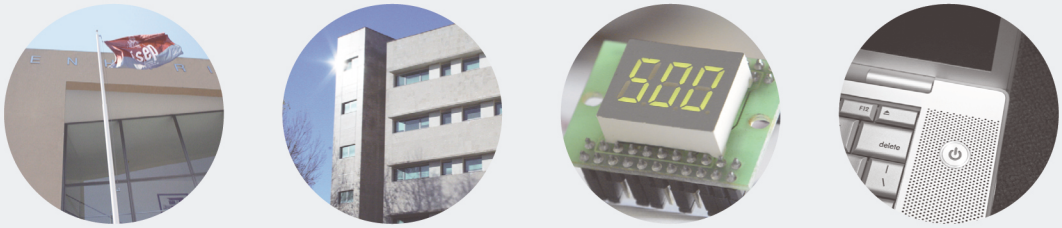
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES



Application Software Components in Autosar

THOR AXEL ACHIM HEINRICH JOSÉ STRUCK
outubro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Application Software Components in AUTOSAR

Master in Electrical and Computer Engineering

Thor Axel Achim Heinrich José Struck

Guidance Teacher:
Cecília Maria Reis

Ano Letivo: 2019-2020

Instituto Superior de Engenharia do Porto
Departamento de Engenharia Eletrotécnica
Rua Dr. António Bernardino de Almeida, 431, 4200-072 Porto

Abstract

This dissertation presents a set of concepts of AUTomotive Open System ARchitecture, AUTOSAR, starting with how AUTOSAR is structured in differents layers, where each layer has a specific task. The Runtime Environment, RTE, is the most important layer when the topic is communication between AUTOSAR layers.

The Virtual Functional Bus, VFB, is key concepts to facilitate the designing an Automotive System, making it possible to relocate some Software Components that belongs to the AUTOSAR Application Layer.

This dissertation will also approach, in a development way, key concepts needed to create Software Components, such as, Data Types, the differents Software Components types, Interfaces Types used in the communication of the different Software Components and where the Software Component will call the implemented code, Runnables.

In this dissertation will be presented the key idea of reusability of the Software Components, and the strong architecture of AUTOSAR.

Keywords: AUTomotive Open System ARchitecture, Runtime Environment, Virtual Functional Bus, Software Components, AUTOSAR, RTE, VFB, SWC, Data Type, Interface, Runnable

Contents

List of Figures

List of Tables

Glossary

Abbreviations	Description
AP	*Adaptive Platform*
API	*Application Programming Interface*
Application	*Application Programming Interface*
ARXML	*AUTOSAR XML*
ASPICE	*Automotive SPICE*
AUTOSAR	*AUTomotive Open System ARchitecture*
BSW	*Basic Software*
BSWM	*Basic Software Modules*
CDD	*Complex Device Driver*
ComSpec	*Communication Specification*
CP	*Classic Platform*
CS	*Client-Server Interface*
DTC	*Data Trouble Code*
E/E	*End-to-End*
ECU	*Electronic Control Unit*
ECUAL	*ECU Abstraction Layer*
EEPROM	*Electrically Erasable Programmable Read-Only Memory*
FIFO	*First In First Out*
FO	*Foundation*
HW	*Hardware*
MCAL	*Microcontroller Abstraction Layer*
MDS	*Mode Switch Interface*
NVM	*Non-Volatile Memory*
OEM	*Original Equipment Manufacturer*
PPort	*Provide Port*
RPort	*Require Port*
RTE	*Runtime Environment*
Runnable	*RunnableEntity*
SR	*Sender-Receiver Interface*
SW	*Software*

Abbreviations	Description
SWC	*Software Component*
VFB	*Virtual Functional Bus*

Chapter 1

Introduction

The software development process has a great impact on the continuous advances in various industries, such as on mobile industries giving the possibility to have more optimized apps, or on the automotive industry to have more reliable software.

The automotive industry is always in constant evolution since it is a very competitive industry. For this reason, there are always changing requirements that, on most of the cases, are implemented.

The changing of requirements can be seen also as new features that need to be created in a way to keep up with the flow of the innovations of the competition.

1.1 Context

Since all companies are being more and more competitive with each other to try to be the leaders in their respective fields and only the most innovative companies can be successful, bringing the most innovative functionalities to the end product.

By trying to be innovative, there could be some requirements for development that are contradictory, for example, "supporting driver assistance systems in critical driving manoeuvers while also improving fuel economy and conforming to environmental standards" [11]. The contradiction of this type of requirements can happen if there is an importation process from previouses generation of requirements without validation of their integrity.

To be able to confirm with the old requirements and the newer ones there is a need to have a new technological approach, it could be a new Software

Architecture for ECU, Electronic Control Unit.

So in 2003, OEM and suppliers joined together in order to create a standard to solve this issue, with that they created AUTOSAR.

Under the slogan "Cooperate on Standards, Compete on Implementation" [12] all participant, that was involved to create AUTOSAR, created the necessaries standards for the basic functionalities leaving room for innovation and competition.

1.2 Objectives

In this section will be present the main objectives that will be discussed in the paper. The main objectives are:

- Referencing some of the main points of the Automotive Industry;
- Study a tool commonly used in this industry by:
  - Exploring how it was created;
  - How it is structured;
  - Key concepts.

1.3 Structure of Essay

This essay will be structured in 4 chapters.

Chapter 1 introduces the dissertation, giving the reader a global picture. It presents the industry that was studied, the main and secondary objectives that this essay will try to deliver.

Chapter 2 gives a better contextualization to the industry, introduces to AUTOSAR explaining how it was founded and will also differentiate both types of AUTOSAR, Classic and Adaptive.

After this, Chapter 3 will address core aspects of classic AUTOSAR like:

- The layered structure;
- How each layer interact with each other.

The following chapter, Chapter 4 will address technical aspects of Classic AUTOSAR such as:

- Interfaces and Data Types;

- Runnables and Functions called via RTE Tasks.

After the presenting the core and techincal topics from Classic AUTOSAR, Chapter 5 will summarize in a structured way creating a possible guide for the SW Development. In this chapter will also present some Use Cases.

Last but not least, Chapter 6 concludes with a summary of all the key topics aborded on the dissertation with also the Pros and Cons about AUTOSAR.

Chapter 2

Automotive

This chapter will be presented in the first stage the history of this industry, from the first steps to the present. The next stage will approach a more technical level where will be explained some key concepts of this industry.

2.1 History

The first thing that comes in mind by referring Automotive Industry is certainly cars, right? So the history of this industry comes since ancient times when was invented the wheel since it is one of the key components to the car.

The wheel is supposed to be invented around 3600-3300 B.C. but there is only record for this invention in ancient Greece Era, between 600-400 B.C [13]. At that time, it was already used as a way to simplify the workload for transporting materials and at that time this item was highly-priced but the buyer would receive their investment back within a few days since the workload was greatly reduced.

In other records, there is evidence that the wheel existed 300 years before but instead of being used as a way to simplify the work for transportation was commonly used for pottery [13].

Years passed by since there wasn't any new invention related to the usage of the wheel until the 15th century where Leonardo da Vinci made a draw of some theoretical plans for a motor vehicle.

Some years have passed and in 1698, an English engineer named Thomas Savery patented the first crude steam engine [14]. The steam engine was used mainly on trains and boats.

However, in 1769, engineer Nicolas Joseph Cugnot invented the first military tractor [1]. Cugnot used for this vehicle a steam engine. This was already a big step to create cars but, there was a shortcoming this vehicle had an autonomy of 10-15 min after that it needs to build again steam power. In the following year, he built a steam-powered tricycle that could carrier 4 passengers, as shown in Figure 2.1.

Figure 2.1: First automobile [1].

Cugnot, in 1771, was the first person to get involved in an automobile accident.

The automobile was perfected in Germany and France in the late 1800s, however, the first country that was the leader in this field was America, in the early 20th century. The first gasoline vehicle was invented in America by two bicycle mechanics, J. Frank and Charles Duryea, in 1893 [15].

Like history as always shown America would produce cars in larger numbers ate lower prices, compared to European countries. In 1908 was introduced the Model T from Henry Ford [15]. The first Model T was sold for $825. Four years had passed and Ford was able to reduce the price of the Model T to $575 and in 1927 optimized the process to sell this same model at almost half the price. In this time was published the Ford Model T slogan "You can have any color as long as it's black." [16]. This slogan happens to have logic since back at that time they had not to have the technology to accelerate the drying process. The black color absorbs faster the heat making it faster to dry.

While other American automobile manufacturers used Ford's mass production techniques European companies only adopted it around the 1930s [15]. The numbers of automakers dropped from 253, in 1908, to 44, in 1929. Around this

time was already possible in America to get a Ford Model T at a reduced price of $290 [15].

The automotive industry had a key time when there was a big development process. This happened during war times, mostly on World War II. During this war, American automobile manufacturers made 75 essential military items, being most of them related to motorized vehicles In total the material provided by this industry was one-fifth of the American nation's war production cost [15].

After World War II have passed other countries and nations joined also into this industry, such as Japan and Europe.

It became questionable the aesthetics nonfunctional styling at the cost of economy and safety issues. Around 1965 the American cars were being delivered with 24 defects per unit, sometimes those defects were safety-related, things that would be unthinkable at today's date. After this being detected it was needed to have some way to guaranty the minimum safety standards. So it was started to have some impositions of federal standards.

In 1966 was implemented standards for automotive safety.

In 1965 and 1970 was implemented standards regarding emission pollutants.

In 1975 was implemented standards for energy consumption.

With all these standards Japan was one of the first that could satisfy these 3 impositions, creating fuel-efficient, functionally designed, well-built small cars [15].

After the American-made car peaked a record of 12.87 million sold cars, they fell to almost half, as the imports increased from 17.7 to 27.9 percent. In the meantime, in 1980, Japan became the world's leading auto producer and still holds this title.

In modern days, every country had already contributed to this industry area, since financial support or for the development process.

In Europe this industry is one key industry, economically speaking, since it links other important industries to the European Union, like for example chemicals, steel, information technology and others [2]. This industry provides jobs for around 12 million people, where 3 million are involved in the manufacturing process, 4.8 million for transport and 4.3 million for sales and maintenance.

The following Figure 2.2 is possible to summarize the outcome of this big industry.

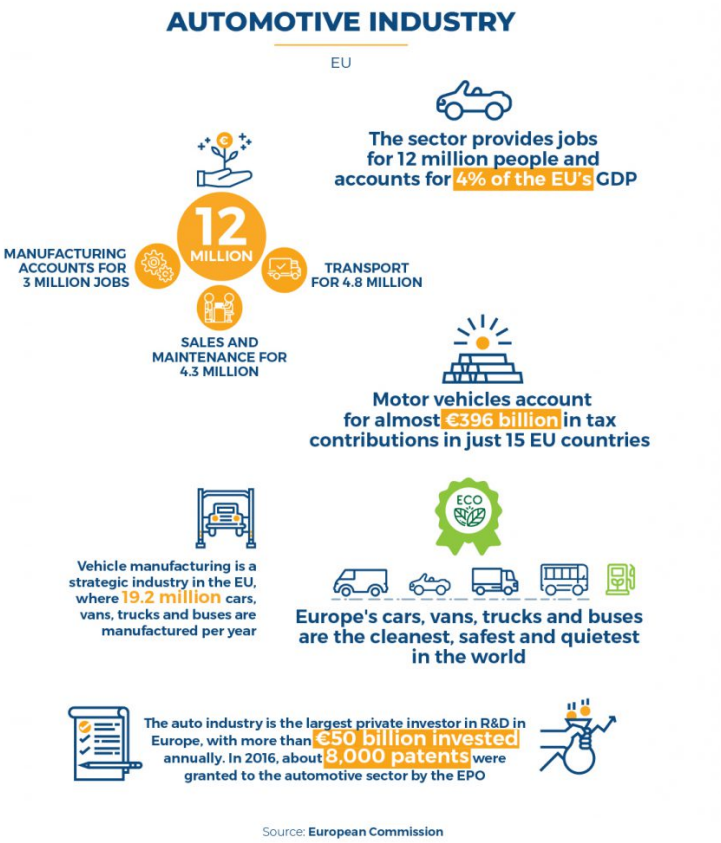


Figure 2.2: Automotive Industry contribution to Europe [2].

2.2 AUTOSAR

As the evolution of technology is going always in direction of Internet of Things and having always more software, SW, inside each new product there surges a need to standardize the concepts. This standard will help to guide and assure the developer of a product that will have quality and consistency. For the company, this will benefit the economy, safety and improvement of the quality, since it will have more and more people joining them and correct the flaws that are discovered.

With that as objective organizations are created, such as AUTOSAR that try to create some standards to be used by most of the companies in the Automotive industry.

2.2.1 Establishment of AUTOSAR

AUTomotive **O**pen **S**ystem **AR**chitecture, or AUTOSAR to abreviate, was founded in Autumn of 2003, but kicked-off in August 2002 with BMW, Bosch, Continental, DaimlerChrysler and Volkswagen [17]. The main objective is to provide a reference Architecture for ECU Software Development in a way to overcome the growing complexity of modern days software necessities since the trend of innovation in the automotive is focusing on software.

The following Figure 2.3 presents how the AUTOSAR Organization is composed.

