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

AUTOSAR FlexRay kommunikációs modulok megvalósítása

Szakdolgozat

Készítette
Faragó Dániel

Belső konzulens
Dr. Sujbert László
BME MIT

Külső konzulens
Dr. Pintér Gergely
ThyssenKrupp Presta Hungary Kft.

Budapest, 2012

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Faragó Dániel*, 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 autentikált 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.

Budapest, 2012. december 16.

Faragó Dániel
hallgató

Kivonat

A mai modern gépjárművek számos szolgáltatással rendelkeznek, melyek egyre több elektronikus vezérlőegység beépítését követelik meg. Biztonságkritikus funkciók esetén elengedhetetlen az egységek közötti megbízható kommunikáció. Ennek biztosítására fejlesztették ki az autóipari környezetben mára alapkövetelménnyé vált AUTOSAR rétegzett szoftverarchitektúrát, mely a különböző beszállítók által gyártott vezérlőegységek közti kommunikációt egységessé, az azokon futó alkalmazások számára pedig transzparenssé teszi.

A FlexRay protokoll redundanciájával és nagy átviteli sebességével a biztonságkritikus rendszerek egyre széleskörűbben alkalmazott kommunikációs protokollja.

Feladatom két – az AUTOSAR szabványban rögzített – szoftvermodul C nyelvű megvalósítása volt. A két modul a FlexRay Interface és a FlexRay ISO Transport Layer.

A dolgozat első fele a FlexRay és az AUTOSAR elméleti hátteréről szól. Az AUTOSAR esetében különös figyelmet fordítok a szabványrendszer kommunikációval kapcsolatos részleteire. A szoftverkomponensek együttműködését egy példán keresztül is bemutatom.

A következő fejezetekben a kapcsolódó szabvány összefoglalását követően a modulok megvalósításáról írok. A FlexRay Interface specifikációjának bemutatása után, ismertetem az általam tervezett konfigurációs és futásidejű adatstruktúrákat, megvizsgálom a konfiguráció függvényében elvégezhető optimalizációs lehetőségeket, és ismertetem a működést megvalósító függvényeket és algoritmusokat. A fejezet végén becslést adok a modul memória és futásidőbeli erőforrásigényére.

A FlexRay ISO Transport Layer esetében a modul által megvalósított protokollt definiáló ISO szabványt is ismertetem, majd az Interface-hez hasonlóan bemutatom a konfigurációs és futásidejű adatszerkezeteket. Az implementált szoftver működését – a protokoll összetettsége miatt – az adatküldés folyamatán keresztül írom le. Ezt a fejezetet is az erőforrásigény becslésével zárom.

A dolgozat utolsó fejezetében a tesztelés elméletéről, majd a modulok tesztelésének módszeréről írok. Ennek első lépéseként definiálok néhány a teszteléssel kapcsolatos fontos fogalmat, és ismertetem a tesztelés menetét. Ezután a FlexRay Interface egy függvényének példáján bemutatok egy egyszerű tesztkészletet.

A dolgozatot az elért eredmények, valamint a továbbiakban elvégzendő teendők összefoglalásával zárom.

Abstract

Today's modern vehicles have several extended services, which requires the incorporations of more and more electronic control units. In case of safety critical features, it is indispensable to provide reliable communication between the units. To ensure that, the AUTOSAR Layered Software Architecture was developed, which became already basic requirement in the automotive environment for today. It's aim to make the communication unified between units and transparent for the applications running on them.

The FlexRay protocol with its redundancy and high transfer rate is getting the most commonly applied communication protocol of safety critical systems.

My task is to implement two software modules – defined in the AUTOSAR standard – in C programming language. The modules are the FlexRay Interface and the FlexRay ISO Transport Layer.

The first part of my thesis is about the theoretical background of FlexRay and AUTOSAR. In case of AUTOSAR I pay special attention to the details of the communication defined in the standard. Cooperation between software modules are also presented through an example.

Next chapters are – after the summary of the related standard – about the implementation of the modules. After the overview of the FlexRay Interface's main tasks, I present the configuration and runtime data structures designed by me. I examine the optimization possibilities depending on configuration, and present the algorithms and functions realizing the modules functionality. Ending of the chapter I estimate the memory and runtime overhead of the software.

In case of the FlexRay ISO Transport Layer the ISO standard – which defines the protocol realized by the module – is also introduced, then similarly to the Interface I present the configuration and runtime data structures. The operation of the implemented software – because of the complexity of the protocol – will be presented through the method of sending data. This chapter will be again closed by estimation of resource demands.

In the last chapter I write about the theory and the method of software testing. The first step of it is to define some important expression in connection with testing, then I explain some simple test cases on a function of the FlexRay Interface.

I close the thesis with the achieved results and the summary of future actions to be carried out.

Bevezető

Ebben a fejezetben a feladatkiírás értelmezéséhez szükséges legfontosabb információkat foglalom össze. Először röviden bemutatom a napjainkra jellemző kommunikációs hálózatok fontos jellemzőit és az AUTOSAR által is támogatott, legelterjedtebben alkalmazott protokollok fontos tulajdonságait. Ismertetem az AUTOSAR célkitűzéseit, a konzorcium létrejöttének okát és alapelveit. A bevezető végén pontosítom a feladatkiírást, és megfogalmazom az elérendő célokat.

Napjaink autóipari kommunikációs hálózatai

A nyolcvanas években az autóipar jelentős elektronizálása indult meg. Kezdetben az autókban egy, legfeljebb két-három, egymástól függetlenül működő vezérlőegység kapott helyet. Később a járművekkel szemben támasztott egyre magasabb követelmények (komfort, biztonság, fogyasztás...) egyre több ECU (Electronic Control Unit – elektronikus vezérlőegység) beépítését tették szükségessé. A fejlesztők hamar felismerték, hogy a több, különálló részfeladatokat megvalósító rendszer összekötésével, egy az erőforrásokat jobban kihasználó elosztott rendszer jön létre, mely sokkal magasabb színvonalon képes megfelelni az elvárásoknak. Az először alkalmazott pont-pont alapú összeköttetéseken alapuló hálózatokat a vezetékezési költség növekedésével egyre inkább a közös buszt használó hálózatok váltották fel. Megjelentek, majd szabványokká váltak a különböző, kifejezetten autóipari célra kifejlesztett kommunikációs megoldások.

Esemény- és idővezérelt kommunikációs hálózatok

Az első időkben még viszonylag kisszámú és egyszerű üzenet továbbítására volt szükség, melyek általában valamilyen esemény bekövetkeztéről tájékoztatták az autó vezérlőegységeit. Ez az igény vezetett el az eseményvezérelt, vagy „igény alapú" kommunikációs protokollok kialakulásához, melyeknek legfőbb jellemzője, hogy bármelyik kommunikációs eszköz bármikor megpróbálhat adást kezdeményezni a hálózaton. Ehhez mindenképpen szükséges a csomagütközések elkerülése – vagy legalábbis detektálása – és az ebből fakadó hibák kezelése.

A két legelterjedtebben alkalmazott eseményvezérelt MAC (Media Access Control – az ISO/OSI modell adatkapcsolati rétegének közeghozzáférési alrétege) protokoll a CSMA/CD (Carrier Sense Multiple Access with Collision Detection – a csatorna figyelésén alapuló többszörös hozzáférés, ütközés detektálással) és a CSMA/CA (Carrier Sense Multiple Ac-

cess with Collision Avoidance – a csatorna figyelésén alapuló többszörös hozzáférés, ütközés elkerüléssel) protokoll. Mindkét megoldás figyeli a közös csatornát, és csak akkor próbál forgalmazásba kezdeni, ha szabadnak látja azt. A különbség a kettő között abban az esetben jelentkezik, ha egy időben két végpont is szabadnak találja a buszt, és adásba kezd. Az előbbi megvalósítás esetén az egyszerre adók észlelik a keretütközést, majd többnyire véletlen ideig tartó várakozást követően újból megkísérlik az adást. Az utóbbi protokoll úgy kerüli el az üzenetek sérülését, hogy egyértelmű prioritásokat rendel hozzájuk, és úgy alakítja ki azokat, hogy ütközés esetén a magasabb prioritású jusson érvényre, az alacsonyabb prioritású üzenet újraküldését kikényszerítve.

Az eseményvezérelt kommunikáció előnye az eseményekre való gyors, aszinkron reakció. Sajnos azonban a CSMA/CA jellegű protokollok prioritásossá teszik az üzeneteket, ennek következtében végülis romlik a hálózat kiszámíthatósága. Ez a hatás gyengén terhelt hálózat esetén még alig észrevehető, azonban a terhelés növekedésével egyre jelentősebbé válik. Szintén nagy előnye a CSMA protokollokon alapuló hálózatoknak az egyszerű skálázhatóság, hiszen – bizonyos határokig – a hálózat bővítése nem igényli annak át- esetleg újratervezését. Az AUTOSAR által támogatott eseményvezérelt protokollok a CAN (Controller Area Network), a LIN (Local Interconnect Network) és az Ethernet protokoll.

A gépjárművek által nyújtott szolgáltatások számának növekedése magával vonta az üzenetek számának növekedését is, növelve a buszon bonyolítandó forgalmat. A CSMA-n alapuló protokollok többé nem garantálták, hogy minden üzenet időben megérkezik a célhoz. Egyre gyakrabban fordult elő az alacsony prioritású üzenetek kiéheztetése. A különböző biztonságkritikus és valósidejű x-by-wire[1] technológiák megbízható kommunikációt igényelnek, ahol a csomagok biztosan megérkeznek a célhoz a kellő időben. Elengedhetetlenné vált a kommunikáció időzítéseinek statikussá tétele. A megoldást az idővezérelt hálózatok alkalmazása jelenti.

Ebben az esetben a közeghozzásférést általában valamilyen TDMA (Time Division Multiple Access – időosztásos többszörös hozzáférés) alapú protokoll vezérli, így az adók többé már nem egymással versengve, hanem előre meghatározott, ciklikusan ismétlődő időszeletekben férnek hozzá a buszhoz. Az előre meghatározott időszeletekben a végpontoknak kizárólagos joga van a csatorna használatára. Ezzel a megoldással nagyobb megbízhatóság érhető el, és az ütközések és újraküldések megszűnésével lehetővé válik a busz teljes sávszélességében való kihasználása.

Hátrány azonban a hálózat rugalmatlansága. Míg az eseményvezérelt hálózatoknál egy-egy új végpont rendszerbe illesztése egyszerűen megtehető volt, az idővezérelt hálózatoknál adott esetben ehhez a teljes rendszer időzítéseit újra kell tervezni. Mivel általában egyszerre van szükség az eseményvezérelt kommunikációt jellemző gyors reakcióidőre és az idővezérelt hálózatok megbízhatóságára, így gyakran a különböző protokollokat együttesen alkalmazzák. A biztonságkritikus egységeket összekötő – általában idővezérelt – fő hálózatra csatlakoznak a kisebb kiterjedésű, helyi, főleg eseményvezérelt alhálózatok.

Az AUTOSAR jelenlegi – e dolgozat alapját is képező – 4.0-s verziója, az eseményvezérelt

[1] A jármű irányításában alkalmazott mechanikus vagy hidraulikus technológiák kiváltása elektromechanikus megoldásokkal, a mechanikus összeköttetés megszüntetésével.

CAN, LIN és Ethernet, valamint az idővezérelt FlexRay és TTCAN (Time Triggered CAN – idővezérelt CAN) protokollokat támogatja.

CAN

Az első, kifejezetten autóipari célra kifejlesztett kommunikációs protokoll a CAN protokoll volt. Fejlesztése 1983-ban kezdődött, a Robert Bosch GmbH.-nál. Az 1.0-s első kiadás 1986-ban látott napvilágot, míg a ma legelterjedtebben használt 2.0-s verziószámú specifikáció publikálására 1991-ben került sor. A CAN protokoll 1993-ban lett ISO szabvány.

A maximálisan 1 Mbps sebességre képes CAN közeghozzáférési rétege CSMA/CA protokollt használ. A keretek első 11 bitje – vagy kiterjesztett keretek esetén az első 29 – az üzenet azonosítója. A busz domináns állapota a logikai 0 értékű, míg a recesszív állapota a logikai 1 értékű bitnek felel meg. Egyszerre adás esetén, ha egy keret sérül, az adó megszakítja a küldést. Mivel mindig a domináns bit jut érvényre, az a fél veszti el az arbitrációt amelyik először küld recesszív bitet, így az üzenet azonosítója, egyben az üzenet prioritását is megszabja. Az alacsonyabb üzenet-azonosító nagyobb prioritást jelent.

A CAN szabvány máig a legelterjedtebben alkalmazott ipari kommunikációs protokoll. A járműveken kívül – ahol szinte minden részegységben megtalálható (váltás- és motorvezérlés, központi zár, riasztó, fényszórók vezérlése, multimédia...) – kiépítésének alacsony költsége miatt – elterjedten használják az egyéb ipari irányítás- és vezérléstechnikai alkalmazásokban is.

LIN

Az 1990-ben autógyártókból létrejött konzorcium fejlesztette ki az olyan egyszerű, szenzorokból és beavatkozókból álló hálózatok kiszolgálására, ahol a CAN kiépítése felesleges és túl költséges lett volna. (Például: központi zárak, elektromos ablakemelők, ülés-, tükör- és szélvédőfűtés vezérlése...) A nem szigorúan eseményvezérelt, master-slave felépítésű hálózat maximálisan 20 kbps átviteli sebességre képes. Egy master és 16 slave kapcsolódhat egymáshoz. Mivel kizárólag a master kezdeményezhet üzenetküldést, nincs szükség csatornafigyelő protokollokra. Az üzenet minden slave-hez eljut, majd ha szükséges, bizonyos slave-ek válaszolhatnak rá. A master általában valamilyen egyszerűbb mikrokontroller, a slave-ek pedig egyszerű, általában ASIC-ekkel (Application-Specific Integrated Circuit – alkalmazás-specifikus integrált áramkör) megvalósított áramkörök.

Ethernet

Az eredetileg számítógép hálózatok felépítésére kifejlesztett Ethernet protokollt alacsony kiépítési költsége miatt egyre gyakrabban alkalmazzák az autóiparban is. CSMA/CD protokollt alkalmaz az ütközések feloldására. Magas átviteli sebessége miatt elsősorban a különböző multimédiás és komfortszolgáltatásokat ellátó rendszerek kommunikációs protokolljaként használják.

