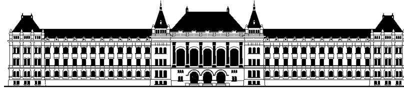


Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Lőz Dávid

AUTOSAR END-TO-END COMMUNICATION PROTECTION LIBRARY MODUL MEGVALÓSÍTÁSA

KONZULENS

Dr. Sujbert László, habilitált docens

(Méréstechnika és Információs Rendszerek Tanszék)

Szikszay László

thyssenkrupp Components Technology Hungary Kft

BUDAPEST, 2020

SZAKDOLGOZAT-FELADAT

Lőz Dávid (HM1L6O)

szigorló villamosmérnök hallgató részére

AUTOSAR End-to-End Communication Protection Library modul megvalósítása

A modern gépjárművek biztonságtechnikai és kényelmi funkcióinak megvalósításában, környezetvédelmi jellemzőinek javításában stb. egyre jelentősebb szerepet kapnak a számítástechnikai megoldások. Ma egy prémium személyautó gyártójának közel 150 elektronikus vezérlőegységből (ECU) és számos fedélzeti kommunikációs sínből kell kialakítani egy megbízhatóan működő elosztott rendszert, amely komoly algoritmus- és kommunikációtervezési, illetve munkaszervezési kihívást jelent. Az így adódó komplexitás uralására alakultak ki különféle szabványok, pl. a megbízható kommunikáció biztosítására a CAN és FlexRay sínek, a valós idejű feladatok futtatására az OSEK operációs rendszer vagy a futási idejű monitorozást támogató XCP protokollcsalád. A vezető autógyártók által 2002-ben életre hívott AUTOSAR konzorcium célja az, hogy ezen szakterületi szabványokra építve specifikáljon egy (i) *alapvető szolgáltatásstruktúrát,* amely eltakarja a hardver sajátosságait és támogatja az alkalmazási szoftver hordozhatóságát (base software stack, BSW), (ii) *egy modellezési nyelvet* az ECU-kon futó alkalmazási szoftver szabványos leírására (software component template), és (iii) az alkalmazások és BSW-k ECU-n belüli és ECU-k közti *transzparens kommunikációját* lehetővé tevő elosztott runtime szolgáltatást (RTE):

- A *base software stack* magában foglalja az alacsony szintű eszközmeghajtókat (pl. EEPROM és Flash driverek), az ezeket eltakaró absztrakciós rétegeket (pl. memória absztrakciós felület) és az ezekre ültetett magas szintű funkciókat (pl. perzisztens adattárolás).
- A *modellezési nyelv* lehetővé teszi, hogy precízen specifikáljuk az adattípusokat, illetve az alkalmazást alkotó komponensek interface-eit és belső felépítését.
- Az *RTE* egy generált glue kód réteg, amely eltakarja az alkalmazáskomponensek elől, hogy az általuk fogadott vagy küldött információ pontosan hogyan jut el a forrástól a célig, potenciálisan ECU-k közötti kommunikációs buszok igénybevételével.

A konzorcium jelentős hangsúlyt fektet az *API-k szabványosítására*, de kifejezetten támogatja a versengést az egyes szolgáltatások *megvalósításában* („Cooperate on standards, compete on implementation"). Az AUTOSAR egy élő, aktívan fejlesztett szabvány, amelynek évente jelenik meg újabb verziója.

A jelölt feladata az AUTOSAR Base Software Stack egy library moduljának a megvalósítása 4.4 szabvány verzió alapján az alábbiak szerint:

- *A szabvány kapcsolódó részeinek megismerése*: ismertesse az AUTOSAR rétegzett BSW struktúráján belül a kommunikációért felelős modulok szerepét. Foglalja össze az E2E Library modul fő funkcióit és kapcsolatát a környező entitásokkal.
- *Szoftvertervezés és megvalósítás*: Első lépésben elemezze a modul követelményeit megvalósíthatóság szempontjából. A szabvány a legtöbb modul megvalósítását egy *statikus* (kézzel írt, minden konfigurációban azonos) és egy *dinamikus* (konfigurációtól függő, tipikusan generált) részre bontással javasolja és megadja a konfigurációs adatok modelljét egy osztálydiagrammal. Az E2E Library modulnak nincsenek konfigurációs beállításai (hasonlóan más library modulokhoz), így dinamikus rész előállítása nem szükséges. *Tervezze* és *valósítsa*

meg a modul *statikus részét* C nyelven; rendelje forráskódhoz az implementált követelményeket.

- *A megvalósítás tesztelése*: A modul helyességének vizsgálatához (i) hozza létre a modulteszt-infrastruktúrát, melyben tervezzen és valósítson meg teszteseteket, (ii) futtassa a teszteket, és győződjön meg arról, hogy megvalósítása megfelel a szabvány által elvártaknak, (iii) szükség esetén javítsa a megvalósítást.

Tanszéki konzulens: Dr. Sujbert László, egyetemi docens
Külső konzulens: Szikszay László (thyssenkrupp Components Technology Hungary Kft.)

Budapest, 2020. október 11.

……………………
Dr. Dabóczi Tamás
tanszékvezető

Tartalomjegyzék

Összefoglaló **7**

Abstract **8**

1 Bevezetés **9**

2 Az AUTOSAR szabványcsalád **11**

2.1 Kialakulása 11

2.2 AUTOSAR architektúra 12

2.3 Basic Software réteg 13

2.3.1 Services Layer 14

2.3.2 ECU Abstraction Layer 14

2.3.3 Microcontroller Abstraction Layer 14

2.3.4 Complex Drivers 15

2.3.5 A BSW logikai felosztása 15

2.4 Kommunikáció az autóiparban 16

2.4.1 Kommunikáció a modern gépjárművekben 17

2.5 Kommunikációs protokollok 18

2.5.1 Controller Area Network 18

2.5.2 FlexRay 20

2.6 AUTOSAR, kommunikáció a rétegek között 21

2.7 Communication Stack 22

3 End-To-End Communication Protection Library **24**

3.1 Biztonsági funkciók autóipari szoftverben 24

3.2 Az E2E modul elhelyezkedése a rétegzett architektúrában 26

3.3 E2E Communication Protection Library működése 27

3.3.1 E2E Library modul felhasználása 27

3.3.2 A modullal szemben támasztott követelmények 28

3.4 E2E Library implementáció 29

3.4.1 E2E Library típusok 29

4 Megvalósítás **35**

4.1 Követelménylista 35

4.2 Statikus forráskód 35

4.2.1 Eclipse fejlesztőkörnyezet 35

4.2.2 Adattípusok....36
4.2.3 E2E Library 4.4 API-k....36
4.2.4 Különbségek az E2E Library profilok között....45
4.2.5 Doxygen dokumentáció....47
4.3 Dinamikus forráskód....48
4.4 Tesztelés....49
4.4.1 V-modell....49
4.4.2 Unit testing....50
4.4.3 Tesztesetek....51
5 Tapasztalatok és további lehetőségek....54
5.1 Tapasztalatok....54
5.2 Továbbfejlesztési lehetőségek....56
5.3 Köszönetnyilvánítás....57
Irodalomjegyzék....58

HALLGATÓI NYILATKOZAT

Alulírott **Lőz Dávid**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2020. 12. 11.

...……………………………………………

Lőz Dávid

Összefoglaló

Manapság az autóipari fejlesztések során folyamatosan növekszik a szoftverfejlesztés fontossága, köszönhetően a mai, modernkori gépjárművekkel szemben támasztott követelményeknek, úgy mint kényelmi és biztonságtechnikai funkciók, környezetvédelmi aspektusok. Ezen rendkívül összetett feladatok megoldására jelentős részben a jármű belső hálózatát is alkotó ECU-k (Electronic Control Unit) a felelősek.

Ahogy kezdtek egyre komplexebbé válni a járművek belső rendszerei és hálózatai, szükségszerű volt szabványosítani az autóiparban történő szoftverfejlesztést. Emiatt született meg az AUTOSAR szabványcsalád a nagy autóipari cégek közreműködésével. Ennek segítségével részegységekre lehet bontani a fejlesztést, elfedve például a különböző hardverek sajátosságait.

Szakdolgozatom keretein belül az E2E Library modul megvalósítása volt a feladatom, amely az AUTOSAR Libraries részegységében található. Ebből kifolyólag először magával a szabvánnyal ismerkedtem meg, kifejezetten az AUTOSAR 4.4-es verziószámú End-To-End Communication Protection Library funkcióival.

A szakdolgozat írása alatt lehetőségem adódott elmélyülni a követelmény alapú szoftverfejlesztésben, első lépésként a szabvány specifikációja alapján követelménylistát készítettem, majd az ezeken történő megvalósíthatósági vizsgálat elvégzése után a modul profiljainak implementálása következett. Mivel beágyazott környezetben nagyon fontos a szoftverek optimalizált működése, ezért számos olyan helyzettel találkoztam, amikor mérnöki döntést kellet hozni a legjobb megoldás kiválasztásával, így rengeteg hasznos tapasztalattal lettem gazdagabb a modul fejlesztése során.

Az E2E Library modul 4.4-es verziójának profiljait sikeresen implementáltam, valamint modulteszt-infrastruktúrát készítettem, amivel ellenőriztem a helyes, szabvány által elvárt működést.

Abstract

Nowadays software engineering and development is getting more and more significant in the automotive industry, thanks to the emerging requirements (comfort, security and environmental issues) on the modern vehicles. For these extremely complex functions the ECU-s (Electronic Control Unit) are responsible and they are making up a significant part of the vehicle's internal network.

As the vehicles internal system were getting more complex, there was a need for standardizing the software development in the automotive industry. This is the reason why the significant automotive companies created the AUTOSAR standard family. With the help of this innovative standard solution, it is possible to break down the development into smaller phases which gives the possibility to work independently on different modules.

The goal of my thesis was to implement the E2E Library module, which can be found in the AUTOSAR Libraries. Therefore firstly I had to be familiar with the End-to-End Communication Protection Library version 4.4.

As working on my bachelor thesis, I had the opportunity to take a deep dive into the requirement based software engineering. For the first step, my task was to create the requirement list based on the standard, on which I completed the necessary check regarding the implementation. As far as we are talking about embedded systems, it is crucial to optimize the software, thus I have had to make engineering decisions with the aim of choosing the best solution. As a result of that, I gained a huge amount of experience in the field of software engineering.

As a result of my work on my thesis, I have successfully implemented the E2E Library module's profiles, and created a module test-infrastructure, with that I could make sure of the correct functionality of the E2E Library version 4.4.

1 Bevezetés

Manapság az autóipari fejlesztések során folyamatosan növekszik a szoftverfejlesztés fontossága, köszönhetően a mai, modernkori gépjárművekkel szemben támasztott követelményeknek, úgymint kényelmi és biztonságtechnikai funkciók, környezetvédelmi aspektusok. Ezen rendkívül összetett feladatok megoldásáért jelentős részben az ECU-k a felelősek.

Ahogy kezdtek egyre komplexebbé válni a járművek belső rendszerei és hálózatai, szükségszerű volt szabványosítani az autóiparban történő szoftverfejlesztést. Emiatt született meg az AUTOSAR szabványcsalád a nagy autóipari cégek közreműködésével. Ennek segítségével részegységekre lehet bontani a fejlesztést, elfedve például a különböző hardverek sajátosságait.

A Library modulok fejlesztése esetén, a többi AUTOSAR modul fejlesztésével ellentétben, nem szükséges dinamikus részt előállítani, tehát nem használunk kódgenerátort, így csak statikus forráskód megalkotása a fejlesztő feladata. Ennek első lépéseként mélyebben megismertem az AUTOSAR szabvány 4.4-es kiadásában szereplő E2E Library modult (a Basic Software Stackkel már az Önálló laboratórium és a szakmai gyakorlat alatt lehetőségem adódott megismerkedni). Ezután követelményelemzést végeztem, mely során a szabvány által felsorolt követelmények implementálhatóságáról döntöttem.

Ezt követte a szakdolgozatom feladatkiírásának legnagyobb kihívása, mégpedig magának az End-To-End Communication Protection Library modul C nyelven történő implementálása a szabvány követelményei alapján. Azokkal a modulokkal szemben, amikkel a korábbi munkáim során megismerkedhettem (Kriptográfiai stack, Secure Onboard Communication), az E2E Library nem a Basic Software Stackben helyezkedik el, hanem egyfajta különálló egységként nyújt API-kat az alkalmazásoknak. A modul működése azon az elven alapul, hogy a biztonságkritikus adatküldés során fellépő hibákat futási időben detektálni tudjuk, illetve védelmet is tudunk biztosítani a kommunikáció során fellépő hibákkal szemben.

Ennek érdekében az E2E Library különböző biztonsági szintek eléréséhez különböző profilokat kínál. Szakdolgozatom egyik hasznos tapasztalata az autóipari szoftverfejlesztéssel kapcsolatban a feladatok részegységre tagolásának megismerése volt. Igaz ez az egész AUTOSAR szabványcsaládra, ahogy a különböző funkciókat és komplex feladatokat részegségekre osztják, ezt hívják rétegzett architektúrának.

Ebbe tökéletesen beleillik a funkciók további szegmentálása, amit az ASIL (Automotive Safety Integrity Level) követelmények miatt biztosítania kell az E2E Librarynak. Tehát az egyes profilok működése fix, a szabvány azonban biztosít bizonyos konfigurációs lehetőségeket az API-k paraméterezésén keresztül.

Az E2E modul különböző védelmi mechanizmusokat kínál egyes profiljain keresztül, melyek alapja a küldött adathoz való plusz mezők csatolása (CRC, counter, stb.), melyek segítségével a fogadó oldalon (szintén az E2E Library meghívásával) biztosítani tudjuk az adatok helyességét, valamint az egyes hibás működéseket (például elvesztett adat) is kezelni tudjuk.

Ezen követelmények és feladatok megértése után kezdtem bele az egyes profilok implementálásába, amely a végére több ezer C kódsorrá duzzadt. Fontosnak tartom kiemelni, hogy az implementálás során lehetőség nyílt, beágyazott környezetről lévén szó, az optimalizációval foglalkozni. Ugyanis több utasítás vagy feltételvizsgálat esetén előre gondolkodva tudjuk segíteni a fordító munkáját, valamint a későbbiekben kevesebb erőforrás kerül majd ténylegesen felhasználásra a célhardveren történő futás során.

Az implementálás után egyes profilokhoz készítettem tesztkörnyezetet is annak érdekében, hogy az alapvető működés helyességét ellenőrizhessem. Ezáltal szükséges volt egyfajta tesztelői gondolkodásmód elsajátítása azért, hogy megbizonyosodjunk arról, hogy a modul esetleges hibás bemenetekre is megfelelően működik.

2 Az AUTOSAR szabványcsalád

A szakdolgozatom ezen fejezetében röviden bemutatom az AUTOSAR szabványcsalád létrejöttét, működését és architektúráját.

2.1 Kialakulása

Az AUTOSAR (**AUT**omotive **O**pen **S**ystem **AR**chitecture) egy szabványcsalád, melyet a nagy autóipari cégek (Daimler, BMW, Bosch stb.) hívtak életre 2002-ben. Célja az autóipari fejlesztések szabványosítása, ugyanis már egy bő húsz évvel ezelőtt is látható volt, hogy a technológiai fejlődés, valamint az igények növekedésének következtében a járművek egyre komplexebbé fognak válni. Ezen folyamatok eredményeként egy mai modern gépjárműben közel 150 ECU (Electronic Control Unit) található.