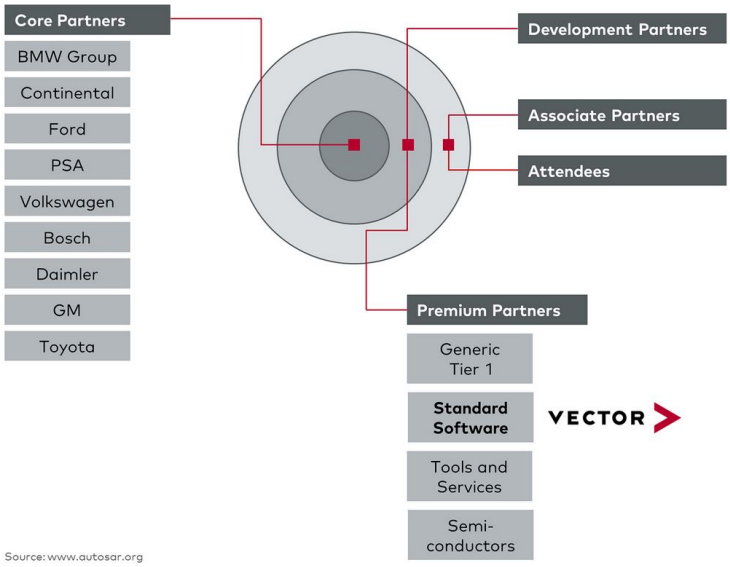


Figure 2.3: AUTOSAR Partnership graphic [3].

This software is implemented for ECU, Electronic Control Unit, and refers to any embedded system in automotive electronics.

In the early stages there was only needed one ECU to control the automotive component, for example when it was presented the electronic fuel injection [4]. This feature is something that needs to be controlled by a component with some logic connected behind it.

When a feature that could improve either the performance of the car or the quality of driving is created, other OEM sees it like an idea to future implement it. Without any notice these features start to be seen as a "must-have" on the car and turns indispensable [4]. The following image, Figure 2.4, presents the evolution of some key features that were developed and turned in most of the

cases indispensable to not being implemented, for legal reasons or marketing purposes.

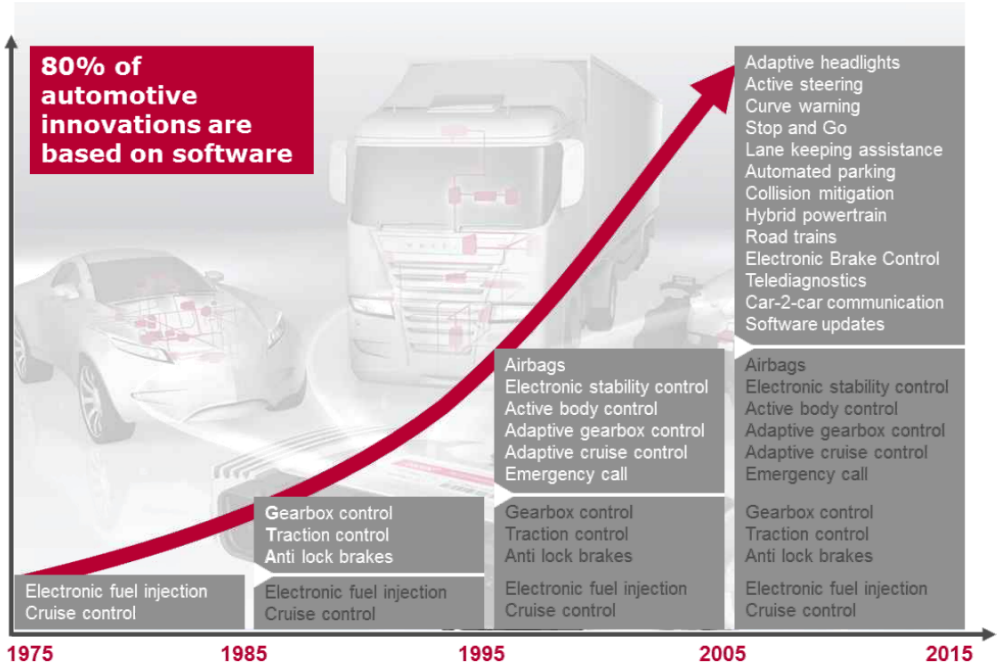


Figure 2.4: Car features progression [4].

With the increase of mandatory features and newer features to catch the eye of the consumer the number of ECU is always increasing [4]. The main reason for this increase is simply because having a lot of ECU dispersed in the vehicle is better than having a very powerful. With this approach is ECU can focus only on one task, receiving the values from all the sensors connected to it and provided that information to the other ECU.

The following figure, Figure 2.5 represents how the car systems have been evolved and the perspective that the AUTOSAR Organization has in view.

The current status is that we have complex electronics systems within the vehicle and each vehicle as at least 100 ECU. With this high number of control units in a product it creates some big challenges like [4]:

- Electrical/Electronic **complexity is growing fast**;
- Quantity of Software is **exploding**;
- Hardware platforms are **not unique**;
- Development **process** and **data formats** are not standardized.

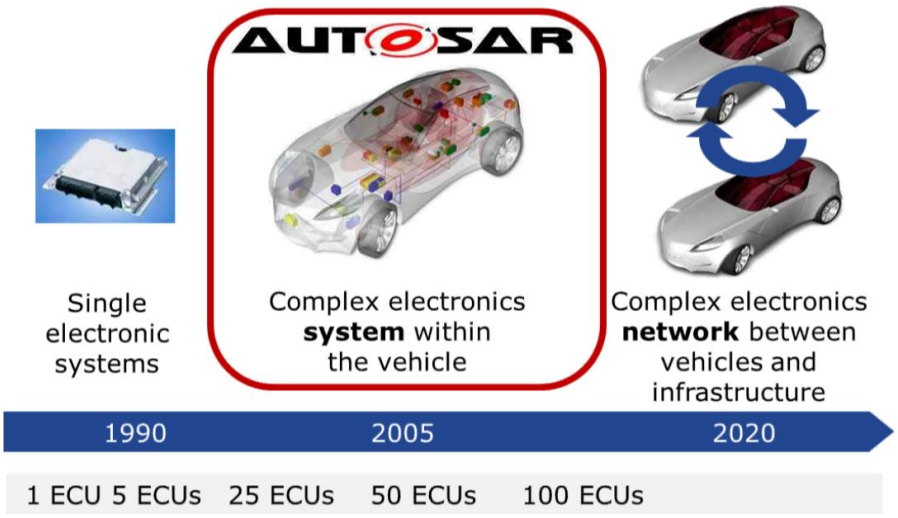


Figure 2.5: Car features progression [4].

The first two points are in sync simply because the number of ECU is always growing and with it, each ECU needs his own SW. For every new product the hardware, HW, is changed normally to improve the performance or to maintain it at a lower price. With this, every single new car line will create an update in SW to bring new features and new HW to provide the required technology needed to fulfill it.

To try to solve these issues AUTOSAR was created and it main objective is "Improve software quality and reduce cost by re-use" [4]. This means that the main focus would be creating software that can be re-used:

- Functions across various car manufacturers;
- Development methods and tools;
- Basic Software.

Herewith the main benefit that the car manufacturers hope to get are [12]:

- Optimize the ECU network by flexible integration, relocation and exchange functions;
- Master over the increasing product and process complexity (Increasing the product value and simplify the complex development process);
- Maintainability over entire product life cycles.

2.2.2 AUTOSAR Standards

The AUTOSAR Organization as any other organization or company delivers a product to the market. In this case, the product is the specification of a concept and methodologies that are being always being worked on. These specifications are delivered to the market as standards.

AUTOSAR tries to address a wide range of use cases in automotive software development with its product, creating the following standards [6]:

- Adaptive Platform, AP;
- Classic Platform, CP;
- Foundation, FO.

Each of the standards has its uses and normally is for a specific type of system. Adaptive Platform is a solution for high-performance computing ECUs and is normally used in autonomous driving [5]. Classic Platform is the solution where hard real-time and safety constraint, for example, Instrument Clusters [6] In meanwhile the Foundation is responsible to enforce the interoperability between both platforms, AP and CP [5][6].

An AUTOSAR standard is a consistent set of AUTOSAR deliverables. These deliverables can be of taking form in several different formats like [5]:

- Explanation in textual format;
- Specification for textual or test;
- Source code and/or Schemas.

At the time of release, all the dependencies, in the standards, are fulfilled. The Foundation is the standard that needs to be able to support all the dependency and inter-communication between the AP and the CP. The following figure is an example of how the dependencies of AUTOSAR Standards, Figure 2.6

In AUTOSAR Standards there are two different types of release numbering:

- Internal Release Numbering;
- Platform Release Numbering.

The easy numbering system that AUTOSAR uses, Platform Release Numbering, is where the release is concrete in time, for example in Adaptive Platform the numbering is like R-<*YY*>.<*MM*>, were the *YY* [5] corresponds to the year

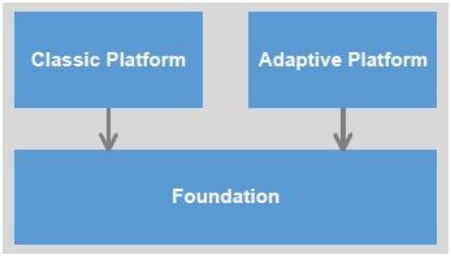


Figure 2.6: Dependencies of AUTOSAR Standards [5].

and the *MM* correspond to the month. This release numbering system leads to the discontinuation of the Internal Release Numbering.

Internal Release Numbering is still maintained for different purposes and it uses a three digit numbering scheme, R<*Major*>.<*Minor*>.<*Revision*>, to identify the releases [6].

A *Major* release is a valid specification and may contain specifications that need to be updated due to some incompatible fixes.

A *Minor* release is a valid specification that only may be changed for backward compatibility. Some draft parts still may be backward incompatible.

A *Revision* will only contain backward compatible bug fixes [6].

The following image, Figure 2.7, will illustrate how is the life cycle of an AUTOSAR Standard release.

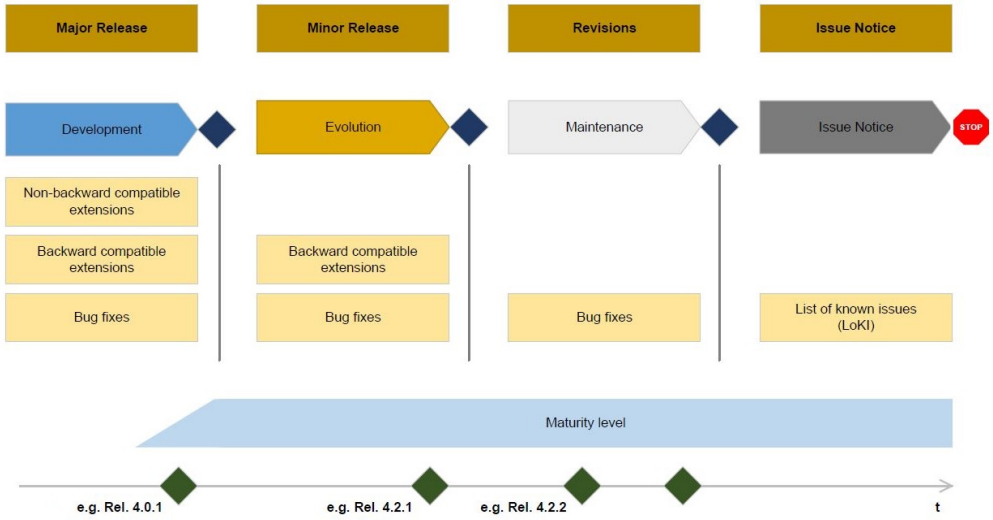


Figure 2.7: Life cycle model of a major release [6].

As represented in Figure 2.7 not all types of adaptation or documentation improvement can be done on its Release step.

On a *Major* release there can be added Non-backward compatible extension, meanwhile this type of improvement cannot be added on a *Minor* or *Revision* release. This goes also for the other types of a fix as shown in Figure 2.7.

Chapter 3

AUTOSAR Concept

In this chapter will be presented some concepts of AUTOSAR more concretely Classic AUTOSAR.

On the first point, will be illustrated how Classic AUTOSAR is structured, after what each part of the structure represents and provide to the system.

Secondly, and to give continuity to the previous chapter, it will be presented a key functionality that AUTOSAR could and has provided to a certain extent. This functionality is to give an easy way to design a vehicle System.

3.1 Structure of Classic AUTOSAR

The main point in AUTOSAR is to be able to create Software that can be reused without any issue, meaning that it is not dependent on the HW. For this to be archive it was decided by AUTOSAR Organization an approach of Layers.

In Classic AUTOSAR, the one that will be a boarded on this essay has 3 main layers [10]like how described on the Figure 3.1:

- Application Layer;
- Runtime Environment;
- Basic Software Layer.

Each layer has his responsibilities and dependencies in the overall software and is described as a top-down approach [7], where the Application Layer is the focal point in this essay.

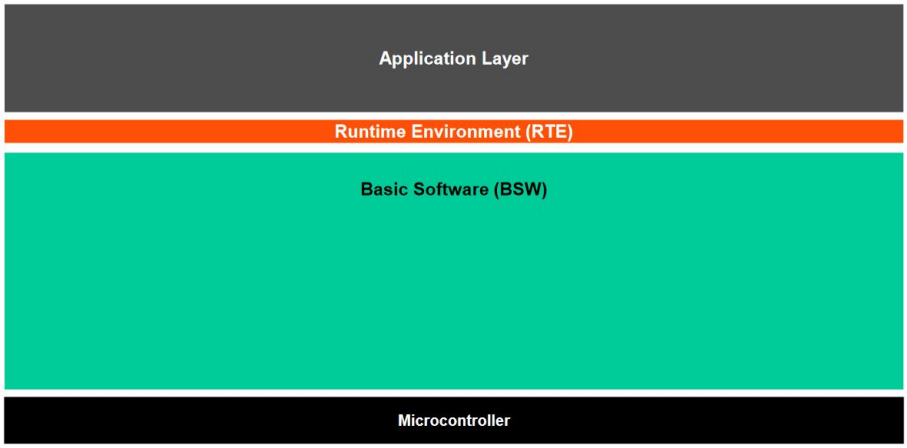


Figure 3.1: The 3 Main Layers [7].

