



DEGREE PROJECT IN INFORMATION AND COMMUNICATION
TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2016

Increasing efficiency in ECU function development for Battery Management Systems

SHIVARAM SINGH RAJPUT



Increasing efficiency in ECU function development for Battery Management Systems

*Master of Science thesis in Embedded Systems (30 ECTS credits)
at the school of Information and Communication Technology*

KTH Royal Institute of Technology

Stockholm, Sweden

October 2016

by,

Shivaram Singh Rajput

Supervisors: Christian Fleischer

Advanced Battery Technology, NEVS

Yuan Yao

ESY ELEKTR. O INBYGGDA SYSTEM, KTH

Examiner: Zhonghai Lu

ESY ELEKTR. O INBYGGDA SYSTEM, KTH

ABSTRACT

In the context of automotive industries today, the focus of ECU function development is always on finding the best possible combinations of control algorithms and parameter. The complex algorithms with broad implementation range requires optimal calibration of ECU parameters to achieve the desired behaviour during the drive cycle of the vehicle.

With the growing function complexity of automotive E/E Systems, the traditional approaches of designing the automotive embedded systems are not suitable. In order to overcome the challenge of complexity, many of the leading automotive companies have formed a partnership in order to develop and establish an open industry standard for automotive E/E architecture called AUTOSAR. In this thesis, toolchain for ECU function development following AUTOSAR standard and an efficient measurement and calibration mechanism using XCP on CAN will be investigated and implemented. Two toolchains will be proposed in this thesis, describing their usage in different stages of ECU function development and in calibration. Both these toolchains will be tested to prove its working.

Keywords: AUTOSAR, XCP, Toolchain, CAN Communication, Measurement and Calibration, TMS570LS1227

SAMMANFATTNING

I området utveckling av funktionalitet på elektroniska styrsystem inom bilindustrin idag, ligger fokus på att finna den bästa kombinationen av reglermetoder och styrparametrar. Dessa avancerade system, med breda användningsområden, kräver bästa möjliga injustering av dess kalibrerbara parametrar, för att nå önskat beteende vid användning av fordonet.

Det ökande omfånget av funktionskraven på styrsystemen, innebär att sedvanlig metodik för utveckling av dessa system inte är lämplig. För att kunna lösa dessa svårigheter, har de stora inom bilindustrin ingått ett samarbete, där de tillsammans skapat och utvecklar en industristandard för funktions- och systemutveckling av styrsystem. Standarden kallas AUTOSAR. Denna rapport beskriver hur en kedja av utvecklingsverktyg som följer AUTOSAR-standarderna kan användas, för att undersöka och använda en metod för systemövervakning och parameterkalibrering, genom användning av XCP över CAN.

Sökord: AUTOSAR, XCP, utvecklingsverktyg, CAN-kommunikation, Kalibrering och övervakning, TMS570LS1227

ACKNOWLEDGEMENTS

I thank National Electric Vehicle Sweden and my industrial supervisor, Dr. Christian Fleischer for giving me an opportunity to work in such an interesting and challenging topic. I thank my examiner Dr. Zhonghai Lu for his support, guidance, and advice throughout my thesis work. It is a great pleasure to acknowledge my deepest thanks to Björn Nyman at NEVS for always supporting and guiding me technically whenever I ran into a trouble spot. I would also like to thank all my colleagues at Advanced Battery Technology department, NEVS for their support and good time we had together.

I thank ArcCore for their technical support through their web portal. I am grateful to AUTOSAR for permitting me to use the pictures from AUTOSAR specifications in this thesis.

I would like to thank EIT Digital Master School for giving me an opportunity to study Master of Science in Embedded Systems at TU Berlin and KTH. I am grateful to all the Professors at AES department, TU Berlin and ICT School, KTH.

Finally, I express my deepest thanks to my parents, to my brother, and to my friends, for supporting and encouraging me throughout my years of study. This accomplishment would not have been possible without their support.

Shivaram Singh Rajput
Trollhättan, Sweden
September 2016

ABBREVIATIONS

A2L	ASAM MCD-2 MC Language
ADC	Analog to Digital Converter
ADT	Application Data Type
API	Application Programming Interface
ARXML	AUTOSAR standard Extensible Mark-up Language
ASAM	Association for Standardisation of Automation and Measuring Systems
AUTOSAR	AUTomotive Open System Architecture
BMS	Battery Management System
BSW	Basic Software
BswM	Basic Software Manager
CAN	Controller Area Network
CanIf	CAN Interface
CanSM	CAN State Manager
CCP	CAN Calibration Protocol
CMD	Command
COM	Communication
ComM	Communication Manager
CPU	Central Processing Unit
CTO	Command Transfer Objects
DAQ	Data Acquisition
DLC	Data Length Code
DTO	Data Transfer Object
E/E	Electrical and Electronics
ECU	Electronic Control Unit
EcuM	ECU Manager
ERR	Error
ETK	Ethernet Kable
EV	Event Packet
FIFO	First In First Out
GIO	General Purpose Input Output
GPL	General Public License
I/O	Input Output
ID	Identifier
IDT	Implementation Data Types
I-PDU	Interaction Layer PDU
LED	Light Emitting Diode
LIN	Local Interconnect Network
L-PDU	Data Link Layer PDU

MC	Measurement and Calibration
MCAL	Micro Controller Abstraction Layer
MCD	Measurement, Calibration and Diagnostics
MCU	Micro Controller Unit
NEVS	National Electric Vehicle Sweden
NM	Network Manager
N-PDU	Network Layer PDU
NV	Non volatile
ODT	Object Description Tables
OEM	Original Equipment Manufacturer
OS	Operating System
OSEK/VDX	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen / Vehicle Distributed Executive
	Open Systems and Corresponding interfaces for Automotive Electronics/ Vehicle Distributed Executive
PC	Personal Computer
PCI	Protocol Control Information
PDU	Protocol Data Unit
PduR	PDU Router
PID	Packet Identifier
PPort	Provides Port
PWM	Pulse Width Modulation
RAM	Random Access Memory
RES	Command Response Packet
RISC	Reduced Instruction Set Computing
RPort	Requires Port
RTE	Run Time Environment
RX	Reception
SCI	Serial Communication Interface
SDU	Service Data Unit
SERV	Service Request Packet
SPI	Serial Peripheral Interface
STIM	Stimulation
SWC	Software Component
SWCD	Software Component Description
SYSD	System Description
TP	Transport
TX	Transmission
USB	Universal Serial Bus
VFB	Virtual Function Bus
XCP	Universal Measurement and Calibration Protocol
XETK	Extended Ethernet Kable

CONTENTS

ABSTRACT	i
SAMMANFATTNING	ii
ACKNOWLEDGEMENTS	iii
ABBREVIATIONS	iv
CONTENTS.....	vi
1 INTRODUCTION	1
1.1 Company	1
1.2 Background	1
1.3 Obejctive	2
1.4 Delimitation	2
1.5 Outline.....	2
2 LITERATURE	3
2.1 AUTOSAR.....	3
2.1.1 AUTOSAR Layered Software Architecture	3
2.1.1.1 Application Layer	4
2.1.1.1.1 Software Components	4
2.1.1.1.2 Ports	5
2.1.1.1.3 Port-Interfaces	5
2.1.1.1.4 Software component communication	5
2.1.2 Virtual Function Bus (VFB) and Run Time Environment (RTE)	6
2.1.2.1 Interfaces	8
2.1.3 Basic Software Layer	9
2.2 AUTOSAR Communication Cluster	10
2.2.1 Controller Area Network (CAN)	10
2.2.2 AUTOSAR Communication Stack.....	11
2.3 Basic Software modules	13
2.3.1 PDU Router	13
2.3.1.1 PDUs	14
2.3.2 AUTOSAR COM.....	15
2.3.3 COM Manager (ComM).....	16
2.3.4 CAN Interface (CanIf)	16
2.3.5 CAN Driver	16
2.3.6 CAN State Manager.....	17
2.3.8 ECU Manager	19
2.3.8 Microcontroller Unit (MCU) driver	20
2.3.9 Port Driver	20
2.4 Universal Measurement and Calibration Protocol (XCP)	20
2.4.1 Communication Model	21
2.4.2 XCP Transport Layer	24
2.4.2.1 XCP over CAN	24

2.4.3 Online Calibration	25
2.4.4 DAQ Lists- Data acquisition lists.....	25
2.4.5 Configuring DAQ lists	27
2.4.6 Data Stimulation Lists- STIM Lists	28
2.4.7 XCP module in AUTOSAR	28
2.4.8 ASAM MCD-2 MC	29
2.5 AUTOSAR Methodology	29
3 PROJECT METHODOLOGY	31
3.1 Method	31
3.2 Development tools.....	32
3.2.1 Software	32
3.2.1.1 Arctic Studio and Arctic Core	32
3.2.1.2 dSPACE- SystemDesk and TargetLink.....	33
3.2.1.3 ETAS INCA.....	34
3.2.1.4 MATLAB/SIMULINK	34
3.2.1.5 Code composer studio	34
3.2.1.6 Vector CANdb++ editor	34
3.2.1.7 BUS MASTER	34
3.2.2 Hardware	35
3.2.2.1 Texas Instruments – TMS570LS1227	35
3.2.2.2 PEAK Systems – PCAN USB adapter	36
3.2.3.3 ETAS ECU and BUS interface module.....	36
4 AUTOSAR FUNCTION DEVELOPMENT.....	37
4.1 Modeling Application layer	37
4.1.1 SWC Model.....	37
4.1.2 Modeling SWC with Arctic Studio	38
4.1.3 Modeling SWC with SystemDesk	40
4.1.4 Runnable	43
4.2 System modeling.....	45
4.2.1 System modeling with Arctic Studio	46
4.2.2 System modeling with SystemDesk	47
4.3 Basic Software Configuration.....	48
4.3.1 Microcontroller Unit	49
4.3.2 ECU Manager	49
4.3.3 Port Driver	50
4.3.4 CAN Driver.....	52
4.3.5 CAN Interface	53
4.3.6 EcuC.....	54
4.3.7 PDU Router	54
4.3.8 COM	56
4.3.9 COM Manager	57
4.3.10 CAN State Manager.....	57
4.3.11 Basic Software Manager	58
4.3.12 XCP	58
4.3.13 Operating System.....	60
4.3.14 Run time environment	63

4.4 Validation and Code generation	63
4.4.1 Executable for the MCU	64
4.4.2 Generating A2L file.....	65
4.5 Measurement and Calibration System Setup	65
4.6 Testing.....	65
4.6.1 Test Setup-1	66
4.6.2 Test Setup -2	66
5 RESULTS	67
5.1 AUTOSAR Toolchain	67
5.2 Complete toolchain.....	68
5.3 Toolchain flow.....	69
5.4 ECU software architecture	70
5.5 Testing.....	70
5.5.1 Testing with setup 1.....	70
5.5.2 Testing with setup -2.....	72
6 DISCUSSION AND CONCLUSION.....	74
6.1 Project Method	74
6.2 Conclusion	74
6.3 Future work.....	75
BIBLIOGRAPHY.....	76
LIST OF FIGURES	79
APPENDIX	81

1 INTRODUCTION

This chapter gives a brief background knowledge along with the purpose of this thesis. The delimitations set for this thesis are also discussed.

1.1 Company

This thesis was conducted at Advanced Battery Technology department, National Electric Vehicle Sweden (NEVS) in Trollhättan. NEVS is a relatively new car manufacturer which acquired the main assets of SAAB Automobile in 2012. The main vision of NEVS is to tackle the global warming problem by shaping the mobility for a more sustainable future. NEVS has dedicated itself, to design premium electric vehicles and mobility experiences that are simple, distinctive and engaging, in-order to shape a brighter and cleaner future for all [1]. The headquarters of NEVS is located in Trollhättan, Sweden with a manufacturing plant and global R&D center.

1.2 Background

In the functional development of an Electronic Control Unit (ECU) software, the parameters of the control algorithm can be only partially decided by software simulation. When the functions algorithm is being executed in the ECU, the parameter values such as curves, maps, characteristics and values can be obtained and tuned only on the test bench or in driving cycles. This process of optimizing or tuning a control algorithm to get the desired behaviour from the system is called as ECU Calibration.

Normally, software development has separate phases, code development phase where a software developer uses a programming language to realize the algorithms and the complete application. Then, an application engineer sets the parameter to the right values without modifying the function itself. The calibration of parameters can either be done online or offline. In offline calibration, the values of the parameters are changed in the binary file and by flashing this new file into the ECU memory, the behaviour of the ECU can be changed. In online calibration, the ECU calibration memory is accessed and the parameter is modified while the application is running on the ECU, this is also known as “on the fly” calibration.

With the increasing complexity of Electrics/Electronics (E/E) in the automotive domain and an increasing number of ECUs led to the establishment of an open standard to manage the growing E/E complexity and improve development efficiency. This Standard is known as AUTOSAR – AUTomotive Open System ARchitecture. The AUTOSAR architecture is a layered architecture consists of whole software stack for ECU communication and system services. This software stack is known as AUTOSAR Basic Software (BSW) which is a platform that integrates the hardware independent SW applications [2].

A basic calibration system consists of an ECU and bus interface, a link to the host PC, and a PC application. The physical connection between the development tool and the ECU is through

a measurement and calibration protocol. In this work, Universal measurement and calibration protocol (XCP) will be used to establish the physical connection between the calibration tool and the ECU. XCP was developed for implementing measurement and calibration via different transmission media, for e.g. XCP on CAN, XCP on Ethernet, XCP on Flexray, etc.

AUTOSAR BSW layer contains XCP module, in this thesis, XCP module will be configured for XCP on CAN. The initial investigation of this work will look into the AUTOSAR toolchain, CAN communication stack configuration and later, the establishment of calibration mechanism between the ECU and calibration tool. The hardware used in this toolchain will be Texas instruments TMS570LS1227, a high performance automotive-grade microcontroller for safety-critical applications.

1.3 Objective

The main objective of this thesis is to establish a toolchain for ECU function development following AUTOSAR standard and to have an optimal calibration mechanism with XCP over CAN. This thesis will be focused on developing a simple example which gets its inputs via CAN communication, the configuration of the communication stack of AUTOSAR BSW for CAN communication, to configure XCP module of the BSW to achieve XCP over CAN, and to establish the calibration mechanism between the ECU and calibration tool.

1.4 Delimitation

In this thesis, the actual ECU software for Battery Management System (BMS) will not be integrated into the application layer, instead, a simple application will be implemented which serves as an example for future work. Though AUTOSAR supports different communications such as LIN, FlexRay, and Ethernet, in this thesis only CAN communication stack will be configured as the BMS application will receive all the signals on CAN bus.

1.5 Outline

The second chapter of the report contains theoretical background of the necessary concepts for AUTOSAR, CAN, calibration, and XCP. The third chapter covers the project methodology and brief introduction to all hardware and software tools used in this thesis. In the fourth chapter, the AUTOSAR development steps followed to design the system considered in this thesis and the test setup is explained in detail. Chapter five presents the result of this thesis work i.e. the toolchains, final ECU software architecture, and the result of testing. The last part of the report includes the discussion, conclusion and proposal of how this thesis can be used for future work.

2 LITERATURE

This chapter introduces to the theoretical pre-study necessary to work on this thesis. AUTOSAR is a vast topic, thus, only the necessary concepts are discussed in this chapter. This chapter also includes concepts of XCP and online calibration.

2.1 AUTOSAR

With the increase in innovative vehicle applications, the modern day automotive E/E architecture has attained a high level of complexities, thus requires different approaches of technology in order to handle it in an adequate manner and fulfil higher expectations of passengers and legal requirements. Leading Original Equipment Manufacturers (OEMs) and Tier 1 suppliers have recognised this challenge and worked collaboratively to overcome this challenge. An open and standardized automotive software architecture called AUTOSAR was jointly developed by OEMs, tier-1 suppliers and tool developers in 2002 [3] [4].

The motto of AUTOSAR is “Cooperate on standards, compete on implementation”. The main motivations of AUTOSAR include: managing complex E/E systems by providing ease of modification; adaptation to the change in software and update options, reusing the solutions for different variants, and improving the quality and reliability of these systems [4].

The main goals of AUTOSAR are to standardize the basic software functionalities of automotive ECUs, define an open architecture, establish a partnership with various partners, adapt the software to different vehicle and platform variants, ease of transfer of software between partners, and develop highly reliable components [5].

2.1.1 AUTOSAR Layered Software Architecture

To increase the reusability of ECU software, the AUTOSAR development partnership has defined a standardized ECU software architecture. AUTOSAR architecture provides a high level of abstraction between the three software layers namely: Application Layer, Runtime Environment Layer and Basic Software Layer which run on a Microcontroller, Figure 2.1 shows the three layers of software [6].

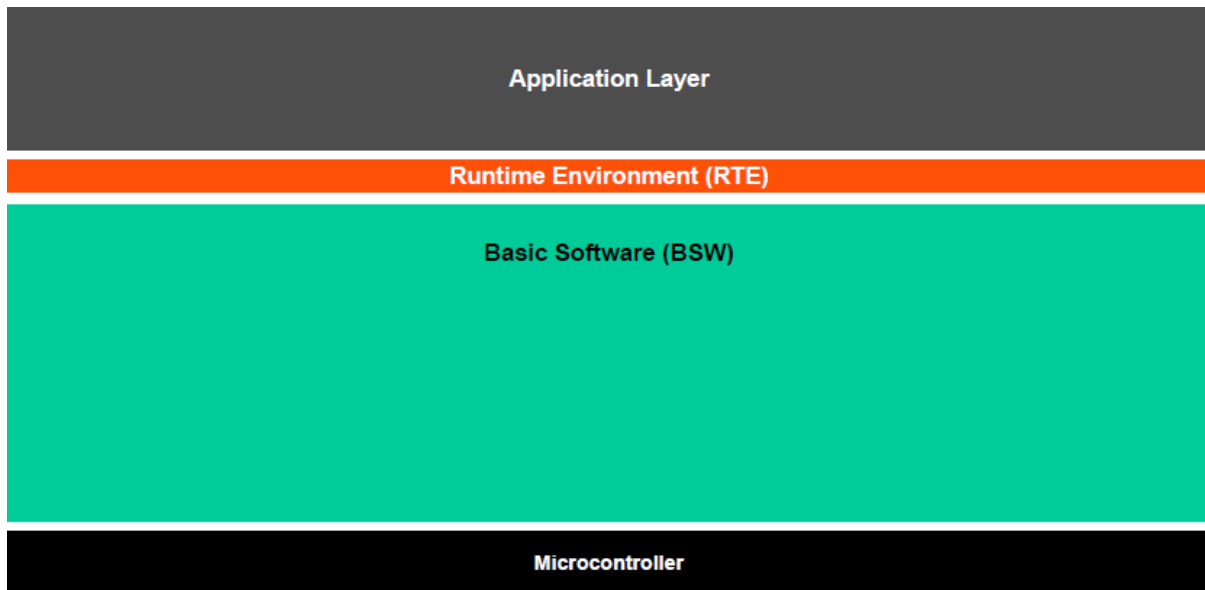


Figure 2.1: AUTOSAR Architecture

2.1.1.1 Application Layer

The top most layer of the AUTOSAR consists of the functional part of the ECU software such as the controller code. It consists of Software components (SWCs) that are mapped to the ECUs of a system. All SWCs together are also called the application layer. SWC is an architectural element that provides and/or requires interfaces and communicate to one another through a Virtual Function Bus (VFB) to fulfil structural requirements. [7]. Each AUTOSAR SWC encapsulates certain part of the behaviour of the application. AUTOSAR does not specify the granularity of SWCs.

2.1.1.1.1 Software Components

SWCs can be divided into the following three categories:

- Composition SWCs
- Atomic SWCs
- Parameter SWCs

Composition SWCs contain other software components and have no functional behaviour of their own. When a component-type (SWC, interfaces) is used within a composition is called a “prototype”. Compositions are generally used to build up hierarchical systems which contains different levels of hierarchies [8].

Atomic SWCs contain an SWC internal behaviour. The internal behaviour defines the functional behaviour of the software architecture. Atomic also means that each instance of an AUTOSAR SWC is statically assigned to one ECU. AUTOSAR defines the following types of atomic software component types:

- Application SWC
- Sensor actuator SWC
- NV block SWC
- Complex device driver SWC
- Service SWC

- ECU abstraction SWC
- Service proxy SWC

Parameter SWCs provide the calibration parameters for the other SWCs.

2.1.1.1.2 Ports

Ports of a component are the communication points between the components. Port of an SWC can be any of the following three types:

- PPort
- RPort
- PRPort

A PPort or a PRPort provides the elements, and an RPort or a PRPort requires the elements defined in a port-interface. Thus, a port can always be typed by only one port-interface [8].

2.1.1.1.3 Port-Interfaces

A port of an SWC is associated with a “port-interface”. The port-interface describes the operations and data elements that the SWC provides and requires. A port interface can be any of the following kinds [8]:

- Client Server Interface
- Sender-Receiver Interface
- Parameter Interface
- Non-Volatile Data Interface
- Trigger Interface
- Mode Switch Interface

2.1.1.1.4 Software component communication

The most common communication patterns between the SWCs in AUTOSAR are Sender-Receiver Communication and Client Server Communication.

In sender-receiver communication, the sender sends the data to one or more receivers. It is an asynchronous distribution of information, where the sender is not blocked and neither expects nor gets a response from the receivers. In sender-receiver communication, the sender sends the information and receiver decides when to use this information. With this type of communication between ports, the sender does not know the identity or the number of receivers. Figure 2.2 shows a sender-receiver communication in Virtual function bus (VFB) view [8].

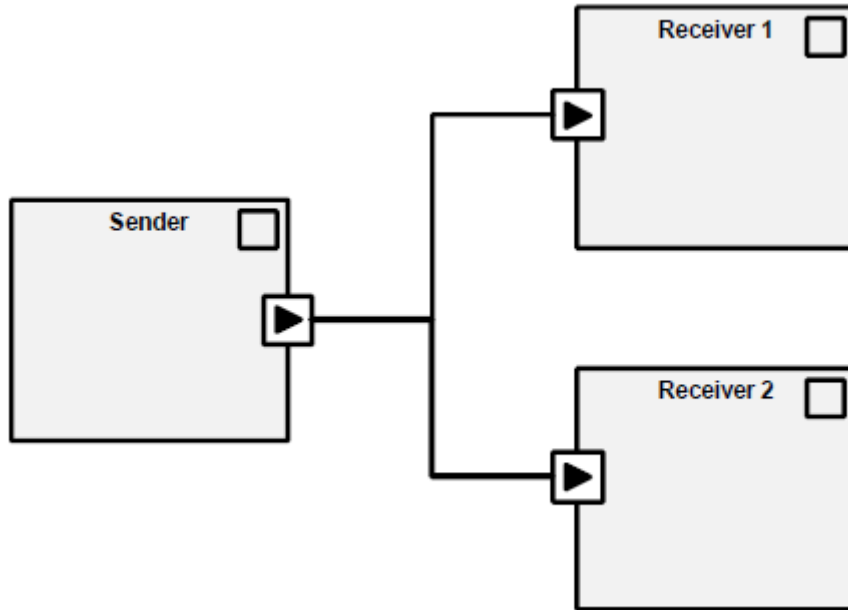


Figure 2.2: Example of a sender-receiver communication in the VFB view

In distributed systems, the most widely used communication pattern is the client-server communication. In client-server communication, the client starts the communication by sending the request to the server to perform a service by giving a parameter set if necessary. The server performs the requested service and transfers a response to the client's request. Figure 2.3 shows the example of a client-server interface [8].

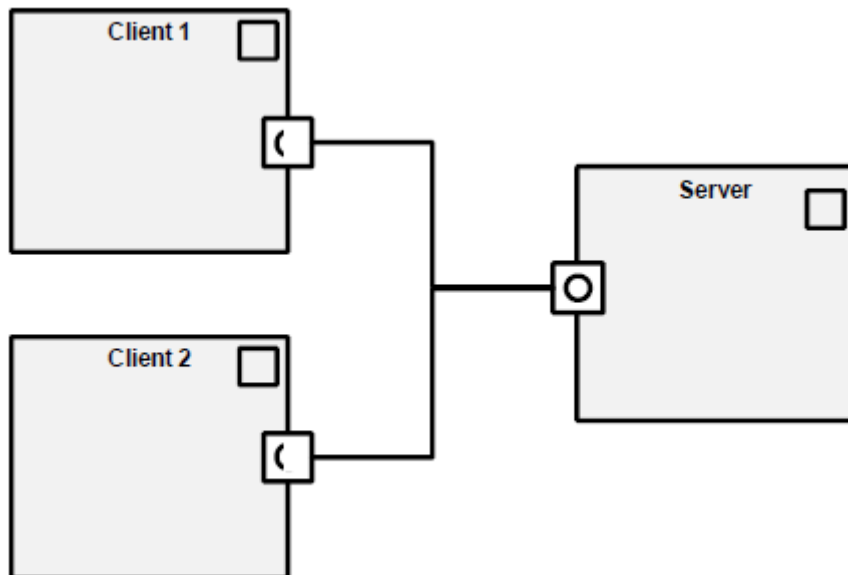


Figure 2.3: Client - server communication in the VFB view

2.1.2 Virtual Function Bus (VFB) and Run Time Environment (RTE)

RTE and VFB are one of the vital parts of the AUTOSAR system design and aids in the portability of SWCs, which is one of the main features of AUTOSAR. In order to attain a high level of transparency with respect to underlying hardware infrastructure, AUTOSAR

introduces VFB and RTE concepts which help in achieving infrastructure independent SW development.

VFB in a general context can be defined as a system modeling and communication mechanism. It gives a virtual framework that is independent of any hardware infrastructure and also gives all services necessary for a virtual interaction between AUTOSAR SWCs. VFB is a SWC interconnection concept that abstracts application development and modeling from the underlying hardware. [9].

Unlike VFB, RTE on the other hand, provides a real implementation of the communication topology and interaction between SWCs. That is to say that, RTE presents an actual depiction of the virtual concepts of the VFB for a specific ECU. A customized RTE is generated for each ECU during the ECU configuration process of AUTOSAR methodology (see section 2.5 AUTOSAR Methodology). The SWCs that are mapped onto the same ECUs communicate through Intra-ECU communication mechanisms while the communication between SWCs mapped on different ECUs can be realised through Inter-ECU communication on the bus system, e.g. CAN, FlexRay, Ethernet etc. [9].

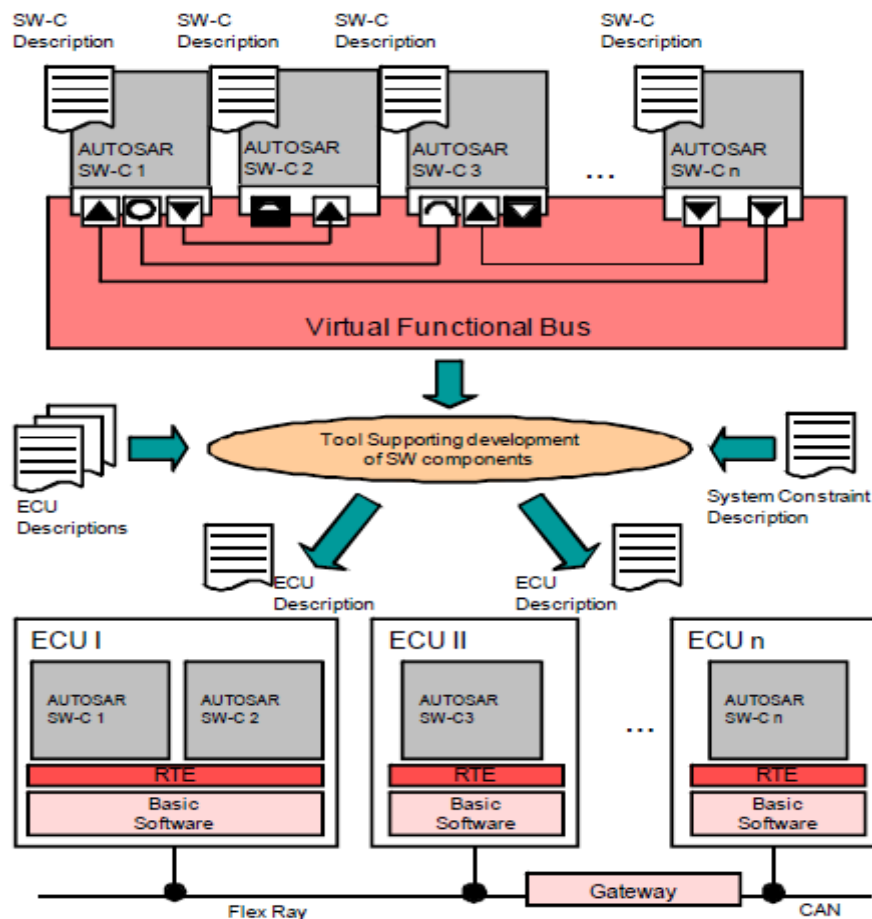


Figure 2.4: Example of VFB to RTE Mapping

Figure 2.4 [8] illustrates how the SWCs that are connected virtually through an VFB are mapped onto different ECUs, each having its own RTE.