TTCAN

A TTCAN (idővezérelt CAN) protokollt az ISO 11894-4 szabvány specifikálja. A protokoll egy, a CAN fizikai és adatkapcsolati rétegére épülő, kiegészítő rétegben valósul meg. Azokban az esetekben alkalmazzák, amikor az arbitrációt használó CAN protokollnál megbízhatóbb, determinisztikusabb, vagy csak egyszerűen ciklikus adatátvitelre van szükség. A későbbiekben bemutatásra kerülő FlexRay-el szemben viszont nem képes redundáns átvitelre és sávszélessége is sokkal alacsonyabb.

FlexRay

A FlexRay protokoll redundanciájával és magas átviteli sebességével a biztonságkritikus rendszerek egyre széleskörűbben alkalmazott kommunikációs protokollja. Lehetőséget biztosít ciklikus üzenetek teljesen idővezérelt és események által triggerelt üzenetek dinamikus átvitelére is. A protokollt a későbbiekben részletesen ismertetem.

Az AUTOSAR

Az AUTOSAR egy fejlesztői együttműködés révén jött létre 2003-ban. A név egy betűszó, mely az AUTomotive Open System ARchitecture (autóipari nyílt rendszer-architektúra) szavak kezdőbetűiből áll össze. Az alapító cégek (BMW, Daimler, Bosch, Continental, Volkswagen) egy olyan szoftverarchitektúra kidolgozását tűzték ki célul mely,

- hatékonyan és megbízhatóan képes kezelni a napjaink autóiparára jellemző bonyolult elosztott rendszereket,
- támogatja az iparra jellemző generikus szoftverekből származtatott variánsok kezelését,
- jól definiált működésének köszönhetően minimalizálja az előforduló hibák számát,
- jól elkülönített moduljai révén a különböző módosítások és frissítések mellékhatásai minimálisak,
- moduláris, skálázható, hordozható és újrahasznosítható kódot eredményez. [1]

A konzorcium 2011. decemberi adatok szerint 9 „mag” és több mint 150 prémium, fejlesztő és közreműködő taggal rendelkezik. [1]

Az architektúra alkalmazása leginkább a következő tulajdonságokkal rendelkező vezérlőegységek számára nyújt előnyöket:

- Erős kapcsolatra van szükség a hardver és a szoftver között (szenzorok, beavatkozók).
- Járműipari kommunikációs hálózatokhoz kapcsolódik.
- 16 vagy 32 bites mikrovezérlőkkel rendelkezik, csekély tár és számolási kapacitással.
- A program végrehajtása külső vagy belső, általában flash memóriából történik. [2]

Az architektúra réteges felépítésű. Az egyes rétegek szigorúan definiált interfészeken keresztül kapcsolódnak egymáshoz. A szervezet – „Cooperate on standards, compete on implementation!”[1] – jelmondatának megfelelően, szintén szigorú megkötések vonatkoznak a rétegek által megvalósított működésre, azonban az implementáció teljes egészében a fejlesztőre van bízva. A projekt – jelenlegi – harmadik fázisában került publikálásra az ezen dolgozat alapját is képező AUTOSAR 4.0 R3 szabvány. Az AUTOAR 4.0 részletes ismertetésére a későbbiekben kerül sor.

A feladat megfogalmazása

Szakdolgozatomat a ThyssenKrupp Presta Hungary Kft.-nél végeztem. A vállalat 1999 óta foglalkozik személyautók elektromos kormányrendszerének fejlesztésével. A kormányrendszer vezérlőegységén futó szoftver – az autóiparban elvárásnak számító – AUTOSAR szabvány szerint épül fel.

Feladatom az AUTOSAR rétegzett szoftverarchitektúrájának gerincét képező Alapszoftver Réteg (Basic Software Layer - BSW) két, FlexRay kommunikációval kapcsolatos szoftvermoduljának megvalósítása és az ehhez kapcsolódó szabvány ismertetése. A megvalósítandó modulok a FlexRay Interface, és a FlexRay ISO Transport Layer.

A dolgozat első fejezetében feladat megoldásához szükséges alapvető elméleti ismereteket foglalom össze. Itt kerül részletes bemutatásra a FlexRay protokoll, valamint az AUTOSAR rétegzett szoftverarchitektúrája.

A második fejezetben az AUTOSAR kommunikációt megvalósító szoftvermoduljainak részletes bemutatása után, egy példán keresztül érthetőbbé teszem azok egymáshoz kapcsolódását.

A harmadik és negyedik fejezetben a két megvalósítandó szoftvermodul általam tervezett adatszerkezetei, valamint algoritmusai kerülnek bemutatásra. Ennek előzményeként mindkét esetben ismertetem az AUTOSAR szabvány modulokkal kapcsolatos előírásait. A FlexRay ISO Transport Layer modul által megvalósított protokollt az ISO 10681-2 szabvány definiálja, így e modul esetében elengedhetetlen az ISO szabvány részletes ismertetése is. Mivel a vonatkozó szabványok szigorúan definiálják a modulok feladatait, igyekszem a szoftverek működését olyan részletességgel bemutatni, hogy jól elkülöníthetővé váljanak az előírások és a tervezési döntéseim. Az egyes, implementációt bemutató fejezetek végén becslést adok a szoftvermodulok tár és futásidőbeli erőforrásigényeire.

Az utolsó, ötödik fejezetben a szoftvertesztelés elméletéről írt rövid összefoglaló után, egy példán keresztül bemutatom a szoftvermodulok tesztelésének módszerét. Mivel időközben kiderült, hogy az AUTOSAR konzorcium – a jelenlegi, 4.0.3 verziótól kezdődően – nem folytatja tovább a TTCN nyelven írt tesztkészletek publikálását, a szoftverek tesztelése a feladatkiírástól eltérően a cég által választott CUnit környezetben zajlik. Ez egyben azt is jelenti, hogy a konzorcium által elkészített tesztek egyszerű futtatása helyett, magam által írt részletes tesztek kidolgozására van szükség. Ebből kifolyólag az ötödik fejezetben a tesztelés menetét egy egyszerű függvény példáján keresztül mutatom be.

[1] „Működj együtt a szabványban, versenyezz az implementációban!”

1. fejezet

Elméleti háttér

Ebben a fejezetben a téma megértéséhez elengedhetetlen elméleti ismereteket foglalom össze. Először részletesen ismertetem a FlexRay protokoll lényeges tulajdonságait, majd áttekintem az AUTOSAR rétegzett szoftver architektúrájának főbb jellemzőit.

1.1. A FlexRay 3.0.1

Ebben az alfejezetben a FlexRay protokoll témámhoz kapcsolódó részleteit ismertetem. A protokoll szinkronizációval és Network Management-el kapcsolatos részleteit nem érintem, mivel ezek ismerete nem szükséges a dolgozat megértéséhez.

A feladatom megvalósításának alapját képező AUTOSAR 4.0 szabvány a FlexRay protokoll 3.0.1-es verzióját támogatja, így a protokoll bemutatása során én is ezt vettem alapul. A fejezet a [3] és [4] dokumentumok alapján íródott.

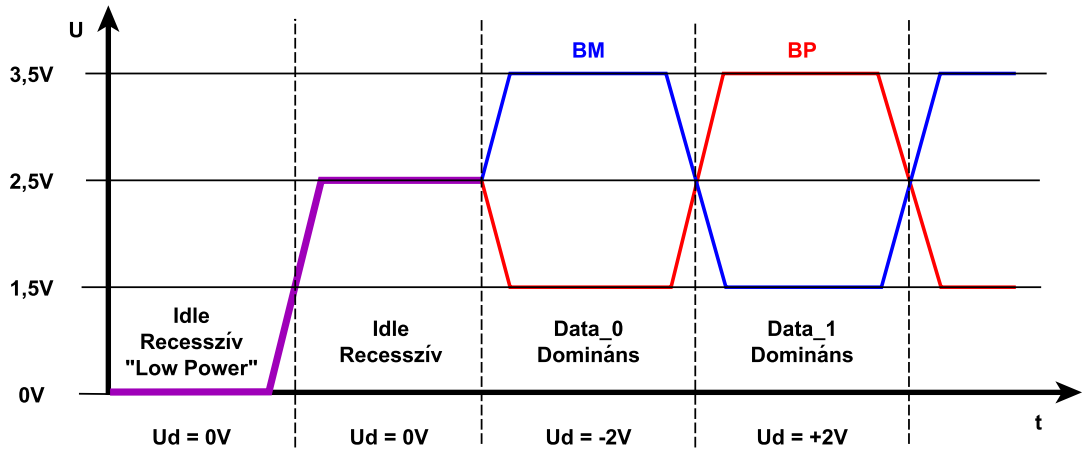
1.1.1. Fizikai réteg

A FlexRay buszon a kommunikáció árnyékolás nélküli, csavart érpárú, UTP (Unshielded Twisted Pair) vezetékeken megy végbe. Habár ez a megoldás a legolcsóbb, és az érpárok megcsavarása az autóipari környezetben leggyakrabban előforduló induktív csatolású (differenciális módusú) zavarfeszültségek ellen jól véd, az árnyékolás hiánya miatt a rendszer továbbra is védtelen marad a kapacitív zavarok ellen.

A FlexRay a kapacitív csatolású (közösmódusú) zavarok csökkentésének érdekében a két ér között értelmezett differenciális U_d feszültségekkel viszi át az információt. A logikai „1” állapotban pozitívabb potenciálon levő ér neve BP (Bus Plus), míg a logikai „1” állapotban negatívabb potenciálon levőé BM (Bus Minus). A busznak az U_d feszültség értéke szerint három állapotát különböztetjük meg *(1.1. ábra)*:

Idle: $U_d = 0V$

Ez a busz üresjárási, recesszív állapota. Ebben az állapotban nem folyik áram a BP és BM vezetékek között. Abban az esetben, ha a buszra csatlakoztatott minden végpont alacsony fogyasztású („Low Power”) módban van, U_P és U_M értéke is egyaránt 0V. Ha a buszon egyik node (végpont) sincs alacsony fogyasztású módban, U_P és U_M



1.1. ábra. *FlexRay jelszintek*

értéke 2,5V. Minden más esetben U_P és U_M valahol 0V és 2,5V között helyezkedik el, de az $U_P = U_M$ feltétel mindenképpen teljesül.

Data_1: $U_d = +2V$
Ez a feszültségszint képviseli a logikai „1” értéket. Ebben az esetben U_P értéke 3,5V, míg U_M értéke 1,5V. Ez az állapot aktívan meghajtott, domináns állapot.

Data_0: $U_d = -2V$
Ez a feszültségszint képviseli a logikai „0” értéket. Ebben az esetben a U_P értéke 1,5V, míg U_M értéke 3,5V. Ez az állapot aktívan meghajtott, domináns állapot.

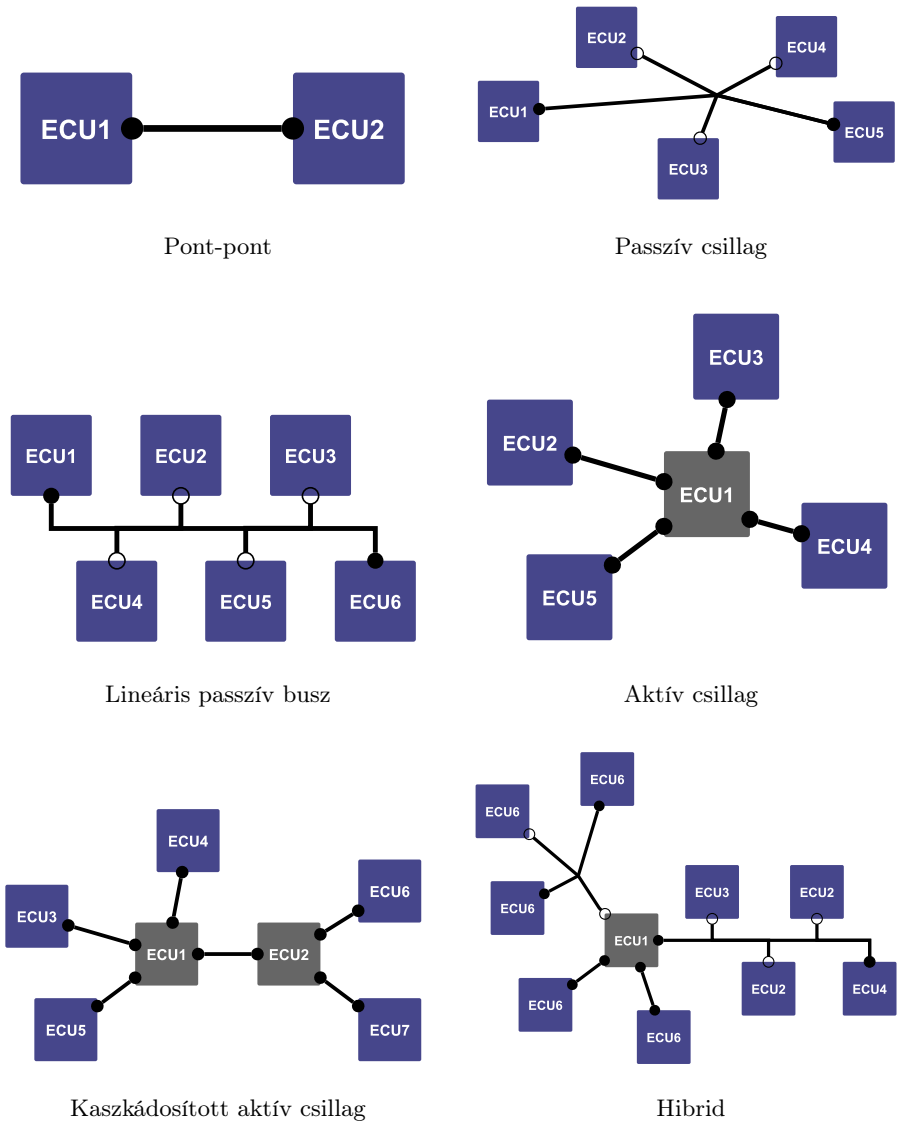
Ha a busz recesszív állapotban van, bármelyik egység Data_0 vagy Data_1 (domináns) állapotba hozhatja azt, a megfelelő U_d feszültség BP és BM közé kényszerítésével.

A véges jelterjedési idők és a magas átviteli sebesség következtében nem elhanyagolható zavarforrás a reflexió jelensége sem. Ennek csökkentésére a szabvány lezáróellenállások használatát javasolja. A specifikáció szerint a teljes busznak – DC (egyenáramú) szempontból – 40-55Ω-os terhelést kell jelentenie, így a lezáróimpedanciák nagyságát nagyjából 80Ω és 110Ω közé kell választani. A kábel hullámimpedanciájának pedig, ennek megfelelően, szintén 80Ω és 110Ω között kell lennie.

