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

Magyar Marcell

AUTOSAR LARGE DATA COM MODUL MEGVALÓSÍTÁSA

KONZULENS

Krébesz Tamás

Szikszay László, thyssenkrupp Components Technology Hungary Kft.

BUDAPEST, 2019

TARTALOMJEGYZÉK

HALLGATÓI NYILATKOZAT

Alulírott Magyar Marcell, 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) közzé tegye. 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, 2019. 12. 05.

.....

Magyar Marcell

Összefoglaló

Az AUTOSAR szabványcsalád az autóipar meghatározó szabványává nőtte ki magát, mert kialakított egy, az autógyártó cégek által használt közös „nyelvet”. Erre azért volt szükség, mert azelőtt nem volt egységes szoftverarchitektúra. A mai járművekben rengeteg vezérlőegység található, amelyeknek kommunikálniuk kell egymással annak érdekében, hogy egy teljes rendszert alkossanak. Ezt a kommunikációt valósítja meg többek között a Large Data COM modul.

A Large Data COM implementálása során annak specifikációja adott útmutatót, pontosabban a követelmények, amiket a modullal szemben támaszt a szabvány. Első modul implementációm lévén sok dolog újként hatott, de végül sikeresen vettem az akadályokat. A legnagyobb megmérettetésként a függvénypointerrel megvalósított megoldásra tekintek, azonban kis irodalomgyűjtés és utánanézés után sikeresen leküzdöttem ezt is.

Dolgozatom során ismertetésre kerültek az AUTOSAR kommunikációs rétegen belüli kommunikációért felelős modulok, maga a kommunikációs réteg, valamint a Large Data COM fő funkciói is. A követelménylista vizsgálata után megírtam a kód statikus részét is, ami a konfigurációfüggetlen része a modulnak. Ezt követte a dinamikus kód implementálása a kódgenerátorral. A statikus kód C nyelven, míg a dinamikus Java nyelven íródott. A megvalósítást leteszteltem smoke test segítségével, ami nem tartalmazza a teljes körű modultesztet, hanem csak a modul alapvető működését ellenőrzi.

Abstract

The AUTOSAR standard family became an important and acknowledged standard by establishing a common ‘language’ for the automotive industry. Before that a unified software architecture never existed. Nowadays there are a lot of electronic control units in cars that need to communicate with each other and build up a whole communication system. Among others, this communication is established by Large Data COM.

During the implementation of Large Data COM, its specification, especially its requirements guided me. As it was my first module implementation a lot of things were brand new but I was able to overcome the difficulties. The biggest challenge was the function pointer-based solution for the configured named functions but some literature research helped me out.

In my thesis I pointed out the function of the communication modules in the communication layer, the communication layer itself and the main function of the Large Data COM module. After examining the requirement list, I implemented the static part of the code which does not depend on the configuration. The dynamic part of the code with the code generator came just after that. The static code was written in C while the dynamic code was written in Java. As for the testing I implemented the so called ‘smoke test’ which does not include the whole module test but it only tests the basic function of the module.

1. Bevezetés

Szakdolgozatomat a thyssenkrupp Components Technology Hungary Kft.-nél készítettem. Feladatom egy modul, név szerint a Large Data COM implementációja volt, amely az autóiparban rendkívül elterjedt AUTOSAR szabvány része.

1.1. Motiváció

A mai, modern autóiparban a növekvő igények és a technológia robbanásszerű fejlődése miatt rendkívül sok elektronikus vezérlő egység található minden egyes autóban. Ezek a vezérlő egységek felelősek többek között a motor, a fékek, az áttétel vezérléséért, de vezérlő egységek irányítják a sokak által kedvelt, mostanra már elengedhetetlen kellékként ismert tempomat-ot is. Ahhoz, hogy ezek az elektronikus vezérlő egységek jól tudják végezni a munkájukat, kommunikálniuk kell más vezérlő egységekkel is.

1.2. A feladat értelmezése

A félév során feladatom volt egy AUTOSAR kommunikációs modul, pontosabban a Large Data COM modul implementálása, amely a Runtime Environment és a PDU Router közötti adatátvitelt valósítja meg. Ez a modul buszrendszeren valósítja meg az adatküldést és –fogadást, továbbá támogatja az adatok szegmentált átvitelét is, amely abban az esetben lényeges, ha az átviendő adat mérete nagyobb, mint az egy keretben szállítható maximális adatméret. Továbbá, jelentős szerepet játszik a diagnosztikai üzenetek továbbításában.

1.3. A feladat indokoltsága

A kommunikáció az autóiparban rendkívül fontos. Ha egy modern autóban nem tudnának kommunikálni a vezérlő egységek az autón belül, akkor nem villanna fel a féklámpa fékezéskor, nem lehetnének automata váltós autók, nem tudná szabályozni a motor a levegő-üzemanyag arányát a gázpedál lenyomásának mértékében stb. A kommunikáció könnyen tetten érhető például a diagnosztikai üzenetek formájában. Elég csak a hibajelző lámpákra gondolni az autó műszerfalán.

1.4. A szakdolgozat felépítése

A szakdolgozatomnak két fő része van: a kapcsolódó téma elméleti háttere, valamint a feladat megvalósítása. Az elméleti részben bemutatom az AUTOSAR szabványcsaládot, valamint annak rétegzett architektúráját. A szabványon belüli kommunikáció ismertetésére nagy hangsúlyt fektettem, a feladatommal való kapcsolatból kifolyólag. A megvalósításban részletesen kifejtem a Large Data COM modul API-jainak működését, valamint a konfigurálhatóságra is kitérek. A szakdolgozatom végén található a függelékben a kódgenerátor egy részlete.

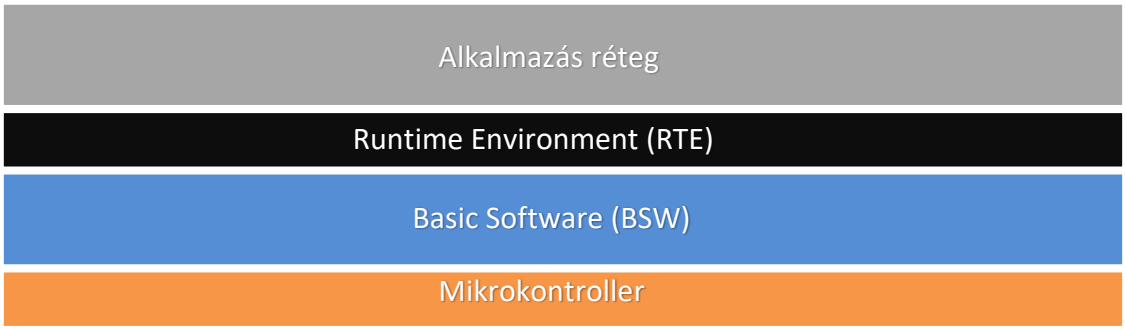
2. Elméleti háttér

Ebben a fejezetben bemutatom a feladatom során felhasznált technológiákat, szabványokat, amik az elméleti alapot adták a kódolási munkához.

2.1. Az AUTOSAR szabvány

Az autóiparban leginkább elterjedt szabványcsalád az AUTOSAR, amelyet a közös érdekeltségű cégek alapítottak 2003-ban. Ez a közös érdek az volt, hogy létrehozzanak egy szabványosított szoftverarchitektúrát, az ECU-kból (Electronic Control Unit), azaz az elektronikus vezérlőegységekből és a köztük levő kommunikációs sínekből álló megbízható rendszer kialakítása érdekében. A vezérlőegységek értelemszerűen nem csak egymással, hanem a jármű egyéb részeivel is kommunikálnak (pl. fényszórók, kamerák). A szabványcsalád fő céljai közé tartozik a skálázhatóság (azaz, hogy alkalmazható legyen az adott szoftver különböző platformokra és járművekre), biztonsági előírások teljesítése és a javíthatóság.

Az AUTOSAR szabványcsaládnak három fő része van, amik rendre (i) a komponensorientált modellezési nyelv, (ii) a gazdag alapszoftverkönyvtár, valamint (iii) az alkalmazási interfészek és a fejlesztési folyamat leírása. A rétegzett szoftverarchitektúrában található rétegeket mutatja be a(z) 2.1. ábra.



2.1. ábra. Az AUTOSAR rétegzett architektúrája

A legalacsonyabb szintű réteg maga a hardver réteg, azaz a mikrokontroller. Ezzel kommunikál a BSW (Basic Software stack), amely olyan alapvető funkciókat valósít meg, mint például a kommunikáció, I/O (input/output) kezelés. Feladata lényegében az alkalmazás kiszolgálása. A futtató környezet (RTE – Runtime Environment) valósítja meg a kommunikációt a komponensek és a BSW réteg által nyújtott szolgáltatások között. A legmagasabb szinten, tehát az alkalmazási rétegben helyezkednek el az egyéb funkciókat megvalósító szoftverkomponensek [1].

2.1.1. BSW stack

A BSW réteg egy szabványos interfészekkel rendelkező, moduláris felépítésű függvénykönyvtár, ami azért felel, hogy a mikrovezérlő szolgáltatásait elérhetővé tegye az alkalmazás számára [6].

A modulok, amik egyaránt tartalmaznak statikus és generált forráskódot is, a vevői követelmények és a mikrokontroller jellemzői szerint testreszabhatóak, azok széleskörben konfigurálhatóságából kifolyóan. Az említett modulokkal szembeni szabványos követelményeket a hozzájuk tartozó specifikáció tartalmazza.

A BSW stack 4 alréteget tartalmaz, amelyek a következők:

- mikrokontroller absztrakciós réteg
- ECU absztrakciós réteg
- szolgáltatás réteg
- komplex driverek

Ezek elhelyezkedését mutatja be a(z) 2.2. ábra:

2.2. ábra. A Basic Software réteg alrétegei

2.1.1.1. Mikrokontroller absztrakciós réteg

A mikrokontroller absztrakciós réteg a stack legalsó rétege, amely a vezérlőegység perifériáihoz való hozzáférést biztosító modulokból áll (ilyenek például a Port Driver, CAN Driver). Feladata az alkalmazott mikrokontroller elfedése, ehhez alacsony szintű meghajtókat használ. Az így kialakult architektúra lehetővé teszi a hordozhatóságot különböző kontroller platformok között. Jellemző a rétegre a konfigurálhatóság, például, hogy a mikrokontroller melyik lábára van kötve egy alkatrész.

2.1.1.2. ECU absztrakciós réteg

Az ECU absztrakciós réteg a driverek szolgáltatásainak segítségével nyújt hozzáférést a mikrokontroller funkcióihoz. Feladata továbbá, hogy a szolgáltatás réteg moduljai számára a vezérlőegységtől független interfészt biztosítson annak funkcióihoz, tehát elfedi

az ECU specifikus I/O kezelést. A konfiguráció ebben a rétegben is megjelenik (pl. melyik csatornát használjuk LIN kommunikációra stb.).