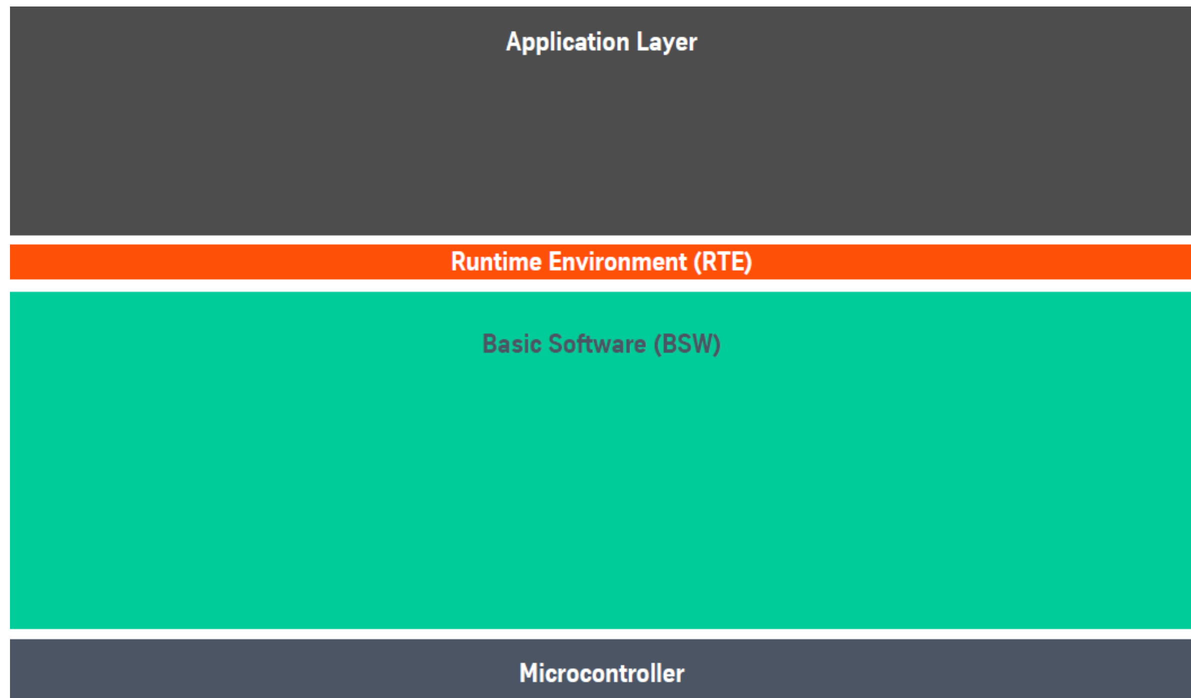
Ez a folyamat azóta sem állt le, sőt töretlenül növekszik, így egyre komplexebb és bonyolultabb rendszerként kell tekintetünk az autókra, melyek manapság az alapműködésen túl számos más funkciót is ellátnak, gondolok itt akár a vezetést támogató rendszerekre, biztonsági és kényelmi funkciókra.

Ebből kifolyólag tehát jogos volt az igény, hogy a különféle autóipari beszállítók termékeit gond nélkül integrálni lehessen a nagy gyártók járműveibe. A szabvány gyakorlatilag egy függvénykönyvtárt definiál, egyfajta API készletet biztosít az alkalmazásoknak.

Ugyanakkor fontosnak tartom kiemelni az AUTOSAR mottóját: *„Cooperate on standards, compete on implementation.”* Ez a mondat annyit jelent, hogy működjünk együtt a szabvány kifejlesztésén, de a magvalósítás alatt már érvényesüljön a piaci verseny. Ennek célja egyértelmű, hiszen minden cég érdeke elsősorban az, hogy nyereséges legyen, ezért a fejlesztés során muszáj versenynek fennállnia, ugyanakkor a szabványosítás segítségével egyfajta rendszerben elhelyezhetővé válik az autóipari szoftverfejlesztés.

2.2 AUTOSAR architektúra

Az AUTOSAR rétegzett architektúrájának köszönhetően (hasonlóan az OSI modellhez) jól elkülöníthetők az egyes modulok, ezáltal különálló logikai egységekbe szervezhetők az egyes feladatok megvalósításai.



1. ábra: AUTOSAR rétegzett architektúra [1]

Az 1. ábrán látható, hogyan kerül tagolásra egy AUTOSAR szoftver. Legfelül található az Application Layer (Alkalmazás réteg), alatta helyezkedik el a Runtime Environment – RTE (Futtató környezet), valamint alatta található a Basic Software (Alap szoftver) réteg.

Az Application Layer tartalmazza a szoftverkomponenseket (Software Components – SW-C), amelyek előre definiált portokkal kommunikálnak többek között a Basic Software réteggel az RTE-n keresztül. Az RTE gyakorlatilag független működést biztosít a szoftverkomponensek számára a kommunikációs mechanizmusoktól, egyfajta generált glue kód rétegként.

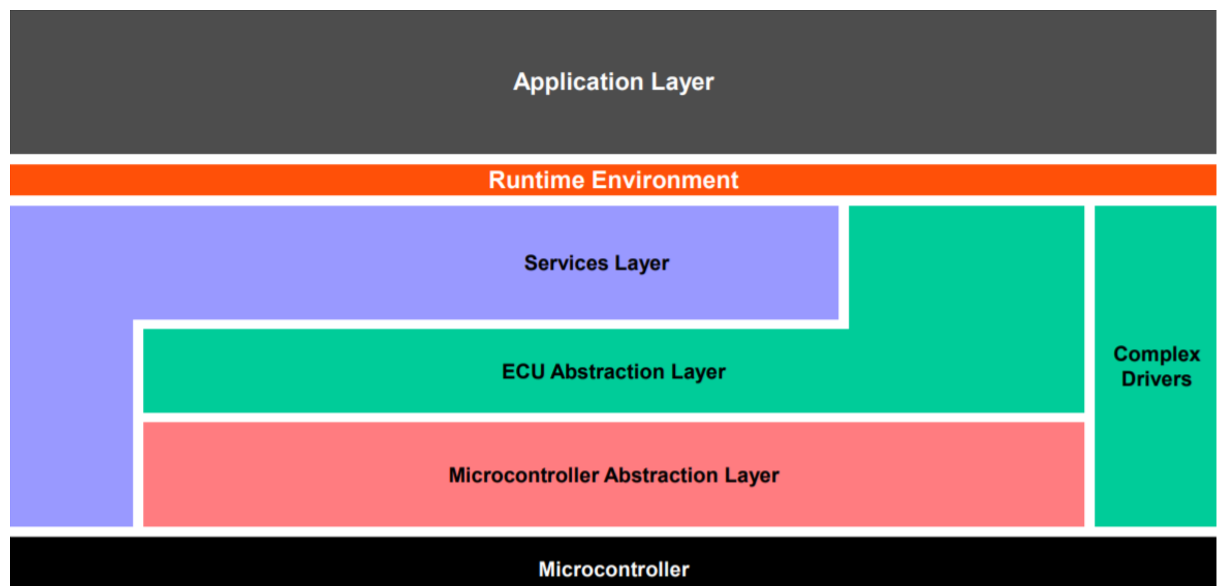
2.3 Basic Software réteg

A Basic Software réteg legfontosabb feladata, hogy elfedje a hardver sajátosságait az alkalmazás számára, úgy hogy egy standard interfészt biztosít. Maga a BSW számos logikai egységbe szegmentált modulból épül fel, amely által gyakorlatilag egy függvénykönyvtárat biztosít az RTE-n keresztül az alkalmazás rétegnek.

Ebből kifolyólag az architektúra jelentős részét teszi ki, ez azonban érthető is, hiszen az alrétegek segítségével számos modult magába foglal, melyek egy rendkívül komplex rendszert alkotnak.

A 2. ábrán ábrán láthatjuk, hogy a BSW mely négy fő részre kerül felosztásra, ezek a Microcontroller Abstraction Layer (Mikrokontroller absztrakciós réteg), ECU Abstraction Layer (ECU absztrakciós réteg), Services Layer (Szolgáltatás réteg), valamint a Complex Drivers (Komplex driverek).

Ez utóbbit a dolgozatomban még részletesen bemutatom, hiszen szoros kapcsolatban áll az E2E Communication Protection Libary-val.



2. ábra: BSW alrétegek [2]

2.3.1 Services Layer

A Szolgáltatások réteg a BSW stack tetején helyezkedik el, legfontosabb feladata, hogy a szoftverkomponensek részére mikrokontroller és ECU független módon biztosítsa az úgynevezett alapvető funkciókat, melyek az alábbiak:

- Memóriamenedzsment (NVRAM Management)
- Diagnosztikai szolgáltatások
- ECU-k közötti kommunikáció és hálózatmenedzsment
- ECU állapotmenedzsment
- Operációs rendszer

2.3.2 ECU Abstraction Layer

Az ECU Absztrakciós rétegre egyfajta interfészként tekinthetünk, amely az alatta található mikrokontroller absztrakciós rétegnek funkcióit biztosítja a szolgáltatás rétegnek, az ECU hardverspecifikus tulajdonságaitól függetlenül.

Az alegység magába foglalja adott esetben a driver modulokat is, amelyek külső perifériák meghajtásáért felelnek. Ez azonban akkor fordulhat csak elő, ha a konkrét vezérlőegységben található olyan periféria, amely nem a mikrokontrollerbe került integrálásra.

2.3.3 Microcontroller Abstraction Layer

A Mikrokontroller absztrakciós réteg a BSW stack legalján található. Lényegében hozzáférést jelent a hardverrel történő kommunikációhoz, ugyanakkor megalkotásának fő célja, hogy a felhasználónak hardverfüggetlen interfészt biztosítson.

Ez különösen nagy jelentőséggel bír, hiszen számos hardver helyezkedhet el alatta, felfelé azonban egy szabványos interfészt biztosít, elfedve ezáltal a hardver sajátosságait. Számos driver modul alkotja, ezek közül a teljesség igénye nélkül néhány: SPI Handler Driver, FlexRay Driver, MCU Driver, CAN Driver stb.

2.3.4 Complex Drivers

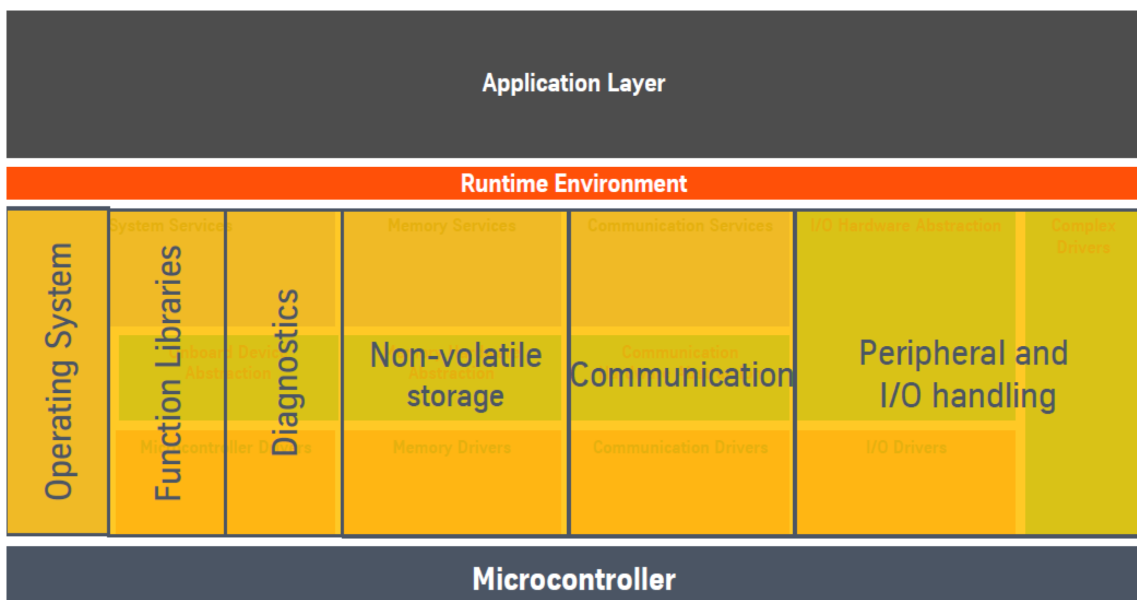
A komplex driverek lényegében olyan szoftvermodulok, amelyek nincsenek definiálva az AUTOSAR által vagy a szabvány által specifikált modul nem nyújt elégséges sávszélességet vagy funkcionalitást. Léteznek olyan speciális esetek, amikor szükséges, hogy a modul alulról közvetlenül a perifériával, felülről pedig közvetlenül az RTE-vel álljon kapcsolatban.

Egy remek példa arra, amikor szükségszerű a szabványtól eltérő funkció megvalósítása, a thyssenkruppnál fejlesztett ECU egyik kritikus szintű bemenete a kormányoszlop nyomatékszenzorjának jele, melyet speciális módon, az AUTOSAR által kínált funkciókban nem implementált eljárással szükséges mérni.

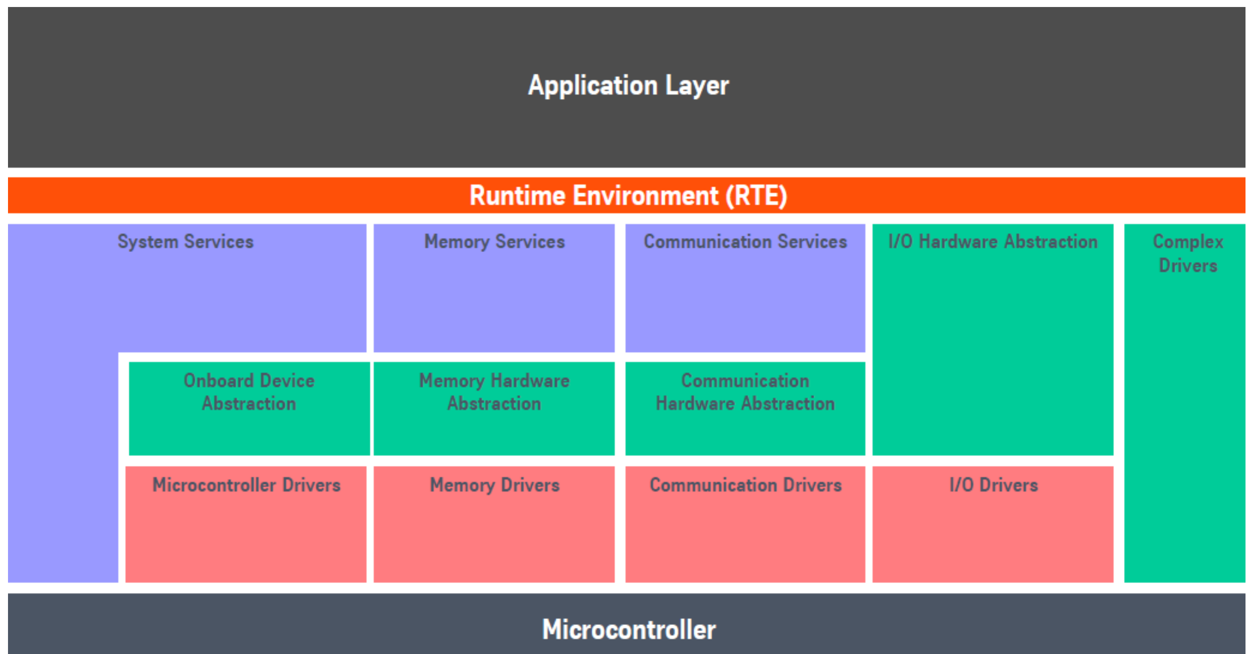
2.3.5 A BSW logikai felosztása

A 3. és 4. ábrákon látható módon a szabvány további különálló funkcionális csoportokra bontja a BSW alrétegeit, biztosítva ezáltal a rendkívül komplex követelmények modulokra bontását, illetve elősegítve az autóipari szoftverfejlesztést.