1.1.2. Topológiák

A FlexRay a kommunikációra két, egymástól független csatornát használ, csatornánként 10 Mbit/s-os sebességgel, így lehetőségünk nyílik az adatokat egyszerre két csatornán is (redundánsan) elküldeni, ezzel növelve a hálózat hibatűrő képességét. Kevésbé biztonságkritikus üzenetek esetén a két csatorna felhasználható a sávszélesség növelésére is, ezzel 20 Mbit/s sebességű kommunikációt biztosítva. Az egymással fizikai kapcsolatban álló végpontok úgynevezett clustert alkotnak. A két csatornán egymástól teljesen független topológiák

alakíthatók ki *(1.2. ábra)* (az ábrákon a csatlakozás pontjánál a satírozott kör a lezáróellenállások tipikus elhelyezését mutatja):



1.2. ábra. *FlexRay topológiák*

Pont-pont

Ez a legegyszerűbb topológia. Két ECU alkotja, melyek közvetlenül össze vannak kötve egymással. A buszt a két végén kell lezárni.

Passzív csillag

Ebben a topológiában több ECU egy közös, passzív csomóponton keresztül galvanikus kapcsolatban áll egymással. A lezárást a két egymástól legtávolabbi ECU-nál ajánlott elhelyezni.

Lineáris passzív busz

A lineáris passzív busz a passzív csillaghoz hasonló topológia, csupán annyiban különbözik tőle, hogy itt nem egy koncentrált csomóponton keresztül vannak összekötve az egységek, hanem egymás után, egy vezeték mentén „elnyújtott", busz jellegű csomópontra csatlakoznak. A lezárást a busz két végén szokás elhelyezni.

Aktív csillag

Ez a topológia lényegében több pont-pont összeköttetésből áll, melynek középpontjában egy gateway-nek (átjárónak) nevezett ECU áll. A többi egység csak ezen keresztül létesíthet egymással kapcsolatot. A lezárást – a pont-pont összeköttetéshez hasonlóan – minden ECU-nál szükséges elhelyezni.

Kaszkádba kötött aktív csillag

A kaszkádosított aktív csillag két aktív csillag topológiából áll, ahol a két gateway összeköttetésben van egymással. Így a két aktív csillag ECU-i a gateway-eken keresztül kapcsolatot tudnak létesíteni egymással. A lezárásokat az aktív csillaghoz hasonlóan minden vezeték két végén el kell helyezni.

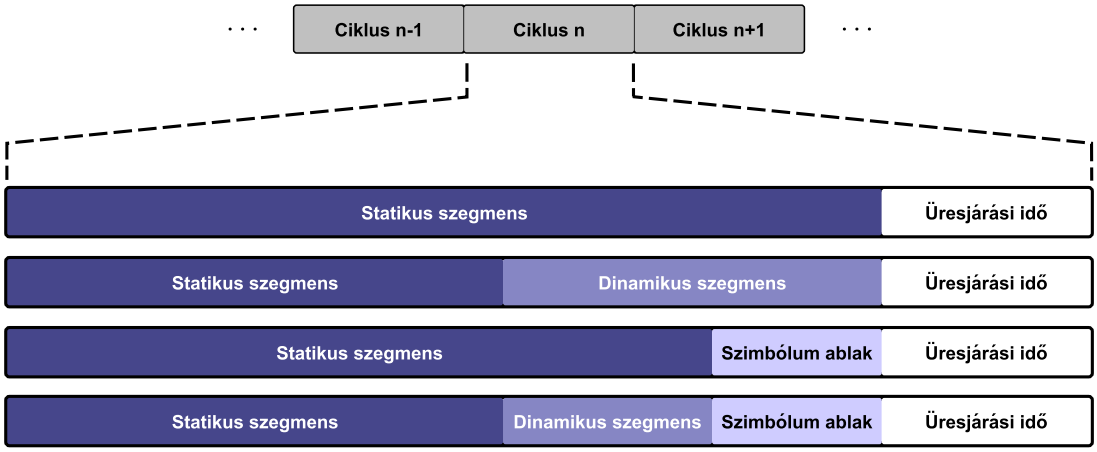
Hibrid

A hibrid topológia az eddig tárgyalt topológiák összekötéséből jön létre. Az ábrán egy aktív csillag látható, mely a többi egyéb topológia alapján felépített hálózatot köti össze. A lezárás az egyes alhálózatokra jellemző módon helyezendő el.

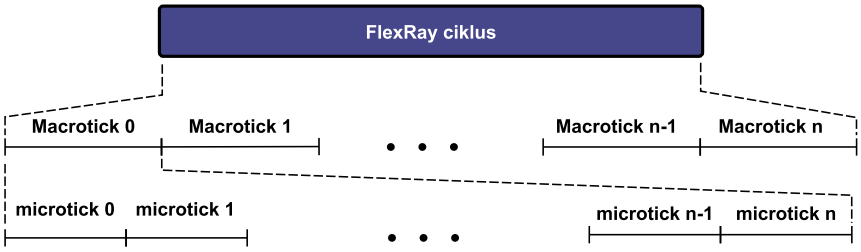
1.1.3. A FlexRay protokoll

A FlexRay a buszhozzáférés ütemezését a determinisztikusságot megkövetelő üzenetek esetén TDMA protokoll, az eseményvezérelt üzenetek esetén pedig FTDMA (Flexible TDMA – rugalmas TDMA) protokollok együttes alkalmazásával oldja meg. Ez szigorú időzítéseket követel. A FlexRay ütemezése egymást követő, ismétlődő ciklusokra van osztva. A ciklusok további egységekből épülnek fel.

Egy FlexRay ciklus mindig tartalmaz statikus szegmenst és egy NIT-nek (Network Idle Time – hálózati üresjárási idő) nevezett üresjárási szegmenst. Ez kiegészülhet igény szerint még a dinamikus szegmenssel és a szimbólum ablakkal, melyek ebben a sorrendben a statikus szegmens és a NIT között foglalnak helyet. Annak érdekében, hogy a node-ok különböző hardverfelépítéséből adódó időmérési különbségek a megengedhető hibahatár alatt maradjanak, a FlexRay ciklus úgynevezett macrotick-ekre van osztva. A macrotick-ek mérete (bizonyos toleranciával) meg kell egyezzen a különböző végpontoknál. A macrotick-ek további egységekre, microtick-ekre vannak osztva, melyek már az adott ECU órája által mérhető legkisebb időegységet reprezentálják. Ebből adódóan egy microtick mérete vezérlőnként eltérő lehet.



1.3. ábra. *A FlexRay ciklus felépítése*

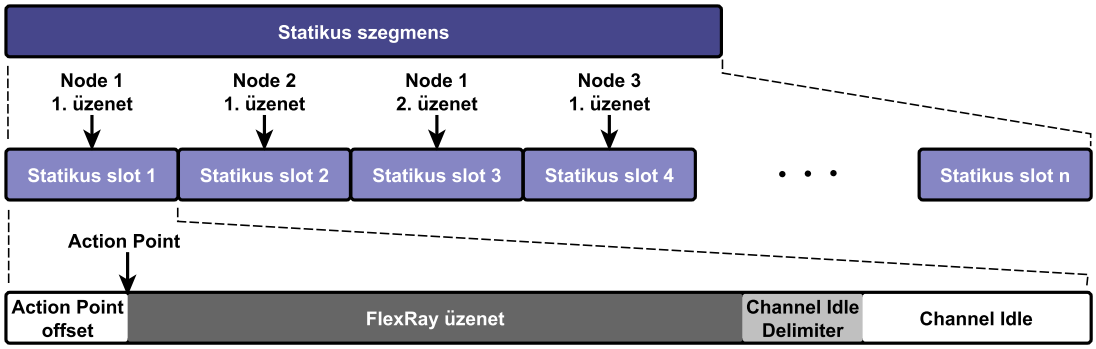


1.4. ábra. *Időegységek*

Statikus szegmens (Static Segment)

Determinisztikus adatátvitelre a statikus szegmensben kerül sor. Ebben a tartományban a közeghozzáférést TDMA protokoll ütemezi. A szegmens további (minimum 2, maximum 1023 darab) azonos nagyságú időrésre, úgynevezett statikus slotra van osztva, melyek mindegyike egy-egy végponthoz van rendelve. Minden slotban, az adott végpont kap kizárólagos jogot a buszhozzáféréshez, így biztosítva a statikus üzenetek periodikus továbbítását. Ennek megvalósításához minden vezérlő saját, helyi számlálóval rendelkezik, melyek minden statikus slot elején inkrementálódnak. Minden számlálóérték egy adott node számára egy adott üzenet elküldésének időpontját jelzi. Ez a hozzárendelés természetesen az A és B csatornát tekintve eltérő lehet, így biztosítva adott esetben a nagyobb biztonságot redundáns adatküldéssel, vagy a nagyobb sávszélességet egy időben különböző üzenetek átvitelével.

A slotok mérete úgy van meghatározva, hogy a legnagyobb megengedhető időeltérést és jelkésleltetést is figyelembe véve beleférjen a leghosszabb elküldhető üzenet. Az egyes slotok további alapegységekre vannak osztva. A slot egy „Action Point Offset” elnevezésű offszettel kezdődik. Ezt követi a nevét is adó „Action Point”, az a pont, ahol a tényleges üzenetküldés elkezdődik. Az üzenet átvitelét a 11 recesszív bitet tartalmazó „Channel Idle Delimiter” mező követi. Az utolsó szakasz a „Channel Idle”, mely a busz üresjárati állapota a következő statikus slotig. Az Action Point Offset idejének és a Channel Idle idejének



1.5. ábra. *A statikus szegmens felépítése*

összege minden esetben állandó, arányukat az aktuális késleltetés határozza meg.

Dinamikus szegmens (Dynamic Segment)

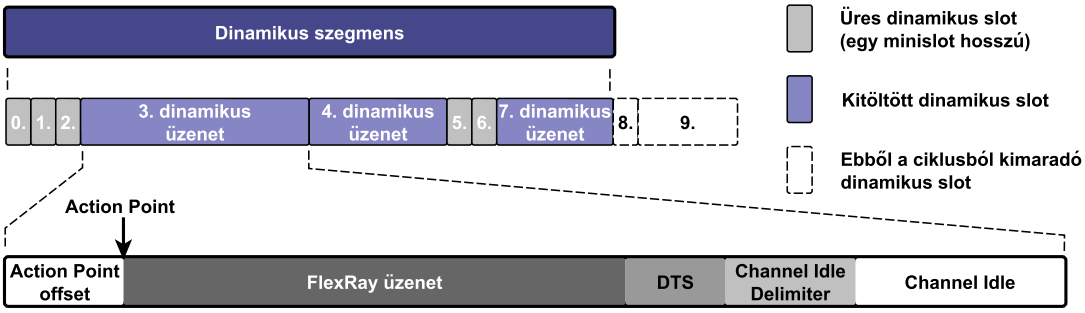
Ebben a szegmensben valósul meg – ha szükséges – az adatok eseményvezérelt átvitele az FTDMA protokoll elvei alapján. Ez a protokoll továbbra is idővezéreltnek tekinthető, de a szigorú TDMA-nál engedékenyebb:

- Nem rögzíti az üzenetek elküldésének pontos idejét, csupán sorrendjét.
- Nem teszi kötelezővé minden ciklusban az ebbe a szegmensbe ütemezett összes üzenet elküldését, csak azokat, melyeket egy a legutóbbi ciklus óta bekövetkezett esemény indokol.

A dinamikus szegmens a statikus szegmenshez hasonlóan további egységekre van osztva. A legkisebb ilyen egység a minislot. Ezek mérete minden ciklusban azonos. A különböző végpontok üzenetei egy-egy dinamikus slothoz vannak rendelve, melyek hossza minimum egy minislot hosszúságú. Ezekben az ablakokban történhet (ha szükséges) az adatok továbbítása. A szegmens elején a végpontok – a statikus szegmenshez hasonlóan – elkezdik számolni a dinamikus slotokat. Amikor egy node az egyik hozzá kijelölt slothoz ér, megvizsgálja, hogy rendelkezésre áll-e az ehhez a slothoz rendelt üzenetben továbbítandó adat. Ha igen, megkezdi az üzenet küldését, ezzel a szükséges hosszúságúra nyújtva a dinamikus slotot.

A dinamikus slotban is megelőzi – a statikus slothoz hasonlóan – az üzenetek tényleges küldését az Action Point Offset. Ha éppen nem volt mit küldeni, a slot üres marad, és a minimális egy minislot hosszúságú idő letelte után a node-ok tovább léptetik számlálóikat, ezzel a következő dinamikus slotba lépve.

Az üzenet elküldése után a dinamikus slot hosszát a DTS (Dynamic Trailing Sequence – dinamikus lezárószekvencia) mező egészíti ki úgy, hogy mindig minislot határon végződjön. A DTS mező egy nullákból álló sorozat, mely a végén egy darab egyest tartalmaz. Ezután következnek a statikus szegmens esetén már ismertetett Channel Idle Delimiter, és Channel Idle mezők.



1.6. ábra. *A dinamikus szegmens felépítése*

Adódhat olyan eset, mikor egy ciklusban több dinamikus üzenet áll készen a küldésre, mint amennyi elférne a szegmensben. Ebben az esetben az utolsó néhány üzenet elküldésére ebben a ciklusban már nem lesz lehetőség, továbbításukra a következő ciklusban kerül sor, így tehát az üzenetekhez rendelt dinamikus slot azonosítója egyben az üzenet prioritását is jelzi. Minél alacsonyabb a sorszám, annál nagyobb valószínűséggel kerül sor egy „túlzsúfolt" ciklusban az adott keret elküldésére.

Szimbólum ablak (Symbol Window)

Ebben az időrésben történik a különböző vezérlő szimbólumok átvitele. Ilyenek például a cluster felébresztését kiváltó „Wake Up" szimbólumok, az egyes végpontok számára az első ciklust megjelölő és az átvitel tesztelésére szolgáló szimbólumok.