The Application Layer is the part where the project-specific features are implemented by Software Components, SWC [10]. On this layer, all SWC communicate with each other and with the Basic Software using interfaces described on the SWC or using some Standard interfaces, Section 4.3, via the Runtime Environment, RTE, where this connection is independent of the hardware [10].

The Runtime Environment, or RTE, is the only connection that all the Application Software Components should have with the Basic Software, BSW, and others Application Software. This will bring the idealogy of Software Components to be ECU independent into reality. The RTE is always generated with an RTE generator from a certified company that ensures that all the standards are being fulfilled. Each ECU has its own customized RTE implementation which is generated during the ECU Configuration process [8].

During the code generation there will be also generated code to execute and call each function inside the SWC. The function in each SWC are known as `RunnableEntity`, topic that will be explained in Section 4.4. In order to call these `RunnableEntity`, there are so call RTE Tasks that are generated during the code generation where each event that will trigger a Runnable is linked to. The event in order to run an extract of code that is inside an SWC is known as RTE Events, and it there are a diverse type of possible events. The most common used RTE Events are:

- Timing Event;
- Data Receive Event;
- Operation Invoke Event.

The last AUTOSAR specific layer is the Basic Software. This layer is responsible to provide all the connection from the microcontroller to the rest of the ECU and needs also to provide some services, making the need to sub-divided this layer in [7]:

- Services Layer;
- ECU Abstraction Layer;
- Microcontroller Abstraction Layer
- Complex Device Driver.

The following image, Figure 3.2, will represent AUTOSAR Classic Standard Architecture with the Basic Software sub-divided into functional layers.

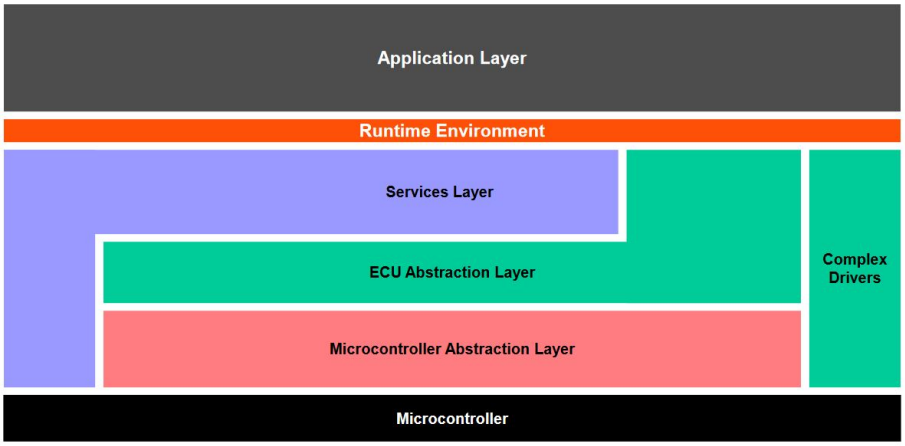


Figure 3.2: High Level Architecture [7].

As presented on Figure 3.2 and mentioned before the BSW is sub-divided into different functional groups.

The Microcontroller Abstraction Layer is where the code is implemented according to the microcontroller that is used for the ECU. This layer contains all the internal drivers with direct access to the microcontroller and its internal peripherals [7].

This layer is very dependent on the HW used for the ECU and its main objective is to make for the higher layers to be independent from the HW [7], making it in a way that the other components do not need to access directly the microcontroller.

The ECU Abstraction Layer is an interface for the higher layered components, such as Software Components of the Application Layer, to the drivers

from Microcontroller Abstraction Layer [7]. Here resides all application programming interfaces, API, to enable the interaction between the SWC or BSWM to peripherals and devices. This applies to any component inside the ECU.

The main responsibility of this layer is to make the layers that resides above it even more independent of the HW.

The Service Layer is the highest layer in the Basic Software which also applies that this is the layer that interacts mostly with the Application Components [7]. While on the ECU Abstraction Layer covers the interaction with the HW the Service Layer needs to provide those services to the Application. This includes:

- Operating System;
- ECU States such as:
  - Communication States;
  - ECU State Management;
  - Mode Management;
- Communication, vehicle network systems and internal ECU Communcation;
- Off-Board Communication;
- Diagnostic Services;
- Memory Services;
- Watchdogs Managers;
- Criptography.

With this it is possible to sub-divide further the Basic Software into their functional groups. The following figure, Figure 3.3, shows how it is possible to sub-divide further the Basic Software.

At last for the Basic Software there is the Complex Drivers. This a type of component that is used for Non-AUTOSAR compatible Applications, such as applications that needs higher timing constrains. This type of component are abboarded also in Section 4.2.1.1.

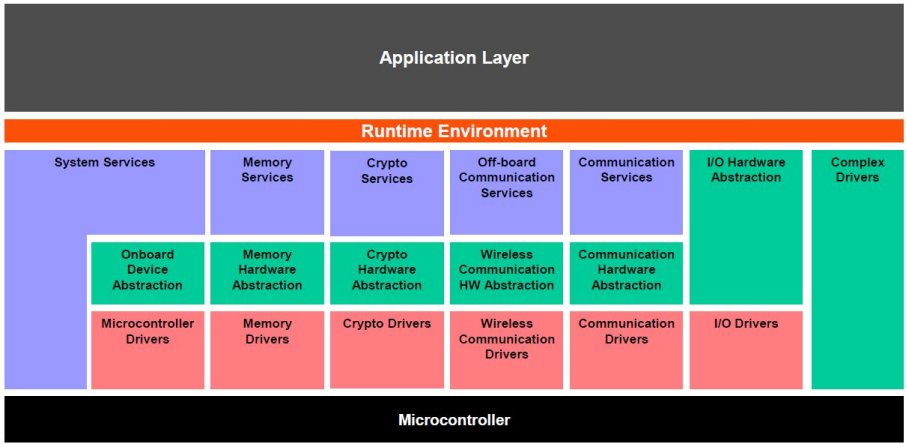


Figure 3.3: Basic Software divided into functional groups [7].

3.2 Conceptualization of Application Features

When planning some features for a vehicle and with the complexity already seen on Section 2.2.1, where every new vehicle has an enormous number of ECUs, making it have also an enormous quantity of SW in it.

For these positions having a way to model the feature without expecting where every SWC will be located will make this task easier, since the SWC needs to be possible to import into newer or other Electronic Control Unit.

Too archive this accomplishment AUTOSAR Standard introduces architectural concepts that facilitate infrastructure independence. Specifically this concepts are *Virtual Functional Bus*, VFB, and *Runtime Environment*, RTE. These two notions are closely related to each other.

From a general perspective the *Virtual Functional Bus*, VFB, can be described as a system modeling and communication concept [8] making a methodology that allows for a strict separation between the application and infrastructure [9]. This is also a feature that AUTOSAR brings since all the SWC needs to be independent of the communication mechanisms.

The VFB can, in a second point, be used for plausible checks in the communication of software components [9]. In the specification of Virtual Functional Bus, it needs to provide concepts for all infrastructure services that need to be created for an automotive application, like for example:

- Communication between SWC, actuators and sensors;
- Accessibility with the Standardized Services;

- Behaviors for different power states

In Figure 3.4, it is illustrated a possible way of how to represent a Virtual Functional Bus. All components are connected through the VFB which enables the software components to be developed and maintained independently of the ECU specifics and details [10]. In other words, all software components, complex device drivers, services and ECU abstraction uses a type of interface to communicate with the VFB.

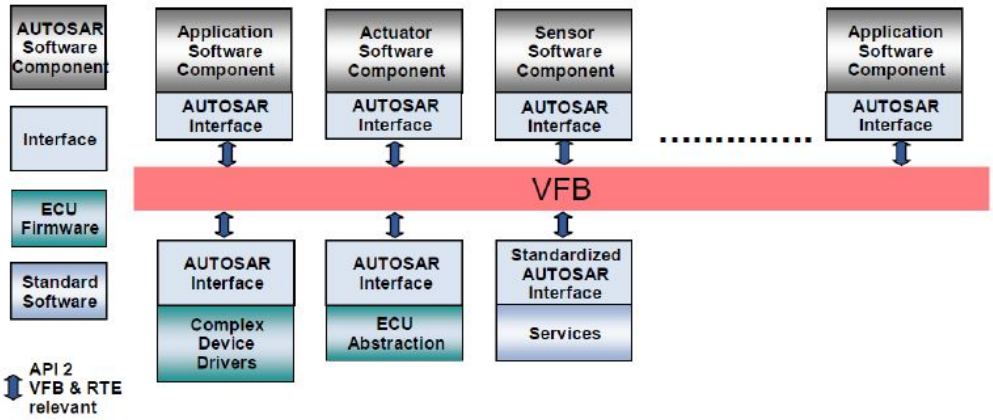


Figure 3.4: Overview of virtual interaction using Virtual Functional Bus [8].

In contrast to the VFB, where all the connections between the SWC are an only concept the Runtime Environment, brings it to realization [8][10].

Software Components that are mapped into the same ECU will communicate with Intra-ECU communication such as function calls. In the case that SWC is mapped into different an ECU, it is needed to use Inter-ECU communication such as communication bus infrastructure, for example, *CAN* or *FlexRay*. The RTE is responsible to create this path of communication for the Intra-ECU and Inter-ECU since the RTE can be seen as a static implementation of specialized communication topology [8].

In Figure 3.5, it is illustrated as an example of how a VFB concept could be translated to the RTE and the allocation of SWC into differents ECUs.

3.3 Interfaces Types of Basic Software

In order to transfer from a VFB to a possible RTE solution, there is needed to define which kind of interface category is necessary to implement an application.

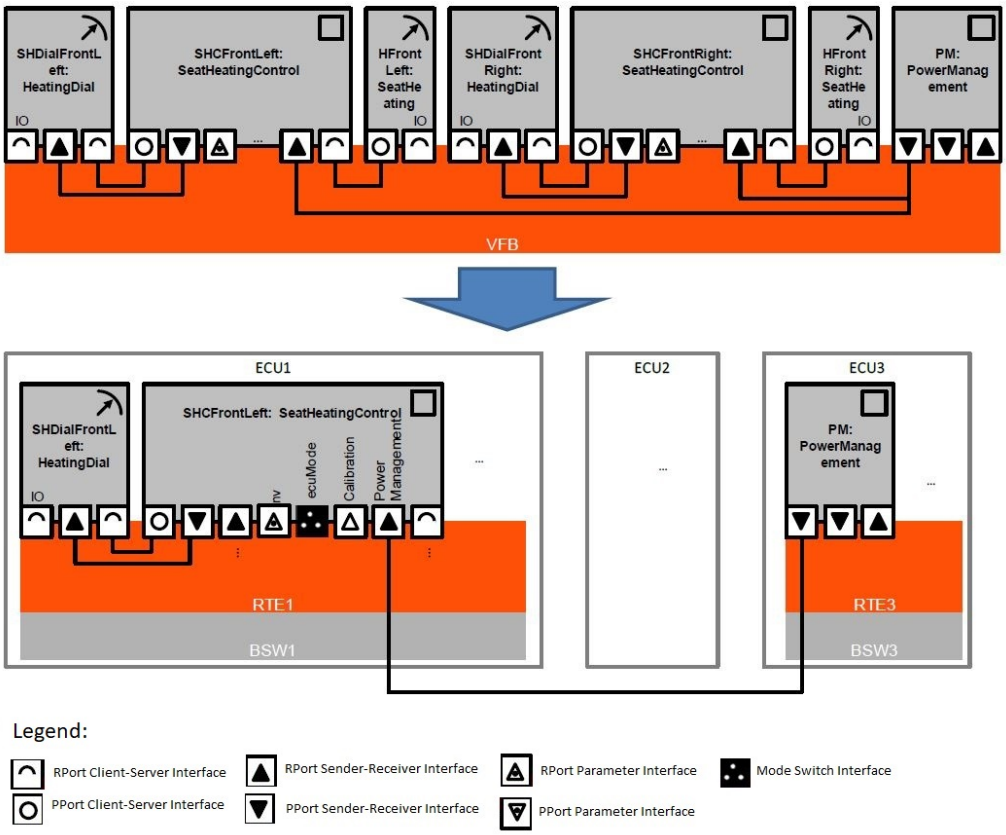


Figure 3.5: Example of VFB to RTE mapping of SWC [9].

The most common category of interface used in the Application Layer is the **AUTOSAR Interface**. In the **AUTOSAR Interface** there are a couple of types of interfaces, known as *PortInterfaces*4.3, that the developer can choose from.

In AUTOSAR there are other categories of interfaces making it in total three, that are:

- **AUTOSAR Interface**;
- **Standardized AUTOSAR Interface**;
- **Standardized Interface**.

An AUTOSAR Interface defines the information exchanged between Software Components and/or BSW Modules [10]. From the BSW Modules the only components that might have, they are not obligatory to have, are ECU Abstraction components and Complex Device Drivers [8] [10].

All the components that are connected to the RTE using an AUTOSAR Interface can connect to ports and in that way, interact with other components [8].

Standardized AUTOSAR Interfaces are like AUTOSAR Interfaces whose syntax and semantics are standardized in AUTOSAR [10]. Normally they are defined injunction with the AUTOSAR Services provided by the BSW.

A software interface is referred to as a Standardized Interface if the Application Programming Interface, API, is defined by the AUTOSAR specification [10]. This type of interface is used in the Basic Software Modules, BSWM.

A Standardized Interface does not use the same approach as the AUTOSAR Interfaces, this means they do not use the RTE like AUTOSAR Interfaces do [10]. These interfaces are typically defined for a specific programming language.

