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

Vajay Levente

AUTOSAR Diagnostic Log and Trace modul megvalósítása

Külső konzulens:
Szikszay László
thyssenkrupp Components Technology Hungary Kft.

Egyetemi konzulens:
Dr. Sujbert László

2020

MŰEGYETEM 1782

SZAKDOLGOZAT-FELADAT

Vajay Levente (GPSH0G)
szigorló villamosmérnök hallgató részére

AUTOSAR diagnosztikai Log and Trace-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 szolgáltatási rétegben lévő moduljának a megvalósítása 4.3 szabvány verzió alapján az alábbiak szerint:

- *A szabvány kapcsolódó részeinek megismerése*: ismertesse az AUTOSAR rétegzett BSW struktúráján belül a kommunikációs és diagnosztikai stack moduljainak szerepét. Foglalja össze a Diagnostic Log and Trace modul fő funkcióit és kapcsolatát a környező entitásokkal.
- *Szoftvertervezés és megvalósítás*: első lépésben elemezze a modul követelményeit megvalósíthatóság szempontjából. A szabvány a modulok 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. A dinamikus kódrészletek előállítása a szabványos adatmodellből az aktuális megvalósítás feladata úgy, hogy az illeszkedjen a statikus részhez és megfeleljen az alkalmazási terület erőforrás-használatra vonatkozó követelményeinek (pl. minimális ROM

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék
1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

használat). *Tervezze* és *valósítsa meg* a modul *statikus részét* C nyelven, a modul *kódgenerátorát* pedig Java 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

HALLGATÓI NYILATKOZAT

Alulírott Vajay Levente, 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. 15.

.....

Vajay Levente

Összefoglaló

Az autóipar világában az egyre bonyolultabb és összetettebb mechanizmusok mellett egyre nagyobb kihívást jelent a beágyazott szoftverek implementálása és tesztelése. Ezért a nagyobb gyártók és beszállítók megalapították az AUTOSAR konzorciumot, amely létrehozott egy egységes szoftverarchitektúrát és modellezési nyelvet. Az architektúrában megtalálható a BSW (Basic Software) réteg, ami egy egységes felületet képez az elektronikai vezérlőegység (ECU) felett, eltakarva annak specialitásait és biztosítva a „hasznos” alkalmazások hordozhatóságát. A szakdolgozat célja BSW-ben megtalálható kommunikációs és diagnosztikai modulok átfogó bemutatása, valamint az általam implementált „Diagnostic Log and Trace” modul felépítésének és működésének részletezése.

Dolgozatom, a félévben megismert irodalmak alapján, az autóiparban alkalmazott szoftverarchitektúra felépítésének bemutatásával kezdődik, majd a kommunikációs rétegben található modulok átfogó működését ismertetem. Ezek után a jármű diagnosztikájáért felelős protokollokat és két főbb komponensét vizsgálom meg. Végül rátérek szakdolgozatom témájára, amelyben elsőként igyekszem egy átfogó képet nyújtani a diagnosztikai modulom funkcionalitásáról és elhelyezkedéséről, majd a szabvány által felállított követelmények klasszifikálásának rendszerét ismertetem áttekintőleg és példák segítségével szemléltetve. Ezután a modul dinamikusan változó részének felépítését ismertetem. Végül a modul által biztosított API-k megvalósítását és a modul tesztelésének módszerét mutatom be.

Abstract

In the automotive world, with increasingly sophisticated and complex mechanisms, implementing and testing embedded software is an increasing challenge. Therefore, major manufacturers and suppliers formed the AUTOSAR consortium, which created a unified software architecture and modelling language. The architecture includes a BSW (Basic Software) layer that forms a single interface over the electronic control unit (ECU), obscuring its specialties and ensuring the portability of "useful" applications. The aim of the thesis is to comprehensively present the communication and diagnostic modules in BSW and to detail the structure and operation of the "Diagnostic Log and Trace" module I have implemented.

Based on the literature reviewed in the semester, my thesis begins with a presentation of the structure of the software architecture used in the automotive industry, and then I describe the comprehensive operation of the modules in the communication layer. I then examine the protocols responsible for vehicle diagnostics and its two main components. Finally, I turn to the topic of my dissertation, in which I first try to provide a comprehensive picture of the functionality and location of my diagnostic module, and then I present the system for classifying the requirements set by the standard with an overview and examples. Next, I describe the structure of the dynamically changing part of the module. Finally, I present the implementation of the APIs provided by the module and the method of testing the module.

1. Bevezetés

Minden ember már gyerekkorától kezdve találkozik a világ mérnöki csodáival, ezért sok fiatalt megkérdezve gyakori válaszként halljuk, hogy foglalkozásként autószerelők szeretnének lenni. Ez velem is hasonlóan történt. Lenyűgöztek és a mai napig foglalkoztat az, hogy mitől is működnek mindennapi használati tárgyaink, az autóink. Az idő előrehaladtával a járművek összetettsége folyamatosan növekszik. A mai világban már minden autó képes eljuttatni a felhasználót a kívánt helyre, így az autóiparban nem a cél elérése, hanem az utazás minél magasabb minősége és biztonsága jelent fejlődési lehetőséget.

Általában, mikor diagnosztikáról beszélünk, az emberek a szervízben való hibakód kiolvasására gondolnak, pedig ez ennél jóval több. Alapvetően minden összetett struktúrának szüksége van ellenőrzésre, a váratlan meghibásodás nélküli működés biztosítására.

A félév során igyekeztem alaposabban megismerni az autóiparban használt szoftverarchitektúrát és azon belül a kommunikációs és diagnosztikai stack moduljainak szerepét. Az AUTOSAR mozaikszó egy szabványcsaládot reprezentál, amely elnevezése az *AUTomotive Open System ARchitecture* angol kifejezésből ered. Számos diagnosztikáért felelős komponens teszi lehetővé számunkra, hogy ha a járműben egy vagy több alkatrész a specifikációja határán működik, (vagy már meg is hibásodott) időben észrevegyük és elkerülhessünk egy komolyabb balesetet vagy nagyobb anyagi kárt.

A megvalósított Diagnostic Log and Trace (DLT) modul az AUTOSAR konzorcium által kiadott szabvány 4.3.0 verziója alapján készítettem el, a thyssenkrupp Components Technology Hungary Kft. segítségével. A modul megvalósítására a cég által fejlesztett modellező programot és a biztosított Eclipse fejlesztőkörnyezetet használtam fel.

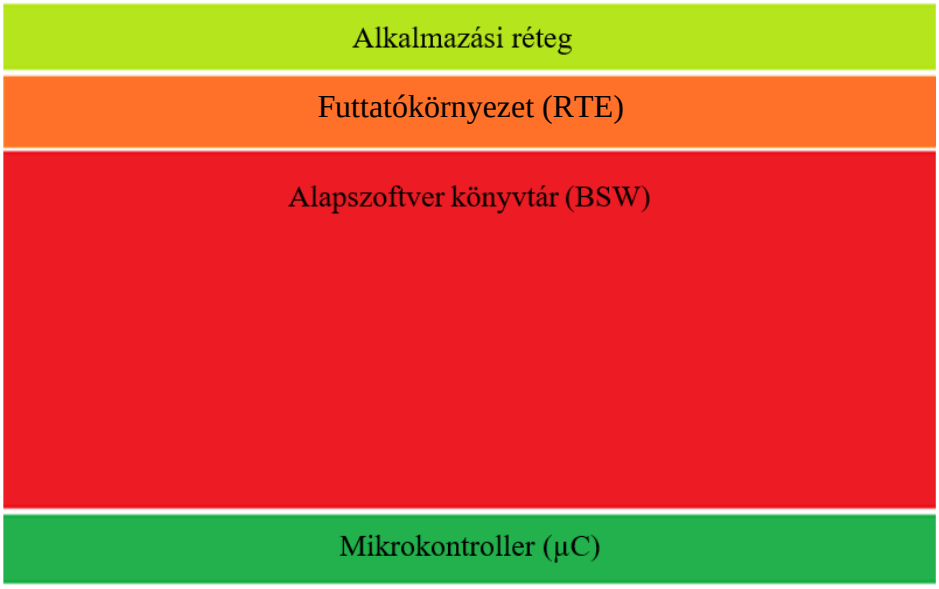
Dolgozatban elsőként bemutatom a kommunikációs és diagnosztikai stack moduljainak szerepét és elhelyezkedését az autóiparban használt szoftverarchitektúrán belül. A diagnosztikáért felelős modulok ismertetése után rátérek a félév során megvalósítandó modulom fejlesztésének lépéseire és részleteire.

2. A BSW elhelyezkedése és felépítése

Az AUTOSAR a mai modern járművek elektronikai vezérlőegységei számára kifejlesztett szabványcsalád. A szabvány tartalmaz alapvető szolgáltatásokat, amelyek a hordozhatóságot szolgálva eltakarják a hardverspecifikus tulajdonságokat és egységes kezelőfelületet mutatnak az alkalmazási szoftverek számára. Ez az alapszoftverréteg a BSW (Basic Software stack) amely tartalmazza az alapszolgáltatásokat. Az autóipari cégek és beszállítóik által mai napig is írt és szerkesztett szabvány, komponensorientált szoftverarchitektúrát fejleszt. Ami azt jelenti, hogy a BSW-ben működő egységek egymástól jól elkülönülő funkcionalitással rendelkeznek. A szabványcsalád három fő részre bontható.

A komponensorientált modellezési nyelv, amely egy szabványos felületet teremt, a gazadag alapszoftverkönyvtár, amely segítségével létrehoztak egy szabványos modellező nyelvet, és lehetővé tették a dinamikus kódgenerálást az egyes modulok együttműködése érdekében. A modulok tesztelhetőségi és megfigyelhetőségi vizsgálatára létrehoztak egy tesztkészletet és folyamatot, amellyel a modulokat vizsgálhatják a követelmények szerint. Mivel teljesítményhatékonyság és ezáltal költség szempontjából a C programozási nyelv a legalkalmasabb beágyazott rendszerek fejlesztésére, így a magasszintű nyelvek hasznos fogalmait nélkülözni kell (Pl.: osztály vagy névtér fogalma). Ezeket konvenciókkal és kódgenerálással helyettesítik, ami magasabb szintű nyelv használatával (Java) történik.

A szabvány három különböző absztrakciós szintet különít el egymástól. Az Alkalmazási réteg, a Futtatókörnyezet, azaz RTE (Runtime Environment) és az Alapszoftver könyvtár (BSW). Ezek rétegződését az 1.ábrán láthatjuk.



1.ábra: Rétegzett szoftverarchitektúra.

Az alkalmazási rétegben kapnak helyet az alkalmazási szintű szoftverek, amelyek teljesen hardverfüggetlenek, így alkalmazásuk minden AUTOSAR kompatibilis környezetben lehetséges. A futtatott szoftverek szabványosított porton kapcsolódnak a Virtual

Function Bus-hoz (VFB), ami az alkalmazásoktól független módon látja el a kommunikációt az alkalmazások és a környezet között, így függetlenítve az applikációkat a vezérlőegységtől.

Az RTE-t a konfigurációtól függően generálják. A BSW legfelső szinten lévő moduljai ezen keresztül képesek kiszolgálni a „hasznos” alkalmazások igényeit és hozzáférést biztosít a mikrovezérlő szolgáltatásaihoz. Ezt a réteget látják az alkalmazások virtuális busznak.

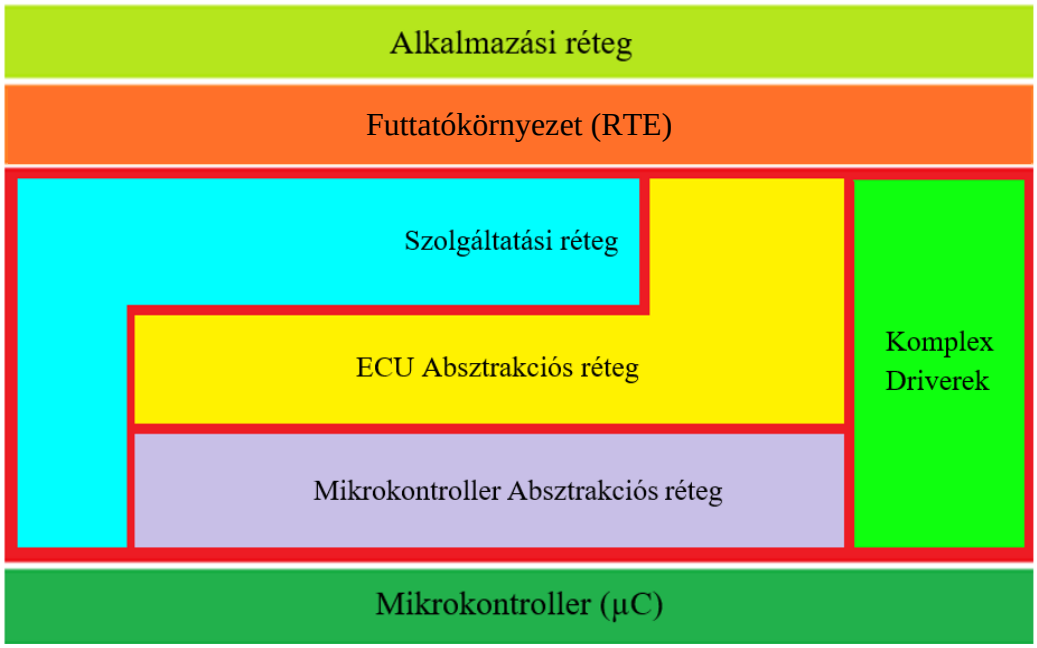
A BSW egy szabványos interfészekkel rendelkező moduláris könyvtár, amelynek egyes elemei a vezérlőegységhez mérten konfigurálhatóak, valamint a követelmények függvényében testreszabhatóak.

A BSW tovább bontható további négy részre. Szolgáltatási, ECU és Mikrokontroller absztrakciós, továbbá az ECU-hoz és RTE-hez közvetlenül kapcsolódó komplex driver rétegre. A Mikrokontroller absztrakciós réteg a BSW legalsó szintje, amely függ az adott vezérlőegységtől. Drivermodulokból áll, amelyek feladata biztosítani a független hozzáférést a mikrokontroller perifériáihoz és funkcióihoz.

A középső, az ECU Absztrakciós Réteg a driverek szolgáltatásaira támaszkodva nyújt hozzáférést a vezérlő-egység funkcióihoz, függetlenül attól, hogy az adott funkciót megvalósító periféria a mikrokontroller belső vagy külső perifériája. Összefoglalva, a réteg egy egységes interfészt valósít meg a funkciókhoz a vezérlőegység fizikai felépítésétől függetlenül.

A Szolgáltatási réteg a BSW legfelső szintje, amely hardverfüggetlenül alapvető szolgáltatásokat nyújt a BSW modulok, valamint az RTE-n keresztül az alkalmazási réteg számára. Ebben a rétegben történik a hálózatkezelési és -menedzselési feladatok lebonyolítása. Ellátja a memóriamenedzsment feladatokat és az ECU állapot és futási módját is kézben tartja. Ezek mellett a jármű diagnosztikájáért felelős modulok is itt találhatóak.

Végül a Komplex drivereket tartalmazó réteg következik. Az AUTOSAR által nem szabványosított szolgáltatásokat végző modulok találhatók itt. Közvetlenül az RTE és a kontroller között helyezkedik el, így direkt módon csatlakozik a hardver szenzor és beavatkozó szerveihez. Az itt található funkcióknak sokszor szigorú időzítési feltételeknek kell megfelelni és különleges biztonsággal kell feladatukat ellátni. A rétegek elhelyezkedését a BSW-n belül a 2.ábra szemlélteti.



2.ábra: A BSW felosztása

Az alapszoftvereket tovább bonthatjuk szolgáltatásaik szerint.

Az illesztőprogramok (driver) a belső vagy külső eszközök vezérlésének és elérésének funkcióit tartalmazzák. A be- és kimenetekért felelős (I/O) szolgáltatások szabványosított hozzáférést biztosítanak az érzékelőkhöz, beavatkozókhoz és a mikrokontroller fedélzeti perifériáihoz. Az I/O hardver absztrakcióhoz tartozó modulok csoportja függetleníti a perifériákat a fizikai elhelyezkedésüktől. Előfordulhat, hogy azonos porton akár több szenzor helyezkedik el.