2.1.2.1 Interfaces

The Figure 2.5 [10] shows three different categories of the AUTOSAR interfaces. Different interfaces exist between different architectural units of AUTOSAR and the figure below shows these interfaces with respect to RTE layer.

- **AUTOSAR Interface** denotes SWC interfaces that can be described using the concept of VFB ports and communication mechanism. On the application layer, application SWCs and sensor/actuator components use these interfaces above RTE, and the ECUAL and Complex Device Drivers on the layer below RTE.
- **Standardized AUTOSAR Interfaces** are similar to the AUTOSAR interfaces, however, they are standardized such that the interface specifications of the components communicating through such an interface is known in advance.
- **Standardized Interface** indicates other SWC interfaces which are not described using the specification of VFB. The components that are connected to RTE with standard interface are not allowed to communicate with the SWCs directly, but through RTE alone. For example, OS module which provides services such as instantiation of the prototype SWCs or scheduling tasks are done via the RTE.

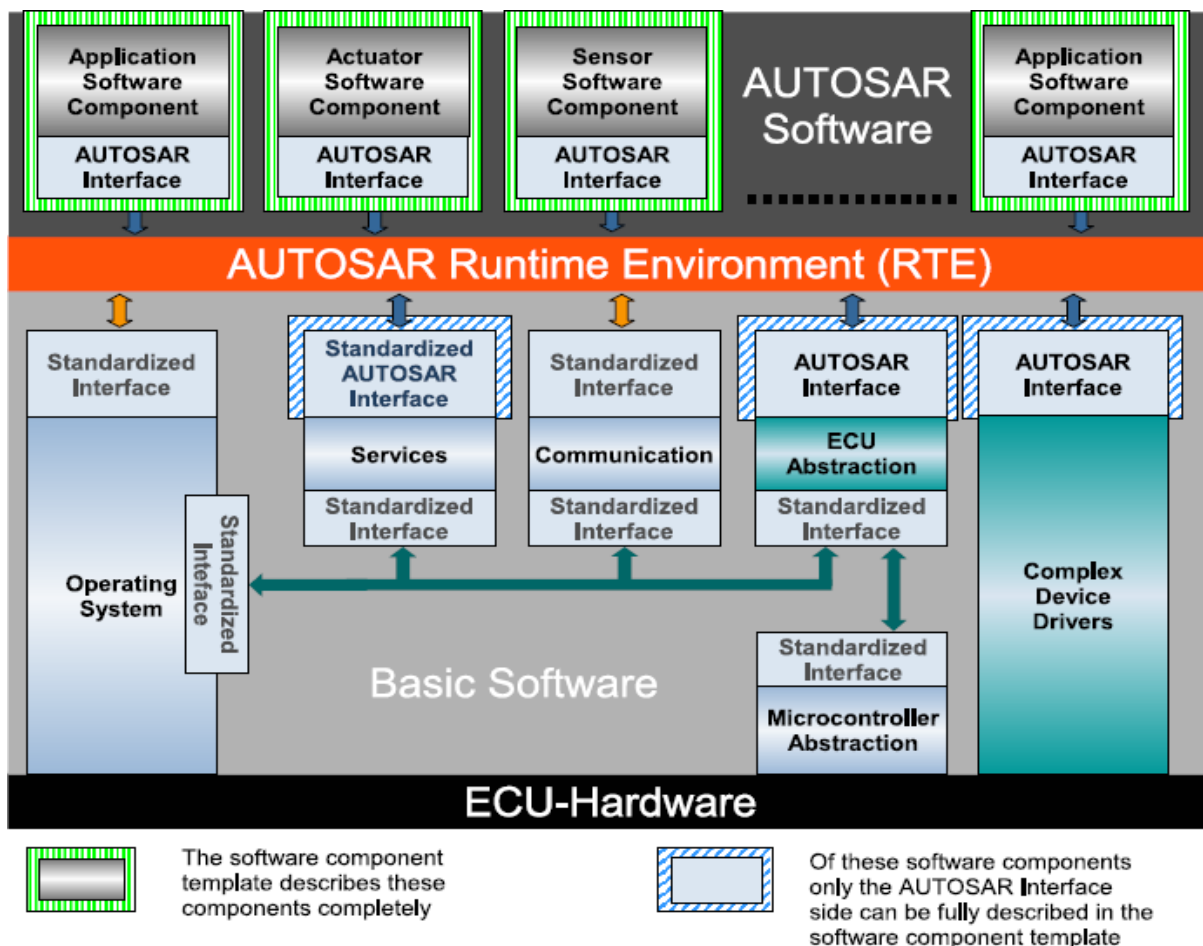


Figure 2.5: Overview of the AUTOSAR Interfaces

2.1.3 Basic Software Layer

AUTOSAR Basic Software (BSW) layer is further divided as follows: Services Layer, ECU abstraction Layer, Microcontroller abstraction Layer and Complex device drivers. Figure 2.6: Basic Software Layer [6] shows this layered structure.

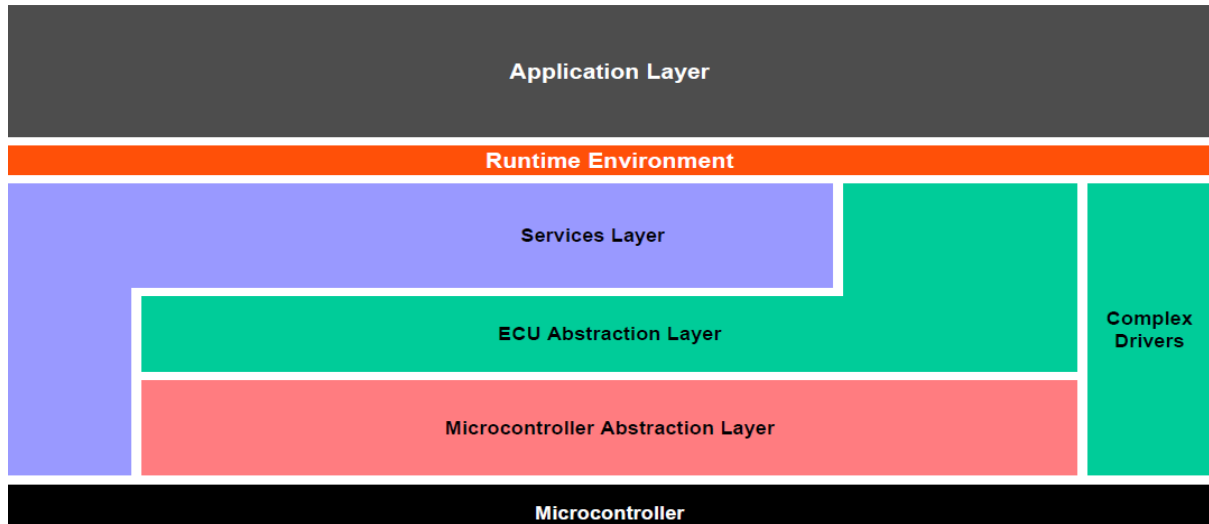


Figure 2.6: Basic Software Layer

Services layer provides basic system services for every AUTOSAR application. These services can be accessed by an AUTOSAR application through standardized AUTOSAR interfaces. Services layer is the top most layer of BSW layer and provides OS functionality, memory services, diagnostic services, vehicle network communication and management services, ECU state and mode management, and logical and temporal program flow monitoring [6].

ECU Abstraction Layer (ECUAL) abstracts higher SW layers independent of the underlying hardware infrastructure. The upper layers with regard to ECUAL need not know any details about the kind of controller on which it is executed. The ECUAL separates the upper layers containing application from the hardware infrastructure by providing a software interface to the electrical values from the ECU. [6].

Microcontroller Abstraction Layer (MCAL) is the bottom most layer of the BSW. MCAL utilizes drivers to separate from particular controllers on the ECU. MCAL comprises of internal drivers, which are BSW modules with direct access to the μ C and internal peripherals. These BSW modules provides interfaces to the ECUAL to activate the general functions of different μ Cs of the same kind [11].

BSW layer is further divided into functional groups as shown in Figure 2.7. Each of the functional groups can have several modules generally referred to as BSW modules. Currently, there are over 80 BSW modules [12].

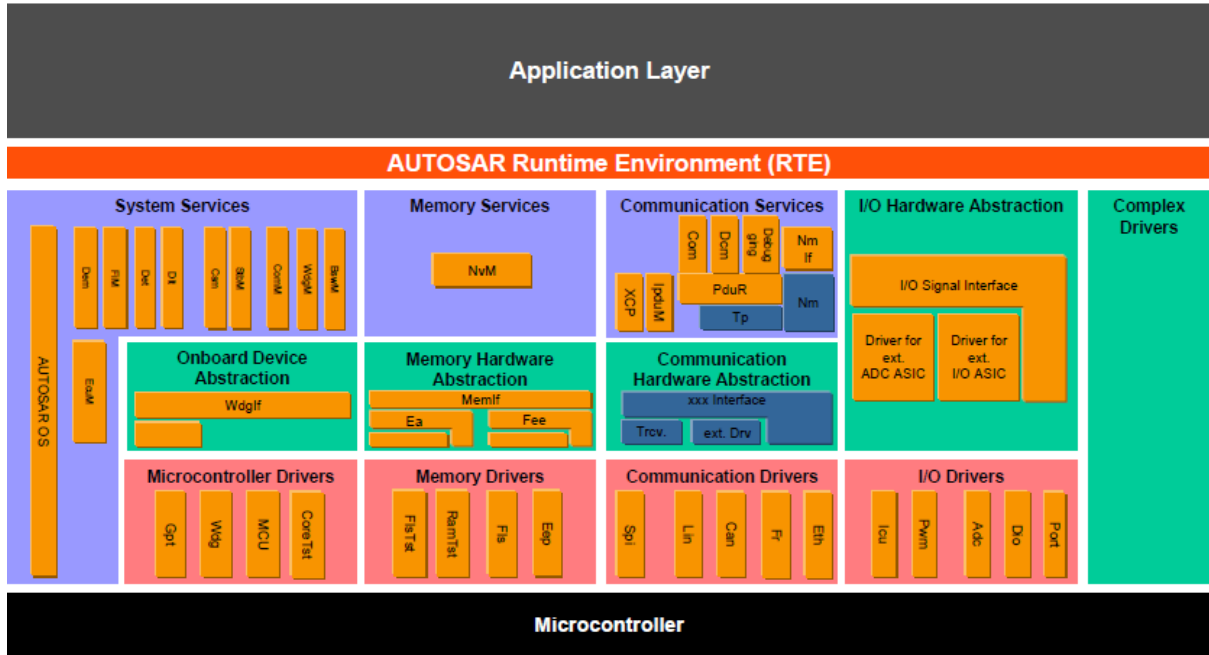


Figure 2.7: Overview of BSW function group and modules

The functional groups that are used in this thesis will be discussed in the following sections.

2.2 AUTOSAR Communication Cluster

AUTOSAR supports different network communication protocols such as CAN, LIN, Ethernet, and FlexRay. This thesis concentrates on CAN communication, as it will be used widely with respect to BMS application. This section briefly introduces CAN communication, and the AUTOSAR Communication Stack with respect to CAN.

2.2.1 Controller Area Network (CAN)

For communication with distributed real-time control with higher requirements on security and integrity, a serial communication protocol known as Controller Area Network is used. Any node on a CAN bus can start transmitting a message when it is free. CAN uses Carrier sense multiple access protocol with collision detection. If more than one node starts transmitting messages at the same instance of time, the bus access conflict is resolved by bitwise arbitration using IDENTIFIER. This arbitration mechanism ensures that neither the data nor time is lost. During arbitration, every transmitting node compares the bit transmitted with the level that is monitored on the CAN bus. As long as these levels are same, the unit will continue to send. When a “dominant” level (logical 0-bit) and “recessive” level (logical 1-bit) is monitored, the node transmitting the recessive level will lose the arbitration and will withdraw immediately [13].

The structure of a CAN frame is as shown in Figure 2.8. Some of the main fields in the frame are, the identifier field is the ID of the CAN message, length is 11 bits in a Standard CAN frame or 29 bits in an Extended CAN format. DLC, Data Length Code indicates the number of bytes in Data Field. The information to be transferred is contained in the Data field, it can contain from 0 to 8 bytes.

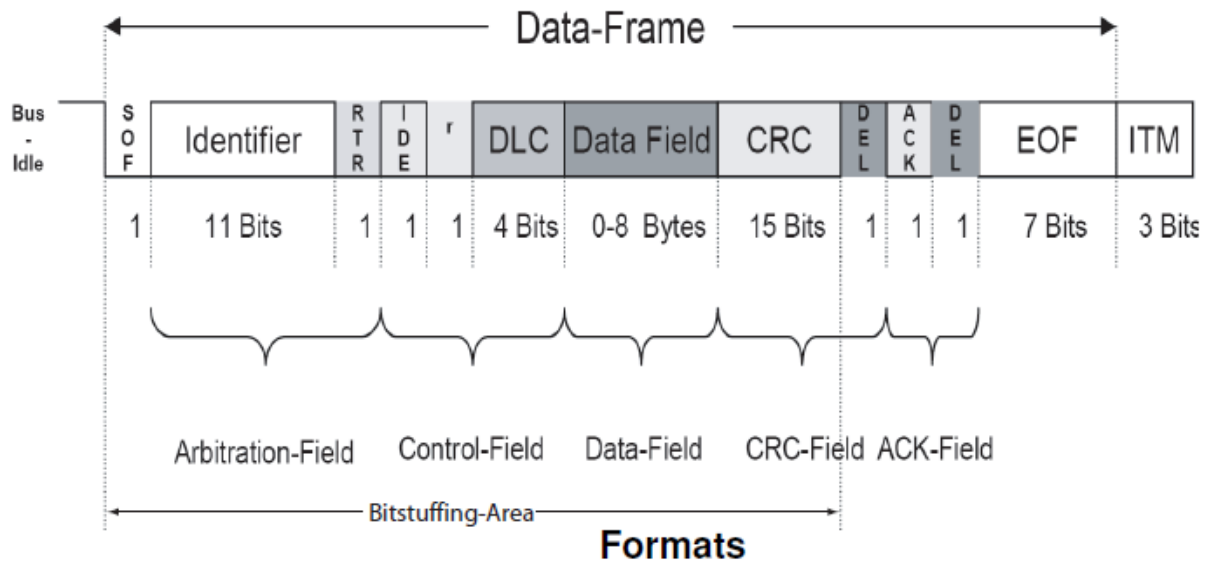


Figure 2.8: CAN Frame

2.2.2 AUTOSAR Communication Stack

Figure 2.9 shows the highlighted BSW function groups that contribute to the communication stack of the AUTOSAR.

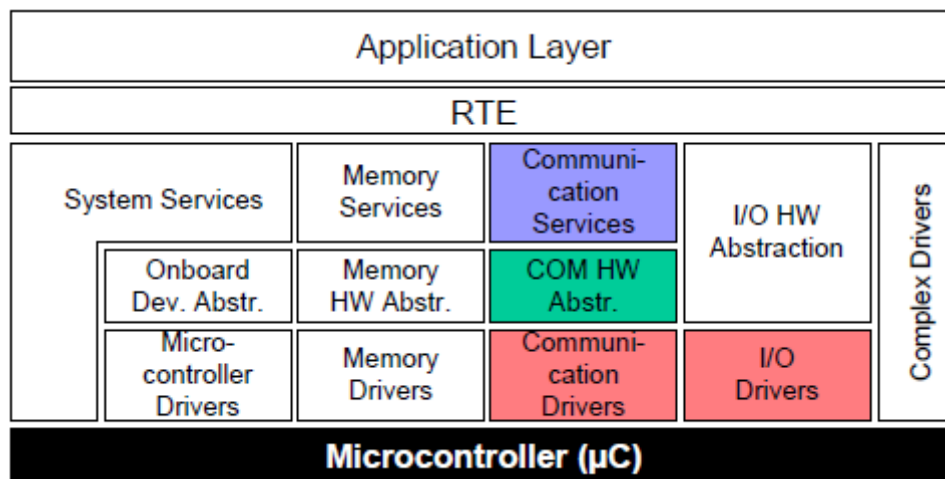


Figure 2.9: Communication Stack related BSW Functional groups

The communication services functional group consists of modules for vehicle network communication (CAN, LIN and FlexRay). An overall AUTOSAR communication stack and some of the modules associated with it is as shown in Figure 2.10 [14]:

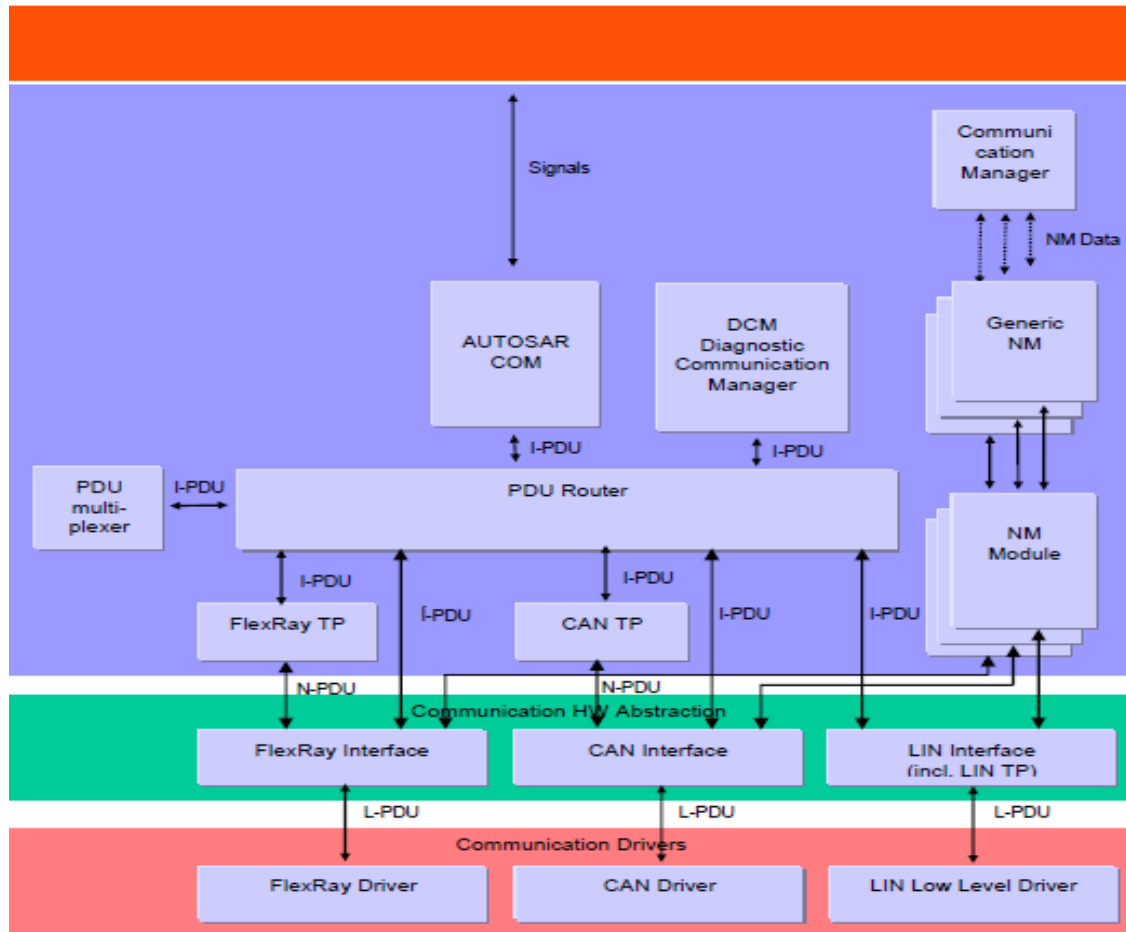


Figure 2.10: AUTOSAR Communication Stack

Since this work uses CAN communication, communication stack with respect to CAN will be discussed further in this section.

The CAN communication services are comprised of modules which provides all the software infrastructure necessary for an uninterrupted interface to the CAN network. The application layer need not know the type of communication protocol used to transfer signals i.e. identifiers, data lengths, bit timing and etc. are all handled by the communication stack. Figure 2.11 shows the CAN communication stack, some of the modules such as AUTOSAR COM, COM Manager and PDU Router are always same irrespective of the network system and these modules exist in every ECU which is on a network bus [6].

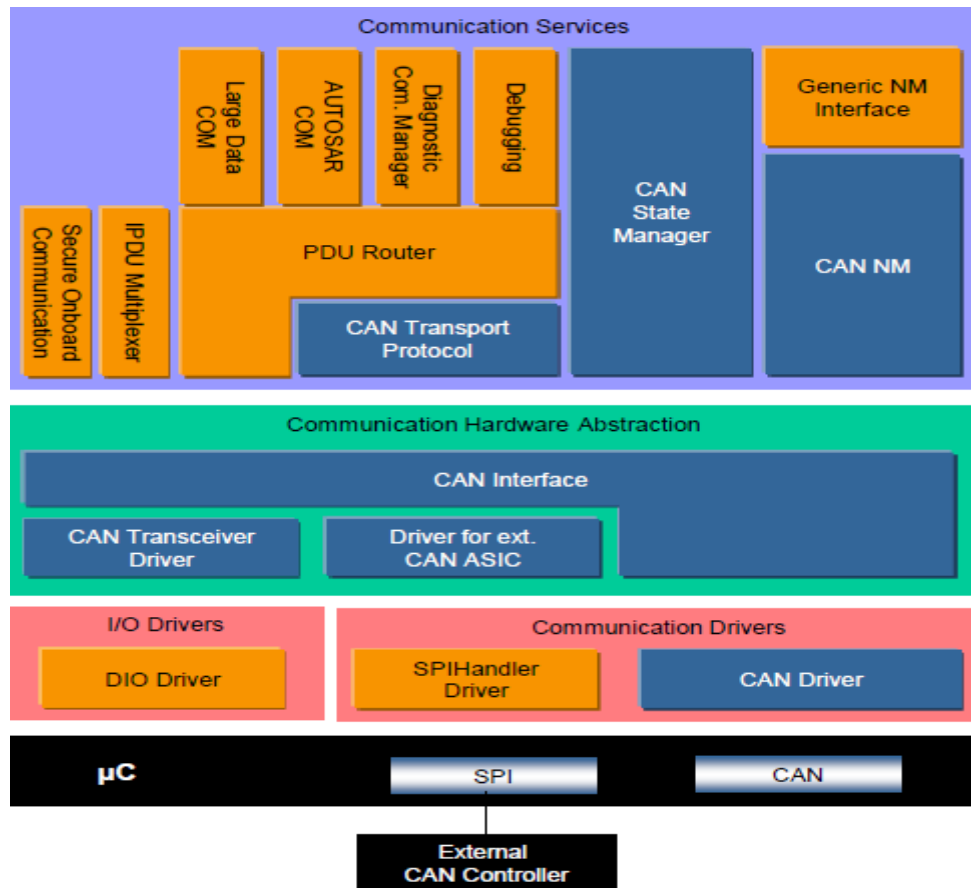


Figure 2.11: CAN Communication Stack

2.3 Basic Software modules

In this section, some of the important BSW modules which will be used in the configuration of CAN communication stack, and other modules from services layer and MCAL layer will be discussed.

2.3.1 PDU Router

Protocol Data Unit (PDU) describes the data of a specific communication protocol. PDU Router routes the PDUs between various abstract com controllers and upper layers.

The PDU router module mainly contains two parts:

- The **PDU Router routing tables**, contains the static routing tables indicating the routing attributes such as IDs, and, source and destination BSW modules for each I-PDU to be routed.
- The **PDU Router Engine** is the one which controls the routing actions according to the PDU router routing tables. The router engine deals with routing the I-PDUs from source to destination and translating the source I-PDU to the destination(e.g. PduR_Transmit to CanIf_Transmit, PduR_CanIfTxConfirmation to Com_TxConfirmation) [14].

Figure 2.12 [14] shows an example of CAN data communication through I-PDUs. The communication starts with COM module calling the PduR_ComTransmit() function, the PduR module will call CanIf_Transmit() with the destination ID as argument. Once the data is received at the CAN receiver, it sends an acknowledgement to CAN interface, which in turn calls PduR CanIfTxConfirmation() and then PduR will call Com TxConfirmation(). All these function calls will take the I-PDU IDs as an argument which is pre-configured in the PDU router.

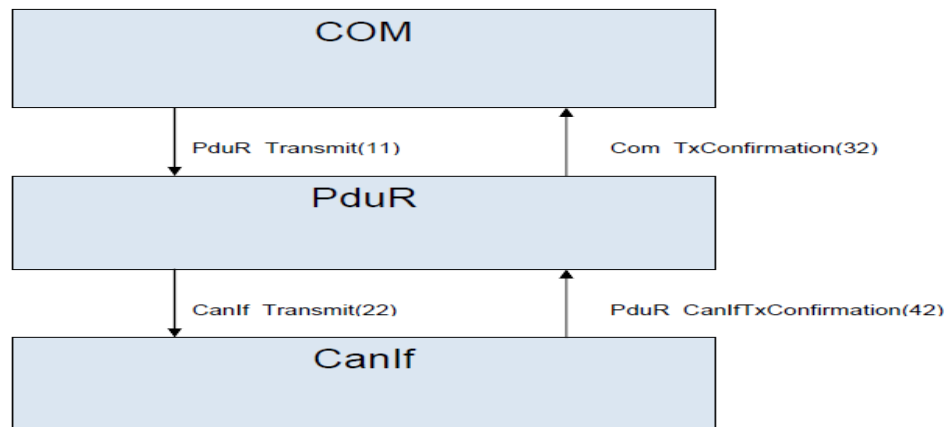


Figure 2.12: I-PDU ID example

2.3.1.1 PDUs

A PDU contains Service Data Unit (SDU) and Protocol control information (PCI). The PDUs are identified by a unique static ID which is assigned to each PDU [14]. When a PDU is sent to lower layers, the lower layer considers the received PDU as an SDU of its own PDU. This is shown in Figure 2.13 [6]. Non-Transport Protocol I-PDUs should not be more than 8 bytes. This is because of the fact that the Data field of CAN is 8 bytes long [15] [16].

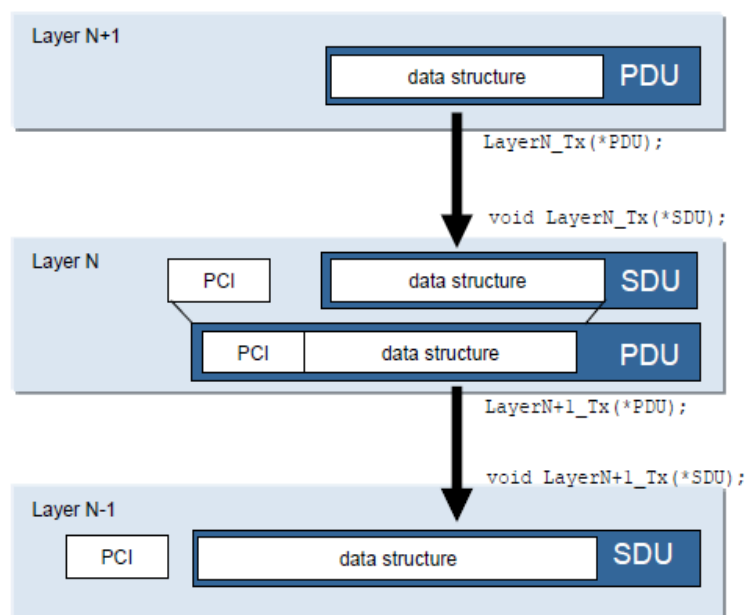


Figure 2.13: PDU over different Layers

With respect to CAN communication, Layer N in Figure 2.13 corresponds to TP and Layer N-1 to CAN IF.

The SDU contains the data sent by upper layer, it also contains the request to send this data to the next layer. The PCI is important for passing SDU from one type of protocol layer to another instance. For example, it contains the source and destination information. The protocol layer attaches the PCI at transmitting node and is removed at the receiving node. This PDU is considered as its SDU by the transmitting node, where PDU passes from the upper layer to the lower layer [6].

To differentiate PDUs at different layers of the software architecture, different prefix is given at each layer. Depending on the layer a PDU can be either I-PDU, L-PDU, or N-PDU. Figure 2.14 [6] shows different kinds of PDUs and the modules with which they interact.

ISO Layer	Layer Prefix	AUTOSAR Modules	PDU Name
Layer 6: Presentation (Interaction)	I	COM, DCM	I-PDU
	I	PDU router, PDU multiplexer	I-PDU
Layer 3: Network Layer	N	TP Layer	N-PDU
Layer 2: Data Link Layer	L	Driver, Interface	L-PDU

Figure 2.14: Interaction of Layers

2.3.2 AUTOSAR COM

The COM module is located between RTE and the PDU router. Main features of COM module are that it provides the service of signal aligned data interface to the RTE and it controls (Start/Stop) the communication of I-PDU groups. It sends the signal similar to the signals transmission type specified in VFB specification. One of the most important features is the endianness conversion of all integer types along with sign extension [16]. There are two ways of configuring the signal indication modes, after an IPDU is received and has been unpacked [15]:

There are two ways of configuring the signal indication modes, after an IPDU is received and has been unpacked:

- Immediate: “Com Rx Indication” performs the signal indication and confirmation.
- Deferred: In case of cyclic tasks for instance, the signal indication and confirmation are deferred.