Ezek a funkcionális csoportok az alábbiak: Operációs rendszer, könyvtárak, diagnosztika, memória-szolgáltatások, kommunikáció, perifériakezelés.



3. ábra: A BSW logikai felosztása [1]



4. ábra: Funkcionális csoportok a BSW Stackben [1]

A következőkben az autóipari kommunikációval és a kommunikációs stack-kel szeretnék foglalkozni, hiszen a szakdolgozatom során kidolgozott modul többek között a kommunikációs hibák detektálásában és az ellenük történő védekezésben játszik szerepet.

2.4 Kommunikáció az autóiparban

Mint ahogy azt már a bevezetőben is említettem, a mai gépjárművekkel szemben állított követelmények (kényelmi, biztonsági és minőségi) folyamatosan növekednek, ezáltal az autókra egyre komplexebb rendszerekként szükséges tekintenünk.

Ebből fakadóan egy bonyolult, elosztott rendszerként kezelhetjük a járműveket, amelyekben minden különálló vezérlőegység kizárólag a saját feladatáért felelős. Azonban a hardver és számítási igény csökkentését elősegíthetjük, amennyiben tudatosan tervezzük meg az egyes feladatokért felelős ECU-k közti kommunikációt.

Mivel a kommunikációs linkek hálózatként működnek, ezért ugyanúgy beszélhetünk itt is hálózati topológiákról, mint a számítógép-hálózatok esetén. Beszélhetünk pont-pont, aktív és passzív csillag, busz topológiákról, valamint a hibrid kialakításokról, amelyek valamilyen formában az előbb megnevezettek ötvözései.

2.4.1 Kommunikáció a modern gépjárművekben

Ebben az alfejezetben röviden ismertetem a járművekben alkalmazott kommunikációs megoldásokat.

2.4.1.1 Szignál alapú kommunikáció

A szignál alapú kommunikáció során az elsődleges cél az ECU-k között végbemenő információcsere, amely többnyire periodikus üzenetváltásokkal valósul meg. A szignál, a kommunikáció alapegysége, azaz olyan fizikai jelentéssel bíró adatelemek összessége, amelyet az információátadás legkisebb egységeként kezelünk. Szignálnak tekinthetjük például a kerék fordulatszámát, vagy a benne mérhető légnyomást.

2.4.1.2 Diagnosztikai kommunikáció

A diagnosztikai kommunikáció nem része az általános információcserének, ugyanis mindig szervizben történik, valamilyen diagnosztikai eszköz és az ECU között. Lényegében akkor használjuk, amennyiben hibakódot szeretnék kiolvasni vagy esetleg törölni, illetve szoftverfrissítésre is használható.

2.4.1.3 Hálózatmenedzsment kommunikáció

A hálózatmenedzsment üzenetek a szignál alapú kommunikációhoz hasonlóan vezérlőegységek között zajlanak, szintén periodikus üzenetváltáson keresztül. Legfőbb célja az alhálózatok ébresztése, illetve lekapcsolása.

2.4.1.4 XCP (Universal Measurement and Calibration Protocol)

Az előbb említett kommunikációs megoldásokkal ellentétben az XCP fejlesztési időben biztosít lehetőséget az ECU memóriájának olvasására és írására. Jelen dolgozat keretein belül azonban elsősorban a futási időben való kommunikációval foglalkozom, így a továbbiakban az ehhez kapcsolódó protokollokat mutatom be.

2.5 Kommunikációs protokollok

A fejezetben már ismertetett okok miatt rendkívül fontos volt, hogy az autóipari igényeket kielégítő kommunikációs eljárások kerüljenek kidolgozásra. Mivel számos, egymástól akár teljesen különböző és független adat kerülhet a jármű belső kommunikációs buszaira, ezért jogos volt az az elvárás, hogy ne egy általános protokoll kerüljön megalkotásra, hanem specifikus, bizonyos körülmények között legjobban használható adatátviteli eljárások szülessenek.

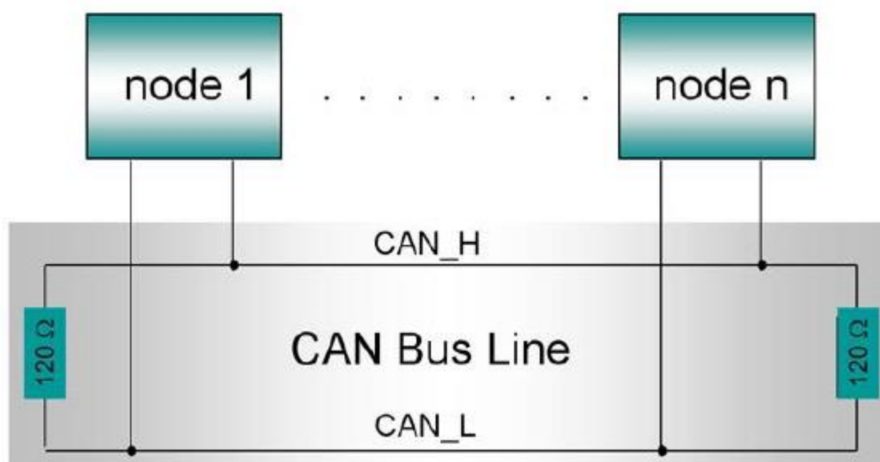
Ezen fejlődésnek első lépése volt még a 80-as években a CAN (Controller Area Network) megalkotása, mely megoldásnak célkitűzése volt, hogy egy biztonságos, egyszerű és olcsó kommunikációs rendszert biztosítson. Az elmúlt negyven évben azonban az igények folyamatosan nőttek, ezért a kezdeti adatátviteli sebességek, illetve az elvárt reakcióidők, valamint a sávszélesség-változással számos protokoll született, speciálisan autóipari felhasználási szándékkal.

Mivel szakdolgozatom a thyssenkrupp-nál végzett fejlesztéseken alapul, ahol pedig a CAN és a FlexRay protokollokkal dolgozunk elsősorban, így ezeket fogom részletesen bemutatni. Fontosnak tartom megemlíteni azonban, hogy ezek mellett számos egyéb adatátviteli megoldás is használatos, úgymint a LIN, melyet elsősorban egyszerű, véghálózati feladatok ellátására fejlesztettek ki (például ablakemelő), vagy akár az autóipari Ethernet, amely jóval komplexebb adatátvitelt és az említett protokollokhoz képest jóval nagyobb sávszélességet biztosít.

2.5.1 Controller Area Network

A Controller Area Network, vagy röviden CAN elsősorban az autóiparba szánt, azonban számos más területen is sikerrel alkalmazott soros adatkommunikációs rendszer. Fontos előnye, hogy felépítése és működése viszonylag egyszerű, valamint valós idejű, megbízható adatátvitelt tesz lehetővé.

Az 5. ábrán ábrán látható a CAN busz felépítése, megfigyelhető hogy a buszon differenciális jeltovábbítás zajlik a két (CAN_High azaz Can_H és CAN_Low, azaz CAN_L) jelvezetéken.



5. ábra: A CAN busz felépítése [3]

Továbbá elmondható, hogy a CAN egy broadcast típusú hálózatot valósít meg, tehát a buszra csatlakozó csomópontok bármelyike küldhet adatot a hálózatra vagy éppen fogadhat adatot a hálózatról. Felmerül a kérdés, hogy akkor az elkerülhetetlen adatütközések esetén melyik küldő információja kerül továbbításra.

Ennek megoldását az üzenetek prioritása rejti, amely információt az azonosító hordozza. A küldésre kész egységek csak akkor helyezhetik fel üzeneteiket a buszra, amikor az üres, tehát nem történik rajta adatátvitel. Abban az esetben, ha két (vagy több) küldő egy rövid időegységen belül mégis egyszerre helyezne fel adatot a buszra, akkor az ütközés detektálása után a kisebb prioritású azonosítóval rendelkező egység befejezi az üzenettovábbítást. Ez az úgynevezett nem destruktív arbitrálás folyamata, amely nagyban hozzájárult a CAN népszerűségéhez.

Ugyanakkor fontos kiemelni, hogy a protokoll átviteli sebessége a 125 kbit/s – 1 Mbit/s intervallummal írható le. Ez számos feladat esetén kielégítő, vannak azonban olyan esetek, amikor már másik, gyorsabb átviteli protokoll használata javasolt.

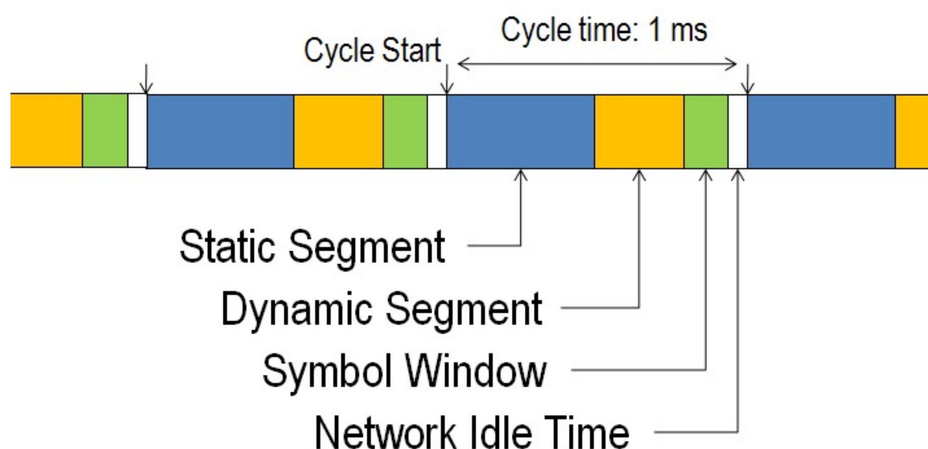
2.5.2 FlexRay

A FlexRay adatátviteli protokoll az előbbi alpontban ismertetett CAN-hez képest nem eseményvezérelt, hanem idővezérelt kommunikáción alapul. Kifejlesztésének érdeme a FlexRay Consortium-hoz köthető, viszont apró érdekességként fontosnak tartom megemlíteni, hogy ez a konzorcium mára már megszűnt, az általuk megalkotott protokollt azonban még számos autógyártó használja.

FlexRay esetén egy nagy sebességű, szinkron és aszinkron adattovábbítást is támogató kommunikációs protokollról beszélünk, mely 10 Mbit/s sávszélességet biztosít, ez számottevően nagyobb (tízszerese) a CAN által kínált maximumnak. Továbbá fontos kiemelni, hogy a redundancia érdekében két kommunikációs csatornát is biztosít, tehát egyszerre két párhuzamos csatorna is kialakítható. Továbbá arra is lehetőséget biztosít, hogy a két csatornán különböző adatokat továbbítsunk, így összesen 20 Mbit/s sávszélesség áll rendelkezésre.

Mivel a FlexRay idővezérelt kommunikáció, így minden résztvevő egység a saját időintervallumában küldhet adatot a buszra, ezáltal ütközés nem állhat elő. Ezen megoldásnak az alapja a Time Divison Multiple Access, azaz a TDMA. A CAN protokollal szemben az üzeneteket az azonosító helyett az elküldés időpontja azonosítja.

A kommunikáció alapegysége a hiperciklus, amely 64 FlexRay ciklusból áll, amelyek felépítése négyféleképpen történhet, ezek elhelyezkedését mutatja a 6. ábra. Egy ciklus tartalmaz mindig statikus szegmenst és üresjárási időt, továbbá tartalmazhat dinamikus szegmenst, illetve szimbólumablakot vagy akár mindkettőt.



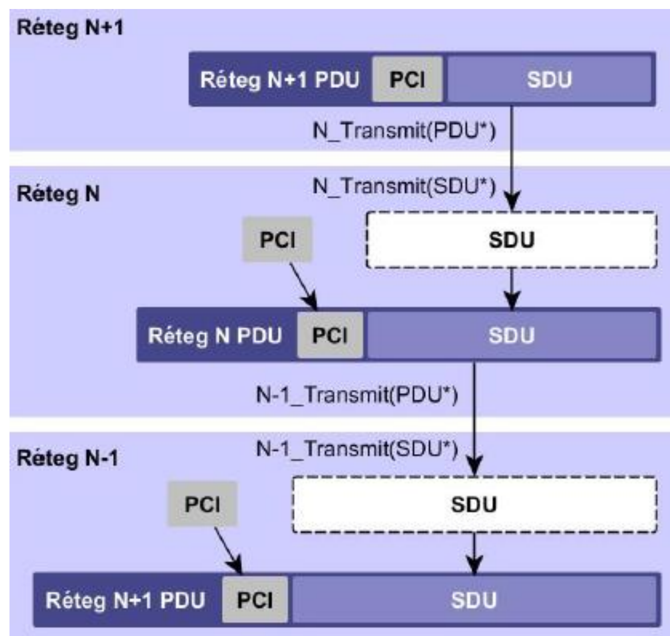
6. ábra: FlexRay ciklus felépítése [4]

A statikus szegmens adja a szigorúan idővezérelt szegmenst. Amennyiben a konkrét ECU nem rendelkezik küldendő adattal, akkor egy úgynevezett Null Frame-et helyez a buszra. A dinamikus szegmensben történhet az eseményvezérelt kommunikáció, amennyiben szükség van rá. A szimbólumablak pedig a vezérlő szimbólumok küldésére szolgál.

Zárásként elmondható, hogy a FlexRay összetettsége nem véletlen, hiszen arra lett tervezve, hogy kiszolgálja az új technológiákat és alkalmazásokat. Ebből kifolyólag a nagy sávszélességnek köszönhetően alkalmas a járművek gerinchálózatának kialakítására, valamint időosztásos kommunikációjának köszönhetően teljesíti a valós idejű követelményeket, amelyek az autóipari elosztott rendszerek esetén elvártak.

2.6 AUTOSAR, kommunikáció a rétegek között

Mint ahogy azt már bemutattam, a szoftvert az AUTOSAR rétegzett architektúrájának köszönhetően különálló rétegekre, ezen belül pedig különböző, jól definiált modulokra bontja. Ezek együttműködésének egyik legfontosabb feltétele a megfelelő kommunikáció, amelyhez a szabvány a 7. ábrán látható egységeket definiálja:



7. ábra: A rétegek közötti kommunikáció alapegységei [1]

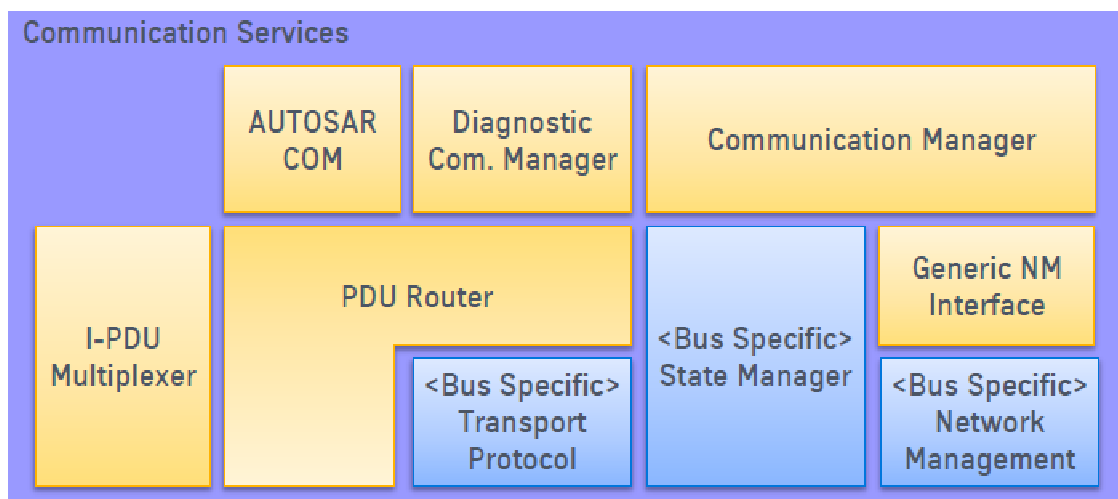
Ezek az alábbiak:

- Protocol Data Unit **(PDU):** A használt protokoll által kezelhető adategység.
- Service Data Unit **(SDU):** Maga a továbbítandó nyers adat
- Protocol Control Information **(PCI):** A használt protokoll megfelelő működéséhez szükséges kiegészítő információ.

