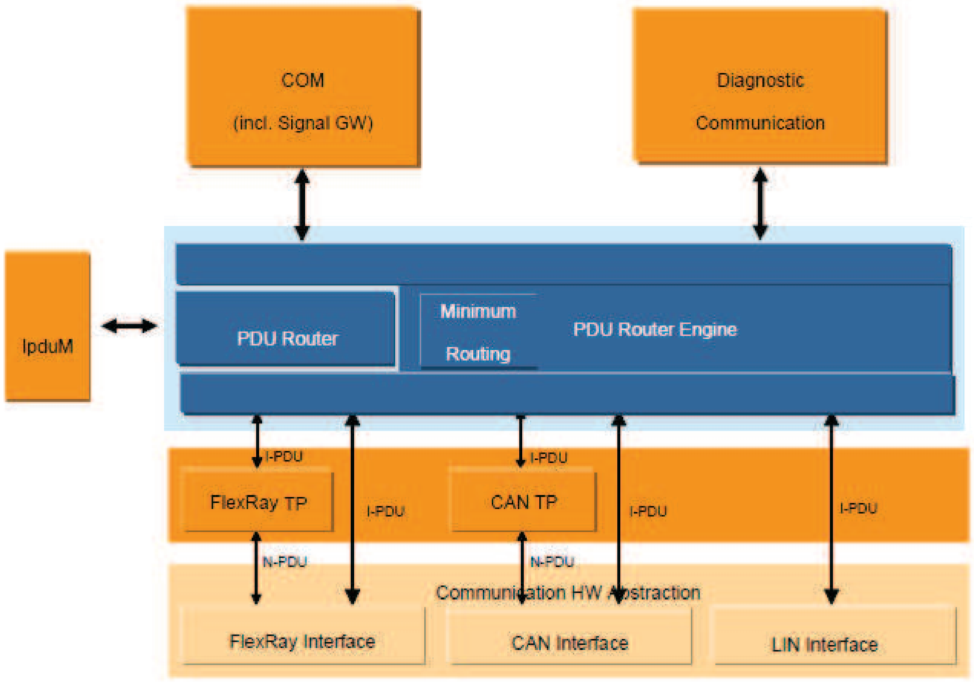


Figure 13 Detailed PDU Router structure [26]

How can be derived from the description of the module above, the PDU Router can transmit I-PDUs to lower layer modules if requested by upper layer modules. Contrarily, it can receive I-PDUs from lower layer modules and forward them to upper layers. However, it can be also used as PDU Gateway, so it can receive I-PDUs from lower layer modules and transmit them via the same or other lower layer modules right after that.
An I-PDU is identified with a unique ID represented by a symbolic name. It is required that each BSW module that handles I-PDUs must contain a list of I-PDUs IDs [26]. The destination of an I-PDU is determined by

using those IDs configured in a static configuration table. In the source code, the file `PduR_PbCfg.c` contains these configuration tables.
First, PDU Router destinations are stored in the structure `PduRDestPdu_type`. This includes a unique PDU ID which will be used by the PDU Router. Plus, the way how a data is provided to communication interface modules, i.e. COM Module (Section 3.11), Can Interface (Section 3.10), is defined. This is called a data provision mode. There are three data provision modes configurable [26]:

- DIRECT – data to be transmitted are provided directly at the transmit request (the PDU Router should not buffer an I-PDU)
- TRIGGER TRANSMIT – data to be transmitted will be retrieved by the interface module via a callback function
- NO PROVISION – there is no data provision

Definition (by providing symbolic name) of a destination module to the configuration structure by Arctic Core is also added in the structure `PduRDestPdu_type`.
Furthermore, the structure `PduRRoutingPath_Type` contains the length of an I-PDU data[9]. Moreover, it specifies the source module for this route, the source ID and the destination of an I-PDU to be routed.

3.10 CAN Interface

The CAN Interface is located in the Communication Hardware Abstraction layer below communication services layer and above communication drivers (Figure 6). The CAN Interface module provides an interface to the services of the CAN Driver. That means it manages hardware devices like CAN controller and transceivers. The main goal is to include all CAN hardware independent tasks which belong to CAN communication to this module, so that underlying layer (CAN Driver) can focus only on access and control of a specific CAN hardware device [27].
According to AUTOSAR specifications, the services can be divided into several groups [27]:

- Initialization
- Transmit request services
- Transmit confirmation services
- Reception indication services
- Controller mode control services
- PDU mode control services

[9] Only required if a TX buffer is configured

3.10.1 Initialization sequence

This section is mentioned because during the initialization of CAN Interface the function of CAN Driver `Can_InitController` (according to [28] can be changed to `Can_ChangeBaudrate`) is called. The function `Can_InitController` should re-initialize the CAN controller and the controller specific settings.
According to the AUTOSAR, the function `CanIf_Init` is called in `EcuM_Callout_Stubs.c` for the initialization of the CAN Interface. The CAN Driver is initialized separately by calling corresponding function in the same file.
First of all, it is ensured that CAN controllers are in the state `CANIF_CS_STOPPED`[10] (this value is actually assigned as the controller mode), and the value of `CANIF_GET_OFFLINE` is assigned as the PDU mode. Next, the function `CanIf_PreInit_InitController` can be called. The most important feature inside this function is that reference[11] to CAN controller configuration data is assigned to the variable `canConfig` which is used as the parameter during call of `Can_InitController` function (located still inside the same function). The function `Can_InitController` is described in Section 3.11.