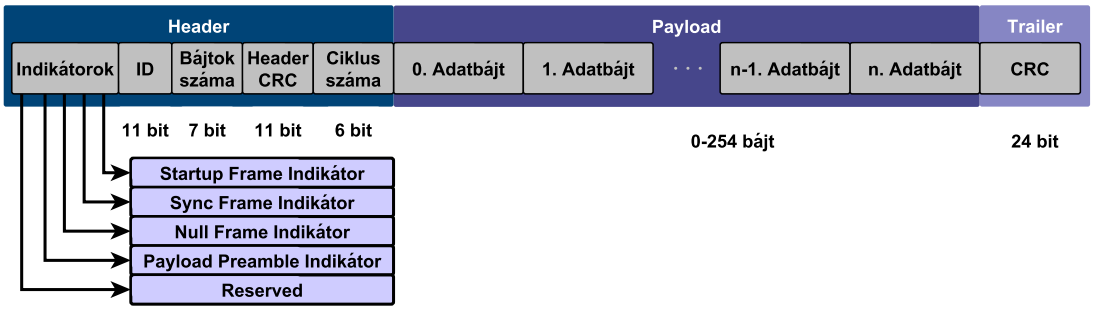
Üresjárási idő (Network Idle Time - NIT)

A statikus- és dinamikus szegmens, valamint a szimbólum ablak által üresen hagyott időt a NIT tölti ki. Ebben a sávban semmilyen adattovábbítás nem történhet. Ez az idő szolgál a végpontok óráinak egymástól való esetleges eltérésének kiszámítására és adott esetben az órák szinkronizálására. Szintén ebben az időben történhet a különböző, megvalósítás függő feladatok ellátása (a ciklus alatt fogadott adatok feldolgozása, felkészülés a következő ciklusra...).

1.1.4. A FlexRay üzenetek felépítése

Minden üzenet három fő részből áll: a Header-ből (fejléc), a Payload-ból (adatbájtok) és a Trailer-ből (lezárás). Ez a három rész kiegészül a buszon való továbbításhoz szükséges további egységekkel. A keret elején elhelyezett TSS-el (Transmission Start Sequence – átvitelindító szekvencia), mely 3-15 db recesszív bitből áll. Erre jelterjedési és időzítési megfontolásokból van szükség. Ezután következik a keret elejét jelző, egy darab domináns bitből álló FSS (Frame Start Sequence – keret kezdő szekvencia), melyet a minden bájtot megelőző két bites BSS (Byte Start Sequence – bájt kezdő szekvencia) után közvetlenül követ a Header első bájtja. A BSS a vevő bájtkezdethez való szinkronizációját segíti elő egy domináns-recesszív élváltással. A keret átvitelének végét a kétbájtos recesszív-domináns

élváltást tartalmazó FES (Frame End Sequence – keret vége szekvencia) jelzi. Ezt követik a dinamikus szegmens esetén már ismertetett DTS, a Channel Idle Delimiter, majd a Channel Idle mezők.



1.7. ábra. *A FlexRay keret felépítése*

Header

A keret fejléce összesen 40 bitből áll. Az első bit „reserved”, vagyis foglalt bit, melyet négy indikátor bit követ. Az indikátor bitjek a keret funkcióját specifikálják. Normál üzenet esetében mindegyik értéke nulla. A bitek sorrendben: Payload Preamble Indicator, Null Frame Indicator, Sync Frame Indicator, Startup Frame Indicator. (Jelentésüket a Payload bekezdésben részletezem.) A következő 7 bit az adatszavak számát határozza meg. Egy szó 16 bites, tehát egy üzenet összesen 254 adatbájtot tartalmazhat. Ezt követi egy 11 bit hosszúságú CRC (Cyclic Redundancy Check – ciklikus redundancia vizsgálat) mező, mely a Header blokk hibavédelmét szolgálja. A fejléc utolsó 6 bitje a ciklus számláló, amely mindig az aktuális ciklus sorszámát tartalmazza.

Payload

A keretnek ebben a részében továbbítódnak a maximálisan 254 bájtnyi információt hordozó adatbájtok. Az adatbájtok számának a statikus szegmens minden üzenetében ugyanannyinak kell lennie, a dinamikus szegmensben azonban ez üzenetről üzenetre változhat.

A statikus szegmensben küldött üzenetek első 12 bájtjában lehetőség van Network Management vektorok[1] továbbítására, melyet a keret elején elhelyezkedő Payload Preamble Indicator jelez. A dinamikus üzenetek esetén a Payload Preamble Indicator bit jelzi, hogy az első két adatbájt a küldött üzenet ID-ja. Így lehetőség nyílik adott dinamikus slothoz több különböző jelentésű üzenet hozzárendelésére is, ha ezeket kis valószínűséggel kell egyidőben elküldeni.

Előfordulhat, hogy a FlexRay ciklusban eljön az idő egy üzenet elküldésére, de a kontroller valamilyen oknál fogva még nem fér hozzá a bufferéhez, és így nem tud érvényes adatot elküldeni. Ebben az esetben a Payload minden bájtja nulla értékű. Ezt jelzi a Header Null Frame Indicator bitje. Ilyenkor az üzenet tartalmával a fogadó feleknek nem kell

[1] A hálózat energiagazdálkodásáért felelős protokollok számára fenntartott üzenetek

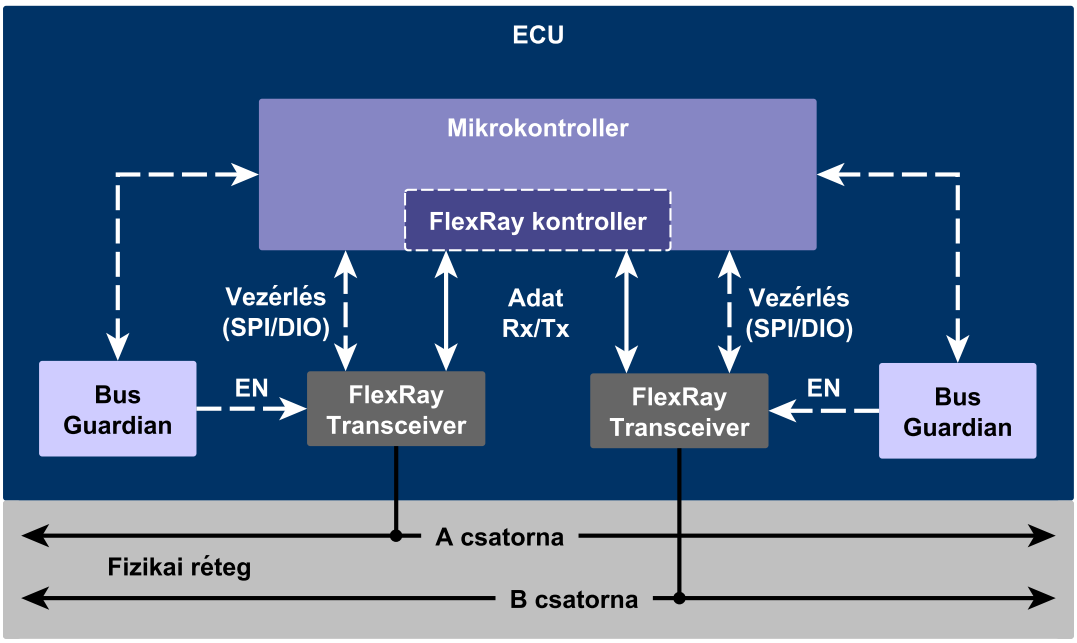
foglalkoznia.

Trailer

A lezárás a Payload védelmére szolgáló 24 bites CRC mezőt tartalmazza.

1.1.5. Az ECU-k illesztése a FlexRay buszhoz

Az ISO/OSI modell adatkapcsolati rétegének, valamint – ennek alrétegének – a közeghozzáférési rétegnek (MAC) a megvalósítása a FlexRay kontroller feladata. A FlexRay kontrollert gyakran integrálják a különböző autóipari célra gyártott mikrovezérlőkbe. A kontroller logikai jeleit a FlexRay transceiver alakítja át a fizikai réteg specifikációjának megfelelő differenciális jelekké, és illeszti a fizikai közeghez. A szabvány javasolja, de nem teszi a kötelezővé az úgynevezett Bus Guardian használatát. Ez egy külön hardver eszköz, mely önálló órával és időzítőkkel rendelkezik. Megfelelő konfigurálás után a kontrollertől függetlenül felügyeli a buszt, és a transceiver kimenetét engedélyező lábán közvetlenül képes a FlexRay időzítésének megfelelően tiltani vagy engedélyezni a busz meghajtását. Segítségével elkerülhető, hogy a transceiver vagy a kontroller meghibásodásából eredő, rosszul időzített keretek ellehetetlenítsék a kommunikációt a teljes buszon.



1.8. ábra. *Egy FlexRay node felépítése*

A FlexRay kontroller

Ez az egység felelős a specifikációnak megfelelő kommunikációs protokoll betartásáért. Elvégzi a küldött és a fogadott üzenetek bufferelését, kódolását, dekódolását és a FlexRay

szigorú ütemezésének megfelelő továbbítását. Ő felel továbbá a szinkronizációért, a wake-up és startup szekvenciák végrehajtásáért, valamint a protokoll üzenetek és szimbólumok feldolgozásáért.

A FlexRay transceiver

A transceiver végzi adás esetén a logikai jelek átalakítását a szabványnak megfelelő differenciális jelekké, vétel esetén pedig a busz felől érkező jeleket konvertálja a kontroller számára logikai jelekké. A kontrolleren kívül a transceiver kapcsolatban áll magával a mikrovezérlővel is, mely a különböző állapotait és funkcióit vezérli. Jellemzően négy alapvető állapotban működhet: Normal, Standby, Sleep és ReceiveOnly. A Sleep és ReceiveOnly állapot nem minden transceiver által támogatott. Adott esetben a transceiver képes a buszon fellépő hibák (szakadás, offszet, keretütközések) észlelésére és jelentésére a FlexRay kontrollernek vagy a mikrokovezérlőnek.

1.2. Az AUTOSAR 4.0

Ebben a részben bemutatom az AUTOSAR szoftverarchitektúrájának felépítését, rétegeinek funkcióját és azok egymáshoz kapcsolódását. Külön ismertetem a kommunikációs stack[1] felépítését, alrétegeinek feladatát. A FlexRay kommunikációban résztvevő modulok szerepének és az alkalmazáskomponensek FlexRay buszon történő üzenetváltásának részletezése már a feladatmegoldás részét képezi, tárgyalása külön fejezetben történik.

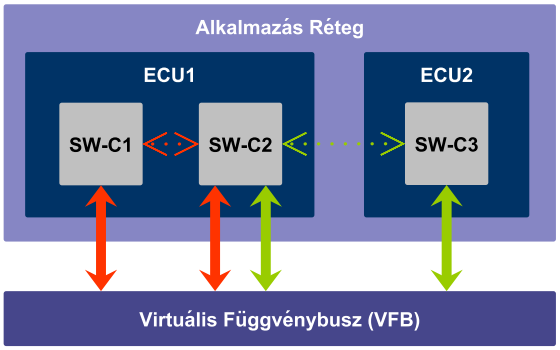
1.2.1. Az AUTOSAR rétegzett szoftverarchitektúrája

Az AUTOSAR alapfilozófiája, hogy az ECU funkcionalitását egymástól és a hardvertől független, atomi szoftverkomponensek (SW-C) valósítják meg. A komponensek egymással és a különböző hardverszinten megvalósított szenzorokkal és beavatkozókkal szabványos AUTOSAR interfészeken keresztül kommunikálnak. A kommunikáció a Virtuális Függvénybuszon (VFB – Virtual Function Bus) keresztül zajlik. A VFB egy absztrakt buszrendszer, melynek feladata, hogy elrejtse az ECU-k határait a szoftvermodulok elől, így a kommunikáció szempontjából mindegy, hogy valójában melyik komponens melyik vezérlőegységen fut. Ezzel a megoldással a szoftver hardverfüggő és hardverfüggetlen rétegekre oszlik, megvalósítva a bevezetőben tárgyalt alapelveket.

A legmagasabb absztrakciós szinten az AUTOSAR három réteget különböztet meg:

- Alkalmazás Réteg - Application Layer
- Futtató Környezet - Runtime Environment (RTE)
- Alap Szoftver - Basic Software (BSW)

[1] Az architektúrának – funkcióját és függőségeit tekintve – jól elkülöníthető része.



1.9. ábra. *Szoftverkomponensek kommunikációja a VFB-n keresztül*

Alkalmazás Réteg

Ez az AUTOSAR teljesen hardverfüggetlen rétege. Itt helyezkednek el az ECU funkcionalitását megvalósító alkalmazáskomponensek. A komponensek méretére és funkciójára az AUTOSAR nem tesz megkötést, csupán annak interfészeit írja elő. Egy komponensből egy rendszerben akár több példány is futhat egymástól függetlenül. Az SW-C-k közvetlenül mindig csak a Futtató Környezettel (RTE) kommunikálnak.

Futtató Környezet (RTE)

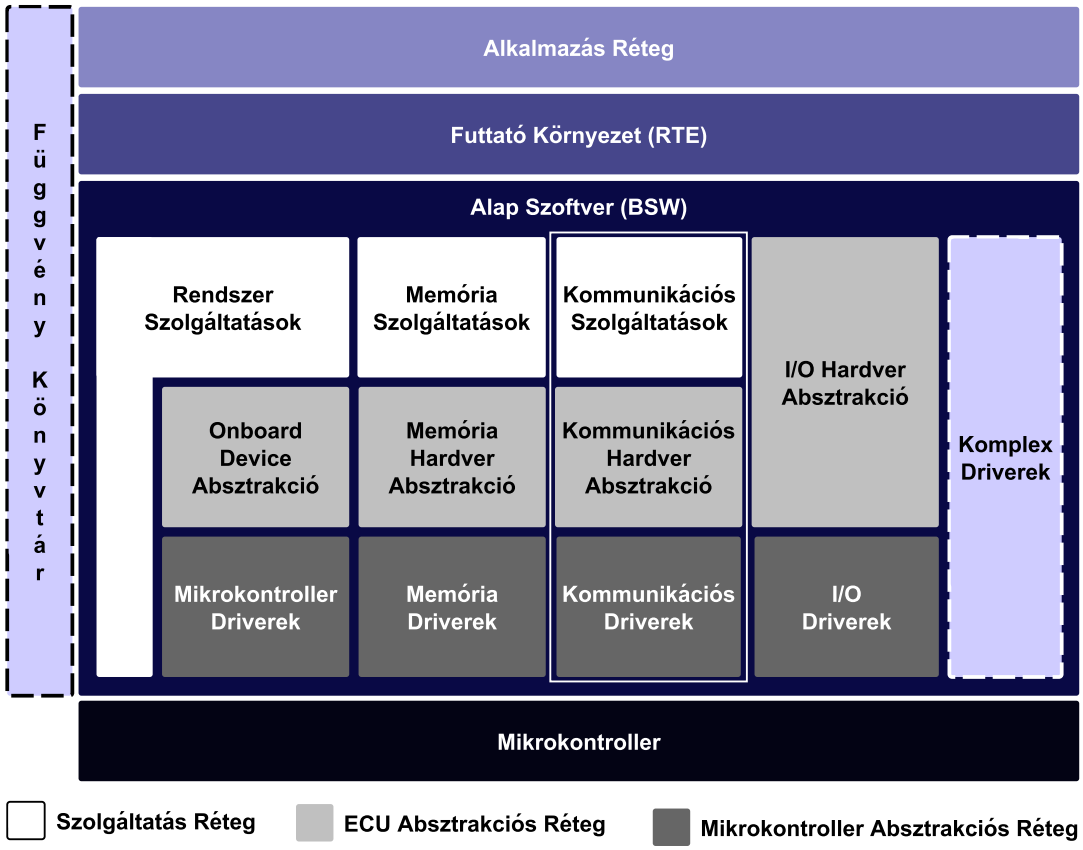
Ez a réteg bonyolítja a kommunikációt két szoftverkomponens, vagy szoftverkomponens és szenzor, illetve beavatkozó között. Megvalósítása generálással történik, egy, az ECU működését leíró modellből. Felfelé – az Application Layer felé – nyújtott interfésze hardver- illetve ECU-független. Az alkalmazáskomponensek által kezdeményezett kommunikáció mindig ezen a rétegen keresztül valósul meg. A komponensek szempontjából az ECU-n belüli és az ECU-k közötti (CAN-en, LIN-en vagy FlexRay-en zajló) kommunikációban nincs különbség.