On a VFB model, the AUTOSAR Interfaces can be defined, for Standardized AUTOSAR Interfaces, the definition can be derived as the AUTOSAR Interface. However, in the case of the Standardized Interfaces, they can be not defined on the VFB model.

The following figure, Figure 3.6, exemplifies where each category of interfaces locates.

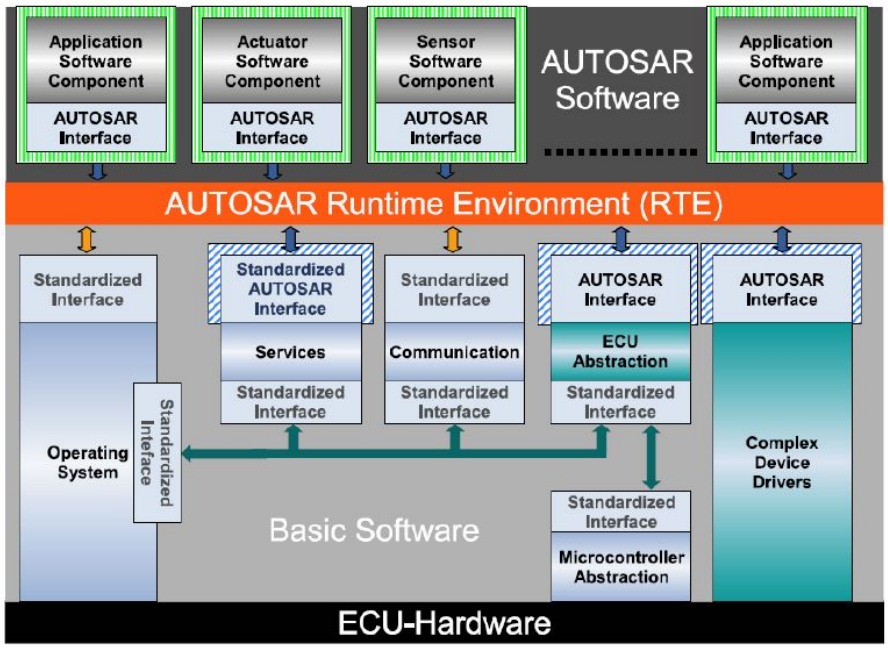


Figure 3.6: Location of each Interface category in the architecture [8].

Chapter 4

AUTOSAR Implementation Concepts

In this chapter will be presented some implementation concepts of AUTOSAR.

First will be explained about the Data Types from AUTOSAR, this are in other words variables used in the communication between Software Components. After it will be showed differents Software Components, how they communicate which each other and how the Code implementation is called in a System that uses AUTOSAR.

4.1 Data Types used in Communication of Software Components

From the concept of how the system should work, as described on Chapter 3.2, and with the implementation of Standardized Interfaces defined by AUTOSAR, Chapter 3.3, there is a need to define how the flow of data is handled. So using C Programming Language as an example, the normal manner to handle the information would be by using a variable, and by creating some variables, System Engineer will have to define:

- Space needed per variable, on RAM and EEPROM;
- Where they will be used, like on function, SWC or even on RTE.

In AUTOSAR they are known as Data Type. However, in AUTOSAR, the data types are a bit more complex since the Application layer needs to be handle

independently from the others. To be able to create another abstraction layer between the Application and the Implementation layer, in AUTOSAR 4, was created the concept of `ApplicationDataType`, making it the third abstraction level for Data Type [10].

There are 3 different abstraction levels for data types:

- Application Data Level;
- Implementation Data Level;
- Base Data Level;

The properties of an `AutosarDataType`, either `ApplicationDataType` or `ImplementationDataType`, can be defined as `SwDataDefProps`. Some of the properties are only applicable for specific data types, being unlocked with certain categories [10].

The `SwDataDefProps` can be defined by several AUTOSAR elements. In case of differents AUTOSAR elements define the same properties, there is a need to prioritize which has a higher criticality in the system, making in a way that there will be cases where the properties will be overwritten by the one with higher priority. The following enumeration is the priority list, which the lower number means it has the higher priority:

1. `McDataInstance`;
2. `FlatInstanceDescriptor`;
3. `ParameterAccess`;
4. `InstantiationDataDefProps`;
5. `AutosarDataType`, Section 4.1;
6. `ImplementationDataType`, Section 4.1.2;
7. `ApplicationDataType`, Section 4.1.1.

4.1.1 Application Data Level

This is the Data Level with the highest abstraction on the System. It is on this level where where all the `ApplicationDataType` and `AutosarDataType` needed for the Application layer to reside. On this level there is no need to have concrete detail of the implementation, like bit-size of data or data struct. With no concrete definition of how it will be implemented, this type of data is commonly during

the system design, allowing, in a way, for the System Specifiers the liberty of thinking what kind of data flow is done [10]. During the creation of the flow on a higher level, like on VFB, the `ApplicationDataType` is used to define the type of data that is transferred by the SWC.

During the integration process of the SWC or the updates of the same, it is done also the mapping of the `ApplicationDataType` to their correspondent `ImplementationDataType` [10].

The following figure, Figure 4.1, shows how are the relations between the differents `ApplicationDataTypes`, showing the division of it into 2 main groups where `ApplicationCompositeDataType` is divided further into 2 groups:

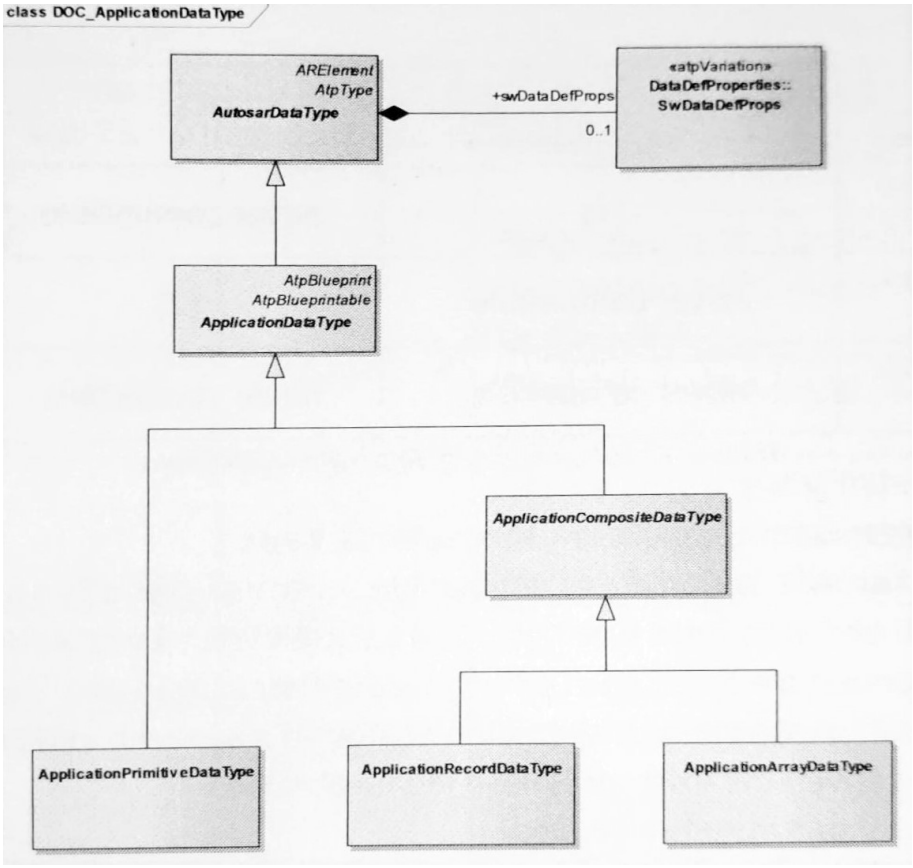


Figure 4.1: Application data types and their relations[10].

- `ApplicationPrimitiveDataType`
- `ApplicationCompositeDataType`
  - `ApplicationRecordDataType`

- ApplicationArrayDataType

The ApplicationCompositeDataType, as the name suggest, it is Data Type that is compose by differents ApplicationDataType.

Like all the learning curves for a programming language it is started with simple variables. On Application Layer the simplest Data Type is the ApplicationPrimitiveDataType. As in any programming language there are some different types of variables to fulfill different needs. In AUTOSAR there is the same concept. The ApplicationPrimitiveDataType with around 10 different DataTypes for different uses.

The following table, Table 4.1, was created to examplefy all the possbile ApplicationPrimitiveDataType and their correspondent maping to ImplementationDataType.

Purpose	Category on ApplicationDataType	Mapped to ImplementationDataType
Calibration Parameters	ApplicationDataType	SwDataDefProps
Single Value	VALUE	VALUE
Enumeration	VALUE	VALUE
String	STRING	ARRAY/STRUCTURE
Value Blocks	VAL_BLK	ARRAY
Boolean	BOOLEAN	VALUE
Common Axis	COM_AXIS	ARRAY
Rescale Axis	RES_AXIS	ARRAY
Curve	CURVE	ARRAY
Map	MAP	ARRAY

Table 4.1: Possible ApplicationPrimitiveDataType.

As mentioned previously structures and data arrays can be defined on Application Layer via ApplicationCompositeDataType. The corresponding types are ApplicationRecordDataType and ApplicationArrayDataType.

The application structures,ApplicationRecordDataType, defintion can contain the any of the following elements:

- ApplicationPrimitiveDataType;
- ApplicationArrayDataType;
- or other ApplicationRecordDataType.

The category of a ApplicationRecordDataType is always STRUCTURE. An ApplicationRecordDataType is a compilation of varios elements, named ApplicationRecordElements, which themselves are also ApplicationDataType.

An `ApplicationArrayDataType` compilation of one or more elements, named `ApplicationArrayElement`. These element needs to be from the same `ApplicationDataType` [10].

It is possible to create multi-dimensional arrays. These type of arrays are created when one of the element references another `ApplicationArrayDataType`.

4.1.2 Implementation Data Level

Implementation Data Level, commonly known as `ImplementationDataType`, is another abstraction layer that will lead to the actual code implementation [10]. The representation of the `ImplementationDataType` is normally represented as **typedef** [10]. This Data Type is normally used for:

- interfaces and data within the BSW;
- interfaces of libraries which operate on a purely numerical level;
- interfaces between SWC and the BSW.

Compared to the `ApplicationDataType` this Data Type has less categories. The `ImplementationDataType` can have of the following kinds represented on table 4.2.

Implementation Data Type	category	Description
Primitive	`VALUE`	
Array	`ARRAY`	Contains ImplementationDataTypeElements for each dimension of the array
Struture/String	`STRUCTURE`	Contains ImplementationDataTypeElements
Union	`UNION`	Contains ImplementationDataTypeElements
Redefinition	`TYPE_REFERENCE`	Refers to another ImplementationDataType
Data Pointer	`DATA_REFERENCE`	The target is reference to a variable
Function Pointer	`FUNCTION_REFERENCE`	The target is referenced to a function

Table 4.2: Implementation Data Type kinds [10].

The following image, Figure 4.2, will represent how the interfaces of two SWC can be connected in case that they use different interfaces. For a connec-

tion of two SWC there is a need that the corresponding Data Type are compatible. This means only if the `ApplicationDataType` from both components are compatible their corresponding `ImplementationDataType` needs to be compatible to establish the connection of both SWC.

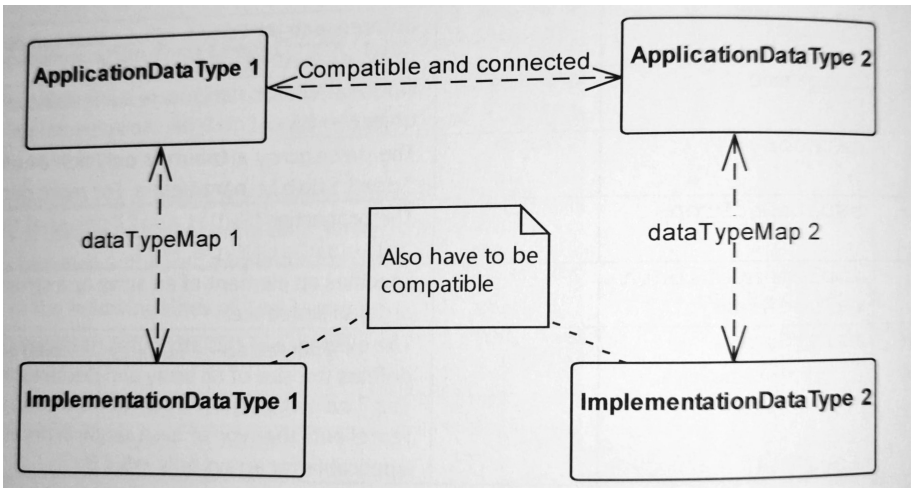


Figure 4.2: Dependency of Application and Implementation Data Type [10].

There 2 special types inserted on this level of abstraction. These are the Platform Types and the Standard Types.

The Platform Types are data types regarding some basic functionality. AUTOSAR as standalone implementation there will not be the basic variables that a programmer is used too, for example, there are not defined in AUTOSAR what an **uint8** is. In AUTOSAR this variable are defined as Platform Types. To define a Platform Type it needs to be implemented as a simple `ImplementationDataType`, with only its name and category set to `VALUE` [10].

The Standard Type are some standard definition normally used for version control of SWC or for Errors definition for the RTE Generated functions. All the generated RTE functions will return a `Std_ReturnType`, this return data type is defined as a **uint8**. This a Standard Type that will be used to return Error code since RTE Error until RTE infrastructure errors.