3.10.2 Hardware Object Handles (HOH)

Hardware object handles are two types, hardware transmit handle (HTH) and hardware receive handle (HRH). They are reference to the CAN mailbox structure that contains parameters such as a CAN ID, data length code or data. They are used as parameters in the calls of CAN interface services and are provided by the CAN Driver configuration. The CAN Interface remains independent of hardware because it acts only as a user of HOH and does not interpret it on the basis of hardware specific information [27] (Section 3.11).

3.10.3 Transmit request sequence

Upper layer modules use the service `CanIf_Transmit` to start a transmit request of a PDU. This function first check initialization of the CAN interface. Then parameters from configuration structures (which are located in `CanIf_Cfg.c`) are assigned to various variables. Among these parameters are PDU ID, data length code, CAN ID for transmit of CAN L-PDU, or the pointer to SDU. Furthermore, based on a input parameter during call of `CanIf_Transmit`, a unique HTH ID, defined in the CAN Driver module (in other words, this parameter refers to the

[10]Located inside `CanIf_Init`
[11]This reference is located inside the structure `CanIf_ControlerConfigType` located in `CanIf_Cfg.c`

particular HTH object in the CAN Driver module configuration), is used as the input parameter for call of the function `Can_Write` together with setting of PDU as a second parameter. `Can_Write` function is located in `Can.c` and is responsible for setting registers necessary for transmission.

3.10.4 Reception indication sequence

The reception indication is used by the CAN Driver in the case when some feedback is required during the reception of a PDU.
In case of new reception on an L-PDU the CAN Driver calls `CanIf_RxIndication` of the CAN interface [27]. This callback is defined together with `CanIf_TxConfirmation` in `Can_Lcfg.c`. The input parameters of this function are HRH, a unique CAN ID, data length code of received data and pointer to received L-SDU. The HRH identifies one CAN hardware receive object, where a new CAN L-PDU arrived.
Considering `CanIf_RxIndication` function, a list of RX PDUs is obtained from the initialization parameters of the CAN Interface as the first step. Then, the configuration parameters of RX PDUs from the file `CanIf_Cfg.c` are compared with the input parameters of this function. This comparing has its meaning. First of all, the received PDU must pass software filtering (the first step is to decide whether HRH is type Basic CAN or Full CAN[12]). After that, if defined as required, data length code check must be performed.
In case temporary buffering is used, the hardware object remains locked until all data is read out and copied to this temporary buffer. If no temporary buffering is used, the hardware object remains locked until all data is read out and the indication service returns [27].

3.11 CAN Driver

The CAN Driver module is located in the communication drivers layer that is the lowest layer above hardware (Figure 6). It provides uniform interfaces for the layer above which is CAN Interface. The CAN Driver module provides services for initiating transmissions and calls the callback functions of the CAN Interface module, independently from hardware. The CAN controllers provide services to control behaviour and the state of controllers (e.g. `Can_DisableControllerInterrupts, Can_CheckBaudrate, Can_SetControllerMode and much more` that belongs to the same CAN Hardware Unit. The CAN Hardware Unit consists of one or more CAN controller of the same type. It may be on-chip or an external device. One CAN driver represents the CAN Hardware Unit. The CAN controller serves one physical channel. [28], [29]

[12]For Basic CAN software filtering is enabled.

The file `Can_Lcfg.c` contains structures with the configuration of controllers and hardware objects, and CAN callbacks.
The structure `Can_ControllerConfigType` contains among others these following parameters:

- Controller ID
- Baud rate in kbps
- Propagation delay in time quantas[13]
- Phase segment 1 and 2 in time quantas
- Reference to the CPU clock configuration (set in the MCU driver)
- Process type (INTERRUPT or POLLING) for busoff, RX and TX processing
- List of HOH IDs that belong to a particular controller

The structure `Can_HardwareObjectType` contains these parameters:

- Handle ID of HRH or HTH
- Type of a hardware object (Basic CAN or Full CAN)
- Type of ID value (EXTENDED, STANDARD or MIXED)
- ID value
- Object type (whether is used as transmit or receive)
- Reference to the filter mask
- Mask which tells the driver whether Hardware Message Box should be occupied by this HOH

During L-PDU transmission, the CAN Driver writes a L-PDU in an buffer inside the CAN controller hardware. During L-PDU reception, the CAN Driver calls the RX indication function. These functions `Can_Write`, `handleRxMsgObject` (`CanIf_RxIndication` is located inside this function), as well as functions such as `Can_Init`, `Can_InitController`, or `Can_SetControllerMode` are located in `Can.c` and all necessary registers of DCAN are set in these functions. The function `Can_InitController` is called only from the CAN Interface and is responsible for setting initial values of DCAN registers, namely CAN Control Register (CTL), Bit Timing Register (BTR), and IF registers[14].

[13]A fixed amount of time which is derived from the CAN controller clock with a prescale factor
[14]For more information refer to [31].

4 ArcCore

In this chapter ArcCore company and their products are introduced. ArcCore open-source licensed products are described in Section 4.2. As we can see in the Section 4.2.1, several demo applications have already been created. List of some of these modules is included. In addition to open-source licensed products, ArcCore also offers commercial plugins with the graphical interface. Description of these plugins can be found in Section 4.3.

4.1 Software Implementation in General

As mentioned above the AUTOSAR partnership advocates the principle "Cooperate on standards, compete on implementation". According to this principle, several software vendors have hit the market with software implementation of the AUTOSAR standard. Some of these software suppliers are namely Vektor Informatik, Continental Engineering Services, Bosch, Freescale, Elektrobit, OpenSynergy, or ArcCore.
Offering of these companies varies. Few of them offer a complete stack that includes implementation of basic software, a basic software configurator, a real-time environment configurator, and also system tools like a software components builder or an extract builder. However, we can find companies among whose products are only some parts of the whole stack, usually lacking a system tooling but some of them offer even less, i.e. only a basic software implementation and a realtime environment configurator.
Nevertheless, almost all companies' products are licensed under the commercial license. ArcCore company provides an AUTOSAR embedded platform and an Eclipse based Integrated Development Environment (IDE) under Open Source license. Besides that, they also offer the commercial plugins for their IDE. The open-source license is the main reason why ArcCore products were chosen for the purpose of this project.