2.3.3 COM Manager (ComM)

ComM module performs resource management and it provides the services to control the communication on the hardware infrastructure. The ComM module receives and coordinates the bus communication access requests from the communication requestors. The purpose ComM module is to facilitate the usage of the bus communication stack for the user, Coordinating the availability of the bus communication stack for multiple independent SWCs on the same ECU, Controlling different communication bus channels of an ECU by implementing a channel state machine for every channel (e.g. CanSM for CAN), and allocates necessary resources for the requested communication mode.

The ComM provides three different communication modes. They are COMM FULL COMMUNICATION, COMM SILENT COMMUNICATION, and COMM NO COMMUNICATION. A particular user can request the ComM module for the Communication mode. The highest communication mode is COMM FULL COMMUNICATION; in which the ComM module allows the transmission and reception on the physical channel. The lowest communication mode is COMM NO COMMUNICATION; in which the ComM module prevents the transmission and reception on the physical channel. The communication mode COMM SILENT COMMUNICATION is used only for network synchronization [32].

2.3.4 CAN Interface (CanIf)

CAN Interface module is placed between the low-level CAN device drivers and the upper communication services layers (i.e. CAN State Manager, CAN Network Protocol, CAN Transport Protocol, PDU Router). CanIf manages different CAN controllers and transceivers which are on the underlying ECU, by providing a unique interface. CanIf initializes the CAN Driver module during the start-up phase. The CAN interface module also provides main control flow and data flow requirements of the PDU Router and upper layer communication modules of the AUTOSAR COM stack. An abstraction is provided by CanIf to the CAN driver and transceiver driver services, to supervise and control the CAN bus [17].

2.3.5 CAN Driver

CAN Driver is part of the lowest layer of the communication stack, performs the hardware access and offers a hardware independent interface to the upper layer. The only upper layer module to which CAN driver has access to; is the CanIf module. Services for initiating transmissions and calling the callback functions of the CanIf module for notifying events, independently from the hardware is done by CAN driver. CAN driver also provides services to control the behaviour and state of the CAN controller that are on the same CAN hardware unit. A single CAN module can control several CAN controllers as long as they belong to the same CAN hardware unit [18].

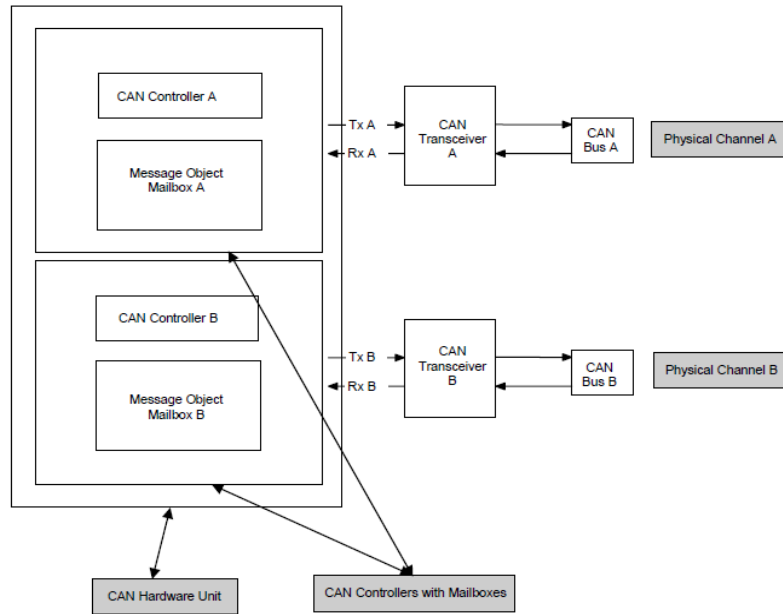


Figure 2.15: CAN Hardware unit with two CAN controllers

Figure 2.15 shows CAN Hardware unit. A CAN driver represents a CAN Hardware, each driver can have one or more than one CAN controllers of the same kind, and one or more RAM areas. The CAN hardware unit is either on-chip or on an external device. Every CAN controller serves exactly one physical channel [18].

The CAN driver stores LPDU inside the buffer in the CAN controller when an LPDU is sent by a node. When an LPDU is received, a receive indication call-back function is called by CAN-driver module along with the ID, Data Length Code and pointer to the LSDU. The CAN-driver module has direct access to the hardware resources and translates the provided data into a format that the hardware understands and triggers the transmission.

2.3.6 CAN State Manager

Each ECU can have different communication networks, each of these networks are identified by a unique network handle, which is assigned during the configuration of ComM module. The ComM module requests communication modes from the networks and in case of CAN, it uses CanSM module.

CAN State Manager (CanSM) is a member of communication service layer and it implements the control for CAN network. CanSM interacts with the Communication hardware abstraction layer i.e. CanIf module and System service layer. CanSM module changes the communication modes of the configured CAN network based on the mode requests from the ComM module. CanIf module notifies CanSM module whenever there are any changes in the CAN Controller modes and CAN Transceiver mode, the CanSM module then notifies ComM and BswM [31].

2.3.7 Operating System

AUTOSAR OS module is located in the system services layer of BSW. AUTOSAR OS is based on the OSEK/VDX (Open systems and corresponding interfaces for automotive electronics/ Vehicle Distributed eXecutive), which is an open automotive standard.

OS has two kinds of tasks namely basic task and extended task. Basic tasks release the processor only if they terminate, if OS switches to a higher priority task or if the processor switches to an interrupt service routine (ISR) caused by an interrupt. Extended tasks, unlike basic tasks are permitted to use the wait event, which results in a waiting state. When a task enters waiting state, the processor is released and the processor is now reassigned to a task with lower priority without terminating the currently running extended task. The task models of basic and extended tasks are shown in Figure 2.16 [19].

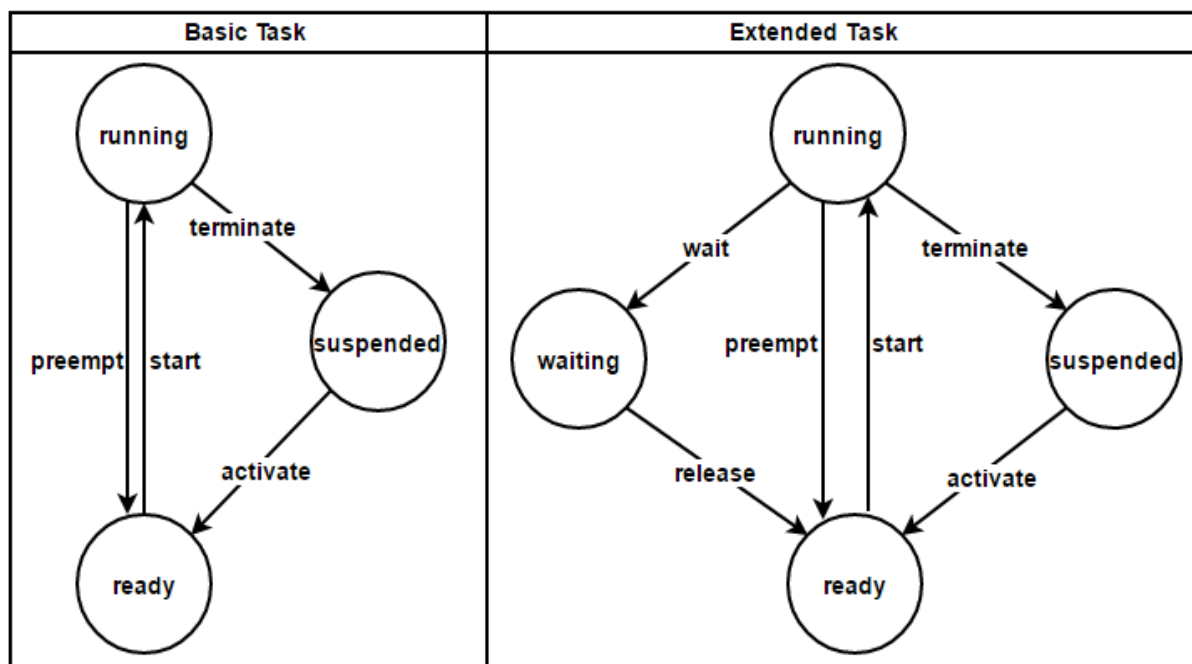


Figure 2.16: Task model: Basic and Extended Tasks

OSEK/VDX scheduling policy has been extended in the scheduling policy used for AUTOSAR. Highest Priority First (HPF) Scheduling is used in AUTOSAR OS, if more than one task share the same priority then the priority is based on FIFO basis. Tasks sharing same priority are considered as a group. Normally, a task of the group automatically locates an internal resource when it gets the processor and releases when terminated, then waits for an event or it invokes schedule service. If a task is not in a group, general pre-emption rules are applicable according to their priority levels. A task cannot pre-empt another task in the same group. When two tasks accesses the same shared resources, AUTOSAR co-ordinates such concurrent access with OSEK- Priority ceiling protocol. Each resource has its own priority, when a task gets a resource, the tasks priority is changed to the resource priority. Hence the tasks sharing the same resource cannot get the processor.

The OSEK timing services such as alarms and counters are also extended in AUTOSAR with the introduction of concept of schedule tables. The counting of "ticks" from the underlying hardware is done by an object known as Counter. Each counter is reset to 0 when it reaches its

maximum value. An alarm in AUTOSAR OS is linked to a counter and a task. The alarm expires when the counter reaches a predefined value. As a result of this an action which is statically defined is performed i.e. either a task associated with the alarm is activated or an event related to the task is set. Schedule tables are an extension of the alarm concept, where they are also linked to a counter but is comprised of a set of expiry points. When the counter reaches the expiry point, one or more actions are taken [20].

2.3.8 ECU Manager

ECU Manager (EcuM) module manages common aspects of ECU states. Specifically, EcuM module initializes and de-initializes the OS, the Schedule Manager and the Basic Software Module as well as some basic software driver modules. When requested, EcuM configures the ECU for SLEEP and SHUTDOWN. It also manages all wakeup events on the ECU. Wakeup validation is used by EcuM to distinguish between ‘real’ and ‘eratic’ wakeup events [21].

There are two variants of AUTOSAR ECU management since release 4.0: flexible and fixed. Flexible ECU management is more powerful than the fixed ECU management. In this thesis, Fixed ECU management will be used. Detailed specification of flexible EcuM can be found in [21].

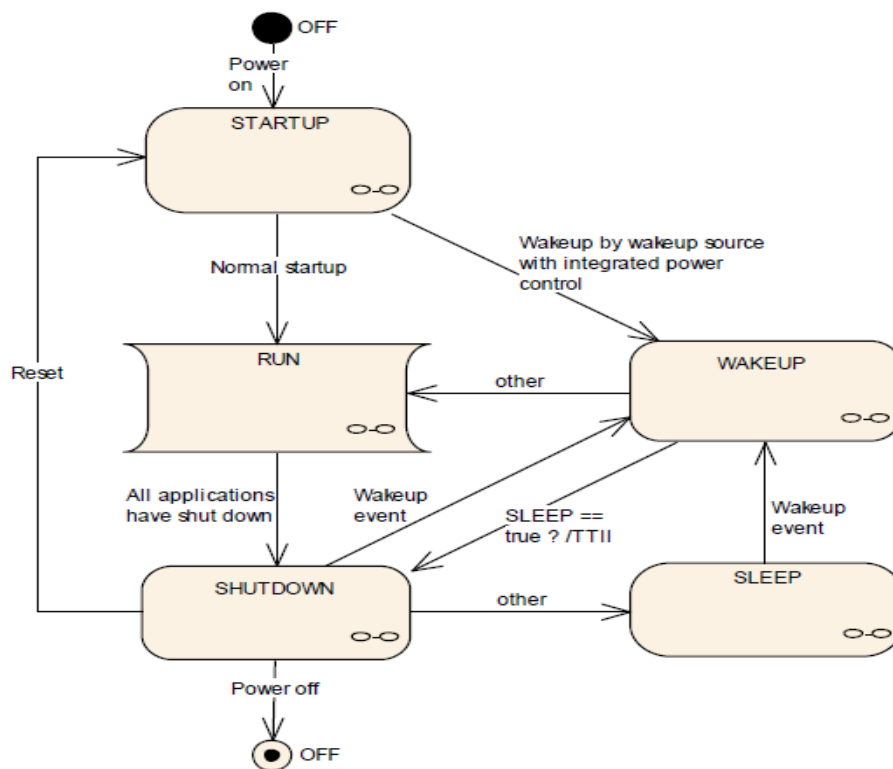


Figure 2.17: ECU Main states

Figure 2.17 [22], ECU main states shows the main state machine provided by the ECU State Manager Fixed module. This state machine manages the ‘life cycle’ of an ECU from OFF through STARTUP and RUN to SLEEP or OFF.

STARTUP State initializes the BSW modules. The STARTUP state has two parts. The first part is finished when the OS is started and second when RTE is started. The reason for splitting

it into two parts is to distinguish services called before the OS is started from those called afterwards and to have a clear visualization.

The **RUN state** is entered after all the BSW modules including OS and RTE have been initialized by ECU state by the ECU state manager fixed module. The RUN state indicates to the SWCs above the RTE that BSW has been initialized and applications start operating. RUN state also provides the mechanism for the synchronous shutdown of the application software.

The **SHUTDOWN state** provides the controlled shutdown of BSW modules and finally results in the selected shutdown target for the ECU: SLEEP, OFF or Reset.

The **SLEEP state** is an energy saving state. No code is executed in this state but the power is still supplied, and if configured accordingly, the ECU is wake-able in this state. SLEEP state can be configured a few sleep modes which typically are a trade-off between power consumption and time to restart the ECU.

The **WAKEUP State** is entered when the ECU comes out of the SLEEP state, due to intended or unintended wakeup. In order to distinguish between the real and erratic wakeups protocol is provided to support the validation.

The **OFF state** is the unpowered ECU; however, the ECU must be start-able (e.g., by reset events) [22].

2.3.8 Microcontroller Unit (MCU) driver

The MCU driver BSW module accesses the microcontroller (μ C) hardware directly and is located in the MCAL. MCU services for Clock and RAM initialization are done by the MCU driver. MCU driver provides services to enable and set the MCU clock, software triggering for a hardware reset, activate MCU reduced power modes [33].

2.3.9 Port Driver

The Port driver provides the services for initializing the complete port structure of μ C. Different functionalities like General purpose I/O, ADC, SPI, SCI, PWM, CAN, LIN etc. can be assigned to ports and port pins. The configuration and mode of these port pins are microcontroller and ECU dependent [34].

2.4 Universal Measurement and Calibration Protocol (XCP)

ASAM MCD-1 XCP is a standard defined by Association for Standardization of Automation and Measuring Systems (ASAM), the initial version of XCP was developed in 2003. It was designed for using in automotive industries in the areas of ECU development i.e. in calibration, and testing. XCP is based on the ASAM standard CAN Calibration Protocol (CCP). CCP had a few disadvantages like timestamping for measurement data was not available and some features, for example, flash reprogramming was not specified in detail [23].

XCP is a master slave communication protocol between ECUs and calibration systems which is independent of the underlying bus system. XCP is used to change the parameters and measure the values of the internal variables of a controller. "X" in XCP indicates that various

transport layer can be used with XCP. XCP standard can be seen as two parts, i.e a base standard and a transport layer. A base standard defines the memory oriented services which are independent of the underlying bus systems. The transport layer can be any of the following communication protocol like CAN, FlexRay, Ethernet, SPI, SCI, and USB. Main objectives of XCP are:

- To scale down the requirements on XCP slave resources, like processor load, flash memory or code memory, and RAM utilization.
- To attain high data transfer rates over the established communication link between XCP slave and master.

XCP finds its application in different phases of ECU development such as function development, ECU testing, and ECU calibration. The ability of XCP to attain high data transfer rates and faster measurement cycle times in the order of micro-seconds, makes this protocol to be used in studying the behaviour of dynamic systems such as electro-chemical systems with respect to automotive use cases.

The main functionality of XCP is to provide read and write access to certain memory locations of ECU. The memory is accessed in an address oriented way, which means, the communication between master (measurement and calibration tool) and slave (the ECU) references address in a memory. Read access to the memory activates measurement of the variables and parameters from the RAM. The write access activates the calibration of parameters in the RAM. The measurement mechanism using XCP happens in a synchronous way, i.e. each measurement is synchronous to an event in the controller [24]. This helps realise the real time behaviour of the system. The measurement of a variable starts with the request of master to slave (i.e. Get value of memory location 0xABCD). A parameter is also calibrated with a request to slave by master (i.e. Set the value of memory location 0x4567 to “x”).

2.4.1 Communication Model

XCP data is exchanged in a message-oriented way between the master and the slave. The XCP packet is contained in a message frame of the transport layer. The frame consists of 3 parts:

- XCP header
- XCP packet
- XCP tail

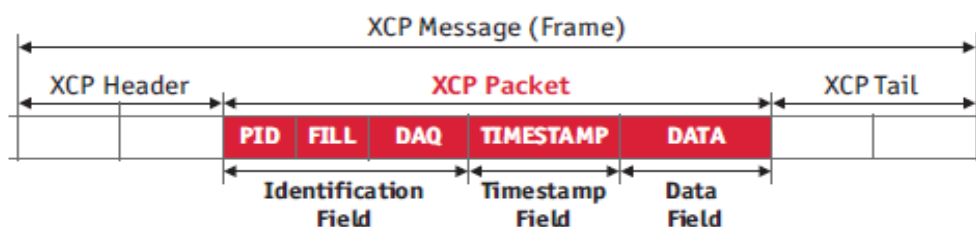


Figure 2.18: XCP Packet

Figure 2.18 [25] shows the XCP message containing the XCP packet. XCP header and XCP tail length varies depending on the transport layer. Whereas, XCP packet is not dependent on the underlying transport protocol. XCP Packet consists of three fields, namely: “Identification

Field” always beginning with the Packet Identifier (PID), “Timestamp Field” and “Data field” containing the payload.

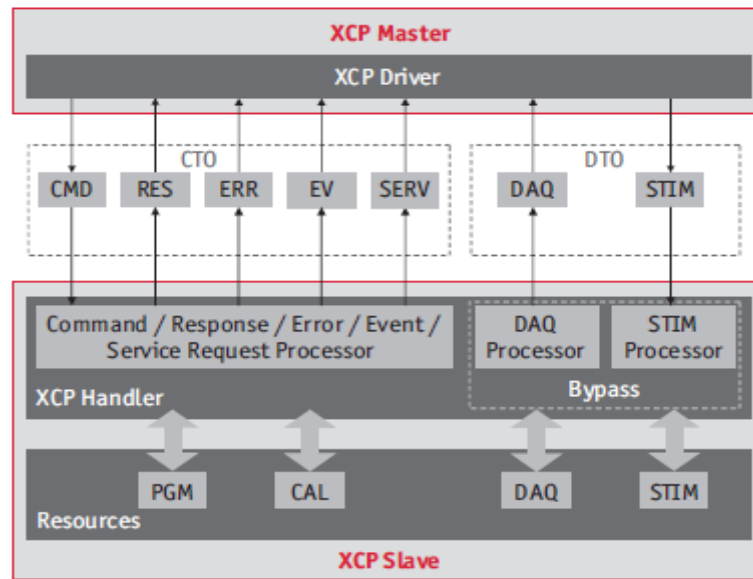


Figure 2.19: XCP Communication Model with CTO/DTO

The Figure 2.19 [25] shows the communication model of XCP with CTO/DTO, it can be seen that the communication through XCP packet is divided into one region for commands (CTO) and one region for sending synchronous data (DTO). The acronyms used in Figure 2.19 stand for:

Command	Packet	Description
CMD	Command Packet	Sends Command
RES	Command Response Packet	Positive response
ERR	Error	Negative response
EV	Event Packet	Asynchronous event
SERV	Service Request Packet	Service request
DAQ	Data AcQuisition	Send periodically measured values
STIM	Stimulation	Periodic stimulation of the slave

The commands between master and slave are exchanged through CTOs. If, the master initiates the contact to slave by sending a CMD, the slave should respond to CMD with RES or ERR. The EV and SERV CTO messages are sent synchronously. To achieve synchronous measurement and stimulation of the data DTOs are used.

The Figure 2.20 [26] shows the structure of a CTO packet.

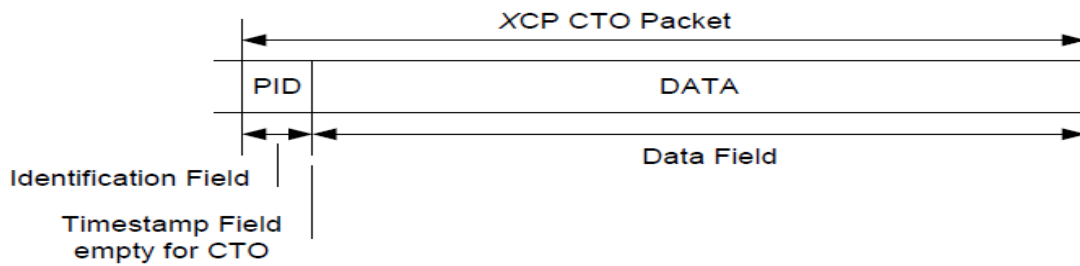


Figure 2.20: The CTO packet

A request to slave is sent by master over CMD. The identification number of CMD is contained in PID field. Parameters specific to different types of CTO packets are sent in the data field. The slave sends a reaction to the master as ERR or RES.

The Figure 2.21 [26] shows the structure of a CTO packet.

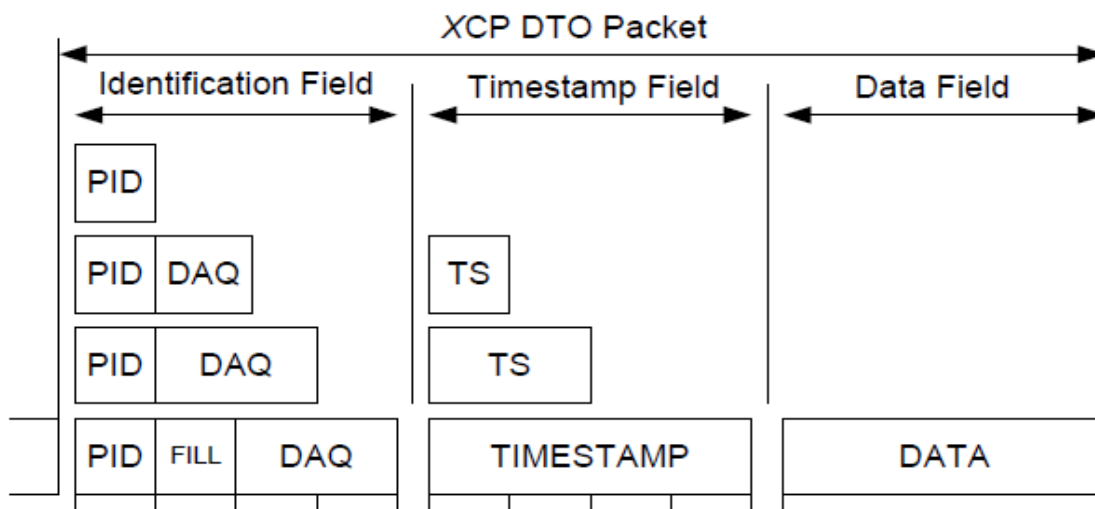


Figure 2.21: The DTO Packet

As indicated in Figure 2.19 [26], the DTOs are used for exchanging data synchronously for measurement and calibration. The Data field of the DTOs contains the data for synchronous acquisition and stimulation.

Master receives the data from the slave by DAQ – synchronous to internal events. Data acquisition happens in two phases: Initially, the master informs the slave regarding the data that slave should send on different events. Then, the master commences the measurement in the slave and the actual measurement phase starts. Now, the desired data is sent to the master by slave and this continues only until master sends a “measurement stop” to the slave.

STIM is used by the master to send data to the slave. This communication also happens in two phases: Initially, master informs to the slave about the data that will be sent. Then, the master sends the data and the STIM processor saves the data. When a STIM event related to a particular data is triggered in the slave, the data is written in to the code memory.

XCP allows different modes for transferring command and reaction between master and slave, namely: Standard, Block and Interleaved mode. The three modes of communication are depicted in Figure 2.22 [25].

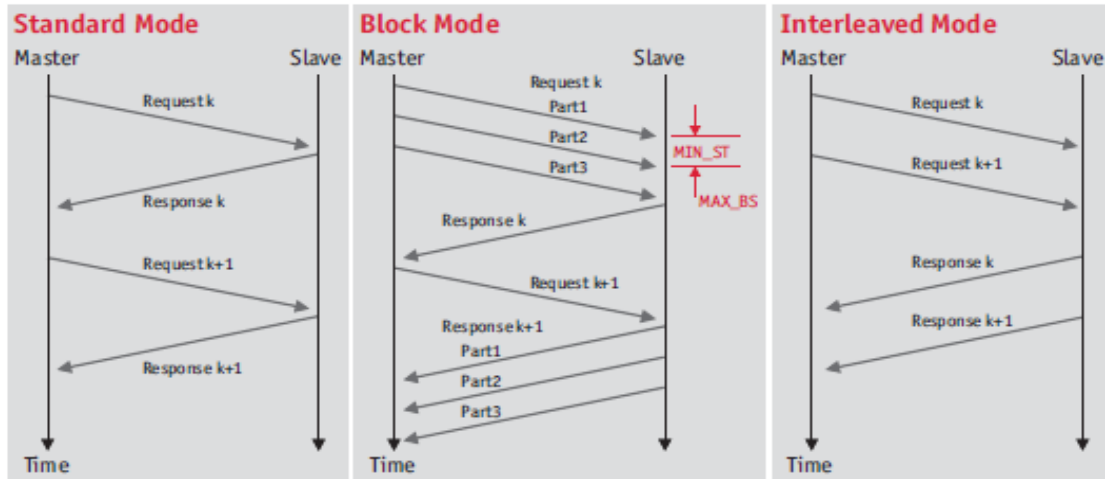


Figure 2.22: Modes of XCP protocol

In the standard communication model, a slave sends a response to every request by a master. This is the classic communication mode. To save time during large data transfer, an optional mode, i.e. Block transfer mode, is used. However, the performance issue should be considered in the slave (ECU). Thus, the least time between two commands (MIN_ST) should be controlled and the number of commands should be with the maximum limit (MAX_BS). Another mode which XCP supports is interleaved mode which is also an optional mode and is used for performance reasons [25].

2.4.2 XCP Transport Layer

The XCP protocol was designed to be transport layer independent so that it can support various types of transport protocol. XCP support transport layers like CAN, FlexRay, Ethernet, SxI and USB [25]. In the following section, XCP over CAN will be discussed.

2.4.2.1 XCP over CAN

XCP is the predecessor of CCP and supports all the requirements for CAN bus. All data that are processed in a networked CAN bus system, as well as their interrelationships, are commonly administered in a central communication database/communication matrix. Most commonly used communication matrix format is DBC format, AUTOSAR also has an ARXML format of this database.

The XCP CAN frames are not entered in the can database, however, there is a link between the CAN database and XCP. In order to use less CAN frames XCP restricts usage of only two CAN IDs that are not used in the database for normal communication is used for XCP communication. One ID is used to send the data/command from master to slave and another ID to send the data/response from slave to master.

The XCP packet (DTO/CTO) size is limited to 8 bytes for XCP on CAN, because the maximum length of data field is 8 bytes on a CAN frame. For XCP, the useful information needed to be exchanged is the command used or the response sent. This information can be sent in the first byte of data field in a CAN frame and the remaining seven bytes can be used to exchange the useful data [25]. The Figure 2.23 depicts the XCP on CAN message.

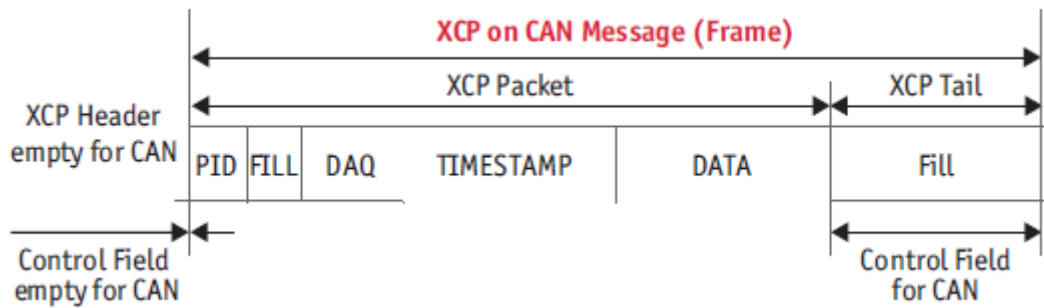


Figure 2.23: XCP on CAN Message

2.4.3 Online Calibration

The most common approach to change the parameter during runtime, i.e. online calibration, is to have the parameters in the available RAM location. Using flash memory for changing the parameters through online calibration is not a practical solution, due to the reason that the flash memory is always organised in the form of large blocks or sectors, which can be erased and written only in whole. Flashing such large sector of memory in-order to change a single parameter is not a practical solution because of the limited resources that are normally available in an ECU [25].