2.1.1.3. Szolgáltatás réteg

A BSW réteg legfelső alrétege a futtató hardvertől teljesen független szolgáltatás réteg, melynek feladata, hogy olyan alapvető szolgáltatásokat nyújtson a szoftverkomponensek számára, mint például:

- operációs rendszer funkciók
- memóriamenedzsment
- mikrokontrollerek közötti kommunikáció

2.1.1.4. Komplex driverek

Az utolsó BSW alréteg a komplex driverek rétege, amely az AUTOSAR szabvány által nem definiált funkcionalitást kelti életre, így biztosítva hozzáférést az alkalmazásnak a speciális funkciókat megvalósító perifériák szolgáltatásaihoz. Emellett akkor is szükség van rájuk, ha nem elégséges az adott BSW modul funkciókészlete. A komplex driverek közvetlen kapcsolatban állnak az RTE-vel és az adott hardverrel.

2.1.2. A Basic Software modulok konfigurálhatósága

Az AUTOSAR szabványcsalád Basic Software modulok implementálása a specifikációjuk alapján történik. A modul implementációjának elkészülte után a következő lépések egyike a tesztelés, amely 100%-os teljesítése után az adott modul helyes működését feltételezhetjük (azért csak feltételezhetjük, mert a kimerítő teszteléshez rengeteg esetet meg kellene nézni, amely teszt futása akár évekig is eltarthatna). Ahhoz, hogy erről minél nagyobb mértékben megbizonyosodjunk, a modul API-jait többféle paraméterrel kell meghívnunk. Ezeket a paramétereket állíthatjuk be a konfiguráció során [6].

A konfigurációs paramétereket három osztályba sorolhatjuk aszerint, hogy mikor kapnak értéket:

- precompile time: az ilyen típusú paraméterek értékét fordítás előtt be kell állítani
- link time: ezen paraméterek értékét a linkelésig kell beállítani
- post-build time: olyan paraméterek, melyek értéke akár a teljes fordítási folyamat után is megváltoztatható

Precompile Time osztály

A precompile típusú paraméterek már fordítás előtt értéket kell, hogy kapjanak. Ez azt jelenti, hogy ezeket az értékeket figyelembe véve meg tudjuk változtatni a modul implementációját. Ez történhet fejlécfájlba generált makrók segítségével (mivel a kódgenerálás a projekt fordítása előtt történik – kódgenerálásról később) és a modul egyes kódrészleteinek generálásával.

A hátránya a precompile paramétereknek az, hogy az értékük megváltoztatása, tehát a kódgenerálás után az egész modult újra kell fordítani. Ennek ellenére nagy előnye az, hogy segítségükkel a kódot, valamint ezzel együtt a modult jelentősen optimalizálni lehet.

Precompile paraméterekre tipikus példa az olyan makró, amely a modul adott funkciójának ki- vagy bekapcsolt állapotától függően kapja értékét (ezt a szabványcsaládban meghatározott STD_ON és STD_OFF jelzi), vagy akár egy tömb maximális méretének meghatározása. Erre egy példa:

```
#define LD_COM_DEVERRORDETECT                (STD_ON)
```

amely a Default Error Tracer működését kapcsolja be és ki, a konfigurációs beállítástól függően.

Link Time osztály

A link time paraméterek segítségével nem tudjuk optimalizálni a kódot, mert annak lefordítása után is kaphatnak értéket. Ezzel szemben előnyük, hogy megváltoztatásukkor nem kell a teljes modult fordítani, hanem elég azokat a fájlokat, amelyek tartalmaznak ilyen paramétereket (ezután a szoftvert újra kell linkelni). Abból kifolyólag, hogy ezek külön fájlban találhatóak, a fordítás után belőlük készített object fájlok kiadhatók harmadik félnek anélkül, hogy a teljes forráskódot oda kellene adni. Így mások is tudják konfigurálni az adott modult.

A link time paraméterek alapján általában extern módosítóval ellátott adatszerkezeteket generálunk (tehát olyan változókat, amelyeket más fájlokból is el lehet érni). A linker (az az eszköz, amely végzi a linkelést) az előre meghatározott változók helyére helyettesíti be az azokra vonatkozó hivatkozásokat.

Post-Build Time osztály

Ebbe az osztályba azok a paraméterek tartoznak, amelyek értékét az adott modul csak futási időben kapja meg, tehát egészen addig változhatnak, ebből következik az, hogy erőforrás-igényes a használatuk, az optimalizáció pedig nem lehetséges. Előnyeik közé tartozik azonban a rugalmasság (nem kell még linkelni sem a megváltoztatásuk után) és az, hogy egyszerre több konfigurációt is tudunk tárolni a vezérlő egységen. Induláskor a többfajta konfigurációból ki lehet választani azt az egyet, amelyiket be akarjuk tölteni.

2.1.3. RTE (Runtime Environment)

A Runtime Environment az AUTOSAR rétegzett architektúrájában a Basic Software réteg és az alkalmazás réteg között helyezkedik el, az ezek közötti kommunikációt valósítja meg. A szoftverkomponensek és a BSW modulok kizárólag az RTE-n keresztül kommunikálnak egymással, függetlenné téve a komponenseket minden vezérlő egységtől és más egyéb komponenstől. A két lényegesebb kommunikációs forma az úgynevezett ECU-k közötti és az ECU-n belüli kommunikáció, melyek közül előbbi magában foglalja a kommunikációs réteget, a COM és a Large Data COM modult is, amelyeket később bővebben kifejtek [3].

A Basic Software modulok csak úgy tudják elvégezni a feladataikat, ha az RTE szolgáltatásait igénybe veszik. A moduloknak lehetnek olyan függvényeik, amelyeket az RTE futtat megadott időközönként (ütemezett függvények), valamint használhatnak olyan kölcsönös kizárási mechanizmusokat, amelyeket ugyanúgy az RTE bocsát felhasználásra az operációs rendszer szolgáltatásai igénybe véve [6].

Minden modulhoz tartozik egy modell, a Basic Software Module Description (BSWMD), amely a hozzá tartozó integrációs tulajdonságait tartalmazza. Az RTE ez alapján a modell alapján tudja elkészíteni a vonatkozó konfigurációt. Ezt a modellt szintén kódgenerátorral készítjük el, mert konfigurációs paraméterek fordulhatnak elő benne.

2.1.4. DET (Default Error Tracer)

A hibák detektálása és kezelése minden egyes szoftver esetében egy nagyon fontos, elengedhetetlen eszköz. Az autóiparban még ennél is inkább fontosabb, ugyanis a járművekben található vezérlőegységek a működésük nagy részét azzal „töltik”, hogy ellenőrizzék, hogy minden rendben van-e, nincs-e hiba [2].

Az AUTOSAR szabványcsalád két nagy hibacsoportot különböztet meg a hibák előfordulása szempontjából:

- hardverhibák
    - oka általában az integrált áramkör fizikai megsérülése, például egy szenzor eltávolításra kerül vagy elveszik
- szoftveres hibák
    - rossz tervezés, helytelen implementáció okozhat szoftverhibákat, ilyenre példa a rossz paraméterekkel való API-hívás

Egy másik szempont a hibák csoportosításakor az, hogy mikor fordult elő az adott hiba:

- fejlesztési idejű hibák
    - a fejlesztési idejű hibák detektálása és kijavítása fejlesztés közben kell, hogy megtörténjen
    - ezeket a hibákat általában a BSW stack-ben detektáljuk, majd jelentjük a Default Error Tracer-nek (DET), ami egy szintén BSW modul
    - példa: API null pointerrel[1] hívódik meg
- futási idejű hibák
    - ezek a hibák a szoftver életciklusa alatt bármikor előfordulhatnak, ezért a detektálást nem lehet kikapcsolni
    - mind a BSW stack-ből, mind az alkalmazási rétegből lehet ezeket jelenteni a Diagnostic Event Manager (DEM) BSW modul részére
    - általában perzisztensen lesznek tárolva, így később bármikor ki lehet olvasni őket (hibakódok, figyelmeztető lámpák a műszerfalon)

Példa a Default Error Trace működésére:

1. A szolgáltatás rétegben található CAN State Manager küld egy kérést a CAN Controller 0-nak, hogy váltson üzemmódot:

```
CanIf_SetControllerMode(0,..);
```

[1] null pointer: egy olyan pointer (a pointer egy olyan változó vagy paraméter, ami egy memóriacímre mutat), amely nem valós objektumra mutat; ennek különböző programozási nyelvekben más és más a jelölése, például NULL, nil

2. A kérés eljut a CAN Interface-hez amely meghívja a CAN Driver adott API-ját egy helytelen azonosítóval:

```
Can_SetControllerMode(12,..);
```

3. A kérés CAN Driver-hez jutása után az elutasítja a kérést a helytelen azonosító miatt, majd jelenti a hibát a szolgáltatás rétegben található Default Error Tracer-nek, ami eltárolja az összes paramétert, amelyek a hibajelentés során át lettek adva:

```
Det_ReportError(ModuleId, InstanceId, ApiId, ErrorId);
```

A kérés a felsőbb szintű rétegek felől az alacsonyabb szintű rétegek felé megy ki, mivel a példa egy üzemmódváltást próbál szemléltetni. A hibás API hívás után a CAN Driver az alábbi paramétereket adja át a DET-nek (Det_ReportError):

- az adott modul azonosítóját (ModuleId)
    - azt mutatja meg, hogy melyik modul esetében történt a hibás hívás, ezt az ID-t a szabványcsalád határozza meg a Basic Software modullistában
    - típusát tekintve uint16
- az adott modul példány azonosítóját (InstanceID)
    - ez akkor lényeges, ha a modulból nem csak egy példányunk van, ha igen, akkor 0-t adunk át InstanceID-ként
- az API azonosítóját, amelyben a hiba előfordult (ApiId)
    - ezek az azonosítók az adott modul szoftverspecifikációjában vannak definiálva, általában konstansokként, számozásuk a specifikációban definiált módon történik
    - például: #define LDCOM_INIT (0x00) – ebben a példában az inicializáló függvény azonosítóját a 16-os számrendszerben írt 0 szám jelzi
- a hiba azonosítóját (ErrorId)
    - ezek az azonosítók is az adott modul szoftverspecifikációjában vannak definiálva, ellenben általában ezek csak „#define” formájában jelennek meg
    - ha olyan hibát észleltünk a szoftvermodulban, ami az AUTOSAR modulhoz tartozó specifikációjában nincs definiálva, akkor ezeket a hibákat dokumentálni kell

A szabványcsaládban található modulok sokszínűségéből adódóan számos hibatípus található, amik közül párat megemlítenék szabvány szerinti elnevezéssel:

- E_PARAM_PIN: az API hibás ID-val lett meghívva
- E_UNINIT: az API úgy lett meghívva, hogy a modul még inicializálatlan állapotban volt
- E_PARAM_POINTER: az API null pointer-rel lett meghívva

A hibák tárolásának és kezelésének megoldása a szoftverfejlesztő dolga, mely alatt az AUTOSAR szabvány szerint a DET-nek való jelentést értem. Egyéb lehetőségek erre a teljesség igénye nélkül:

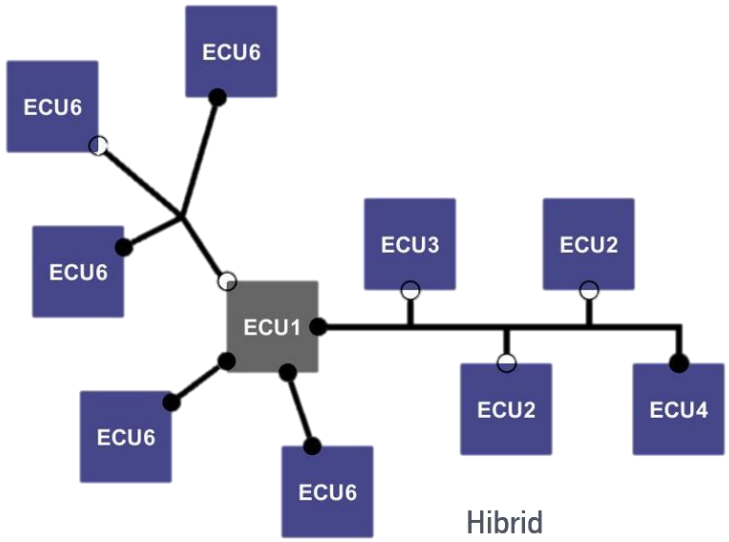
- breakpoint-ok használata azon az API-n belül, ami a hibát jelzi
- jelentett hibák számolása
- az adott hívás és a paraméterek eltárolása egy tárolóban (buffer-ben)
- hibák kiíratása a konzolra

2.2. Kommunikáció a modern járművekben

A járművek alapfunkciójukat, azaz a közlekedést tekintve már elérték a kívánt szintet. Az utasok el tudnak jutni A pontból B pontba, csomagokat tudnak szállítani. Így az autóipari versengés hangsúlya a „hogyan” kérdésre helyeződött át. A járműgyártó cégek a minőségben, a biztonságban és a kényelemben tudnak kiemelkedőt nyújtani. Az ezekkel a tulajdonságokkal szemben támasztott követelmények szintje manapság egyre jobban emelkedik. Ennek köszönhetően, hogy a gyártók ezeket az igényeket ki tudják szolgálni, a járművek elektronikus rendszerének komplexitása, és ezzel egyenesen arányosan a vezérlő egységek száma is folyamatosan nő (ilyenre példa a több processzormagos rendszer, a 100-150 ECU egyetlen járművön belül stb.). Ezeknek az elektronikus egységeknek külön-külön megvan a saját, előre meghatározott feladatuk. A vezérlő egységek csak azokat a jeleket mérik vagy számolják, amelyek közvetlen összeköttetésben, kapcsolatban vannak a saját feladatukkal. A maradék szükséges információ a többi ECU-tól származik, így rendkívül nagy számítási kapacitás spórolható meg.

Ahhoz, hogy a járműben az egyes elkülönített vezérlő egységek adatai el tudjanak jutni másik vezérlő egységhez, kommunikálniuk kell egymással. Az adatküldéssel szemben támasztott biztonsági és megbízhatósági követelmények az éppen küldött adatok szerepétől függ (példának okáért fékezéskor az adatokat elsődleges prioritással és biztonságosan kell elküldeni a féklámpa felé, ezzel szemben alacsonyabb prioritást élvez az, ha az

elektromos tükröket szeretnénk beállítani). A kommunikáció topológiáját tekintve a legfontosabbak a pont-pont és busz alapú kommunikációk. Ezeket természetesen lehet ötvözni, ami a mai modern járművekre jellemző, így megkaphatjuk a(z) 2.3. ábrán látható, bonyolultabb hibrid topológiát.



2.3. ábra. Hibrid topológia [6]

Az adatküldés jellege lehet:

- jelalapú kommunikáció
    - periodikus információváltás a vezérlő egységek között
    - a jelek a kommunikáció legkisebb egységei előre meghatározott fizikai jelentéssel (pl. guminyomás, kerék fordulatszám)
- hálózati üzenetek
    - feladatuk a kommunikációs hálózat karbantartása és menedzselése (alhálózatok be- és kikapcsolása, felébresztése)
    - ECU-k között periodikusan
- diagnosztikai kommunikáció
    - nem a belső, általános kommunikáció része, mindig kell hozzá valamilyen külső diagnosztikai eszköz, amivel a hibakódokat kiolvasni, vagy akár törölni lehet
    - az üzenetek szállítási réteg protokollon keresztül terjednek

Az autóiparban legelterjedtebb kommunikációs protokollok rendre a LIN, a CAN és a FlexRay, ezeket szeretném most bemutatni.

2.2.1. Kommunikációs protokollok

Ahogy még sok más iparágnak, az autóiparnak is szüksége van arra, hogy az üzeneteket lehessen küldeni, erre vannak a kommunikációs protokollok, amelyek különböző vezérlő egységek és más hardverek közötti kommunikációt valósítják meg [4].

2.2.1.1. LIN (Local Interconnect Network)

A LIN egy alacsony megbízhatóságú soros hálózati protokoll, amely azért jött létre, mert a CAN busz túl drága volt ahhoz, hogy minden autón belüli kommunikációt azon keresztül oldjanak meg. Lényegében a CAN buszt egészíti ki. A LIN adatátviteli sebessége megközelítőleg 20 kbit/s. Master-Slave alapú kommunikáció, amely azt jelenti, hogy a master küldi ki az üzeneteket, amire csak az a slave fog válaszolni, aminek az azonosítója megegyezik az elküldött üzenetben szereplő azonosítóval. Előnyei közé tartozik az olcsóság és az egyszerűség. Leggyakoribb alkalmazási területei a sebességtartó automata, fényérzékelők, ülésállítás, visszapillantó tükrök, ablaktörlők [5].

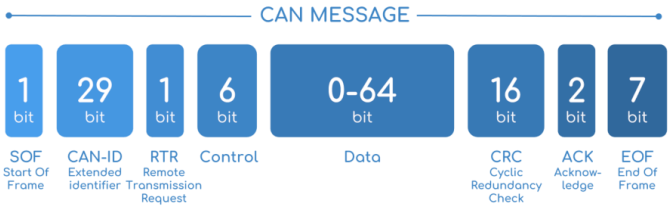
2.2.1.2. CAN (Controller Area Network)

A CAN az autóiparban alkalmazott protokollok közül a legelterjedtebb. Az autóipar mellett rendkívül jelentős a használata az orvosi felszerelésekben is. Jóval magasabb sávszélességre képes, mint a LIN, általában 250 kbit/s és 500 kbit/s között szokták alkalmazni. Esemény vezérelt, aszinkron.

Az üzeneteknek van egy saját azonosítójuk, amely egyértelműen meghatározza az adott üzenetet, azonban csak azt, tehát sem a küldőt, sem a fogadót. Az azonosítók alapján az üzenetek prioritási sorrendbe sorolhatók.

Topológiát tekintve busz alapú, ami azt jelenti, hogy egyszerre csak egy üzenet terjedhet egyszerre ugyanazon a buszon. Ha szabad a busz, akármelyik vezérlő egység tud üzenetküldést kezdeményezni akármikor. Azonban, ha egyszerre kettő vagy több ECU próbál üzenetet küldeni, azok ütközést észlelnek és a már említett prioritás alapján az alacsonyabb prioritású üzenetküldés leállításra kerül úgy, hogy a félbeszakított küldésben szereplő adat nem sérül.

A(z) 2.4. ábrán a CAN üzenet felosztása látható.



2.4. ábra. CAN üzenetkeret felépítése

A legelső bit a SOF, azaz a Start-of-frame, ez jelzi az üzenet kezdetét. Ez minden esetben egy 0 értékű bit. A sorban ezt követi az általában 11 (extended frame, azaz kiterjesztett keret esetében 29, ez látható az ábrán) bit hosszú azonosítóból és a Remote Transmission Request (RTR)-ből álló arbitrációs mező. Az adatmező előtt található a vezérlési mező. A CRC, azaz Cyclic Redundancy Check biztosítja a hasznos információ hibamentességét, tehát jelzi azt, ha hiba van a keretben. Az ACK segítségével nyugtázhatjuk, hogy legalább egy vezérlő egység megkapta az üzenetet. Az End Of Frame-hez tartozó 7 bit adja meg a keret lezárását [8].

A CAN busz kiegészítője a CAN FD, azaz a CAN with Flexible Data-Rate. A mai, modern autókban már ez található meg az eredeti CAN helyett. A váltás azért következett be, mert megnőtt az igény a realtime, azaz a valós idejű adatküldésre, amelyet a sávszélesség növelésével és az adatok közötti késleltetések minimalizálásával érték el. Ezek mellett a CAN FD keretmérete is jelentősen megnőtt a CAN-éhez képest, az eredeti 8 byte helyett egyszerre 64 byte-ot tud szállítani [7].

2.2.1.3. FlexRay

A CAN-nél jóval gyorsabb (maximum sávszélesség: 10 Mbit/s) FlexRay protokollt a FlexRay Consortium fejlesztette ki. A FlexRay idő vezérelt, szinkron, azonban emellett képes esemény vezérelt, prioritást támogató kommunikációra is.

Az üzenetek úgynevezett „időszeletekben” kerülnek elküldésre, amik előre meg vannak határozva bizonyos vezérlő egységekhez, így az üzeneteket a CAN-nél levő azonosító helyett ezesetben a küldés időpontja azonosítja. Az időzítés egysége a hiperciklus, ami 64 FlexRay ciklusból áll. A ciklusfelépítés sokféle lehet, azonban egy konfiguráción belül a ciklusfelépítés ugyanolyan kell, hogy legyen. A ciklusban van egy szigorúan idő vezérelt statikus szegmens és (nem minden esetben) egy esemény- és idő vezérelt dinamikus szegmens is, melyekben csak az ahhoz tartozó ECU küldhet adatot. A különbség akkor

mutatkozik a két szegmens között, amikor nincs küldés. Statikus szegmensnél ezesetben egy speciális Null Frame-t kerül átvitelre (azért, hogy meg lehessen különböztetni az érvényes, csak 0-kat tartalmazó kerettől), míg dinamikusnál ilyenkor „minislot” hosszúságú marad az adott szelet, azaz nem változik a hossza. A maximális keretméret 254 byte [11].

Üzenetátvitelre van lehetőség egyszerre két redundáns csatornán is, ami jelentősen növeli a sávszélességet és a hibatűrőképességet, azonban a gyakorlatban általában egy csatorna van használva.

Az alábbi táblázat összefoglalja az eddig említettnél több szempont szerint a 3 protokollt:

	LIN	CAN	FlexRay
Üzenetküldés	Determinisztikus	Esemény vezérelt	Idő- és esemény vezérelt
Szinkron	Globális referenciaidő	Prioritásalapú arbitráció	Globális referenciaidő
Vezérlés	Master/Slave	Autonóm	Autonóm
Hibakezelés	CRC Paritásbitek	CRC, hibakeretek	2 CRC mező
Átvitel	20 kbit/s	250-500 kbit/s	5 Mbit/s, 10 Mbit/s (2 csat)
Fizikai réteg	1 vezetékes	2 vezetékes	2 csatorna, 2 vezetékes
Felhasználás	Kijelzők, világítás, riasztó, klíma, ablaktörlő, fényszórók		Motor, váltómű, fékrendszer
Ár	Alacsony	Alacsony	Magas
Valós idejű	Gyenge	Gyenge	Erős

2.2.1.4. Szállításiréteg-protokollok

A szállításiréteg-protokollokat arra találták ki, hogy azokat az üzeneteket is el lehessen küldeni, amelyek mérete meghaladja a maximális fizikai keretméretet. Ezek a protokollok

az üzenetet feldarabolják küldés előtt, majd a darabokat egyesítik, miután megérkezett. Két esetben van rájuk szükség:

- diagnosztikai kommunikáció
    - lényegében a szállításréteg-protokollokat a diagnosztikai üzenetek küldésére találták ki
    - a szereplő üzenetek szinte minden esetben meghaladják a CAN keret által nyújtott 8 byte-ot (CAN FD esetén a 64 byte-ot), összetettebb esetben még a FlexRay maximális keretméretét, a 254 byte-ot is, erre példa a szoftverfrissítés esete
- jel alapú kommunikáció
    - amíg a keretméretet meghaladó diagnosztikai üzenetek rendkívül gyakoriak, addig az olyan csak jeleket/jelcsoportokat tartalmazó üzenetek jóval ritkábbak, amelyek meghaladnák akár a CAN maximális keretméretét

A két leggyakrabban használt szállításréteg-protokoll a CAN Transport Layer és a FlexRay Transport Layer [6].