4.2 Open Source Licensed Products

Open source means that source code is downloadable free of charge and a user is allowed to modify and distribute the (modified) source code. An open source product by ArcCore is called Arctic Core. Artic Core

is the AUTOSAR embedded platform. It includes all basic software modules and their source codes as well as build scripts needed to build system according to AUTOSAR version 3.1. Nonetheless, as mentioned in the official website's latest news at the time of writing this thesis "global Tier1 has decided to base their next safety critical ECU for mass-production on ARCCORE AUTOSAR 4.x product portfolio" [9], so the transition to AUTOSAR version 4.x is expected.
Second free of charge product is called Arctic Studio. It is a complete Eclipse based IDE. It also includes GCC compilers for target boards. Additionally, a user can extend those by purchasing commercial compilers such as Freescale Codewarrior. Moreover, commercial plugins exist for easier development of AUTOSAR system. These plugins are discussed in Section 4.3.

4.2.1 Source Tree Examples

Arctic Core currently include examples for boards with MCUs such as Freescale MPC5xxx, HCS12, Arm Cortex M3 and R4. The examples illustrate functional implementation of some AUTOSAR basic software modules. These examples served as a basis for examples developed for our target boards. Detailed explanation is mentioned in subsequent chapters. Several examples for different target boards are located in the source tree of Arctic Core. The target boards STM3210C-EVAL and TMDX570LS20SMDK are based on ARM microcontrollers, the same microcontroller board family that is used for our target boards. These examples are:

- Simple OS: Demonstrates basic OS functionality – an alarm sets an event in a task, a task sets another event and waits for a new event.
- Simple RTE: Demonstrates basic Runtime Environment (RTE) functionality between software components.
- Simple CAN: Shows example of CAN communication – sends one CAN frame and receives one CAN frame.
- LED RTE: An application that flashes a LED on a board (with RTE defined).
- Ledmaster: An application that flashes a LED on a board. The pulse-width modulation is used. The frequency is changed by sending a new period in a CAN message.

These examples were created for following target boards:

- STM3210C-EVAL – Simple OS
- TMDX570LS20SMDK (TI TMS570LS) – Simple OS, Simple RTE
- QRtech MPC5567 G3 – Simple OS, LED RTE
- iSYSTEM MPC551x – Simple OS, Simple CAN
- ELMICRO CARD12 (HCS12) – Simple OS, Ledmaster

4.3 Commercial Plugins

ArcCore company offers commercial tools for Arctic Studio. They contain a graphical interface and are used to generate source code of the AUTOSAR system. By using these tools, a developer can easily define software components, used basic software modules and set up all necessary configuration of either BSW or RTE. The workflow of the toolchain is illustrated in Figure 14.

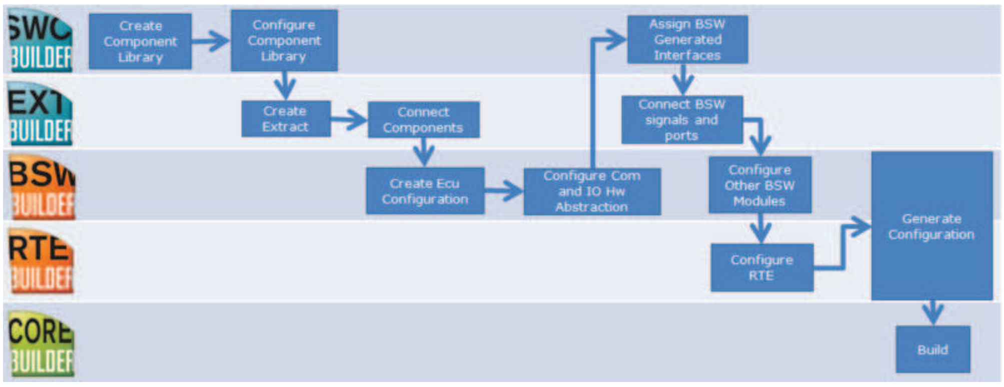


Figure 14 ArcCore Toolchain Workflow [11]

Most importantly, during the development of our examples, none of the commercial plugins were available. Therefore, all code had to be modified and created manually.