Alap Szoftver (BSW)

A Basic Software réteg feladata garantálni a fölötte elhelyezkedő rétegek hardverfüggetlenségét. Szabványos szoftvermodulokból áll, melyek további rétegekbe csoportosíthatók. A szoftvermodulok jól definiált interfészeikkel – API függvényeiken keresztül – kapcsolódnak egymáshoz. A rétegek alulról felfelé haladva egyre jobban elvonatkoztatnak a konkrét hardvertől:

Szolgáltatás Réteg - Services Layer

Ez a BSW réteg legfölső alrétege. Alapvető szolgáltatásokat nyújt az RTE és a BSW modulok felé. Feladatai közé tartozik például az operációs rendszer funkciók, a kommunikáció- és memória-menedzsment, különböző hibakezelő, diagnosztikai és watchdog szolgáltatások, valamint az ECU energiamenedzsmentjét kezelő szolgáltatások megvalósítása. Független az ECU és a mikrovezérlő architektúrájától.



1.10. ábra. *Az AUTOSAR rétegzett szoftverarchitektúrája*

ECU Absztrakciós Réteg - ECU Abstraction Layer

Egységes felületet nyújt a mikrovezérlő belső és külső perifériáihoz való hozzáféréshez, függetlenül az ECU-n belüli elhelyezkedésüktől és az ECU-hoz való kapcsolódásuk módjától. A külső perifériákhoz tartozó driverek is ebben a rétegben kapnak helyet. Ennek a rétegnek a megvalósítása már független az alkalmazott mikrovezérlőtől, de erősen függ az ECU kialakításától.

Mikrokontroller Absztrakciós Réteg - Microcontroller Abstraction Layer

A BSW réteg legalsó alrétege. Ebben az alrétegben vannak megvalósítva a különböző mikrokontroller-specifikus driverek, melyek közvetlen hozzáférést biztosítanak a kontrollerhez és annak belső perifériáihoz. Megvalósítása az alkalmazott mikrovezérlő típusától erősen függ.

Komplex Driverek - Complex Drivers

Az AUTOSAR által nem specifikált eszközök és driverek kezelésére, vagy időkritikus programrészek szabványtól eltérő módon való implementálására szolgál. Ezen keresztül zajlik az Alkalmazás Réteg szoftvere által közvetlenül vezérelt mikrokontrollerperifériák, szenzorok és beavatkozók elérése. Megvalósítása erősen kötődik a mikrokontroller típusához és az ECU-hoz.

A BSW réteg moduljai funkcionális szempontból a következő blokkokra, vagy úgynevezett stack-ekre oszthatók:

I/O
: Szabványosított hozzáférést biztosít szenzorokhoz, beavatkozókhoz és a mikrovezérlő külső perifériáihoz.

Memória - Memory Stack
: Szabványosított hozzáférést biztosít a külső és belső – elsősorban nem felejtő – memóriákhoz.

Kommunikáció - Communication Stack
: Egységes felületet nyújt az RTE számára a kommunikációs hálózatokhoz való hozzáféréshez.

Rendszer - System
: Ide tartoznak az operációs rendszer, a különböző időzítők, hibakezelés és rendszerszolgáltatások.

Az architektúra igény esetén kiegészülhet egy függvénykönyvtárral (Library), mely a komponensek által gyakran használt közös függvényeket tartalmazza.

1.2.2. A kommunikációs stack

A kommunikációs stack az architektúra kommunikációért felelős szoftvermoduljainak halmaza. A stack a BSW réteg felépítése szerint felbontható Driver, Absztrakciós és Szolgáltatás alrétegekre, a modulok által megvalósított kommunikációs protokollok szerint pedig CAN, LIN, Ethernet és FlexRay stackekre. A különböző stackek felépítése nagyon hasonló.

Kommunikációs Driverek réteg
: Ez a réteg tartalmazza a mikrovezérlő belső, kommunikációs perifériáit kezelő, alacsony szintű driver modulokat, a CAN, LIN, Ethernet és FlexRay kontroller, valamint transceiver drivereket. Ebben a rétegben kapott helyet az SPI Handler Driver is, mely az SPI perifériát hivatott kezelni.

Kommunikációs Hardver Absztrakciós réteg
: Ebben a rétegben találhatók a külső perifériákat vezérlő driverek, a külső CAN, LIN, Ethernet és FlexRay kontroller, valamint transceiver driverek. Ezek a modulok legtöbbször SPI buszon, vagy általános I/O lábakon keresztül férnek hozzá a hardverhez.

  Szintén itt helyezkednek el a drivereket eltakaró Interface modulok is (CAN Interface, LIN Interface, Ethernet Interface, valamint FlexRay Interface). Ezek a modulok egységes felületet biztosítanak az azonos protokollokhoz, de különböző típusú perifériákhoz tartozó driverek eléréséhez.

Kommunikációs Szolgáltatások réteg
: A Kommunikációs stack legfelső rétegében valósul meg a szignálok PDU-kba (Protocol Data Unit – protokoll adategység) csomagolása (lsd. később), majd továbbítása a

különböző protokollok moduljainak. Ha szükséges, szintén ebben a rétegben történik az adott protokoll számára túl nagyméretű PDU-k darabolása, majd újraegyesítése (CAN, LIN és Flexray Trasnport Protocol), de itt történik a Hálózat Menedzsment megvalósítása, a buszok energiagazdálkodásának vezérlése is.

1.2.3. Konfiguráció

Az AUTOSAR – ahol lehet – a generikusságra törekszik. Egy BSW modulnak a konfigurációjától függően több variánsa is lehet. A szabvány a konfigurációs paramétereket három osztályba sorolja aszerint, hogy értékük a szoftver életciklusának melyik szakaszában dől el. A modulok specifikációja minden paraméterhez megadja, hogy milyen osztályba tartozik. Ha egy paraméterhez több osztály is meg van adva, az implementációra van bízva, hogy végül milyen osztályba kerül. A szabvány a következő három osztályt definiálja:

Fordítás előtti - pre-compile time

Ide tartoznak a különböző preprocesszor makrókkal megvalósított, konstans paraméterek. A makrók értékének függvényében lehetséges kódrészletek kihagyása, vagy forráskódba illesztése, így téve lehetővé bizonyos funkciók engedélyezését, illetve tiltását. Értéküket a fordítás előtt meg kell határozni, így a fordító a paraméterek függvényében képes a kód optimalizálására. Előnye ennek a konfigurációs osztálynak, hogy sokkal hatékonyabb kódot eredményez, hátránya azonban, hogy változtatás esetén mindenképpen újrafordítást igényel.

Linkelési idejű - link time

Az ebbe az osztályba sorolt paraméterek a modul forrásán kívül vannak definiálva. A linker feladata feloldani a hivatkozásokat, így ezek függvényében már további optimalizálás, kódrészlet elhagyása nem történhet. Ez a megvalósítás nem igényli a modul újrafordítását csupán linkelését. Lehetőséget biztosít az alkalmazott konfiguráció fordítás utáni kiválasztására. Módosítás esetén nincs szükség a modul forráskódjára, csupán a lefordított objektum állományokra.

Futásidejű - post-build time

Ezek a paraméterek – hasonlóan a link time paraméterekhez – a modul forrás állományain kívül vannak definiálva. Fontos különbség azonban, hogy ezeknek a memóriában külön terület van lefoglalva, ahol adott esetben több variánsuk is elhelyezésre kerülhet. Előnyük, hogy változtatásuk nem igényli a modul újrafordítását vagy újralinkelését, csupán a paraméterek értékének újratöltését a megfelelő memóriaterületre. Ezekkel a paraméterekkel valósítható meg a modul konfigurációjának futásidőben történő változtatása. A modul az inicializáció során kapja meg a megfelelő memóriaterületre mutató pointert, majd beolvassa onnan a konfigurációs adatokat.

A modulok konfigurálása

A modulok különböző osztályba sorolt konfigurációs paraméterei a következő állományokban vannak megvalósítva:

<modulnév>_Cfg.h A pre-compile time paramétereket tartalmazza, általában előfeldolgozó makrókként megvalósítva.

<modulnév>_Lcfg.c A link time konfigurációs adatokat tartalmazza. Legtöbbször struktúrákként, vagy egyszerű adattípusokként megvalósítva.

<modulnév>_PBcfg.c A post-build time konfigurációs paramétereket tartalmazza. Struktúrákat és egyszerű adattípusokat is tartalmazhat, de ezek mindegyike egy kiindulási, úgynevezett „gyökér" struktúrába vannak összefogva. A modul az inicializáció során egy a kiindulási struktúrára mutató pointert kap meg, és a későbbiek során a post-build time konfigurációs adatokat csak ezen keresztül érheti el.

Az állományokat egy PC-s generátor szoftver hozza létre egy, az ECU felépítését és teljes működését leíró adatmodell alapján.

1.2.4. Hibakezelés és Diagnosztika

A szoftver életciklusának szempontjából két hibatípust különböztetünk meg:

Fejlesztési hibák - Development Errors
A fejlesztési hibákat általában nem megfelelő konfiguráció vagy hibás kód eredményezi. Felderítésük és kijavításuk még a fejlesztési fázisban meg kell történjen. Az ilyen hibák detektálása és naplózása a végtermékben pre-compile time paraméterekkel kikapcsolható.

Termék hibák - Production Errors
A termékhibákat általában valamilyen hardver hiba vagy futásidejű kivétel okozza. Ezeket nem lehet előre kivédeni, kezelésükre, detektálásukra és naplózásukra a végtermék esetében is szükség van. Ez a funkció nem kapcsolható ki.

A hibák kezelése és jelzése a BSW rétegben többféleképpen is megvalósulhat:

API függvény visszatérési értékében
A modulok API függvényeinek többsége visszatérési értékében jelzi, hogy végrehajtása sikeres volt-e, vagy esetleg problémába ütközött. Ha a hiba csak a visszatérési értékben van jelezve, az általában csak valamilyen erőforrás foglaltságára utal, a hívó félnek később lehetősége van a függvény újrahívására.

Úgynevezett Callback függvényeken keresztül
Nem blokkoló függvényhívások esetén a hibás függvény egy „visszahívó, vagy Callback" függvény segítségével értesíti a hibáról a hívó felet.

Development Error Tracer BSW modulon keresztül
A Development Error Tracer (DET) modul a fejlesztési hibák naplózására és kezelésére szolgál. Ez egy szabványos BSW modul, mely hiba esetén a megfelelő API függvényének meghívása után elvégzi a hiba naplózását. A hibákhoz ErrorHook-ok rendelhetők, melyek az adott hiba bekövetkeztekor végrehajtandó programot tartalmazzák (legtöbbször break-point vagy végtelen ciklus).

Diagnostic Event Manager BSW modulon keresztül

A Diagnostic Event Manager (DEM) modul a termékhibák naplózására és diagnosztizálására szolgál. Szintén szabványos BSW modul, melynek működése a DET-hez hasonló, de képes az alkalmazáskomponensek értesítésére, valamint bizonyos ECU funkciók letiltására is.

2. fejezet

A kommunikáció folyamata az AUTOSAR-ban

Ebben a fejezetben először ismertetek néhány, a kommunikáció folyamatának megértéséhez szükséges fogalmat, majd röviden összefoglalom a kommunikációs stack FlexRay stack-el összefüggésben álló moduljainak szerepét. Végül egy példán keresztül bemutatom az adatküldés és -fogadás folyamatát.

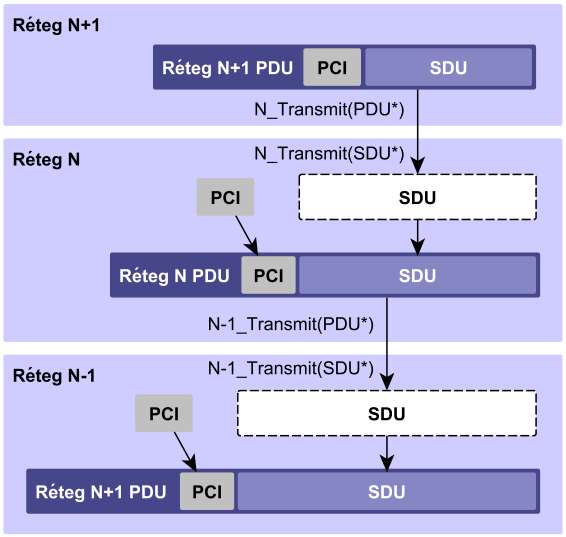
2.1. A kommunikáció alapjai

A kommunikációs stack feladata elsősorban az alkalmazás rétegbeli komponensek közötti adatátvitel biztosítása. A komponensek szabványos AUTOSAR interfészeken keresztül kommunikálnak egymással. Az adatokat az RTE úgynevezett szignálokká alakítja, melyeknek továbbítása a kommunikációs stack feladata.