CAN Transport Layer

A CAN Transport Layer azoknak a vezérlő egységeknek az adatszállító protokollja, amelyek a CAN buszon keresztül kommunikálnak. A szabványcsalád lehetőséget biztosít jel alapú adatátvitelre is a protokoll segítségével, ennek ellenére nem ez az eredeti funkcionalitása, hanem a diagnosztikai kommunikáció [12].

A küldés sikerességéről visszaigazolást nem kapunk, azonban ezt ellensúlyozza az úgynevezett STmin és BS segítségével. Az STmin meghatározza, hogy mennyi időt kell legalább betartani két egymás után elküldött üzenetkeret között. A másik áramlásszabályozási eszköz a BS, azaz a Block Size, amely az átvitelre szánt blokk maximális méretét határozza meg (azaz az egymás utáni keretek számát).

A maximális keretméret 64 byte, ami sokszor nem elég arra, hogy nagyobb méretű adatot is átküldjünk. Ennek a problémának a megoldására alakult ki a protokoll két fajtájú átvitele: a nem szegmentált átvitel (tehát az elküldendő adat belefér a keretbe, tehát maximum 7 byte, mert a 8. byte ilyenkor az üzenet hosszát jelző byte) és a szegmentált átvitel (az adatot több keretben tudjuk csak elküldeni).

A szegmentált adatátvitel a következőképpen néz ki:

1. a küldő első keretként elküldi a FirstFrame-t, amelyben jelzi az adat teljes hosszát
2. a fogadó erre reagál egy FlowControl keretben, amely tartalmazza a BS-t és az STmin-t abban az esetben, ha elfogadta az átvitelt (CTS – ContinueToSend)
3. a küldés az átvitel elfogadásával a BS paraméterrel meghatározott blokkméretű (ConsecutiveFrame) adatcsomag küldésével folytatódik STmin időközönként elküldve (tehát, ha a BS = 4, STmin = 10, akkor a küldő egyszerre 4-szer 8 byte-ot küld el 10 ms-ként)
4. miután a blokk át lett küldve, a küldő ismét vár a fogadó FlowControl keretére
5. az utolsó blokk átküldése után az üzenetküldés sikeres

Összefoglalásként felsorolom a CAN Transport Layer-ben használt PDU típusokat:

- FirstFrame: a legalább 8 byte hosszú üzenetek szegmentált átvitele a FirstFrame-mel kezdődik
- FlowControl: a fogadó ezzel a kerettel szabályozza az érkező keretek áramlását, jelezhet várakozási kérést (WAIT), továbbá, hogy nem tudja fogadni az adott hosszúságú üzenetet (OVFLW)
- ConsecutiveFrame: szegmentált átvitel során a szegmensek FirstFrame után ConsecutiveFrame-ekben továbbítódnak, a FirstFrame-ben meghatározott egész üzenet hosszából ki tudja számolni a fogadó az utolsó keret hosszát, így az lehet rövidebb is (SequenceNumber határozza meg a keretek sorszámát, mely megmutatja, ha egy keret elveszett, 0-tól 15-ig számol, majd újra 0-tól, a FirstFrame a 0-s sorszámú)
- SingleFrame: a maximum 7 byte hosszú üzenetek SingleFrame-ben szállítódnak (nem szegmentált átvitel)

FlexRay Transport Layer

A FlexRay Transport Layer működését tekintve hasonló a CAN Transport Layer-höz, azonban van pár különbség a működésükben.

A FlexRay Transport Layer-t ugyanúgy adatszállításra használják, azonban a CAN 64 byte-os maximális keretméretével szemben a FlexRay-jel 254 byte-ot is el lehet küldeni egy keretben. Egy másik előnye a CAN busszal szemben, hogy a teljes üzenet átküldése után a fogadó egy PDU-val jelzi, hogy az sikeresen át lett küldve (a teljes átvitelt a

LastFrame zárja). Emellett előre nem ismert hosszúságú üzenetek küldésére is lehetőség van [6].

2.3. AUTOSAR kommunikációs stack

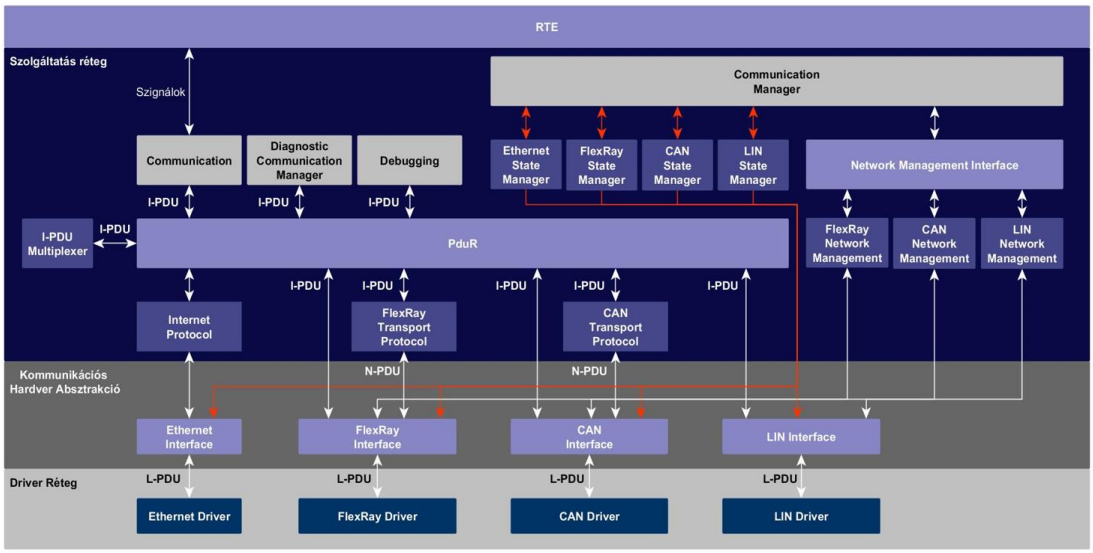
A kommunikációs stack a Basic Software rétegben található meg.

A rétegen belüli szolgáltatások célja, hogy az alkalmazás és a Basic Software modulok részére egységes interfészt nyújtsanak az RTE-n keresztül, miközben elrejtik a protokollt az alkalmazás elől. Ez az egységes interfész a diagnosztikai kommunikáció és a hálózat-menedzsmenthez szükséges.

A kommunikáció alapegységei a PDU-k és az SDU-k, előbbiből több fajta létezik a rétegen belül:

- PDU (Protocol Data Unit): a protokoll által értelmezhető adat
  - L-PDU (Data Link Layer PDU): az adatkapcsolati réteghez kapcsolódó PDU, interfész és driverek között
  - N-PDU (Network Layer PDU): a hálózati réteghez tartozik, szállításréteg-protokollokhoz tartozó adatok
  - I-PDU (Interaction Layer PDU): interakciós réteghez tartozó PDU, a COM modul által értelmező adatokat tartalmaz
- SDU (Service Data Unit): a protokoll által továbbításra szánt adat

A kommunikációs stack-en belüli és a közvetlen szomszédos, egyéb rétegek közötti adatáramlás szemléltetésére szolgál a(z) 2.5. ábra:



2.5. ábra. A kommunikációs stack felépítése [6]

Szó volt már korábban a LIN, CAN, FlexRay kommunikációs protokollokról, amik jelentős szerepet kapnak az adattovábbításban. A Driver rétegből eljut az adat a kommunikációs hardver absztrakció rétegbe, ahonnan a megfelelő interfészektől kiindulva közvetlenül vagy közvetve (szállításréteg-protokollok - Transport Protocols - segítségével) megérkezik a szolgáltatás rétegbe. A szolgáltatás rétegben aztán az adatok haladnak tovább a PDU Router-hez. A PDU Router – ahogy neve is mutatja – elvégzi az „irányítást" és továbbítja a megfelelő modulhoz.

2.3.1. PDU Router

A PDU Router az AUTOSAR Basic Software Stack-ben, azon belül is a kommunikációs stack-ben helyezkedik el, feladata az azon belül áramló I-PDU-k irányítása. Ezt az irányítást egy útválasztási tábla alapján teszi. A nem a szolgáltatás rétegből érkező I-PDU-kat közvetve vagy közvetlenül a már korábban említett kommunikációs protokollok interfészeitől kapja (közvetett esetben a szállításréteg-protokollok nyújtják a köztes szolgáltatást). A PDU Router a szolgáltatás réteg „közepén" helyezkedik el, azaz nála magasabb (pl. COM) és alacsonyabb (pl. CAN transport protocol) rendű modulok is megtalálhatóak a rétegen belül [10].

A PDU Router az I-PDU-k irányítását azok statikusan definiált azonosítója alapján végzi el. Az adott I-PDU alapján kikeresi egy előre konfigurált táblából annak végcélját, majd tovább küldi oda anélkül, hogy megváltoztatná a PDU tartalmát.

Két nagyobb része van: a PDU Router irányítási útvonalak és a PDU Router Motor, előbbi értelemszerűen az I-PDU-k irányításának attribútumait írja le, míg utóbbi a tényleges irányítást végzi az irányítási útvonalak alapján.

Mivel a PDU Router feladata az adatszállítás, így olyan funkciói nincsenek, mint például az adatkonvertálás, adathelyesség ellenőrzése, vagy akár adatváltoztatás. Ezen kívül a PDU Router nem támogatja az adatküldést TP modulok és kommunikációs interfész modulok között.

A modul API-jai a teljesség igényével (feladatom során a Large Data COM a PDU Router-rel kommunikál, ebből kifolyólag sok API elő fog fordulni a modulomban, amire az <Up> előtagú függvények vonatkoznak):

- <Lo>_Transmit
- <Lo>_CancelTransmit
- <LoTp>_Transmit
- <LoTp>_ChangeParameter
- <LoTp>_CancelReceive
- <LoTp>_CancelTransmit
- <Up>_StartOfReception
- <Up>_CopyRxData
- <Up>_CopyTxData
- <Up>_TpRxIndication
- <Up>_TpTxConfirmation
- <Up>_RxIndication
- <Up>_TxConfirmation
- <Up>_TriggerTransmit

Ahol a <Lo>, <LoTp>, <Up> szimbólumok rendre a PDU Router-nél alsóbb szintű kommunikációs interfész modulra, alsóbb szintű szállításréteg-protokoll modulra és a felsőbb szintű modulra utalnak.