4.3.1 SWC Builder

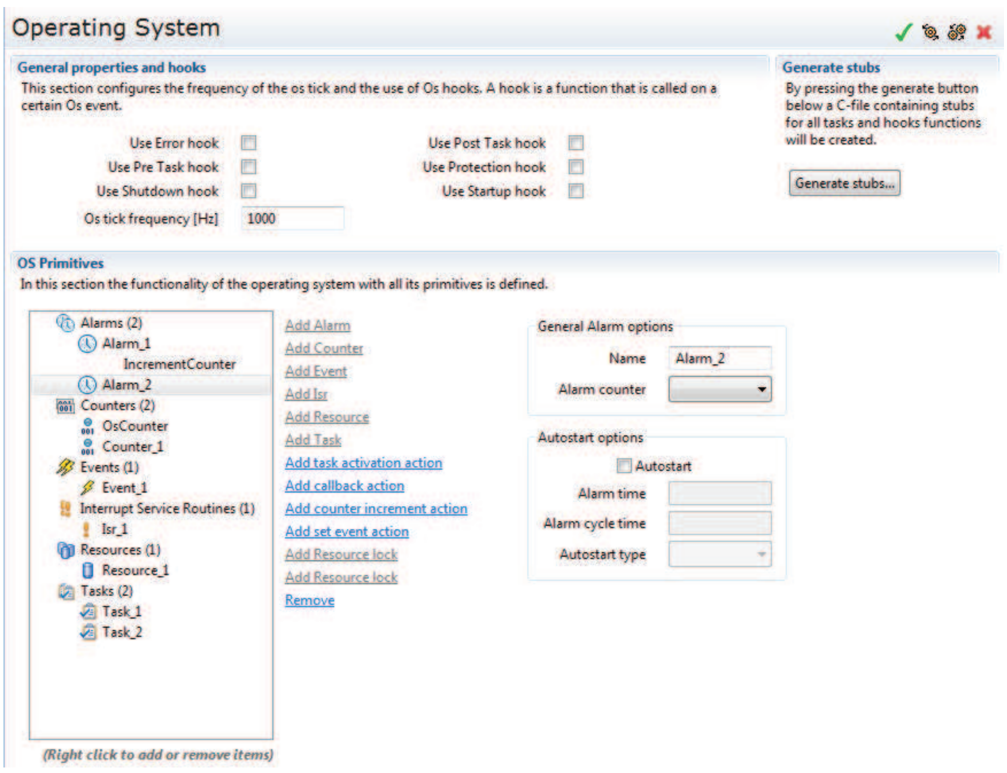
SWC Builder is a tool for creating and editing AUTOSAR Software Components, Port interfaces, etc. It is compatible with the AUTOSAR exchange format (*.arxml) as mentioned in 2.2. A user can pre-define a kit that will include objects needed to get started with the development, e.g. a server kit including server behaviour (events and runnable entities) and server implementation. Software components can be then created by using some of these pre-defined kits. Consequently, components, interfaces, data types and so on may be put into packages to keep them organized. The tool has also validation rules to identify incorrect configurations [10].

4.3.2 Extract Builder

Extract Builder is a tool for creating ECU extract. The tool connects all components with each other. Software components can be either created by using SWC builder or any third party software. All components included in an Eclipse project will be added to a library of components in the Extract Builder environment. After that, a user can create component instance, in other words add them to an ECU extract. Components added to an ECU extract can be also connected to ports and system signals. All connections are created either manually or can be created automatically by the Extract Builder. For larger models there is an overview available to display all components and their connections [10].

4.3.3 BSW Builder

BSW Builder is a tool for editing and generating BSW configurations for AUTOSAR. It is the key tool for making the development easier and faster. A developer can select which BSW modules to use and after that configure them through an editor tailored for a specific module. After the whole configuration is done, the tool generates configuration files for the platform based on Arctic Core. Not to mention that a validation tool is included, thus a developer receives immediate feedback about invalid or missing configuration [10].
To illustrates this, Figure 15 shows BSW Builder in use.



Operating System

General properties and hooks

This section configures the frequency of the os tick and the use of Os hooks. A hook is a function that is called on a certain Os event.

Use Error hook ☐
Use Pre Task hook ☐
Use Shutdown hook ☐
Os tick frequency [Hz] 1000
Use Post Task hook ☐
Use Protection hook ☐
Use Startup hook ☐

Generate stubs

By pressing the generate button below a C-file containing stubs for all tasks and hooks functions will be created.

Generate stubs...

OS Primitives

In this section the functionality of the operating system with all its primitives is defined.

- Alarms (2)
  - Alarm_1
    - IncrementCounter
  - Alarm_2
- Counters (2)
  - OsCounter
  - Counter_1
- Events (1)
  - Event_1
- Interrupt Service Routines (1)
  - Isr_1
- Resources (1)
  - Resource_1
- Tasks (2)
  - Task_1
  - Task_2

(Right click to add or remove items)

Add Alarm
Add Counter
Add Event
Add Isr
Add Resource
Add Task
Add task activation action
Add callback action
Add counter increment action
Add set event action
Add Resource lock
Add Resource lock
Remove

General Alarm options

Name Alarm_2
Alarm counter

Autostart options

☐ Autostart
Alarm time
Alarm cycle time
Autostart type

Figure 15 BSW Builder module configuration view of the Operating System [10]

4.3.4 RTE Builder

RTE Builder is a tool for configuration of the RTE. Even though, the tool is also a plugin to the Arctic Studio, ArcCore labels this tool as a plugin module to BSW Builder. A developer can use RTE Builder for mapping runnables to tasks. The runnables (or runnable entities) are executable functions of software components. In other words, the RTE Builder is used to specify how specific software components should be scheduled and executed on the platform. It uses an ECU extract as an input. The extract may be either generated by Extract Builder or any third party software. After setting up all configurations, the RTE code is generated. Moreover, the validation tool again checks whether the RTE code contains any errors [10].

5 Demo Applications

This chapter describes the configuration of the BSW modules and calling sequence within these modules for the demo applications that were created as part of this thesis. The goal of the first demo, called LED Blinker (Section 5.1), is to make a LED located on TMS570LS3137 HDK blinking. For the RPP board, a LED pin is not included on the board, thus the LED was added to a pin with different pin configured as GPIO. This is mentioned in Section 5.1.4. The goal of the second demo, called CAN communication (Section 3.11), is to exchange messages between two CAN controllers, namely DCAN1 and DCAN2 located on both TMS570LS3137 HDK and the RPP board.

5.1 LED Blinker – HDK

As mentioned in Chapter 3, the MCU Driver, the EcuM and the Port Driver are used in both demos. The MCU specific settings is configured in file `Mcu_Cfg.c`. The configuration set contains the clock setting, i.e. PLL setting. Since the number or RAM sectors is set to 0, no RAM sector configuration data are included. Other used modules are DIO Driver and IoHwAb.