The following table, Table 4.3 will provide the layout for the differents `Std_ReturnType`. It is possible to observe that the 5 first bits are allocated for error codes that the System Engineer can define. For Application SWC these 5 bits could be used to trigger the error codes necessary for the receiving SWC knows that some error has occurred.

In AUTOSAR there are also some predefined Error Code. For that information there is the following table to summarize all the predefined error codes,

Bit	Value in Hex/Dec	Flag Description
7	0x80 128d	Immediate Infrastructure Error Flag.
6	0x40 64d	Overload Error Flag
5	0x20 32d	Available for error codes
4	0x10 16d	
3	0x08 8d	
2	0x04 4d	
1	0x02 2d	
0	0x01 1d	

Table 4.3: `Std_ReturnType` Layout [10].

Table A.1 and A.2. In these two tables it is possible to verify the AUTOSAR name for the error, the default value and a short description of the Error Code.

4.1.3 Base Data Level

On this level, the Data Types will describe the basic level of the `ImplementationDataTypes` in terms of bits and bytes. All `AutosarDataTypes` are finally defined by a `BasaType` which will be used on the RTE generator to create the corresponding C code implementation. For the `BaseType` there are only 2 categories:

- `FIXED_LENGTH`;
- `VARIABLE_LENGTH`.

It is on this level of detail where the common used variable types, such as, *uint16* are defined.

4.2 Implementation of AUTOSAR Software Components

Everything mentioned previously, except functionalities that are allocated to the BSWM, needs to be implemented Software Components, SWC. Thon file

type known as AUTOSAR XML files, ARXML. This is the filetype used in AUTOSAR and it is mainly used to deliver the configuration from OEM to the Tier 1 and other parties that are involved on the project. Never less is to say that all the files produced by the other parties will also be available to the OEM.

These files serves for several purposes:

- Configuration of the System;
- Implementation of Data Types, Section 4.1;
- Implementation of SWC, Section 4.2.1;
- Connection of the components;
- Implementation of Interfaces, Section 4.3.

On this section of the dissertation it will approach the topic of SWC.

4.2.1 Software Components

While the standard functionality are defined and implemented on the Basic Software Modules, all the non-standards are implemented by several Sofware Components [10]. These components can be of several types to fulfill differents use cases.

One concept that is very important and needs to take into account while creating an SWC is to define its scope. This means that on the system, the System Engineer, needs to have only a component to fulfill a task. For example, all the Application related SWC if they need to access some data from the NVM, should use only an SWC to fulfill this task. In AUTOSAR there is an SWC type that is specially for the NVM access, `NvBlockSwComponentType`.

The communication between SWC and SWC with BSWM is always explicit. For an SWC to communicate with other SWC it needs to have *Port* on its skeleton (topic will be approach on Section 4.3).

For the Software Components there are a total of 2 main groups:

- Atomic Sofware Component, Section 4.2.1.1;
- Parameter Software Component.

Additionally for those types there is one more that is the Composite Software Component. This SWC is compose by 2 or more SWC previously.

The following class diagram, Figure 4.3, will represent how the SWC are connected showing all the possibles SWC defined by AUTOSAR.

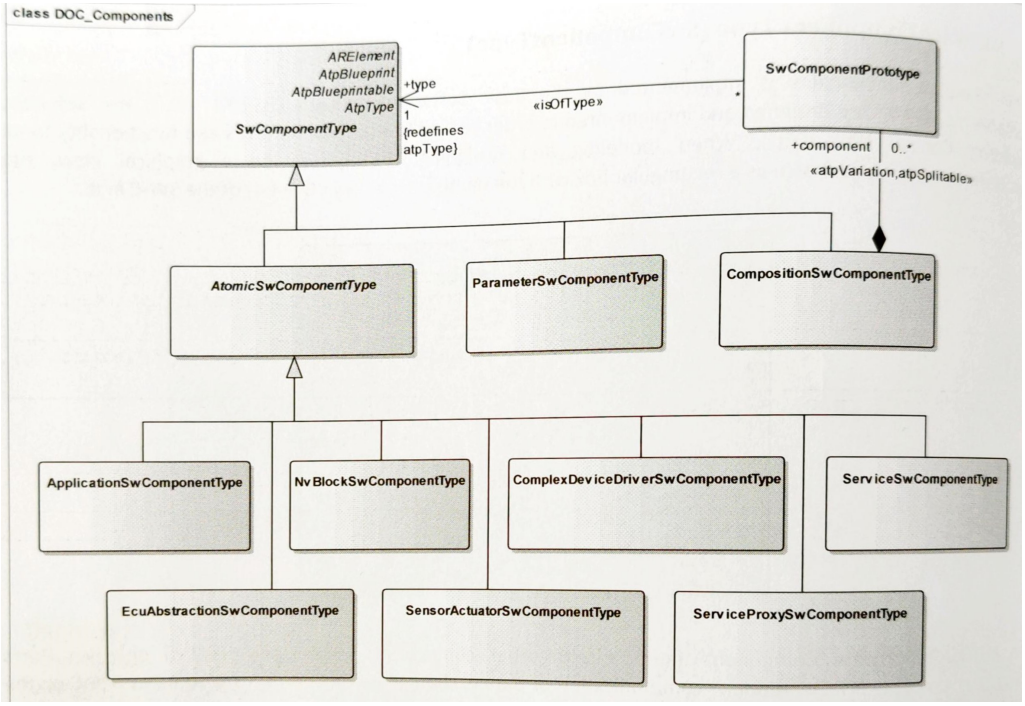


Figure 4.3: Software Component class diagram [10].

4.2.1.1 Atomic Software Component

An Atomic Software Component, `AtomicSwComponentType`, is a SWC that cannot be split into further SWC [10].

An `AtomicSwComponentType` is a specific form os `SwComponentType` consisting of the same attributes plus one optional for the `SwcInternalBehaviour` and some additional optional ones for Symbols Properties, `SymbolsProps` [10].

The `SymbolsProps` provides the ability to attach a symbolic name to `shortName` of a SWC [10]. This will avoid naming inconsistency during the RTE Generation. This property is specially useful if there are components from various suppliers that uses the same names by coincidence [10]. If that occurs the `shortName` of the SWC will be overwritten by the symbolic name. However this name will not be applied to the file names.

In terms `AtomicSwComponentType` there are some sub-categories. These subcategories are divided by their specialization such as:

- Application Software Component (`ApplicationSwComponentType`);
- Service Software Component (`ServicesSwComponentType`);

- Sensor-Actuator Software Component (`SensorActuatorSwComponentType`);
- ECU Abstraction Software Component (`EcuAbstractionSwComponentType`);
- CDD Software Component (`ComplexDeviceDriverSwComponentType`);
- Service Proxy Component(`ServiceProxyComponentType`);
- Non-Volatile Block Software Component (`NvBlockSwComponentType`).

An Application Software Component, `ApplicationSwComponentType`, is a SWC that is hardware independent.

It implements the software application, or only a part of it, and can use all AUTOSAR communication mechanisms and services [10]. In order for it to interact with sensors and actuators, it cannot do it directly [10]. This sort of interaction needs to be handled wit a Sensor-Actuator SWC. The communication between an Application SWC with the BSWM is done via an AUTOSAR Service interface [10].

While modeling and designing the system it is easier to distinguish all the Software Components. The following symbol, Figure 4.4, is an example of how an Application SWC and its *Port*s are represented.

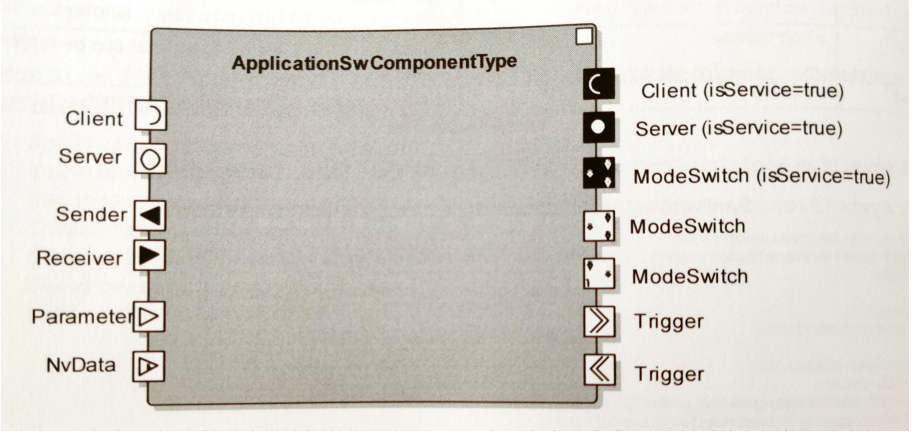


Figure 4.4: Symbol for Application Software Component [10].

The Service Software Component, `ServicesSwComponentType`, is a Software Component which will provide AUTOSAR Services to the Application. This component uses Standardized AUTOSAR Interfaces and it may interact directly with the BSWM [10].

The following symbol, Figure 4.5, is an example of how a Service SWC and its *Port*s are represented.

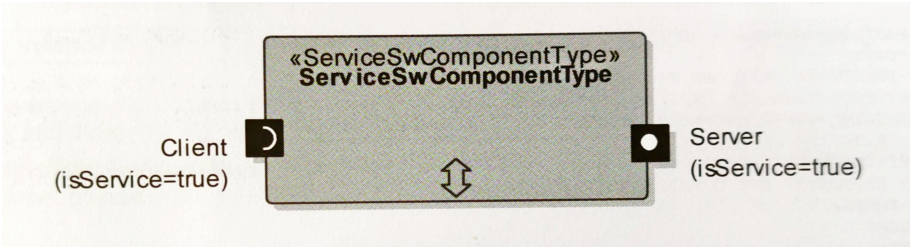


Figure 4.5: Symbol for Service Software Component [10].

The Sensor-Actuator SWC, `SensorActuatorSwComponentType`, is a special type of an `AtomicSwComponentType` as it depends on specific hardware making it hardware dependent and cannot be easily moved to another ECU [10]. The decision on where this type of SWC should be allocated, needs to be decided on an early stage of system designing, making it crucial to be defined on the VFB-level.

The main reason for this SWC to be HW dependent is because it need to be able to interact with a certain type of sensor or actuator. This makes that the implementation for this type of SWC are intended to:

- is written for a specific interaction with a specific sensor or actuator located on a certain ECU;
- read the value via an AUTOSAR Signal provided by the `EcuAbstractionSwComponentType` and transforms it to a representation of a physical value;
- provides the physical value to other `ApplicationSwComponentType`.

This Software Component will act as a link between other SWC and the `EcuAbstractionSwComponentType`.

In terms of implementation, this SWC normally comes with a *Port* that will use the Client-Server Interface, CS, to call an operation from the `EcuAbstractionSwComponentType` either to get the value from a sensor, **OP_GET**, or set an output to the actuator, **OP_SET**.

In order to illustrate the importance of this type of component, the following diagram, Figure 4.6, will show 3 different types of SWC interacting in order to receive the data from a sensor or providing value to an actuator. Is to note that the `SensorActuatorSwComponentType` can be connected to one or more `ApplicationSwComponentType`.

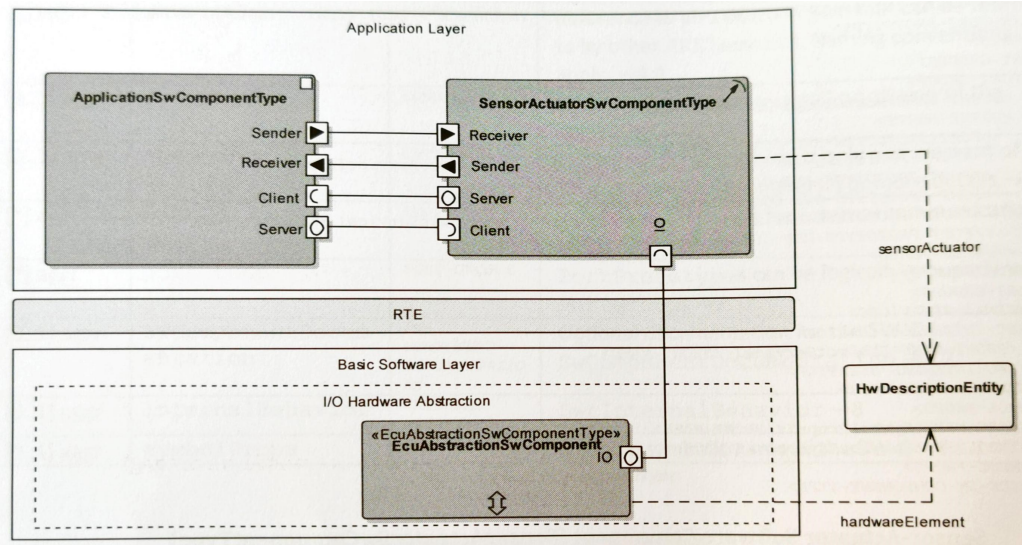


Figure 4.6: Example of utilization of a Sensor-Actuator Software Component [10].

The following symbol, Figure 4.7, is graphical representation of `SensorActuatorSwComponentType`. This type of symbol is normally used for designing the system in both VFB-level or Implementation-level.

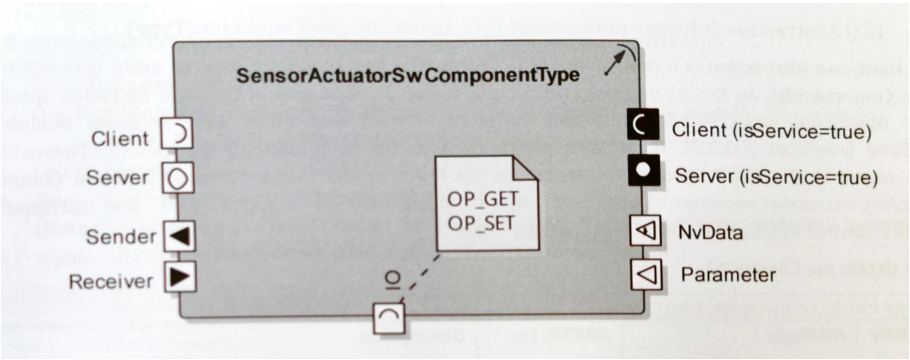


Figure 4.7: Symbol for Sensor-Actuator Software Component [10].