The logical memory layout of the ECU is described by objects called memory segments. Memory segments have attributes which describe the content and the type of access to the parameters, for e.g. DATA+RAM or CODE+FLASH.

XCP introduces the concept of memory paging, which makes the implemented address translations accessible for the XCP page switching service commands. If these XCP services are available, the calibration tool is able to control the active page. If the calibration tool switches a memory segment from a Flash page to a RAM page, a parameter in this memory segment can be modified during the runtime of an application, also known as online calibration [24]. Every segment in the memory has an ECU active page and an XCP active page. The currently active page is the memory area which the XCP master can read and write to. If the ECU allows XCP to point a same memory location as it is pointing, the changes are reflected directly.

2.4.4 DAQ Lists- Data acquisition lists

DAQ measurement is one of the main features of XCP, this method helps in high data rate transfer in minimum time and with low bus load. XCP achieves fast data transfer by linking the acquisition of measured values to the events in the ECU. The bus load on XCP is less as the measurement process happens in two phases, i.e. during the configuration phase, the slave receives the information about the list of values that master is interested in, and the next phase involves only sending the measured values from the slave to the master.

When the user selects certain signals which he wishes to measure, it is not necessary that each signal uses the complete data field of the message, the message packets consist of combination of signals from the slave. This combination of the signal into message packets are not decided

independently by the slave, or else the master cannot interpret the data when it receives the messages. Thus, the master sends an instruction describing how the slave must arrange the signal values in the message.

To describe how the arrangement of bytes in to the message has to be done by the slave, Object Description Tables (ODTs) are used. A DAQ list consists of several such ODTs, which in turn consists of several ODT entries as shown in Figure 2.24 [25]. As seen in Figure 2.25 [25], each entry in the ODT list refers to a memory location in the RAM by the address and length of the object. Each DAQ list is assigned to an ECU event.

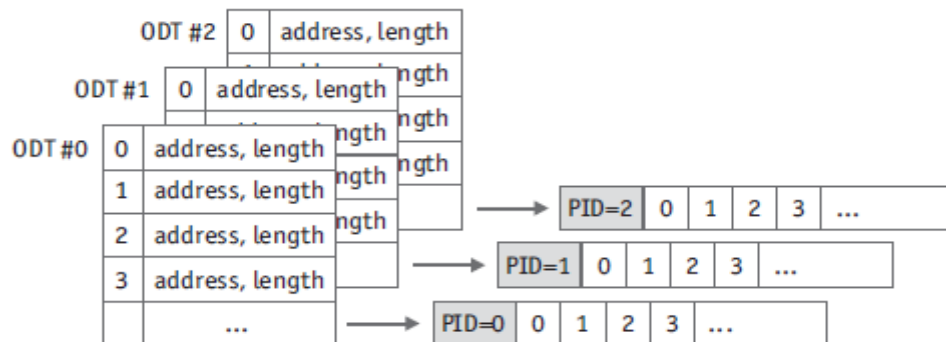


Figure 2.24: DAQ list with 3 ODTs

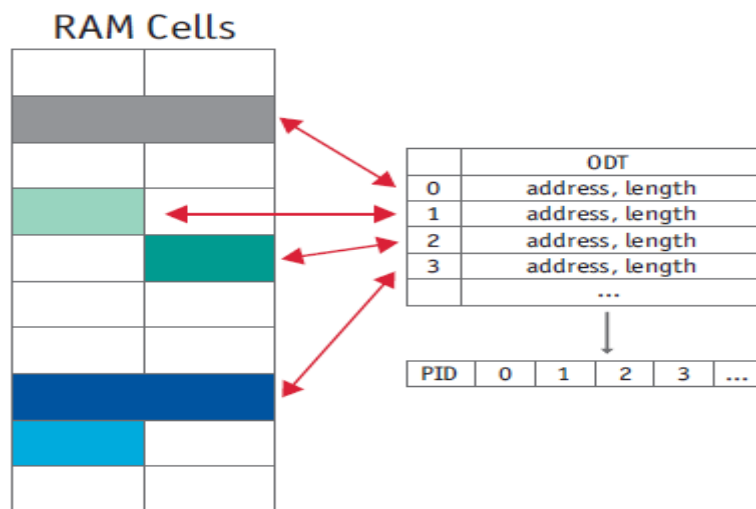


Figure 2.25: ODT: Allotment of RAM addresses to DAQ-DTO

ODT describes the allotment of contents of RAM from the slave to arrange in the message that will be transmitted on the bus as a DAQ DTO.

There are three types of DAQ lists: Static, Predefined and Dynamic.

In **Static DAQ lists**, though there is no information on definition of the measurement parameters in the ODT list, both the DAQ lists and ODT lists are defined permanently. These definitions are generally set in the A2L file and the ECU code in case of Static DAQ lists.

Predefined DAQ lists as the name indicate, in this type the DAQ lists and ODT tables are defined permanently along with the measurement parameters. This method lacks in providing the flexibility to the user, as a result, this type of DAQ lists are not used practically.

In **Dynamic DAQ lists**, the measurement parameter of the DAQ and ODT lists are not pre-defined, but only the parameters relate to memory locations is used for the DAQ lists. This type of DAQ lists provides an advantage to the measurement tool by providing flexibility in putting the DAQ lists together and dynamically structure the DAQ lists.

2.4.5 Configuring DAQ lists

DAQ lists can be configured either statically or dynamically. The slave can have certain fixed limits for the number of DAQ lists, number of ODTs for each DAQ list and the number of ODT entries of every ODT. Depending on the slave the type of configuration is also determined, master cannot request for a particular type of configuration. The master is allowed to configure the DAQ lists' direction, pre-scalar, priority and to which event channel it should be connected.

When DAQ is configure statically, the information about the structure of DAQ- lists with ODTs and the ODTs entries are available to slave. The master get the information about, the maximum number of DAQ lists (MAX_DAQ), maximum number of ODTs each DAQ can have (MAX_ODT) and the maximum number of ODT entries each ODT can have (MAX_ODT_ENTRIES). The master can edit the ODT entries, i.e. it can change the address and also the address extension that is linked to a memory space.

Dynamic configuration is the most preferred way of configuring the DAQ lists as it provides more flexibility. Dynamic configuration is flexible but also has limits on the minimum number of DAQ list number range; number of configurable DAQ lists; rules to be followed on the allocation of first PID, numbering of DAQ lists and event channels.

The configuration of DAQ lists is done with the commands FREE_DAQ, ALLOC_DAQ, ALLOC_ODT and ALLOC_ODT_ENTRY. These commands get an error response ERR_MEMORY_OVERFLOW, if there is not enough memory to allocate the requested objects. If such an overflow error occurs the whole DAQ list configuration is invalid. During dynamic configuration, the master has to follow a special sequence for the use of the commands; failing to do so, the slave returns an ERR_SEQUENCE.

Initially, the previously allocated DAQs must be cleared using the command FREE_DAQ. Secondly, the master has to allocate DAQ lists with ALLOC_DAQ command. Thirdly, the master has to allocate all ODTs to all DAQ lists with ALLOC_ODT commands. Finally, the master has to allocate all ODT entries to all ODTs for all DAQ lists with ALLOC_ODT_ENTRY commands. Failing to allocate DAQs using commands in this order, slave returns a negative response. The table in Figure 2.26 indicates the allowed sequence for configuring DAQ lists dynamically. These rules make sure that the slave can allocate the different objects in a continuous way to the available memory which optimises its use and simplifies its management [27].


 after	FREE_DAQ	ALLOC_DAQ	ALLOC_ODT	ALLOC_ODT_ENTRY
FREE_DAQ	✓	✓	ERR	ERR
ALLOC_DAQ	✓	✓	✓	ERR
ALLOC_ODT	✓	ERR	✓	✓
ALLOC_ODT_ENTRY	✓	ERR	ERR	✓

Figure 2.26: Sequence of using commands for allocating DAQs dynamically

2.4.6 Data Stimulation Lists- STIM Lists

The memory location in the slave can be written from the master in a controlled way with the help of STIM lists. The use of STIM lists is based on exchanging of DTO messages with the communication that is synchronised to an event in the slave. Thus, the master knows the events in the slave to which it can synchronise to. When master sends data to slave through STIM, the slave must be informed in-advance about the memory location in which it can find the calibration parameter. STIM lists execute only at specific time intervals when the program is executing on the ECU, ensuring that none of the control parameters are modified directly online when the control-loop is executing. Instead, STIM lists provides ECU a mechanism to assign new parameters at different points in time in a controlled manner. STIM lists are similar to DAQ lists in its structure, it consists of ODTs and ODT entries.

2.4.7 XCP module in AUTOSAR

XCP module in AUTOSAR is located above the bus specific interfaces in case of CAN and FlexRay as shown in Figure 2.27 [28], and in the case of Ethernet, it is located above the Socket Adaptor. For transmitting and receiving XCP messages, unique PDU-IDs must be used. AUTOSAR XCP module supports the ASAM XCP Specification Version 1.1. AUTOSAR XCP module supports all the main features specified in ASAM XCP such as Synchronous data acquisition (measurement), Dynamic DAQ configuration, Synchronous data stimulation, on-the-fly memory calibration, Timestamped data transfer through DTOs, Block communication mode, Bypassing, and Seed & Key [28].

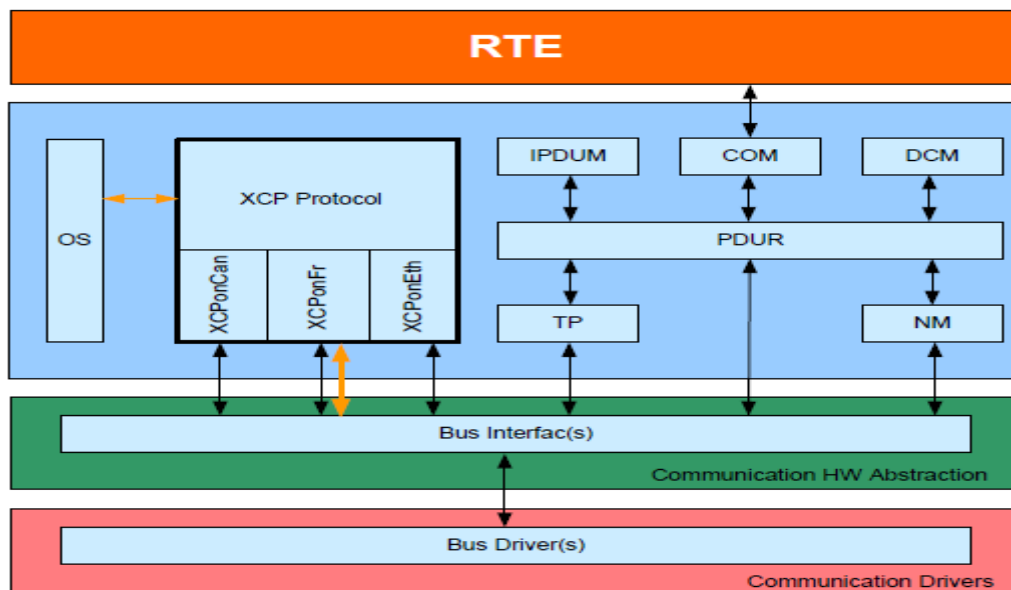


Figure 2.27: AUTOSAR XCP

When XCP over CAN is used, the PDUs have to be sent and received using transmit and receive APIs provided by the AUTOSAR CAN Interface. For transceiving XCP data via CAN, at least two different CAN identifiers have to be configured to be used by XCP as explained in section 2.4.2.1 XCP over CAN.

2.4.8 ASAM MCD-2 MC

ASAM MCD-2 MC (Measurement and Calibration) specification provides the description of the internal ECU variables used in measurement and calibration. MC systems need this description for parameterization of scalar constants, curves, and maps of the ECU software and for recording the response of the system through measurement variables during real time testing. The file extension of this description file is “A2L” which is an abbreviation of ASAM MCD-2 MC Language. The A2L file defines extensive support for look-up table of up to 5 dimensions [49].

The A2L file consists of details about memory segments, data types, record layouts, and dimensions and memory locations of ECU variables. The details about how the values should display in the measurement and calibration system, regardless of how the ECU-internal data are defined. The interface information between the ECU and the measurement and calibration system for read and write access is contained in the A2L file. For example for our system, the A2L file must contain the information of XCP on CAN interface to have a successful communication between the MC-system and the ECU.

2.5 AUTOSAR Methodology

AUTOSAR methodology provides certain technical approaches for some stages of development of a system. The AUTOSAR methodology as shown in Figure 2.28 [29] describes all the important stages of system development, from configuration at system level to the generation of an executable which will run on the ECU. AUTOSAR doesn't prescribe a precise sequence in which the activities shall be carried out.

The activities involved in the AUTOSAR Methodology as shown in Figure 2.28 are described briefly as follows:

1. System Configuration Input: Initially the system configuration input has to be defined. The software components and the hardware have to be selected, and overall system description has to be identified. The system level configuration involves defining the inputs by entering the data to or editing the format that AUTOSAR provides for formal description through the information exchange format (arxml) and the use of the following templates.

1a. Software component description: in this step, the descriptions such as data types, interfaces, ports, etc. which are required by SWCs are given.

1b. ECU Resource description: here the specifications for each ECU regarding the processor unit, memory, peripherals, communication interfaces etc. are described.

1c. System description: in this step the system constraints regarding the bus signals, the network topology used, communication matrix, gateway table and mapping/clustering of relevant SWCs.

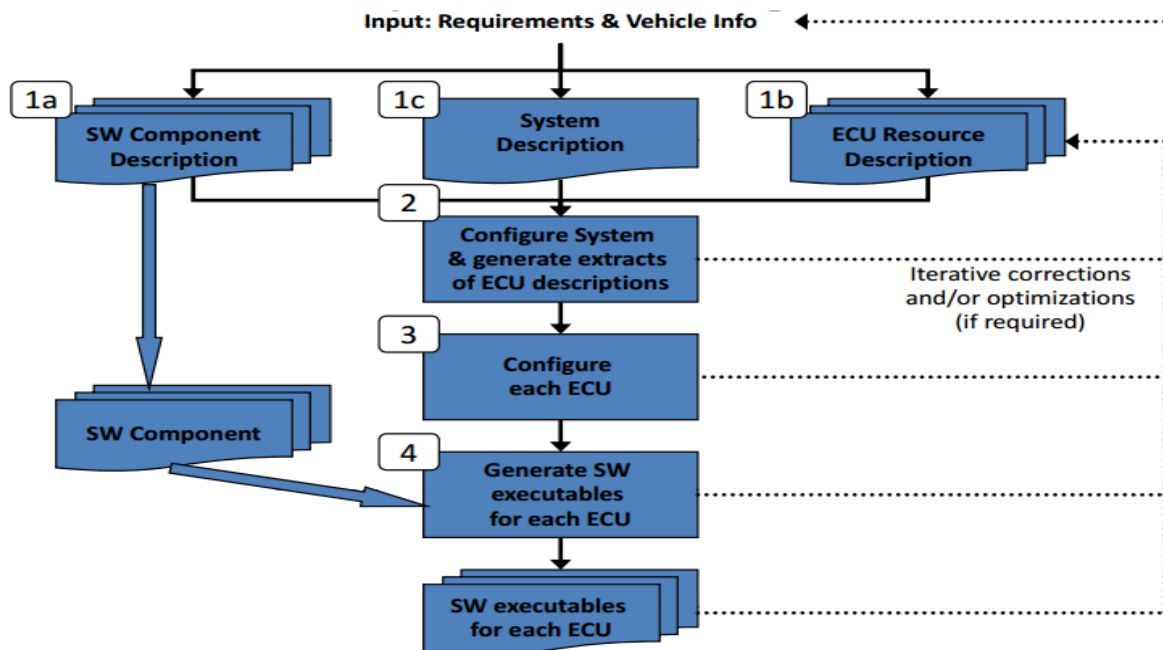


Figure 2.28: AUTOSAR System Design Methodology

2. Configure system & generate extracts of ECU descriptions: Configure system involves mapping of the SWCs to the ECUs with respect to resource and timing requirements. The outcome of configure system is **System Configuration Description**. Such a description consists of all the details of the system such as signal mapping, bus/network topology, and the mapping of SWCs to different ECUs. Next, the information required for a specific ECU is extracted from System configuration description.

3. Configure each ECU: This step adds all the required information to configure the core operations of the ECU such as configuring operating system to schedule the tasks, configuration of a communication stack, assigning SWC's runnable entities to a task created in operating system, etc.

4. Generate Executable: Finally, the code is generated from the configuration of the ECU done in previous step. This involves validating and generating the code from configured BSW modules and the RTE, and the source code written or generated for SWCs. These code files are linked and compiled into an executable which can be flashed to an ECU.

Along with the above described steps of the system design methodology, the SWCs are implemented according to the definitions required by the VFB. This is required to integrate the SWCs into the complete system, e.g. generating the components API and implementing the components functionality. However, the implementation of SWC is more or less independent from the ECU configuration. This is a key feature of AUTOSAR methodology [30].

3 PROJECT METHODOLOGY

In this chapter, the methodology used in this thesis will be introduced and the different hardware and software development tools used will be briefly explained

3.1 Method

As mentioned in section 1.3 , the objective of this thesis is to identify toolchain for ECU function development using AUTOSAR Standard and to have an optimal calibration mechanism using XCP over CAN. The thesis started with study phase for an understanding of AUTOSAR, CAN communication, XCP, and the stages of ECU function development. The understanding of AUTOSAR software architecture was very important in order to decide the toolchain and to contact the tool vendors. AUTOSAR is a consortium of OEMs, semiconductor manufacturers, tools and service providers. The latter group was researched and few vendors were shortlisted based on their reputation in AUTOSAR tools and location, keeping in mind the ease of communication. Tool vendors that were shortlisted are Vector, Mentor Graphics, dSPACE, ETAS, Electrobit and Arc Core. All these vendors were contacted asking for the price quote and the possibility of providing the evaluation license. Table below shows the different tool vendors and the tools that they provide.

Vendor	Origin	SWC	RTE	BSW
ArcCore	Sweden	SWC Builder	RTE Builder	BSW configurator
Elektrobit	Finland		EB tresos Autocore	
ETAS	Germany	ISOLAR A & ISOLAR EVE		
dSpace	Germany	System Desk and Targetlink		
Mentor Graphics	USA	Volcano V-Star & System Architect		
VECTOR	Germany	Davinci Developer	MICROSAR	

Arc Core was one of the first companies to respond to the query and provide an evaluation license. All other companies replied later and eventually presented their tools at NEVS including ArcCore. ArcCore's Arctic Studio was an obvious choice to start working on the thesis. Except dSPACE other tool vendors had a price on their evaluation license, hence the other AUTOSAR tool that was used in this thesis is dSPACE SystemDesk and TargetLink. ArcCore provides complete AUTOSAR toolchain; while dSPACE SystemDesk is limited to design network modeling and application layer SWC modeling according to the methodology defined by AUTOSAR.

Next step was to learn to use both the tools, Arctic Studio has a few example projects which were used as a reference while implementing our application, on the other hand, SystemDesk provides a step by step implementation document for a tutorial project, which was implemented to get familiar with the tool. After understanding the working of both the tools, it was decided that two toolchains will be tried for application layer of AUTOSAR. AUTOSAR toolchain-1 uses only Arctic Studio for complete function development for the ECU and toolchain-2 is to develop the application layer using dSPACE SystemDesk and TargetLink, and configuring

BSW and generating RTE with Arctic Studio. The latter helped in realising one of the core features of AUTOSAR i.e. abstraction between different software layers in its architecture.

The hardware used in this work is Texas instruments TMS570LS1227, as this MCU was available at NEVS and it is an automotive-grade μ C used for safety critical applications like BMS. Among the different MCUs supported by Arctic Studio, TMS570LS1227 is one of them. It was important to understand the features of MCU in order to do the BSW configuration of CAN cluster, MCU, and Port drivers. To get familiarise with the hardware, simple projects such as blinking led, establishing CAN communication between the development board and PC were implemented using Halcogen and Code composer studio.

To check the behaviour of the designed AUTOSAR system, the system was tested by sending CAN signals through BUS MASTER to the board and receive the output as CAN signal and read on BUS MASTER. Once the AUTOSAR toolchain was tested, the next step was to establish communication with the Measurement and Calibration (MC) tool. The MC tool that is used in this thesis is INCA from ETAS, as this software was available at NEVS along with the ECU and BUS Interface needed to connect between the MC tool and ECU. The A2L file is necessary for communication between the ETAS ES590.1 and the TMS570LS1227, the A2L file generate by Arctic Studio was incomplete, and as a result, A2L file was written manually using the A2L file available on the ASAM webpage as reference.

3.2 Development tools

This section lists all the software and hardware used during the course of this thesis and its brief introduction.

3.2.1 Software

The software tools used in AUOTSAR toolchain and in MC system are discussed in this section. Halcogen which was used to learn programming the TMS570 is not discussed in this section as it is not part of the final toolchain.

3.2.1.1 Arctic Studio and Arctic Core

Arctic Studio is an Eclipse-based IDE from ArcCore, which can be used to develop ECU function according to AUTOSAR standard. AUTOSAR version 4.0.2 was used in this work. ArcCore provides four plugins in Arctic Studio to develop a complete AUTOSAR solution, namely SWC builder, Extract builder, BSW builder, and RTE builder.

BSW builder provides a complete tool for editing and generating BSW configuration which is tailored to take advantage of the Arctic Core AUTOSAR platform. BSW builder allows users to add the desired BSW modules and configure each of these modules. Validation mechanism is built-in which points out the invalid values entered during the configuration. Use of AUTOSAR standard XML-format (arxml) and the extension points in eclipse provides ways for the users to integrate other tools to Arctic Studio and the BSW builder.

RTE builder is the tool used to configure the RTE and generate RTE code. RTE Configurator allows mapping of RTE runnables to tasks and OS events, validation to ensure a runnable RTE, and generation of RTE header and source files. Since RTE configuration needs access to the

information of OS configuration, RTE builder is tightly integrated with the BSW builder, this means that the tool can access all BSW configured for the specified ECU.

SWC builder allows users to create and edit AUTOSAR SWCs, ports, interfaces, data types and much more. The tool not only creates the SWCs, it also allows to import and modify the existing SWCs and validation rules are included in the tool which helps to identify the missing objects and incorrect configurations of the SWCs. SWCD language is used to create and edit the SWCs, ports, interfaces etc.

Extract builder is used to create an AUTOSAR ECU from the SWCs created using SWC builder or imported from a third party tool. In this tool, the ECU integrator defines the software architecture with all the SWCs, ports and the connection between them. The integrator also connects to the outer world i.e. System signals are connected to environment ports of the software architecture. Extract builder is also closely integrated with the BSW builder. SYSD language is used to create SWC prototypes, to connect SWCs to ports and to system signals [35].

Arctic core is the name given to the AUTOSAR embedded platform by ArcCore, which provides everything needed to build software for an automotive ECU. It includes all features required in an automotive ECU including communication services, diagnostic services, and a real time OS. Arctic core package supports different μ C architectures like PowerPC, ARM Cortex and Renesas architectures [36].

Arctic Studio and Arctic Core, unlike other AUTOSAR vendor tools, are open source. A commercial license can be bought for developing products for commercial purpose which removes the obligation of releasing the product under GPL.

3.2.1.2 dSPACE- SystemDesk and TargetLink

SystemDesk is a system architecture tool which is used for modeling AUTOSAR architectures and systems for application software. SystemDesk provides a comprehensive graphical modeling, this feature results in efficient and less error working during the large projects. ARXML files can be imported and exported to and from SystemDesk. Communication matrices in different formats such as DBC, LDF etc. can be imported to SystemDesk [37].

TargetLink is another software from dSPACE, which generates the production code (C code) directly from the model based environment such as MATLAB/Simulink/Stateflow. TargetLink provides a library block-set similar to the Simulink libraries which provide an extended dialogue box for entering the implementation specific information. An optional TargetLink AUTOSAR software helps using the TargetLink modeling, simulation and code generation options available for designing the AUTOSAR SWCs. TargetLink contains a database known as a data dictionary, which contains all the information that is relevant for a designing a model, generating code and implementation details for an ECU. When generating code from the modelled behaviour using TargetLink, AUTOSAR- compliant code can be generated [38].

TargetLink and SystemDesk work well together with the sophisticated feature of exchanging data between the two through the SWC container exchange, where the data from TargetLink data dictionary such as data types, units, initial values of ports and the generated code can be mapped to respective SWCs in SystemDesk. On the other hand, the software architecture that is modelled in SystemDesk can be imported to TargetLink to get the frame model.

SystemDesk and TargetLink are both licensed software and an evaluation license was provided by dSPACE which was used in this thesis.

3.2.1.3 ETAS INCA

INCA provides a complete functionality for measurement and calibration as well as the software tools for calibration data management, measurement data analysis and flash programming. INCA supports ECU descriptions for MC-systems, measurement data exchange and protocols such as CCP and XCP for communication compliant to standards such as ASAM MCD-2, ASAP3, ASAM MCD-3 MC, CCP and XCP [39].

INCA v7 was used in this thesis work and it licensed software which was available at NEVS.

3.2.1.4 MATLAB/SIMULINK

Simulink is a software from MathWorks which has a graphical programming environment for control system modeling, simulation and analysing multi domain systems with dynamic behaviour. Simulink supports simulation, automatic code generation, and continuous test and verification of embedded systems [40]. It is one of the most widely used graphical programming environment in automotive industries for model-based design of the plant and control model.

MATLAB/Simulink is a licensed software and the license used for this thesis was an academic license.

3.2.1.5 Code composer studio

Code composer studio (CCS) is an IDE that supports Texas Instruments Microcontroller and Embedded processors. CCS has a suite of tools for debugging and developing the embedded applications. Different kinds of licenses are available for CCS depending on the purpose of use and the features used. For this thesis, a free license was used [41].

3.2.1.6 Vector CANdb++ editor

One of the important artefacts which is needed for a distributed ECU network based on CAN is the communication description or communication matrix. Most commonly used format of this communication description files are DBC (Database for CAN) files. The DBC databases contain the descriptions of CAN network, the ECUs connected to the CAN bus, and the CAN messages and signals. To create and edit the DBC file for this work, Vector's CANdb++ was used [42].

CANdb++ editor is included in most of the Vector CAN tools and can be also found in the demo software. The CANdb++ used in this thesis was from the demo software.

3.2.1.7 BUS MASTER

BUS MASTER is an open source software for design, monitoring, analysis and simulation of CAN, LIN, and FlexRay networks. It was conceptualised, designed and developed by Robert Bosch Engineering India (RBEI) and is currently a joint project of RBEI and ETAS GmbH [43]. BUS MASTER supports a number of different CAN Bus interface modules, creation, and editing of CAN databases, logging and replay of CAN messages [44].

3.2.2 Hardware

The hardware components used in this thesis are discussed in this section.

3.2.2.1 Texas Instruments – TMS570LS1227

The TMS570LS1227 device is a high-performance automotive grade microcontroller family for safety critical systems. Battery Management System is an application which is safety critical, this was one of the reasons to use TMS570LS1227. The safety architecture of this device includes dual CPUs in lockstep, ECC on both the flash and the data SRAM, CPU and memory BIST logic, parity on the peripheral I/Os. TMS570LS1227 contains ARM Cortex-R4F floating point CPU. It supports the big endian [BE32] format [46].