A kommunikációs stack-ben a rétegek közti adattovábbítás úgynevezett PDU-kban (Protocol Data Unit - protokoll adategység) történik. A PDU-kban az átviendő hasznos információ egy az adott rétegbeli protokollra jellemző, kiegészítő információval ellátott csomagban kerül továbbításra. Ezt a kiegészítő információt a PDU PCI (Protocol Control Information – protokoll vezérlő információ) mezője tartalmazza. A PCI mező alapján történik – az átvitelt követően – a fogadó oldalon az azonos protokollt megvalósító réteg által az adatok feldolgozása és továbbítása a felette lévő rétegnek. A küldő rétegtől kapott továbbítandó adatokat az SDU (Service Data Unit – szolgáltatás adategység) mező tartalmazza. Az adatcsomagok rendelkeznek még egy azonosítóval is, az úgynevezett PDU ID-val, mely az adott kontextusban egyértelműen azonosítja azokat.

Az ábrán a PDU-k rétegek közötti átvitelét követhetjük nyomon. Látható, hogy az egyik réteg által küldött PDU az alatta levő réteg számára már SDU-ként jelenik meg. Az alsó réteg a kapott SDU alapján elkészíti a saját PDU-ját a megfelelő PCI mezővel, majd továbbítja azt az alatta levő rétegnek, ahol ez a PDU megint SDU-ként jelenik meg. Ez így megy egészen a legalsó – általában driver – rétegig, ahol a PDU már közvetlenül a buszon kerül továbbításra a fogadó ECU driver rétegének.

A fogadás során az adatok alulról felfelé haladnak. A driver a kapott PDU-ból a PCI



2.1. ábra. *A PDU-k felépítése*

mező alapján kicsomagolja az SDU-t, majd továbbítja azt a felette lévő rétegnek, mely ezt PDU-ként kezelve a neki szóló PCI mező alapján ismételten feldolgozza az SDU-t majd továbbítja felfelé. Ez a legfelső rétegig folytatódik, ahol a PDU-ból végül az RTE-nek szóló szignálok kerülnek kibontásra. Az RTE megkapja a szignálokat, és a bennük tárolt adatokat eljuttatja az alkalmazáskomponensekhez.

Attól függően, hogy egy adott PDU a kommunikációs folyamat melyik rétegéből származik, három alapvető típust különböztetünk meg:

I-PDU (I-SDU)
: Az Interakciós Réteg, vagy az OSI modell szerinti Megjelenítési Réteg PDU-i. Összeállításuk a COM modulban történik az RTE-től kapott szignálokból. Egy I-PDU több szignált is tartalmazhat.

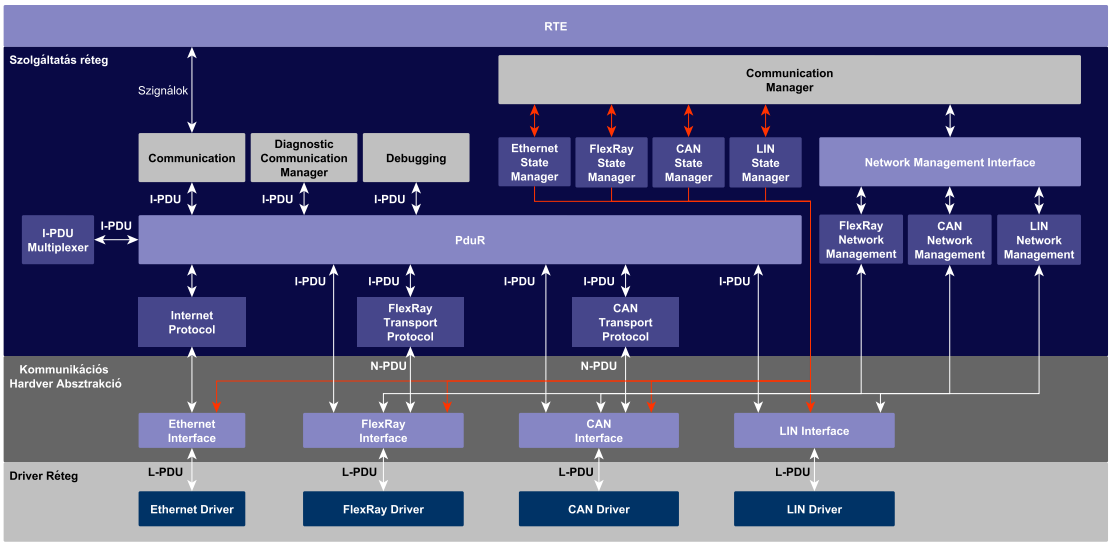
N-PDU (N-SDU)
: A kommunikáció Hálózati Rétegének PDU-ja. A transport protokollok hozzák létre őket, az adott kommunikációs protokoll által küldhető maximális méretet meghaladó I-PDU-k darabolásával.

L-PDU (L-SDU)
: Az Adatkapcsolati réteghez tartozó PDU-k. A hardver absztrakciós modulok (Interface-ek) hozzák létre őket I-SDU-kból és N-SDU-kból, majd a driver réteghez kerülnek a fizikai buszon való továbbításra. A FlexRay esetén egy L-PDU-ban több I-PDU is továbbítható.

Az adatáramlás iránya szerint megkülönböztetünk Tx és Rx I/N/L-PDU-kat. Az adott ECU szempontjából továbbítandó PDU-kat Tx előtaggal, míg az érkező PDU-kat Rx előtaggal jelöljük. A modulokon belül a Tx és Rx PDU-k külön vannak kezelve, így egy kontextuson belül előfordulhat azonos ID-val Tx és Rx PDU is.

2.2. A kommunikációs stack moduljai

A kommunikációs stack által nyújtott szolgáltatások feladata kettős. Egyrészről felelnek az adatforgalom lebonyolításáért (adatküldés esetén az RTE-től érkező szignálokat PDU-kba csomagolva továbbítják a hálózaton, adatfogadás esetén a hálózaton érkező PDU-kból kicsomagolt szignálokat továbbítják az RTE-nek), másrészről pedig, feladatuk a Network Management és a buszok-állapotainak vezérlése. Ennek megfelelően a Szolgáltatás Réteg moduljai is két csoportra oszthatók. Az adatok továbbításáért felelős modulokra, melyeket a Communication modul (COM) irányít, és a hálózati menedzsmentért felelős modulokra, melyeket pedig a Communication Manager (ComM) modul felügyel. Az Hardver (vagy ECU) Absztrakciós Rétegben a szálak közösen haladnak tovább, hiszen az egész stack alját képező driver rétegek csak az Interface modulokon keresztül érhetők el.



2.2. ábra. *A kommunikációs stack moduljai*

2.2.1. Protokoll-független modulok

A kommunikációs stack Szolgáltatás Rétegében helyezkednek el a protokoll-független modulok. A szignálok küldését kezdeményező és fogadás esetén a feldolgozását végző modulok, a hálózati menedzsmentet és diagnosztikai kommunikációt megvalósító modulok, valamint a PDU-k indításáért és irányításáért felelős modulok tartoznak ide.

Communication - COM

A Communication modul feladata az RTE felé nyújtott kommunikációs szolgáltatások megvalósítása felfelé szignál alapú, míg lefelé PDU alapú interfészek biztosításával. Ez a modul fogadja az RTE-től érkező, a szoftverkomponensek által küldött adatokat tartalmazó szignálokat vagy szignálcsoportokat, majd I-PDU-kba csomagolva továbbítja őket a PDU Router modulnak. A másik irányból nézve a COM feladata a PDU Router-től érkező I-PDU-k

által tartalmazott szignálok vagy szignálcsoportok továbbítása az RTE részére. Képes továbbá átjáróként is funkcionálni, vagyis alulról érkező I-PDU-k bizonyos szignáljait vagy szignálcsoportjait egy másik I-PDU-ban továbbküldeni. Ezen feladatainak megvalósítása közben elvégzi a szükséges bájt-sorrend konverziókat, szűri a fogadott és küldött adatokat, valamint gondoskodik a PDU-k optimális időbeli ütemezéséről. Támogatja változó hosszúságú PDU-k és szignálok kezelését is. [5]

Diagnostic Communication Manager - DCM

A DCM modul külső diagnosztikai eszközök ECU-hoz való kapcsolódását teszi lehetővé, egy általános, hálózatfüggetlen diagnosztikai célú API nyújtásával. Vezérli a diagnosztikai adatok adás-vételét, valamint a különböző diagnosztikai állapotokat. Lefelé a BSW moduloktól, felfelé – az RTE-n keresztül – pedig a szoftverkomponensektől képes diagnosztikai adatok kérésére, majd továbbítására a PDU Routeren keresztül a megfelelő külső eszköznek. Szerepe a fejlesztéstől kezdve, a gyártáson át a kész termék esetleges későbbi karbantartásáig vagy szervízeléséig is kiterjed. [6]

Debugging

A Debugging modul képes a BSW modulok és az RTE adatainak gyűjtésére, tárolására, külső eszközre való továbbítására és módosítására. Segítségével felderíthetők a fejlesztési hibák, nyomon követhetőek a modulok belső állapotai. Hálózatfüggetlen interfészével a PDU Routeren keresztül képes a külső eszköz parancsainak fogadására. Mivel képes az ECU belső működésének és a memóriában tárolt adatoknak futásidőben történő módosítására, biztonsági okokból használata csak a fejlesztés idejére korlátozódhat. [7]

Communication Manager - ComM

A Communication Manager modul feladata a kommunikációhoz szükséges erőforrások és a kommunikációs stack BSW moduljainak felügyelete. Összegyűjti a különböző forrásokból származó buszkommunikációs kéréseket, és koordinálja azok kiszolgálását. Feladata a hardver elrejtése a kommunikáló egységek elől és a felhasználói szoftvermodulok által kért, absztrakt kommunikációs módokba váltás menedzselése. Az RTE felé egységes interfészt biztosít a kommunikációs erőforrások igénylésére. Képes bizonyos szignálok küldésének letiltására, hogy elkerülje az ECU felesleges felébresztését. [8]

Network Management Interface - NmIf

A Network Management Interface egy generikus modul, mely interfészként szolgál az alatta elhelyezkedő busz specifikus network management modulokhoz. Egységes felületet biztosít a ComM modul számára a hálózat menedzseléséhez különböző network management protokollok esetén is. Képes a buszok szinkronizált ki- és bekapcsolására. [9]

I-PDU Multiplexer - IpduM

Előfordulhat, hogy bizonyos üzenetek elküldésére csak nagyon ritkán van szükség, ezért számukra külön PDU-t és PDU ID-t fenntartani felesleges. Ebben az esetben a PDU multiplexelés lehetőséget biztosít arra, hogy azonos PCI mezővel és ID-val, de más és más felépítésű és tartalmú SDU mezővel küldjünk el üzeneteket. Ilyenkor az SDU-ba bekerül egy szelektor mező, mely egyértelműen meghatározza a multiplexált PDU-ban érkező üzenet tartalmát. A PDU Multiplexer feladata a COM-tól érkező PDU-k új, multiplexált PDU-ba csomagolása, majd visszaküldése a PDU Router-nek, ahol megtörténik annak továbbítása. A fogadóoldalon a Multiplexer megkapja a multiplexált PDU-t, meghatározza annak tartalmát, majd a PDU Routeren keresztül továbbküldi az üzenetet a megfelelő címzettnek. [10]

PDU Router - PduR

A PDU Router feladata a PDU-k irányítása a különböző protokollokat megvalósító modulok és a Szolgáltatás Réteg moduljai között. A PduR alatt találhatók a Transport Protocol valamint Interface modulok, felette pedig a COM, DCM, és Debugging modulok. Az IPDU Multiplexer modul a PduR szempontjából egyszerre értelmezhető felső és alsó modulként is. A PDU-k irányítását a modul kizárólag a PDU ID és a forrás ismeretében teszi, statikus (pre-compile time) konfigurációja alapján, miközben a PDU-kat az adott célmodulban érvényes új ID-val látja el. Nem biztosít lehetőséget a PDU-k tartalma, vagy futásidőben változó paraméterek szerinti útvonal irányításra.

A legegyszerűbb esetben a PduR függvényeinek hívását az adott paraméterekkel egyszerűen továbbítja a megfelelő modul felé. Előfordulhat azonban, hogy a modul gateway funkciót is megvalósít, vagyis egy alsó rétegből érkező üzenetet egy másik alsó rétegnek kell továbbítania. Ezt a feladatot a PDU Router az üzenet bufferelését elvégezve, tulajdonképpen egy virtuális felső rétegként viselkedve látja el. [11]

2.2.2. A FlexRay stack moduljai

A FlexRay stack a kommunikációs stack általános felépítését követi. A Driver (vagy Mikrokontroller Absztrakciós) Rétegben helyezkednek el a hardverrel szoros kapcsolatban álló, belső FlexRay kontroller driverek. A Hardver (vagy ECU) Absztrakciós Rétegben a külső (nem a mikrovezérlőbe integrált) kontrollerek és a szintén külső transceiver-ek driverei vannak megvalósítva. Ebben a rétegben foglal helyet a FlexRay Interface modul is. A Szolgáltatás Réteg FlexRay stack-hez tartozó moduljai a FlexRay Transport Layer, a FlexRay State Manager és a FlexRay Network Management modulok.