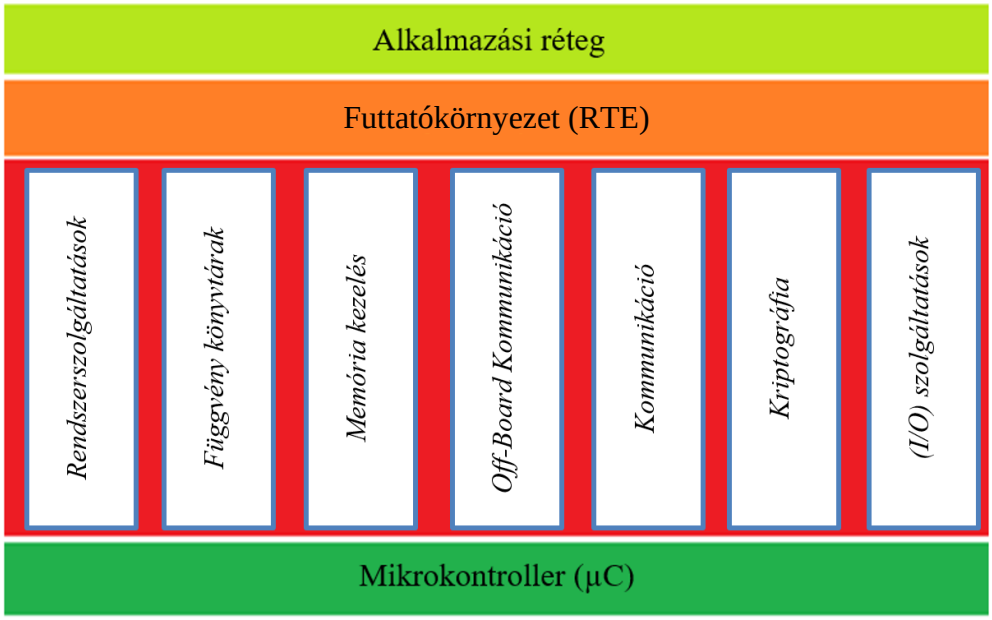
A modulok egy csoportja a nemfelejtő memóriához való szabványosított elérést teszi lehetővé, függetlenül a hardverelrendezéstől, lehet az külső vagy belső EEPROM, az elérési metódus nem függ tőle.

A perifériák illesztését a driverek végzik, amelyek a mikrokontroller és ECU absztrakciós rétegekben helyezkednek el.

A kriptográfiáért felelős modulok kontrollerfüggetlenül kínálnak egységes hozzáférési mechanizmust a belső és szoftveres titkosítási eszközökhöz. A hibakezelést, a feladatok végrehajtásának ütemezését és a valósidejűségért felelős operációsrendszeri feladatokat (pl.: időzítési szolgáltatások) a rendszerszolgáltatások csoportja látja el. Valamely rendszerszolgáltatás hardver, és kontrollerspecifikus (állapot- és időzítésmenedzsment).

A járművön belüli kommunikációs hálózatban elkülöníthető egy vezetéknélküli hálózat megvalósításáért felelős csoport, ami a torlódásszabályozásért és alapszintű protokollkezelési feladatok ellátásáért felelős, valamint egységes elérési felületet biztosít a vezetéknélküli Ethernet hálózathoz.

A belső kommunikációért felelős modulok standardizált metódust nyújtanak a vezérlőegységek és a szoftverkomponensek közötti információcseréhez. Minden kommunikációs rendszerhez egy specifikus kommunikációs hardver-absztrakció szükséges (pl.: LIN, CAN, FlexRay)[1]. A BSW funkcionalitás szerinti felbontását a 3.ábra mutatja be.



3.ábra: BSW Funkcionális felosztásban

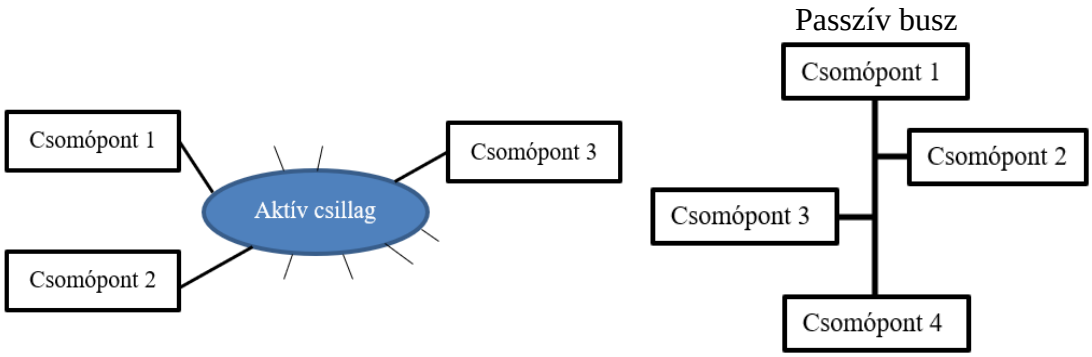
3. Kommunikációs stack

A járműben található vezérlőegységek elosztott hálózatként működnek. Minden vezérlő saját egyéni feladatát kell, hogy ellássa. Folyamatos kapcsolatban állnak egymással, így csak a feladatukhoz legszorosabban kötődő szenzorok által mért jelek alapján szükséges számításokat végezniük. Az egyéb, működéshez szükséges adatot, a többi egységtől gyűjtik be. Ezekből következik, hogy az alkalmazások hatékonysága erősen függ a kommunikáció minőségétől.

Az autóiparban leggyakrabban alkalmazott kommunikációs protokollok a CAN, LIN és FlexRay. A különböző feladatkörök ellátásában különböző sávszélesség és adatátviteli sebesség szükséges. A LIN kisebb és kevésbé kritikus egységek elérésére szolgál. Alhálózataként, buszrendszerben működik a CAN-hez kapcsolódva egy köztes eszköz segítségével (*gateway*), ezáltal csatlakozva a központi hálózatra és így a vezérlőegységekhez. A CAN aszinkron és eseményvezérelt. Adatátviteli sebessége legfeljebb 1Mbit/s és egy keret maximum 8 bájt méretű lehet. A járművekben központi hálózatként, buszrendszerben működik.

A nagyobb bitráta és az átvihető keretek növelése érdekében a klasszikus CAN helyett inkább az újabb fejlesztésű CAN FD (Controller Area Network Flexible Data-Rate) protokollt használják, amivel 64 bájtos keretméret és az ISO 15765-2:2016 verzió alapján, már 4.294.967.295 bájt átvitele is lehetséges[2]. A FlexRay a legfiatalabb protokoll. Nagysebességű (10 Mbit/s), hibatűrő és redundáns, ezáltal idő- és biztonságkritikus feladatok esetén is alkalmazható.

A különböző hálózatok és vezérlők kapcsolata topológia szerint lehet aktív, passzív vagy hibrid. Passzív topológiára jó példa a busz, aktívra a csillag és ezeket kombinálhatjuk is (4. ábra).[1]



4.ábra: Topológiák

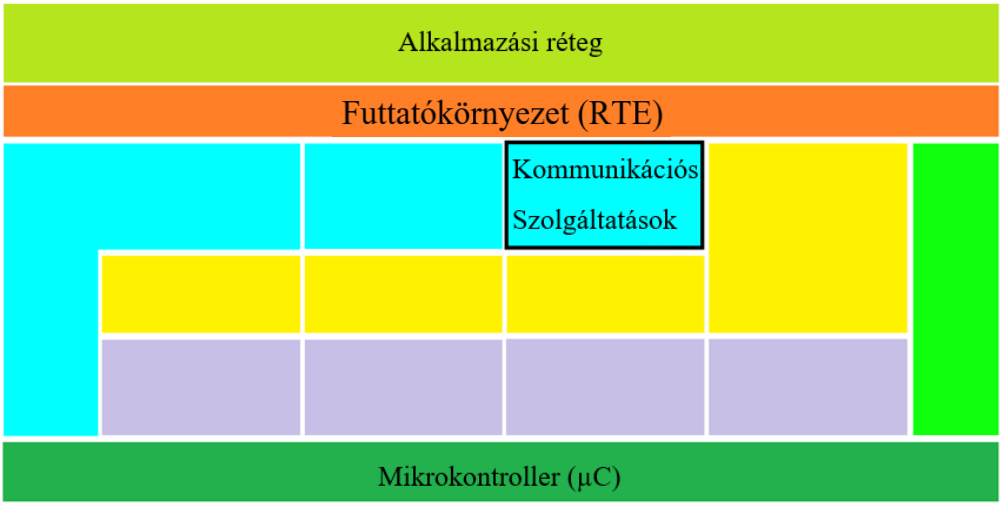
A modern járművekben a kommunikáció célja szerint is elkülöníthetünk nagyobb csoportokat. A legkisebb, önmagukban értelmes egységei a kommunikációnak a szignálok. Szignálalapú kommunikációt a vezérlők közötti periodikus üzenetváltásra használunk, amely a szenzoroktól érkező fizikai mennyiségek továbbítására alkalmas.

A hálózatmenedzsment üzenetek egyes alhálózatok lekapcsolásáért vagy ébresztéséért felelősek. Ezek az üzenetek is periodikusan, a vezérlőegységek között hordoznak információt, ezáltal karbantartva a jármű kommunikációs hálózatát.

A diagnosztikai kommunikáció esetén az üzenetváltás nem folyamatos. Kérések és válasz üzenetek „közlekednek” valamilyen külső eszköz és az ECU között. Szoftverfrissítésre, hibakódok kiolvasására használják, amelyek általában CAN vagy FlexRay Transport Layer szállításiréteg-protokollon keresztül történnek.

3.1. Rétegek közti kommunikáció

A kommunikációs szolgáltatások a szolgáltatási rétegben találhatók (5. ábra), a járműhálózat kommunikációs moduljainak egy csoportja tartozik ide(CAN, LIN, FlexRay és Ethernet). A kommunikációs hardver-absztrakción keresztül kapcsolódnak a kommunikációs meghajtókhoz. A szolgáltatások egységes interfészt nyújtanak az alkalmazási réteg számára, az RTE-n keresztül a jármű hálózatikommunikációhoz és a diagnosztikai-információcseréhez. Elrejtik a protokoll- és üzenettulajdonságokat továbbá egységes szolgáltatásokat nyújtanak a hálózatkezeléshez.



5.ábra: Kommunikációs szolgáltatások a BSW-ben

A kommunikációs szolgáltatásokon belül az egységeket két csoportba sorolhatjuk. Protokollfüggő és -független modulok csoportjába. Független modulokból ECU-nként egy példány létezik. Ilyen modul az AUTOSAR COM, a Communication Manager, a Generic Network Management Interface, a PDU Router és a Diagnostic Communication Manager.

Az AUTOSAR COM modul feladata továbbítani az alkalmazásoktól érkező szignálokat, megszűrni, majd I-PDU-kba (Interaction Layer Protocol Data Unit) csomagolni azokat. Avagy a hálózatról érkező PDU-k kicsomagolása és ezek alapján szignál továbbítása az adott alkalmazásnak az RTE-n keresztül.

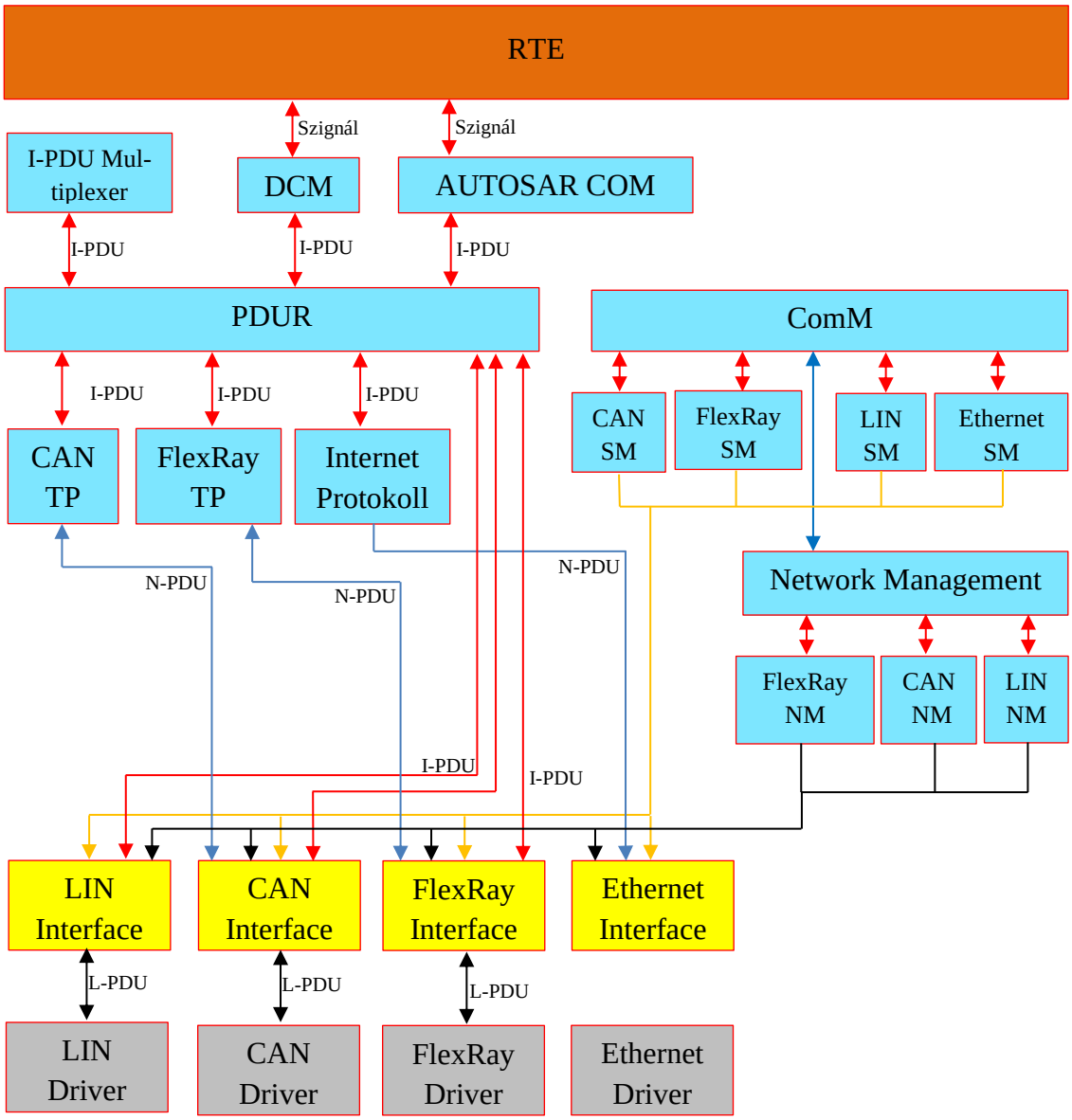
A Diagnostic Communication Manager (DCM) a diagnosztikai jellegű üzenetek forgalmáért felelős. Biztosítja a diagnosztikai adatáramlást és kezeli a diagnosztikai állapotokat, különösen a diagnosztikai munkameneteket és a biztonsági állapotokat. Ezenkívül ellenőrzi a diagnosztikai szolgáltatáskérés támogatottságát, hogy a szolgáltatás az aktuális munkamenetben végrehajtható-e a diagnosztikai állapot szerint[4].

A Communication Manager (ComM) kezeli a hálózati kommunikációs erőforrásokat, a hálózatok buszait és hardveregységeit összehangoltan és megbízhatóan működteti. A kommunikációs buszok használatát egyszerűvé teszi a felhasználók számára. A modul egyszerre több kommunikációs csatorna kezelésére is képes, amelyekhez egyenként tartozik egy állapotgép, ami a hardveregységeket állítja a megfelelő State Manager (SM) modul segítségével. A busz állapotát a Network Management (NM) modulon keresztül képes kezelni.

A Network Management Interface egy adaptációs réteg az AUTOSAR Communication Manager és az AUTOSAR buszspecifikus hálózatkezelő modulok (pl. CAN Network Management és FlexRay Network Management) között.

A PDU Router a PDU-k közlekedését irányítja egy kapcsolási tábla alapján a kommunikációs stackben a modulok között. A felsőbb réteg felől érkező PDU-kat a tábla alapján továbbítja az alsóbb réteg moduljainak, kommunikációs protokollt társítva hozzájuk. Üzenetfogadáskor az alsóbb rétegekből továbbít a felsőbb modulok felé. Gateway funkcióként kommunikációs hálózatok összeköttetésére is szolgál, amelyek lehetnek azonos vagy eltérő típusúak is.