5.1.1 Configuration of Port Driver

The type of `Port_PinType` uint32 is used because of bit shifting that exceeds 16-bit integer. The parameters for our LED example are the activation of pull-up/pull-down functionality and enabling open-drain capability. Configuration of the pin used in the example is:

- Pin mode: GIO
- Pin direction: Output
- Pin direction changeable: No
- Pin mode changeable: No
- Pulldown/pullup selection: None
- Open-drain: Disabled (push/pull mode)
- DMM used: No

As mentioned in Chapter 3, pointers used to access different ports are grouped into an array. In our LED blinker example, the N2HET1 base address was added:

```
PORT_2_BASE ((Port_RegisterType *)0xFFF7B848)
```

Due to our configuration, Data Output Register (DOUT) and Pull Disable Register (PULDIS) are set by calling `Port_RefreshPin` (Chapter 3, Section 3.3). In case of different configuration, Data Direction Register (DIR) or functionality of a pin (FUN register) may be changed. As mentioned in the Chapter 3, pins are defined with symbolic names and values. Values are defined in a way that a port number, a mask number and a pin number can be derived from a particular value. How this number are derived can be found in the source code. The value of our pin is set to 0×021b.

5.1.2 Pin Multiplexing

The used LED for TMS570LS3137 HDK is driven by signal N2HET1[27]. The TMS570LS3137 device uses I/O Multiplexing and Control Module to control the input/output multiplexing on the device. Multiplexors are controlled by the memory-mapped PINMMRx registers that effect functionality of the particular pin ball.
For our case, the base address of 0×FFFFEB10 that represents byte field PINMMR0[31:0] is utilized. The N2HET1[27] is then represented by the selection bit PINMMR0[26]. The code that set and clears particular bits was added in order to hack N2HET1[27] to the pin A9.
In the LED blinker examples, after the RTE calls the function for setting a digital output, the signal is mapped to a IoHwAb port where the quality of signal is determined. If the quality is good, the task provides setting of the digital output via the call of DIO function. Detailed calling sequence is described in section 5.1.5.

5.1.3 Configuration of DIO Driver

Similarly as configuration of the Port driver, the DIO driver includes types that are used to define channels and ports. `Dio_ChannelType` is the type representing a numerical ID of a channel. A numerical ID of a port is described by defining `Dio_PortType`.
In our examples, DIO channels are described by a symbolic name defined in `Port_Cfg.h`. That means values of `Dio_ChannelType` and `Port_PinType` remain the same.

5.1.4 Changes for the RPP Board

The RPP board does not include a LED located on the board. Due to this fact, a pin marked as FANCONTROL was used as GPIO. The LED was connected to this pin. Since a Data Modification Module (DMM) pin was used for the GPIO purpose, the parameter whether DMM is used was added. The only change in the configuration of the used pin is the change of the parameter DMM used to value Yes. Furthermore, instead of the N2HET1 base address, the DMM base address was added as follows:

```
PORT_3_BASE ((Port_RegisterType *)0xFFFFF76C)
```

Last change for the LED blinker to be functional for the RPP board was to change the value of the pin to 0×0300.

5.1.5 Calling sequence

This section describes sequence of calling of functions in order to set the new level value of a pin which will lead to LED blinking.
After the booting a device, an overall initialization is performed. The Operating System (OS) finds the tasks with the highest priority which results in call of `StartupHook`. During this procedure, the OS is started. After that, the OS starts the application by calling `ActivateTask` (in our case activated task is the function `Scheduled`). The sequence is carried out through the following modules: OS, RTE, IoHwAb, DIO and results in setting the new level value of a pin. The sequence is shown in Figure 16.
All the functions are encapsulate within the function that precedes it. For example, the function `Dio_WritePort` is called from the function `Dio_WriteChannel` which is called from the function `IoHwAb_Set_Digital_DigitalSignal_Led4` and so on as we can see in the Figure 16. The following were used (starting from the RTE, ending in the DIO module):

- `Scheduled`
- `Rte_BlinkerRunnable`
- `BlinkerRunnable`
- `Rte_Call_Blinker_LED_Port_Set`
- `Rte_Digital_Output_Set`
- `DigitalOutput_Set`
- `IoHwAb_Set_Digital`
- `IoHwAb_Set_Digital_DigitalSignal_Led4`
- `Dio_WriteChannel`
- `Dio_WritePort`