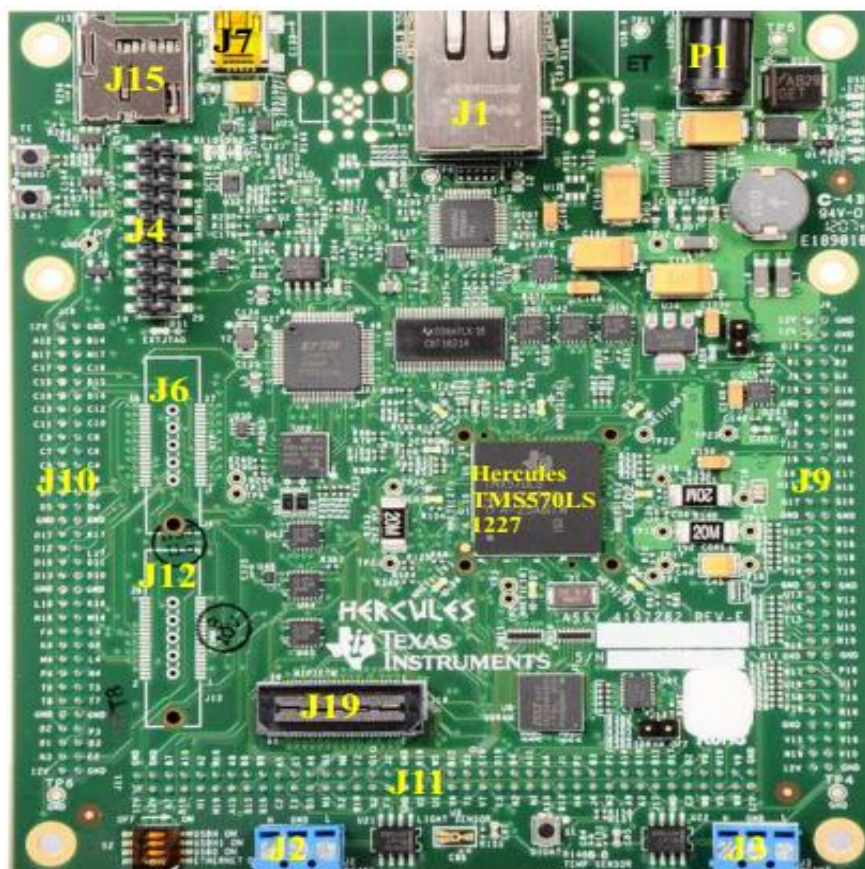


Figure 3.1: TMS570LS1227 HDK Board

Connector	Size	Function
J1	RJ45	Ethernet
J2	3 terminal, 2.54mm	DCAN1
J3	3 terminal, 2.54mm	DCAN2
J4	10x2, 2.54mm	ARM 20pin JTAG header
J6	19x2, mictor	RTP
J7	4pin, Mini-B USB	XDS100V2 USB
J9	33x2, 2mm	Exp P1, SPI1, SPI5, ADC
J10	33x2, 2mm	EXP P2, SPI2, EMIF, ECLK
J11	40x2, 2mm	EXP P3, SPI3, GIO, NHET, DCAN, LIN
J12	19x2, mictor	DMM
J15		SD card
J19	30x2, MIPI	ETM MIPI Header
P1	2.5mm	+12 V In

Figure 3.2: Connectors on TMS570LS1227

The Figure 3.1 and Figure 3.2 show the TMS570LS1227 HDK board and its connectors respectively [45]. The device has multiple communication interfaces for SPI, LIN, SCI, CAN, I2C, Ethernet and FlexRay communications. There are 3 DCANs and it supports CAN 2.0 protocol standard and uses a serial, multi-master communication protocol and communication rates of up to 1Mbps. The detailed description of different features of TMS570LS1227 can be found in [46].

3.2.2.2 PEAK Systems – PCAN USB adapter

The PCAN-USB adapter establishes a simple connection to CAN networks. The opto-decoupled version of the adapter provides galvanic isolation of up to 500 Volts between the PC and CAN side. The voltage supply to the adapter is via USB. It supports bit rates from 5 Kbits up to 1 Mbits [47].

3.2.3.3 ETAS ECU and BUS interface module

The ETAS 590.1 is a compact hardware module that supports the ECU and bus interfaces Ethernet/XETK, ETK, CAN and Serial interfaces. This device is also suitable to be used in-vehicle. Main features supported by this device are ECU calibration, diagnostics and flash programming. ETAS 590.1 module has seamless integration with INCA and it can be used for Bus monitoring, logging, and collection of time synchronous data [48].

4 AUTOSAR FUNCTION DEVELOPMENT

This section presents the implementation of an example application, describing the steps involved in developing the application layer according to AUTOSAR standard with Arctic Studio, and SystemDesk and TargetLink. Then, defining the system signals and creating ECU extract will be discussed in the System Design section, the configuration of different BSW modules are presented in the BSW configuration section. Finally, building the executable and programming the MCU will be described.

4.1 Modeling Application layer

The application layer consists of SWCs. As this thesis is aimed at identifying possible toolchains for function development with AUTOSAR, a simple application was considered. Two tools were used to design the application layer. In this section, designing using both SystemDesk and Arctic Studio will be discussed.

4.1.1 SWC Model

The application consists of one SWC called Power which has two RPorts and one PPort. Figure 4.1 depicts the system that will be modelled at the application layer. The two RPorts are called RpCurrent and RpVoltage, and the PPort is called PpPower. RpCurrent, RpVoltage, and PpPower ports are all assigned to sender-receiver interfaces if_current, if_voltage and if_power respectively. The SWC also contains a parameter which is used for calibration called GainFactor. The implementation of the SWC is the multiplication of the values of the signals current, voltage and calibration parameter GainFactor. The data signals Current and Voltage are received via CAN communication and Power is sent on the CAN signal. The calibration of the parameter GainFactor is done through the MC tool which is discussed in section 5.5.2 Testing with setup -2.

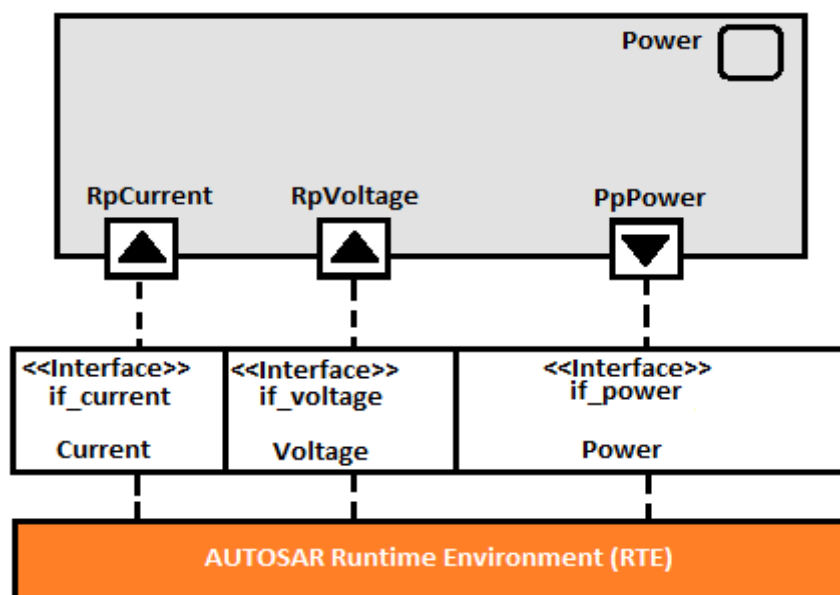


Figure 4.1: System architecture

The major steps involved in designing such a system with Arctic Studio and with SystemDesk will be discussed in the next two sections.

4.1.2 Modeling SWC with Arctic Studio

To model SWC in Arctic Studio, ARtext SWCD language is used. Modeling SWC includes: data model development where the data types and interfaces are defined, Atomic SWCs definition where the ports corresponding to the application are assigned, modeling the internal behaviour of atomic SWCs, and specifying mapping sets.

Firstly, the data model development involves the definition of application data types (ADTs), implementation datatypes (IDTs), and port interfaces. Since AUTOSAR 4, AUTOSAR distinguishes ADT and IDT. ADTs define data in the application view as physical values, whereas the same data is represented as internal values by IDTs and it also defines the implementation details. ADTs are not mandatory, however, when used they need to be mapped to the corresponding IDTs.

The interfaces defined are all sender-receiver interfaces and the data element of each of the interface is one of the defined types. For the system shown in Figure 31, the data model is as shown in the code below.

```
/*-----Current,Voltage,Power -----*/
/*-----Application data type -----*/
int app ACurrent
int app AVoltage
int app APower
/*-----Implementation data type -----*/
int impl ICurrent extends uint16
int impl IVoltage extends uint16
int impl IPower extends uint32
/*===== */

/*-----Calibration Parameter-----*/
int app AGainFactor
int impl IGainFactor extends uint16
/*===== */

/*-----Interfaces-----*/
interface senderReceiver If_Voltage{
    data AVoltage voltage_v
}
interface senderReceiver If_Current{
    data ACurrent current_v
}

interface senderReceiver If_Power{
    data APower power_v
}
```

To map IDT with ADT, a keyword dataTypeMappingSet is used and the mapping is done as shown below.


```

/*-----Data type Mappings----- */
dataTypeMappingSet DTMappings {
    map ICurrent ACurrent
    map IVoltage AVoltage
    map IPower APower
    map IGainFactor AGainFactor
}

```

Once the data types and interfaces are defined, Atomic SWCs can be defined. There is only one SWC in our system and it is of the application SWC type. The code below shows the modeling of Power SWC containing ports which are assigned to its respective interfaces.

```

/*-----SWC----- */

component application Power{
    ports{
        receiver RpVoltage requires If_Voltage
        receiver RpCurrent requires If_Current
        sender PpPower provides If_Power
    }
}

```

Next step is to design the actual implementation for the defined SWC. Every application SWC must contain internal behaviour which describes the RTE aspects of a component that is the calibration parameters, data type mappings, kind of access to the data access to the data elements, runnable entities, and the events they respond to. A calibration parameter has to be defined in the enclosing internal behaviour. Runnable entities are the smallest code fragments that are provided by the component and are a subject for scheduling the underlying operating system.

For the system under consideration, a calibration parameter GainFactor is defined and a runnable Get_Power. The Get_Power runnable entity contains the information about the type of data access to the data element of its ports and a timing event which triggers the runnable every 0.01s. The code implementation of this is as shown below.

```

/*-----internal behavior----- */
internalBehavior PowerInternalBehaviour for Power{
    dataTypeMappings {
        DTMappings
    }
    instanceParam AGainFactor GainFactor "0" // Initial Value = 0
    runnable Get_Power[0.0]{ // Minimum start interval = 0.0s
        dataReadAccess RpVoltage.voltage_v
        dataReadAccess RpCurrent.current_v
        dataWriteAccess PpPower.power_v
        timingEvent 0.01 as GetPower_timingEvent
        paramAccess GainFactor
    }
}

```

AUTOSAR consists of various kinds of atomic SWC, but only an application SWC possess an implementation. The purpose of implementation is to identify application SWC that is

implemented, link to code (source code, object code, etc.), and optionally specify required build environment. The implementation for PowerInternalBehaviour is as shown in the ARtext SWCD code below.

```
implementation IMPL_Power for PowerInternalBehaviour {
    language c
    codeDescriptor "src"
}
```

When an SWC is defined, it should be instantiated and linked to other SWCs via ports. This is known as composition-SWC. The role of an AUTOSAR composition is to allow encapsulation of functionality by aggregating existing SWCs. Though the system in Figure 4.1: System architecture contains only one SWC, a composition is necessary. This is done in SWCD code as shown below.

```
composition RootSwcComposition{
    prototype Power Power
}
```

4.1.3 Modeling SWC with SystemDesk

Unlike Arctic Studio, SWC modeling in SystemDesk is via dialog boxes and graphical diagrams. Here the exact system which was designed using Arctic Studio will be designed. As the modeling of SWC with system desk involves many dialog windows and views, only the major steps will be presented in this section. There are two ways to create an SWC in SystemDesk i.e. by adding SWCs in the folder structure in project manager view or by adding SWCs graphically in the composition SWC. The latter method will be discussed here.

To create a new project, one can import AUTOSAR templates into the project and this results in a folder structure as shown in Figure 4.2.

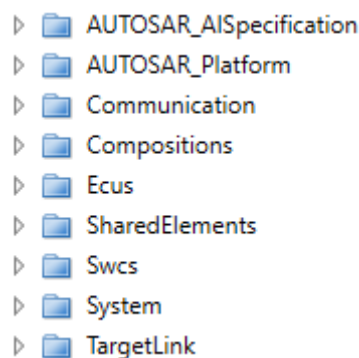


Figure 4.2: Folder Structure in SystemDesk

Firstly, a composition SWC was added under the Composition folder and a composition diagram. In the composition diagram, an application SWC was added and named as Power. To this SWC block, two RPorts namely RpCurrent and RpVoltage and a PPort named PpPower.

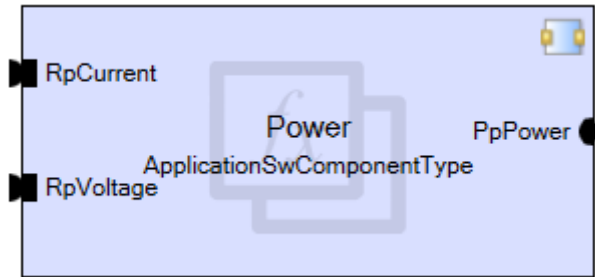


Figure 4.3: Application SWC with ports

Figure 4.3 shows SWC with ports without any interface assigned to it. The interfaces can be added under the folder SharedElements > Interfaces folder seen in Figure 4.4. Figure 4.4 shows the added sender-receiver interfaces along with their data elements. Figure 4.5 shows SWC after assigning the interfaces to the ports at composition level and sub level.

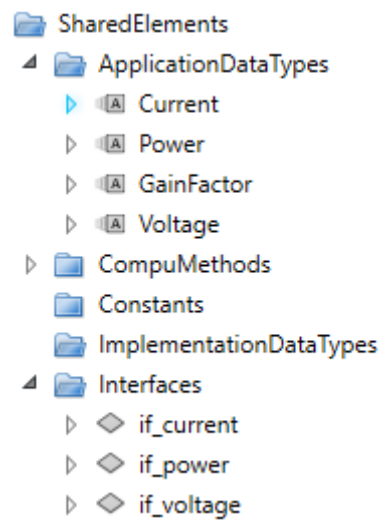


Figure 4.4: Application Data Types and Interfaces

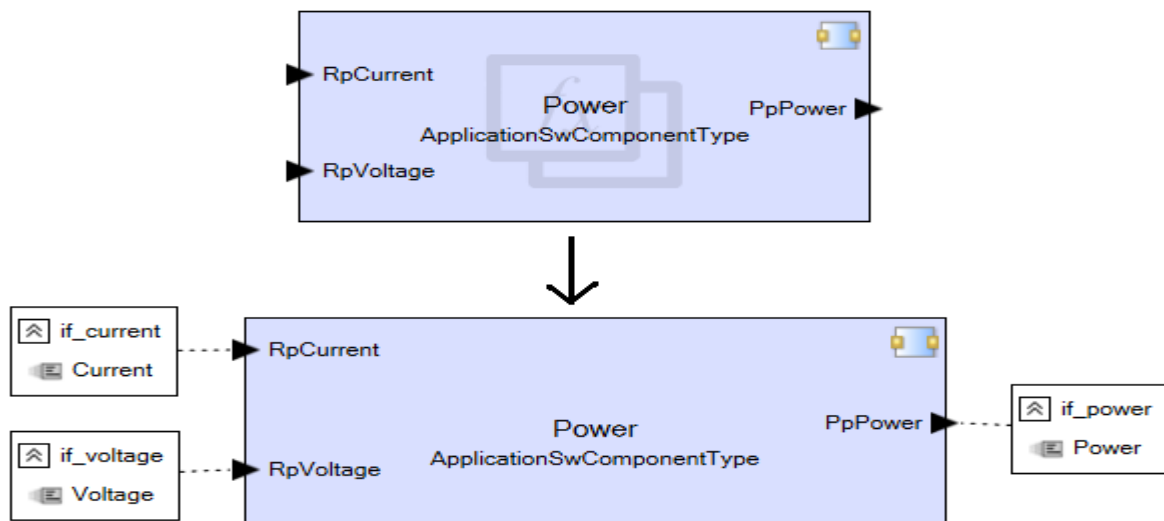


Figure 4.5: After assigning the interfaces to the port

As mentioned in the previous section, each application SWC must have its internal behaviour and implementation. The internal behaviour, data mapping set in SystemDesk can be added to the SWC as shown in Figure 4.6. Internal behaviour consists of runnable Get_Power.

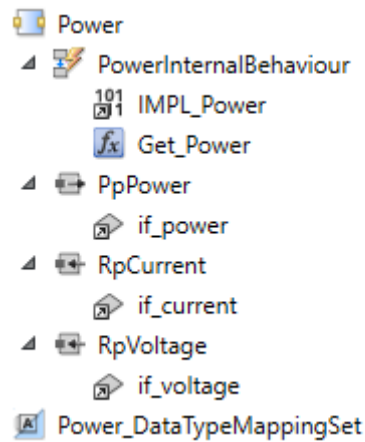


Figure 4.6: Internal Behaviour and Data type Mapping Set

Information such as data access, calibration parameter, and events that trigger the runnable is given in a dialogue shown below in Figure 4.7, which contains all the required tabs.

Runnable Entity: Get_Power

Arguments		Mode Access Points	Mode Switch Access	Wait Points	Exclusive Areas	Special Data
General	Triggered by	Data Access	Internal Triggering Points	External Triggering Points	Parameter Access	
Name(s)		Port	Element	Access/Direction	Data Type	
DRA_RpCurrent_Current		RpCurrent	Current	Read		Current
DRA_RpVoltage_Voltage		RpVoltage	Voltage	Read		Voltage
DWA_PpPower_Power		PpPower	Power	Write		Power

Figure 4.7: Runnable entity dialogue

Finally, the implementation of the SWC is as shown in Figure 4.8, this indicated the implementation of the behaviour of the runnable will be in C language and will be generated in TargetLink or hand coded. This generated code can be linked to the implementation under the code descriptors tab.

Required Generator Tools		Compilers	Special Data	Advanced
General	Code Descriptors	Required Artifacts	Generated Artifacts	
Short name:	IMPL_Power			
Desc:				
Category:				
Behavior ref:	PowerInternalBehaviour			
Programming language:	C			
Used code generator:	TargetLink/HandCoded			

Figure 4.8: SWC implementation dialogue

4.1.4 Runnable

Implementation of SWC as seen in both Arctic Studio and in SystemDesk was mentioned as the C language. The C code can be implemented by hand code or can be a generated code from code generator such as TargetLink. In this thesis, both hand coded and generated code were implemented. Since the behaviour of the runnable considered is very simple, which involves the multiplication of the current, voltage signal and parameter GainFactor. The hand coded part of the implementation is not presented in this section, only the model-based implementation in Simulink and the AUTOSAR compliant code generated are presented.

Once the SWC is modelled in SystemDesk, it is possible to export the information to Data dictionary of TargetLink. With this information, a frame model can be generated as shown in Figure 4.9.

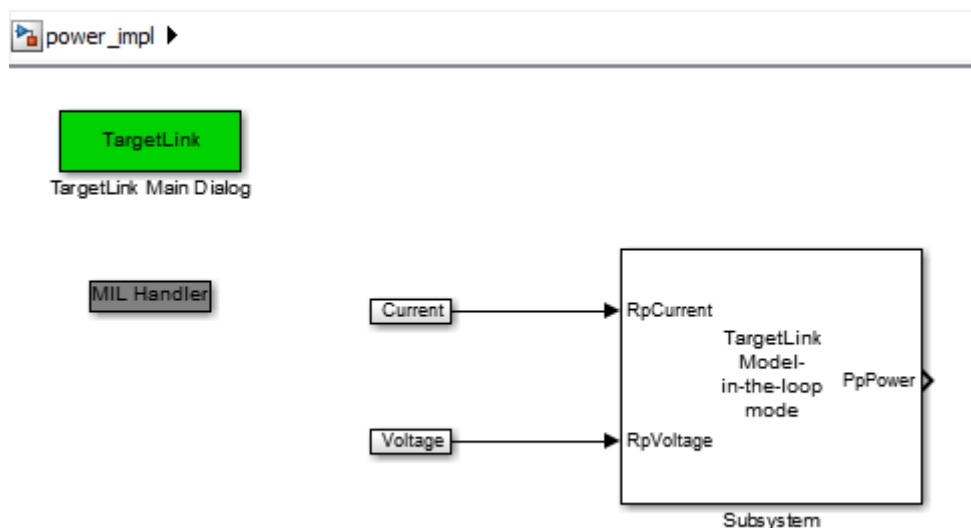


Figure 4.9: Generated Frame model

The frame model contains subsystem with a runnable block indicating that the behaviour of this runnable should be modelled at that level of the subsystem. Figure 4.10 shows the

subsystem with a runnable block for Get_Power runnable and the dialogue showing the properties of this runnable.

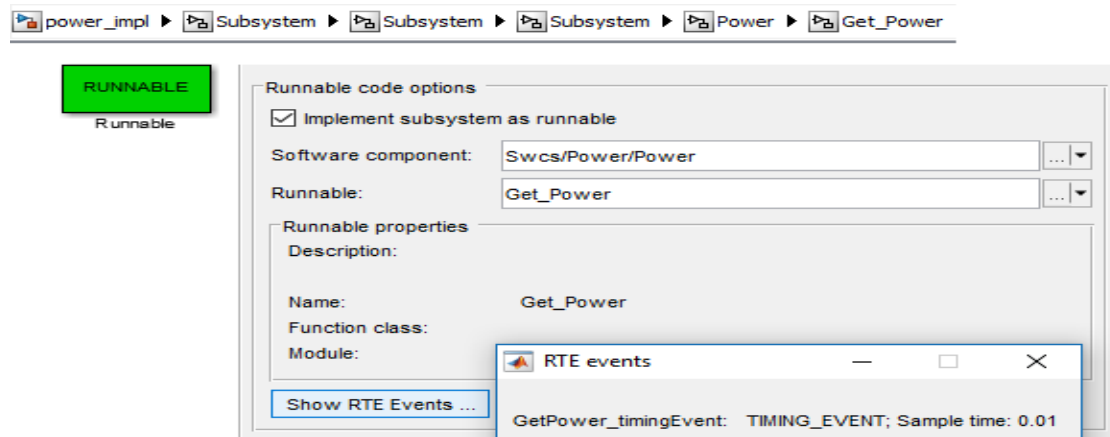


Figure 4.10: Get_Power Runnable & Properties

The behaviour of the runnable Get_Power is modelled in the subsystem when its corresponding runnable block is found. As mentioned earlier, the implementation for this runnable is simple and is as shown in Figure 4.11.

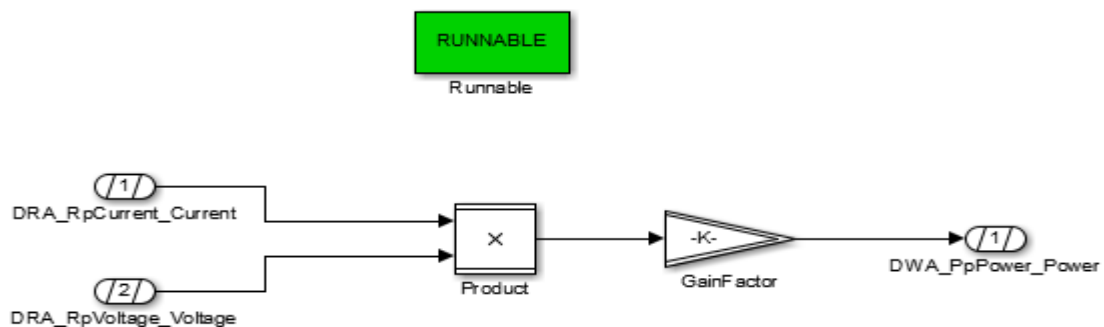


Figure 4.11: Get_Power behaviour implementation

Next step is to generate the code from this implemented model. TargetLink has to be informed that the code that will be generated should be AUTOSAR compliant. This can be indicated in the TargetLink main dialogue box as shown in Figure 4.12.

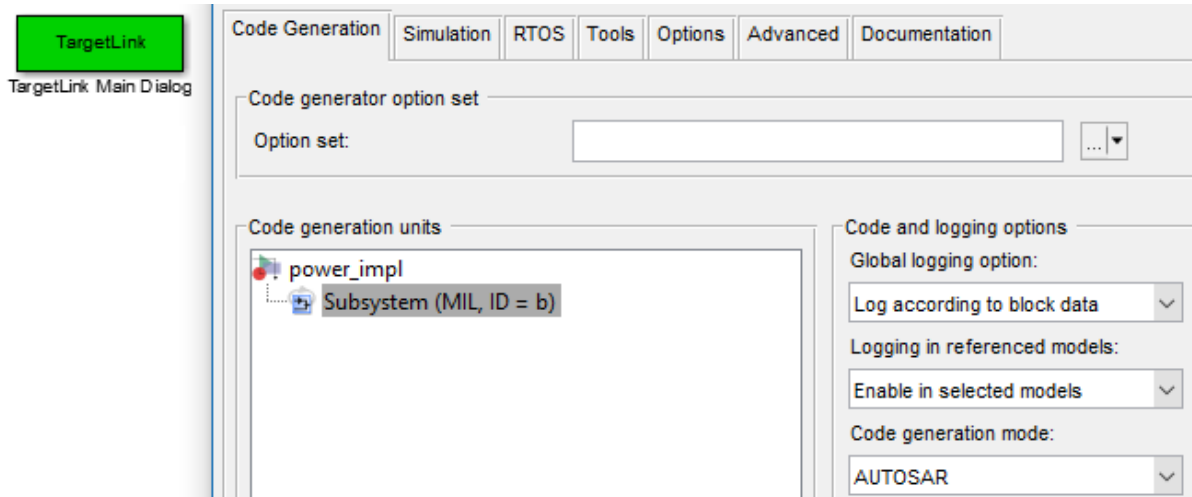


Figure 4.12: TargetLink Main Dialogue

The AUTOSAR compliant code is generated and is as shown below. The code generated should be AUTOSAR compliant as RTE expects, this ensures successful communication between the SWC and RTE. According to AUTOSAR specification for RTE, RTE API includes API calls to support data read access and data write access. The API calls `Rte_IWrite` and `Rte_IRead` access the data copies for write and read access respectively. `Rte_CData` provides access to the calibration parameter an AUTOSAR SWC defined internally [9].

```
FUNC(void, Power_CODE) Get_Power(void)
{
    Rte_IWrite_Get_Power_PpPower_Power((uint32) (((sint16) (uint8)
    (((uint8) Rte_IRead_Get_Power_RpCurrent_Current()) *
    ((uint8) Rte_IRead_Get_Power_RpVoltage_Voltage())) *
    ((sint16) Rte_CData_Cal_GainFactor())));
}
```

The name of each of the functions in the above code snippet is according to AUTOSAR RTE specification. For example, to access the data elements defined with `DataWriteAccess` the format of the function should be “`Rte_IWrite_<re>_<p>_<d>` ([IN RTE_Instance], IN <type>)”. Where <re> is the name of runnable entity, <p> is the port name, <d> the name of data element, an optional field for RTE_Instance and <type> is the data type of the data element [9].

4.2 System modeling

After implementing all the SWCs and creating a top level software composition for our software architecture, each of the SWC must be mapped to an ECU instance. ECU extract and System extract will be used interchangeably in this section. Along with mapping the SWC to ECU, to complete the modelling of the system defined in Figure 4.1, the system signals and the network topology must be specified. In this section creating the system signal, importing network topology and creating ECU extract with both Arctic Studio and SystemDesk will be discussed.

4.2.1 System modeling with Arctic Studio

In Arctic Studio, System modeling can be considered in two steps i.e. defining system signal & its communication, and creating ECU extract or system design. For system communication modeling in Arctic Studio, another ARtext language known as System description (SYSD) language is used.

Firstly the system signals and the internal signals are defined and the internal signals are mapped to the respective PDUs as shown in the code below.

```
//System Signals
systemSignal SCurrent
systemSignal SVoltage
systemSignal SPower

// internal signal corresponding to the above definitions
isignal for SCurrent as ICurrent
isignal for SVoltage as IVoltage
isignal for SPower as IPower

// internal signals mapped to its corresponding PDU Frames
isignalPdu BatterySignals{
    isignalIPduMapping for ICurrent as Current
    isignalIPduMapping for IVoltage as Voltage
}

isignalPdu POWER_CALCULATED{
    isignalIPduMapping for IPower as Power
}
```

ECU extract or System Extract contains the composition SWC which was previously created as root composition, implementation mappings where the implementation is mapped to the SWC, and signal mappings where the data elements are mapped to the system signals. The code for ECU extract is as shown in the code snippet below.

```
// creating an ECU/System Extract
system EcuExtract{
    // Composition SWC
    rootComposition RootSwcComposition

    // Implementation mapped to SWC
    mapping as ImplementationMapping {
        implMap IMPL_Power to Power
    }

    // System signal to data mapping
    mapping as SignalMappings{
        signalMap Power.PpPower.power_v to SPower
        signalMap Power.RpCurrent.current_v to SCurrent
        signalMap Power.RpVoltage.voltage_v to SVoltage
    }
}
```


With ECU extract the system modeling with respect to application layer is complete. The ECU extract or System extract is now exported in ARXML format and is integrated with the BSW configurator.

4.2.2 System modeling with SystemDesk

In SystemDesk, a system is managed and configured in the system manager. A System in a system manager can contain at most one root SW composition, an ECU instance, and a network topology. CAN communication is used in our system and the CAN network with signals and node was defined in a DBC file using Vector CANdb++ editor. The network contains one node called MCU that will be TMS570LS1227 and two CAN messages namely BatterySignals containing two signals Current and Voltage, and POWER_CALCULATED containing one signal called Power. Figure 4.13 shows the described topology and Figure 4.14 shows the details of the three signal.