FlexRay Transport Layer - FrTp/FrArTp

A FlexRay Transport Layer modul feladata a FlexRay által küldhető, maximum 254 bájtos méretet meghaladó PDU-k feldarabolása, majd továbbítása a FlexRay buszon. A fogadó oldalon ez a modul végzi a kapott PDU-k alapján az üzenet összeállítását, majd továb-

bítását a PduR modulon keresztül a felső moduloknak. Korábban az AUTOSAR egy saját transport protokollt alkalmazott ezen feladatok megvalósítására, ám a 4.0-s verziótól kezdve az ISO 10681-2 szabványban definiált protokoll megvalósítását javasolja. Kompatibilitási okokból a korábbi AUTOSAR protokoll továbbra is a szabvány részét képezi, így két Transport Layer modul is definiálva van, a FlexRay ISO Transport Layer és a FlexRay AUTOSAR Transport Layer. Az adatok továbbítását a FlexRay Interface modulon keresztül valósítja meg. [12] [13]

FlexRay Interface - FrIf

A FlexRay Interface feladata a hardverek és az azokat kezelő driverek számától független, egységes felületet biztosítani a FlexRay perifériák vezérléséhez. A FlexRay perifériákat vezérlő driverek a FlexRay kontroller és a FlexRay transceiver driverek. A FlexRay esetében – lévén idővezérelt protokollról van szó – szintén az Interface feladata a fentről jövő I-PDU-k és N-PDU-k L-PDU-kba csomagolása, majd a buszciklushoz ütemezett továbbítása a driverek felé. [14]

FlexRay Driver - Fr

A FlexRay driver feladata a FlexRay kontroller funkcióinak szoftveres kezelése. Általában SPI-on vagy I/O lábakon keresztül vezérli a hardvert. A driver segítségével egységes felületen keresztül érhetők el a kontroller üzenet bufferei és konfigurációs beállításai. A kontrollerhez való mindenfajta hozzáférés csakis a driveren keresztül valósulhat meg. Egy driver modul képes több azonos típusú kontroller kezelésére is, ilyenkor a kontrollerek nullától kezdődő indexeléssel érhetőek el. [15]

FlexRay Transceiver Driver - FrTrcv

A FlexRay driverhez hasonlóan a FlexRay Transceiver Driver is közvetlenül a hardverrel létesít kapcsolatot. Minden hardverhez való hozzáférés kizárólag az FrTrcv-n keresztül lehetséges. Vezérli a transceiver buszállapotait és engedélyezheti vagy tilthatja a különböző wake-up kéréseket. [16]

FlexRay Network Management - FrNm

A FlexRay Network Management egy hardverfüggetlen protokollt valósít meg a FlexRay buszállapotok vezérlésére (normal és bus-sleep módok). Az alapfunkciója kiegészülhet a hálózatot alkotó többi node állapotának a detektálásával is. Az FrNm modulok képesek a buszon egymással kommunikálni és így összehangolt működést megvalósítani. A hardverhez minden esetben a FlexRay Interface modulon keresztül fér hozzá. [17]

FlexRay State Manager - FrSM

A FlexRay State Manager feladata a hálózat be- és kikapcsolásának komplex folyamatát irányítani. Absztrakt interfészt biztosít a ComM modul számára a kommunikáció elindí-

tására vagy kikapcsolására egy clusterben. Az FrSM szintén az FrIf modulon keresztül fér hozzá a hardverhez. [18]

2.2.3. A modulok közötti adatátvitel során használt függvények

Az adatok átvitele a modulok között azok API függvényeinek segítségével történik. A függvények visszatérési értéke általános esetben `Std_ReturnType` típusú, melynek kér értéke lehet: `E_OK`, ha a függvény futása közben nem történt hiba, és `E_NOT_OK`, ha a függvény végrehajtása közben hiba történt. A modulok közti kommunikáció során a PDU-k azonosítására és átadására két függvényparaméter szolgál.

PduIdType PduId

Ez egy 8 vagy 16 bites, előjel nélküli egész típus, mely a PDU ID-ját tartalmazza.

PduInfoType* PduInfoPtr

A `PduInfoType` egy összetett adattípus, mely tartalmaz egy `uint8*` típusú, a PDU adatbájtjaira mutató pointert (`SduDataPtr`), valamint egy `PduLengthType` típusú, előjel nélküli egész változót, mely a PDU hosszát adja meg (`SduLength`).

A modulok – általános esetben – adatküldés során a következő függvényeket használják:

Transmit

```
Std_ReturnType <modulnév>_Transmit(
    PduIdType id,
    PduInfoType* info
);
```

A hívó – felső modul – ezen a függvényen keresztül jelzi adatküldési szándékát a hívott – alsó – modul számára. A hívott fél – konfigurációjától függően – Immediate vagy Decoupled Buffer Access metódussal dolgozza fel a kérést. (lsd. később)

TriggerTransmit

```
Std_ReturnType <modulnév>_TriggerTransmit(
    PduIdType id,
    PduInfoType* info
);
```

Ez a függvény szolgál – Decoupled Buffer Access esetén – a PDU ismételt elkérésére. A felső modul nyújtja az alsó modul számára.

TxConfirmation

```
void <modulnév>_TxConfirmation(
     PduIdType id
 );
```

Az alsó modul a felső modul ezen függvényének meghívásával jelzi az átvitel sikeres megtörténtét.

CancelTransmit

```
Std_ReturnType <modulnév>_CancelTransmit(
    PduIdType id
);
```

Ez a függvény a `Transmit` függvénnyel már elindított, de `TxConfirmation`-el még nem visszaigazolt kérések megszakítására szolgál. Általában akkor kerül sor egy kérés megszakítására, ha a megadott időn belül nem érkezik meg a visszaigazolás.

RxIndication

```
void <modulnév>_RxIndication(
    PduIdType id,
    PduInfoType* info
);
```

Ezen a függvényen keresztül történik, a felső modul értesítése egy PDU fogadásáról, és a PDU átadása az `info` paraméteren keresztül.

2.3. Példa az adatküldés folyamatára

A következőkben egy példát ismertetek, melyben egy – a COM-ban összeállításra kerülő – I-PDU útját követhetjük nyomon egészen a driver rétegig. A példában a FlexRay kontroller a mikrovezérlő belső perifériája, így drivere a Driver Rétegben található. Mivel a FlexRay Transport Layer modult a későbbiekben részletesen bemutatom, és a modulok együttműködésének megértését ezen a ponton nem segíti elő, ezért példámban szándékosan nem szerepeltetem.

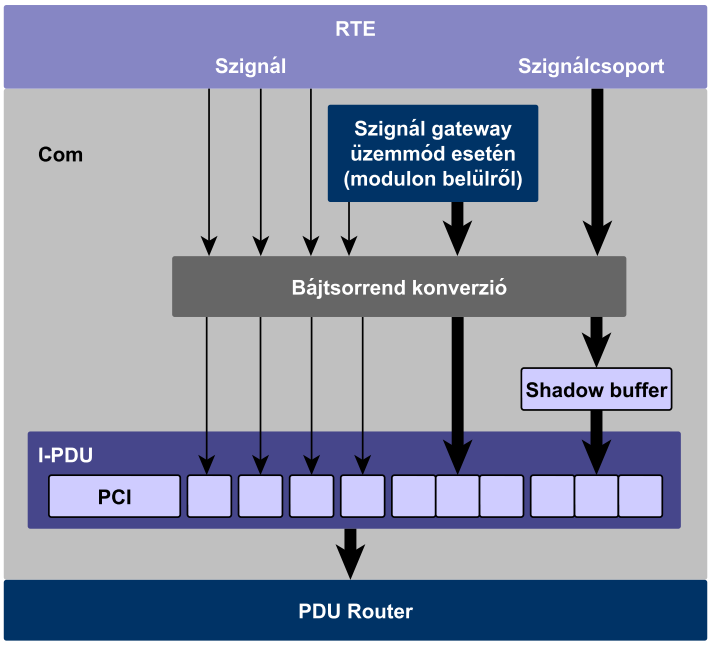
Az I-PDU-k összeállítása (COM)

A szignálok – vagy összetett adattípusok esetén a szignálcsoportok – I-PDU-kba csomagolása a COM feladata, innen indul az adatküldés folyamata. A szignálok származhatnak az RTE-től egy alkalmazás rétegbeli komponens adataiból, de – amennyiben az adott ECU olyan szignál alapú átjáróként is szolgál, melyen keresztül két, más ECU-kon elhelyezkedő komponens válthat üzenetet – származhatnak egy a COM által korábban fogadott I-PDU-ból is.

Egy szignál küldését az RTE a `Com_SendSignal` vagy szignálcsoportok esetén a `Com_UpdateShadowSignal`, illetve `Com_SendSignalGroup` függvények hívásával kezdeményezi. A hívás paraméterei tartalmazzák a szignál azonosítóját valamint egy pointert arra memóriaterületre, ahol a szignál által tartalmazott adatok találhatók. A COM elvégzi a szükséges byte-sorrend konverziókat, majd a megfelelő bufferbe másolja a szignálo-

kat. A `Com_SendSignal` hatására a küldött szignál egyből az I-PDU bufferében a számára fenntartott helyre másolódik, szignálcsoportok küldése esetén az egyes szignálok először a `Com_UpdateShadowSignal` függvény hatására egy – a csoport számára fenntartott – úgynevezett Shadow Bufferbe másolódnak, majd az összeállított szignálcsoport a `Com_SendSignalGroup` függvény hívásával az I-PDU bufferébe kerül. Ezzel elkerülhető az esetlegesen nem konzisztens szignálokból álló szignálcsoportok elküldése.

Az ábrán egy példa látható az I-PDU-k összeállítására.



2.3. ábra. *Az I-PDU-k összeállítása a COM-ban*

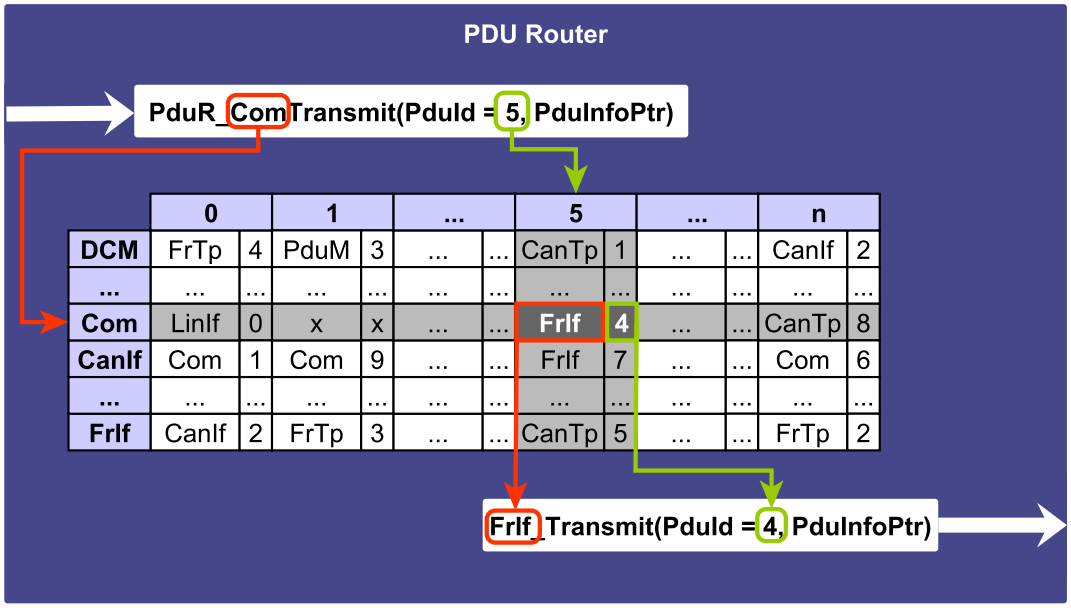
A szignál bájtjainak elhelyezése után a COM összeállítja a szignálhoz tartozó PCI-t, majd továbbítja az I-PDU-t a PDU Router modulnak a `PduR_ComTransmit` függvény meghívásával.

Az I-PDU-k útvonalválasztása (PduR)

A PduR modul miután a `PduR_ComTransmit` függvényének meghívásával megkapta a továbbítandó PDU ID-ját és egy pointert a PDU bufferére, egy – a statikus (pre-compile time) konfigurációjában meghatározott – táblázat alapján *(2.4. ábra)* kiválasztja azt az alsó modult, melynek a kérést továbbítania kell, valamint a PDU új, az alsó modulban érvényes ID-ját. Ebben az esetben ez a modul a FlexRay Interface, melynek `FrIf_Transmit` függvényét meghívva átadja annak az új ID-t (4) és a kapott `PduInfo` pointert.

Az L-PDU összeállítása és továbbítása (FrIf, Fr)

Az FrIf modul a `Transmit` függvényében kétféleképpen folytathatja a kérés feldolgozását. (Az alkalmazott mód PDU-nként konfigurálható.) Az úgynevezett Immediate Buffer Access



2.4. ábra. *A PDU Router működése*

(azonnali buffer hozzáférés) esetén egyből továbbítja L-PDU-ként a kapott I-PDU-t a driver réteg felé az `Fr_TransmitTxLPdu` függvény meghívásával. Az Immediate Buffer Access-el küldött I-PDU-k általában a FlexRay ciklus dinamikus szegmensébe kerülnek. Ebben az esetben az L-PDU-ban egyetlen I-PDU helyezhető el.

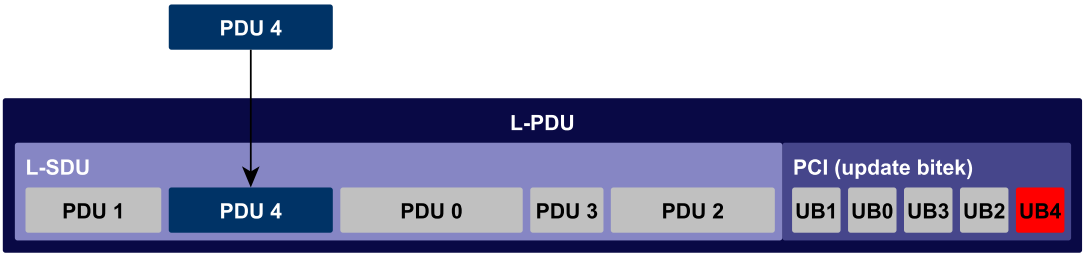
A másik mód az úgynevezett Decoupled Buffer Access (függetlenített buffer hozzáférés). Ebben az esetben a `Transmit` függvény hatására az FrIf egy flag beállításával megjegyzi, hogy az adott PDU készen áll a küldésre, majd a FlexRay ciklus megfelelő időpontjában a `PduR_FrIfTriggerTransmit` függvényen keresztül az FrIf ismét elkéri a PDU-t a felső rétegtől. Ez a PDU Routeren keresztül valósul meg. (Ebben a példában a `Com_TriggerTransmit` függvényt kell továbbhívni.)

Ezután az Interface – a konfigurációja szerint – elhelyezi az ebben az időben elküldésre kijelölt L-PDU-ban az I-PDU-kat, majd beállítja az L-PDU PCI mezőjét. *(2.5. ábra)* Ezesetben a PCI az úgynevezett update bit-eket tartalmazza, melyek azt jelzik, hogy az L-PDU-ban elhelyezett adott PDU tartalmaz-e érvényes információt. Miután a teljes L-PDU-t összeállította, az FrIf az L-PDU-hoz konfigurált driver `Fr_TransmitTxLPdu` függvényén keresztül továbbítja az L-PDU-t.

A driver a kapott L-SDU-t egy dedikált memóriaterületre, az úgynevezett Message Bufferbe másolja, majd később a kontroller a bufferből kiolvasva a FlexRay ütemezésének megfelelően elküldi a frame-et.