2.3.2. Large Data COM

A rétegzett architektúrán belül a Large Data COM (LdCom) a már korábban megemlített PDU Router és az RTE között helyezkedik el, az ezek közötti kommunikációt bonyolítja le. Fő funkciói:

- jel-orientált adat interfész biztosítása az RTE számára
- a kapott jelek továbbítása az RTE-nek
- dinamikus és nagy méretű adattípusok támogatása
- IF (interface) és TP (transport) alapú kommunikáció támogatása
- PDU-orientált adat interfész biztosítása a PduR számára

Ahogy már említettem, a Large Data COM az RTE és a PDU Router közötti kommunikációt valósítja meg. A Large Data COM-hoz képest az RTE a szabvány alapján felsőbb rétegben helyezkedik el.

A másik irányban az alacsonyabb szintű PDU Router-rel van kapcsolatban; ennek mindegyik felső rétegű API-ját használja (felső rétegű API-k alatt azokat az API-kat ér-

tem, amelyek segítségével a PduR a hozzá képest magasabb szintű modulokkal kommunikál). Ezek az API-k egyrészt azokra a modulokra vonatkoznak, amelyek használnak TP-t és azokra, amelyek nem. Ezt külön azért lényeges kiemelni, mert az LdCom az I-PDU üzeneteket vagy darabolás nélkül küldi, vagy darabolva, TP-t használva. A PDU Router-nak jeleznie kell a beérkező I-PDU-kat és az adott I-PDU elküldéséről a megerősítést, valamint támogatnia kell az adatküldést TP kommunikáció keretein belül [9].

A modul kapcsolatban áll még a DET-tel is, ahogy a szabványban a modulok többsége, ezzel eredményezve, hogy hibás API hívások esetén tudja jelezni a hibát.

3. Tervezés és megvalósítás

A szoftverfejlesztés során csak az adott, hogy a végeredmény mi kell, hogy legyen, az viszont nem, hogy hogyan. Rengeteg módja van egy adott funkcionalitás leírásának, implementálásának, ezért is lehet azt mondani, hogy ez egy kreatív mérnöki tevékenység. A fejlesztés során igénybe lehet venni segédeszközöket, mint például kódolási útmutatókat, azonban ez sem kötelező. Az elkészült szoftverrel kapcsolatban a lényeges kitételek azok, hogy megfeleljen a követelményeknek, valamint az, hogy az elkészült kód minőségi kód legyen. Ehhez azonban sok tapasztalat kell, ami a sok kezdetleges erőfeszítés és mások „review" tevékenysége által szerezhető. Kezdő szoftverfejlesztőként ezt a tapasztalathiányt próbáltam pótolni kódolás közben a cégnél rendelkezésre álló fejlesztési útmutatót használva.

3.1. Követelmények

Az AUTOSAR szabványcsaládban található modulok mindegyikéhez tartozik egy specifikáció, melyek séma alapján épülnek fel (5-ös fejezet: más modulokhoz való függés, 7-es fejezet: funkcionális leírás, 8-as fejezet: API specifikáció, 10-es fejezet: konfigurációs lehetőségek). A specifikációban találhatóak meg a követelmények is, amelyek alapján kell a fejlesztőnek implementálnia a szoftvert a fejlesztés során. Ezek a következő formában jelennek meg a dokumentációban [9] (konkrét példával):

[SWS_LDCOM_<követelmény száma>] [<követelmény leírása>]

[SWS_LDCOM_00010] [When LdCom_Transmit is invoked, LdCom shall invoke PduR_LdComTransmit by replacing the Signal Id by the according PDU Id.]

Ahogy már korábban említettem, a fejlesztő a követelmények alapján dolgozik. Az implementáció első lépéseként létre kell hozni egy követelménylistát tartalmazó fájlt ahhoz, hogy az összes, a modulhoz tartozó követelmény össze legyen gyűjtve. Ez egy xml formátumban megírt lista, amely a specifikációban található és az esetlegesen a fejlesztés során pluszba felvett, szükségesnek ítélt követelményeket tartalmazza (előfordulhatnak olyan esetek, amikor a szabvány nem határozza meg konkrétan a működést, ekkor kell új követelményeket felvenni). Ebben a listában olyan sorrendben vannak leírva a követelmények, ahogy a specifikációban, ez a könnyebb átláthatóság érdekét szolgálja.

A fentebb példának hozott követelmény a követelménylistát tartalmazó fájlban a következő formában található meg:

```
<req acceptance="accepted" id="SWS_LDCOM_00010" implementable="true" testable
="false" sourceRef="ArLdCom">
  <text>
       When LdCom_Transmit is invoked, LdCom shall invoke PduR_LdComTransmit
by replacing the Signal Id by the according PDU Id.
  </text>
</req>
```

A követelmények „acceptance” attribútuma a követelmény implementer által elfogadott voltát, az „implementable” a megvalósíthatóságát, míg a „testable” a tesztelhetőségét jelzi. Ha kell módosítani az adott követelményen, az elfogadottságra vonatkozó attribútum „accepted_with_modification” -nel lesz egyenlő.

3.2. Statikus kód

A modul implementálása a statikus kód megírásával folytatódott. Statikus kód alatt az olyan C nyelven íródott kódot értem, amely nem konfigurációfüggő.

A legelső dolgom a modul fájlstruktúrájának létrehozása volt. A Large Data COM modulnak elkészítettem a könyvtárt, ami magát a modul implementációját tartalmazza.

A struktúra létrehozásában, mint ahogy az implementálásban végig, útmutatást nyújtott a specifikáció, azon belül is az ehhez tartozó követelmények. Eszerint a projekt könyvtárjában kell lennie egy LdCom.c forrás- és egy LdCom.h fejlécfájlnak.

A fejlécfájlnak tartalmaznia (include-olnia) kell a következő fájlokat:

- LdCom_Types.h: az LdCom saját típusait, struktúráit tartalmazó fejlécfájl
- ComStack_Types.h: az AUTOSAR szabványcsalád típusait, struktúráit tartalmazó fejlécfájl, külön szoftver

A forrásfájlnak pedig az alábbi fájlokat:

- PduR_LdCom.h: a PDU Router Large Data COM-mal kapcsolatos függvényeit tartalmazó fejlécfájl
- LdCom_Cfg.h: a Large Data COM generált fejlécfájlja, makrók, ’extern’ jelzővel ellátott változók, függvénydeklarációk találhatóak meg benne
- LdCom.h: a Large Data COM fejlécfájlja
- Det.h: a Default Error Tracer fejlécfájlja, csak akkor kell include-olnia, ha a hiba detektálás be van kapcsolva

3.2.1. Hibakezelés

Az AUTOSAR szabványcsalád a Basic Software stack moduljainak hibáit a DET (Default Error Tracer) modul segítségével kezeli, melynek jelenteni lehet a hibás API-hívásokat. Ezeket a hibajelentő függvényeket a Det.h és Det.c fájlokba deklaráltam és implementáltam, amelyek a tesztkönyvtárban lettek elhelyezve.

A DET modulhoz tartozó fájlok létrehozása és a megfelelő függvény implementálása után a modul forrásfájljába (tehát az LdCom.c fájlba, ahol a tényleges implementáció történik) a hibakezelő függvényeket kellett definiálni. Az, hogy milyen hibákat kell kezelni és jelenteni, a követelménylistából derül ki. A hibakezelő függvényekre belső, céges ajánlás szerint a láncolható hibák módszerét használtam.

Ez a módszer a következő: a modul éppen vizsgálandó API-ján belül létrehozok egy változót, ami a hibakódot tartalmazza. A hibakezelő függvények ennek a változónak a címét kapják meg, majd hiba esetén ebbe a változóba írják bele a hibához tartozó hibakódot. Minden egyes függvény úgy épül fel, hogy csak akkor fut le az adott vizsgálat érdemi része, ha nincs még hiba. Ezzel a szerkezettel el lehet kerülni az egymásba ágyazott feltételeket (pl. akkor nincs hiba, ha érvényes a kérés azonosítója, de ezt csak akkor kell ellenőrizni, ha a függvény nem Null-pointer, amit csak akkor kell ellenőrizni, ha a modul már inicializálva van). Ezt váltja ki a hibakezelő függvények meghívásainak egymásutánja. A módszerre példa:

```
void assertUninit(uint8* errorCode) {
	if (LDCOM_E_OK == *errorCode) {
		if (isInitialised == FALSE) {
			*errorCode = LDCOM_E_UNINIT;
		}
	}
}
```

Ezt az API-n belül a következőképpen hívom meg:

```
uint8 errorValue = LDCOM_E_OK;
assertUninit(&errorValue);
assertPointer(&errorValue, PduInfoPtr);
if (LDCOM_E_OK == errorValue) {
	functionality()
} else {
	reportError(LDCOM_TRANSMIT, errorValue);
}
```

A hibakezelő függvényeket egymás után hívom meg az API implementációjában, majd ezután következik az ellenőrzés. Ha nem volt hiba, az adott függvényhez tartozó funkcionalitás kerül végrehajtásra, ha volt, a Default Error Tracer függvényét hívom meg,

amivel a hibát lehet jelenteni. Ennek négy paramétere van, melyből az első kettő egy modulon belül konstans, a harmadik az, hogy melyik API-ban történt a hiba, a negyedik pedig maga a hibakód.

Azt, hogy melyik API-ban történt a hiba, az LdCom.h fejlécfájlban felvett makrók segítségével lehet jelezni. Ezek a makrók a már megszokott #define formában kerültek implementálásra, értékük a követelménylista alapján került beállításra.

A hibakódok a specifikációban, azon belül annak 7-es fejezetében (a funkcionalitás részben) vannak leírva. A szabvány által előre definiált hibakódok a következők:

Hiba típusa	**Kapcsolódó hibakód**	**Értéke [hex]**
Ha az LdCom_GetVersionInfo kivételével akármelyik másik függvény azelőtt lett meghívva, hogy az LdCom modul inicializálva lett volna az LdCom_Init függvénnyel, vagy azután, hogy deinicializálva lett az LdCom_Deinit függvénnyel.	LDCOM_E_UNINIT	0x02
Ha az adott függvény Null-pointerrel lett meghívva	LDCOM_E_PARAM _POINTER	0x03
Ha az adott függvény rossz PDU azonosítóval (ID-val) lett meghívva	LDCOM_E_INVALID _PDU_SDU_ID	0x04
Ha az adott függvény rossz jel azonosítóval (Signal ID) lett meghívva	LDCOM_E_INVALID _SIGNAL_ID	0x05

Ezeket a hibakódokat az LdCom.h fejlécfájlban makrókként definiáltam, például:

```
#define LDCOM_E_UNINIT                  (0x02)
```

A tényleges hibajelentés viszont csak akkor fog megtörténni, ha a konfigurációban be van kapcsolva a fejlesztés idejű hiba detektálása (konfigurációfüggő, erről később).

```
void reportError(uint8 apiId, uint8 errorId) {
#if (STD_ON == LDCOM_DEVERRORDETECT)
      Det_ReportError(LDCOM_MODULE_ID, LDCOM_INSTANCE_ID, apiId, errorId);
#endif
}
```

Nem helyes paraméterek esetén viszont ettől függetlenül sem szabad meghívni az adott funkcionalitást, erre szolgál az API hívásokban az if-else feltétel.