Protokollfüggő BSW modul a Transport Protocol (TP), State Manager és a Network Management. Ezekből a protokollspecifikus kommunikációs modulokból minden egyes kommunikációs hálózattípusra jut külön-külön[1]. A kommunikációs stacken belüli útvonalakat a 6. ábra mutatja be.



6.ábra: Kommunikációs stack

4. Diagnosztikai modulok és működésük

4.1. Protokollok, funkciók és szolgáltatások

A rendszerszolgáltatások olyan modulok és funkciók csoportja, amiket az összes réteg modulja használ. A diagnosztika egy olyan szolgáltatás, amely során megvizsgáljuk a hibás állapotok természetét és környezetét, majd ezekből következtetéseket vonunk le, amelyek alapján meghozhatjuk a szükséges döntéseket. Egy autóban több mint száz ECU található, és ez a szám, valamint a szoftverek összetettsége folyamatosan növekszik. Tipikusan a szerelők azok, akik az elkészült gépjármű élettartama során interakcióba lépnek az ECU-kkal. Az autóipari kommunikáció lehetővé teszi az összes releváns ECU elérését központi helyről. A vezérlők megfigyelhetőségét, valamint a működésükbe való beavatkozás lehetőségét biztosítani kell egy esetleges szoftverfrissítés vagy meghibásodás esetén.

Az előzőekben említett kommunikációs protokollok bevált működést biztosítanak a diagnosztikai kommunikációra is. Protokollokat használunk, amelyek kiszolgálják a tipikus igényeket, viszont ezek nem korlátozódnak a naplózott hibák és a körülmények felderítésére. Általános céllal felhasználhatók amikor ECU és felhasználó közötti adatcserére van szükség.

A DTC-k (Diagnostic Trouble Code) hibákról szóló bejegyzések, amelyek a nemfelejtő memóriában kerülnek eltárolásra az ECU-k által. Az OBD (On-board Diagnostic) és az UDS (Unified Diagnostic Services) protokollok a legelterjedtebbek az autóiparban. Mind a kettő szabványcsalád a diagnosztikán kívüli funkciókat is definiál.

Az OBD protokoll egy fedélzeti diagnosztika, amely esetén a jármű és a felhasználó kliens-szerver kapcsolatban állnak. A gépjárműben a beágyazott ECU-k rendszere testesíti meg a szervert, és az OBD csatlakozón keresztül kapcsolódó eszközzel a felhasználó a kliens. A kommunikáció kérés-válasz alapú, azaz a felhasználó kér, és a szerver pozitív vagy negatív választ küld. Az ISO szabványban egységesítették a hibakódok és az üzenetek formátumát, valamint egységes csatlakozó jelent meg.

Az ISO 14229 szabvány szerint az UDS diagnosztikai szolgáltatás, amit diagnosztikai rendszerekben használnak a járművek vezérlőegységeivel (ECU) történő kommunikációra[5]. A diagnosztikai szolgáltatások bővültek, és már nemcsak a károsanyagkibocsátás kapcsán érintett ECU-k szabványos elérésére van lehetőség, így a diagnosztikai szolgáltatások a járművek szervízelésének szerves részét alkotják. A protokoll tulajdonságai az OBD szabványhoz képest rugalmasabbak. A konfiguráció széleskörű és testreszabható szolgáltatásokat kínál, magasszintű funkcionalitás jellemzi és a részletek implementációspecifikusak. Az azonosítók gyártófüggőek és az adatok leírása nem fix formátumban megadott. Így, mivel nincs egységesített formátum, a jármű és a tesztelő oldalon is szükség van az azonosítók és adatleírók adatbázisára. A hibanaplózás során a hibakódok (DTC) kezelése már kiegészül a státuszinformációval is. Tehát az UDS diagnosztikai protokoll általános célú, amely széleskörűen konfigurálható és az igényeknek megfelelően célra szabható.

A szerver alapvető állapotjellemzője a session (munkamenet). Amikor egy session aktív, a diagnosztikai szolgáltatások egy részhalmaza érhető el. A szolgáltatást és a hozzá tartozó sessiont a konfigurációnál határozzák meg, valamint a szabvány szigorú időzítési követelményeket és állapotgépet is definiál, amely alapján a sessionök közötti váltás megtörténik.

A szolgáltatások és adatelemek védelmének érdekében információbiztonsági szinteket vezettek be, amelyekben az adott szint eléréséhez tartozó biztonsági hozzáférés (SecurityAccess) szolgáltatás szükséges. Ha a kliens szeretne hozzájutni egy adott funkcióhoz, aminek biztonsági szintjéhez autentikáció szükséges, akkor egy *seed-key* üzenetváltás történik. A szerver kiküld egy értéket, ami alapján a kliens előállít egy kulcsértéket. A kulcsérték generálása egy titkos algoritmus alapján történik. Ezután a kliens visszaküldi a szervernek, ami ellenőrzi és egyezés esetén engedélyezi a biztonsági szinthez tartozó funkciók elérését.

Az UDS protokoll esetén a szolgáltatásokat az OBD szabványhoz hasonlóan SID-ek azonosítják, amelyek az üzenetek első bájtján helyezkednek el. A második bájton az alfunkciót (*subfunction*) azonosító paraméter helyezkedik el, ami a kérések további finomítására szolgál. Amennyiben tartozik a szolgáltatáshoz subfunction, a hozzá tartozó második bájt hetedik bitje, ha 1 értékű, akkor a kérés teljesítésekor nincs szükség pozitív nyugtázásra. A Diagnosztikai és kommunikációs menedzsment egy subfunctionnel rendelkező szolgáltatása a DiagnosticSessionControl, amellyel sessionváltást lehet megvalósítani. Az UDS által definiált munkamenetek a *DefaultSession*, *ProgrammingSession* és *ExtendedDiagnosticSession.* A *DefaultSession* az alapértelmezett munkamenet, amikor csak az alapszolgáltatások érhetők el és azok is csak korlátozottan. A *ProgrammingSession* a szoftverfrissítés megvalósításához keretet adó szint. Az utolsó szint már a kritikus adatok manipulálását is lehetővé teszi (pl.: kilométeróra-állás)[6].

4.2. Modulok

A legfontosabb diagnosztikai modulok a DCM (Diagnostic Communication Manager) és a DEM (Diagnostic Event Manager). A DCM feladata a kommunikáció és a szolgáltatások kezelése. Kiszolgálja vagy továbbítja a szabványos kéréseket az AUTOSAR diagnosztikai stackjébe. Ismeri a szabvány üzenetformátumokat, kezeli a pozitív és negatív válaszokat és a szolgáltatást megvalósító szoftverek felől érkező válaszokat is továbbítja. Amennyiben külső feldolgozás szükséges, a teljes nyers kérést továbbítja a feldolgozó felé, ami pozitív vagy negatív választ küld a DCM-nek, ahonnan tovább halad a kommunikációs stack felé. Belső feldolgozás esetén nem kizárt az interakció más modulokkal. A lényeges információkat kinyerve a kérésből, más komponensek által szolgáltatott adatokból összeállítja a pozitív választ. Abban az esetben, ha bármilyen probléma merülne fel, a válasz negatív lesz.

A Diagnostic Event Manager ISO szerinti szabványos OBD és UDS diagnosztikai hibanaplózó funkciókat lát el. A komponensek jelentik az állapotukat, ezáltal a DEM kezdeményezi a hibák naplózását, valamint reagál rájuk. Bizonyos feladatkörök betöltése

esetén a modul sürgős reakciójára lehet szükség, ekkor a funkciói biztonságkritikusak. A naplózás során információt tárol el az időről és a minőségről. A Diagnostic Event státusszal rendelkező, széleskörűen konfigurálható alapvető egység, amely a hozzá tartozó szoftverkomponensek vagy modulok tesztelésekor keletkező eredményeket tárolja. A DTC-khez egy vagy több event is (event combination) tartozhat.

Az eseményekhez tartozik egy státusz is, amely a tesztelések eredményeképp lehet bukott (*Failed)* vagy sikeres (*Passed*). A tesztelések kétféle mechanizmussal történnek, idő- vagy számlálóalapon. Az eredmény lehet: *Failed, Passed, PreFailed* vagy *PrePassed*. A *PrePassed* és *PreFailed* egy-egy teszt eredménye, amiből az adott eseményhez tartozó státusz végeredménye kialakul. Számláló alapú eredményszámítás (debouncing) esetén a konfigurált mennyiségű *PreFailed* teszteredmény után a státusz *Failed* végeredménnyel zárul. Viszont bizonyos számú *PrePassed* esetén a végeredmény *Passed* lesz. Ez azt jelenti, hogy ha meghibásodik a járművünk, és ezt az autó jelezte felénk, akkor a meghibásodás kijavítása után, ha nem töröljük az eseményhez tartozó hibakódot, akkor az előbb-utóbb automatikusan törlődni fog. Amennyiben *PreFailed* vagy *PrePassed* eredmény érkezik és bizonyos ideig nem érkezik más, ebben az esetben a végeredmény megegyezik a kapott „*Pre*” értékkel. A számláló és időzítő alapú *debouncing* eseményenként konfigurálható.

A DEM tárolja a DTC-ket és a hozzájuk tartozó információkat, viszont a kevés hely miatt a modul a bejegyzésekkel kapcsolatban finoman hangolható tevékenységet folytat. Az elavult információkat eltávolítja, valamint a bejegyzések prioritás alapján kapnak helyet a memóriában, így takarékoskodva a tárterülettel.

A műszerfalon található indikátor lámpákat nem közvetlenül kezeli a modul. A DEM absztrakt indikátor entitásokat kezel a küszöbértékek és státuszinformációk alapján. A figyelmeztető lámpákat (*warning lamp*) illesztő (*driver*) modulok ezektől az entitásoktól kérhetik le az entitások állapotát. A hibák elévülésével az indikátorok állapota is megváltozik[6].

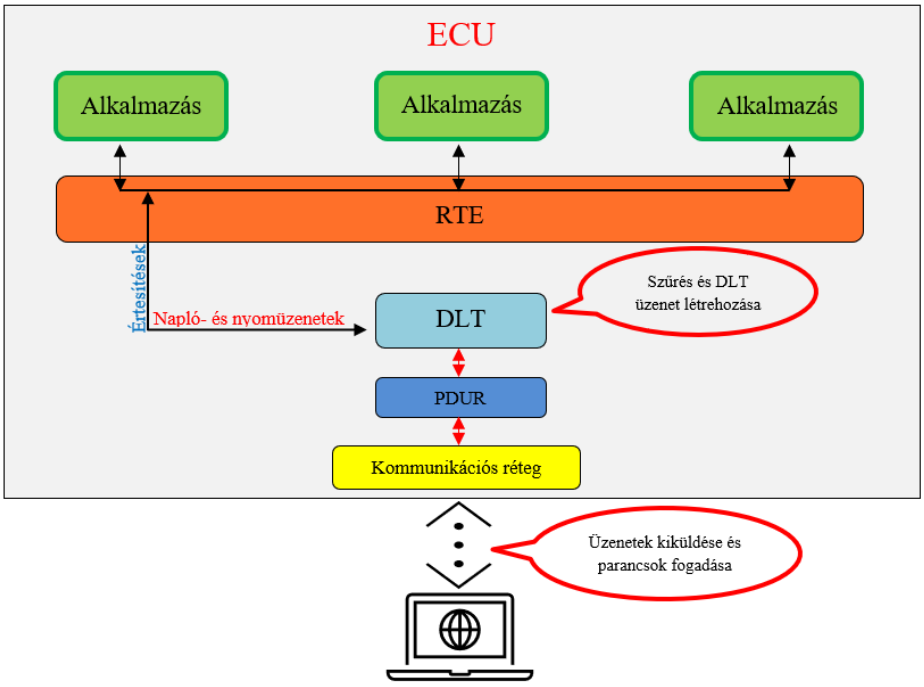
5. A DLT Modul

5.1. A modul funkcionalitása

A Diagnostic Log(napló) And Trace(nyom) (DLT) modul, a többi diagnosztikai modulhoz hasonlóan a BSW stackben, a Szolgáltatási rétegben helyezkedik el. Kapcsolatban áll az RTE-vel, ami segítségével napló és nyom üzeneteket generálhatnak az egyes szoftverkomponensek, valamint kapcsolatba léphetnek a DLT modullal és meghívhatják a kiküldő API-t (Application Programming Interface), ciklikusan. A napló és nyom üzenetek minden adatot tartalmaznak az események naplózására és nyomon követésére[1].

A DLT modul összegyűjti a naplóüzeneteket és átalakítja azokat szabványosított formátumba. Továbbítja az adatokat a PDUR-nek, amely elküldi a konfigurált kommunikációs busznak, végül az üzenet eljut a külső felhasználóhoz (7. ábra). Egy üzenet a fejlécből és a hasznos adatrészből áll. Tartalmaz minden adatot és opciót, ami az esemény leírásához szükséges egy szoftveren belül. Továbbá a modul külön API-t biztosít a DEM és DET moduloktól történő üzenetek fogadására. Képes nyomon követni az RTE tevékenységét, valamint engedélyezheti vagy tilthatja az egyes üzenetek áthaladását. Ezek mellett az üzenetek szűrésére vonatkozó naplózási szintek állíthatók. Továbbá a fejlesztési és gyártási szakaszban biztonsági mechanizmusok állnak rendelkezésre a visszaélések megelőzésére[1].

A szoftverkomponensek képesek a DLT által szolgáltatott interfészen keresztül üzenetek küldésére. A szolgáltatott API meghívásakor az adott függvény megkapja a hívó azonosítóit és a „hasznos” adatot. Egy üzenettel három különböző azonosító érkezik. Az alkalmazásazonosító (*ApplicationId (AppId)*) a küldő nevének rövidítését tartalmazza,



7.ábra: A DLT kapcsolata az entitásokkal és a külső felhasználóval

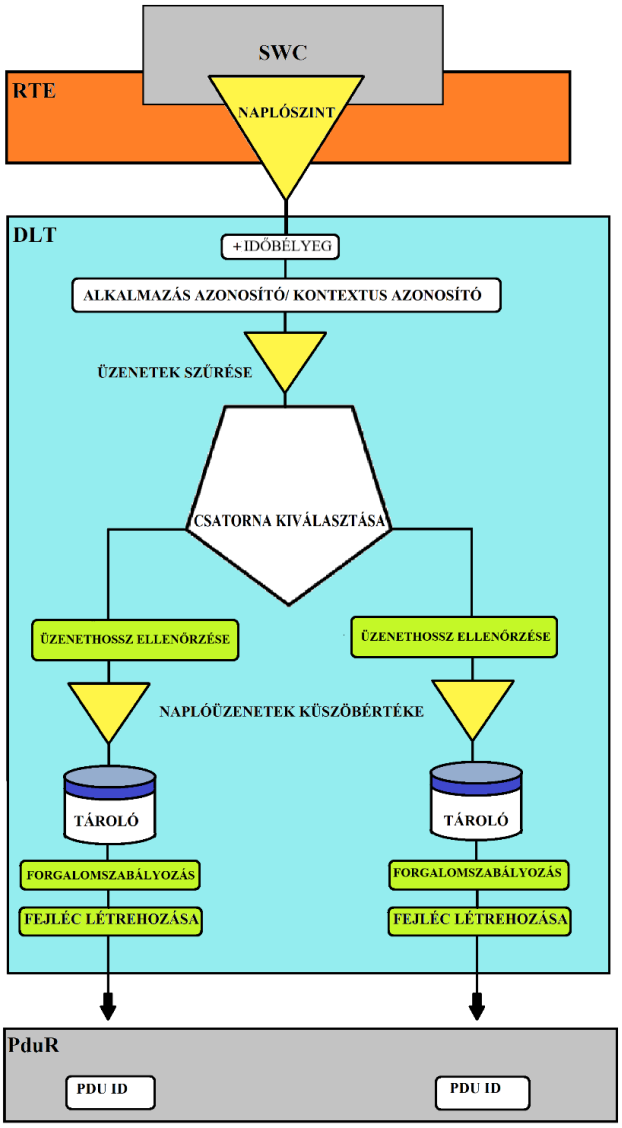
amely azonosítja az alkalmazást vagy a komponenst. A kontextusazonosítót (*ContextId)* a felhasználó definiálja. A napló- és nyomkövető üzenetek (Log és Trace) azonosítására alkalmas. Minden *AppId*-hoz több egyedi *ContextId* is tartozhat. Az üzenetek azonosítását ennek a két azonosítónak a párosa adja. Az egyes szoftverkomponensekből több példány is létezhet, amik azonos *AppId/ContextId* párost is használhatnak, ezért a DLT egy plusz paraméterrel, a *SessionId*-val különbözteti meg a beérkezett üzeneteket a komponensek különböző példányaitól.