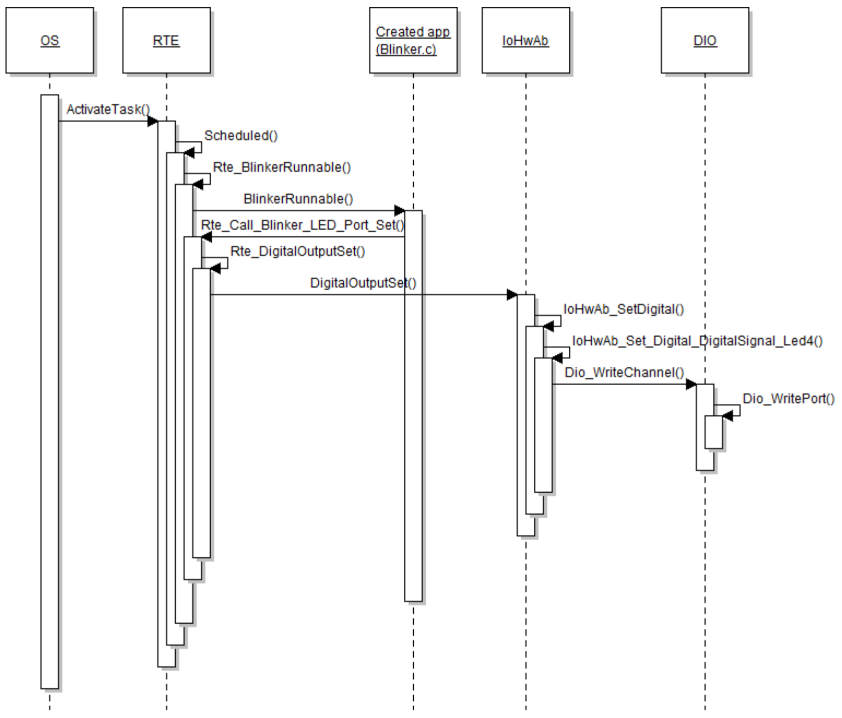


Figure 16 UML sequence diagram for the LED blinker

5.2 CAN Communication – HDK, RPP

This demo application was developed for the both board with the same settings. This demo has been successfully made working only with one CAN controller defined (DCAN1). With the both controllers (DCAN1 and DCAN2) an unknown error forced the devices to stop working. The application was successfully flashed on the target boards. The programming was fine, however, after executing the code, unknown sequence caused the JTAG lock. We have not been able to connect to the device through JTAG since that. It detects JTAG chain and targed ID but then locks on the attempt to initialize ARM debug port. It has been assumed that an error that might have caused this problem is that 128-bit code in the OTP (one time programmable) address of AJSM an Advanced JTAG Security Module (AJSM) was changed which caused securing the device. Namely, the error during trying to debug was *"Unable to access DAP"* (Debug Access Point).

5.2.1 Configuration of Port Driver

The only important thing here is that the base addresses were added for DCAN1 and DCAN2 controller: 0×FFF7DDE0, 0×FFF7DFE0 respectively. The values of pins were defined as 0×0800 for DCAN1 transmission, 0×0801 for DCAN1 reception, 0×0900 for DCAN2 transmission, and 0×0901 for DCAN2 reception.

5.2.2 Configuration of COM

The signals used in our CAN communication application are `Arg1`, `Arg2`, `ResultSig`, `FreqIndSig`, and `FreqReqSig`. Signals `Arg1` and `Arg2` are both integers (e.g. 3 and 5). These two numbers are then multiplied and the resulting value is send back as `ResultSig`. Signal `FreqReqSig` sends a new frequency for LED blinking. This signal is received as `FreqIndSig`. Tables 1 and 2 show configuration of all these signals:

ComSignal	Arg1	ResultSig	Arg2
HandleId	0	1	2
IPduHandleId	2	3	2
TransferProperty	PENDING	TRIGGERED	PENDING
SignalInitValue	5	0	3
BitPosition	7	7	15
BitSize	8	8	8
SignalType	uint8	uint8	uint8
Notification	NULL	NULL	NULL

Table 1 Configuration of COM signals 1

ComSignal	FreqIndSig	FreqReqSig
HandleId	3	4
IPduHandleId	0	1
TransferProperty	TRIGGERED	PENDING
SignalInitValue	0	1000
BitPosition	7	7
BitSize	32	32
SignalType	uint32	uint32
Notification	NULL	Rte_COMCbk_FreqReqSig

Table 2 Configuration of COM signals 2

The Table 3 shows the configuration of PDUs. These PDUs are `RX_PDU`, `TX_PDU`, `FreqInd`, and `FreqReq`. They contain signals defined above. Specifically, `RX_PDU` contains signal `Arg1` and `Arg2`, `TX_PDU` contains the signal `ResultSig`, `FreqInd` contains the signal `FreqIndSig` and `FreqReq` contains the signal `FreqReqSig`.

ComIPdu	FreqInd	FreqReq	RX_PDU	TX_PDU
SignalProcessing	DEFERRED	DEFERRED	DEFERRED	DEFERRED
IPduSize	8	8	8	8
IPduDirection	SEND	RECEIVE	RECEIVE	SEND
TxMode	DIRECT	NONE	NONE	DIRECT

Table 3 Configuration of I-PDUs

I-PDU signal lists in the source code looks as follows:

```
const ComSignal_type * const ComIPduSignalRefs_FreqInd[] = {
  &ComSignal[ FreqIndSig ],
  NULL, };
const ComSignal_type * const ComIPduSignalRefs_FreqReq[] = {
  &ComSignal[ FreqReqSig ],
  NULL, };
const ComSignal_type * const ComIPduSignalRefs_RX_PDU[] = {
  &ComSignal[ Arg1 ],
  &ComSignal[ Arg2 ],
  NULL, };
const ComSignal_type * const ComIPduSignalRefs_TX_PDU[] = {
  &ComSignal[ ResultSig ],
  NULL, };
```

5.2.3 Configuration of PDU Router

Configuration of PDU Router was implemented for following PDUs: `RX_PDU`, `TX_PDU`, `FreqInd`, and `FreqReq`. These PDUs are described in Section 5.2.2. The table 4 shows the destination module (`DestModule`) and the data provision mode for the PDUs. The table 5 shows the source module (`SrcModule`) and the SDU length. The configuration used in our examples is following:

PduRDestPdu	DestModule	DataProvision
FreqInd	CANIF	NO_PROVISION
FreqReq	COM	NO_PROVISION
RX_PDU	COM	PDUR_DIRECT
TX_PDU	CANIF	PDUR_DIRECT

Table 4 Configuration of destination PDUs

PduRRoutingPath	FreqInd	FreqReq	RX_PDU	TX_PDU
SrcModule	COM	CANIF	COM	CANIF
SduLength	0	0	8	8

Table 5 Configuration of routing paths for PDUs

From the tables 4 and 5, we can see that for transmission the source module is the COM module and the destination module is the CAN Interface. Contrarily, for reception the source module is the CAN Interface module and the destination module is the COM module.