ECU Abstraction Software Component, `EcuAbstractionSwComponentType`, belongs to the ECU Abstraction Layer and it consist of one or more ECU Abstraction Components [10]. Since this component is part of the Basic Software Modules it can use the Standardized Interfaces in order to communicate with other BSWM.

This component type can interact directly with the IO from the microcon-

troller via the Sensor-Actuator Software Component [10].

The following symbol, Figure 4.8, is the graphical representation of `EcuAbstractionSwComponentType`.

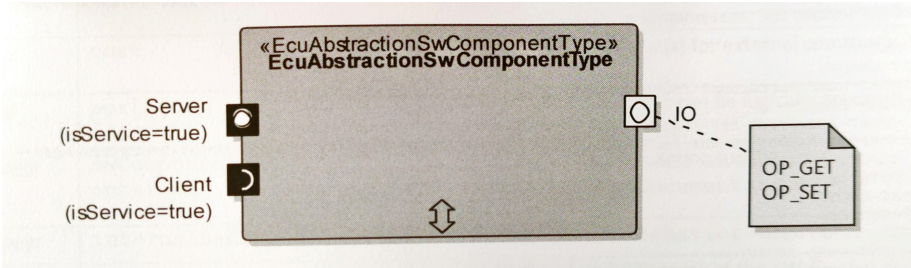


Figure 4.8: Symbol for ECU Abstraction Software Component [10].

Complex Device Driver (CDD) Software Component is used to model a function outside the standard AUTOSAR Basic Software stack for complex or resource critical sensor evaluation or actuator control [10]. This kind of component is used mostly when the HW component is not supported by the AUTOSAR Standard.

The CDD Component is a mix of ECU Abstraction SW Component and the Application SW Component, since it can interact directly with both of them via the Standardized Interface, for BSWM interaction, and AUTOSAR Interfaces for the SWC that belong on the Application Layer [10].

The following symbol, Figure 4.9, is the graphical representation of `ComplexDeviceDriverSwComponentType`.

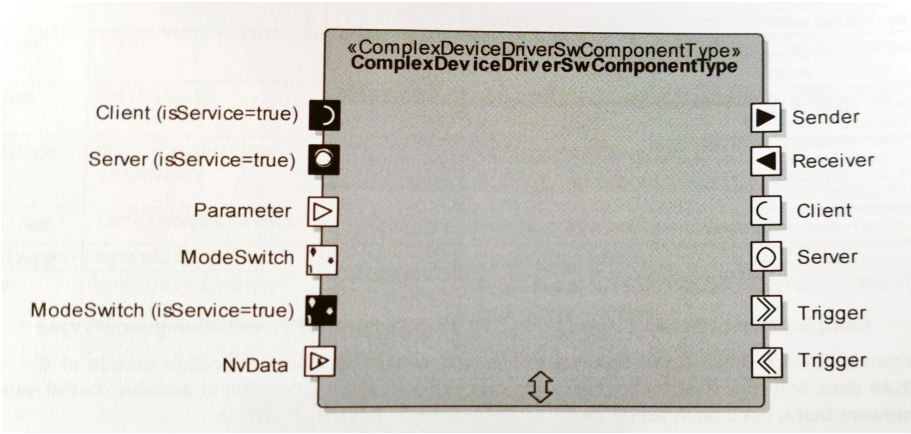


Figure 4.9: Symbol for Complex Device Driver Software Component [10].

The Service Proxy Software Component, `ServiceProxySwComponentType`, is a Component that acts as a proxy provider access to Internal Services for one or more remote ECUs [10].

The main use case for this SWC is to distribute a service throughout the system where a Mode Manager is part of the Basic Software.

A `ServicesProxySwComponentType` is similar to an `ApplicationSwComponentType`, where the only difference being that the `ServiceProxySwComponentType` is instantiated during the system design into several ECUs [10]. In the System Design the Service Proxy SWC will be different since it will be connected as a 1:n ratio where the typical Application SWC are in a 1:1 ratio.

The following diagrams will represent how this SWC is represented in both VFB, Figure 4.10, and Implementation, Figure 4.11.

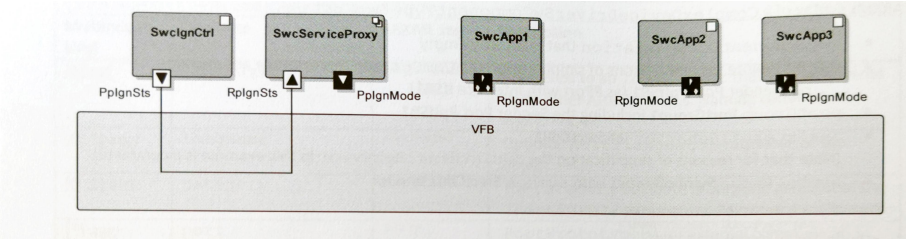


Figure 4.10: Virtual Functional Bus view of Service Proxy Software Component [10].

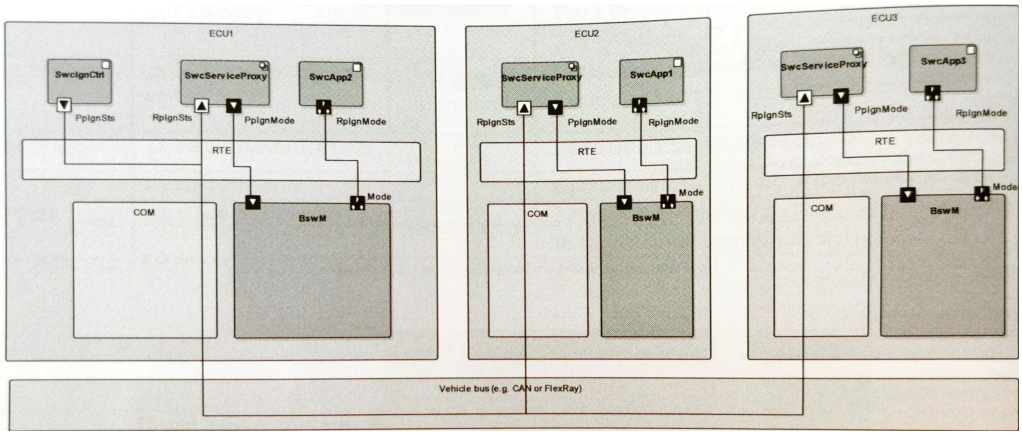


Figure 4.11: Connection diagram for the usage of Service Proxy Software Component [10].

The following symbol, Figure 4.12, is a graphical representation of `ServicesProxySwComponentType`.

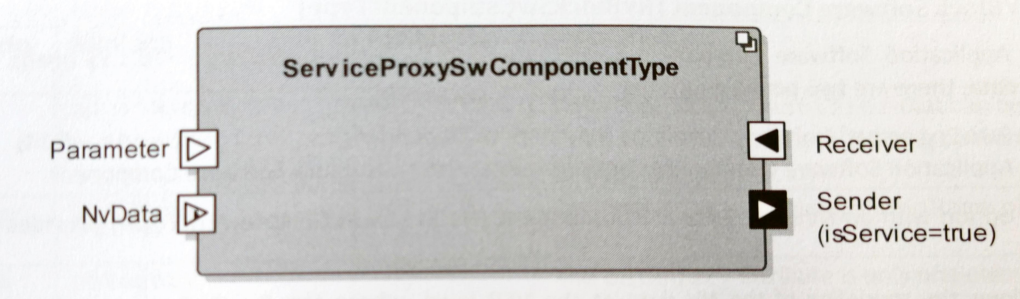


Figure 4.12: Symbol for Service Proxy Software Component [10].

The NVBlock Software Component, `NvBlockSwComponentType`, is a component type specially designed to allow the Application SWC to be able to interact with the Non-Volatile Memory, NVM.

This is not the only approach that the Application SWC can use to store or read data from the NVM. The other approach is to use the Services available in the NVM Block [10].

By using this SWC Type there are two main advantages:

- it allows the modelling of the NV data at the VFB leve;
- it can be used to reduce the number of `NvBlocks`
  - one block can be used to store different small data elements
  - the same data elements can be used by different `SwComponentTypes`

The `NvBlockSwComponentType` maps indvidual non-volatile data elements to `NvBlocks` and interacts with the NVRAM Manager. The implementation of this SWC is solely generated.

An `NvBlockSwComponentType` can also have `Ports` with CS Intefaces that are meant to interact witht the NVRAM Manager via Standardized AUTOSAR Interfaces [10], for the following Services:

- NvMService to send commands to the NVM.
  - EraseBlock
  - GetDataIndex
  - GetErrorStatus

  - InvalidateNvBlock
  - ReadBlock
  - RestoreBlockDefault
  - SetDataIndex
  - SetRamBlockStatus
  - WriteBlock

- NvMNotifyJobFinished to notify the end of job.
  - JobFinished
- NvMNotifyJobInitBlock to request the application SWC to provide the default values in the RAM mirror.
  - InitBlock
- NvMAdmin to invoke some administrative operations
  - SetBlockProtection

The following symbol, Figure 4.13, is the graphical representation of a `NvBlockSwComponentType`.

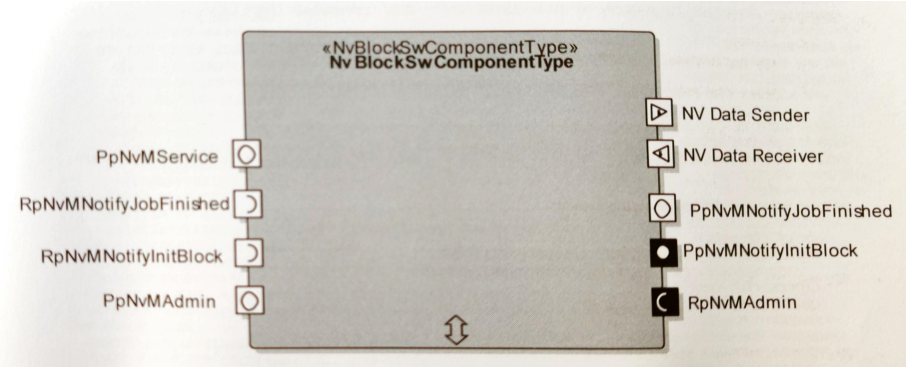


Figure 4.13: Symbol for NVBlock Software Component [10].

4.2.1.2 Parameter Software Component

The Parameter Software Component, `ParameterSwComponentType`, previously called Calibration Parameter Software Component is an SWC that provides parameter values which can be fixed data, const and variable [10]. In contrast to `AtomicSwComponentType` it can not have a Internal Behaviour and does not support *RPort*s [10].

A `ParameterSwComponentType` is simplisly a container that will provides parameters to other SWC via Parameter Ports.

The following symbol, Figure 4.14, is a graphical representation of `ParameterSwComponentType`.

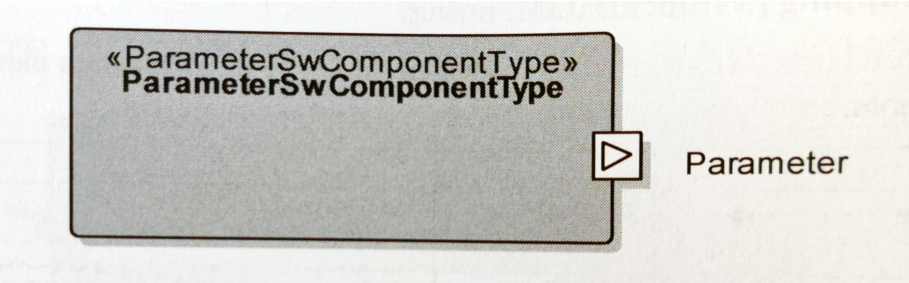


Figure 4.14: Symbol for Parameter Software Component [10].

4.3 Communication between AUTOSAR Components

Like in any Software project not every file will contain all the code, since splitting the code in diverse files and creating diverse functions is a better option to be able to organize the code implementation. This allows to group everything that shares the same type of family into the same file or API. For this to be affective there is a need to manage the communication between components. In AUTOSAR there is already specified how the communication between Software Component should look like.

In this section of the dissertation it will explain how the communication between SWC is done and where they are specified.

The ARXML files contains everything regarding the component, from it skeleton, its internal behavior, the communication type and flow of the communication.

For each component, there will be some AUTOSAR syntax that will tell to the developer the possible inputs and outputs to the component. This information normally is located between the tag *Port*.

In AUTOSAR there exist mainly two types of port types[10]:

- Provide Port;
- Require Port.

The Provide Port, or *PPort* to abbreviate, is the port that will output data to the RTE. This type of port is necessary if a component needs to provide some kind of data to another component.

The Require Port, *RPort*, is the input port for the component. Here is where the component will read a determinate value from the RTE.

In rare cases, there is used a mix of these two port types. This type of port can provide and receive data, this port has the name Provide-Require Port, released on AUTOSAR Classic 4.3. However, the use case for this type of port is new and often it is still not used for it possible functionality.

This type of ports are more common to be used in SWC or CDD, where they have direct access to the RTE and the main communication is done via the RTE.

As mentioned in Section 3.3, the common interface category used for Application SWC is **AUTOSAR Interface** and in some cases when there is a need to access to a service provided by the BSW it is used the **Standardized AUTOSAR Interface**.