A modul konfigurációja során regisztrálhatóak a komponensek, a hozzájuk tartozó azonosítóval. Amennyiben egy modul igényli, értesülhet a naplószint és nyomkövetési állapot változásáról, egy erre a célra fenntartott kimeneti porton.

Amikor beérkezik egy üzenet, elsőként megkapja a hozzátartozó időbélyeget, ha ezt a konfigurációban beállították. Ezután az azonosítókhoz tartozó szűrési feltételek alapján az egyes üzeneteket a modul tovább engedi vagy eldobja. A naplóüzenetek, az RTE által definiált, hétféle besorolási szinttel rendelkezhetnek, a nyomkövetés pedig be- és kikapcsolható. A megfelelt üzeneteket, továbbra is az *AppId/ContextId* páros alapján, a hozzárendelt csatornába helyezheti a modul, ha az adott csatorna (*LogChannel*) befogadja. Az egyes tárolók a hozzájuk beállított naplószinthez és üzenethosszhoz mérten fogadhatják el a beérkezett üzeneteket.

Az üzenetek kiküldés előtt szabványos formát öltenek, miszerint egy üzenet egy fejlécből és a hasznos adatból áll. Egy kiküldési esemény alkalmával kiküldhetjük egyszerre az összeset, vagy a csatornák előre meghatározott paraméterei szerint egy bizonyos mennyiséget. A kész üzenetet átadja a modul a PDUR-nak, ami továbbítja azt a felhasználó számára (8.ábra)[8].

A félév során a DLT modul 4.3.0 verziója alapján dolgoztam, azaz a követelmények és a modul funkcionalitásának megvalósítása a 4.3.0 alapján épül fel.



8.ábra: Üzenetek feldolgozása

5.2. Követelmények és osztályozásuk

Az AUTOSAR BSW modulok funkcionalitását a szabvány határozza meg, így a legtöbb követelmény elég jól meghatározott formátumban érhető el az AUTOSAR konzorcium honlapján. Ennek a megközelítésnek a következtében az alapszoftverhalmazzal kapcsolatos követelménykezelési stratégia az alábbiak szerint foglalható össze.

Az egyes követelmények eredetét a nyomon követhetőség javítása érdekében kezelni és tárolni kell. Alapértelmezés szerint minden szabványos dokumentumban meghatározott követelményt be kell vezetni és tesztelni kell.

Mivel lehetnek ellentmondások, gyengén definiált követelmények és az alkalmazási terület szempontjából irrelevánsok is, ezért minden követelménynél világosan meg

kell jelölni, hogy elfogadjuk-e vagy elutasítjuk. Az alapértelmezett döntés az elfogadás, minden más döntést megfelelő indoklással kell alátámasztani.

Mivel egy adott elfogadott követelmény túl általános lehet ahhoz, hogy közvetlenül végrehajtható vagy tesztelhető legyen, minden egyes követelménynél világosan meg kell jelölni, hogy megvalósítható-e vagy sem. Ugyanígy, szükséges megállapítani a tesztelhetőséget (tesztelhető-e vagy sem). Az alapértelmezett döntés szerint, végrehajtható és tesztelhető az adott követelmény, minden más döntést megfelelő indoklással kell alátámasztani.

A cégen belül létrehoztak egy XML (eXtensible Markup Language) séma-definíciót a követelményeket tartalmazó reqlist.xml fájlok formátumának egyesítésére. A séma tartalmazza a felhasználható elemek típusát, bizonyos típusú elemek attribútumait és az XML forma struktúráját. Például minden követelménye a BSW modulnak, a leszármazottja a követelménylista elemeinek. Az Eclipse fejlesztőkörnyezet lehetőséget biztosít az XML file szerkesztésére és vizsgálatára. Felismeri és különböző szintaxis kiemeléseket és vizsgálatokat végez. Különböző nézetekben megjeleníti és segít kiválasztani az adott attribútumok új értékét változtatás esetén.

5.2.1. Követelmények osztályozása

Minden egyes követelményhez a BSW szabványból tartozik egy req példány a reqlist.xml-ben. Elsőként létre kellett hoznom egy reqlist.xml fájlt és a szabványban megtalálható követelményeket a dokumentum szerinti sorrendben be kellett illesztenem. A DLT modul 4.3.0 szabvány összesen 202 követelménnyel rendelkezik. Első lépésben a szöveges fájlnak a formai követelményeknek kellett megfelelnie. Az egy sorban szereplő karakterek száma limitálva van és az egyes követelménytáblázatok alakja és kitöltési formája is szabályozott. Minden elem rendelkezik egy eredeti azonosítóval (alapértelmezett), amely a megvalósítás alapjául szolgál, az AUTOSAR SWS-ben meghatározottak szerint. Miután a formai követelményeknek megfelelt a szöveges fájl, a klasszifikálás következett.

A követelmények osztályozása során az egyes elemek attribútumait racionális alapon kellett kitöltenem. Az „elfogadottság” jelzi, hogy elfogadták-e az adott követelményt. Alapállapotban „nyitott”, azaz a követelményt még nem sorolták be. „Elfogadott” állapot esetén a követelményt módosítás nélkül elfogadták az AUTOSAR BSW-ben megtalálható definíció szerint. Az elfogadottság továbbá történhet a követelmény eredeti szövegének módosításával, ekkor „elfogadott módosítással”. „Visszautasított” követelmény esetén az implementálhatóságot és a tesztelhetőségek egyaránt el kell utasítani.

A *sourceRef* attribútum határozottan azonosítja a standard verzióját. A megvalósíthatósági attribútum jelzi a megvalósíthatóságot. Igaz vagy hamis értékeket vehet fel. Alapértelmezetten igaz értéket vesz fel, ekkor a követelményt elfogadják és olyan statikus, strukturális vagy viselkedési jellemzőre vonatkozik, amelyet a modul meg tud és meg kell valósítania. Bármilyen más esetben az érték hamis lesz. Ilyen eset, ha a követelmény túl általános, hogy leképezhető legyen a forráskód egy adott pontjára, vagy nem

vonatkozik a modul által megvalósított szolgáltatásra. Abban az esetben, ha egy követelmény megvalósíthatónak minősül, akkor legalább egyszer hivatkozni kell rá „@reqimpl{<req_id>}” címkével a forráskód komment mezőjében, célszerűen a megvalósítás közelében.

A tesztelhetőség hasonlóan igaz vagy hamis értéket vehet fel. Alapértelmezetten ugyancsak igaz értéket vesz fel, ekkor a követelmény, vagy annak hatása, amelyre a követelmény vonatkozik, az alkalmazott tesztelési módszerek segítségével megfigyelhető (feketedoboz teszt). Bármely más esetben, például a modul statikus vagy strukturális jellemzőire vonatkozó követelmények estén, hamis értéket vesz fel.

Az adott követelmény elem megjegyzést tartalmazhat, ami a követelmény szövegét vagy az el nem fogadott, nem megvalósítható vagy nem tesztelhető minősítés okát magyarázza, vagy pontosítja. Viszont nem terjesztheti ki az elfogadott követelmény funkcionalitását. Amennyiben szükség van módosításra, akkor a követelmény szövegét kell megváltoztatni, és modifikációval elfogadni az adott követelményt.

A BSW modulok API függvényeinek definíciói többnyire táblákba vannak rendezve, amelyek tartalmazzák a függvények nevét, szintaxisát, paramétereit stb. Ezeknek a tábláknak ugyancsak van követelményazonosítójuk, amelyeket szintén osztályozni kell. Nyilvánvaló, hogy ezeket az API függvényeket, ha elfogadják, akkor megvalósíthatóként is elfogadják, viszont a tesztelhetősége nem egyértelmű. Mert ha az adott API valamilyen belső feladatot lát el, ami nem vizsgálható blackbox (feketetoboz) teszttel, akkor kívülről nem tudunk hozzá tesztesetet felállítani, ami igazolná a helyes működést.

5.2.2. DLT modul követelményei és osztályozásuk

Ahhoz, hogy a követelmények osztályozását megkezdhessem, elsőként a modul működését kellett megismernem. A modul implementálása és a követelmények megvalósítása közben is változtattam az előzetes besorolásokon. A következő néhány klasszifikációs példán keresztül szeretném bemutatni a követelmények elfogadásának vagy elutasításának néhány indokát.

```
<req acceptance="accepted" id="SWS_Dlt_00377" implementable="true" testable="true" sourceRef="ArDlt">
    <text>
        The ApplicationID, ContextId and Message ID of a Log Message sent for a DEM event shall have the
        following values:
        ApplicationID = "DEM"
        ContextId = "STD0"
        MessageID = 0x00000001
    </text>
</req>
```

9.ábra: Elfogadott, implementálható és tesztelhető követelmény

Az *SWS_Dlt_00377* követelmény kimondja, hogy az *ApplicationID*, a *ContextId* és a *MessageId* a DEM eseményhez elküldött naplóüzenet esetén a megadott értékeknek kell lenni (9.ábra). A követelményt elfogadtam, valamint implementálhatónak és tesztelhetőnek jelöltem, mert az értékadás a forráskódhoz egyértelműen hozzárendelhető és az egyes naplóüzenetek „kívülről” vizsgálhatók, ezáltal a beállított paraméterek értéke is.

```
<req acceptance="accepted" id="SWS_Dlt_00648" implementable="true" testable="false" sourceRef="ArDlt">
    <text>
        When the Dlt_Init is called, the optional timer DltGeneralStartUpDelayTimer shall be started if configured.
    </text>
</req>
```

10.ábra: Elfogadott, implementálható, de nem tesztelhető követelmény

Az *SWS_Dlt_00648* követelmény szerint a Dlt_Init API hívását követően, ha konfigurálva van, az opcionális *DltGeneralStartUpTimer* időzítőt el kell indítani (10.ábra). A követelményt elfogadtam és implementálhatónak minősítettem, viszont a tesztelhetőséget elutasítottam, mivel ez egy belső viselkedésre vonatkozik és az értéke nem vizsgálható kívülről.

```
<req acceptance="accepted" id="SWS_Dlt_00650" implementable="false" testable="false" sourceRef="ArDlt">
    <text>
        The following steps describe the logical order, in the context of calls to
        Dlt_SendLogMessage or Dlt_SendTraceMessage:
          1. Generate timestamp (see 7.1.9.1)
          2. Filter message (see 7.1.9.2)
          3. Select target LogChannel(s) (see 7.1.9.3)
          4. Check Message length (see 7.1.9.4)
          5. Apply the current LogChannel threshold (see 7.1.9.5)
          6. Copy Dlt message to LogChannel specific buffer (see 7.1.9.6)
    </text>
    <note>
    This requirement is too general to be implemented or tested directly thus classified as non-functional
    </note>
</req>
```

11.ábra: Elfogadott, de nem implementálható és nem tesztelhető követelmény

Az *SWS_DLT_00650* követelmény leírja a logikai lépések egymásutánját a *DLT_SendLogMessage* és *DLT_SendTraceMessage* API-k kontextusában (11.ábra). A követelményt elfogadtam, viszont túl általános ahhoz, hogy a kód egy adott pontjához hozzá lehessen rendelni., vagy közvetlenül tesztelhető legyen, így a másik két attribútum hamis értéket kapott.

```
<req acceptance="rejected" id="SWS_Dlt_00653" implementable="false" testable="false" sourceRef="ArDlt">
    <text>
        If the parameter DltHeaderUseTimestamp is set to TRUE,
        but the Dlt module cannot fetch a timestamp for any reason,
        the timestamp shall be set to 0x00000000.
    </text>
    <note>
        The timestamp is always available. There is no option for the GPT module not to provide data,
        so we do not need to be prepared for such a situation
    </note>
</req>
```

12.ábra: Visszautasított, nem implementálható és nem tesztelhető követelmény

Az *SWS_DLT_00653* szerint a *DLTHeaderUseTimestamp* paraméter igaz értéke esetén, ha a DLT modul nem tudja lekérni az időbélyeget, akkor a bélyeget 0 értékűre kell állítani (12.ábra). Az időbélyeg mindig rendelkezésre áll, nincs olyan szituáció, hogy az adott Gpt_GetTimeElapsed függvény ne adjon adatot, ezért nem kell készülni ilyen szituációra. Ebből kifolyólag a követelményt elutasítottam.

5.3. Dinamikusan generálható kód és adatstruktúrák

A modul dinamikusan generálható részei a konfiguráció függvényében változnak. A DLT modulhoz, az AUTOSAR konzorcium által kiállított konfigurációs paraméterlista alapján, egy a thyssenkrupp által fejlesztett modellező programban beállíthattam a kívánt értékeket. Megismertem a Java programozási nyelvet és felhasználásával írtam egy programot, ami a belsős modellező eszközzel együttműködve képes az elő- és utófordítási definíciók és adatstruktúrák generálására C nyelven.

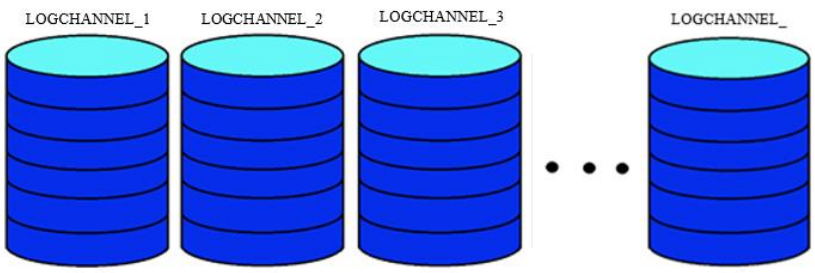
A cég számos könyvtárat írt a generálás elősegítésére, viszont nem minden adattípusra vonatkozóan. Ezért nemcsak a Java nyelv sajátosságai, de a könyvtárak felépítésének és működésének megismerésére is szükség volt, hogy az általam létrehozott Java generáló függvények a környezetbe ágyazva hibátlanul tudjanak működni.

A DLT specifikációjának a 10. fejezetében megtalálhatóak a konfigurálható paraméterek csoportosítva, konténerekben. Elsőként ezeknek a konténereknek megfelelően létrehoztam egy-egy publikus osztályt, amelyek függvényei a felhasznált modellező programban beállítható konfigurációs értékeket lekérik és szolgáltatják. A generált kódban gyakorlatilag egymásba ágyazott adatstruktúrákat példányosítok és megfelelő nagyságú adattárolókat hozok létre.

Elsőként bemutatom a generált pufferek és adatstruktúrák funkcionalitását, majd osztálydiagrammal szemléltetem a konfigurációs adatok modelljét. A modul képes informálni az egyes komponenseket a naplószintváltozásról, ezért a komponenseket regisztrálni kell a modulban az *AppId/ContextId* páros alapján. Minden egyes ilyen párost el kell tudni tárolni, konfigurációs és futási időben egyaránt.

Az üzenetek kiküldése előtt össze kell állítani az adott üzenethez tartozó fejlécet, ami tartalmazza a szükséges metaadatokat. Viszont az, hogy mit tartalmaz, konfigurációfüggő. Ezért minden konfigurációsfeltételhez egy logikai értéket társítok, amiket a statikus kódból könnyen elérek és a feltételektől függően az üzenetek fejlécében elhelyezem a szükséges adatokat. A fejlécben található egy bitmező, ami az üzenet tartalmát azonosítja a konfigurációban megadott feltételektől függően. Ezt a bitmezőt a konfigurációval létrehozom, ezzel is csökkentve a későbbi erőforrásigényt.