3.2.2. A Large Data COM API-jai

A Large Data COM implementációja az API-k vázának elkészítésével folytatódott. Úgy, mint minden AUTOSAR modul specifikációjában, ezesetben is annak 8. fejezetében található meg az összes API. Ezeket a függvényeket az LdCom.c forrásfájlban implementáltam, az LdCom.h fejlécfájlban pedig deklaráltam.

3.2.2.1. LdCom_Init

Az inicializáló függvény minden Basic Software modul alapfüggvénye, ennek segítségével lehet a modult inicializált állapotba helyezni. Ezt az inicializált állapotot legegyszerűbben egy úgynevezett flag beállításával tudjuk jelezni (ez lényegében egy boolean változó). Ennek a flag-nek az alapértéke ’false’, amely az inicializáló függvényben ’true’-ra lesz beállítva.

```
boolean isInitialised = FALSE;
```

A függvény kap egy LdCom_ConfigType pointer típusú paramétert, amelyet egy, a forrásfájlban globálisként deklarált változó értékeként állítok be. Ezt a globális változót használják azok a függvények, amelyek konfiguráció függőek.

Ebben a függvényben is ellenőrizni kell azt, hogy a paraméterül kapott pointer nem Null-pointer-e. A szintaktika ugyanaz, mint a már korábban említett esetben: ha a lokális változóként felvett hibakód OK, akkor megtörténik az inicializálás, ellenkező esetben a DET megfelelő függvényét hívom meg.

3.2.2.2. LdCom_DeInit

Az LdCom_DeInit függvénynek a segítségével lehet az inicializáló flag-et ’false’ értékre állítani, tehát deinicializált állapotba hozni a modult. Az LdCom_DeInit meghívása után az LdCom_GetVersionInfo függvény kivételével egyik API-t sem lehet használni a következő LdCom_Init hívásig.

3.2.2.3. LdCom_GetVersionInfo

Ez a függvény megadja a modul verziójával kapcsolatos információkat. Paraméterként egy, már előre definiált Std_VersionInfoType pointer típusú változót kell átadni, melyben a függvény eltárolja a modul megfelelő információit. Mivel a paraméter pointer, a hibakezelést itt is implementálni kell az assertPointer segédfüggvényt használva, mely a Null-pointerrel való hívást ellenőrzi.

A modul verzióval kapcsolatos információi konstansok, így ezeket is makrókként definiáltam. Ezek az információk rendre:

```
#define LDCOM_VENDOR_ID                    (0x57U)
#define LDCOM_MODULE_ID                    (49U)
#define LDCOM_SW_MINOR_VERSION             (0U)
#define LDCOM_SW_MAJOR_VERSION             (1U)
#define LDCOM_SW_PATCH_VERSION             (0U)
#define LDCOM_INSTANCE_ID                  (0U)
```

Az ’U’ módosító jelző a számok után arra utal, hogy előjel nélküli számokról van szó.

Ez a függvény csak akkor érhető el, ha a konfigurációs fejlécfájlban makróként generált LDCOM_VERSIONINFOAPI kapcsoló értéke STD_ON.

3.2.2.4. LdCom_Transmit

Az LdCom_Transmit segítségével egy jel küldését lehet kezdeményezni, akár IF, akár TP küldést használva. A függvénynek két paramétere van:

- Id: az elküldendő adat azonosítója
- PduInfoPtr: az adat hosszát és az adat tárolójára mutató pointert tartalmazó struktúra

Az eddig előforduló inicializált állapot és pointer ellenőrző függvények mellett ebben az esetben ellenőrizni kell az azonosító helyességét is. Ehhez készítettem egy másik segédfüggvényt, az isHandleIdEqual nevű függvényt.

3.2.2.5. isHandleIdEqual

A modul specifikációjában a követelmények a PDU azonosítókkal kapcsolatban úgy szólnak, hogy ha a PDU Router meghívja a Large Data COM modult, az előbb említett hívásban szereplő azonosítót a hívott modulnak származtatnia kell a konfigurációban szereplő I-PDU-k HandleId-jából (a HandleId a konfigurált I-PDU azonosítója). Ez lényegében azt jelenti, hogy az LdCom-nak meg kell néznie, hogy a konfigurációban szerepele olyan I-PDU, amelynek a HandleId-ja megegyezik a hívott függvény paraméterei között

szereplő PDU Id-val, melyet a PDU Router általi híváskor kap. A függvény kódja a következő:

```
uint16 isHandleIdEqual (uint8* errorCode, PduIdType id) {
	uint16 i;
	uint16 foundID = 0xFFFFU;
	if (LDCOM_E_OK == *errorCode) {
		for (i=0; i<LdCom_SelectedCfgPointer->iPduCount; i++) {
			if (LdCom_SelectedCfgPointer->IPdu[i].handleId == id) {
				foundID = i;
				break;
			}
		}
	}
	return foundID;
}
```

A funkcionalitása a következő: végigmegy azon a konfiguráción, amellyel inicializálva lett a modul, majd megnézi az összes konfigurált I-PDU-t, és összehasonlítja azok azonosítóit (HandleId-jait) a paraméterként (a PDU Router által) kapott azonosítóval. Ha talál egy olyan I-PDU-t, amely esetében egyeznek ezek a paraméterek, visszatér ennek az I-PDU-nak a konfigurációs tömbben elfoglalt indexével, ha nem, akkor pedig az ennek a változónak eredetileg beállított 0xFFFF értékkel (azért 0xFFFF, mert az azonosítóhoz tartozó hibakezelő függvény erre az értékre is hibát jelez).

A függvény paraméterlistájában megtalálható a hibakód is. Ez azért fontos, mert csak akkor fut le a függvény lényegi része, ha eddig még nem volt hiba abban az API-ban, ahol ez a függvény meg lett hívva (azért „eddig”, mert ezt a függvényt a hibakezelő függvények után hívom meg).

3.2.2.6. Adatközvetítő API-k

A Large Data COM adatokat közvetít az RTE és a PDU Router között. Az adatközvetítés típusa szerint lehet küldés és fogadás. A modulnak mindegyik típuson belül van négy-négy API-ja, melyek közül megemlítenék párat szekvenciadiagram csatolásával, könnyebb megértésük érdekében.

Transmission – Küldés (Tx)

A legfontosabb adatküldő API az LdCom_CopyTxData, ami az LdCom_Transmit függvény hívása után hívódik meg. Mint ahogy neve is sugallja, ez a funkció egy I-PDU szegmens küldésére használatos.

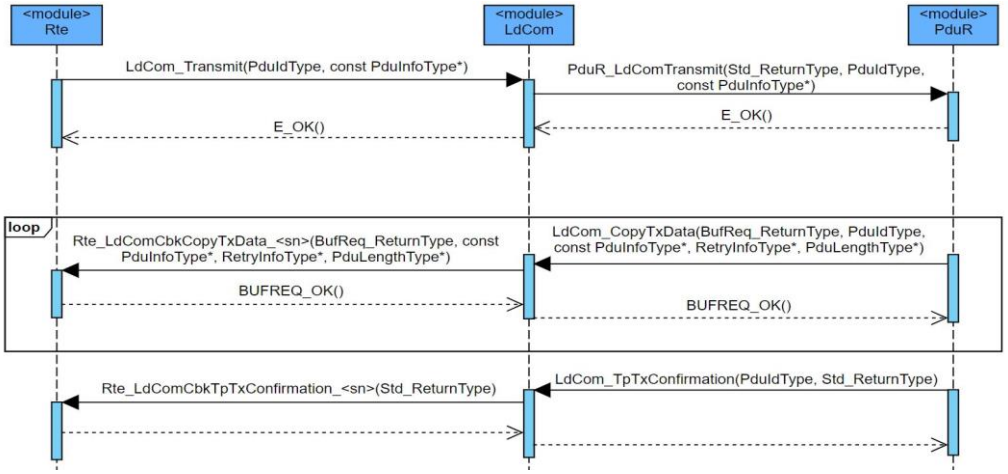
A függvénynek négy paramétere van:

- id: az átküldendő I-PDU azonosítója
- info: a céltárolót és az átviendő byte-ok számát tartalmazza
- retry: ez a paraméter arra szolgál, hogy a küldés sikerességét vagy sikertelenségét jelezze
- availableDataPtr: a felsőbb rétegű modul (RTE) Tx tárolójában hátramaradó byte-ok számát jelzi

A függvénynek BufReq_ReturnType típusú visszatérési értéke van, amely azt jelzi, hogy az adat át lett-e teljesen másolva a ’Transmit’ tárolóba. Az ilyen típusú változóknak három értéke lehet:

- BUFREQ_OK: az adat teljes egészében át lett másolva
- BUFREQ_E_BUSY: a kérés nem teljesíthető, ugyanis nem áll rendelkezésre elegendő mennyiségű Tx adat, erre az alacsonyabb rétegű modul hívhat egy retry-t (újra próbálhatja a küldést)
- BUFREQ_E_NOT_OK: a kérés nem teljesült, az adatot nem sikerült átmásolni a tárolóba

Az adatok küldése szegmensenként történik, melyet a(z) 3.1. ábra hivatott ábrázolni:



3.1. ábra. Adatküldés LdCom_CopyTxData segítségével

Az adatküldés azzal kezdődik, hogy az RTE meghívja a LdCom_Transmit függvényt, amit majd a Large Data COM fog továbbítani a PDU Router-nek a PduR_LdCom-Transmit függvényhívással.

A kérés megérkezése után a PDU Router egy ciklusban elkezdi küldeni az adatokat az RTE részére az LdCom_CopyTxData függvény meghívásával, amit szintén a Large Data COM fog továbbítani az Rte_LdComCbkCopyTxData_<sn> függvényhívással (ennek a függvénynek a neve, ahogy a nevében szereplő <sn> - service name - is utal rá, konfigurálható). Ez a ciklus addig fog folytatódni, amíg minden adat át nem lesz küldve. A visszatérési értékként feltüntetett BUFREQ_E_OK() a szegmens küldésének sikerességét jelenti.

Miután a ciklus futása véget ért (sikeresen vagy sikertelenül), a PDU Router az LdCom_TpTxConfirmation függvény hívásával jelzi az adatküldés sikerességét, vagy ellenkező esetben sikertelenségét. A Large Data COM ezt az üzenetet továbbítja az RTE részére az Rte_LdComCbkTpTxConfirmation_<sn> függvény segítségével.

Ez a fent említett TP folyamat érvényes IF folyamat esetén is, azaz amikor az adat elfér egy szegmensben. A különbség annyi, hogy a ciklus helyett csak egyetlen LdCom_CopyTxData függvény hívódik meg, valamint az LdCom_TxConfirmation (Tp nélkül) opcionális.

Az említett küldési sikerességet jelző LdCom_TxConfirmation (vagy TP esetben LdCom_TpTxConfirmation) függvénynek két paramétere van:

- TxPduId (id): az elküldött I-PDU azonosítója
- Result (result): az adott I-PDU küldésének eredménye

Van egy speciálisabb függvénye a modulnak, az LdCom_TriggerTransmit. Azért speciálisabb, mert ezt az adatküldési kérést nem az RTE kezdeményezi, hanem a PDU Router. A PDU Router küldi az LdCom_TriggerTransmit kérést a Large Data COM-nak, mely továbbítja azt az RTE-nek az Rte_LdCom_CbkTriggerTransmit_<sn> használatával. Miután az RTE megkapta a kérést, ellenőrzi, hogy az elérhető adat belefér-e a PDU Router által jelzett tárolóba (aminek maximális méretét a PduInfoPtr->SduLength-ben jelez). Ha belefér, átmásolja az adatokat abba a tárolóba, amit a PDU Router szolgáltat, majd frissíti az SduLength-et (azaz az adat hosszát jelző változót) a tényleges adathosszra. Ha nem fér bele, egy E_NOT_OK hibaüzenettel tér vissza anélkül, hogy megváltoztatná a PduInfoPtr-t.