5.2.4 Configuration of CAN Interface

The functionality of the CAN Interface is described in Chapter 3, Section 3.10. Chapter 3, Section 3.10 also describes the used parameters and sequences for a transmit request or a reception indication. The detailed configuration will not be described in this chapter because of extensiveness and can be found in the source code.

5.2.5 Configuration of CAN Driver

There are two controller and two HOH objects defined. The controllers are `DCAN1` and `DCAN2`. Both have the same configuration except their IDs, references to CPU clocks and lists of HOHs. Their baud rate is 500, propagation delay is 5 and phase segments are 3. All of processing types are `PROCESS_TYPE_INTERRUPT` which is represented by the value of 0×00000400. HOH for transmission has an ID `TxHwObject` and its type is `CAN_OBJECT_TYPE_TRANSMIT`. For reception an ID is `RxHwObject` and its type is `CAN_OBJECT_TYPE_RECEIVE`. Both have handle type `HANDLE_TYPE_BASIC` which represents Basic CAN, ID type `CAN_ID_TYPE_EXTENDED`. Furthermore, both have also the same mask with the value of 0×000007FF.

5.3 Calling sequence

This section describes the calling sequence for sending a signal with subsequent receiving of a signal.

5.3.1 Transmission

As the first step, a function e.g. `Rte_TesterRunnable` is called in the file `Rte.c`. Writing of signal data is mapped into this function while generating the RTE. This function is subdivided into three sub-functions. One part for transmission, one part for reception, and one part where operations like multiplying of received signals, or setting an alarm may be defined.
The part for transmission includes the function `Com_SendSignal`. Parameters of this function are a signal ID and pointer to signal data. Within this function `Com_WriteSignalDataToPdu` is called[1]. Besides,

[1]Located in `Com_com.c`

based on configuration, an update bit may be set or signal may be arranged for transmission trigger. Inside Com_WriteSignalDataToPdu we finally get data by calling Com_WriteSignalDataToPduBuffer. After writing signal data to a PDU buffer, the following OS task is called.
This following task includes two function: Com_MainFunctionTx and Com_MainFunctionRx. Com_MainFunctionTx [2] causes the transmission. After verification that an I-PDU should really be sent, the transmission is triggered by calling Com_TriggerIPduSend. Inside this function PduR_ComTransmit is called, which forwards a PDU from the COM module to the PDU Router module. Parameters of PduR_ComTransmit are an ID of an outgoing I-PDU and pointer to the routing table. Within this function PduR_ARC_Transmit is called. The function determines the destination and returns the function PduR_ARC_RouteTransmit. Based on a destination, PduR_ARC_RouteTransmit calls the function for transmitting into a right module, in our case to the CAN Interface module, so CanIf_Transmit is called. Within this function located in CanIf.c, Can_Write is called, which copies a PDU into CAN hardware by setting registers in the file Can.c.

5.3.2 Reception

As mentioned above, a function such Rte_TesterRunnable consists of three sub-functions. The part for reception includes the function Com_ReceiveSignal[3]. This function is responsible for reading signal data from PDU buffer. This is achieved by calling the function Com_ReadSignalDataFromPduBuffer.
For the indication of reception, Com_MainFunctionRx is called. It is located in the same task as Com_MainFunctionTx. Com_MainFunctionRx is located in Com_Sched.c. If all necessary configuration is set in right way so all conditions in if statements were met, it invokes receive indication of a particular signal by this sequence:

```
  if (signal->ComNotification != NULL) {
    signal->ComNotification();
  }
    Arc_Signal->ComSignalUpdated = 0;
```

This notification may be e.g. the RTE callback function that would set an alarm.

[2]Located in Com_Sched.c
[3]Located in Com_com.c

6 Conclusion

At the very beginning of realization of the project, I had to get familiar with TMS570LS3137 Hardware Development Kit (HDK) by Texas Instruments and the AUTOSAR on its own. Software by Texas Instruments, including Code Composer Studio and HalCoGen, was used at the beginning to demonstrate basic applications running on this hardware development kit. Then, I had to develop several demo application based on ArcCore platform. The main disadvantage of the development process was that commercial plugins to Arctic Studio were not available. Due to this fact, the development was in form in creating and modifying source code by hand. This was very helpful from my point of view because by studying the source code instead of using tools that would generate it, I had a chance to get a little deeper insight into basic software modules included in the AUTOSAR architecture.
As for writing the code for demo examples, configuration files of basic software modules needed for LED blinker and CAN communication examples had to be created. The key ideas were taken from examples for other target boards such as STM3210C-EVAL, iSYSTEM MPC551x, or TMDX570LS20 SMDK. Nevertheless, to make demo applications running, some core files of basic software modules included in Artic Core architecture had to be modified as well. Our examples were meant to be developed for TMS570LS3137 HDK and subsequently for RPP board which uses the same microcontroller. During the development process, documentation of the target board and its microcontroller have had to be studied to get better idea how the device works. Nonetheless, AUTOSAR documentation have had to be simultaneously studied as well to understand the concept of this standard and functionality and interconnection of its basic software modules.
LED Blinker application was successfully made running for both target boards. For TMS570LS3137 HDK GIO pins with LEDs were used. However, RPP board does not contain any LEDs, so a LED was connected to DMM pin which is labeled as `FAN_CONTROL` in the documentation.
CAN communication application was compiled. This example was functional only if one CAN controller on TMS570LS3137 HDK was defined (specifically DCAN1). From this controller a message was sent and process of reception seemed to be functional as well. However, after DCAN2 was initialized as well in order to exchange messages between these two controllers, the hardware (CPU) stopped working. This bug caused

JTAG device to stop working. Due to this reason, we were not able to connect to the device through JTAG after that. The detailed explanation of this error is described in Chapter 5, Section 3.11. Several experiments have been tried in order to make the JTAG working. By the date of finishing this project (May 2013), this problem has not been solved, even with the help of Texas Instruments (`http://e2e.ti.com/support/microcontrollers/hercules/f/312/t/264498.aspx`).