A 7. ábra nagyon jól szemlélteti az adatok becsomagolását. Felülről nézve a „Réteg N+1" PDU-ja küldés esetén a „Réteg N"-ben már SDU-ként jelenik meg, ahol hozzáfűzésre kerül a „Réteg N" vonatkozó PCI-vel és így egy újabb PDU-t alkotva fog megérkezni a „Réteg N-1"-be. Az információ fogadása esetén a folyamat fordítva zajlik le. A lentebbi rétegtől kapott adatcsomagból (PDU) minden réteg a PCI alapján kiolvassa az SDU-t (tehát a nyers adatot), majd továbbítja a felette lévő rétegnek, amely a saját PDU-jaként kezeli az előbb leírt módon.

2.7 Communication Stack

A kommunikációs szolgáltatások legfontosabb feladata, hogy az RTE-n keresztül egységes interfészt biztosítsanak az alkalmazásnak, úgy, hogy teljes mértékben elfedi az információcseréhez használt protokollt és annak tulajdonságait.



8. ábra: Communication Stack szolgáltatás rétege [1]

A 8. ábrán látható, hogy mely modulok alkotják a kommunikációs stack szolgáltatás rétegét, a következőkben röviden ismertetem őket:

- *COM:* Elsődleges feladata az alkalmazás felől beérkező szignálok PDU-kba történő helyezése, illetve fogadás esetén a szignálok kicsomagolása az érkező PDU-kból. Fontos kiemelni, hogy protokollfüggetlen modul.
- *Diagnostic Communication Manager:* A modul feladata a diagnosztikai funkciók ellátása, úgymint: diagnosztikai sessionök, jogosultságok és állapotok, valamint a diagnosztikai kommunikáció kezelése.
- *Communication Manager:* A Communication Manager feladata a hálózati kommunikációs erőforrások menedzselése, protokollfüggetlen modul.
- *PDU Router:* A PDU Router feladata, hogy útvonalat választ a kommunikációs stackben közlekedő PDU-knak.
- *<Bus Specific> Transport Protocol:* Olyan BSW modulok, melyek feladata a szállítási réteg protokollok megvalósítása. Itt már protokollfüggő modulokról beszélünk.
- *<Bus Specific> State Manager:* Protokollfüggő modulok, amelyek célja a hálózat állapotának menedzselése.
- *Generic NM Interface:* Feladata egységes interfész nyújtása a felette lévő modulok számára a hálózati menedzsment funkciókhoz, protokollfüggetlen modul.
- *<Bus Specific> Network Management:* Protokollfüggő modulok, feladatuk a protokoll működéséhez szükséges hálózati funkciók biztosítása.

3 End-To-End Communication Protection Library

Az alábbi fejezetben elsőként röviden néhány szót ejtek a szabvány által kínált biztonsági megoldásokról. Még mielőtt azonban rátérnék a szakdolgozatom alapjául szolgáló E2E modulra, fontosnak tartom bemutatni az előző időszak projekttantárgyai során megismert modulok fontosságát.

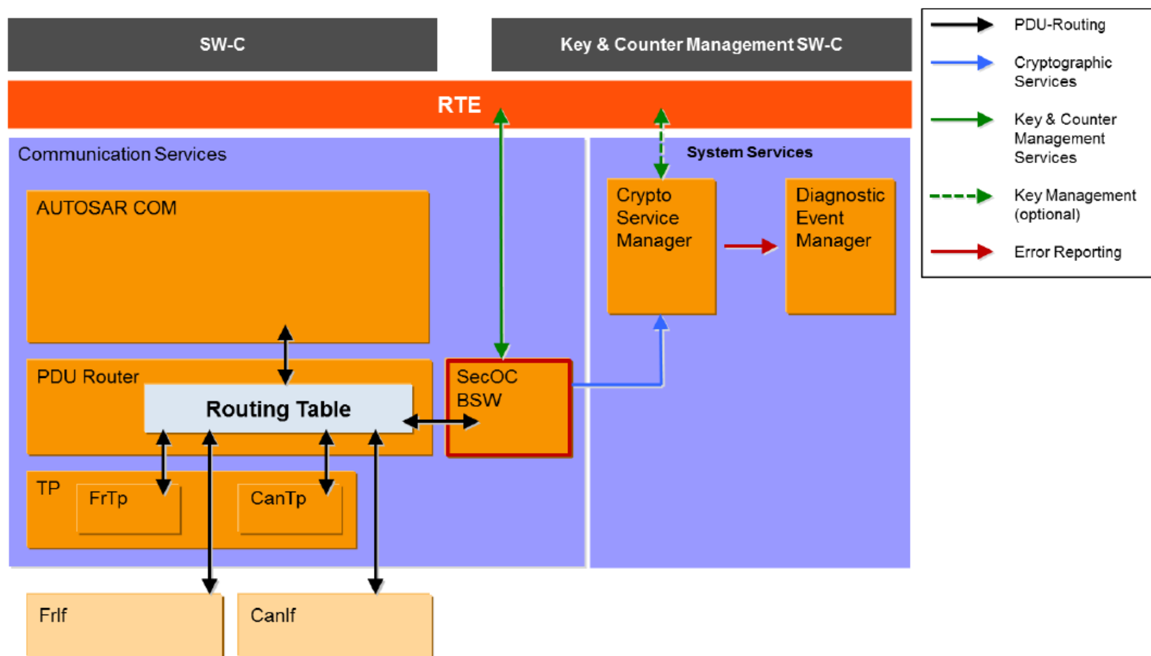
Ezután ismertetem az E2E Library elhelyezkedését az AUTOSAR-ban, bemutatom a kapcsolatát a környező entitásokkal. Majd rátérek a modul bemutatására, funkcióinak ismertetésére.

3.1 Biztonsági funkciók autóipari szoftverben

A mai modern gépjárművek esetén egyre nagyobb szerepet kapnak a vezetéstámogató és részleges önvezető funkciók, ebből kifolyólag a járműben található vezérlőegységeknek bizonyos esetekben kommunikálniuk kell a külső infrastruktúrával, ezáltal egyre kevésbé tekinthetünk a belső kommunikációs hálózatokra zárt rendszerként.

Ennek egyik nagy veszélye, hogy az eddig külső hatásoktól teljesen izolált kommunikációs buszok nem megfelelő és körültekintő tervezés esetén befolyásolhatók az informatika területéről már jól ismert különböző támadásokkal, amelyeknek - autóiparról lévén szó - akár életveszélyes következményei is lehetnek.

Ezért szükséges volt különböző kriptográfiai funkciókat bevezetni a fejlesztésbe, a szabvány erre a 9. ábrán látható megoldást kínálja:



9. ábra: SecOC elhelyezkedése [5]

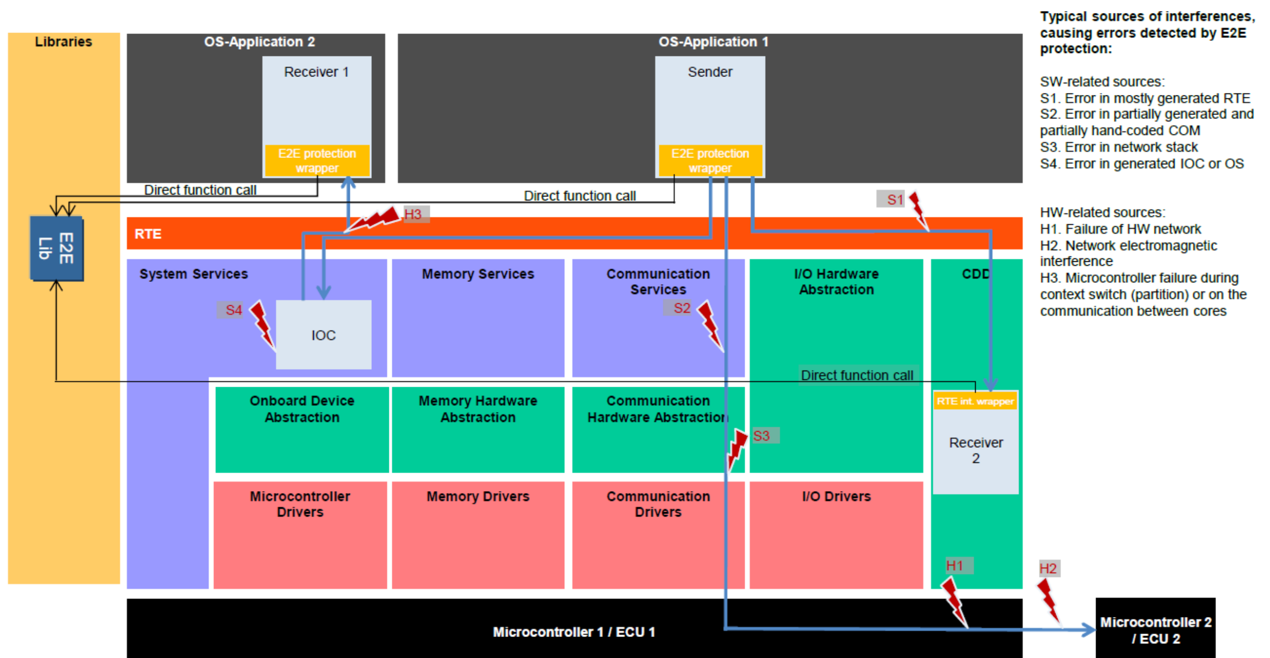
A System Services-ben (Rendszer Szolgáltatások) elhelyezkedő új modul, az úgynevezett Crypto Service Manager (CSM) a kriptográfiai funkciók biztosításáért felel (például digitális aláírás, hitelesség ellenőrzése stb.). A kommunikációs stack és a kriptográfiai szolgáltatások között a kapcsolatot pedig a Secure Onboard Communication modul, vagy röviden SecOC teremti meg.

Az ábrán látható, hogy a PDU Router-rel egy szinten helyezkedik el, vele áll kapcsolatban. Röviden annyi a SecOC működésének lényege, hogy amennyiben egy üzeneten kriptográfiai műveletet szükséges elvégezni, akkor a SecOC MAC generálás/ellenőrzés és aláírás generálás/ellenőrzés szolgáltatást kérhet a CSM-től. Tehát ha az üzenetet a PDU Router továbbítja a SecOC modulnak, amely a kriptográfiai műveleteket kívánó adategységet elküldi a CSM-nek. Amennyiben sikeres volt az ellenőrzés, akkor a PDU-t visszaküldi PDU Routernek.

Ugyanakkor fontosnak tartom kiemelni, hogy a SecOC (a CSM által) a BSW stack-ben kínál védelmi és hitelesítő funkciókat. Fontos azonban a kommunikációs buszokon közlekedő üzenetek további védelme különféle meghibásodások ellen, ezért alkották meg a az End-To-End Protection Library-t. Tehát a két megoldás teljes mértékben független egymástól, azonban mégis kiegészítik egymást.

3.2 Az E2E modul elhelyezkedése a rétegzett architektúrában

Mint ahogy a modul megnevezése is mutatja (End-To-End Communication Protection Library, magyarra fordítva: Végpontok közötti kommunikáció védelmét biztosító könyvtár), a kommunikáció védelmét hivatott szolgálni a végpontok között. Ebből kifolyólag az eddig ismertetett modulokkal ellentétben nem a BSW-n belül helyezkedik el, hanem a többi Library-hez hasonlóan külön egységet képez a rétegzett architektúrán belül, mint ahogy az a 10. ábrán látható is:



10. ábra: E2E Library elhelyezkedése az architektúrában [6]

Az AUTOSAR könyvtárak gyakorlatilag speciális célú függvények gyűjteményei. Meghívhatók a BSW modulok által, szoftverkomponensek által, valamint Library-k által is. Viszont a Library modulok csak egyéb Library-kat hívhatnak meg.

Itt fontosnak tartom kiemelni, hogy a többi BSW modullal ellentétben, amelyek számos másik modul függvényeit hívhatják meg a működésükhöz, az End-To-End Protection modul csak a CRC Library szolgáltatásait használja. CRC függvényekre az elvégzendő cheksum kalkulációk miatt van szükség, amely a modul által biztosított védelmi mechanizmusok egyik alapja.

Továbbá fontos megjegyezni, hogy a Library-k mindig a hívó kontextusában futnak le, feladatuk elvégzése után mindig visszatérési értékkel (úgynevezett StatusType-pal) jelzik a művelet eredményét.

Másik fontos különbség a többi BSW modulhoz képest, hogy nincs szükség inicializációra a Library meghívása esetén és nincsenek belső állapotaik. Szinkron működésűek, ebből kifolyólag nem használnak várakozási pontokat.

Az előbb ismertetett tulajdonságaik alapján viszonylag egyszerűbb, „lightweight" modulokként tekinthetünk a könyvtárakra, melyből összesen nyolcat definiál az AUTOSAR.

3.3 E2E Communication Protection Library működése

Az E2E Protection azon a koncepción alapszik, hogy a biztonságkritikus adatok továbbítására szolgáló kommunikációt futási időben szükséges védeni az esetlegesen fellépő hibáktól. Ezek lehetnek véletlenszerű hardvermeghibásodások (például a CAN vevőegység regisztere meghibásodik), EMC miatt fellépő zavarok vagy az úgynevezett Virtual Functional Bus (VFB) implementációjában fellépő hibák.

Ezeket a védelmi mechanizmusokat többek között checksum kalkulációk felhasználásával teszi lehetővé a modul. A gyakorlatban ez annyit jelent, hogy az E2E Library plusz információt csatol a buszon közlekedő üzenetekhez, melyet a küldendő adaton végzett matematikai műveletekkel számol ki.

3.3.1 E2E Library modul felhasználása

A modul által kínált védelmi mechanizmusok helyes felhasználásért a hívó felek felelősek. Természetesen a modul felkészült az esetleges rossz paraméterezés kezelésére is (például NULL pointer átadása.

Az E2E Library a következő lehetőséget kínálja:

- RTE-n keresztül továbbítandó biztonságkritikus adatok védelme hozzáadott ellenőrző adatokkal
- RTE-n keresztül érkező biztonságkritikus adatelemek ellenőrzése (szintén a hozzácsatolt ellenőrző adatokon keresztül)

- Amennyiben hibás adategységek érkeztek, úgy azt jelzi a fogadó szoftverkomponensnek (SW-C), amelyek kezelése ezután már a komponens feladata

3.3.2 A modullal szemben támasztott követelmények

Mivel a járműiparban biztonságkritikus rendszereket fejlesztünk, ezért elég szigorú követelményeknek szükséges a szoftvernek megfelelnie. Ezen követelményeket szabványosították, így született meg az ASIL (Automotive Safety Integrity Level), ezt mutatom be röviden a következőkben.