A modul specifikációja szerint több, egymástól független, a konfigurációban megadott mennyiségű csatornát kell fenntartani, amelyekben a napló- és nyomüzeneteket eltárolom azok kiküldéséig. Ezek a csatornák az úgynevezett *naplócsatornák* (*Logchannel)(13.ábra).*

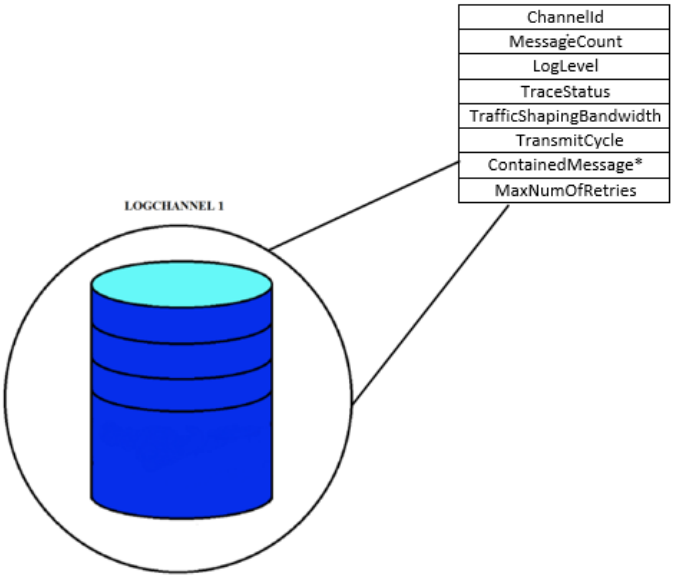


13.ábra: Csatornák

A csatornákat egyenként, az *AppId*-hoz hasonló módon, 4 ASCII karakterrel azonosítom. Mindegyik mérete, azaz, hogy hány üzenet kaphat benne helyet legfeljebb, és hogy egy üzenet hossza mekkora lehet, konfigurációs időben eldől és futási időben már nem változtatható. Mielőtt bekerülne egy üzenet a tárolóba, meg kell felelnie a méretének és a csatorna küszöbértékénél (*LogLevelThreshold*) nem lehet nagyobb a naplószintje. Hasonlóan, a nyomüzenetek esetén egy logikai érték (*TraceStatus*) felel az üzenet elhajításáért vagy befogadásáért. A naplószint és a nyomstátusz konfigurációs időben beállítható, viszont a megfelelő parancsok meghívásával futási időben megváltoztatható az értékük.

Az üzenetek kiküldésénél hasonlóan nagy szerepet vállalnak a csatornák. Mivel beállítástól függően, ha forgalomszabályozást engedélyeztek a konfigurációban, akkor az egy csatornától egyszerre kiküldhető adatmennyiség korlátozott. Az elküldhető üzenetek száma függ a másodpercenként kiküldhető bitek számától (*TrafficShapingBandwith*), és az adatátvitel lehetséges periódusidejétől (*TransmitCycle*)(14.ábra).

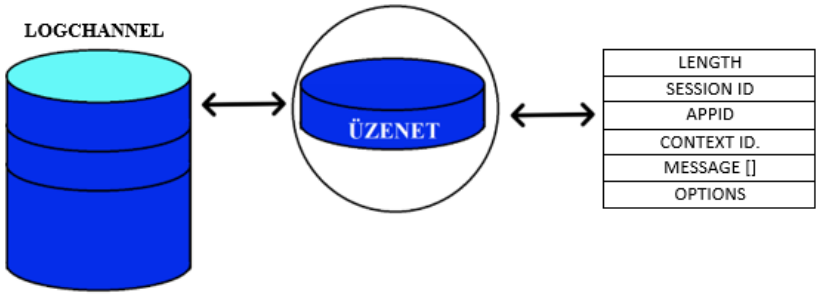
Amennyiben egy csatorna megtelik, az üzenetek fogadásakor egy túlcsordulást jelző bit bebillen, majd az üzenetek kiküldéskor felszabaduló hely következtében törlődik. Ha egy üzenetet nem sikerült elküldeni, akkor a konfigurált próbálkozási lehetőség szerint újra küldjük, ha továbbra sem sikerül, akkor eldobjuk. A limit *(MaxNumOfRetries)* csatornafüggő.



14.ábra: Egy csatorna főbb tulajdonságai

Minden egyes csatorna legfeljebb a konfigurált mennyiségű üzenetet tartalmazhatja. A csatorna egy eleme egy adatcsomagot tartalmaz, ami nemcsak a hasznos üzenetet tartalmazza, de a küldő komponens azonosítására is alkalmas *SessionId*-t és *AppId/ContextId* párost is. Továbbá a kiküldendő üzenet tartalmazza az üzenet beérkezésekor aktuális összes addig beérkezett üzenetek számát. Ezzel észlelhetővé válik a külső kezelőnek az üzenetvesztés. A tároló konfigurált méretei alapján, létrehozok bájtokat tartalmazó tömböket. Annyit hozok létre, amennyi üzenetet a csatorna eltárolhat, és olyan elemszámút, amibe a megadott maximális méretű üzenet belefér. Az egyes üzeneteket a csatornák struktúrájában elhelyezett mutatók segítségével érem el.

Eltárolom továbbá az üzenetekhez tartozó kiküldési próbálkozások számát, ameddig sikeres nem lesz a küldés, vagy el nem éri a küszöbértéket, amikor is elengedjük az üzenetet. Mivel a napló és nyomüzeneteket nem külön tárolókban helyezzük el, így az „options” változóban specifikálhatjuk az üzenetek típusát.



15.ábra: Egy üzenet tulajdonságai

A szoftverkomponensek (*Software Component - SWC*), amelyek üzentet küldhetnek, és regisztrálhatják *AppId/ContextId* párosaikat, a konfiguráció során jegyezhetők be. Egy SWC-hez konfigurálható a SessionId és az elküldött üzenetek maximális hossza, napló és nyomüzenetekre külön-külön. Valamint kiválaszthatjuk, hogy az adott SWC értesüljön-e az *AppId/ContextId*-hoz rendelt nyomstátusz vagy naplószint változásokról. A komponensek regisztrálhatnak újabb, hozzájuk tartozó *AppId/ContextId* párost és törölhetnek is meglévőket. Amikor beregisztrál egy komponens, akkor kettő legfeljebb 255 karakterhosszúságú karakterláncot is el kell tárolnunk, amik az *AppId* és a *ContextId*-hoz tartozó leírásokat tartalmazzák.

A szabvány nem nyilatkozik az egy komponenshez maximálisan felvehető azonosítópárok számáról, valamint regisztrációja megszüntetésének a menetét sem specifikálja. Csupán arról nyilatkozik, hogy a regisztráció megszüntetése után a törölt elemekhez tartozó SWC-ket a DLT modul nem próbálja meg értesíteni naplószintváltozáskor. Ezért a Java kódban beállítható paraméter szerint több helyet foglalok le, ahova a később beregisztráló elemek adatait helyezem el. Az elmentett hasznos adatok mellé elmentek egy logikai változót is, ami alapján eldöntöm, hogy „törölt” vagy még élő az adott elem. Ha élő, akkor a változásoknál értesítést küldök a hozzá tartozó komponensnek, ha nem, akkor akár más komponensek is igénybe vehetik az adott memóriaterületet és fölülírhatják azt saját adataikkal.

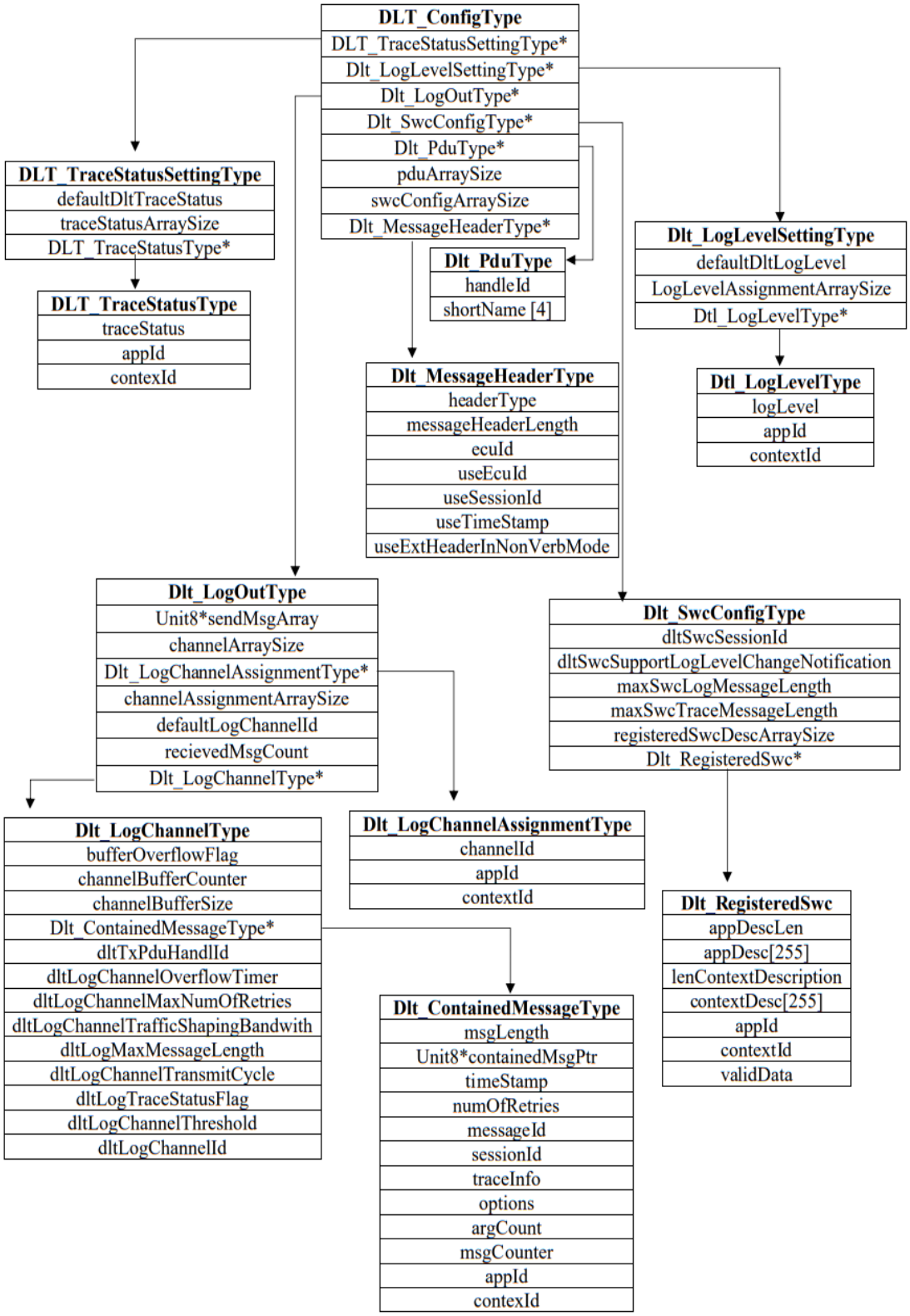
A konfiguráció során a szoftverkomponensekhez meg kell adni a maximális üzenethosszt, ami érkezhet tőlük. A kódgenerálás során ez alapján pontosan létrehozható az üzeneteknek szánt tömbök mérete csatornára szabva, így minimalizálva a lefoglalt tárterületet.

Az AUTOSAR DLT 4.3.0 verziója szerint az egyes *AppId/ContextId* azonosítópárosokhoz hozzárendelhetünk egy nyomstátuszt[8]. Amennyiben nincs hozzárendelve megfelelő, egy alapértelmezett értéket veszünk alapul az üzenetek szűrése során. Ezen felül hozzájuk rendelhetünk egy vagy több kimeneti csatornát is, avagy, ha nem rendelkezünk erről, az alapértelmezett csatornába kerülnek az üzenetek. Ezek eltárolására két elkülönített tárolót hoztam létre, amikből egyszerűen kikereshető az adott azonosítópároshoz tartozó érték.

A statikus kód írása során érezhető volt, hogy egy nyom és egy naplóüzenet kezelésének megvalósítása nagyon hasonlóan elvégezhető. Elsőként a naplóüzenetek és csak utána a nyomüzenetek szűrésével foglalkoztam. Itt derült ki számomra, hogy az általam feldolgozott szabványverzió egy ponton hibás. Az AUTOSAR konzorcium által kiadott legfrissebb DLT R20-11-es [9] verziójában már, hasonlóan a nyomüzenetekhez, külön konténer tartozik az azonosítók és naplószintek összerendeléséhez, viszont a 4.3.0 verzióban ezt még kihagyták. Így ezt a részét a kódgenerálásnak, a friss, javított szabvány alapján végeztem el, és az üzenetek szűrését is javítottam (megvalósítottam) a statikus kódban. (Valamint így értelmet nyertek a naplószintek lekérdezésére és beállítására vonatkozó API-k is.)

A konfiguráció során a *DltGeneral* konténer felsorolja az összes globális DLT funkciót, amelyek engedélyezhetők vagy tilthatók a fordítás előtti időpontban, az erőforrás-felhasználás optimalizálása érdekében. Ezeket a funkciókat kifejezetten a „Dlt_Cfg.h” fájlba kellett legenerálni, ahol definiálni kell az adott funkciót és az értékét.

A konfigurációs adatok modelljét a 16. ábrán láthatjuk. A modell tartalmazza azokat a paramétereket is, amiket külön nem emeltem ki, viszont szükséges mennyiségeket tárolnak el a modul működéséhez. Például az aktuálisan konfigurált csatornák számát, az egyes tömbök méretét stb.



16.ábra: Konfigurációs adatok modellje

5.4. API-k implementálása

Az egyes funkciók megvalósítása nem a leírásuk sorrendjében történt. Tapasztalataimnak megfelelően, a félévben elsőként a statikus fogadó függvényeket és a küldő függvényt próbáltam megvalósítani, viszont ahogy ezeket írtam, és folyamatosan újra olvastam a szabványt, kezdtek világosabbá válni a modul egyes részletei és működése. A félév során a dinamikusan generálható kód fejlesztése közben ébredtem rá sok hibámra, így a statikus kód nagy része újraírást igényelt. A dolgozatban igyekszem funkcionalitás szerint, logikusan felépítve bemutatni az egyes részek működését.

5.4.1. Szoftverkomponensek regisztrálása és paramétereik beállítása

Az információ, hogy melyik szoftverkomponens melyik *AppId /ContextId*-ért felelős, konfigurációs időben megadható, és futási időben frissíthető a *RegisterContext* és *UnregisterContext* API-k megfelelő hívásával.

A regisztráció során csak a konfigurált komponensekhez rendelhető újabb azonosító páros. Amennyiben nem létezik a még regisztrálni kívánt páros, a lefoglalt memóriát megvizsgálom szabad hely után kutatva. Ez egy logikai érték vizsgálatából áll minden elem esetén, ami megadja, hogy mely helyek felülírhatóak. A keresés során a kapott adatokat bemásolom az első szabad helyre. Az azonosító párost egy-egy 4 elemű tömbben, míg a hozzájuk tartozó leírásokat a konfiguráció során létrehozott nagyméretű (255) tömbbe másolom a paraméterként megkapott hosszuk alapján. Ezek után eltárolom a leírások hosszát is, így nem kell a kiolvasáskor végigolvasni a tömböt és csak a hasznos adatokat kapjuk vissza.

A külső felhasználó, akinek az üzeneteket is kiküldjük, futási időben lekérheti az aktuális naplószintet és a nyomkövetési státuszt minden regisztrált komponenshez. Ezen felül az alapértelmezett és az egyes csatornákhoz tartozó előzetesen konfigurált értékek szintén megváltoztathatóak. Ezekre külön API-k készültek, amik hívásakor az *AppId/ContextId* páros, vagy a csatorna egyedi azonosítója szükséges. Amikor beállítunk egy nyomstátusz vagy naplószint értéket, akkor az összes komponens azonosítóját meg kell vizsgálni és minden egyezésnél be kell állítani a kapott értéket.

Amikor egy komponens leáll, akkor köteles minden egyes hozzá tartozó *AppId/ContextId* párost törölnie. Ez a DLT-ben a már említett, regisztrált elemekhez tartozó érvényességet jelző logikai érték átállításával, erőforrás-takarékosan megoldható.