In order for a port, either require or provider, there is a need to exist a Port Interface, `PortInterface`. It defines the contract that must be fulfilled by the ports [10]. Each Port can have only one `PortInterface` assigned. There are four basic types and three subtypes as the following list [10]:

- Data Interface
  - Sender-Receiver Interface
  - Parameter Interface
  - Nonvolatile Data Interface
- Client-Server Interface
- Trigger Interface
- Mode Switch Interface

Each of the previous types of `PortInterface` can be used for *PPort* and *RPort*. In these interfaces, there is not specified whether the port communication is Queued or Non-Queued. This definition is done in the Communication Specification, `ComSpec`, where the *Port* is created [10].

A *Port* can interact with any Service provided by the BSW with any of the previously mentioned `PortInterface` only if the parameter `isService` is TRUE [10]. To configure the type of service that the *Port* will interact is needed to give a value to the attribute `serviceKind` [10]. When a `PortInterface` has the

parameter `isService` set, makes that this interface could be inserted into the interface category **Standardized AUTOSAR Interface**.

In order to be able to connect *Port*s is by using a `SwConnector`, there is needed that the following points are met:

- `PortInterfaces` and `AutosarDataTypes` must be compatible;
- Both Ports needs either to be Service Ports or not (attribute `isService`).

To ease the process of search to find which kinds of interfaces are compatible and to speed up the development process there exist tables that shows how the different types of port interface are compatible. The following table, Table 4.4 is one of the types that could ease up the process of development.

PortInterface		Require					
		Receiver	Parameter	NV Data	Client	Trigger	Mode Switch
Provide	Sender	Yes	No	Yes	No	No	No
	Parameter	Yes	Yes	Yes	No	No	No
	NV Data	Yes	No	Yes	No	No	No
	Server	No	No	No	Yes	No	No
	Trigger	No	No	No	No	Yes	No
	Mode Switch	No	No	No	No	No	Yes

Table 4.4: Port Interface compatibility [10].

During the implementation of SWC and during the software production, there will be cases that some ports are not connected. This can happen if either the provider port is not created or if the require port is not created.

To allow for the development to continue, simply because the responsible haven't align every single topic in advance there are there is the possibility to use Port Communication Specification [10]. This Specification is mandatory for the require port [10], mainly to give some init value and allow to have the implementation done in advance. Normally these Init Values should be some value that will trigger a condition of not initialize and in that way the SWC would know in advance not to use that value before consuming CPU resources.

Also, another particular utilization of these specifications is to define how the data is being sent either as a Queued format or NonQueued. This is normally used on the case of Sender-Receiver Interfaces.

4.3.1 Client-Server Interface

From all the Port Interfaces the Client-Server Interface, CS, is one of the most used interfaces. In this interface, there is some easier way to distinguish which

is the PPort and RPort. Here all the PPort will be known also as Server and all the RPort will be known as Clients [10].

Another particular situation using this type of interface is that it is possible to have multiple client ports to **only** a server port (1:n clients) [10].

The main reason that CS Interfaces are used so often is giving to the fact of how the interaction of the different SWC is done. This interface allows the Client to call an operation or a function that is located on the Server [10]. It is very useful because it allows the System Engineer to plan all the tasks corresponded to a type of functionality to be done in a unique SWC. A simple example would be the monitoring of error to trigger some type of **Data Trouble Code** that will be saved on the memory of the ECU. Instead of having multiple components to interact with the diagnostic stack it is possible to restrict that type of access using one SWC that will gather all the application errors and interact with the diagnostic stack and the memory stack.

A call to operation via CS Interface can be either blocking or non-blocking, where synchronous communication is referent to blocking and asynchronous for non-blocking [10]. In both cases, the client waits until the response from the server. The type of communication is represented on the client-side, where it turns possible that the server can be called by both synchronously and asynchronously [10].

For this type of communication, the normal return-value will always be an RTE Error Code. This means that for a client to receive any values via this interface it needs to give arguments [10]. Via those arguments, the processed value will be return, e.g. Unit Conversion SWC with a CS Interface that will receive 3 parameters value, current unit and desired unit.

On RTE Generation the generator will create a list of all the Error Codes that are applied to that CS Interface. With this information, it is possible to create some type of filter to invalid operation calls. This will allow the client SWC to have a more robust implementation.

It is also possible to configure how long a queue of requests for a certain operation. On the server-side the queue must be defined to a value higher or equal to 1 [10]. This means to the number of asynchronous operation calls is being done for that server operation. If the number of the queue is equal to 1 then when there are 2 clients to try to call the operation the server will only respond to one where the other will be exited with an Error Code, normally will be a Timeout Error. The response of the server follows the method *First In First Out*, FIFO [10]. On the client-side the queue must be 1 since it is the one that request the operation.

For details of the Synchronous and Asynchronous CS Communication see Chapter 4.4.

4.3.2 Sender-Receiver Interface

The AUTOSAR concept of Sender-Receiver Interface, SR, is used to exchange data in an asynchronously and in one direction way between a Software Component and another one or a Basic Software Module [10]. In this type of interface the sender is not blocked and neither expects or receive any response, from the receiver components. On the receiver side, it decide asynchronously when it needs the data and how it will use it, making the flow of this type of communication-based on the receiver needs.

Here the Sender Port is a PPort, providing the data to one or more receivers [10]. The Receiver Port is an RPort, receiving the data from one or multiple components [10].

This makes this type of interface totally different from the CS, were the CS could only have communication 1 server to **n** clients, and here the same interface can have multiple provider ports and multiple require ports.

For the correct use of this interface there is the need of a clear understanding of which Data Types are in use in this type of interface. This impacts mainly in the way to fulfill the communication. There cannot be 2 components that will provide the same type of data on the same interface.

On this type of interface the scheduling of when the send of data is not always on the same other as the receiver can receive. For example we can have a component that will send 3 data structures or 3 values. Since this type of communication is never done End-to-End, E/E, there is always the RTE in between it is possible that the order of arrival of the data sent by a component can differ. The following Figure 4.15, shows the send of three differents Data Type and their correspondent arrival order.

When it is necessary that multiple data need to be sent simultaneously they should be combined into the composite data structure, (`ApplicationDataType` Section 4.1.1). The sender will write the complete data structure with the required data before sending it to the RTE [10].

This type of interface allows that there could be more than one data struct to be send on it. As presented on Section 4.1.1, there are a lot of possible Data Types that could be used. This allows flexibility on how the data can flow between different AUTOSAR Components.

The Sender-Receiver Interface uses an RTE buffer that is generated during the RTE Generation. This buffer is initialized with a constant and when the Provider SWC writes on this buffer the data is updated. The buffer is created for all SR Interfaces Data Elements, this means that every SWC could write to a different Data Element of the same SR Interface, n:1 communication [10].

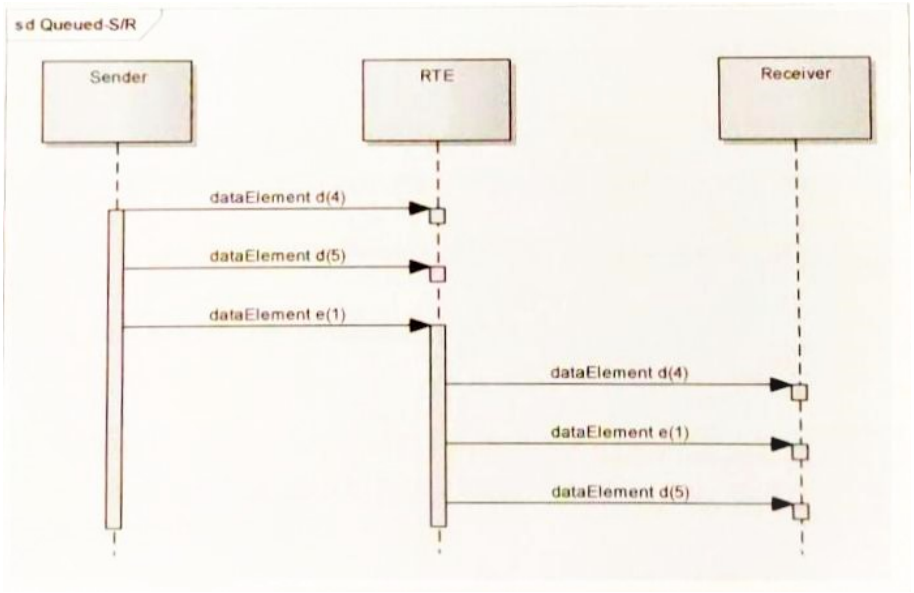


Figure 4.15: Non-deterministic sequence for SR communication [10]

The following figure, Figure 4.16, shows how two different SWC writes to the same Data Element of an SR Interface. The example is for an SR Interface with a Queued communication to allow that there is no loss of data.

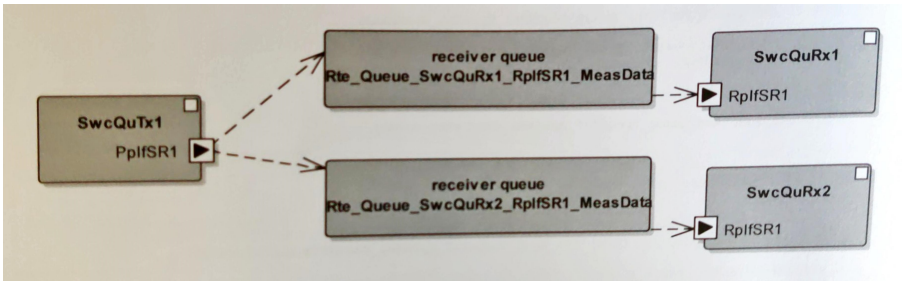


Figure 4.16: **1** Provider Software Component to **N** Receiver Software Component [10]

On the following figure, Figure 4.17, it shows how a Queued SR Communication is done using two different Queues for the two different SWC that will receive that data.

4.3.3 Mode Switch Interface

The Mode Switch Interface, MDS, is used to notify a Software Component of a mode switch [10] This type of interface commonly used interface to allow any

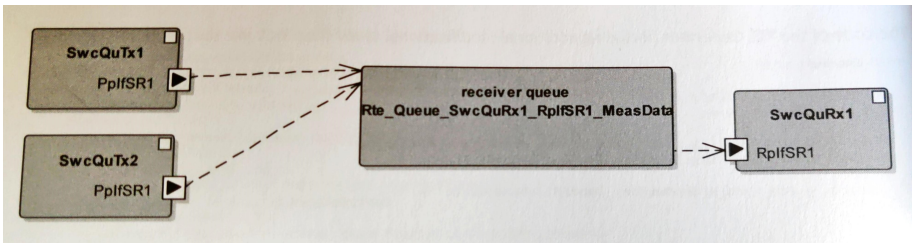


Figure 4.17: **N** Provider Software Component to **1** Receiver Software Component [10]

SWC on the Application Layer to now some state transition on the ECU, like for example on the Initialization process of an ECU. This interface can only be used inside an ECU. In case this information is needed on different ECUs this needs to be done by a `ServiceProxySwComponentType`.

This type of interface is similar do the Client-Server Interface since this can only have a connection of type 1 to any. Other types of communication, like the one described on the Sender-Receiver Interface, Section 4.3.2, is not allowed, for example N:1 and N:M communication.

4.4 Call of Implemented Code

The missing part to have an SWC implementation complete is how it will be linked with the code implementation. In order to create this link there is a need to have a `RunnableEntity`, Runnable, in the SWC ARXML. On AUTOSAR this runnable definition will create show the resources that in a specific task the function/s, that is been called by the system via the RTE Events, will be able to access.

In AUTOSAR, Runnables belongs to the group of Internal Behaviours of an SWC [10].

In other words a Runnable is the implementation of code inside the Atomic SWC. It is invoked by the RTE via the RTE Events, that could be timed based or event as a response of something being received by the component [10]. The resources that a specific Runnable can access is regarding the data is received by the Software Component, like for example data received on a `RPort` and the sending of data via a `PPort`.

The Runnables runs in the context of an RTE Task `AutosarComp`. The mapping of a Runnable into a task is done during the ECU configuration.

The following figure, Figure 4.18, shows how a Runnable is present in an SWC. Is to note that a Software Component can only have a `RunnableEntity` if

there is an Internal Behaviour in the SWC [10].

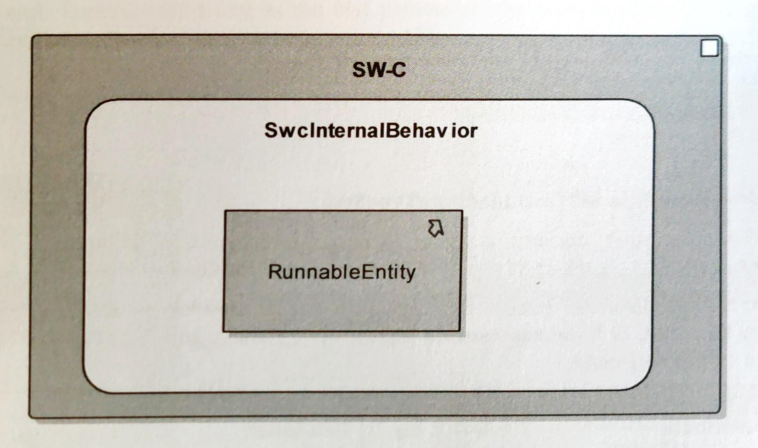


Figure 4.18: Runnable location inside a Software Component [10]

4.4.1 Access Points

As mentioned previously it is defined on the Runnable which kind of data it can interact with on the System. This is defined explicitly on the ARXML file of the SWC Access Points.

A Runnable can basically interact, via two different ways for communication:

1. Communication via Interrunnable Variables;
2. Communicating using the data outside the SWC via Ports.

These two different ways to interact with the system is done via Access Points.