3.3.2.1 Automotive Safety Integrity Level

Az ASIL egy kockázatosztályozó rendszer, amely az ISO 26262 szabvány része, célja a jármű biztonsági szabványba foglalása. Gyakorlatilag különböző biztonsági követelményeket támaszt az adott hiba bekövetkezésének valószínűsége és az okozható kár mértékének alapján.

A szabvány négy szintet különböztet meg, ASIL A, B, C, illetve D. Az A szint jelenti a legkisebb veszélyt, a D pedig legnagyobbat. Például a féklámpa meghibásodása által okozott veszély ASIL B kategóriába tartozik, a kormányszervo helytelen működése esetén kialakult vészhelyzet viszont már az ASIL D szintbe esik.

Ezáltal az E2E Library-nak képesnek kell lennie ASIL D által támasztott követelményeket is kielégíteni. Fontos azonban kiemelni, hogy az E2E modul használata önmagában nem megfelelő az ASIL D követelmények teljesítésére. Erről a hívó félnek kell meggyőződnie, hogy a kiválasztott E2E profil (az elérhető nyolc közül) megfelelő hibadetektáló képességeket biztosítson a használandó kommunikációs hálózathoz.

3.4 E2E Library implementáció

A fejezet ezen részében bemutatom az End-To-End Communication Protection Library modul típusdefinícióit és a külvilág felé kínált API-kat. Egy modul fejlesztése során az adattípusokat a szabvány biztosítja, így jelen esetben nem ezek implementálása a kihívás. A mérnöki feladat a tényleges működés megvalósítása a követelmények alapján, a típusdefiníciókat és az API deklarációkat felhasználva. A modul által a külvilág felé biztosított interfészen kívül a fejlesztőmérnök döntése, hogy milyen függvényekkel és hogyan valósítja meg az elvárt működést.

3.4.1 E2E Library típusok

3.4.1.1 Importált típusok

Általánosan a legtöbb modulról elmondható, hogy több importált típussal is rendelkezik. Ennek magyarázata az AUTOSAR rétegzett architektúrájában keresendő, melynek egyik fő célja a megvalósítandó funkciók részegységekre tagolása. A legtöbb BSW modul esetén az importált típusok száma igen magas is lehet, az E2E Library esetén azonban mindösszesen a StandardTypes-ra van szükségünk, ezen belül is csak az Std_ReturnType illetve az Std_VersionInfoType kerül felhasználásra.. A szabványban található pontos követelmény erre vonatkozóan a 11. ábrán látható.

[SWS_E2E_00017] ⌈

Module	*Header File*	*Imported Type*
Std_Types	StandardTypes.h	Std_ReturnType
	StandardTypes.h	Std_VersionInfoType

⌋ (SRS_E2E_08528)

11.ábra: Importált típusok [6]

Az Std_VersionInfoType verziószámokat tartalmaz, míg az Std_ReturnType visszatérési értéket ír le, melyek leggyakrabban használt értéke az E_OK és E_NOT_OK.

3.4.1.2 E2E Library típusdefiníciók

Az E2E Library mind a 8 profiljának saját típusokat definiál, profilonként ötöt, ezek profilról profilra minimálisan változnak. Ebben az alpontban az egyes profil típusain keresztül fogom bemutatni az E2E modul adatszerkezetét, mivel a többi profil alapvetőleg néhány paraméterben tér el. Ezekre a különbségekre még a későbbiekben kitérek.

A 12. ábrát példaként említi a szabvány leírása a P01-es profil adattömbjének kialakítására. Mint ahogy a P01ConfigType struktúránál is látható, az offset-et mindig bitben értelmezzük. Továbbá több különböző offset értéket is definiál a szabvány, ezek azért fontosak, mert az adattömb kezdőcíméhez viszonyítva szükséges meghatároznunk az adott változó pozícióját.

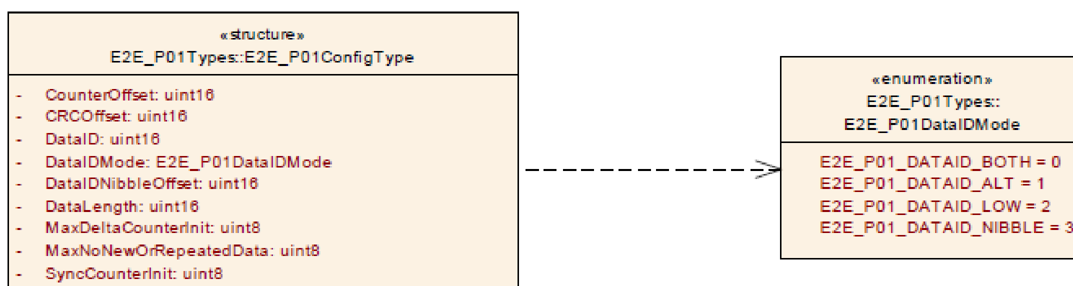
	MSB							LSB
Data[0]	7	6	5	4	3	2	1	0
	CRC with bit offset 0							
Data[1]	15	14	13	12	11	10	9	8
	User data with bit offset 12				Counter with offset 8			
Data[2]	23	22	21	20	19	18	17	16
	User data with bit offset 20				User data with bit offset 16			

12. ábra: P01 adattömb felépítése [6]

E2E_P01ConfigType:

A struktúra az alábbi mezőket definiálja:

- *CounterOffset*: A számláló offsetje, négy többszöröse kell hogy legyen.
- *CRCOffset:* A CRC kalkuláció eredményének offsetje, értéke nyolc többszöröse kell hogy legyen.
- *DataID:* Egyedi azonosító, amellyel minden adattömböt ellát az egyes profil.
- *DataIDNibbleOffset:* A felső bájt alsó négy bitjének offsetje.
- *DataIDMode:* A DataID-n történő CRC számítás módját határozza meg.
- *DataLength:* Az adat hossza, bitben megadva. Nyolc többszöröse kell hogy legyen, és értéke kisebb egyenlő mint 240.
- *MaxDeltaCounterInit:* A maximális megengedett különbség két egymás után érkezett helyes adat számlálója között.
- *MaxNoNewOrRepeatedData:* A hiányzó vagy ismételt adat maximális mennyisége, amennyit normális kommunikációs körülmények között tolerál a fogadó fél.
- *SyncCounterInit:* A számláló konzisztenciájának ellenőrzéséhez szükséges adatmennyiség.



13. ábra: ConfigType struktúra és DataIDMode enumeráricó kapcsolata [6]

E2E_P01DataIDMode:

A DataIDMode azt határozza meg, hogy a CRC számításhoz a modul a DataID mely bájtjait használja fel, az enumeráció az alábbi értékeket definiálja:

- *E2E_P01_DATAID_BOTH = 1:* Ez esetben a CRC kalkuláció a DataID mindkét bájtján kerül végrehajtásra.
- *E2E_P01_DATAID_ALT = 2:* A CRC kalkulációban a DataID 16 bitjéből 8 vesz részt, a counter paritása szerint váltakoznak a felső és alsó bitek.
- *E2E_P01_DATAID_LOW = 3:* Csak az alsó bájt kerül felhasználásra a CRC kalkuláció során.
- *E2E_P01_DATAID_NIBBLE = 4:* Ebben az esetben szintén az alsó bájt vesz részt a CRC kalkulációban, a felső bájt alsó négy bitje azonban továbbításra kerül az adattal.

A 13. ábra a ConfigType struktúra és a DataIDMode enumeráció kapcsolatát mutatja be, mivel a ConfigType struktúrának része egy DataIDMode típusú változó is.

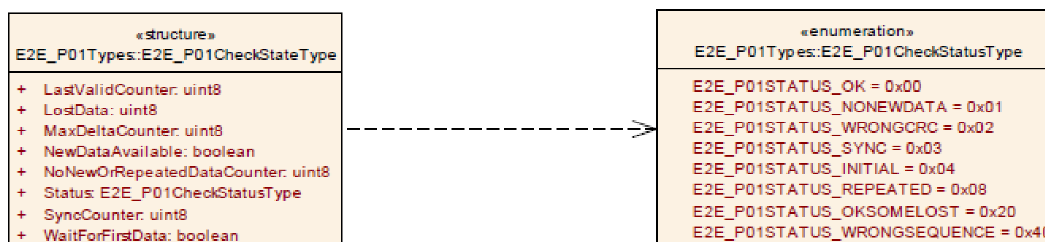
E2E_P01ProtectStateType:

Ez a struktúra tartalmazza az uint8 típusú *Counter*-t, amely egy védelmi mechanizmust kínál a checksum számítás mellett a modulban.

E2E_P01CheckStatusType

Az adatverifikáció eredményét tartalmazza, amelyet a profil Check függvénye határozott meg. Az alábbi értékeket definiálja:

- *E2E_P01STATUS_OK = 0x00:* Az új adatot fogadta a modul, a CRC értéke helyes, a számláló értéke inkrementálásra került.
- *E2E_P01STATUS_NONEWDATA = 0x01:* Meghívásra került a Check függvény, azonban nem érkezett új adat.
- *E2E_P01STATUS_WRONGCRC = 0x02:* Az adatot fogadta a modul, a CRC értéke azonban helytelen.
- *E2E_P01STATUS_SYNC = 0x03:* Az adatot azután fogadta a modul, hogy a számlálónak egy nem várt értékét kapta.
- *E2E_P01STATUS_INITIAL = 0x04:* Az új adatot fogadta a modul, a CRC értéke helyes, azonban a fogadó inicializációja óta ez az első adattömb, így a számláló értéke még nem ellenőrizhető.
- *E2E_P01STATUS_REPEATED = 0x08:* A modul fogadta az adatot, a CRC értéke helyes, azonban a számláló értéke megegyezik a korábban fogadottal.
- *E2E_P01STATUS_OKSOMELOST = 0x20:* Azt jelenti, hogy valószínűleg néhány adattömb elveszett az utolsó vétel óta, de ez még a toleranciahatáron belül van.
- *E2E_P01STATUS_WRONGSEQUENCE = 0x40:* A legutolsó vétel óta túl sok adat veszett el.



14. ábra: CheckStateType struktúra kapcsolata a CheckStatusType enumerációval [6]

E2E_P01CheckStateType:

Az alábbi adattípus a fogadó fél állapotát írja le, összesen nyolc változót tartalmaz, melyek az alábbiak:

- *LastValidCounter:* Az utoljára fogadott számláló értéke.
- *MaxDeltaCounter:* Meghatározza a maximális különbséget két egymás után kapott érvényes üzenet számlálója között.
- *WaitForFirstData:* Amennyiben igaz, akkor azt jelenti, hogy még nem fogadott a modul érvényes adatot.
- *NewDataAvailable:* A modul számára új adat elérhető, amin ellenőrzést szükséges végrehajtani.
- *LostData:* Az utolsó érvényes adat óta elvesztetett üzenetek száma.
- *Status:* Az ellenőrzés eredménye, amit a Check függvény határozott meg.
- *SyncCounter:* Amennyiben az utoljára fogadott számláló nem volt megfelelő, a SyncCounter értéke azt határozza meg, hogy mennyi adat fogadása után tekinthető helyesnek a fogadott számláló.
- *NoNewOrRepeatedDataCounter:* Az értéke azt határozza meg, mennyi vételi ciklusban fordulhat elő, hogy nem érkezett új adat, vagy ugyan az az adat érkezett ismételten.

Az előző oldalon található 14. ábra mutatja a CheckStatusType enumeráció és a CheckStateType struktúra kapcsolatát, ugyanis a CheckStateType-nak fontos eleme a CheckStatusType típusú Status változó.

Összefoglalva elmondható, hogy sok másik modullal szemben az E2E Library-nak nincsenek belső állapotai. Az egyes profilok funkciói az ismertetett struktúrákkal és enumerációkkal megvalósíthatók.

4 Megvalósítás

Ezen fejezetben ismertetem magának a megvalósításnak a lépéseit. A követelményelemzéssel, a funkciók megismerésén és implementálásán keresztül a teszteléssel bezárólag bemutatom a szakdolgozatom, legfontosabb részét, valamint vázolom az implementációt és kihívásait, illetve ezek megoldásait.

4.1 Követelménylista

Mint ahogy azt már jelen dokumentumban említettem, egy AUTOSAR modul fejlesztése mindig a megfelelő szabványverzió követelményeinek kigyűjtésével kezdődik. Ezután az egyes követelményeket megvalósíthatóság szempontjából vizsgáljuk, mindezek tárolására célszerű XML fájlt használni. Ennek oka, hogy emberi szem által is könnyen olvasható, ugyanakkor az egyes tool-okkal is viszonylag könnyen kezelhető.

A fejlesztés során figyelni kell a cégspecifikus követelményekre is, annak érdekében, hogy a céges környezetbe integrálható legyen az adott modul. Továbbá a statikus forrás előállítása során az egyes követelményeket megvalósító részeket mindig szükséges az adott követelményhez rendelni, mivel ellenőrizhető az implementáció követeleményfedettsége automatizáltan is. Ezenfelül amennyiben az adott modul fejlesztése más mérnökhöz kerül, vagy a modul újabb verziójának fejlesztése esetén az előbbi verziót vennénk alapul, akkor könnyebben megérhető a megvalósítás.

4.2 Statikus forráskód

4.2.1 Eclipse fejlesztőkörnyezet

A modul implementálása során az Eclipse fejlesztőkörnyezetet használtam. Ennek oka, hogy maga az IDE letisztult, jól használható fejlesztéshez. Továbbá automatikus kiegészítést biztosít és rendkívül könnyen lehet a „Debug” módját használni.

4.2.2 Adattípusok

A szakdolgozatom előző fejezetének utolsó pontjában már részletesen bemutattam őket, a megvalósítási folyamatban is fontosnak tartom azonban megemlíteni. Tehát miután megtörtént a követelményelemzés, a funkcionalitás implementálása előtt megismerjük a modul adatstruktúráját, majd ezt átvesszük a szabványból a megfelelő header fájlokba.

Az E2E Library esetén a statikus forráskód biztosítja a modul funkcionalitását. Ezen alpontban szeretném ismertetni a szabvány által definiált API-kat, illetve részletesen bemutatom a két nagyobb függvényt.

4.2.3 E2E Library 4.4 API-k

A szabvány mind a nyolc profilra öt API-t definiál, melyek profilonként a nevükben természetesen különböznek. A Protect és Check mechanizmusok profilról profilra változnak, az egyes profil rendelkezik azonban a legkomplexebb logikával, ezért ezt szeretném részletesen bemutatni. A profilok közötti különbségeket a 4.2.4-es alpontban ismertetem.

4.2.3.1 E2E_P01ProtectInit

A ProtectInit inicializálja a számlálót tartalmazó struktúrát, természetesen a StatePtr-t először vizsgálnunk kell, hogy NULL pointeren ne végezzünk műveletet. Visszatérési értéke Std_ReturnType típusú, jelen esetben E2E_E_OK, ha sikeresen megtörtént az inicializálás. Amennyiben NULL pointerrel került meghívásra a függvény, akkor E2E_E_INPUTERR_NULL-lal tér vissza. Deklarációja az alábbi:

```
Std_ReturnType E2E_P01ProtectInit(E2E_P01ProtectStateType *StatePtr);
```

4.2.3.2 E2E_P01CheckInit

A CheckInit inicializálja az ellenőrzéshez szükséges változókat tartalmazó struktúrát. Itt is fontos a NULL pointer ellenőrzés, a visszatérési értékek megegyeznek a ProtectInit-nél leírtakkal, deklarációja az alábbi:

```
Std_ReturnType E2E_P01CheckInit(E2E_P01CheckStateType *StatePtr);
```

4.2.3.3 E2E_P01MapStatusToSM

A MapStatusToSM API hozzárendeli a Check függvény eredményét (státuszát) egy általános check-státuszhoz. A deklarációja:

```
E2E_PCheckStatusType E2E_P01MapStatusToSM(Std_ReturnType CheckReturn,
                                          E2E_P01CheckStatusType Status,
                                          boolean profileBehavior);
```

Bemenetei az alábbiak:

- *Std_ReturnType CheckReturn:* A Check API visszatérési értéke
- *E2E_P01CheckStatusType Status:* A Check API státusza
- *boolean profileBehavior:* Korábbi verziókkal való kompatibilitás miatt szükséges boolean változó.