Appendix A

Content of the Attached CD

- The thesis in *.pdf format
- The source code of Artic Core and created demo applications.
- Diff file showing changes made during the development of the demo applications (*.patch format)

Bibliography

[1] *TMS570LS3137*. `http://rtime.felk.cvut.cz/hw/index.php/TMS570LS3137`. (Visited on 05/12/2013).

[2] Simon Fürst and Heiko Dörr. “AUTOSAR – An open standardized software architecture for the automotive industry”. In: 1st AUTOSAR Open Conference & 8th AUTOSAR Premium Member Conference. BMW. Cobo Center, Detroit, MI, USA, Oct. 2008.

[3] *AUTOSAR Basics*. `http://www.autosar.org/index.php?p=1&up=1&uup=0`. (Visited on 05/12/2013).

[4] *AUTOSAR Technical Overview*. `http://www.autosar.org/index.php?p=1&up=2&uup=0`. (Visited on 05/12/2013).

[5] AUTOSAR GbR. *AUTOSAR - The Worldwide Automotive Standard for E/E Systems - EN*. `http://www.autosar.org/download/papersandpresentations/AUTOSAR_Brochure_EN.pdf`.

[6] Rao Nagarjuna Kandimala. *Automotive Open System Architecture & DaVinci Developer Software*. Tech. rep. Czech Technical University in Prague, Prague, Aug. 2012.

[7] Nico Naumann. *AUTOSAR Runtime Environment and Virtual Function Bus*. Tech. rep. Department for System Analysis and Modeling, Hasso-Plattner-Institute für Softwaresystemtechnik, Potsdam, 2009.

[8] Robert Warschofsky. *AUTOSAR Software Architecture*. Tech. rep. Hasso-Plattner-Institute für Softwaresystemtechnik, Potsdam, 2009.

[9] *Global Tier1 chooses ARCCORE solutions*. `http://www.arccore.com/2013/05/global-tier1-chooses-arccore-solutions/`. 2013-05-06. (Visited on 05/12/2013).

[10] *ArcCore Products*. `http://www.arccore.com/products/`. (Visited on 05/12/2013).

[11] *Toolchain Workflow*. `http://www.arccore.com/wiki/Toolchain_Workflow`. (Visited on 05/12/2013).

[12] AUTOSAR GbR. *Model Persistence Rules for XML*. 2.4.0. Nov. 2011.

[13] AUTOSAR GbR. *Specification of RTE*. 3.2.0. Oct. 2011.

[14] AUTOSAR GbR. *Project Objectives*. 3.0.0. Nov. 2011.

[15] AUTOSAR GbR. *Layered Software Architecture*. 3.2.0. Oct. 2011.

[16] AUTOSAR GbR. *Specification of ECU State Manager*. 3.0.0. Nov. 2011.

[17] AUTOSAR GbR. *Specification of MCU Driver*. 3.2.0. Dec. 2012.

[18] AUTOSAR GbR. *Requirements on MCU Driver*. 3.0.0. Mar. 2009.

[19] AUTOSAR GbR. *Specification of Port Driver*. 3.2.0. Nov. 2010.

[20] AUTOSAR GbR. *Requirements on Port Driver*. 2.0.5. Dec. 2009.

[21] AUTOSAR GbR. *Specification of I/O Hardware Abstraction*. 3.2.0. Nov. 2011.

[22] AUTOSAR GbR. *Specification of DIO Driver*. 2.5.0. Sept. 2011.

[23] AUTOSAR GbR. *Requirements on DIO Driver*. 2.1.0. Oct. 2010.

[24] AUTOSAR GbR. *Specification of Communication*. 4.2.0. Nov. 2011.

[25] AUTOSAR GbR. *Requirements on Communication*. 3.1.0. Sept. 2011.

[26] AUTOSAR GbR. *Specification of PDU Router*. 3.2.0. Nov. 2011.

[27] AUTOSAR GbR. *Specification of CAN Interface*. 5.0.0. Dec. 2011.

[28] AUTOSAR GbR. *Specification of CAN Driver*. 4.0.0. Nov. 2011.

[29] AUTOSAR GbR. *Requirements on CAN*. 4.0.0. Oct. 2011.

[30] AUTOSAR GbR. *Specification of Operating System*. 5.0.0. Nov. 2011.

[31] Texas Intruments Inc. *TMS570LS31x/21x 16/32-Bit RISC Flash Microcontroller - Technical Reference Manual*. SPNU499A. Nov. 2012.

[32] Texas Instruments Inc. *TMS570LS3137 16/32-Bit RISC Flash Microcontroller*. SPNS162A. Nov. 2012.

[33] Texas Instruments Inc. *TMS570LS31x Hercules Development Kit (HDK) - User's Guide*. SPNU509A. Sept. 2012.

[34] ARM Holdings. *Cortex-R4 and Cortex-R4F Technical Reference Manual*. r1p4. 2011.

[35] Gareth Leppla. "Mapping Requirements To AUTOSAR Software Components". Master's Thesis. Waterford Institute of Technology, 2008.

[36] Michal Horn. "Software obsluhující periferie a FlexRay na automobilové řídicí jednotce". Master's Thesis. Czech Technical University in Prague, 2013.