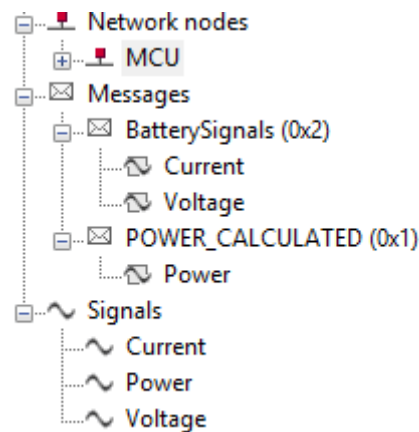


Figure 4.13: CAN Network topology

Name	Message	Startbit	Length [Bit]	Byte Order	Value Type	Initial Value	Unit	Comment
Current	BatterySignals	0	16	Intel	Unsigned	0	A	Current from battery
Voltage	BatterySignals	16	16	Intel	Unsigned	0	V	Voltage from Battery
Power	POWER_CALCULATED	0	32	Intel	Unsigned	0	W	Output power

Figure 4.14: Signals

The information about the network is imported from DBC file to SystemDesk and the System looks as shown in Figure 4.15 containing the SW composition created previously, along with an ECU instance and the network information.

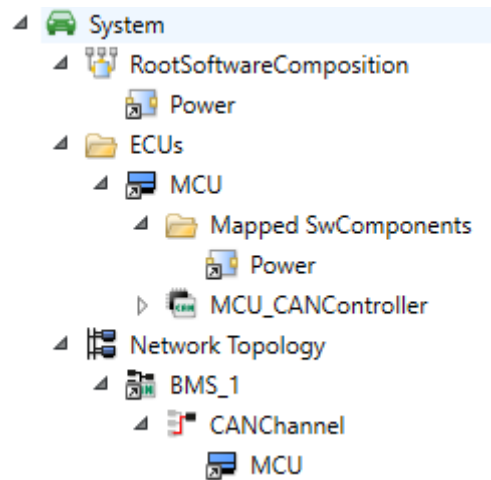


Figure 4.15: System

Figure 4.15 shows the mapping of SWC Power to the ECU instance. Along with SWC-to-ECU mapping, data elements are mapped to the system signals added through DBC files as shown in Figure 4.16. SWC to Implementation mapping, SWC-to-ECU mappings is done under other tabs in Figure 4.16.

System: System

General	Network Clusters	ECUs	Gateways	SWC-to-ECU Mappings
Data Mappings	SWC-to-Impl Mappings	Special Data	Advanced	

Sender receiver data mappings:

Filter				
Data element instance				System signal (group)
Component Instance	Port	Data Element		
Power	RpCurrent	Current	Current	
Power	RpVoltage	Voltage	Voltage	
Power	PpPower	Power	Power	

Figure 4.16: System - Data Mappings

Once the system is configured, the system extract or ECU extract can be exported to an ARXML file and integrated with BSW. This artefact from SystemDesk will be used in the Arctic Studio during BSW configuration. This is the points where the two tools converge.

4.3 Basic Software Configuration

In BSW configuration, different BSW modules used will be configured. Though there are different modules, some of the modules are dependent on the other, thus, the order in which the BSW modules are configured is important. For example, when configuring CAN cluster, CAN driver needs to be configured first, then the CAN interface module in order to provide communication to the upper modules like PDU router and COM modules. Figure 4.17 shows the different BSW module used in this work. The following sections describe the configuration of each module.

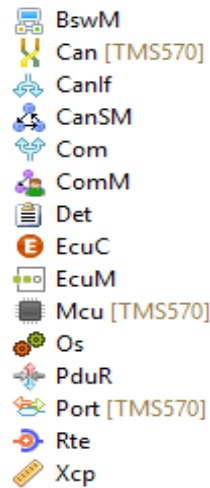


Figure 4.17: Used BSW Modules

4.3.1 Microcontroller Unit

The MCU driver module accesses the microcontroller hardware directly. The frequency of the main clock which will be used by other modules is configured here as shown in Figure 4.18. The different mode settings of MCU i.e. NORMAL, RUN, SLEEP modes are also configured for MCU module. This mode setting will be used by ECU manager.

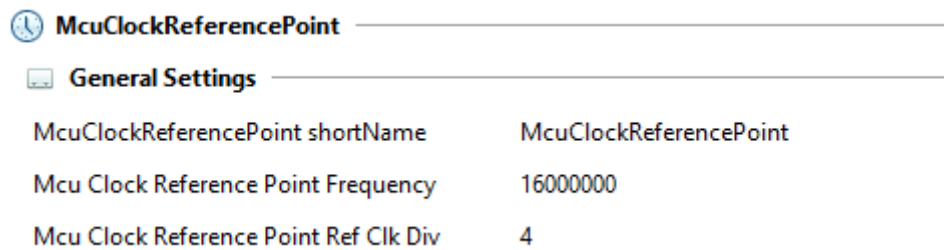


Figure 4.18: MCU Clock Reference Point

4.3.2 ECU Manager

The EcuM module is an important module to be configured, as it handles the initialization, sleep and wake-up process of the processor. EcuM used in this work is fixed EcuM, the configuration of this is shown in Figure 4.19. Here the channel for which the ECU should call ComM_CommunicationAllowed is configured.

EcuMFixedConfiguration

General Settings

EcuMFixedConfiguration shortName	EcuMFixedConfiguration		
Ecu MCom MCommunication Allowed List	<input checked="" type="checkbox"/>		ComMChannel
Ecu MNormal Mcu Mode Ref	NORMAL		
Ecu MNvram Readall Timeout	10.0		
Ecu MNvram Writeall Timeout	10.0		
Ecu MRun Minimum Duration	10.0		

Figure 4.19: EcuM Fixed Configuration

The corresponding MCU mode for EcuM sleep mode and the reference to EcuM wake-up source are configured as shown in Figure 4.20.

EcuMSleepMode

General Settings

EcuMSleepMode shortName	EcuMSleepMode		
Ecu MSleep Mode Mcu Mode Ref	SLEEP		
Ecu MWakeup Source Mask	<input checked="" type="checkbox"/>		EcuMWakeupSource

Figure 4.20: EcuM Sleep mode

4.3.3 Port Driver

To initialise required ports and configure its port direction, slew rate, and port modes, a port driver module is used. As the aim here is to achieve CAN communication, the CAN ports on the MCU must be configured. Apart from CAN ports GIO ports were also initialized and connected to LEDs on the board for debugging the software and verification of the board. Figure 4.21 and Figure 4.22 shows the configuration of the CAN port for sending and receiving the CAN messages respectively.

PortPin

General Settings

PortPin shortName	CAN_TX
Arc Port Pin Open Drain Enable	False
Arc Port Pin Pull	PORT_PIN_PULL_NONE
Arc Port Pin Slew Rate	PORT_PIN_SLEW_RATE_MIN
Port Pin Direction	PORT_PIN_OUT
Port Pin Id	89
Port Pin Initial Mode	PORT_PIN_MODE_CAN
Port Pin Level Value	PORT_PIN_LEVEL_LOW

Hex

Figure 4.21: Port Configuration for CAN_TX

Here the Port direction is set as IN for receiving and OUT for sending port. Port is set to CAN mode on initialization. It is important to enter the right Pin Id as this value will be assigned to the symbolic name derived from the port pin container short name. The Pin Id for CAN_TX port is 89 and CAN_RX port is 90, this information can be accessed from the datasheet of TMS570LS1227 [46].

PortPin

General Settings

PortPin shortName	CAN_RX
Arc Port Pin Open Drain Enable	False
Arc Port Pin Pull	PORT_PIN_PULL_NONE
Arc Port Pin Slew Rate	PORT_PIN_SLEW_RATE_MIN
Port Pin Direction	PORT_PIN_IN
Port Pin Id	90
Port Pin Initial Mode	PORT_PIN_MODE_CAN
Port Pin Level Value	PORT_PIN_LEVEL_LOW

Hex

Figure 4.22: Port Configuration for CAN_RX

4.3.4 CAN Driver

CAN driver module is located in the MCAL and is the bottom most module in the CAN stack. Here the CAN controller and its baud rate, and the CAN Hardware objects for transmission and reception are configured. CAN controller activation and baud rate are configured to 500kbps as shown in Figure 4.23.

The screenshot displays two configuration panels. The top panel, titled 'CanController', includes a 'General Settings' section with the following fields: 'CanController shortName' set to 'CanController0', 'Arc Use Loopback' as an unchecked checkbox with an 'Unset' dropdown, 'Can Controller Activation' set to 'True', 'Can Controller Default Baudrate' set to 'CanControllerBaudrateConfig', and 'Can Controller Id' set to '0' with a 'Hex' button. The bottom panel, titled 'CanControllerBaudrateConfig', also has a 'General Settings' section with: 'CanControllerBaudrateConfig shortName' set to 'CanControllerBaudrateConfig', 'Can Controller Baud Rate' set to '500' with a 'Hex' button, 'Can Controller Prop Seg' set to '8' with a 'Hex' button, 'Can Controller Seg1' set to '5' with a 'Hex' button, and 'Can Controller Seg2' set to '2' with a 'Hex' button.

Property	Value	Buttons
CanController shortName	CanController0	
Arc Use Loopback	<input type="checkbox"/> Unset	
Can Controller Activation	True	
Can Controller Default Baudrate	CanControllerBaudrateConfig	
Can Controller Id	0	Hex

Property	Value	Buttons
CanControllerBaudrateConfig shortName	CanControllerBaudrateConfig	
Can Controller Baud Rate	500	Hex
Can Controller Prop Seg	8	Hex
Can Controller Seg1	5	Hex
Can Controller Seg2	2	Hex

Figure 4.23: CAN Controller Configuration

The Hardware objects for transmission and reception of the CAN message is configured by setting the type of CAN ID i.e. Standard or Extended or Mixed mode, referencing it to corresponding CAN controller, and the hardware object to support FIFO or message box for the messages. Figure 4.24 shows the configuration of receive hardware object.

The screenshot shows the 'CanHardwareObject' configuration panel with a 'General Settings' section. The fields are: 'CanHardwareObject shortName' set to 'Ctrl_0_Rx_0', 'Arc Can Fifo Enable' set to 'False', 'Can Controller Ref' set to 'CanController0', 'Can Handle Type' set to 'BASIC', 'Can Hw Object Count' as an unchecked checkbox with a 'Hex' button, 'Can Id Type' set to 'STANDARD', and 'Can Object Type' set to 'RECEIVE'.

Property	Value	Buttons
CanHardwareObject shortName	Ctrl_0_Rx_0	
Arc Can Fifo Enable	False	
Can Controller Ref	CanController0	
Can Handle Type	BASIC	
Can Hw Object Count	<input type="checkbox"/>	Hex
Can Id Type	STANDARD	
Can Object Type	RECEIVE	

Figure 4.24: CAN Hardware object configuration

4.3.5 CAN Interface

While configuring CAN communication stack, the next module to configure is the CanIf module in hardware abstraction layer which is above the CAN driver module. The configuration of this module involves configuring each receive and transmit CAN L-PDUs, also the receive and transmit CAN hardware objects configured in CanIf module is linked to the CAN hardware objects from CAN driver. Two receive PDUs and two transmit PDUs were configured as shown in Figure 4.25.

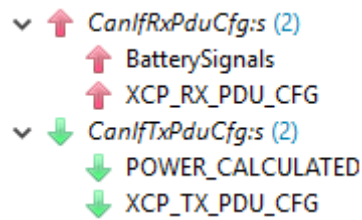


Figure 4.25: Configured CanIfPDUs

Configuring each of the CAN L-PDUs involve setting the CAN ID, CAN ID type, and data length code. The upper layer module to which the confirmation of successfully transmitted CAN TX/RX PDU ID has to be routed via is set as PDUR. Figure 4.26 shows the configuration of one of the CanIfTxPdu.

CanIfTxPduCfg

General Settings

CanIfTxPduCfg shortName	POWER_CALCULATED
Can If Tx Pdu Buffer Ref	CanIfBufferCfg
Can If Tx Pdu Pn Filter Pdu	<input type="checkbox"/> Unset
Can If Tx Pdu Ref	POWER_CALCULATED
Can If Tx Pdu Type	STATIC

CAN Properties

Can If Tx Pdu Can Id	1	Hex
Can If Tx Pdu Can Id Type	STANDARD_CAN	
Can If Tx Pdu Dlc	8	Hex

Transmit Confirmation Settings

Can If Tx Pdu User Tx Confirmation Name	<input type="checkbox"/>	
Can If Tx Pdu User Tx Confirmation UL	<input checked="" type="checkbox"/>	PDUR

Figure 4.26: CanIfTxPdu Configuration

4.3.6 EcuC

EcuC module is a virtual module to collect ECU configuration specific or global configuration information such as partitions defined for the ECU, predefined variant elements containing the definition of values for software system constants, and PDU objects that flow through the COM-Stack. For CAN communication stack, a PDU object for each CAN message had to be configured. The PDU objects used in this thesis are shown in Figure 4.27, and the configuration of one such PDU object in EcuC module is shown in Figure 4.28.

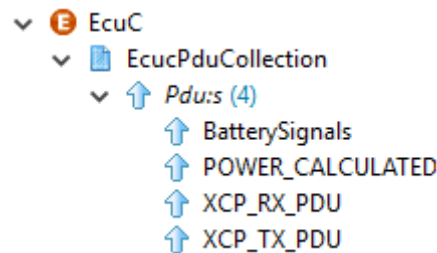


Figure 4.27: ECU PDU Collection

Figure 4.28: One PDU flowing through the COM- Stack

4.3.7 PDU Router

I-PDUs through the CAN communication stack is routed by PDUR. The PDUR module has to be configured indicating the other BSW modules with which it communicates and how the routing takes place. For CAN cluster considered in this thesis, the PDUR communicates with the COM and CanIf modules. The routing path for both these modules is configured as shown in Figure 4.29.

PduRBSwModules

General Settings

PduRBSwModules shortName

Canlf

PduR Lower Module

True

PduR Upper Module

False

PduRBSwModules

General Settings

PduRBSwModules shortName

Com

PduR Lower Module

False

PduR Upper Module

True

Figure 4.29: PDUR BSW modules

The configuration done in Figure 4.29 is important and it is also required to configure each I-PDUs routing path indicating the direction in which the signal flows and referring each I-PDU to the PDU objects configured in EcuC module. Here configuration involves indicating the source and destination of each I-PDUs. Figure 4.30 and Figure 4.31 shows the configuration for one of the I-PDUs.

PduRDestPdu

General Settings

PduRDestPdu shortName

PduRDestPdu

Arc Overriden Dest Module

☒

Com

PduR Dest Pdu Data Provision

☐

PduR Dest Pdu Ref

BatterySignals

PduR Dest Tx Buffer Ref

☐

PduR Tp Threshold

☐

Hex

Figure 4.30: Destination of BatterySignals w.r.t PDUR

PduRSrcPdu

General Settings

PduRSrcPdu shortName

PduRSrcPdu

Arc Overriden Src Module

☒

Canlf

PduR Src Pdu Ref

BatterySignals

Figure 4.31: Source of BatterySignals w.r.t PDUR

4.3.8 COM

COM module must be configured regardless of the type of communication network implemented. COM configuration involves configuring of different type of objects like I-PDUs, I-PDU Groups, Signals and Signal Groups.

AUTOSAR uses a concept of complex data type i.e. it contains more than one data element, this can be interpreted as a struct in C-language containing more than one primitive data type. Signal groups are represented as complex data type. Though it contains more than one signal, it can be considered as a single data element. RTE decomposes the complex data in single signals and sends them to the COM module [16]. There was no necessity of such a signal group in our case as Current and Voltage are considered as separate signals. It would have been necessary for a scenario where there was only one RPort in our system and both Current and Voltage signals had to be received from the same port. The COM signals used in our system are Current, Voltage and Power. The configuration of COM signals involves the setting of bit position, bit size, endianness and data type of the signal and reference to the ISignalToIPduMapping. When a DBC file is imported to Arctic Studio, most of these settings are automatically filled in. Figure 4.32 shows the configuration of a COM signal.

ComSignal

General Settings

ComSignal shortName: Current

Com System Template System Signal Ref: ☒ Current

Com Transfer Property: ☐ [Dropdown]

Signal Data Settings

Com Bit Position: 0 [Hex]

Com Bit Size: ☒ 16 [Hex]

Com Signal Endianness: LITTLE_ENDIAN [Dropdown]

Com Signal Init Value: ☐ [Text]

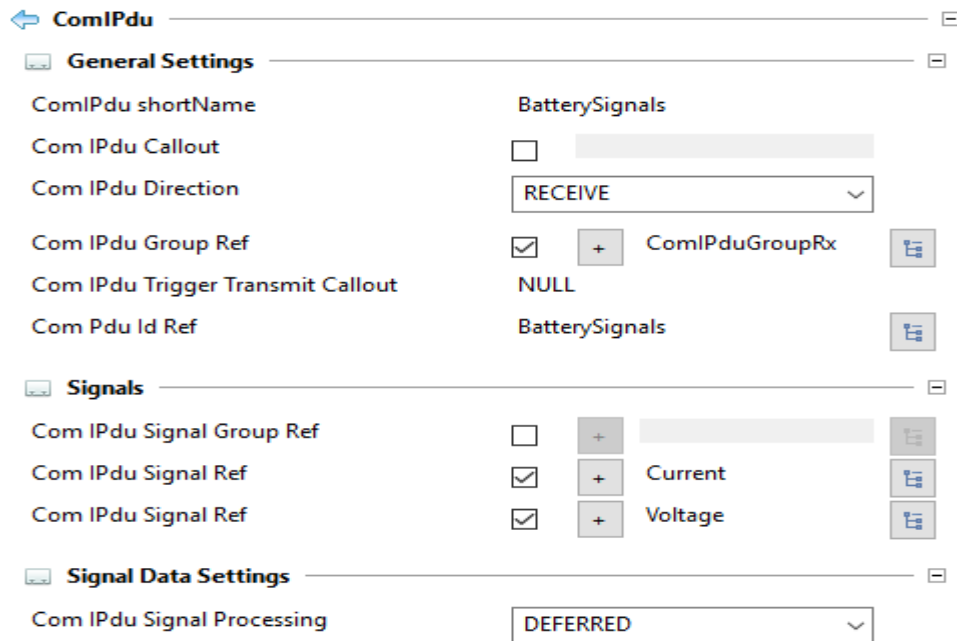
Com Signal Length: ☐ [Text] [Hex]

Com Signal Type: UINT16 [Dropdown]

Com Update Bit Position: ☐ [Text] [Hex]

Figure 4.32: COM Signal Configuration

The I-PDU contains the configuration of data messages that will be received by COM module or that will be sent from the COM module. I-PDUs, in this case, can be seen as a CAN message, each I-PDU must be referred to the COM signal contained in it. Figure 4.33 shows the configuration of BatterySignals I-PDU which is linked to two COM signals Current and Voltage. Each I-PDU has to be referred to the I-PDU group, for our system two IPDU groups were configured one for receive I-PDU and one for transmit I-PDU.



ComIPdu

General Settings

ComIPdu shortName

BatterySignals

Com IPdu Callout

Com IPdu Direction

RECEIVE

Com IPdu Group Ref

☒ + ComIPduGroupRx

Com IPdu Trigger Transmit Callout

NULL

Com Pdu Id Ref

BatterySignals

Signals

Com IPdu Signal Group Ref

☐ +

Com IPdu Signal Ref

☒ + Current

Com IPdu Signal Ref

☒ + Voltage

Signal Data Settings

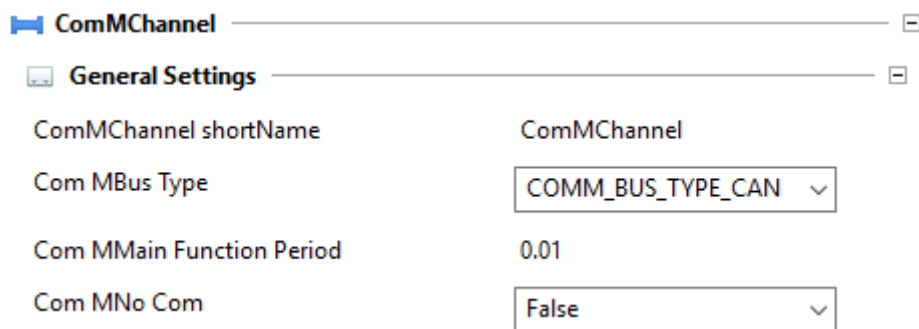
Com IPdu Signal Processing

DEFERRED

Figure 4.33: COM IPDU Configuration

4.3.9 COM Manager

ComM module involves configuration of communication bus channels parameters that are common to the whole communication stack. ComM channels are configured here, indicating the type of bus and how often the communication manager main function should be called. For our system the bus type is CAN and the ComM main function period is set to 10ms. The scheduling of ComM should be at least as fast as the communication stack and according to ComM specification, the time range for scheduling is 1ms to 100ms because of the schedule longer than 100ms makes no sense for communication [32]. Figure 4.34 shows the configuration of ComM communication channel.



ComMChannel

General Settings

ComMChannel shortName

ComMChannel

Com MBus Type

COMM_BUS_TYPE_CAN

Com MMain Function Period

0.01

Com MNo Com

False

Figure 4.34: ComM channel configuration

4.3.10 CAN State Manager

CanSM module is present in the communication services layer along with the ComM module. ComM requests communication modes from the networks and in the case of CAN it uses CanSM module. In ComM configuration, we configured a unique network handle for CAN i.e. ComMChannel, this unique handle has to be referenced to the CanSMNetwork and also the CAN controller managed by the CanIf has to be referred as shown in Figure 4.35.

The image shows two configuration windows. The top window is titled 'CanSMManagerNetwork' and contains a 'General Settings' section with the following fields: 'CanSMManagerNetwork shortName' set to 'CanSMManagerNetwork', 'Can SM Com MNetwork Handle Ref' set to 'ComMChannel', and 'Can SM Transceiver Id' with an unchecked checkbox. The bottom window is titled 'CanSMController' and contains a 'General Settings' section with the following fields: 'CanSMController shortName' set to 'CanSMController' and 'Can SM Controller Id' set to 'CanIfCtrlCfg'.

Figure 4.35: CanSMNetwork and CanSMController Configuration

4.3.11 Basic Software Manager

The main purpose of configuring BswM is to define rules which evaluate a logical expression, and based on the result of evaluation execute different actions. Logical expressions are created by chaining together different mode conditions using the usual logical operators. For our system, BswM was configured for the communication mode switch among ComM full communication, ComM no communication and Start communication condition. Figure 4.36 shows the configuration of mode condition for ComM full communication condition, where the BswComMIndication is linked to the communication channel handle as shown in Figure 4.37.

The image shows the 'BswMModeCondition' configuration window. It has a 'General Settings' section with the following fields: 'BswMModeCondition shortName' set to 'ComMFullCommunicationCondition', 'Bsw MCondition Mode' set to 'BswComMIndication', and 'Bsw MCondition Type' set to 'BSWM_EQUALS'.

Figure 4.36: BswM mode condition configuration

The image shows the 'BswMComMIndication' configuration window. It has a 'General Settings' section with the following fields: 'BswMComMIndication shortName' set to 'BswMComMIndication' and 'Bsw MCom MChannel Ref' set to 'ComMChannel'.

Figure 4.37: BswM ComM Indication

4.3.12 XCP

Configuration of XCP involves: general settings, configuring transmission and reception PDUs and configuring the Event channel. In XCP general setting, the information regarding the maximum CTO, DTO and minimum DAQ indicating the number of predefined DAQ lists are configured. The type of DAQ configuration was set to Dynamic and XCP identification field

type which is important for the slave to transfer DAQ packets to the master and the same identification field is used by the master to transfer STIM packets to the slave, was set to Absolute. This general XCP setting is shown in Figure 4.38.

The screenshot shows the 'XcpGeneral' configuration window. It has a title bar with a minimize, maximize, and close button. Below the title bar is a 'General Settings' section. The settings are as follows:

Property	Value	Buttons
XcpGeneral shortName	XcpGeneral	
Arc Xcp Max Rx Tx Queue	10	Hex
Xcp Counter Ref	OsCounter	Hex
Xcp Daq Config Type	DAQ_DYNAMIC	
Xcp Dev Error Detect	True	
Xcp Identification Field Type	ABSOLUTE	
Xcp Max Cto	8	Hex
Xcp Max Dto	8	Hex
Xcp Min Daq	0	Hex

Figure 4.38: XCP general settings

XCP PDUs for transmission and reception were configured by linking to the PDUs configured in EcuC module (section

4.3.6 EcuC). The configuration of XCP receive PDU is as shown in Figure 4.39.

The screenshot shows the 'XcpRxPdu' configuration window. It has a title bar with a minimize, maximize, and close button. Below the title bar is a 'General Settings' section. The settings are as follows:

Property	Value	Buttons
XcpRxPdu shortName	XcpRxPdu	
Xcp Rx Pdu Ref	XCP_RX_PDU	Hex

Figure 4.39: XCP receive PDU

XCP event channel configuration is shown in Figure 4.40. This involves the setting of the maximum amount of DAQ lists that are handled by this event channel, channel ID, sampling period used for this event channel and the event channel type.

XcpEventChannel

General Settings

XcpEventChannel shortName	XcpEventChannel		
Xcp Event Channel Max Daq List	4		Hex
Xcp Event Channel Number	0		Hex
Xcp Event Channel Time Cycle	1		Hex
Xcp Event Channel Time Unit	<input checked="" type="checkbox"/>	TIMESTAMP_UNIT_10MS	
Xcp Event Channel Triggered Daq List Ref	<input type="checkbox"/>	+	
Xcp Event Channel Type	DAQ_STIM		

Figure 4.40: XCP event channel configuration

4.3.13 Operating System

OS module can be considered as the heart of all BSW modules and it is closely linked to the RTE module. OS configuration involves configuring OS Events, OS Alarms, OS Task and OS Application.

OS Event configuration involves adding the only event in the system; GetPower_TimeEvent a timing event which was configured during SWC modeling. This event will activate the RTE Task. To activate the timing event, an alarm dedicated to this event has to be configured. OsAlarm10ms was created to trigger the GetPower_TimeEvent every 10ms. Figure 4.41 shows the configuration of OsRteTask. Figure 4.42 shows the configuration of timing event and the alarm corresponding to it.

OsTask

General Settings

OsTask shortName	OsRteTask		
Arc Os Task Stack Size	2048		Hex
Os Task Accessing Application	<input checked="" type="checkbox"/>	+	OsApplication
Os Task Activation	2		Hex
Os Task Event Ref	<input checked="" type="checkbox"/>	+	GetPower_TimeEvent
Os Task Priority	2		Hex
Os Task Resource Ref	<input type="checkbox"/>	+	
Os Task Schedule	FULL		

Figure 4.41: OsRteTask Configuration

The code for OsRteTask is auto-generated and is as shown in the code snippet.

```

// RTE Task
void OsRteTask(void) {
    EventMaskType Event;
    do {
        //Waits till the RTE Timing event is received
        SYS_CALL_WaitEvent(EVENT_MASK_GetPower_TimeEvent);
        SYS_CALL_GetEvent(TASK_ID_OsRteTask, &Event);

        //If the right event is received
        if (Event & EVENT_MASK_GetPower_TimeEvent) {
            //Event is cleared for the next iteration
            SYS_CALL_ClearEvent (EVENT_MASK_GetPower_TimeEvent);
            // SWC Runnable is called
            Rte_Power_Get_Power();
        }
    } while (RTE_EXTENDED_TASK_LOOP_CONDITION);
}

```

OsEvent

General Settings

OsEvent shortName: GetPower_TimeEvent

Os Event Mask: ☐ Hex

OsAlarmSetEvent

General Settings

OsAlarmSetEvent shortName: OsAlarmSetEvent

Os Alarm Set Event Ref: GetPower_TimeEvent

Os Alarm Set Event Task Ref: OsRteTask

OsAlarmAutostart

General Settings

OsAlarmAutostart shortName: OsAlarmAutostart

Os Alarm Alarm Time: 10 Hex

Os Alarm Autostart Type: RELATIVE

Os Alarm Cycle Time: 10 Hex

Figure 4.42: Os Event and Os Alarm configuration

The second task configured for our system was OsStartupTask, which runs once and initializes some of the BSW modules. The code for OsStartupTask is as shown below. The PowerUser configured in the EcuM configuration is set to run and some of the BSW modules which are used in the system are initialized by the function EcuM_StartupTwo().

```

void OsStartupTask( void ){
    // The ECUM user PowerUser is requested to RUN
    (void)EcuM_RequestRUN( (EcuM_UserType)ECUM_USER_PowerUser);

    // ECU drivers and BswM,RTE,ComM BSW modules are initialised
    EcuM_StartupTwo();

    // Polling of the CAN controller for mode transitions.
    Can_MainFunction_Mode();

    (void)TerminateTask();
}

```

The other OS task configured was OsXCPTask, which is linked to the alarm OsXcpAlarm which triggers the OsXcpTask every 10ms. This task basically triggers the XCP event channel, the code for OsXcpTask is as shown below.

```

void OsXCPTask ( void ) {

    EcuM_StateType state = ECUM_STATE_OFF;

    // Get current ECU state
    (void)EcuM_GetState(&state);

    if( state <= ECUM_STATE_STARTUP_ONE) {
        (void)TerminateTask();
    }

    if( state <= ECUM_STATE_STARTUP_TWO) {
        (void)TerminateTask();
    }

    // invoke scheduling of the event channel for every 10ms
    Xcp_Arc_WrapperForEventChannel_XcpEventChannel();

    (void)TerminateTask();
}

```

The OsApplication must have all the tasks and alarms linked to it in- order to use the above configured tasks and alarms. Figure 4.43 shows the configuration of OS application.