Visszatérési értéke pedig egy immáron profiltól független státusz lesz.

4.2.3.4 E2E_P01Protect

Ez a függvény felelős küldő oldalon az adat védelméért. Deklarációja az alábbi:

```
Std_ReturnType E2E_P01Protect(const E2E_P01ConfigType *ConfigPtr,
                              E2E_P01ProtectStateType *StatePtr,
                              uint8 *DataPtr);
```

Látható, hogy a visszatérési értéke Std_ReturnType típusú, mely az alábbi értékeket veheti fel:

- *E2E_E_INPUTERR_NULL*: Ez akkor fordulhat elő, ha a bemeneti paraméterek közül valamelyik NULL pointer, ezek vizsgálata fontos minden esetben.
- *E2E_E_INPUTERR_WRONG*: Ezzel az értékkel out-of-range hibák esetén térünk vissza. Például ha a ConfigPtr DataLength eleme nagyobb értéket tartalmaz, mint a profil megengedett maximális adathossza (profil egy esetén ez 240 bit).
- *E2E_E_INTERR*: A belső műveletek során hiba lépett fel.
- *E2E_E_OK*: Minden rendben, sikeresen lefutott a CRC művelet, értékét beleírtuk az adattömb megfelelő indexű elemébe, továbbá lekezeltük a számláló értékét is, amit szintén beleírtunk az adattömbbe.

Bemenetei az alábbiak:

- *E2E_P01ConfigType* ConfigPtr:* Statikus konfigurációt tartalmazó struktúrára mutató pointer.
- *E2E_P01ProtectStateType* StatePtr:* A számlálót tartalmazó struktúrára mutató pointer.
- *uint8* DataPtr:* A küldendő adattömb kezdőcímére mutató pointer.

A függvény megvalósítása során első lépésként meghívtam a fentebb ismertetett ProtectInit API-t, ami végrehajtja a NULL pointer ellenőrzést a StatePtr-en, illetve beállítja a számláló értékét nullára (tehát inicializálja a struktúrát).

Mivel autóiparban biztonságkritikus rendszereket fejlesztünk, valamint törekednünk kell a megoldások optimalizálására, ezért úgy kell kialakítunk a függvények belső logikáját, hogy mindösszesen egy return utasítás legyen benne.

Ebből kifolyólag a ProtectInit visszatérését változóban letárolva feltételvizsgálatot szükséges végezni, hogy csak akkor végezzünk további műveleteket, amennyiben a ProtectInit E2E_E_OK-val tért vissza.

Fontos elvégezni a NULL pointer ellenőrzést a DataPtr-en is, illetve ezután a ConfigPtr-en is. Ehhez azonban egy külön segédfüggvényt írtam, ugyanis itt az out-of-range hibákra is figyelni kell már. A segédfüggvényből kiragadott részek, amely az előbb említett vizsgálatot végzik el a P01 profil esetén:

```
/* Check DataLength requirements */
        if ((ConfigPtr->DataLength > E2E_MAX_P01_DATA_LENGTH) ||
  (ConfigPtr->DataLength & 7U != 0U)) {
            ret = E2E_E_INPUTERR_WRONG;

        }

/* Check CounterOffset and CRCOffset requirements */
        if ((ConfigPtr->CounterOffset & 3U != 0U) || (ConfigPtr-
  >CRCOffset & 7U != 0U)) {
            ret = E2E_E_INPUTERR_WRONG;

        }

/* Check that DataLength is capable of storing CRC+Counter+Data */
        if ((ConfigPtr->CRCOffset + 8U > ConfigPtr->DataLength) ||
  (ConfigPtr->CounterOffset + 4U > ConfigPtr->DataLength)) {
            ret = E2E_E_INPUTERR_WRONG;
        }
```

Itt fontosnak tartom kiemelni az optimalizációt. Ugyanis például az első feltételnél szükséges vizsgálni azt, hogy a DataLength nyolc többszöröse-e. Az ehhez szükséges modulo 8 művelet a háttérben bitenkénti ÉS (jelen esetben: & 7) műveleteket jelent, ezért célszerű már a forráskódban megtenni ezt az egyszerűsítést.

Ha itt is E2E_E_OK a visszatérési érték, akkor hozzáláthatunk a tényleges védelmi mechanizmusok megvalósításához.

Először beleírjuk a számlálót az adattömb második elemébe. Amennyiben a ConfigPtr DataIDMode eleme egyenlő E2E_P01_DATAID_NIBBLE-vel, akkor az adattömb ConfigPtr DataIDNibbleOffset által mutatott elemébe kell írnunk. Itt azt vizsgáljuk, hogy a DataID bitjeit hogyan írjuk bele az adattömbbe, hiszen a DataID-n is végzünk CRC kalkulációt (ammenyiben E2E_P01_DATAID_NIBBLE van megadva). Az imént leírtakért felelős kódrészlet:

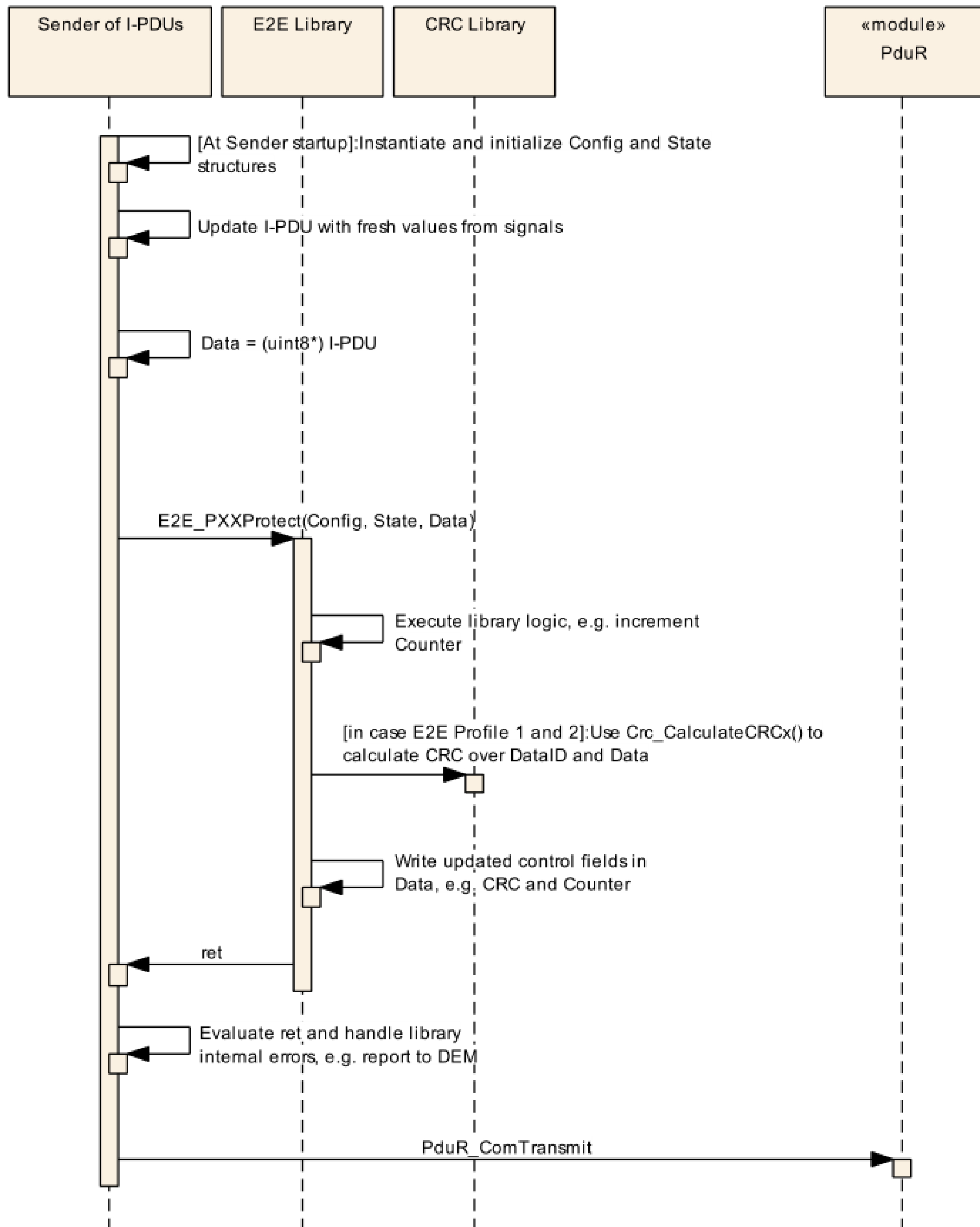
```
/* Write DataID nibble in Data,
 * if E2E_P01_DATAID_NIBBLE configuration is used
 */
  if(ConfigPtr->DataIDMode == E2E_P01_DATAID_NIBBLE){
        if(ConfigPtr->DataIDNibbleOffset & 7U == 0U){
              *(DataPtr+(ConfigPtr->DataIDNibbleOffset/8U)) =
                    (*(DataPtr+(ConfigPtr->DataIDNibbleOffset/8)) &0xF0U) |
                    ((ConfigPtr->DataID >> 8U) & 0x0FU);
        }
        else{
              *(DataPtr+(ConfigPtr->DataIDNibbleOffset/8U)) =
              (*(DataPtr+(ConfigPtr->DataIDNibbleOffset/8U)) & 0x0FU) |
              ((ConfigPtr->DataID >> 4U) & 0xF0U);
        }
  }
```

Ezután következik a checksum kalkuláció, amelyhez egy segédfüggvényt írtam, amely meghívja a CRC Library megfelelő metódusát, majd az eredmény a DataPtr által mutatott cím plusz a CRCOffset indexű tömbmezőbe kerül, P01 esetén a DataPtr[0]-ba.

```
/* CRC calculation, DataPtr[0], in case of Profile 1 */
  *(DataPtr+(ConfigPtr->CRCOffset/8U)) = E2E_P01_calculateCrc(ConfigPtr
                                          StatePtr->Counter, DataPtr);
```

Az E2E_P01_calculateCrc függvény először meghív még egy rutint, amely a DataID-n végez checksum kalkulációt, a DataIDMode függvényében. Ezután pedig magán az adaton is elvégzi a CRC számításokat, majd az így megkapott uint8 típusú eredménnyel visszatér.

A 15. ábra szemlélteti az E2E_PXXProtect() API meghívását I-PDU szinten. A folyamatábra rendkívül jól szemlélteti a küldés folyamatát, látható ahogy a küldő fél előállítja az adatot, majd meghívja a megfelelő pointerekkel a E2E_PXXProtect-et és ennek visszatérési értékének függvényében vagy jelenti az esetleges hibát, vagy továbbítja a modulnak az adatot.



15. ábra: E2E Library szerepe küldés esetén [6]

4.2.3.5 E2E_P01Check

Ez az API felelős a fogadott adat ellenőrzéséért. Deklarációja az alábbi:

```
Std_ReturnType E2E_P01Check(const E2E_P01ConfigType *Config,
                            E2E_P01CheckStateType *State,
                            const uint8 *Data);
```

Visszatérési értéke, ugyanúgy, mint a Protect esetén, Std_ReturnType típusú, mely ugyanúgy a fent már részletezett értékeket veheti fel, melyek az alábbiak:

- *E2E_E_INPUTERR_NULL*
- *E2E_E_INPUTERR_WRONG*
- *E2E_E_INTERR*
- *E2E_E_OK*

Bemenetei pedig az alábbiak:

- *E2E_P01ConfigType* Config:* Statikus konfigurációt tartalmazó struktúrára mutató pointer.
- *E2E_P01ProtectStateType* State:* Az ellenőrzéshez szükséges struktúrára mutató pointer.
- *uint8* Data:* A fogadott adattömb kezdőcímére mutató pointer.

Itt a Protect-hez hasonlóan első lépésként az API Init függvényét, tehát a CheckInit-et hívtam meg, amely szintént elvégez egy NULL pointer-ellenőrzést a State pointeren, majd inicializálja a struktúrát.

Amennyiben a visszatérési értéke megfelelő, illetve az adattömbre mutató pointer sem NULL pointer, akkor meghívhatjuk a CheckConfiguration segédfüggvényt a Config pointeren, amely a fentebb már ismertetett módon vizsgálja a NULL pointert és az esetleges out-of-range hibalehetőségeket.

Ha itt is helyes a visszatérési érték (E2E_E_OK), akkor ellenőrizzük, hogy van-e új adat. Ha nincs, akkor a State pointer státuszát átállítjuk E2E_P01STATUS_NONEWDATA-ra és a következő feltételvizsgálatnál ugyanúgy vizsgáljuk az új adat elérhetőségét, annak érdekében, hogy ne végezzünk műveleteket üres adattömbön feleslegesen.

Ezután a fogadott adattömbből megfelelő indexű elemeiből kiolvassuk a számlálót és a kapott CRC értékét, majd az adattömbön mi is elvégezzük a CRC kalkulációt:

```
/* get the receivedCounter */
   if (Config->CounterOffset & 7U == 0U){
           receivedCounter = *(Data+(Config->CounterOffset/8U)) & 0x0FU;
   } else{
        receivedCounter = (*(Data+(Config->CounterOffset/8U)) >> 4U) & 0x0FU;
   }


/*get receivedCrc and calculate CRC on Data */
   receivedCrc = *(Data + (Config->CRCOffset / 8U));
   calculatedCrc = E2E_P01_calculateCrc(Config, receivedCounter, Data);
```

Amennyiben a kapott CRC és az itt kiszámolt CRC értéke nem megegyező, akkor a State pointer státuszát átállítjuk E2E_P01STATUS_WRONGCRC-re és a további műveletek elvégzése előtt a státuszt szükséges vizsgálnunk, hogy ne fussunk hibás működésbe és megfelelő státuszkóddal tudjuk visszatérni.

Itt fontosnak tartom kiemelni a checksum kalkuláció jelentőségét, ugyanis így biztosítható az, hogyha bármilyen hiba esetén (hardveres hiba, interferencia, szoftveres implementáció hibája stb.) adatvesztés vagy adatmódosulás következett be, akkor azt detektálni tudjuk és jelezzük az E2E Library-t hívó modulnak.