Reception – Fogadás (Rx)

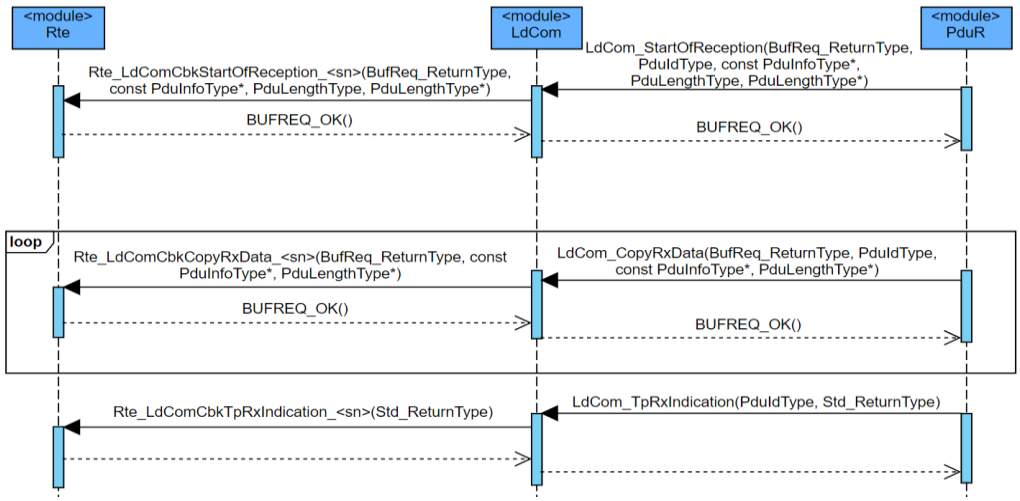
Az adatfogadás első lépése a PDU Router által hívott LdCom_StartOfReception függvény. Ez a függvény jelzi a kezdetét egy N-SDU fogadásának. Ez az N-SDU állhat egyetlen, vagy akár több N-PDU-ból is (szegmentált átvitel esetén). A Large Data COM megkapja ezt a kérést, amit az Rte_LdComCbkStartOfReception_<sn> hívásával továbbít. Ha nem történt hiba a kérés közben (Null-pointer, inicializálatlan modul stb.), a fogadás következő lépése következik.

Ez pedig az, hogy a PDU Router meghívja az LdCom_CopyRxData függvényt. Ez a függvény szolgáltatja az egy I-PDU szegmensből fogadott adatot az RTE számára. A függvény minden egyes hívása az adott I-PDU adat következő szegmensét tartalmazza. A fogadás addig tart, ameddig az összes adat át nem lett másolva a megfelelő tárolóba, vagy valami hiba miatt a folyamat leáll.

Három paramétere van:

- id: a fogadott I-PDU azonosítója
- info: az átviendő byte-ok számát és a kezdeti tárolót tartalmazó struktúra
- bufferSizePtr: a küldés utáni, még elérhető tárolóméret

A küldés oldali LdCom_CopyTxData-hoz hasonlóan ennek a függvénynek is BufReq_ReturnType a visszatérési értéke, mely a fogadás sikerességét jelzi. Ha hiba történt az adatmásolás közben, azt a visszatérési értékként beállított BUFREQ_E_NOT_OK értékkel lehet jelezni. A küldéssel szemben ebben az esetben ez az érték nem lehet BUFREQ_E_BUSY. A fogadás folyamatát jelzi a(z) 3.2. ábra:

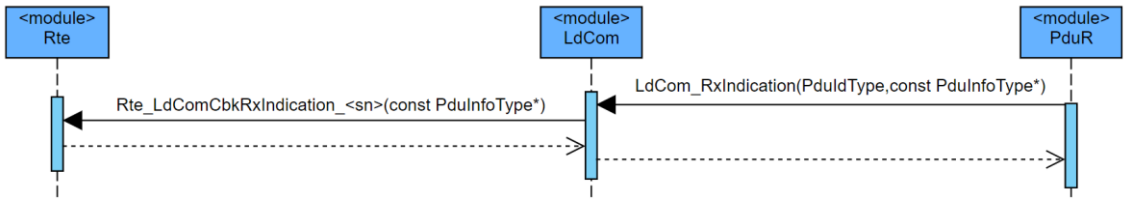


3.2 ábra. Adatfogadás LdCom_CopyRxData segítségével

Az adatok fogadása a már említett LdCom_StartOfReception függvénnyel kezdődik el, mely folyamat az RTE-hez közvetve eljutott kérés után, hiba nélküli esetet feltételezve az LdCom_CopyRxData API-val folytatódik. Ez a fogadás ciklusban történik az API egymás utáni meghívásaival.

Az adatfogadás végével a PDU Router visszajelzésként az LdCom_TpRxIndication függvényt hívja meg, melyben jelzi a folyamat sikerességét az RTE számára (természetesen ekkor is a Large Data COM van a közvetítő szerepben, az RTE már az LdCom-ból hívott Rte_LdComCbkTpRxIndication függvényhívást látja).

Ez érvényes a TP esetére, IF esetben más a folyamat, melyet a(z) 3.3. ábra szemléltet:



3.3 ábra. LdCom_RxIndication szekvenciadiagramja

Itt kizárólag a PDU Router hívja meg az LdCom_RxIndication függvényt, amely annak a jelzésére szolgál, hogy érkezett egy PDU egy alsóbb rétegű kommunikációs interfész modultól. A paraméterek tartalmazzák az érkezett PDU-val kapcsolatos információkat (azonosító, adat hossza, a tárolóra mutató pointer).

3.3. Dinamikus kód, kódgenerátor

Az AUTOSAR Basic Software modulok nagy része konfigurálható a vevői követelmények és a modult futtató mikrokontroller jellemzői szerint, elérve ezzel a széleskörű felhasználhatóságot. A modulok konfigurációját egy cégen belül fejlesztett modellező-eszközben lehet elvégezni, ahol szerepel az összes BSW modul definíciója. A kódgenerátor és a hozzá tartozó modellbejárás is Java nyelven íródott.

3.3.1. Modul definíció

Az AUTOSAR szabvány előre megszabja minden modul konfigurálhatósági tulajdonságait, ezeket a tartalmazza a modul definíciója.

A Large Data COM esetében ezek a konfigurációs beállítások a következők:

- LdComConfig: a modul konfigurált I-PDU-it tartalmazó konténer, multiplicitását tekintve 1...∞, azaz legalább egy ilyen konténernek lennie kell a konfigurációs beállításokban (az I-PDU-knak egyes paraméterei 0...1 multiplicitásúak, ami azt jelenti, hogy nem kell bekonfigurálni, de maximum egy értéke lehet)
- LdComGeneral: a modul általános konfigurációs paramétereit tartalmazó konténer, multiplicitása 1, azaz mindenképpen be kell állítani, de csak egyszer

A felsorolt konténerek az alábbi konfigurációs paramétereket tartalmazzák:

LdComGeneral

Konfigurációs paraméter	Multiplicitás	Leírás
LdComDevErrorDetect	1	Kapcsoló, mely a hibadetektálást és –jelentést engedélyezi/tiltja
LdComVersionInfoApi	1	Kapcsoló, amely aktiválja/deaktiválja az LdCom_GetVersionInfo API-t

LdComIPdu

Konfigurációs paraméter	Multiplicitás	Leírás
LdComApiType	1	A konfigurált I-PDU továbbítás típusa, értékei: IF / TP
LdComHandleId	1	Az I-PDU azonosítója, ami alapján továbbítani kell a megfelelő modulhoz
LdComIPduDirection	1	Az I-PDU továbbítás irányát adja meg, értékei: SEND / RECEIVE
LdComRxCopyRxData	0...1	Az alábbi meghívandó függvény: Rte_LdComCbkCopyRxData_<sn>

LdComRxIndication	0...1	Az alábbi meghívandó függvény: Rte_LdComCbkRxIndication_<sn>
LdComRxStartOfReception	0...1	Az alábbi meghívandó függvény: Rte_LdComCbkStartOfReception_<sn>
LdComTpRxIndication	0...1	Az alábbi meghívandó függvény: Rte_LdComCbkTpRxIndication_<sn>
LdComTpTxConfirmation	0...1	Az alábbi meghívandó függvény: Rte_LdComCbkTpTxConfirmation_<sn>
LdComTxConfirmation	0…1	Az alábbi meghívandó függvény: Rte_LdComCbkTxConfirmation_<sn>
LdComTxCopyTxData	0…1	Az alábbi meghívandó függvény: Rte_LdComCbkCopyTxData_<sn>
LdComTxTriggerTransmit	0…1	Az alábbi meghívandó függvény: Rte_LdComCbkTriggerTransmit_<sn>

3.3.2. Modellbejárás

Az AUTOSAR Architect definíció tanulmányozása és konfiguráció hozzáadása után a modellbejárást kellett elkészítenem annak érdekében, hogy a kódgenerátor tudjon generálni.

A modellbejárás lényegében egy olyan osztályon belüli szerkezet kialakítása, amely megfelel a definícióban meghatározott felépítésnek, vagy ahhoz hasonló. A lényege az, hogy egy könnyen kezelhető struktúra legyen kialakítva. Ez az I-PDU-k esetében a következőképpen néz ki:

```
private String name;
private String LdComApiType;
private Long LdComHandleId;
private String LdComIPduDirection;
private String LdComRxCopyRxData;
private String LdComRxIndication;
private String LdComRxStartOfReception;
private String LdComTpRxIndication;
private String LdComTpTxConfirmation;
private String LdComTxConfirmation;
private String LdComTxCopyTxData;
private String LdComTxTriggerTransmit;
```

Az ApiType és az IPduDirection is String-ként lett definiálva (enumeráció helyett), a könnyebb elérés érdekében. A Java nyelvnek megfelelően az osztályhoz tartoznak getter függvények is, melyek segítségével a privát változók elérhetőek az osztályon kívülről.

3.3.3. API mapping

A korábban említett, I-PDU-hoz tartozó olyan konfigurációs paraméterek, melyek multiplicitása 0..1, függvénypointer-es megoldást igényeltek. Ez azt jelenti, hogy fel kellett venni ehhez új típusokat. Ezek a típusok olyan pointerek, amelyek adott visszatérési értékű és paraméterű függvényekre mutatnak, így el tudom érni azt, hogy az I-PDU-khoz tartozó, a konfigurációban beállított nevű függvényeket meg lehessen hívni a modul API-jaiból. Ezekre a függvényekre mutatnak azok a függvénypointerek, amelyeket felvettem a struktúrában.

```
typedef BufReq_ReturnType (* Rte_LdComCbkCopyTxData)(const PduInfoType*, Ret-
ryInfoType*, PduLengthType*);
```

Ez egy olyan függvényre mutató függvénypointer, amely visszatérési értékének típusa BufReq_ReturnType, paraméterei pedig rendre const PduInfoType*, RetryInfoType* és PduLengthType* típusúak.