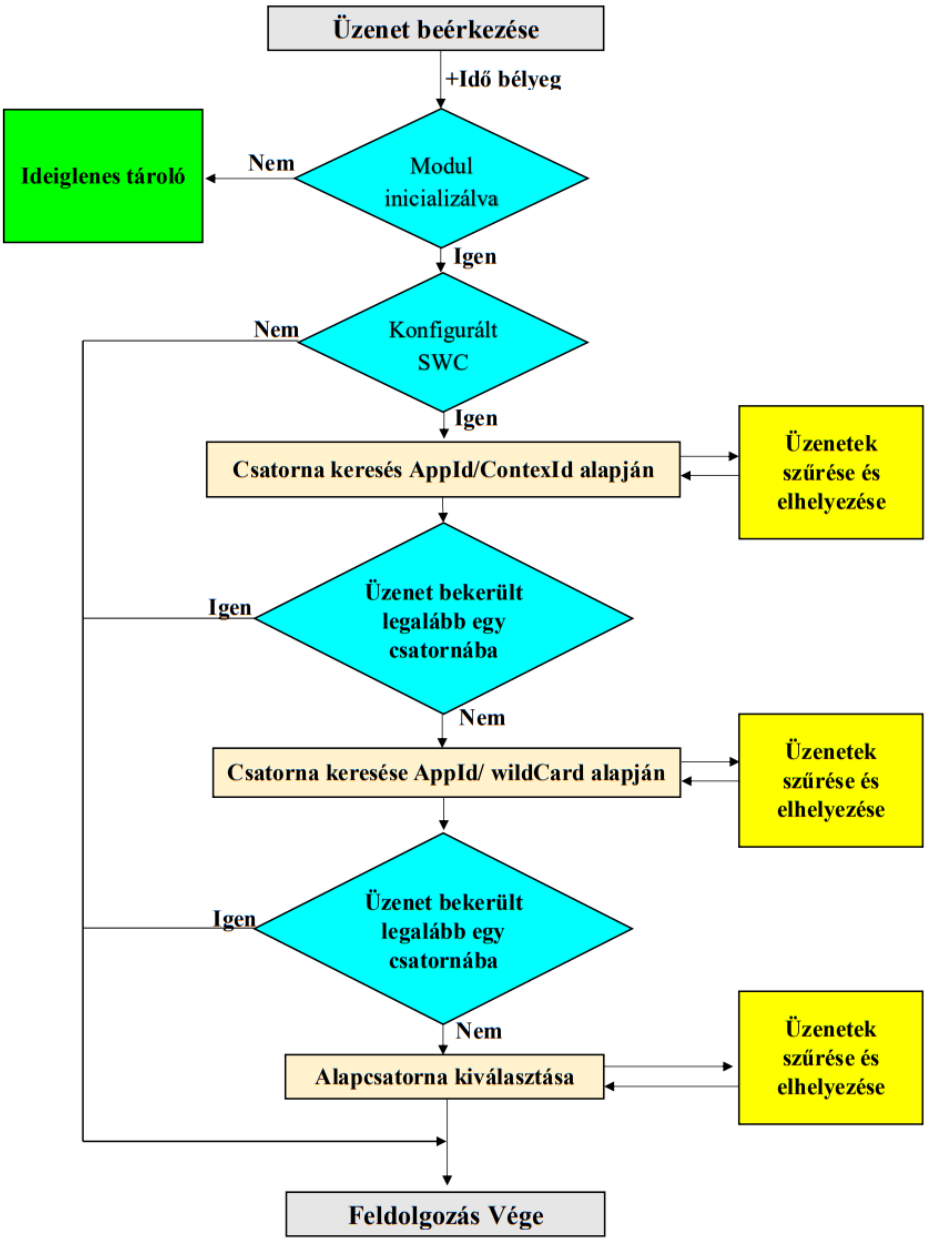
5.4.2. Üzenetek fogadása és eltárolása

Üzenetek fogadására a DLT modul négy API-t biztosít. A Dlt_DetForwardErrorTrace a DET (Default Error Tracer) modultól fogad üzeneteket. A DET egy hibakereső modul, amelyhez a fejlesztési és futásidejű hibákat jelenthetik a BSW modulok vagy a szoftverkomponensek. A DLT által biztosított függvény megfelelő hívásával, ezeket továbbíthatják a DLT modulnak. A DET függvénye, a Det_ReportError összes paramétere továbbításra kerül a DLT-be.

A DEM (*Diagnostic Event Manager*) modul belsőleg eltárolja a szoftverkomponensek és a BSW modulok által generált eseményeket. Ezeket az eseményeket eseményazonosítók jellemzik. A DEM-ben található eseményekhez további információk tartoznak. Ez az információ a diagnosztikai hibakód (DTC), a kibővített adatrekordok és a rögzített keret (*FreezeFrame*). Amikor egy esemény állapota megváltozik, a DEM meghívja a Dlt_DemTriggerOnEventStatus függvényt, amivel értesíti a DLT-t a változásról. A DLT API függvényhívása során csak az esemény azonosítója érkezik, a hibakód és a *FreezeFrame* adatok lekéréséhez a DEM által biztosított API-kat kell felhasználnom.

Az alapvető napló- és nyomüzenetek beérkezése és feldolgozása hasonló módon történik. A naplóüzenetek fogadása a Dlt_SendLogMessage függvény hívásával történik. Amennyiben ezt a konfiguráció kéri, elsőként az üzenethez szükséges időbélyeget kérjük le, amit a GPT (*General Purpose Timer*) modul szolgáltat. Ha inicializálva van a DLT modul, akkor megkezdődik az üzenettel érkező metaadatok vizsgálata. Elsőként a *SessionId*-t vizsgálom meg, ami segítségével kiderül, hogy konfigurálták-e az adott modult vagy komponenst. Ha nem, akkor nem folytatom tovább a mechanizmust és jelzem a küldőnek a „problémát” (17.ábra).

Ha sikeresen beazonosítottam a küldőt, akkor megkezdem az üzenethez tartozó csatorna keresését az *AppId/ContextId* páros alapján. Az azonosítópáros-csatorna összerendelés konfigurációs időben és futási időben is létrehozható. Egy pároshoz több csatorna is hozzárendelhető, ezért a csatornakeresést minden alkalommal végig kell futtatni az összes hozzárendelésen. Ha találok legalább egy megfelelő csatornát, ahová sikeresen el tudtam helyezni az adott üzenetet, akkor a procedúra sikeresen befejeződött. Viszont, ha nem, meg kell vizsgálni az egyezést újra, úgy, hogy a *ContextId* helyére a *wildCard*, azaz a nullaértékű *ContextId* kerül. Ha így sem sikerül egy hozzárendelt csatornát sem találni, akkor az alapértelmezett csatornába próbálom elhelyezni.



17.ábra: Üzenetek beérkezése és csatorna kiválasztása

Ha találtam egy megfelelő csatornát, az üzenetek szűrése következik. Előhívom az eltárolt *AppId/ContextId* pároshoz tartozó naplószintet, vagy ha nem találom (nincs hozzárendelés), akkor az alapértelmezett szintet használom fel. Ha az érkező üzenet naplószintértéke nagyobb, mint a letárolt küszöbérték, akkor az adott üzenetet elengedem. Ha megfelelt, akkor a választott csatorna szűrési feltételeinek is meg kell felelnie. A csatorna naplószint küszöbértékét (*LogChannelThreshold*) sem haladhatja meg az üzeneté, valamint a csatornába nem kerülhet a konfigurált méretnél hosszabb (nem férne bele). A megfelelt üzeneteket és a hozzájuk tartozó adatokat az előre lefoglalt, strukturált tárterületre helyezem el (18.ábra). A beérkezett adatokon kívül plusz információt is elmentek a későbbiekben való egyszerű működés érdekében.

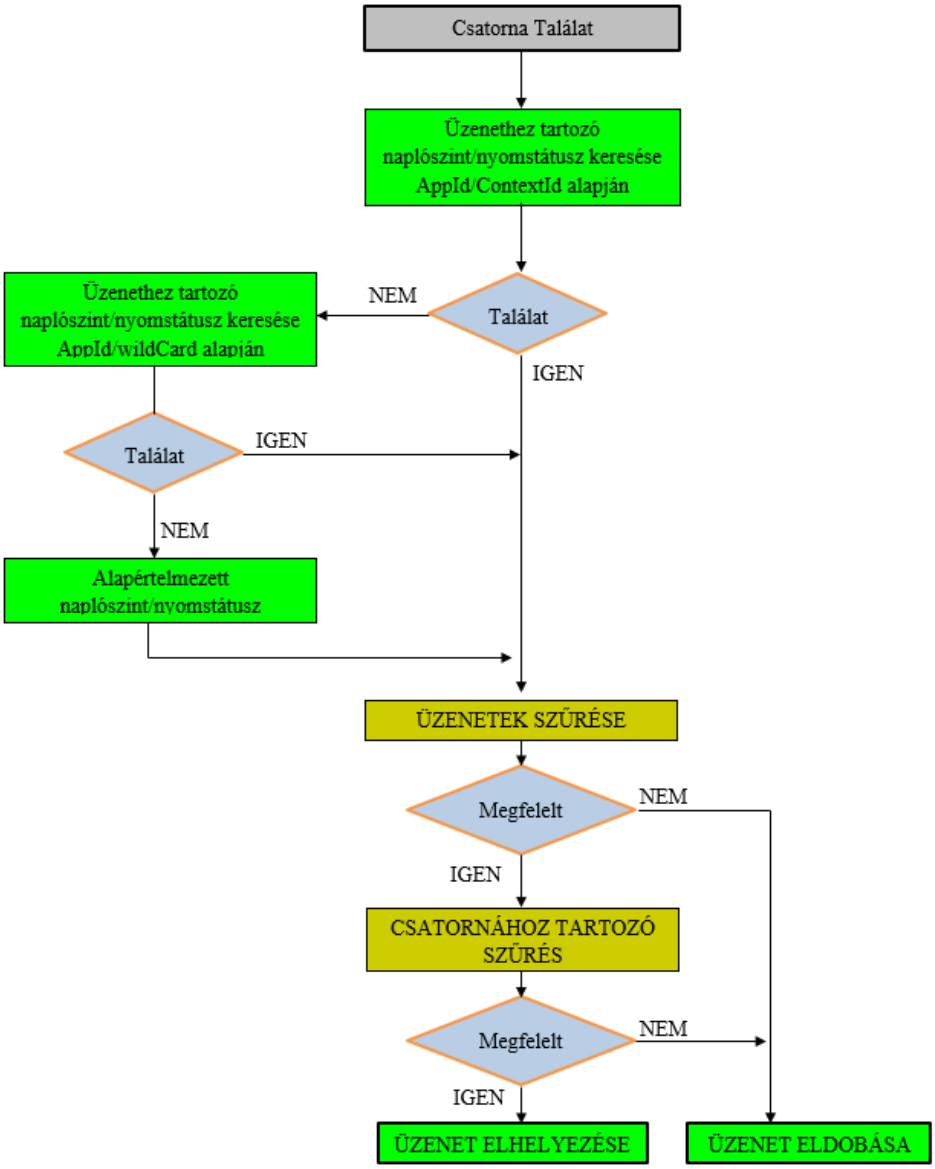
Mentésre kerül a *messageId*, ami *nonverbose* (nembeszédes) módban azonosítja az adott üzentet, valamint elmentem az aktuálisan beérkezett üzenetek számát, amit a kiküldött üzenetből később a felhasználó kiolvashat és megállapíthatja az adatvesztést, ha keletkezik. Sikeresnek mondható az üzenet fogadása, amennyiben legalább egy csatornában el tudtuk helyezni.

A Dlt_SendTraceMessage API, a nyomüzenetek feldolgozásáért felel. A naplóüzenetek folyamatához hasonló működéssel dolgozom fel az adatokat. A megfelelő csatorna keresésével kezdek, és találatkor megszűröm az *AppId/ContextId*-hoz tartozó nyomstátusz alapján. Ez egy logikai érték, így, ha az igaz, akkor feldolgozzuk az üzenetet, egyébként nem. Hasonlóan a naplószintküszöbértékhez, az adott csatorna konfigurációja szerint fogadhatja a nyom üzeneteket vagy elutasíthatja azokat.

A csatorna kiválasztás után, a szűrési és elhelyezési feladatokat külön függvénybe gyűjtöttem, így csökkentve a kódsűrűséget a nyom- és naplóüzeneteknél egyaránt.

A DET modul esetén a szolgáltatott API nem tartalmazza az *AppId/ContextId* párost, mivel a modult maga a függvényhívás azonosítja. A DET-től kapott Error Trace üzenetet naplóüzenetként mentjük és továbbítjuk. A további azonosításra viszont szükség van *App*- és *ContextId*-ra, így a szabvány által meghatározott módon, *AppId*-ként megkapja a „DET” azonosítót, és *ContextId*-ként az „STD0” -t. Naplószint értékként „ERROR” -t kap és a hozzárendelt *messageId* értéke kettő lesz. A DLT által biztosított függvényben a hívótól megkapjuk a hívóazonosítót, a példányazonosítót, az alkalmazásazonosítót és a hibaazonosítót. Ezek lesznek a későbbi, összeállított üzenet hasznos részei. Miután szabványos formára hoztam az üzenetet, a megfelelő csatorna kiválasztása és az üzenet szűrése következik, ami megegyezik a korábban leírt naplóüzenet feldolgozásával.

A DEM esetén, bejövő paraméterként az eseményazonosítót kapjuk meg. A DET-hez tartozó API-tól eltérően, itt a szabvány megköveteli a Dlt_SendLogMessage API felhasználását az üzenet további feldolgozására. Itt elsősorban a szabványos forma összeállítása volt a feladat. Hasonlóan a DET-től kapott üzenetekhez, itt is a szabvány által rögzített *AppId/ContextId* értékeket kellett felvenni. Az eseményazonosítóhoz tartozó DTC és *FreezeFrame* a DEM által biztosított API-k segítségével lekérdezhetőek. Mivel a lekért adatok mérete nem rögzített, csupán a visszakapott adat formátuma a "*DEM_DTC_FORMAT_UDS*" által, ezért a hasznos adatoknak lefoglalt tömb méretét a megkapható legnagyobb érvényes adatméretnek választottam. Ez nem erőforrás takarékos, viszont a hibátlan működéshez nélkülözhetetlen. Az üzenet összeállítása után a Dlt_SendLogMessage-et meghívva a modul feldolgozza az üzenetet.



18.ábra: Üzenetek szűrése és elhelyezése

A modulnak lehet olyan állapota, amikor még nincs inicializálva, viszont a többi modul és komponens már üzenetet szeretne küldeni. Ezért létrehoztam a statikus kódban egy struktúrát és a struktúrával egy-egy tömböt a nyom- és naplóüzenetek számára, ami képes átmenetileg fogadni és eltárolni a beérkező üzeneteket. Minden üzenet beérkezésekor megvizsgálom, hogy megtörtént-e az inicializálás. Ha még nem történt meg, akkor eltárolom az üzenetet, amíg van rá kapacitás. Ha megtörtént, az inicializálás fő lépései után, az eltárolt üzenetek számához mérten egy-egy ciklusban meghívom a Dlt_SendTraceMessage és a Dlt_SendLogMessage függvényekben is alkalmazott csatorna kiválasztó és szűrő belső függvényeket, amik a szűrési feltételeknek megfelelően megpróbálják feldolgozni az ideiglenesen eltárolt üzeneteket.

Az Függelékben megtalálható 21. ábra segít megérteni az üzenetek eltárolásának folyamatát.

5.4.3. Üzenetek kiküldése

A DLT üzenetek kiküldése a vevő függvényektől elkülönítve történik. Minden összegyűjtött üzenetet egy helyről, a Dlt_TxFunction törzséből küldök ki. A kiküldésnek kétféle módja lehetséges. Kiküldhetjük az összes üzenetet egy függvényhívás alkalmával, vagy korlátozott mennyiségben, a csatornákban megadott paraméterek alapján. A konfigurációnál meghatározott *DltGeneralTrafficShapingSupport* határozza meg a módot.

Amennyiben ki van kapcsolva a forgalomszabályozás, az üzeneteket egy függvényhívás alkalmával kell kiküldeni. Minden konfigurált csatorna összes üzenetén végig kell iterálni. Elsőként össze kell állítani az adott üzenet fejlécét. Minden fejléc egy állandó és egy változó részből áll. Az állandó rész hat különböző értéket tárolhat és legfeljebb tizenhat bájtból áll. Az állandó részben biztosan helyet kap a DLT üzenetről általános információkat hordozó egy bájtos bitmező. Ez tartalmazza az opcionális részek jelenlétét, így a külső fogadó fél megfelelően tudja értelmezni a kapott üzenetet[7].