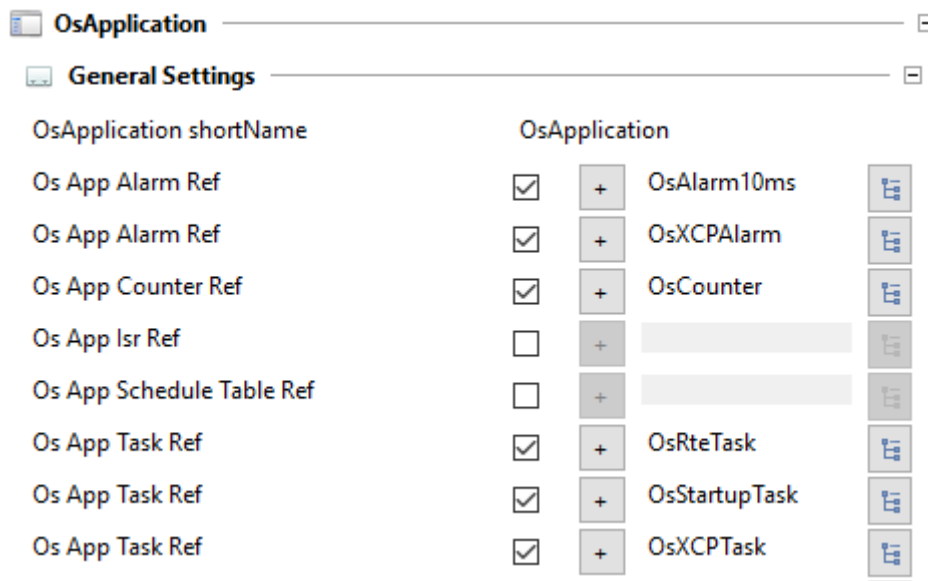


Figure 4.43: OS Application

4.3.14 Run time environment

The configuration of RTE is straight forward in Arctic Studio provided all the runnable, OS tasks and OS events are configured without any errors. The main function of RTE is to give a seamless integration between the BSW layer and the application layer. RTE is added as a BSW module but it can be edited in BSW editor and also in RTE editor. When edited in RTE editor it gives the option of mapping the runnable to its corresponding OS task and OS event. Figure 4.44 shows the mapping of Get_Power runnable to OsRteTask and GetPower_TimeEvent.

Entity	Status	Runnable	Task	Event	Position
▼ Power	Instantiated				
⚙️ GetPower_TimeEvent	Mapped	⚙️ Get_Power	📄 OsRteTask	⚡ GetPower_TimeEvent	

Figure 4.44: RTE Configuration

4.4 Validation and Code generation

Once all the required BSW modules are configured, these modules can be validated individually or together to check whether the configuration is done correctly. The validator indicates if there are any settings that are missing or invalid. If there are no validation errors, then the code can be generated. The same steps are applicable for validating and generating code for RTE, however, the option generate A2L tags has to be selected as shown in Figure 4.45.



Figure 4.45: Generate RTE with A2L Tags

Selecting the option generate A2L Tags generates the same code if this option is not selected, however, by selecting this option the comments are added to the code related to calibration

parameters as shown in code snippet below. When an A2L file is exported from Arctic Studio, it scans through the code and gets the information from these comments.

```
/**
 * @a2l_characteristic Rte_CalPrm_Rom_Power_Power_Cal_GainFactor
 * @desc generated by rte
 * @desc
 * @min 0
 * @max 65535
 * @type uint16
 */
ARC_DECLARE_CALIB(uint16, Rte_CalPrm_Rom_Power_Power_Cal_GainFactor) = 2;
#define Power_STOP_SEC_CALIB_UNSPECIFIED
#include <Power_MemMap.h>
#define Power_START_SEC_VAR_NO_INIT_UNSPECIFIED
#include <Power_MemMap.h>

/**
 * @a2l_characteristic Rte_CalPrm_Ram_Power_Power_Cal_GainFactor
 * @desc generated by rte
 * @desc
 * @min 0
 * @max 65535
 * @type uint16
 */
uint16 Rte_CalPrm_Ram_Power_Power_Cal_GainFactor;
#define Power_STOP_SEC_VAR_NO_INIT_UNSPECIFIED
#include <Power_MemMap.h>
uint16 Rte_CDData_Power_Power_Cal_GainFactor(void) {
    return Rte_CalPrm_Ram_Power_Power_Cal_GainFactor;
}
```

4.4.1 Executable for the MCU

The Code generated from BSW configurator and RTE generator, the source code of application containing runnable, and the Arctic core software from ArcCore to which our project is referenced to, are linked and compiled into a binary executable file in elf format. To compile the software successfully some of the environment variables must be configured in Arctic Studio eclipse environment. The environment variables configured for our project are as shown in the Figure 4.46. The environment variables set here are BDIR referring to the current project path, BOARDDIR set to the MCU used, CROSS_COMPILE referring to the cross compiler used and the PATH referring to eclipse home. This information is used by the makefile to generate the executable in elf format, which is then programmed to the development board using Code composer studio.

Variable	Value
BDIR	\${ProjDirPath}
BOARDDIR	tmdx570ls12hdk
CROSS_COMPILE	/opt/arm-none-eabi/bin/arm-none-eabi-
PATH	\${eclipse_home}\msys\bin

Figure 4.46: Environment Variables

4.4.2 Generating A2L file

There is a provision to export A2L file for a selected project in Arctic Studio. When an A2L file was generated for our project from Arctic Studio, it only contained the information of the measurement variable and calibration parameters. Though the XCP on CAN configuration was done in Arctic Studio, the A2L file did not contain the interface information for XCP on CAN and also the information about the code and data MEMORY_SEGMENT were missing. The A2L file generated was incomplete and the measurement and calibration tool did not recognise this file. When contacted ArcCore, a reply was received saying that the generation of XCP interface related information to an A2L file is not supported by Arctic Studio and such interface information has to be added manually to the A2L file. The missing information was added to our A2L file by using an example A2L file available on the ASAM MCD-2 MC web page as reference.

An A2L file used for MC must contain an A2ML section which defines the description of parameters that describe the communication between MC-System and ECU. The parameters described in A2ML section contains the configuration of the protocol stack and are needed to create messages for measurement and calibration objects such as MEASUREMENT and CHARACTERISTIC. The actual values for the parameters are given in the IF_DATA block and the contents of this block must match the syntax described in A2ML block. The IF_DATA block in our case contained the XCP configuration information such as DAQ, EVENT, and PROTOCOL_LAYER, and most importantly XCP_ON_CAN containing the information of CAN IDs configured for XCP Rx and Tx.

The other important block which is necessary for an A2L file is the MOD_PAR block, which contains the general parameters of the ECU. The data that was necessary to be added in our A2L file under this block are the description of the organisation of the ECUs memory via the keyword MEMORY_SEGMENT. Two memory blocks were described i.e CODE and DATA in our A2L file. Some of the important blocks of our A2L file are shared in the APPENDIX section of this report.

4.5 Measurement and Calibration System Setup

The MC System in our case contains ETAS INCA running on PC and the ETAS ES590.1, ECU and BUS Interface module. The ES590.1 module is connected to TMS570 MCU board via CAN bus and to control ES590.1, it is connected via Ethernet Cable to PC where the INCA software is running. Next step is to create a project in INCA by adding the A2L file containing all the required information and the corresponding data file(binary file). In our case, the binary was in elf format (see 4.4.1 Executable for the MCU) and was converted to motorola srecord format (s19), which is one of the formats that INCA supports. Then, this project is added to a workspace in INCA and linked to the hardware device ES590.1 via XCP on CAN.

4.6 Testing

The main aim of the thesis was to identify a toolchain for ECU function development following AUTOSAR standard and to have an optimal toolchain for measurement and calibration. Now that an example application was developed using SystemDesk and Arctic Studio and the system

extract from both were successfully integrated with the BSW of Arctic Studio, it was time to test the whole system by programming the development board. Testing was performed in two stages first only the AUTOSAR toolchain was tested and finally, the MC software and hardware was included for MC.

4.6.1 Test Setup-1

The first test setup is as shown in block diagram of Figure 4.47. Here the CAN communication and the behaviour of the application was tested by sending the desired CAN message i.e BatterySignals containing signals Current and Voltage. Bus Master was used to configure sending and receiving of CAN messages from the PC. The USB end of the PCAN USB interface was connected to the PC and the other end to the TMS570LS1227 board. During this test, the calibration parameter was kept at its initial value 2, only the signals in BatterySignals message were changed. The output i.e POWER_CALCULATED CAN message is received and the behaviour is verified.

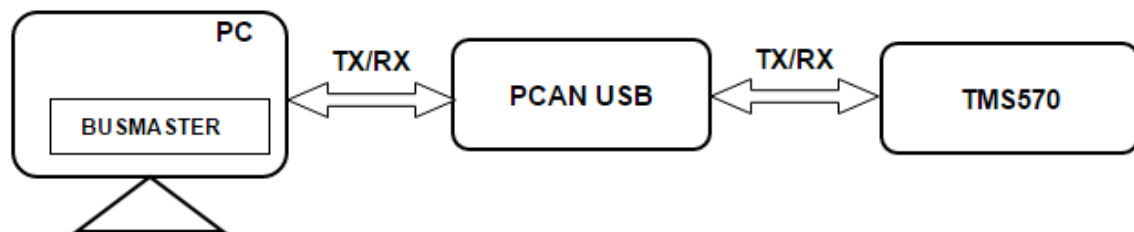


Figure 4.47: Setup 1

4.6.2 Test Setup -2

The second test setup as shown in Figure 4.48 has the MC system connected to the test setup shown in Figure 4.47. This setup is required for measurement of variables and calibration of parameters online. Here the calibration of the parameter was done using INCA and monitor the behaviour of the CAN messages on BUS MASTER and on INCA. Now, two more CAN messages were expected on the CAN bus, which was configured for XCP RX and TX.

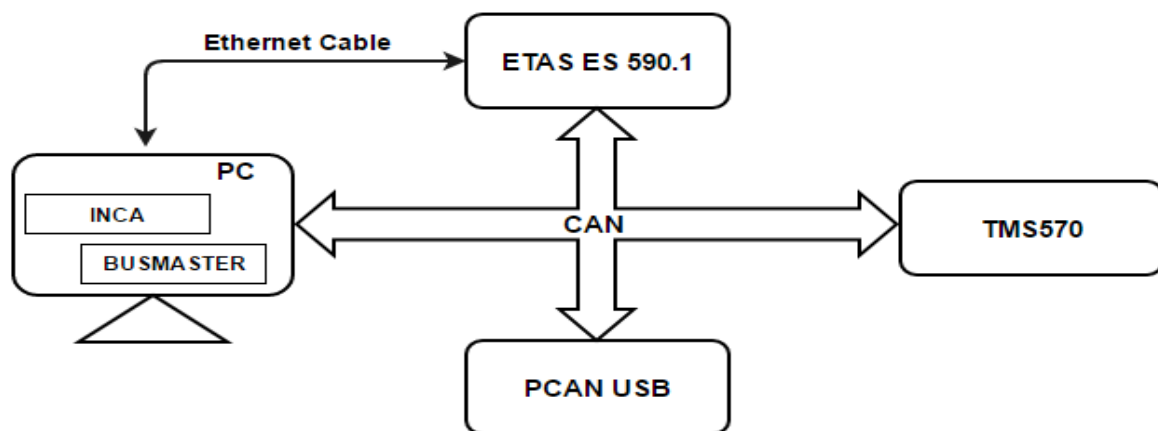


Figure 4.48: Setup-2

5 RESULTS

This section discusses the outcome of this thesis by presenting the two toolchains for AUTOSAR and a complete toolchain involving MC system. The toolchain flow and the results of testing both the two toolchains are also presented in this chapter.

5.1 AUTOSAR Toolchain

Two toolchains for ECU function development following AUTOSAR standard were tried out. The first toolchain only involves Arctic Studio and Arctic Core as it provides the complete AUTOSAR toolchain. In this case, the SWC builder is used for modeling the application layer and the system, extract builder to generate system extract, BSW builder for configuring the BSW modules, and RTE builder to have a continuous interface for communication between the BSW and the application layer. This toolchain is as presented in Figure 5.1.



Figure 5.1: Toolchain-1

The second toolchain shown in Figure 5.2 uses different tools for modeling in the application layer. This involves SystemDesk for modeling SWC, MATLAB/Simulink for implementation of the behaviour of SWCs, and generate the code for runnables with TargetLink. The System extract is exported from SystemDesk to Arctic Studio for BSW configuration with BSW editor and RTE builder.

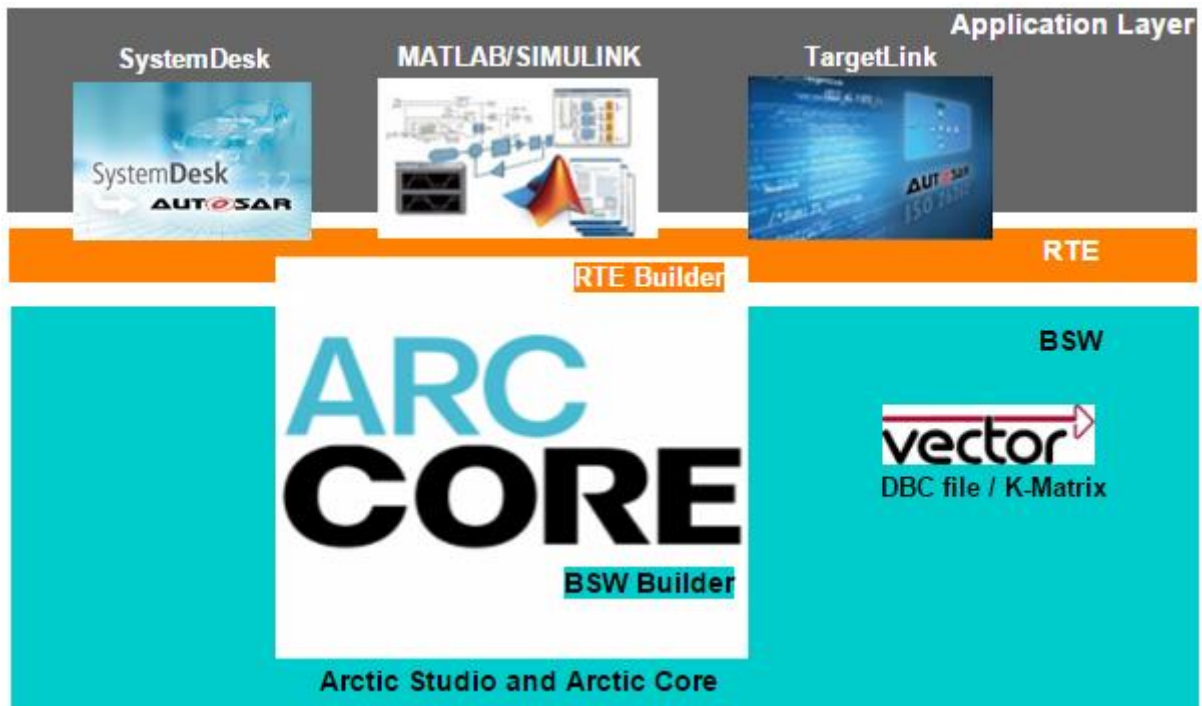


Figure 5.2: Toolchain-2

Tools like Vector CANdb++ editor are needed for creating a network description file which is imported to BSW editor for communication cluster configuration.

5.2 Complete toolchain

The complete toolchain here refers to the AUTOSAR toolchain along with the MC system. This can contain any of the two AUTOSAR toolchains presented in section 5.1 AUTOSAR Toolchain. The complete toolchain shown in Figure 5.3 has the ETAS bus and ECU interface and INCA tool in addition to toolchain 1.

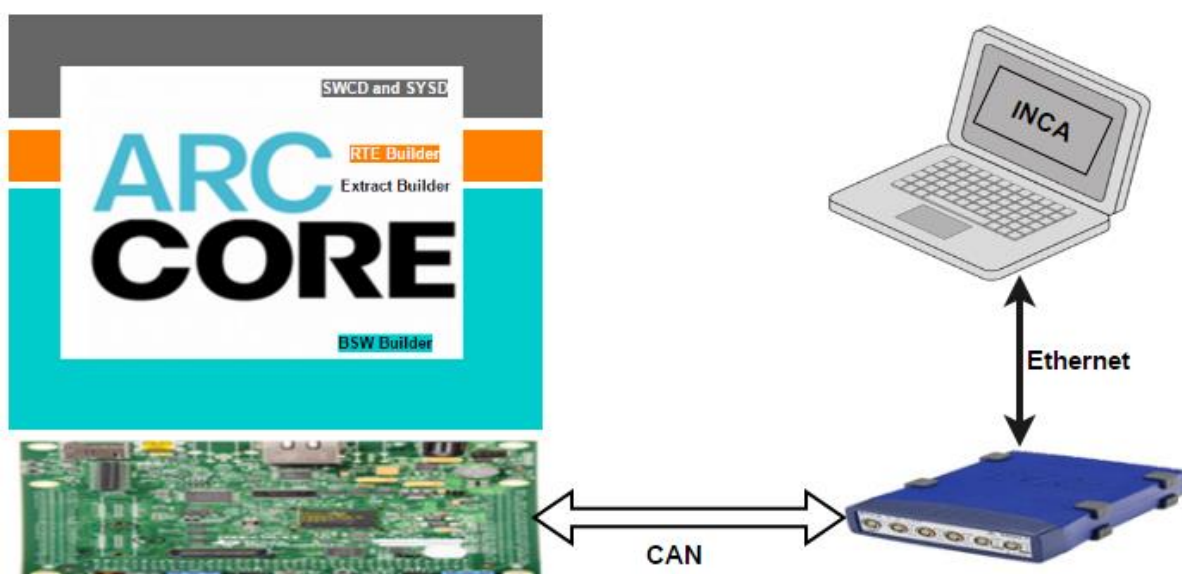


Figure 5.3: Complete toolchain

5.3 Toolchain flow

Figure 5.4 shows the complete flow of both the toolchains, interdependencies between different tools, and the artefacts generated and used by the tools.

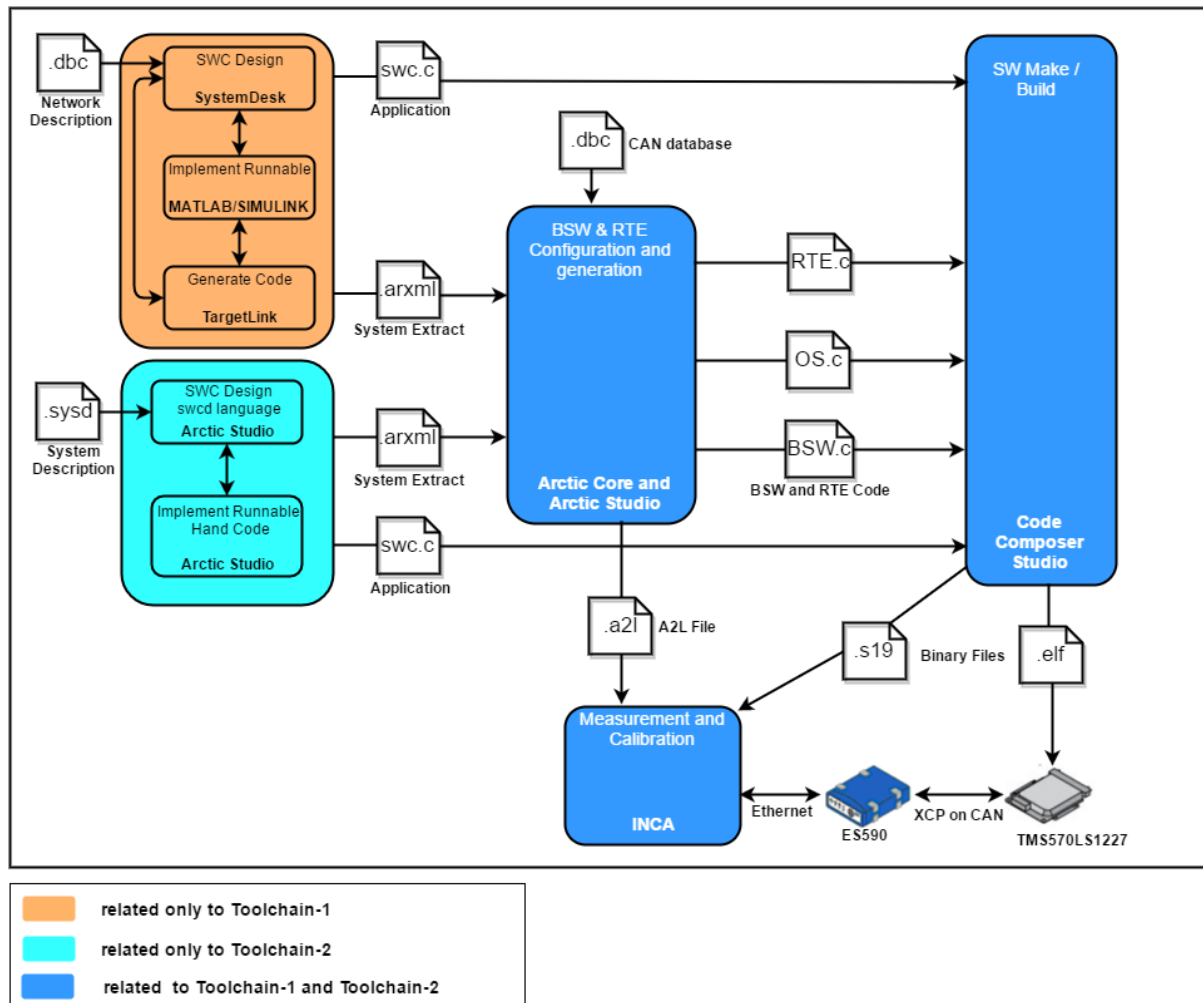


Figure 5.4: Toolchain flow and artefacts at different stages

Figure 5.5 shows table depicting the type of license used for software in both the tool chains.

Purpose	Software	Available License Type	License used
SWC Desgin	dSpace - SystemDesk	Commercial	Evaluation License
	Arctic Studio - SWC builder	GPL & Commercial	GPL
Implemet SWC runnable	MATLAB/SIMULINK	Commercial	Student license
	Arctic Studio - Eclipse IDE	GPL & Commercial	GPL
Generate C Code	dSpace - TargetLink	Commercial	Evaluation License
BSW & RTE Configuration and Generation	Arctic Studio and Arctic Core BSW Configurator	GPL & Commercial	GPL
Make and Build SW	Code Composer Studio	GPL & Commercial	GPL
Measurement and Calibration	ETAS -INCA	Commercial	Commercial

Figure 5.5:License used for different Softwares

5.4 ECU software architecture

In this thesis, the ECU software was developed to have AUTOSAR system architecture. The final software architecture contains an SWC Power, Basic software configured for CAN communication, and the MCAL for TMS570 microcontroller board. The final software architecture is as shown in Figure 5.6.

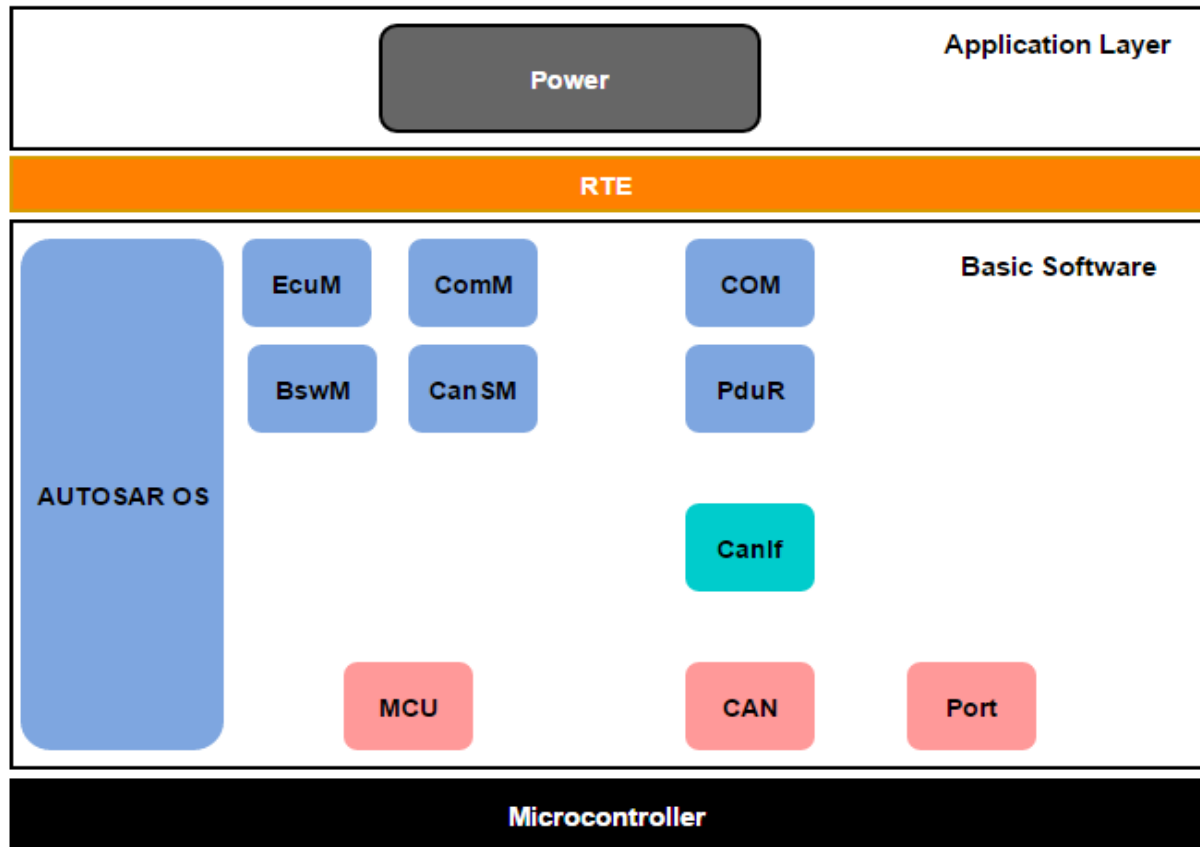


Figure 5.6: ECU Software Architecture

5.5 Testing

The testing was conducted in two steps as mentioned in section 4.5 Measurement and Calibration System Setup. In this section, the sample inputs with which the system was tested and the output obtained will be discussed.

5.5.1 Testing with setup 1

The BatterySignals CAN message was configured in BUS MASTER and sent to the development board where the software is running. Two CAN messages for BatterySignals were configured to be sent at every 1700ms and 1900ms. The CAN frame sent every 1700ms had the Current signal value set to 6A and Voltage signal to 2V and the CAN frame sent every 1900ms had the Current signal value set to 11 A (0xB) and 4V. The calibration parameter GainFactor was unchanged at initial value 2. As expected the output CAN message POWER_CALCULATED had the Power Signal with value 24 W and 88 W at every 1700ms and 1900ms respectively. This ensured the working of both the application and the CAN

communication. The Signal watch window for both the CAN messages is as shown in Figure 5.7.

Signal Watch - CAN			
Message	Signal	Physical Value	Raw Value
POWER_CALCULATED	Power	24 W	18
BatterySignals	Voltage	2 V	2
BatterySignals	Current	6 A	6

Signal Watch - CAN			
Message	Signal	Physical Value	Raw Value
POWER_CALCULATED	Power	88 W	58
BatterySignals	Voltage	4 V	4
BatterySignals	Current	11 A	B

Figure 5.7: CAN Signal Watch

The signal graph in Figure 5.8 shows the behaviour of signals for 10 seconds and the signal colouring is represented in Figure 5.9.