3.3.4. Struktúra létrehozása

Ahhoz, hogy a kódot tudjak generálni, a Java nyelvben íródott modellbejárás mellett implementálnom kellett a már korábban említett, C nyelvben írt struktúrákat is, amelyeket szintén a modul definíciója alapján kell meghatározni. Ezeket a struktúrákat az LdCom_Types.h fejlécfájlban definiáltam, melyek a következők:

```
typedef struct {
	boolean isTP;
	uint8 handleId;
	boolean isSend;
	Rte_LdComCbkCopyTxData copyTxData;
	Rte_LdComCbkTpTxConfirmation tptxConfirmation;
	Rte_LdComCbkRxIndication rxIndication;
	Rte_LdComCbkStartOfReception startOfReception;
	Rte_LdComCbkCopyRxData copyRxData;
	Rte_LdComCbkTpRxIndication tprxIndication;
	Rte_LdComCbkTriggerTransmit triggerTransmit;
	Rte_LdComCbkTxConfirmation txConfirmation;
}LdCom_IpduCfg;

typedef struct {
	LdCom_IpduCfg* IPdu;
	uint8 iPduCount;
}LdCom_ConfigType;
```

Ahogy a fenti kód is mutatja, megjelennek a függvénypointer típusú struktúra elemek is. Így például a konfigurációban beállított nevű CopyTxData függvényre copyTxData néven tudok hivatkozni.

Az LdCom_ConfigType tartalmazza az ahhoz a konfigurációhoz tartozó összes I-PDU-t, valamint azok számát. Ez a típus a teljesen különböző konfigurációkat hivatott ábrázolni.

3.3.5. Kódgenerátor

A modellbejárás és a struktúra elkészülte után elkezdtem írni a kódgenerátort. A kódgenerátor szintén Java nyelven íródott, feladata a felhasználói beállításnak megfelelő konfiguráció generálása, amelyet a modul inicializálására tudunk használni.

A kódgenerálás során két fájlt hozok létre, egy LdCom_Cfg.h nevű fejléc- és egy LdCom_Cfg.c nevű forrásfájlt.

A fejlécfájlba generálom ki a szükséges kapcsolókat (makrókat), az egyes konfigurációkra mutató extern jelzővel ellátott pointereket, valamint az I-PDU-khoz tartozó konfigurált nevű függvények definícióját.

A forrásfájlba az I-PDU konfigurációja során beállított értékeket generálom. Ezeket az I-PDU-kat összegyűjtöm különböző LdCom_ConfigType típusú, az adott konfigurációhoz tartozó összes I-PDU-t tartalmazó konténerekbe, melyekre beállítok külön-külön egy pointert. Ezek a pointerek azok, amelyeket a fejlécfájlban extern jelzővel vettem fel. Így érhető el a projekten belül a különböző fájlokból is a konfigurációs struktúra.

A generált forrásfájlhoz tartozó kódgenerátor kódjának részlete megtalálható a dokumentum függelékében.

3.4. Smoke Test

Egy modul implementálása után annak érdekében, hogy helyes működéséről bizonyosságot lehessen szerezni, tesztelni kell azt.

A tesztelés két módon történhet: a modul teljes tesztelésével, vagy a jóval kisebb volumenű smoke test megírásával. A teljes tesztelés alatt a modul API-jainak sok különböző értékű paraméterrel való meghívását értem, feltételezve annak tényleges helyes működéséről (kimerítő tesztelést nem tudunk végrehajtani). Ez egy nagyobb, különálló feladat, melyet általában az implementertől (azaz attól az embertől, aki a modult implementálta) különböző ember végez. Ebből kifolyólag a teljes modultesztelést nem csináltam meg.

A smoke test viszont az implementáláshoz tartozik. Ez a fajta tesztelés azt foglalja magába, hogy a modul összes API-ja egyetlen „paraméterlistával” van meghívva, ezzel ellenőrizve azt, hogy egyáltalán működik-e az adott függvény.

A modul tesztkönyvtárában található egy forrásfájl, amiben egy teszteset van. Ebben a tesztesetben vannak definiálva a függvényhívásokhoz szükséges típusú változók (érvényes vagy akár érvénytelen értékkel, ezzel a hibadetektálást és –kezelést is tesztelve), majd a tényleges függvényhívások. A függvények működésének ellenőrzésére hasznos, egyben egyszerű megoldás a „printf” használata, azaz az API kimenetének kiíratása a konzolra.

```
static void tcLdComSmokeTest(void) {
	PduIdType id = 1U;
	Std_ReturnType result = E_OK;
	Std_VersionInfoType version;

	LdCom_Init(LdComConfigPtr);
	LdCom_TpTxConfirmation(id, result);
	LdCom_GetVersionInfo(&version);
	printf("%x %x", version.vendorID, version.moduleID);
}
```

Ahogy a smoke test kódrészletéből is látható, a szükséges változók definiálása után a függvények meghívása található. Az LdCom_Init függvény hívásakor annak paramétereként átadott LdComConfigPtr a korábban említett, a .h konfigurált fejlécfájlban extern

jelzővel ellátott, a konfigurációra mutató pointer. A konzolra íratás segítségével lehet ellenőrizni, hogy a ’version’ változóba ténylegesen azok az értékek kerültek betöltésre, amelyek előre definiálva lettek.

3.5. Követelmények bejelölése, kód refaktorálása

A smoke test megírása, ezzel együtt a modul API-jainak ellenőrzése után az utolsó lépések egyike, a követelmények bejelölése. Ez ahhoz szükséges, hogy jelezzük a kódban, hogy a specifikációban található követelményeket hol valósítottuk meg. A modul tesztelésekor a teszternek ezek a követelmények alapján kell tesztelnie a modult anélkül, hogy akár a forrásfájlokat, akár a fejlécfájlokat megnézné (és ezek a követelmények alapján kell majd bejelölnie a tesztelt követelményeket, ezzel jelezve a tesztelés akkori, százalékos állapotát).

A követelmények bejelölése a következő formában történt:

```
/**
 * %a követelmény rövid leírása%
 * @reqimpl{%a követelmény specifikációban megadott azonosítója%}
 */
```

A követelmények azonosítója az SWS_LDCOM_00011 alakban volt megadva.

Végül már csak a kód refaktorálása maradt hátra. Ez a következő teendőket foglalta magában:

- konzolra íratások kitörlése a modul forrásfájljából
- kikommentezett kódsorok törlése
- a függvényekhez, azok elé dokumentáció írása (amely tartalmaz egy rövid leírást a működéséről és a paramétereiről)
- a megírt fájlok első soraiba a fájl fejlécének megírása (ez tartalmazza a fájl nevét, leírását, elkészítésének dátumát, valamint az implementer és a cég nevét)
- egyenlő távolságok hagyása a függvények között és az azokon belüli sorok esetében is a szép, rendezett kód érdekében

4. Összegzés

A félév során feladatom egy AUTOSAR Basic Software modul, pontosabban a Large Data COM megvalósítása volt. Ahogy már említettem, ez volt az első modul implementációm, így rengeteg dolog újként hatott. Nagy segítséget nyújtottak konzulenseim, illetve az, hogy útmutatóként szolgált a modul követelménylistája.

4.1. Értékelés

A szabvány által meghatározott formai és minőségi követelmények betartásával, valamint az implementációt az adott funkcionalitásokhoz tartozó leírással kiegészítve sikerült egy magas színvonalú kódot írnom, mely megállja a helyét a többi AUTOSAR modul mellett.

4.2. Továbbfejlesztési lehetőségek

Továbbfejlesztési lehetőségként meg lehet említeni az RTE-vel, illetve a CAN vagy FlexRay stack-kel való integrációt, valamint azt, hogy a szoftver céleszközön is ki legyen próbálva. Abból kifolyólag, hogy én smoke test-et írtam csak a modulhoz, jövőbeli célként fel lehet venni a teljes modultesztet is.

Irodalomjegyzék

[1] FPT Software, *What is AUTOSAR and why is it important?*, 2019, https://www.fpt-software.com/automotive-tech-blog/what-is-autosar-and-why-is-it-important/

[2] AUTOSAR szabványok, *Default Error Tracer*, 2019, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_DefaultErrorTracer.pdf

[3] Automotive Wiki, *Runtime Environment*, 2015, https://automotive.wiki/index.php/Runtime_Environment

[4] Bokor József, Gáspár Péter, *Járműfedélzeti kommunikáció*, Typotex kiadó, 2012

[5] National Instruments, *Introduction to the Local Interconnect Network (LIN) Bus*, 2019, https://www.ni.com/hu-hu/innovations/white-papers/09/introduction-to-the-local-interconnect-network--lin--bus.html

[6] AUTOSAR alapú autóipari szoftverrendszerek, *Basic Software Communication*, 2017, https://www.mit.bme.hu/system/files/oktatas/targyak/vedett/10727/02_bsw_communication_0.pdf

[7] Can Cia, *CAN FD – The basic idea*, https://www.can-cia.org/can-knowledge/can/can-fd/

[8] CSS Electronics, *Simple Intro to Can Bus*, 2019, https://www.csselectronics.com/screen/page/simple-intro-to-can-bus/language/en

[9] AUTOSAR, *Specification of Large Data COM*, 2019, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_LargeDataCOM.pdf

[10] AUTOSAR, *Specification of PDU Router, 2017*, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_PDURouter.pdf

[11] National Instruments, *FlexRay Automotive Communication Bus Overview*, 2019, https://www.ni.com/hu-hu/innovations/white-papers/06/flexray-automotive-communication-bus-overview.html

[12] UAVCAN, *CAN bus transport layer*, 2019, https://uavcan.org/Specification/4._CAN_bus_transport_layer/

Függelék

A generált forrásfájl kódgenerátorának egy részlete:

```
private void generateBody(StringBuilder buffer) {
	for (LdComConfig cfg : parsedConfig.getLdcomConfig()) {
		printLine(buffer, "LdCom_IpduCfg" + cfg.getName() + "[" + cfg.getLdCom-
IPduList().size() + "] = {");
		for (LdComIPdu pdu : cfg.getLdComIPduList()) {
			printLine(buffer, "   /* " + pdu.getName() + " */");
			printLine(buffer, "   {\n     /* isTP */");
			if (pdu.getLdComApiType().equals("LDCOM_TP")) {
				print(buffer, "     TRUE,\n");
			} else {
				print(buffer, "     FALSE,\n");
			}
			printLine(buffer, "     /* handleId */");
			print(buffer,"    "+ pdu.getLdComHandleId().toString() + "U,\n");
			printLine(buffer, "     /* isSend */");
			if (pdu.getLdComIPduDirection().equals("LDCOM_RECEIVE")){
				printLine(buffer, "     FALSE,");
			} else {
				printLine(buffer, "     TRUE,");
			}
			printLine(buffer, "     /* copyTxData */");
			if (pdu.getLdComTxCopyTxData() != null) {
				printLine(buffer,"    &"+pdu.getLdComTxCopyTxData()+", ");
			} else {
				printLine(buffer, "     NULL_PTR,");
			}
		}
	}
}
```