Mivel az itt megtalálható bitek értéke, egyet kivéve, konfigurációfüggő, ezért ezt a részt már ott legeneráltam. Így erőforrást megtakarítva, nem kell minden egyes üzenet alkalmával újra és újra létrehozni ezt a fejléctípust. A létrehozása a Java kódban egyszerű matematikai műveletek segítségével történik. Ha egy feltételhez igaz logikai érték társul a konfigurálás során, akkor a hozzá tartozó bit helyiértékének megfelelő számot adok hozzá. Így nincs szükség a bitek erőforrásigényes tologatására. A modul inicializálásakor ezt az értéket kimentem egy statikus globális változóba. A kivételes bit, amiről nem rendelkeztem a kódgenerálás során, az a platformtól függő, bájtsorrend megadása (little-vagy bigendian). Az inicializálás során, a megváltoztatandó bit helyi értékének megfelelő értékkel növelem a változóban tárolt értéket, az adott felület függvényében.

Az üzenetek kiküldése a *PduR_DltTransmit* függvény segítségével történik. Ezt a PDUR modul szolgáltatja, ami az üzeneteket eljuttatja a megfelelő kommunikációs buszra. A függvény két paramétert vár. A *PduId* előre konfigurált paramétert, és a kiküldendő struktúrára vonatkozó pointert (*PduInfoType**). Meg kell adni az adott üzenet teljes hosszát, ami már tartalmazza a fejlécet is. A fejléc hosszát a konfigurációs beállításokból következtethetjük, ezért ezt is elvégeztem a dinamikus kód generálása során, így már csak össze kell adni az adott üzenet hosszával. A mutatott struktúra másik eleme a teljes üzenetet tartalmazza. A csatornákban eltárolt üzenetekhez hasonló módon, a dinamikus kód generálásakor létrehozok a DLT modullal kapcsolatban álló konfigurált komponenseknél megadott legnagyobb fogadható üzenet méret és a fejléchossz alapján, egy tömböt, amibe a kiküldés során bemásolom a teljes kiküldendő üzenetet. Így a pointer gyakorlatilag mindig ugyanarra a címre mutat, csak a tartalom változik.

Mivel az üzenetek beérkezésekor minden szükséges értéket eltároltunk, így most már csak ki kell olvasni és el kell helyezni őket a kiküldésre lefoglalt tárterületen. Sorrendben haladva, az első helyekre elhelyezzük a fejléc típusát, majd az üzenetszámlálót, ami az üzenet beérkezésekor eltárolt összes üzenet számát adja, majd a teljes üzenet hosszát. A fejléc állandó részébe opcionálisan belekerülhet az ECU ID, ami a küldő ECU-t

azonosítja, valamint a *SessionId* és az időbélyeg. Ezen felül, ha a *DltUseExtHeaderInNonVerbMode* konfigurációs feltétel be van kapcsolva, akkor a kiterjesztett fejlécet is alkalmazni kell, ami további tíz bájtnyi információt tartalmaz.

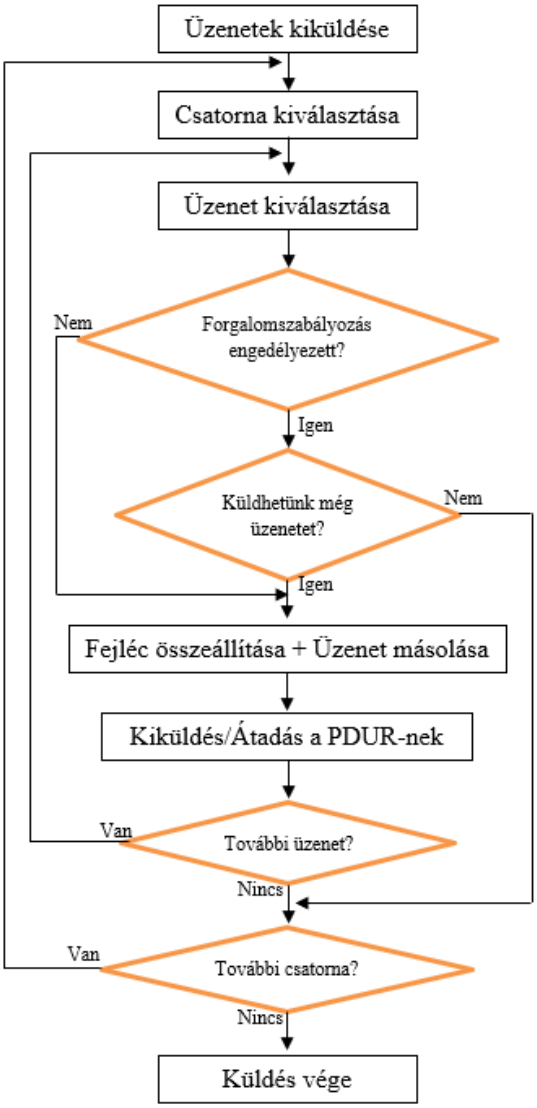
A kiterjesztett fejlécben elsőként, az állandó részhez hasonlóan, egy bitmező kapott helyet, ami megadja az adott üzenet típusát, azaz, hogy nyom- vagy naplóüzenetet tartalmaz. Továbbá az üzenethez tartozó naplószintet vagy a nyomra vonatkozó információt is magában hordozza. A kiterjesztett fejléc további paraméterei az üzenet argumentumszámára és a küldőt azonosító *App*- és *ContextId*-ra vonatkoznak. Így összesen a teljes fejléc hossza legfeljebb huszonhat bájt[7].

Miután megvan a fejléc, bemásolom mögé a „hasznos” üzenetet és meghívom a PDUR API függvényét. Ezután hasonló módon elvégzem a kiküldési procedúrát minden csatorna összes üzenetére. Egy csatorna kiürítése után nullázom a hozzá tartozó üzenetszámlálót, ezzel „törölve” a benne lévő összes üzenetet.

Amennyiben forgalomszabályozás van érvényben, a folyamat azonos, viszont korlátozott mennyiségű adatot engedünk ki minden csatornából. A szabályozás csatornánként eltérő lehet. A konfiguráció során megadták a csatorna kiküldésének ciklusidejét [s] és a maximális sávszélességet [bit/s]. Ez alapján az adott csatorna üzeneteinek kiküldése előtt meg tudom mondani, hogy hány bájtot engedek ki az adott hívásban.

Az egy csatornából aktuálisan kiküldött adatmennyiséget eltárolom és minden üzenet kiküldésével növelem az adott üzenet hosszával. Minden üzenet kiküldése előtt megvizsgálom, hogy az adott üzenet hossza és az eddig kiküldött adatmennyiség összege meghaladja-e a maximálisan kiengedhető bájtmennyiséget. Ha nem haladja meg, akkor az adott üzenethez összeállítom a fejlécet, átadom a PDUR-nek és növelem a kiküldött mennyiséget a teljes üzenet hosszával. Egyébként a következő csatornát kezdem el vizsgálni. Minden elküldött üzenet után csökkentem egyel a csatornához tartozó üzenetszámlálót, ezáltal „törölve” őket. A DLT üzenetek kiküldésének folyamatát a 19. ábra szemlélteti.

Az Függelékben megtalálható 22. ábra segít megérteni az üzenetek kiküldésének folyamatát.



19.ábra: Üzenetek kiküldés forgalomszabályozással és nélkül

5.5. A megvalósított DLT modul tesztelése

5.5.1. A modul tesztkörnyezete

Az Eclipse fejlesztőkörnyezetben a DLT modul és tesztprojekt kapott helyet. A modul könyvtárban a már megismert DLT API függvényei, adattípusai és a konfigurációhoz kapcsolódó Java generáló kód helyezkedik el. A tesztprojekt a BSW-t, és azon belül a környező modulok viselkedését és elérhetőségét helyettesíti. Elsőként létrehoztam a kapcsolódó modulok header fájljait, amikben implementáltam a szükséges adattípusokat és függvénydefiníciókat, amiket a DLT modul is használ. Ilyenek a generált, RTE által definiált adattípusok is, amik segítségével a modulok és komponensek üzenetet küldhetnek a DLT-nek.

Például itt definiálom az egyes naplószinteket, nyominformációkat és azonosítókat, amiket alapvetően nem a DLT szolgáltat, viszont ezek nélkül a modul önmagában nem lenne képes lefordulni. A modulból meghívott függvények esetén szükséges az API-k pontos megírására, csupán az interfészt kell nyújtani, ami mögül megfigyelni és regisztrálni lehet a DLT modul viselkedését.

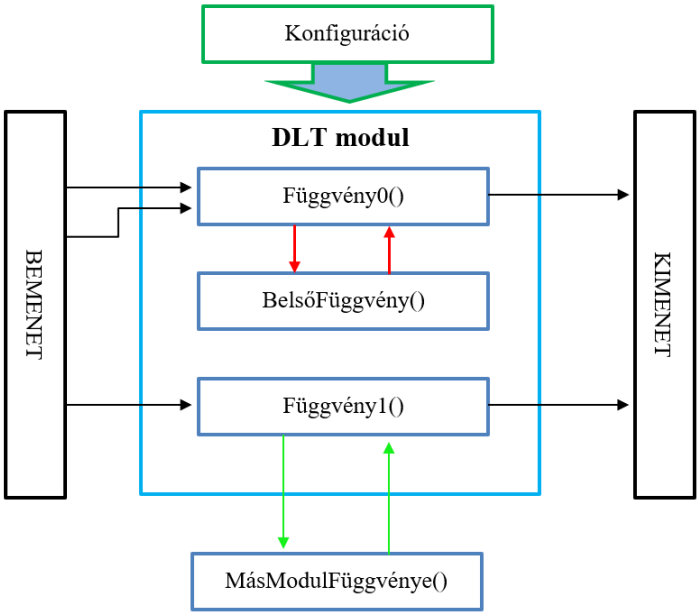
Jó példa a Dlt_DemTriggerOnEventData függvényében meghívott DEM modul API függvényei. Elsőként szükség volt egy valódihoz hasonló eseményazonosítóra, amit megkaphat a DLT függvény, és lehívhatja vele az eseményhez tartozó hibakódot (DTC) és a *FreezeFrame*-et. A Dem_GetDTCOfEvent API-t reprezentáló függvényem implementációja bizonyos eseményazonosítót elfogadott és visszatért egy fiktív hibakóddal (DTC), viszont a teszteléshez hozzátartozik a sikertelen függvényhívás esete is, amit egy másik eseményazonosító esetén szolgáltat a függvény.

A modul teszteléséhez különböző konfigurációs beállításokat és bementi paramétereket választok, aminek hatására a kimeneten, azaz a kapcsolódó modulok API függvényeinek törzsében és az API-k kimenetén különböző értékek jelennek meg (20.ábra).

Az egész modult meghatározza a konfigurációja. Alapvetően a konfigurációban való bármilyen változtatás nagy hatással lesz a modul viselkedésére. A DltGeneral konténerben, az általános globális viselkedést határozzuk meg. A DltSwc a szükséges konfigurációs paramétereket tartalmazza, hogy mely modultól vagy szoftverkomponenstől kaphatunk üzenetet. Ezek minden esetben felhasználásra kerülnek a modul működése során.

A ConfigSet tároló felsorolja az összes lehetséges DLT-funkciót, amiket a konfiguráció során, fordítás előtti időben engedélyez vagy letilt az erőforrás-fogyasztás optimalizálása érdekében. A Configset-ből több példány is létrehozható, viszont egyszerre csak egy példány lehet aktív a modul futása alatt. Inicializálás során az alkalmazott konfigurációra vonatkozó átadott pointer a ConfigSet konténernek csak egy példányát tartalmazza.

A függvények vizsgálata vizuálisan történik a standard kimenetre való kiíratásokkal, így figyelemmel kísérhetem az egyes függvényhívások pillanatát és a paraméterek értékét. Amikor egy függvényt meghívunk, először kiírjuk az adott függvény nevét, valamint egy számláló segítségével vizsgáljuk, hogy egy hívásból hány további hívás történik. Ezáltal felállíthatjuk a modul mélységi fáját. A DLT és a kapcsolódó modulok függvényeinek hívását különböző teszt függvények segítségével vizsgálom.



20.ábra: A modulteszt felépítése

A modul konkrét értékekkel való teszteléséhez létrehoztam három forrásállományt. Egy az értékek kiíratásáért, egy a meghívott függvények paramétereinek vizsgálatáért, és egy a különböző tesztesetek felállításért felelős.

Az értékek kiíratására különböző függvényeket hívhatunk meg. Külön az egészekhez, a tömbök elemeinek értékéhez és a visszatérési értékekhez. A megjelenítő függvényekben kiírom az adott paraméter nevét és értékét. A legjellemzőbb értékek az előjelnélküli egész számok. Mivel sokszor bitszinten szükséges gondolkodni, ezért a függvény hívásakor kiválasztható a megjelenítés formája, így nemcsak decimális, de hexadecimális értékeket is kiírhatunk.

A DLT modulban gyakori felhasználással jelentkeznek az ASCII karakterből álló tömbök. Ezért az adott tömb elemeinek kiíratása történhet decimálisan vagy karakterként. Így a legtöbb azonosító könnyen felismerhető.

A függvények fontos paramétere a visszatérési érték. A DLT modul függvényeinek szabványos visszatérési értékei lehetnek, amik a függvény feladatának sikeres, vagy sikertelen végrehajtását mutatják. Általános értékek az E_OK és E_NOT_OK, viszont definiál a szabvány több, az adott függvényekben a hibákra jellemző értékeket is.

A tesztprojektben, egy külön forrásállományban, minden általam megvalósított DLT API vizsgálatához egy-egy segédfüggvényt hoztam létre. A segédfüggvények azonos bemeneti paraméterekkel rendelkeznek, mint az eredeti API, és a tesztelés során azonos értékekkel is hívom meg őket. A segédfüggvények kontextusában a hívó paraméterek és a visszatérési érték kiíratása történik. Ezáltal megfigyelhető, hogy aktuálisan milyen értékkel hívjuk meg az adott függvényt. Amennyiben lekérünk adatokat, a segédfüggvényt a DLT API előtt és után is meghívjuk. Erre azért van szükség, mert a lekért adatokat az átadott tárolókban kapjuk vissza sikeres lefutás esetén és így könnyedén kiírathatjuk azokat.

5.5.2. A tesztelés lépései

A félév során, ahogy a DLT megvalósított API készlete növekedett, úgy igyekeztem vele párhuzamosan bővíteni a tesztstruktúrámat. A félév vége felé egy egységes tesztet szerettem volna kialakítani, amivel a modul integrált működését lehet szemléltetni. A modul teszteléséhez elsőként a megfelelő konfigurációs paraméterek megválasztása szükséges. A DltGeneral konténerben több általános feltétel is meghatározható a DLT modul működésére vonatkozóan. Az általam eddig megvalósított DLT függvények viselkedése az időbélyeg-támogatástól és a sávszélesség-korlátozástól függenek. A DltGeneral-ban található többi feltételtől még nem.

A DltSwc konténerekben a komponensekkel való kapcsolatok kerülnek meghatározásra. Összesen két konténert vettem fel a tesztelések érdekében, amik egy-egy szoftverkomponens konfigurációját tartalmazzák. Létrehoztam mindkettőben három-három különböző *AppId/ContextId* párost, amikből egyet-egyet csupa nulla *ContextId*-val, azaz a *wildCard*-al. Ezenkívül a két komponensnek egy-egy azonosító párosa megegyezik. Megadható a komponensektől kapható legnagyobb üzenethossz értéke. Ezeket komponensenként más-más értékre állítottam, ezzel is tesztelve a kódgenerálás megfelelő működését is, hiszen ezen értékek maximuma alapján generálok tároló részt a kimenő üzenetek számára.

ConfigSet konténerből kettőt hoztam létre. Ebből futás során csak egy lehet aktív, az, amelyiket az inicializáció során átadtuk a *DLT_Init* függvénynek. A két elemben beállíthatóak a csatornák paraméterei. Mind a két lehetőségnél a csatornák azonos id-val rendelkeznek, viszont méreteik és egyéb paramétereik eltérnek. A csatorna-hozzárendeléseket az azonosítópároshoz ugyanúgy definiáltam a két konténerben. Az azonosítópároshoz tartozó szűrési feltételeket (naplószintküszöbérték és nyomstátusz) különböző értékűre állítottam, így a különböző tesztesetek során az üzenetek más szűrési feltételek alapján kerülhetnek be ugyanabba a tárolóba.