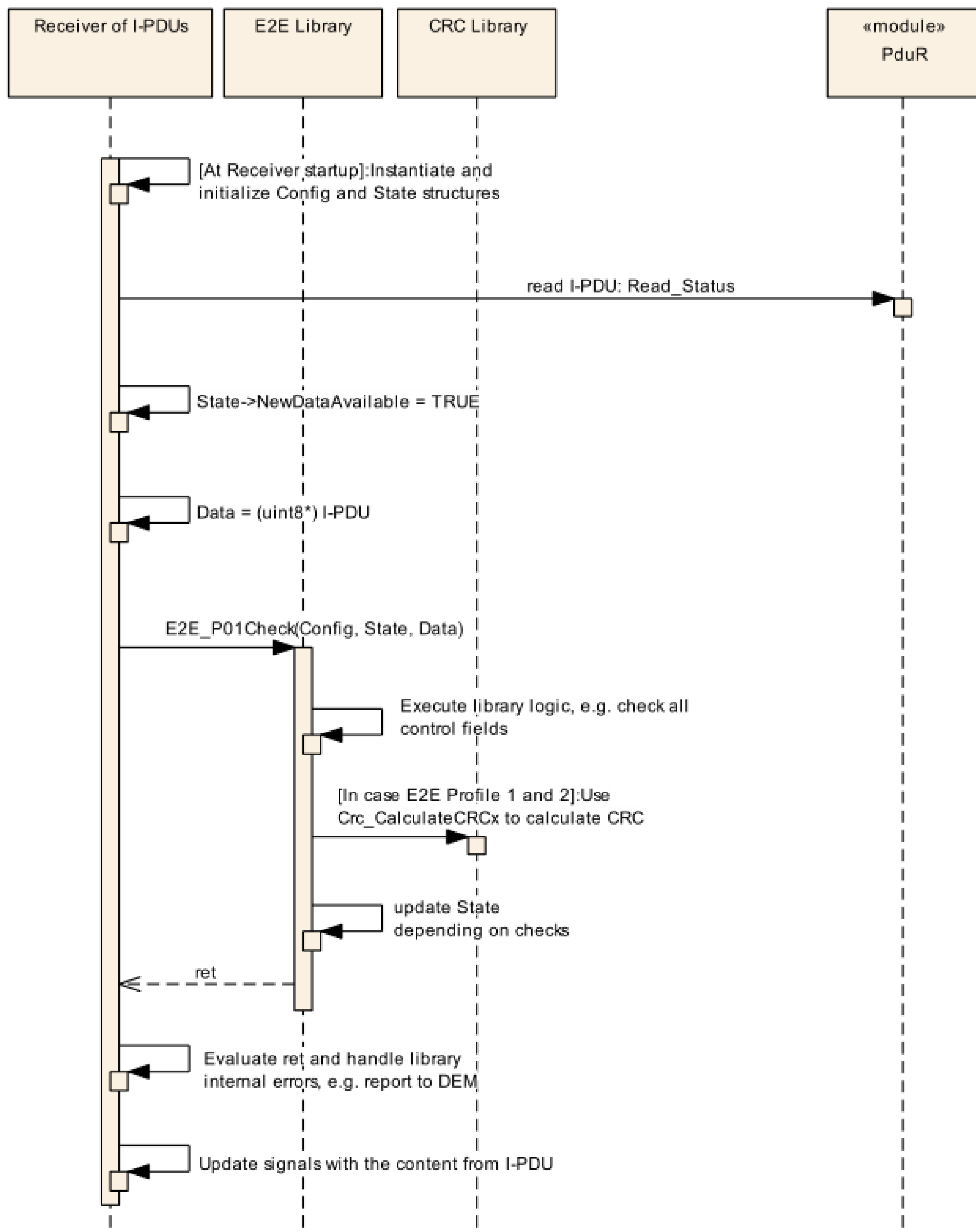
Mint ahogy a dolgozatom során már említettem, ehhez a CRC Library-t hívjuk meg. A CRC számítást ezért nem is részletezem, mivel egy különálló modulba van szervezve (rétegzett architektúra, feladatok kisebb részegségekre bontása) és egy másik modul fejlesztőjeként egyedül a megfelelő API-t szükséges meghívni. P01-es profil esetén 8 bites CRC kalkulációt használunk (tehát az ehhez megfelelő függvényt kell meghívni, amely az alábbi: Crc_CalculateCRC8()), a többi profil esetén használunk továbbá 16 és 32 bites checksum kalkulációt is.

Abban az esetben, ha érkezett új adat, akkor kiszámítjuk a különbséget a fogadott számláló és a State pointer LastValidCounter értéke között. Erre azért van szükség, mivel így tudjuk megvalósítani az úgynevezett timeout monitoring-ot. Tehát, ha bármi hiba lép fel és nem kapunk érvényes adatot egy meghatározott ideig, akkor a profilnak a szabvány által definiáltak szerint kell ezt kezelnie. Ez a statikus forráskódban így néz ki:

```c
/* Calculate the difference between counters */
  delta = E2E_P01_calculateDeltaCounter(receivedCounter,
                                        State->LastValidCounter);


/* UC_E2E_00206, normal functioning */
  if (delta == 1U) {
       State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
       State->LastValidCounter = receivedCounter;
       State->LostData = 0U;
       State->Status = E2E_P01STATUS_OK;
  }

/* If the (same) data arrived again with the same counter */
  else if (delta == 0U) {
       State->Status = E2E_P01STATUS_REPEATED;
  }

/* Lost Data */
  else if (delta <= State->MaxDeltaCounter) {
       State->MaxDeltaCounter = Config->MaxDeltaCounterInit;
       State->LastValidCounter = receivedCounter;
       State->LostData = delta - 1U;
       State->Status = E2E_P01STATUS_OKSOMELOST;
  }

/* Last possibility, wrong sequence */
  else {
       State->Status = E2E_P01STATUS_WRONGSEQUENCE;
  }
```

Amennyiben a kiszámított delta értéke egy, akkor átállítjuk a State struktúra szükséges elemeit és visszatérünk. Ha értéke nulla, akkor az azt jelenti, hogy a már egyszer fogadott adat érkezett még egyszer. Ha értéke az előzőktől különböző és a State pointer MaxDeltaCounter által meghatározott értéknél kisebb, akkor az azt jelenti, hogy a küldés folyaman két sikeres fogadás között valamennyi adat elveszett. Amennyiben az előbbi esetek egyike sem áll fent, akkor pedig rossz sorrendben érkeztek az adatok.

A 16. ábra remekül szemlélteti az E2E_P01Check meghívását szignál szinten. Látható, hogy a fogadott adat ellenőrzésében játszik szerepet az E2E Library, amelynek visszatérési értékének függvényében a fogadó fél jelenti az esetleges hibát vagy sikeresen megtörtént az adat ellenőrzése, a fogadott szignál megfelelő volt.



16.ábra: E2E Library szerepe fogadás estén [6]

4.2.4 Különbségek az E2E Library profilok között

Az E2E Communication Protection Library modul egyes profiljai mögött meghúzódó logika alapvetőleg ugyanaz, főleg a konfigurációs struktúra elemeiben van eltérés, ebből fakadóan az ezt felhasználó műveleteket kellett máshogy megalkotni.

4.2.4.1 E2E P02 profil

A kettes profil legfontosabb különbsége az egyeshez képest, hogy jóval nagyobb adattömbbel tudunk dolgozni. Az egyes profil esetén a maximális adathossz 240 bit, míg a kettes profil esetén 2048 bit. Ebből kifolyólag nincs lehetőség meghatározni, hogy a *DataID* mely bitjein történjen a CRC kalkuláció. Ennek oka, hogy a jóval nagyobb adathossz miatt ezen profil esetén egy 16 ID-t tartalmazó *DataIDList*-et definiál a konfigurációs struktúra, amelyek mindegyikén történik a checksum kalkuláció.

4.2.4.2 E2E P04 és P05 profilok

Ezen profilok valamivel egyszerűbbek, mivel kevesebb mezőt tartalmaznak a konfigurációs struktúrák, ezáltal a profil mögött húzódó logika is jelentősen egyszerűsödik. A P05-ös profil kisebb adattömbök kezelésére lett kifejlesztve, konfigurációs struktúrája mindössze négy mezőből áll.

A P04-es profil esetén nincs konkrét adathossz a konfigurációs struktúrában, hanem egy intervallumot definiál minimum és maximum értékkel. A profil API-k meghívásakor ugyanakkor szükséges egy uint16 típusú *Length* változót átadni, amely az adat hosszát jelenti. Erre a profilra azért van szükség, mert előfordulhatnak olyan kommunikációs helyzetek (mivel az E2E Librarynak az összes modult megfelelően ki kell szolgálnia), amikor az adatnak nem fix hossza van, hanem ez egy tartományon belül mozoghat, erre egy rugalmas megoldást kellett találni (mint említettem az adathossz külön kerül átadásra, a pointer aritmetika miatt ez különösen fontos).

4.2.4.3 E2E P06 és P07 profilok

Hasonlóan a négyes és ötös profilokhoz, a hatos és hetes profilok adatstruktúrái is kevesebb mezőt tartalmaznak. A P06-os profil konfigurációs struktúrája nagyon hasonlít a négyes profilnál látottra, apró különbségekkel. Ugyanazon mezőket biztosítja, azonban más sorrendben és rövidebb adathosszúságokkal.

Míg a négyes profilnál 32 bites *DataID*-t használ a modul, addig a hatos profil esetében ez csak 16 bit. Hasonló módon, a *MaxDeltaCounter* és a *Counter* értéke is 8 bit lesz a 16 helyett a hatos profil esetén.

A P07-es profil viszont hosszabb adatok kezelését teszi lehetővé. Az előbb említett különbségekkel szemben mindegyik struktúra elem 32 bit hosszúságú. Mint ahogy a dolgozatban már említettem, az első két profil megértése után már könnyeben lehet haladni a fejlesztéssel, mivel a profilok között javarészt paraméterezésbeli különbségek vannak. A P04, P06 és P07 profilok alapvető funkcionalitása tehát megegyezik, viszont más adathosszúságú elemekkel dolgoznak, illetve a struktúrák mezőinek sorrendje is különböző.

4.2.4.4 E2E P11 és P22 profilok

A P11-es profil jelentősen hasonlít az első profilra, azonban annak egy egyszerűsített változata. A *ConfigType* struktúra három mezővel kevesebbet használ (*MaxDeltaCounterInit*, *MaxNoNewOrRepeatedData*, *SyncCounterInit*), a *CheckStateType* struktúra viszont már csak a státuszkód tárolásáért felelős Status változót tartalmazza. Ezekből következik, hogy a profil belső logikája jóval egyszerűbb mint az egyes profil esetén.

A P22-es profil a kettes profil egyszerűsített verziója. A ConfigType adatstruktúra három elemmel kevesebbet tartalmaz, mint a P02-es profil konfigurációs struktúrája (*MaxDeltaCounterInit*, *MaxNoNewOrRepeatedData*, *SyncCounterInit*). Hasonlóan a kettes profilhoz, a *DataID* helyett itt is *DataIDList* kerül felhasználásra, ezáltal a többi profilhoz képest hosszabb adatokon hajtja végre a modul a checksum kalkulációt.

4.2.5 Doxygen dokumentáció

A Doxygen egy kóddokumentációs eszköz, használatához az Eclipse fejlesztőkörnyezetben be kell állítani pluginként, majd a szokásos blokk-komment első sorát (/*) kell kiegészíteni az alábbi módon: /**. Amennyiben ezt egy függvény definíciója előtt tesszük, a Doxygen automatikusan kigyűjti a rutin paramétereit, illetve visszatérési értékét.

Szoftverfejlesztés során mindig fontos szerep jut a megalkotott logika dokumentálásának. Ez többféle módon történhet, akár egymást kiegészítve. Jelen esetben ezt a Doxygen tool-al, kommentekkel végezzük. Egy jól kommentezett kódot később sokkal könnyebb újra megérteni (legyen az akár a fejlesztő saját maga vagy egy másik kolléga), mint kommentek nélkül.

Példaként itt a Check API Doxygen dokumentációját említem, amely az alább látható módon néz ki:

```
/**
 * SWS_E2E_00158
 * @param Config: Pointer to static configuration.
 * @param State: Pointer to port/data communication state.
 * @param Data: Pointer to received data.
 * @return Std_ReturnType flag
 * Description: Checks the Data received using the E2E profile 1.
 * This includes CRC calculation, handling of Counter and Data ID.
 */
```

4.3 Dinamikus forráskód

A legtöbb modul fejlesztése esetén beszélhetünk statikus és dinamikus forráskódról is. Ennek magyarázata az, hogy sok konfigurációfüggő beállítás létezhet egy nagyobb modul esetén, amelyet statikusan nem tudunk a kódba írni, hanem a modellező szoftverben megalkotott definíciót alapul véve kódgenerálást végzünk. Ennek a folyamatát szemlélteti az alábbi ábra:

17. ábra: Kódgenerálás folyamata

A generálás Java alapokon nyugszik. A folyamat elején a kódgenerátor bejárja a modul konfigurációját, az így kiolvasott adatokból struktúrát hoz létre. Ezen a beolvasott struktúrán konzisztenciaellenőrzést hajt végre, majd az adott konfigurációnak megfelelően meghívásra kerülnek a generáló metódusok, amelyek előállítják a forráskódot. Ezután az így előállított kód fájlba íródik, így elkészült az immáron C nyelvű (konfigurációból előállított dinamikus) kód, amelyet egybefordítunk a statikus kódrésszel.

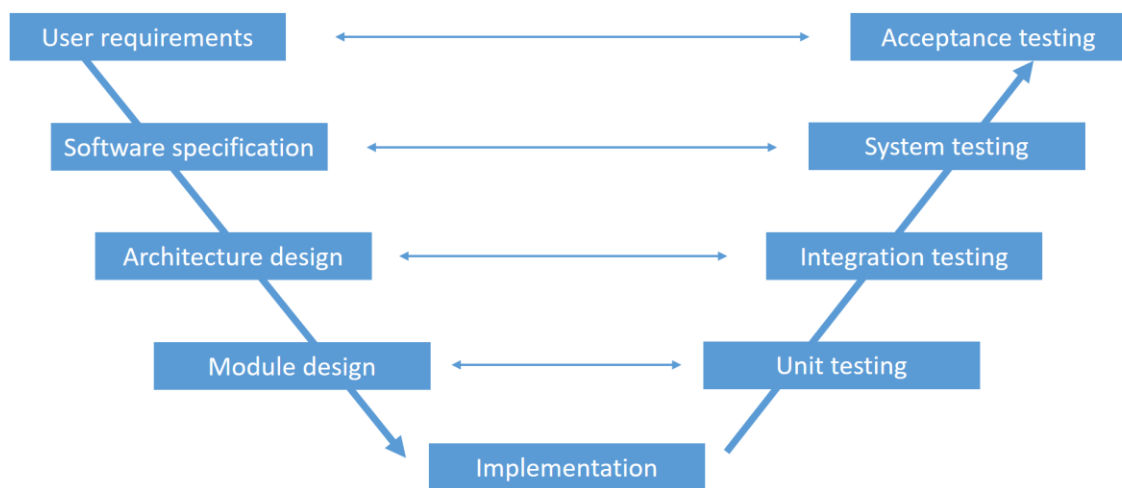
A Library-k úgy lettek specifikálba az AUTOSAR-on belül, hogy esetükben tipikusan nem szükséges dinamikus forráskódot előállítani. Fontosnak tartottam azonban nagyon tömören ismertetni, hiszen egyéb, nem Library BSW modulok fejlesztése esetén rendkívül fontos szerep jut a konfigurációfüggő forráskód előállításának.

4.4 Tesztelés

A tesztelés fontos szerepet kap szoftverfejlesztés esetén, az autóiparban ez hatványozottan igaz. Mielőtt a céges részlegen belül megírt szoftver valódi autók vezérlőegységein futna, számos felülvizsgálatnak és tesztelésnek kell megfelelni, hiszen biztonságkritikus rendszerről beszélünk, ahol egy apró hiba akár emberéletekbe is kerülhet.

4.4.1 V-modell

Az autóipari szoftverek fejlesztését és tesztelését az úgynevezett V-modell foglalja össze, mely a 18. ábrán látható.