An Access Point is simply a point of access to acquire data, invoke operations and providing data.

The following table, Table 4.5, shows the different kinds of communication with their respective tags.

Kind Of Communication	Access Point Tag	Direction
Interrunnable	`readLocalVariable`	IN
	`writtenLocalVariable`	OUT
Receiver	`dataReadAccess`	IN
	`dataReceivePointByArgument`	IN
	`dataReceivePointByValue`	IN
Sender	`dataWriteAccess`	OUT
	`dataSendPoint`	OUT
Client	`SynchronousServerCallPoint`	IN/OUT
	`AsynchronousServerCallPoint`	IN/OUT
	`AsynchronousServarCallResultPoint`	IN
Parameter	`ParameterAccess`	IN/OUT
Mode	`ModeAccessPoint`	IN/OUT
	`ModeSwitchPoint`	IN/OUT
Trigger	`ExternalTriggeringPoint`	IN

Table 4.5: Port accessing tags for a `RunnableEntity` [10].

Chapter 5

Guidance and Use Cases

This chapter will provide a detailed summary the AUTOSAR Architecture and some key points that needs to be highlighted. In this chapter, the main focus is to give beginner guidance of which steps need to be followed to create a Software Component or to make an analysis of an AUTOSAR layer, specially the Application.

AUTOSAR has a complex Architecture at first glance however this concept helps to create an Embedded System that is scalable and reusable, since application components needs to be independent from the microcontroller. The AUTOSAR Architecture is based in 3 main layers where the lower the location of the layer the more Hardware dependent it is. In a top-down approach the layers are:

- Application Layer;
- Runtime Environment;
- Basic Software Layer.

The main point that this dissertation highlighted about AUTOSAR is its concept of reusability of Software Components. This is only possible if the Application Software Component are independent from the Hardware, making the Application Layer as independent from the lower layers.

In some cases the Application Software Components has some particular implementation that is heavily dependent of the configuration of either the Basic Software or by the Hardware. This kind of situation should be minimal, to be the most concordant to the AUTOSAR Concept, and if possible this should not happen at all.

The Runtime Environment is generated code for a specific ECU. The code generator needs to be certified to be seen as a trustworthy tool. This tool uses various configuration files to generate the code. The most commonly used are the ARXMLs Files, AUTOSAR XML. There is also possible to have other file types that could be tool dependent, this other file serves as to configuration for the Basic Software Modules.

The last layer, Basic Software, resides the modules that are Hardware dependent. This layer provides services and libraries to higher layers in order to facilitate the usage of the microcontroller interaction.

On the Basic Software also resides the Operating System as well as all the standard services defined in AUTOSAR, such as Diagnostic, Memory and Communication.

To recap, the previous figure, Figure 3.3, summarizes the AUTOSAR Architecture, showing all the differents types of services that the Basic Software can provide.

The main layer approached in this dissertation is the Application Layer. It is here where all the Project Specific Application resides in for of Software Component, which will implement part of a functionality or even all of it.

During the implementation of Software Components in the Application Layer there are the following key notes to take:

- Software Component type;
- Provider/Require Ports;
- Port Interfaces;
- Compatibility of Interfaces;
- Data Types used in the Interfaces;
- Trigger for the Runnable;
- Runnables.

For the types of Software Components the most used are the standard Application Software Component and the Complex Device Driver. Both of them are simply to use and allow all the Port types making it an easy way to interact with other components on either Application or Basic Software layers.

The main difference of the usage of these two Software Component types is that the Complex Device Driver will be Hardware dependent since it needs to interact with the microcontroller. The Complex Device Driver is also used only

on a critical task like for example when the output needs to be executed with higher priority or when the task needs to be executed on the precise timing.

The Application Software Component is a downgraded version of the Complex Device Driver since it is restrictive in the communication. It needs to use the Runtime Environment in order to interact with other Application Software Components, Provider/Require Port or Basic Software Modules, via Service Ports.

All the relevant information regarding input data, output data and how a behavior starts is available on an ARXML file. This file is the description of a Software Component and inside this file it is possible to see the "skeleton" of the Software Component. This "skeleton" has only the information needed in order to generated the Runtime Environment code necessary to interact with the Component. In this file it is possible to observe all the Ports that the component has, the Port Interfaces that the Port has, the Data Type for some specific Interface Types that are created on the Software Component and as well the Runnable that it contains.

There are a couple of Port Interfaces that are commonly used and, in my opinion, needs to be memorized what they are and their use cases. These interfaces are the Sender-Receiver Interface and Client-Server Interface.

To summarize the Sender-Receiver is commonly use when sending data from one Software Component to another. Here there is not the need to evaluate how the data is send or if the data is consume on the other receiver.

On the receiver Software Component the data can be handle by different means. For example the receiver Software Component could only want to read the data that it received on a context of a Runnable. Here it is only needed to have an Access Variable point on the Runnable where the data will be consumed and processed.

Other use case for this type of interface would be that when receiving a data tit should trigger a Runnable. This is done with an RTE Event, namely Data Receive Event. In this use case it is possible to start an operation or a series of operations when the data is received.

The other commonly used Port Interface is the Client-Server. This type of interface serves to call a functionality or an operation from another Software Component. The Client Software Component is responsible to request action to the Server Software Component. On this request there could be the passed a variable in case that there is needed to pass some type of data to the Server. The variable that is passed could also serve as an output of the operation run on the Server Software Component.

A possible use case for this type of Port Interface would be the interaction with Data Trouble Code, DTC. There could be a Software Component that act as

a Server to all Application Software Components. On this Server there could be various operations such as:

- Read state of DTC;
- Set state of DTC;
- Clear DTC;

This approach simplifies how the analysis of a certain type of functionality of a System. In a way having Software Component specialized on only a type of task is always better and turns the System more modular and so complying with the AUTOSAR concept.

Another key point to take into consideration is the interface compatibility. This one key notice needed to know if the two different Ports can be connected in order to interact with each other.

As explained previously, in Section 4.3, some different interface types can still interact with each other.

Another topic to take into consideration is the Data Type used on the interfaces. This small topic can also help to identify if the interfaces are compatibles, since each interface needs to be able to receive and send the same type of data.

A use case for the Interface compatibility could be seen as having three Software Component, sharing three Ports with the same Interface. In the interface definition it is possible to see two Data Types. Two of the three Software Components will have a Provider Port using the same interface however each one of them will send a different Data Type. The last Software Component will have a Required Port. This is a valid scenario if and only if the Data Types are alligned as per the definition of the Interface.

This type of information is already sufficient to start designing the communication flow of a feature. However this is only possible if the starting point is already known, for example we could use a CAN Message to start the analysis.

This way of system analysis inside an ECU is something that needs to be done with calmly and it will take some time to get used to. This process helps to understand most of the AUTOSAR concepts in the Application Layer.

For an analysis of the system the point that I suggest, after writing this dissertation, are:

1. Communication flow via the Port and Port Interfaces;
2. Trigger point for the Runnable (RTE Events);

3. Usage of the data inside the Runnable (Access Variable);
4. Function that is called by the Runnable.

Chapter 6

Conclusion

This chapter will conclude this dissertation revealing some difficulties while writing this dissertation and some possible improvements for this dissertation.

The main struggle with this theme is that it is very complex having a lot of topics that could be referred to, like for example the Basic Software components and if wanted to go to a certain detail even it might have been possible to explain only a Basic Software Stack. AUTOSAR is one methodology that is in a constant evolution where most of the new concepts are still being improved and discussed with all parties that uses and improves AUTOSAR, starting by Car OEM and Tier 1 to IT Companies.

The other struggle faced during the investigation and the writing is the confidentiality topics. Here it ended up not possible to provide any concrete functional example simply because all the projects that has AUTOSAR as a methodology used are extremely classified, making in the end this dissertation mostly theoretically.

Considering this points and thinking the only point that could be improved would be trying to create some Software Components using AUTOSAR for some theoretical functionality. This was not possible giving the time constraint.

Other than coming up with an example like state before the other point that could be improved would be the scope of the dissertation. With this, the scope could be more detailed in certain topics, like for example, Run Time Environment and/or Basic Software and their stack.

However, this dissertation presents a summary of AUTOSAR for those who are starting with software development in the Automotive Industry. It also presents a guide and use cases that could be an asset for new developers.

Chapter A

Annex

RTE Return Error Codes	Value Hex Value Dec	Description
`RTE_E_OK`	0x00 0d	No Error
`RTE_E_LOST_DATA`	0x40 64d	If new data is received and the queue is already full, then the RTE discards the new data and sets an error flag. For the next read on the queue, the `Rte_Receive` call returns the available data together with a status where the flag `RTE_E_LOST_DATA` is set.
`RTE_E_MAX_AGE_EXCEEDED`	0x40 64d	An `Rte_Read` or `Rte_IStatus` call indicates that the available data has exceeded the `aliveTimeout` limit.
`RTE_E_COM_STOPPED`	0x80 128d	An IPDU group was disabled while the application was waiting for the transmission acknowledgment. No value is available. This is not necessarily considered a fault.

Table A.1: `Std_ReturnType` predefined error code (Part 1)[10].

RTE Return Error Codes	Value Hex Value Dec	Description
RTE_E_TIMEOUT	0x81 129d	A blocking API call (Rte_Receive, Rte_Call, Rte_SwitchAck) returned due to expiry of a local timeout. OUT buffers are not modified.
RTE_E_LIMIT	0x82 130d	An internal RTE limit (like queue size) has been exceeded (e.g. for Rte_Send, Rte_Call, Rte_Switch, Rte_Trigger, Rte_IrTrigger). OUT buffers are not modified.
RTE_E_NO_DATA	0x83 131d	No data was available for the API call. This is not (necessarily) to be considered as an error. OUT buffers are not modified.
RTE_E_TRANSMIT_ACK	0x84 132d	Transmission acknowledgment received.
RTE_E_NEVER_RECEIVED	0x85 133d	No data received since system start or Partition restart.
RTE_UNCONNECTED	0x85 133d	The corresponding Port used for communication is not connected.
RTE_E_IN_EXCLUSIVE_AREA	0x87 135d	The RunnableEntity could not enter a wait state because of another RunnableEntity of the current Task call stack is running in an ExclusiveArea.
RTE_E_SEG_FAULT	0x88 136d	The parameters contain a direct or indirect reference to memory that is not accessible from the caller's Partition.

Table A.2: Std_ReturnType predefined error code (Part 2)[10].

References

[1] S. D. A. Museum. Automotive history. https://sdautomuseum.org/automotive-history. Accessed: 2019-04-28. [Quoted on p. iii, 6]

[2] P. Goverment. Automotive industry. http://www.portugalin.gov.pt/automotive-industry/. Accessed: 2019-07-27. [Quoted on p. iii, 7, 8]

[3] V. I. GmbH. The autosar development partnership. https://www.vector.com/int/en/know-how/technologies/autosar/autosar-classic/. Accessed: 2020-01-26. [Quoted on p. iii, 9]

[4] O. G. Stephen Waldron. Introduction to autosar. https://bit.ly/2MQLcEY. Assited the Webinar on 2019-08-31. [Quoted on p. iii, 9, 10, 11]

[5] A. Organization. Adaptive platform release overview r19-11. https://www.autosar.org/fileadmin/user_upload/standards/adaptive/19-11/AUTOSAR_TR_AdaptivePlatformReleaseOverview.pdf. Accessed: 2020-01-11. [Quoted on p. iii, 12, 13]

[6] ——. Classic platform release overview r19-11. https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_TR_ClassicPlatformReleaseOverview.pdf. Accessed: 2020-01-11. [Quoted on p. iii, 12, 13]

[7] ——. Layered software architecture. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf. Accessed: 2019-12-27. [Quoted on p. iii, 15, 16, 17, 18, 19]

[8] N. Naumann. Autosar runtime environment and virtual function bus. https://hpi.de/fileadmin/user_upload/fachgebiete/giese/Ausarbeitungen_AUTOSAR0809/NicoNaumann_RTE_VFB.pdf. Accessed: 2020-02-22. [Quoted on p. iii, 16, 19, 20, 21, 22]

[9] A. Organization. (2017) Virtual funtional bus. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_VFB.pdf. Accessed: 2020-02-22. [Quoted on p. iii, 19, 21]

[10] O. Sheid, *AUTOSAR Compendium - Application and RTE*, Bruchsal, DE, December 2015. [Quoted on p. iii, iv, v, 15, 16, 20, 21, 22, 24, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 57, 58]

[11] V. I. GmbH. Introduction to autosar | autosar motivation. https://elearning.vector.com/mod/page/view.php?id=438. Accessed: 2020-01-26. [Quoted on p. 1]

[12] ——. Introduction to autosar | autosar initiative. https://elearning.vector.com/mod/page/view.php?id=439. Accessed: 2019-09-29. [Quoted on p. 2, 11]

[13] I. Engineering. The history and evolution of the wheel. https://interestingengineering.com/history-and-evolution-wheel. Accessed: 2019-04-28. [Quoted on p. 5]

[14] M. Bellis. How do steam engines work? https://www.thoughtco.com/steam-engines-history-1991933. Accessed: 2019-04-28. [Quoted on p. 5]

[15] H. Editors. Automobile history. https://www.history.com/topics/inventions/automobiles. Accessed: 2019-05-05. [Quoted on p. 6, 7]

[16] D. T. Kurylko. Model t had many shades black dried fastest. https://www.autonews.com/article/20030616/SUB/306160713/model-t-had-many-shades-black-dried-fastest/. Accessed: 2020-03-01. [Quoted on p. 6]

[17] A. Organization. History. https://www.autosar.org/about/history/. Accessed: 2019-09-28. [Quoted on p. 9]