Először a komponensek regisztrálásával, valamint a beállító és lekérő függvények tesztelésével foglalkoztam. Ez csak úgy működhet, ha a modult inicializáljuk. Esetemben a komponensek felvétele/regisztrálása során nem számít, hogy melyik ConfigSet konténert alkalmazom. A regisztrációhoz létrehoztam az egyik szoftverkomponens azonosítójához egy *AppId/ContextId* párost és a hozzájuk tartozó leírásokat. Ezek után meghívtam a DLT_RegisterContext API-t és ezzel együtt meghívtam a hozzátartozó segédfüggvényt, ami kiírta a függvényhívás paramétereit. A regisztráló függvény sikeres lefutást jelentett, tehát elviekben bekerült az adott azonosítópáros. A modul lehetőséget kínál az egyes azonosítókhoz tartozó információk lekéréséhez a DLT_GetLogInfo API segítségével. Tehát a megfelelő API hívással megbizonyosodhatunk, hogy a kívánt helyre került az általunk regisztrált páros és az azonosítókhoz tartozó leírás. A teszt során megfelelő hívás esetén visszakaptam a beírt értékeket. Viszont a lekérdező függvény nem csak a regisztrációkor bekerült értékeket adja vissza.

A regisztráció során nem kerül az adott azonosítópároshoz nyomstátusz vagy naplószint. Mivel a szabvány nem definiál újabb regisztráció esetén alapértelmezett naplószint vagy nyomstátusz hozzárendelést, ezért a lehívás során az alapértelmezett adatokat

adja vissza a függvény a nem definiált értékekre. A modul működése ettől függetlenül jó, mert a naplószint- és nyomstátuszkezelés nem az azonosítópáros regisztrálásának helyével azonos tárolóban található. Ezért, ha az adott elemhez be szeretnénk állítani a megfelelő értékeket, akkor meg kell hívni a beállító függvényt, azaz a DLT_SetLogLevel vagy DLT_SetTraceStatus függvényt. A beállított értékeket itt ugyanazzal a függvénnyel olvassuk vissza. A változtatások sikeresen bekerülnek.

Az alapértelmezett beállítások megváltoztatása és azok értékének lekérése, az üzenetek szűréséhez és az egyes csatornák paramétereihez a megfelelő API függvényekkel történik, amelyek hívásakor a segédfüggvények segítségével kiíratom a hozzájuk tartozó értékeket.

Amikor leáll egy komponens, akkor meg kell szüntetnie az összes hozzá tartozó *AppId/ContextId* páros regisztrációját. A folyamat sikerességét közvetlenül ellenőrizni nem lehet. Viszont a fogadófüggvény (Pl.: DLT_SendLogMessage) visszatérési értéke, a törölt azonosítókkal beérkezett üzenet esetén, igazolja a törlés lefutását.

A modul viselkedését meghatározza, hogy inicializáció előtti vagy utáni állapotban vagyunk. Amennyiben előtte vagyunk, az érkező üzenetekhez időbélyeget társítunk, ha szükséges, és eltároljuk őket egy átmeneti tárolóba.

Miután meghívtuk a *DLT_Init* API-t, a tárolók rendelkezésünkre állnak és az addig eltárolt üzenetek feldolgozásra kerülnek. Ha megfeleltek a szűrési feltételeknek bekerülnek a megfelelő csatornákba. A DLT üzenetek kiküldésekor az időbélyeg segítségével egyszerűen ellenőrizhető a sikeres működés. A tesztelés során különböző azonosítókkal, különböző hosszúságú és tartalmú üzeneteket kell eltárolni, hogy minél több lehetséges esetet lefedjünk. Az üzenetek tartalmában a várt végeredményt és a tesztet azonosítom, ezáltal is megkönnyítve a várt és a valós kimenetek összehasonlítását. A szűrési feltételeknek nem megfelelő üzenetek esetén a függvények visszatérési értéke beszédes, és látható, hogy az adott „rossz” üzenet a megfelelő indok miatt bukott-e el.

A kiküldés folyamatának helyességét a DLT_TxFunction meghívásával állapíthatjuk meg. Amennyiben a forgalomszabályozás be van állítva, többszöri hívás szükséges az összes üzenet elküldéséhez. Az összeállított szabványos üzeneteket a PDUR stubolt függvénye fogadja, ahol az egyes üzenetek kiírása megtörténik.

A félév során a tesztelések többször segítettek javítani az implementáción. Mindezek után kijelenthető, hogy a tesztek sikeresen lefutnak és nincs ismert hiba a modulban. A tesztelő struktúra működőképesnek bizonyult, viszont így is nagy odafigyelés szükséges az eredmények összehasonlításához. Ezért automatizáltan kell majd ellenőrizni, ami a későbbiekben a normál modulteszt feladata lesz.

6. Összefoglalás és továbbfejlesztési lehetőségek

A félév során megismertem az AUTOSAR szabvány által leírt autóipariszoftverarchitektúra felépítését és működését. Megismertem a rétegek felosztását és általános működésüket, továbbá az egyes modulok kapcsolatát, valamint a kommunikációért és diagnosztikáért felelős modulok funkcionalitását. A szabvány angol nyelvű, így a szakmai nyelvismeretem is bővült.

Az általam megvalósított DLT modul a szabványnak megfelelően működik, viszont nem teljesíti minden követelményét a 4.3.0 verziónak. A klasszifikált követelmények közül a megvalósíthatóak 55% -át sikerült a félév során implementálni.

Követelmények száma	202
Implementált követelmény	81 (55%)
Implementálható követelmények	147
Tesztelhető Követelmények	110

5.6.1. Táblázat: Kalsszifikált követelmények teljesítése

Ezért továbbfejlesztési lehetőségként a további funkciókat megvalósító API-k implementálása tűzhető ki. A modul feladatai közé tartozik a DLT felhasználók értesítése naplószint vagy nyomstátusz változásakor, valamint az aktuális konfiguráció elmentése a nemfelejtő memóriába, és az onnan való visszahívása.

Szükség van az értesítéseket megvalósító API-kra, amiket az alsóbb szintek moduljai tudnak meghívni. Például küldés esetén alulról érkezik visszajelzés a sikeres vagy sikertelen teljesítésről.

A szabvány folyamatosan fejlődik. Az általam megvalósított modul legfrissebb szabvány verziójában (R20-11) az általam is felfedezett kisebb-nagyobb hibákat nagyrészt már javították. Így az ott felsorolt újítások hasznosak lehetnek a 4.3.0 megvalósítása esetén is, azzal együtt is, hogy nem a legfrissebb verziót valósítom meg.

A konfiguráció javítása érdekében plusz paramétereket vennék fel, hogy a lefoglalt tárterületeket egyszerűbben lehessen beállítani. Az init előtti üzenetek részére az átmeneti tárolók méretét és az egy komponenshez felvehető komponensek számát konfigurálhatóvá tenném a jobb kezelhetőség érdekében.

Jelenleg a naplószintek, a nyomstátusz kezelés és az azonosítók csatornához rendelése külön tárterületen helyezkedik el. Többször eltárolom ugyanazt az azonosítót, ami átláthatóság és működés szempontjából jobb és biztonságosabb, viszont nagy adatmennyiség esetén jóval erőforráspazarlóbb is. Ezért ezeknek a tárolóknak az optimalizálása és a hozzájuk kapcsolódó lekérdező és beállító API-k biztonságosabb működtetése lenne célszerű.

Irodalomjegyzék

[1] AUTOSAR - Layered Software Architecture: https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (2020.11.22.)

[2] ISO 15765 - 2:2016 https://www.iso.org/obp/ui/#iso:std:iso:15765:-2:ed-3:v1:en

[3] Dr. Fodor Dénes, Dr. Szalay Zsolt - Autóipari kommunikációs rendszerek, https://regi.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0042_autoipari_kommunikacios_rendszerek/ch04s02.html (2020.10.23.)

[4] AUTOSAR - Specification of Diagnostic Communication Manager https://www.autosar.org/fileadmin/user_upload/standards/classic/3-1/AUTOSAR_SWS_DCM.pdf (2020.12.05)

[5] ISO 14229 - 1:2020

[6] AUTOSAR - Diagnostic Event Manager (SWS) https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_DiagnosticEventManager.pdf (2020.12.03.)

[7] AUTOSAR - Diagnostic Log and Trace (SRS): https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SRS_DiagnosticLogAndTrace.pdf (2020.12.03.)

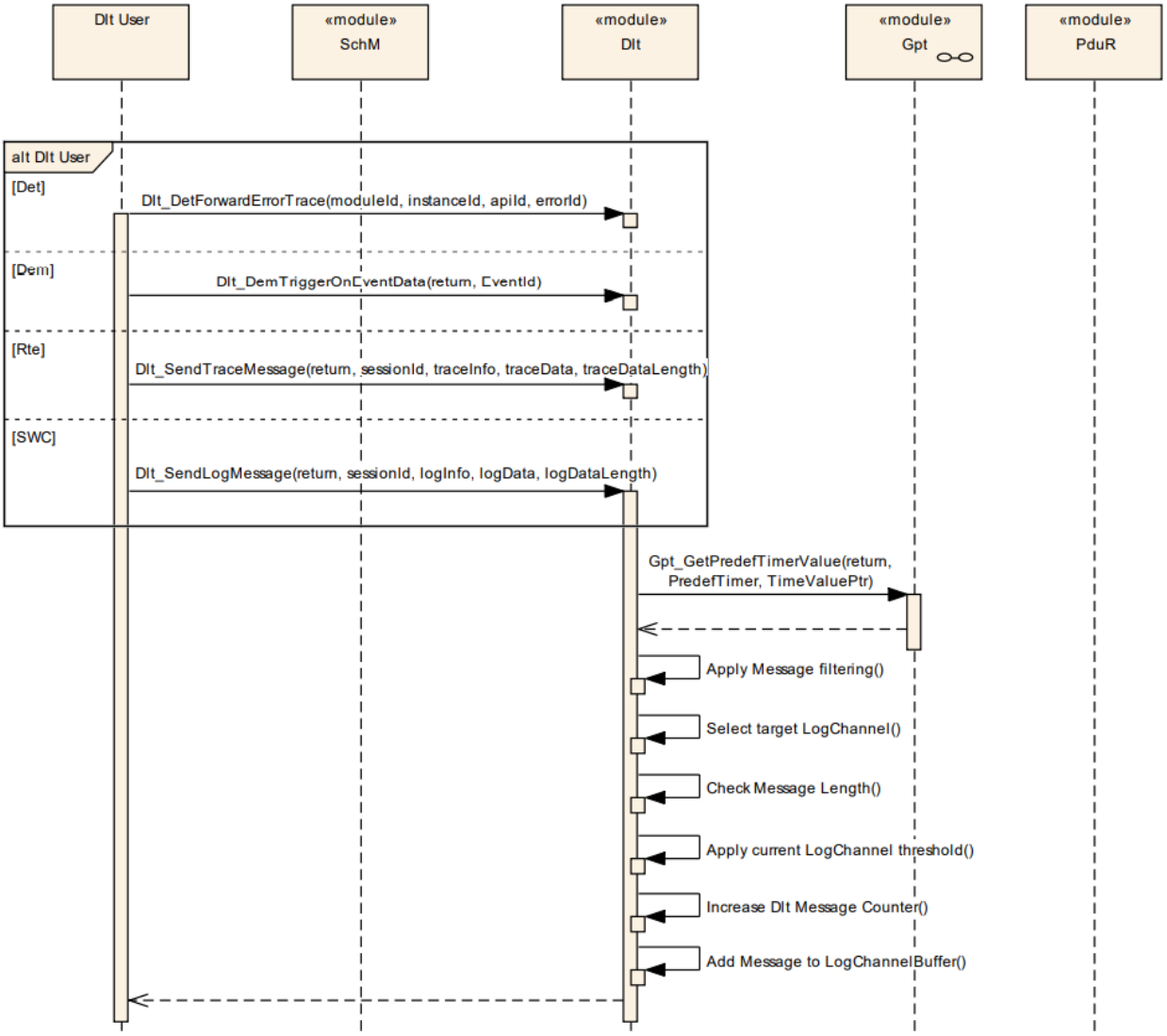
[8] AUTOSAR - Diagnostic Log and Trace 4.3.0 (SWS)

[9] AUTOSAR - Diagnostic Log and Trace R20-11 (SWS) Specification of Diagnostic Log and Trace (autosar.org) (2020.12.05)

[10] AUTOSAR - Specification of Network Management Interface https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_NetworkManagementInterface.pdf (2020.12.05)

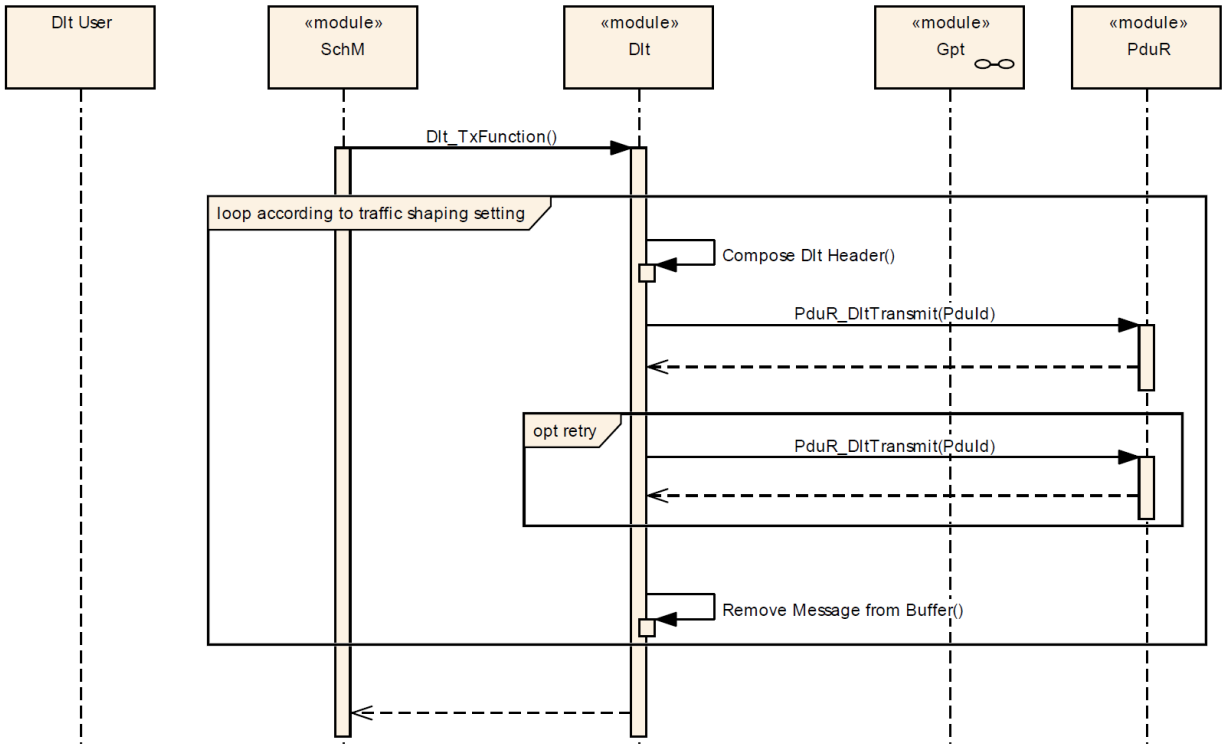
Függelék

A függelék az AUTOSAR- Diagnostic Log and Trace specifikációjából kimásolt folyamatábrákat tartalmazza. Az ábrák segítenek megérteni az üzenetek fogadását, feldolgozását és kiküldését a DLT modulban[8].



21.ábra: Egy üzenet fogadása egy csatornán

A 21. ábrán látható, hogy a folyamatot egy DLT felhasználó indítja, a modul által biztosított API segítségével. Az üzenet beérkezése után a modul lekéri az üzenethez tartozó időbélyeget, elvégzi a megfelelő szűréseket az adott üzenethez tartozó feltételek, majd a kiválasztott csatorna alapján. Ezek után (ha megfelelt) elhelyezi az adott tárolóban.



22.ábra: Üzenetek kiküldése

A 22. ábra egy csatornából való üzenetkiküldéshez szükséges függvényhívások menetét mutatja be. A ciklus ismétlődésének száma a forgalomszabályozástól függhet. Elsőként az adott üzenethez tartozó fejléc összeállítása történik, majd a kiküldése és „eltávolítása”.