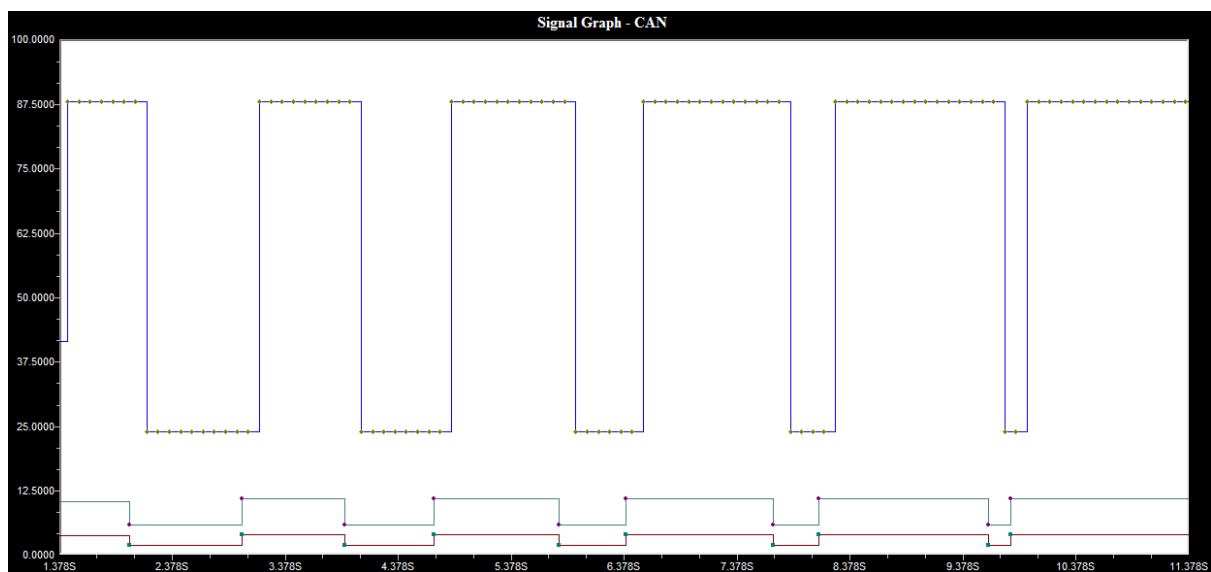


Figure 5.8: CAN Signal Graph

Element List	
Category	Element
BatterySignals	Current
BatterySignals	Voltage
POWER_CALCULATED	Power

Figure 5.9: Signals related to graph

5.5.2 Testing with setup -2

The same test signals as in test setup 1 were used here, but the main aim here was to change the value of the calibration parameter through INCA. Once the hardware is initialised, the CAN IDs 0x3E8 and 0x3E9 which were configured for XCP to exchange CTOs and DTOs were seen on the CAN bus. Figure 5.10 shows these CAN messages from the BUSMASTER window.

Message Window - CAN						
Tx/Rx ...	Channel...	Msg ...	ID ...	Message	DLC...	Data Byte(s)
Rx	1	s	0x001	POWER_CALCULATED	8	58 00 00 00 00 00 00 00
Tx	1	s	0x002	BatterySignals	8	0B 00 04 00 00 00 00 00
Rx	1	s	0x3E8	0x3E8	8	EB 03 00 00 00 00 00 00
Rx	1	s	0x3E9	0x3E9	8	FE 20 40 08 00 08 01 01

Figure 5.10: XCP CAN messages

The Calibration parameter GainFactor was added to the calibration window and when changed, the value of the POWER_CALCULATED signal also changed. Figure 5.11 shows one such test input, where the parameter was set to 8 and the CAN signals Current and Voltage are unchanged from the test setup 1.

Measure Window [1]	
Current	6.00 [A]
Power	96.00 [W]
Voltage	2.00 [V]

Calibration Window [1]	
Rte_CalPrm_Rom_Power_Power_Cal_GainFactor	8.000000000000000

Measure Window [1]	
Current	11.00 [A]
Power	352.00 [W]
Voltage	4.00 [V]

Calibration Window [1]	
Rte_CalPrm_Rom_Power_Power_Cal_GainFactor	8.000000000000000

Figure 5.11: Calibration window INCA

By setting different values to GainFactor from the calibration window, the changes in the output were seen during runtime. This is depicted in Figure 5.12, where the GainFactor was changed from 8 to 10 to 5.

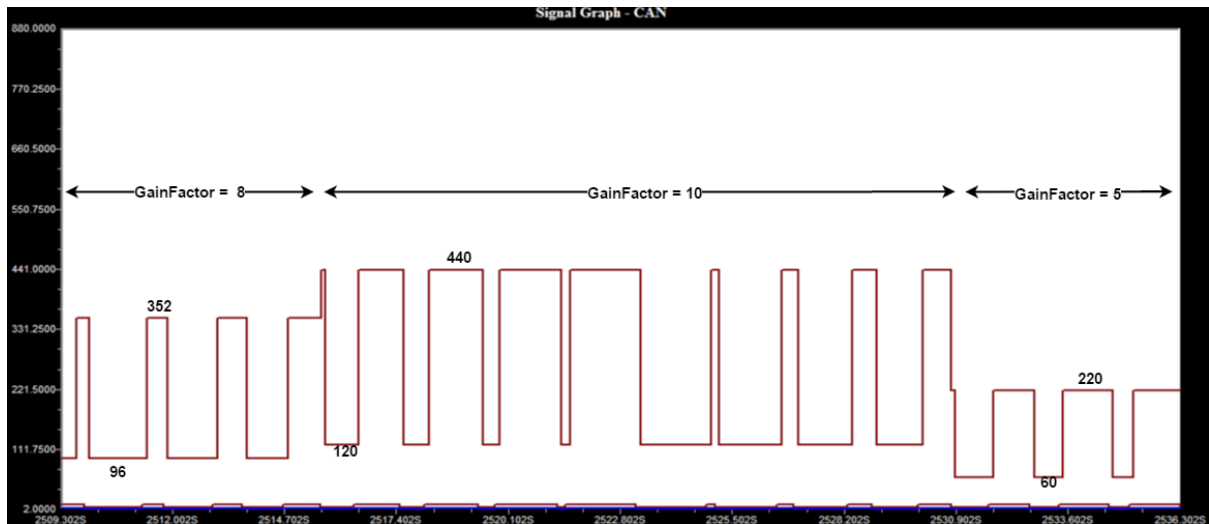


Figure 5.12: Signal Graph - Online Calibration

The actual setup of the project is as shown in Figure 5.13. It consists of a supply to ETAS ES590.1 module, Ethernet cable connected from ES590.1 to the PC and the CAN cable from ES590.1 connected to the ECU and PEAK CAN USB.



Figure 5.13: Project Setup

6 DISCUSSION AND CONCLUSION

The discussion on overall project method will be presented in this chapter. Here, the conclusion of this thesis will be presented along with the scope for future work.

6.1 Project Method

The overall project method suited well for this thesis. With no prior knowledge on AUTOSAR and ECU function development, the theoretical pre-study of AUTOSAR followed by contacting the tool vendors, and practical pre-study of using different AUTOSAR tools resulted in a sound understanding of various concepts in AUTOSAR software architecture. AUTOSAR architecture is complex as well as its methodology, however, the methodology was simplified to some extent for the system considered in this thesis. Nevertheless, fourteen BSW modules were used in the thesis and at times it was difficult to realise the sequence in which the modules had to be configured. Initially, deciding the order of methodology steps was also a challenge, as the AUTOSAR methodology does not specify a strict order to follow. Implementing the application layer in both Arctic Studio and SystemDesk, helped in understanding the concepts of SWC modelling in detail. Developing the SWC in SystemDesk and exporting the ECU extract artefact in arxml format, and integrating this artefact with the BSW configurator of Arctic Studio, facilitated in realizing the abstraction between different layers of AUTOSAR.

The BSW modules configuration started with configuring mandatory modules such as OS and MCU, CAN cluster configuration, then configuring related modules in services layer, and finally the XCP module. As Arctic Studio doesn't come with a debugger, the project was imported into Code composer studio for deploying the executable to the development board and for debugging purposes. The A2L file generated from Arctic Studio did not contain the XCP on CAN interface and the memory segment information, thus the A2L file had to be modified manually to establish the communication between, ECU and the MC system. As there is little information about how an A2L file has to be written, manually editing A2L file took a considerable amount of time.

Although the project flow suited this thesis, there were a few drawbacks to it. There were no pre-defined test mechanisms for every stage of the project, this led to move back in the workflow when there was an error, which could have been corrected if tested in earlier stages. The A2L file, which was important to achieve online calibration had to be edited manually as the tools from the current toolchain did not generate the complete file.

6.2 Conclusion

The purpose of this thesis specified in chapter 1 has been fulfilled. Two toolchains are proposed in this thesis for ECU function development with AUTOSAR. The difference between the two is in the application layer development. Although two toolchains are proposed, a toolchain has to be selected based on the project requirement, i.e. if the application is developed by writing C/C++ code manually, toolchain-1 is most suitable. On the other hand, to develop complex functionalities, graphical programming tool such as Simulink is most commonly used. Complex functions generally are divided into many sub-functions, thus many SWCs in the

application layer. For such a system, toolchain-1 is not a suitable option because creating SWCs using SWCD and SYSD language will be a challenge for a system with many SWCs. Toolchain-2 will be the most suitable option for systems with many SWCs, because of the graphical interface provided by SystemDesk which is easy to maintain and update SWCs. However, toolchain-2 is most suited when the model-based function development approach is used.

The complete toolchain containing MC system along with any of toolchain-1 or toolchain-2 is also proposed and online calibration is tested. The MC part of the tool chain is simpler and easy to configure, if and only if a valid A2L file and corresponding data file (binary) are available. XCP on CAN communication is used to achieve calibration, which is more effective in terms of data transfer and hence commonly used in automotive industry to read measurement data and to write parameters to ECUs during development, testing and in-vehicle calibration.

This thesis work can also be used to learn the ECU function development following AUTOSAR standard and online calibration. The overall structure of the report is kept simple and it is well suited for future expansion.

6.3 Future work

Considering the purpose of this thesis, the application developed in this work has no practical use in the car. Nevertheless, the framework to develop an ECU function is proposed and this can be used in many ways. For example, the simple application layer developed in this thesis can be replaced with the Battery management system application. The CAN stack configured in this work can serve as an example to configure CAN stack for any other ECU. The measurement and calibration mechanism proposed can be used for calibrating any ECU. Only XCP on CAN is configured in this thesis, which can have transmission rates up to 1Mbit/s. To achieve higher transmission rates, other communication protocols i.e. XCP on Ethernet and XCP on FlexRay can be configured. For example, to establish XCP on Ethernet, Ethernet port should be added to the port BSW module and other modules such as Ethernet Interface, Ethernet State Manager, TCP/IP, and Socket Adapter in the communication stack has to be configured to achieve successful Ethernet communication.

BIBLIOGRAPHY

- [1] NEVS, "Home," 2016. [Online]. Available: <https://www.nevs.com/> [Accessed 20 August 2016].
- [2] Simon Fürst, "AUTOSAR – An open standardized software architecture for the automotive industry," in *1st AUTOSAR Open Conference & 8th AUTOSAR Premium Member Conference*, Detroit, MI, USA, 2008.
- [3] AUTOSAR, "Home," AUTOSAR, 2014. [Online]. Available: <https://www.autosar.org/>. [Accessed 6 July 2016].
- [4] AUTOSAR, "Motivation & Goals," AUTOSAR, 2014. [Online]. Available: <https://www.autosar.org/about/basics/motivation-goals/>. [Accessed 6 July 2016].
- [5] AUTOSAR, "Basics," 2014. [Online]. Available: <https://www.autosar.org/about/basics/>. [Accessed 6 July 2016].
- [6] AUTOSAR, "Layered Software Architecture 4.2.2," AUTOSAR, 2015.
- [7] AUTOSAR, "Glossary," AUTOSAR, 2015.
- [8] AUTOSAR, "Virtual Function Bus," AUTOSAR, 2015.
- [9] AUTOSAR, "Specification of RTE," AUTOSAR, 2015.
- [10] Robert Warschofsky, "AUTOSAR Software Architecture," Hasso-Plattner-Institute für Softwaresystemtechnik, Potsdam, 2009.
- [11] AUTOSAR, "List of Basic Software Modules," AUTOSAR, 2015.
- [12] BOSCH, "CAN Specification Versio 2.0," Robert Bosch Gmbh, Stuttgart, 1991.
- [13] AUTOSAR, "Specification of PDU Router," AUTOSAR, 2015.
- [14] Nico. Naumann, "AUTOSAR Runtime Environment and Virtual Function Bus," Department for System Analysis and Modeling, Hasso-Plattner Institute für Softwaresystemtechnik, Potsdam, 2009.
- [15] Johannes. Gosda, "AUTOSAR Communication Stack," Hasso-Plattner Institute, Potsdam, 2009.
- [16] AUTOSAR, "Sepcification of Communication," AUTOSAR, 2015.
- [17] AUTOSAR, "Specification of CAN Interface," AUTOSAR, 2015.
- [18] AUTOSAR, "Specification of CAN Driver," AUTOSAR, 2015.
- [19] OSEK/VDX, "Operating System Specification," OSEK, 2005.

- [20] Syama R and Devika K, "An Overview of AUTOSAR Multicore Operating System Implementation," *International Journal of Innovative Research in Science, Engineering and Technology*, vol. 2, no. 7, pp. 3163-3169, 2013.
- [21] AUTOSAR, "Specification of ECU State Manager," AUTOSAR, 2015.
- [22] AUTOSAR, "Specification of ECU State Manager with the fixed state machine," AUTOSAR, 2015.
- [23] ASAM, "ASAM Connects - Standard detail," ASAM, May 2015. [Online]. Available: http://www.asam.net/nc/home/standards/standard-detail.html?tx_rbwmbmasamstandards_pi1%5BshowUid%5D=3144. [Accessed July 2016].
- [24] ASAM Wiki, "ASAM MCD-1 XCP - Standards," ASAM, May 2015. [Online]. Available: <https://wiki.asam.net/display/STANDARDS/ASAM+MCD-1+XCP>. [Accessed July 2016].
- [25] VECTOR gmbh, "XCP - The Standard Protocol for ECU Development," Vector Informatik, 2015.
- [26] ASAM, "XCP Part 2 - Protocol Layer Specification," ASAM, 2008.
- [27] ASAM, "The Universal Measurement and Calibration Protocol Family - Part1 - Overview," ASAM, 2003.
- [28] AUTOSAR, "Specification of Module XCP," AUTOSAR, 2015.
- [29] Jacob. Axelsson, "AUTOSAR Overview," in *FESA workshop at KTH*, Stockholm, 2010.
- [30] AUTOSAR, "Methodology," AUTOSAR, 2015.
- [31] AUTOSAR, "Specification of CAN State Manager," AUTOSAR, 2015.
- [32] AUTOSAR, "Specification of Communication Manager," AUTOSAR, 2015.
- [33] AUTOSAR, "Specification of MCU Driver," AUTOSAR, 2015.
- [34] AUTOSAR, "Specification of Port Driver," AUTOSAR, 2015.
- [35] ArcCore, "Arctic Studio Development Tools," ArcCore, 2016. [Online]. Available: <https://www.arccore.com/products/arctic-studio>. [Accessed 23 August 2016].
- [36] ArcCore, "Arctic Core - the Embedded Platform," ArcCore, [Online]. Available: <https://www.arccore.com/products/arctic-core>. [Accessed 23 8 2016].
- [37] dSPACE, "SystemDesk," dSPACE, 2016. [Online]. Available: https://www.dspace.com/en/inc/home/products/sw/system_architecture_software/system_desk.cfm. [Accessed 23 08 2016].
- [38] dSPACE, "SystemDesk," dSPACE, 2016. [Online]. Available: <https://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm>. [Accessed 23 08 2016].

- [39] ETAS, "INCA Base Product," ETAS, 2016. [Online]. Available: <http://www.etas.com/en/products/inca.php>. [Accessed 23 08 2016].
- [40] MathWorks, "Simulink - Simulation and Model based design," MathWorks, 2016. [Online]. Available: www.mathworks.com/products/simulink/index.html. [Accessed 23 08 2016].
- [41] Texas Instruments, "Code Composer Studio (CCS) Integrated Development Environment (IDE)," 2016. [Online]. Available: <http://www.ti.com/tool/ccstudio>. [Accessed 23 08 2016].
- [42] Vector InformatiK Gmbh, "DBC communication Database for CAN," 2016. [Online]. Available: http://vector.com/vi_candb_en.html. [Accessed 23 08 2016].
- [43] BOSCH and ETAS, "BUS MASTER," 2016. [Online]. Available: <https://rbei-etlas.github.io/BusMaster/>. [Accessed 23 08 2016].
- [44] ETAS, "BUS MASTER," 2016. [Online]. Available: http://www.etas.com/en/products/applications_BusMaster.php. [Accessed 23 08 2016].
- [45] Texas Instruments, TMS570LS1227 HDK - Users Guide, Texas Instruments, 2013.
- [46] Texas Instruments, TMS570LS1227 16 and 32-bit RISC Flash Microcontroller, Texas Instruments, 2015.
- [47] Peak System, "PCAN - USB," Peak System, 2016. [Online]. Available: <http://www.peak-system.com/PCAN-USB.199.0.html?L=1>. [Accessed 23 08 2016].
- [48] ETAS, "ES59X - Universal Interface Modules," ETAS, 2016. [Online]. Available: <http://www.etas.com/en/products/es59x.php>. [Accessed 23 08 2016].
- [49] ASAM, "ASAM MCD-2 MC," ASAM, 29 Jun 2015. [Online]. Available: <https://wiki.asam.net/display/STANDARDS/ASAM+MCD-2+MC>. [Accessed 2016 August 20].

LIST OF FIGURES

Figure 2.1: AUTOSAR Architecture.....	4
Figure 2.2: Example of a sender-receiver communication in the VFB view	6
Figure 2.3: Client - server communication in the VFB view	6
Figure 2.4: Example of VFB to RTE Mapping.....	7
Figure 2.5: Overview of the AUTOSAR Interfaces	8
Figure 2.6: Basic Software Layer	9
Figure 2.7: Overview of BSW function group and modules.....	10
Figure 2.8: CAN Frame	11
Figure 2.9: Communication Stack related BSW Functional groups	11
Figure 2.10: AUTOSAR Communication Stack.....	12
Figure 2.11: CAN Communication Stack	13
Figure 2.12: I-PDU ID example	14
Figure 2.13: PDU over different Layers	14
Figure 2.14: Interaction of Layers	15
Figure 2.15: CAN Hardware unit with two CAN controllers.....	17
Figure 2.16: Task model: Basic and Extended Tasks	18
Figure 2.17: ECU Main states	19
Figure 2.18: XCP Packet	21
Figure 2.19: XCP Communication Model with CTO/DTO	22
Figure 2.20: The CTO packet	23
Figure 2.21: The DTO Packet.....	23
Figure 2.22: Modes of XCP protocol	24
Figure 2.23: XCP on CAN Message	25
Figure 2.24: DAQ list with 3 ODTs	26
Figure 2.25: ODT: Allotment of RAM addresses to DAQ-DTO	26
Figure 2.26: Sequence of using commands for allocating DAQs dynamically	27
Figure 2.27: AUTOSAR XCP	28
Figure 2.28: AUTOSAR System Design Methodology.....	30
Figure 3.1: TMS50LS1227 HDK Board	35
Figure 3.2: Connectors on TMS570LS1227	36
Figure 4.1: System architecture	37
Figure 4.2: Folder Structure in SystemDesk.....	40
Figure 4.3: Application SWC with ports	41
Figure 4.4: Application Data Types and Interfaces	41
Figure 4.5: After assigning the interfaces to the port.....	41
Figure 4.6: Internal Behaviour and Data type Mapping Set	42
Figure 4.7: Runnable entity dialogue	42
Figure 4.8: SWC implementation dialogue	43
Figure 4.9: Generated Frame model.....	43
Figure 4.10: Get_Power Runnable & Properties.....	44
Figure 4.11: Get_Power behaviour implementation	44
Figure 4.12: TargetLink Main Dialogue	45
Figure 4.13: CAN Network topology	47
Figure 4.14: Signals	47
Figure 4.15: System.....	48

Figure 4.16: System - Data Mappings	48
Figure 4.17: Used BSW Modules.....	49
Figure 4.18: MCU Clock Reference Point	49
Figure 4.19: EcuM Fixed Configuration.....	50
Figure 4.20: EcuM Sleep mode	50
Figure 4.21: Port Configuration for CAN_TX.....	51
Figure 4.22: Port Configuration for CAN_RX.....	51
Figure 4.23: CAN Controller Configuration	52
Figure 4.24: CAN Hardware object configuration	52
Figure 4.25: Configured CanIfPDUs.....	53
Figure 4.26: CanIfTxPdu Configuration	53
Figure 4.27: ECU PDU Collection	54
Figure 4.28: One PDU flowing through the COM- Stack	54
Figure 4.29: PDUR BSW modules	55
Figure 4.30: Destination of BatterySignals w.r.t PDUR	55
Figure 4.31: Source of BatterySignals w.r.t PDUR.....	55
Figure 4.32: COM Signal Configuration.....	56
Figure 4.33: COM IPDU Configuration	57
Figure 4.34: ComM channel configuration	57
Figure 4.35: CanSMNetwork and CanSMController Configuration	58
Figure 4.36: BswM mode condition configuration	58
Figure 4.37: BswM ComM Indication.....	58
Figure 4.38: XCP general settings.....	59
Figure 4.39: XCP receive PDU.....	59
Figure 4.40: XCP event channel configuration	60
Figure 4.41: OsRteTask Configuration	60
Figure 4.42: Os Event and Os Alarm configuration	61
Figure 4.43: OS Application.....	63
Figure 4.44: RTE Configuration	63
Figure 4.45: Generate RTE with A2L Tags	63
Figure 4.46: Environment Variables.....	64
Figure 4.47: Setup 1	66
Figure 4.48: Setup-2	66
Figure 5.1: Toolchain-1	67
Figure 5.2: Toolchain-2	68
Figure 5.3: Complete toolchain.....	68
Figure 5.4: Toolchain flow and artefacts at different stages	69
Figure 5.5: License used for different Softwares	69
Figure 5.6: ECU Software Architecture	70
Figure 5.7: CAN Signal Watch.....	71
Figure 5.8: CAN Signal Graph	71
Figure 5.9: Signals related to graph	71
Figure 5.10: XCP CAN messages.....	72
Figure 5.11: Calibration window INCA	72
Figure 5.12: Signal Graph - Online Calibration.....	73
Figure 5.13: Project Setup.....	73

A2L File

The important blocks of the A2L file configured for this thesis is shared in this part.

MOD_PAR containing MEMORY_SEGMENT

```
/begin MOD_PAR
  "For TMS570LS1227"
  /begin MEMORY_SEGMENT_ECU_CODE
    "Code Area"
    CODE      /* Program type */
    FLASH     /* Memory type */
    INTERN    /* Memory Attribute */
    0x00000000 /* Start address */
    0x00010940 /* Size */
    -1 -1 -1 -1 -1 /* Offsets */

    /begin IF_DATA_XCP

    /begin SEGMENT
      0x1      /* Segment logical number */
      0x1      /* number of pages */
      0x0      /* address extension */
      0x0      /* compression method */
      0x0      /* encryption method */

      /begin CHECKSUM
        XCP_ADD_11
        MAX_BLOCK_SIZE 0xFF
      /end CHECKSUM

      /begin PAGE
        0x0 /* page number */
        ECU_ACCESS_WITH_XCP_ONLY
        XCP_READ_ACCESS_WITH_ECU_ONLY
        XCP_WRITE_ACCESS_NOT_ALLOWED
      /end PAGE

      /end SEGMENT
    /end IF_DATA_XCP
  /end MEMORY_SEGMENT_ECU_CODE

  /begin MEMORY_SEGMENT_ROM
    "Calibrations"
    DATA
    FLASH
    INTERN
    0x00020000
    0x00000004
```

```

-1 -1 -1 -1 -1
/begin IF_DATA XCP

/begin SEGMENT
0x0    /* Segment logical number */
0x2    /* number of pages */
0x0    /* address extension */
0x0    /* compression method */
0x0    /* encryption method */

/begin CHECKSUM
XCP_ADD_11
MAX_BLOCK_SIZE 0xFF
/end CHECKSUM

/begin PAGE
0x0
ECU_ACCESS_WITH_XCP_ONLY
XCP_READ_ACCESS_WITH_ECU_ONLY
XCP_WRITE_ACCESS_NOT_ALLOWED
/end PAGE

/begin PAGE
0x1
ECU_ACCESS_WITH_XCP_ONLY
XCP_READ_ACCESS_WITH_ECU_ONLY
XCP_WRITE_ACCESS_WITH_ECU_ONLY
/end PAGE

/begin ADDRESS_MAPPING
0x00020000
0x08000cd0
0x00000004
/end ADDRESS_MAPPING

/end SEGMENT
/end IF_DATA
/end MEMORY_SEGMENT
/end MOD_PAR

```

CHARACTERISTIC Block containing details of the Calibration Parameter.

```
/begin CHARACTERISTIC
/* Name      */ Rte_CalPrm_Rom_Power_Power_Cal_GainFactor
/* LongIdentifier */ "generated by rte "
/* Type      */ VALUE
/* Address    */ 0x00020000
/* Deposit    */ __UWORD_Z
/* MaxDiff    */ 0
/* Conversion */ NO_COMPU_METHOD
/* LowerLimit */ 0
/* UpperLimit */ 65535
ECU_ADDRESS_EXTENSION 0x0
EXTENDED_LIMITS      0 65535
FORMAT                "%.15"
/begin IF_DATA CANAPE_EXT
100
DISPLAY 0 0 255
/end IF_DATA
/end CHARACTERISTIC
```

IF_DATA containing XCP and XCP ON CAN configuration information.

```
/begin IF_DATA XCP
/begin PROTOCOL_LAYER
0x100 /* XCP Protocol Layer 1.0 */
400 400 400 400 400 80 3200
0x8 /* MAX_CTO */
0x8 /* MAX_DTO */
BYTE_ORDER_MSB_FIRST
ADDRESS_GRANULARITY_BYTE
OPTIONAL_CMD GET_ID
OPTIONAL_CMD SET_REQUEST
OPTIONAL_CMD GET_SEED
OPTIONAL_CMD UNLOCK
OPTIONAL_CMD SET_MTA
OPTIONAL_CMD UPLOAD
OPTIONAL_CMD SHORT_UPLOAD
OPTIONAL_CMD BUILD_CHECKSUM
OPTIONAL_CMD DOWNLOAD
OPTIONAL_CMD SHORT_DOWNLOAD
OPTIONAL_CMD SET_CAL_PAGE
OPTIONAL_CMD GET_CAL_PAGE
OPTIONAL_CMD COPY_CAL_PAGE
OPTIONAL_CMD CLEAR_DAQ_LIST
OPTIONAL_CMD SET_DAQ_PTR
OPTIONAL_CMD WRITE_DAQ
OPTIONAL_CMD SET_DAQ_LIST_MODE
OPTIONAL_CMD START_STOP_DAQ_LIST
OPTIONAL_CMD START_STOP_SYNCH
OPTIONAL_CMD GET_DAQ_CLOCK
/end PROTOCOL_LAYER
```

```

/begin DAQ
    DYNAMIC      /* DAQ_CONFIG_TYPE */
    0x4          /* MAX_DAQ */
    0x3          /* MAX_EVENT_CHANNEL */
    0x0          /* MIN_DAQ */
    OPTIMISATION_TYPE_ODT_TYPE_32
    ADDRESS_EXTENSION_FREE
    IDENTIFICATION_FIELD_TYPE_ABSOLUTE
    GRANULARITY_ODT_ENTRY_SIZE_DAQ_BYTE
    0x4          /* MAX_ODT_ENTRY_SIZE_DAQ */
    NO_OVERLOAD_INDICATION
    PRESCALER_SUPPORTED
    RESUME_SUPPORTED

/begin EVENT
    "XcpEventChannel" /* name */
    "XcpEvent"        /* short name */
    0x0               /* Event */
    DAQ_STIM          /* Type */
    0x4               /* MAX_DAQ_LIST */
    0xA               /* TIME_CYCLE */
    0x4               /* TIME_UNIT */
    0x0               /* PRIORITY */
/end EVENT

/end DAQ

/begin XCP_ON_CAN
    0x100            /* XCP on CAN 1.0 */
    CAN_ID_MASTER 0x3E8 /* XCP_RX ID */
    CAN_ID_SLAVE 0x3E9 /* XCP_TX ID */
    BAUDRATE 500000
/begin PROTOCOL_LAYER
    0x100
    400 400 400 400 400 80 3200
    0x8
    0x8
    BYTE_ORDER_MSB_FIRST
    ADDRESS_GRANULARITY_BYTE
    OPTIONAL_CMD FREE_DAQ
    OPTIONAL_CMD ALLOC_DAQ
    OPTIONAL_CMD ALLOC_ODT
    OPTIONAL_CMD ALLOC_ODT_ENTRY
    OPTIONAL_CMD SET_MTA
    OPTIONAL_CMD UPLOAD
    OPTIONAL_CMD SHORT_UPLOAD
    OPTIONAL_CMD BUILD_CHECKSUM
    OPTIONAL_CMD DOWNLOAD
    OPTIONAL_CMD SHORT_DOWNLOAD
    OPTIONAL_CMD SET_CAL_PAGE

```

```
    OPTIONAL_CMD GET_CAL_PAGE
    OPTIONAL_CMD COPY_CAL_PAGE
    OPTIONAL_CMD START_STOP_DAQ_LIST
    OPTIONAL_CMD START_STOP_SYNCH
  /end PROTOCOL_LAYER
/end XCP_ON_CAN
/end IF_DATA
```

TRITA TRITA-ICT-EX-2016:155