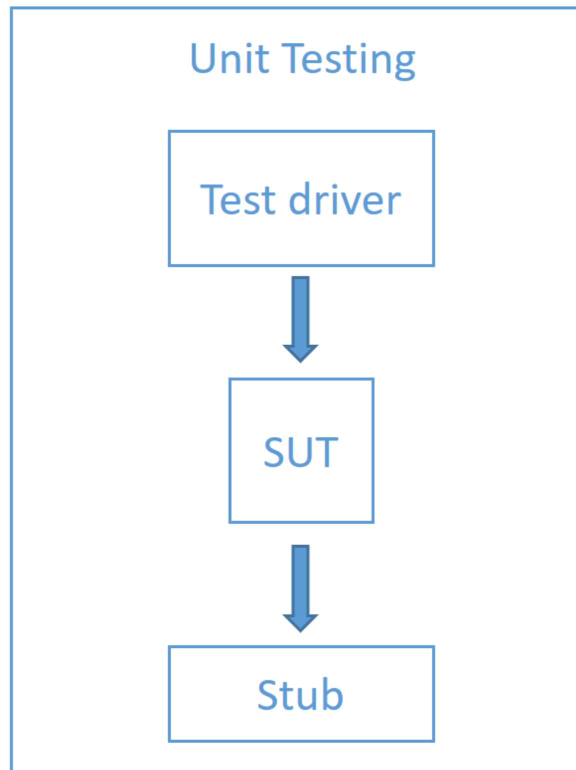


18. ábra: V-modell

A bal oldali ág mutatja a fejlesztési, a jobb oldali ág pedig a tesztelési lépéseket. Miután egy modul implementálása befejeződött, a következő lépés a Unit testing, amely a modulon belül az egyes függvényeket ellenőrzi. Ezt követi az Integration testing, ahol a modulok interfészei kerülnek összefordításra és ellenőrzésre. Ezután következik a System testing, ami során az egyes alrendszerek kerülnek tesztelésre. Itt példaként megemlíthető a CAN mint alrendszer, ugyanis számos modulból tevődik össze, amelyeket egybefordítva és tesztelve bizonyosodhatunk meg a CAN szolgálatások megfelelő működéséről. A legfelső lépcsőfok a V-modell jobb oldali ágában pedig az Acceptance testing, ahol a vevői igényeknek és konfigurációknak megfelelően egy utolsó ellenőrzésen esik át a szoftver.

4.4.2 Unit testing

Mint ahogy az előbb említettem, a V-modell első lépcsőfoka az úgynevezett Unit testing. Szakdolgozatom keretein belül ennek egy alapverzióját készítettem el az implementáció ellenőrzése érdekében, ezért ezt röviden szeretném ismertetni, működésének folyamata a 19. ábrán látható.



19. ábra: Unit testing

Unit test esetén követelmény alapú tesztelési megközelítést használunk. Ennek lényege, hogy a teszteseteket és adatstruktúrákat a követelményekből származtatjuk. A hívó felet a test driver helyettesíti, ami a tesztesetek alapján, különböző bemenetekkel hajtja meg a szoftvert. A tesztelés alatt álló szoftvert a 19. ábra közepén elhelyezkedő SUT szemlélteti, amely a Software Under Test kifejezés rövidítése. A legalul található Stub pedig helyettesíti a SUT által használt más modulok API függvényeit, a teszt ezen függvények megvalósításában így képes megfigyelni a SUT külvilággal történő interakcióit.

Az úgynevezett smoke teszteket a fejlesztő a saját munkájának ellenőrzésére készíti még a hivatalos, más által írt unit teszt előtt.. A smoke teszt lényege, hogy a függvények alapvető működését ellenőrzi. Amennyiben itt hiba lép fel, akkor a további

tesztelések előtt ezeket ki kell javítani. Ha a smoke tesztek sikeresen lefutottak, akkor következhet a függvények többféle paraméterekkel (és ezáltal adatstruktúrákon) való ellenőrzése.

4.4.3 Tesztesetek

Az alábbiakban röviden felvázolom a tesztesetek négy nagy csoportját, amelyeket a modul ellenőrzéséhez definiáltam.

4.4.3.1 NULL pointer

A tesztelés során mindig fontos ellenőrizni, hogy a megvalósítás miként kezeli le a NULL pointereket (mert ezek különösen komoly hibákat idézhetnek elő). Ahol az adott függvénynek pointert adunk át, ott célszerű a tesztelés során NULL pointerrel is elvégezni a függvényhívást.

4.4.3.2 Out-of-range

Az out-of-range hibák akkor fordulnak elő, amikor egy átadott változó a szabvány által definiált intervallumon kívül esik vagy nem teljesül az oszthatóság feltétele (például a számlálóoffset 8 többszöröse). Ennek kezelésének vizsgálatára a ConfigType struktúra DataLength elemét rosszul inicializálva adtam át az E2E_P01Protect függvénynek, az alábbi módon:

```
void setConfigPtr_CounterOffset_wrong(E2E_P01ConfigType *ConfigPtr){
	ConfigPtr->CounterOffset = 8;
	ConfigPtr->CRCOffset = 0;
	ConfigPtr->DataLength = 33;
	ConfigPtr->DataIDNibbleOffset = 0;
	ConfigPtr->DataID = 100;
	ConfigPtr->MaxDeltaCounterInit = 2;
	ConfigPtr->MaxNoNewOrRepeatedData = 14;
	ConfigPtr->SyncCounterInit = 2;
  }
```

Itt a DataLength kívül esik a szabvány által definiált tartományon (maximum 32 bit P01-es profil esetén).

4.4.3.3 Hibás cheksum vagy counter érték

P01-es profil esetén a CRC az adattömb 0. indexű elemében, míg a számláló az 1. indexű elemében kerül letárolásra. Amennyiben ide bármilyen hiba folytán más érték kerül, mint a Check függvény által számított, azt is megfelelően kell kezelnie a profilnak.

4.4.3.4 Helyes értékek

Helyesen inicializált adatstruktúrákkal is szükséges a profilokat vizsgálni, hiszen ez a normális működés.

```
----------------P01 Check begin----------------

 Enum decode:
       E2E_E_INPUTERR_NULL = 19
       E2E_E_INPUTERR_WRONG = 23
       E2E_E_INTERR = 25
       E2E_E_OK = 0
       E2E_E_WRONGSTATE = 26

  ----P01_Protect_check----

 Null pointer check should be 19, is: 19

 CheckConfiguration, with good params should be 0, is: 0
 CheckConfiguration, with out-of-range params should be 23, is: 23

 ----P01_Check_check----

 Null pointer check returns should be 19, is: 19

 Check result should be 0, is: 0

 Status should be: 2, is: 2


 ----P01_MapStatusToSM_check----

 profileBehavior = TRUE
 should be: 0, is: 0
 should be: 0, is: 0
 should be: 0, is: 0
 should be: 3, is: 3
 should be: 1, is: 1
 should be: 5, is: 5
 should be: 2, is: 2
 should be: 2, is: 2

 profileBehavior = FALSE
 should be: 0, is: 0
 should be: 0, is: 0
 should be: 0, is: 0
 should be: 3, is: 3
 should be: 1, is: 1
 should be: 5, is: 5
 should be: 2, is: 2
 should be: 2, is: 2


----------------P01 Check end----------------
```

20. ábra: P01-es profil teszteredmények

Összességében elmondható, hogy az implementáció néhány apróbb logikai hibától eltekintve helyesen működött, a tesztek az elvárt értékekkel tértek vissza, mint ahogy az a 20. ábrán is látható. A kisebb hibák alatt gondolok a feltételvizsgálatoknál elkövetetett pontatlanságokra, valamint az adattömb offsetekkel való indexelését kellett pontosítani. A detektált hibákat javítottam és így nincs ismert hiba az implementációban.

5 Tapasztalatok és további lehetőségek

5.1 Tapasztalatok

A szakdolgozatom írása során lehetőségem volt folytatni az Önálló Laboratórium, valamint Szakmai gyakorlat alatt már megkezdett elmélyedést az autóipari szoftverfejlesztés világában. Ezen projekttárgyak alatt a kriptográfiai és különböző biztonsági funkciók működésébe nyerhettem betekintést, ami, úgy gondolom, hogy az iparági fejlődéseket figyelembe véve rendkívül hasznos tudást adott.

Ugyanakkor különösen érdekes volt abban is elmélyedni, hogy a kommunikáció során hogyan detektálunk különféle hibákat és tudunk megoldást biztosítani a biztonságos működésre ezek fennállása mellett is, legyenek ezek akár szoftveres, akár hardveres eredetűek.

Az End-To-End Communication Protection Library egy különálló egységet képez (library mivoltából fakadóan) az AUTOSAR architektúrájában, szoros kapcsolatban áll azonban a Basic Software réteggel. Ebből kifolyólag az előző projekttárgyak során elsajátított tudáshalmazhoz tökéletesen kapcsolódott, rendkívül hasznosnak érzem a félévek során tanultakat.

A szakdolgozatom első lépéseként a háttérismeretek elsajátítása volt a feladatom. Ez magában foglalja az AUTOSAR rétegzett architektúrájában a BSW réteg kommunikációért felelős moduljainak megismerését. Jelenti továbbá az E2E Library modul fő funkcióinak és a környező entitásokkal történő kapcsolatának megismerését. Ezen lépés legnagyobb kihívását az architektúra részletekbe menő átlátása jelentette, ugyanis a szabványleírás alapvetőlen egyfajta tényszerű közléseket tartalmaz, viszonylag kevés megértést segítő résszel.

A következő lépés a modul követelményeinek elemzése volt a megvalósíthatóság szempontjából. Ezt követte a modul C nyelven történő implementálása a követelménylista alapján. Alapesetben egy AUTOSAR modul statikus és dinamikus részt is tartalmaz. Azonban az E2E modul esetén dinamikus rész előállítása nem volt szükséges, mivel a library-k csak statikus forráskódot tartalmaznak, konfigurálható részt nem.

Gyakorlatilag ez a lépés jelentette a munka oroszlánrészét, hiszen itt kellett biztosítani azt, hogy a modul a követelményeknek megfelelően működjön. Az AUTOSAR szabvány definiálja a külvilág felé elérhető API-kat, ezek implementálása már a fejlesztőmérnök feladata.

Szakdolgozatom ezen részében számos kihívással találkoztam, mivel beágyazott környezetben az aktuális problémára a több megoldási mód közül az optimálisat keressük. Ezt a dolgozatban már részletesen kifejtettem, röviden azért igényel speciális figyelmet az optimalizált működés, mert így erőforrást és ezáltal gyártási költséget tudunk csökkenteni. Ilyen volt például a modulo osztás bitenkénti ÉS kapcsolattal történő megoldása, vagy akár a logika tömörebb kialakítása.

Az E2E Library modul mind a nyolc profilját sikeresen implementáltam. Itt fontosnak tartom kiemelni, hogy az egyes profilok között paraméterezésbeli különbségek vannak javarészt. Ez azt jelenti, hogy ha a mögöttes logikát sikerül elsajátítani, akkor már könnyebbé válik az egyes profilok implementálása. Ebből kifolyólag a szakdolgozatom egyik legnagyobb kihívása a modul működési logikájának megértése volt.

Összegezve: a feladatkiírásban foglaltakat a háttérismeretek elsajátításával, valamint a modul tervezésével és implementálásával kapcsolatban sikeresen teljesítettem.

Mindezek után fontos része volt a feladatomnak az implementált modul megfelelő működését tesztesetekkel ellenőrizni. A munkám ezen részében egy olyan egyszerűbb modulteszt infrastruktúrát hoztam létre, amelyben a megtervezett tesztesetekkel ellenőrizhető a megvalósított modul helyessége.

Külön kiemelten kell vizsgálni az olyan eseteket, amikor (például hardveres meghibásodás miatt) nem megfelelő adattal kerül a modul API-ja meghívásra. Gondolok itt többek között a NULL pointerekre, vagy akár az out-of-range változó értékekre.

Valamennyi profilra készítettem teszteseteket, amelyek segítségével javítani tudtam az implementáción. Fontos tanulság volt tesztelői szemmel megközelíteni az addig elkészített munkámat, azonban úgy gondolom, hogy számos tapasztalattal lettem gazdagabb a feladatom ezen részének megoldása alatt.

Összefoglalva a fentebb leírtakat, szakdolgozatom alatt sokat tanulhattam az autóipari szoftverfejlesztésről. Sikeresen teljesítettem a feladatkiírásban megfogalmazottakat, kezdve a háttérismeretek elsajátításával, folytatva a modul megvalósításával, befejezve a különböző tesztesetek megalkotásával és lefuttatásával.

Ami azonban talán a legfontosabb, hogy immáron el tudom helyezni az E2E Communication Protection Library modult az AUTOSAR architektúrájában, megismerve a működését lehetőségem volt megérteni, hogy miért olyan fontos a komplexebb feladatok részegységekre történő bontása.

5.2 Továbbfejlesztési lehetőségek

Mint ahogy fentebb említettem, a modul összes profilja implementálásra került a szakdolgozatom keretein belül. A tesztek sikeresen lefutottak, azonban a teszteseteket szeretném kibővíteni, célként az egész modul átfogóbb, alaposabb ellenőrzését kitűzve.

A tesztelés előtt azonban még célszerű egy követelmény ellenőrzést elvégezni, amiben egy szenior munkatárs fog segítséget nyújtani. Erre azért van szükség, nehogy kimaradjon valamilyen szükséges követelmény, továbbá a szabvány követelményei között lévő esetleges ellentmondások feloldásra kerüljenek.

5.3 Köszönetnyilvánítás

Ezúton szeretném megköszönni a szakdolgozatom írása közben nyújtott szakértelmet és szakmai segítséget Szikszay László konzulensemnek és a többi munkatársnak a thyssenkrupp Components Technology Hungary Kft. AUTOSAR/BSW részlegén.

Véleményem szerint még hallgatóként úgy lehet az egyetemen tanultakat a legjobban kamatoztatni az iparban, hogy olyan szintű feladaton dolgozhatunk, mint a már végzett (és természetesen nálunk sokkal tapasztaltabb) mérnökök, mégis több támogatást kapva. Ez azért nagyon fontos, mert a kezdő kolléga így maga küzd meg az egyes kihívásokkal, rengeteget tanulva ezáltal, de ha mégis nagyobb akadály kerül az útba, akkor van kihez fordulni, aki egy teljesen más szemléletet biztosítva segít megoldani az adott problémát. Úgy gondolom, hogy ez a thyssenkruppnál maximálisan megvalósult a projekttantárgyak alatt és ezért hálás vagyok a konzulensemnek és a kollégáknak.

Továbbá szeretném megköszönni egyetemi konzulensemnek, Dr. Sujbert László habilitált docensnek a segítségét, mely magában foglalja mind a szakmai, mind a technikai jellegű javaslatokat, amellyel támogatta szakdolgozatom elkészítését.

Irodalomjegyzék

[1] BME MIT, AUTOSAR alapú autóipari szoftverrendszerek, https://www.mit.bme.hu/oktatas/targyak/vimiav15/jegyzet (2020. nov.)

[2] AUTOSAR, Layered Software Architecture, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (2020. nov.)

[3] BME AUT, Dr. Tevesz Gábor, Mikrokontroller alapú rendszerek, 3. fejezet https://www.aut.bme.hu/Course/VIAUAC06 (2020. nov.)

[4] National Instruments, FlexRay Automotive Communication Bus Overview, https://www.ni.com/de-at/innovations/white-papers/06/flexray-automotive-communication-bus-overview.html (2020. nov.)

[5] AUTOSAR, Specification of Secure Onboard Communication, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_SecureOnboardCommunication.pdf (2020. nov.)

[6] AUTOSAR, Specification of SW-C End-to-End Communication Protection Library https://www.autosar.org/fileadmin/Releases_TEMP/Classic_Platform_4.4.0/Libraries.zip (2020. nov.)