Visszaigazolás

A FlexRay Interface később (a FlexRay ciklus megfelelő, előre konfigurált idejében) a driver `Fr_CheckTxLPduStatus` függvényén keresztül lekérdezi az előzőleg elküldött L-



2.5. ábra. *Az L-PDU összeállítása*

PDU állapotát. Amennyiben az L-PDU továbbítása sikeresen megtörtént, a FlexRay Interface visszaigazolja az összes – az L-PDU által tartalmazott – I-PDU átvitelét a `PduR_FrIfTxConfirmation` függvény meghívásával a PDU Router-en keresztül a megfelelő felső moduloknak. A PDU Router modul a függvényhívást a példánkban szereplő I-PDU esetén az ismert módon átirányítja a COM-nak a `Com_TxConfirmation` függvényen keresztül. A COM a visszaigazolást követően úgynevezett notification-ökön keresztül értesíti az RTE-t és visszaigazolja az I-PDU-ban elküldött összes szignál átvitelét. Ezzel az adatküldés folyamata lezárult.

Amennyiben a COM részére a visszaigazolás a konfigurációban megadott időn belül nem érkezik meg, értesíti az RTE-t a hibáról, majd a `PduR_ComCancelTransmit` függvény meghívásával törli a PDU elküldésének kérését. A PDU Router a FlexRay Interface-hez irányítja a kérést, melynek hatására az FrIf felszabadítja az ehhez a kéréshez kapcsolódó erőforrásait, és a drivert is utasítja az átvitel felfüggesztésére a `Fr_CancelTxLPdu` függvényen keresztül.

2.4. Példa az adatfogadás folyamatára

Az adatok fogadásának menetét a küldésnél megismert példán, a FlexRay drivertől a COM modulig követem nyomon.

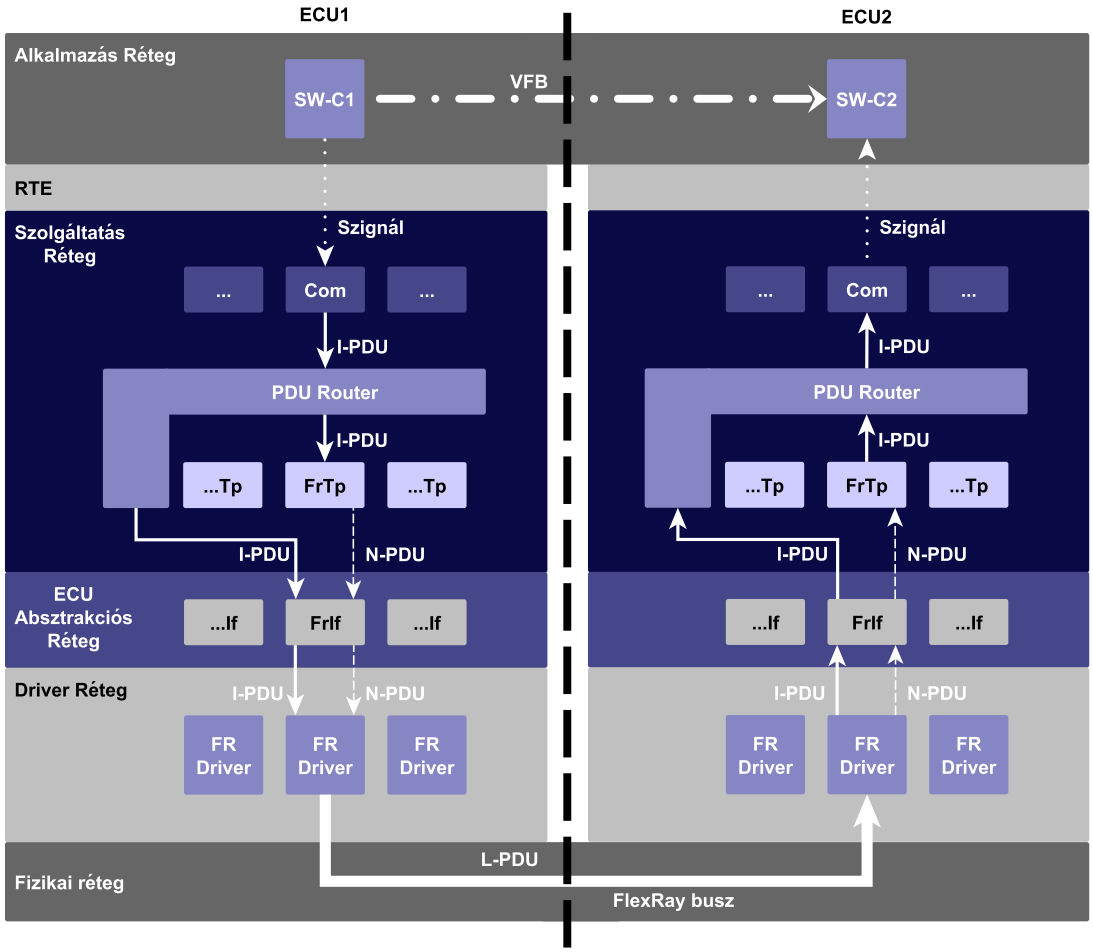
Az L-PDU feldolgozása (Fr, FrIf)

Amikor a FlexRay ciklus egy adott L-PDU fogadásának idejéhez ér, a FlexRay Interface az `Fr_ReceiveRxLPdu` függvény meghívásával lekérdezi a drivertől az L-PDU állapotát. Amennyiben valóban megérkezett a kérdéses L-PDU, hozzáférést ad az FrIf számára annak bufferéhez. Az FrIf modul az L-PDU PCI mezőjében az általa szállított összes I-PDU update bit-jét megvizsgálja, és amennyiben valamelyikük frissítve lett – tehát érvényes adatot tartalmaz – a küldés folyamatához hasonlóan kétféleképpen reagálhat. A PDU Routeren keresztül azonnal jelzi a felső modul számára – a `PduR_FrIfRxIndication` függvény meghívásával – az adott I-PDU rendelkezésre állását, és átadja neki az I-PDU-t, vagy egy helyi bufferben eltárolja az adatokat, és csak később – egy a konfigurációban meghatározott időpontban – értesíti a felső modult.

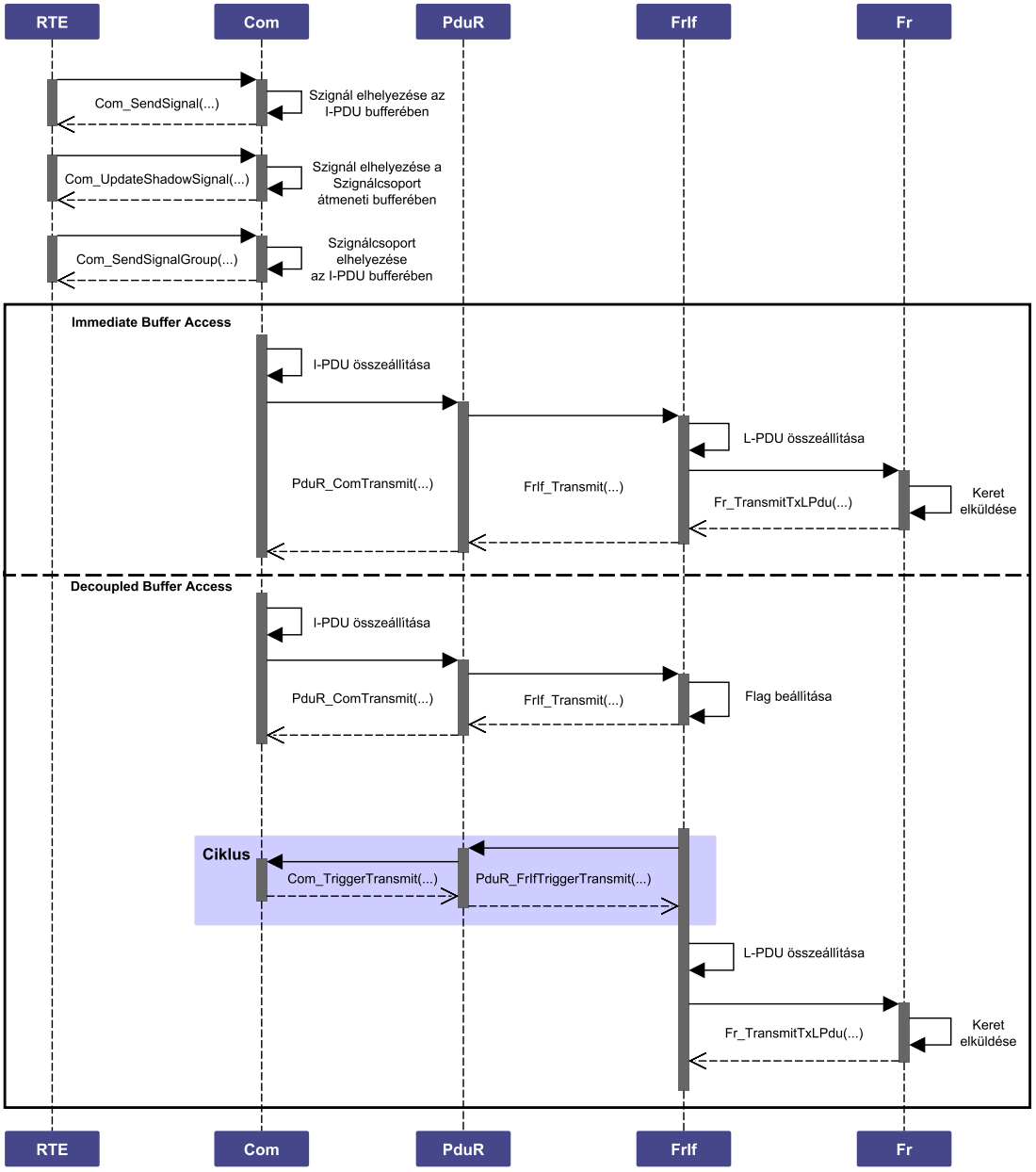
Az I-PDU feldolgozása

Az `PduR_FrIfRxIndication` függvény a mi esetünkben – a PDU Routeren keresztül – a COM `Com_RxIndication` függvényére lesz átirányítva. A COM miután megkapta az I-PDU-t, – a PCI mezőben tárolt információk alapján – kiveszi belőle a rendelkezésre álló szignálokat, majd egy notification-ön keresztül értesíti az RTE-t az érkezett szignálokról. Az RTE később a küldésnél megismert függvények párjain keresztül (`Com_ReceiveSignal, Com_ReceiveSignalGroup, Com_ReceiveShadowSignal`), egyesével elkéri a szignálokat vagy szignálcsoportokat. Ezzel lezárul az adatok fogadásának folyamata.

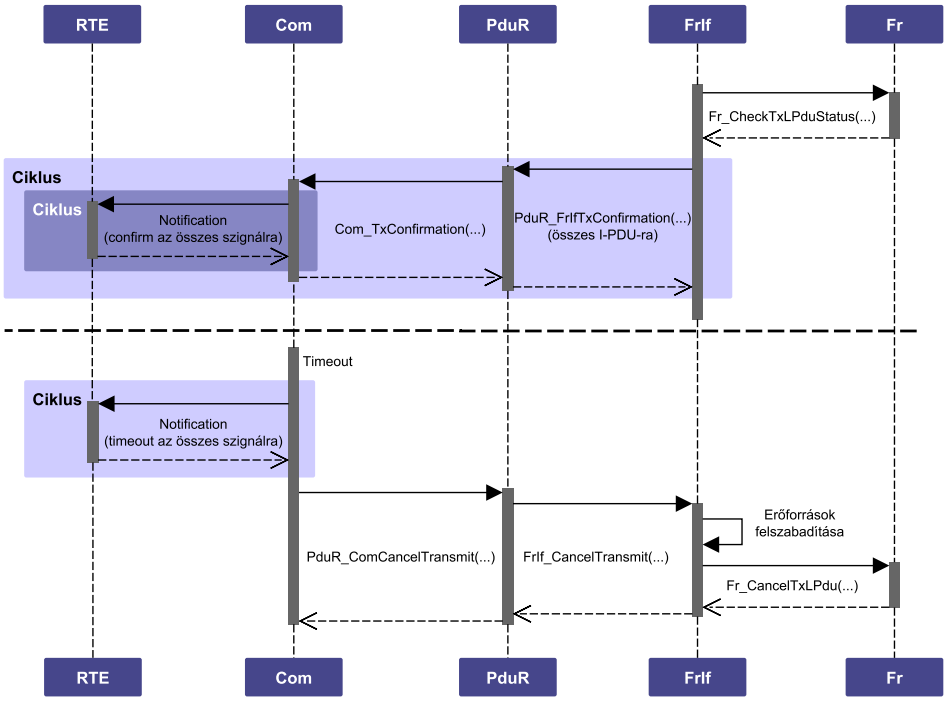
Azokat a szignálokat, melyeket esetlegesen – gateway üzemmódban – egy másik ECU-nak kell továbbítani, a COM elhelyezi a megfelelő továbbítandó I-PDU-ban, majd elküldi a hálózaton.



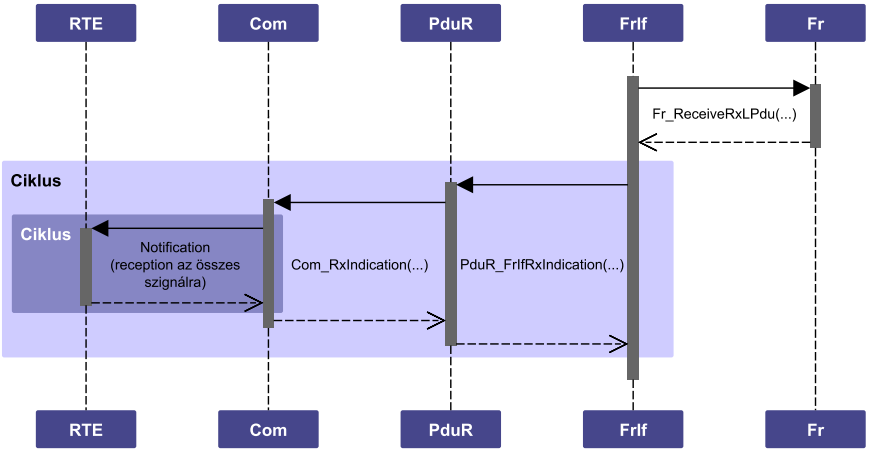
2.6. ábra. *A kommunikáció teljes folyamata*



2.7. ábra. *Az adatküldés szekvenciája*



2.8. ábra. *A visszaigazolás és a visszavonás szekvenciája*



2.9. ábra. *Az adatfogadás szekvenciája*

3. fejezet

A FlexRay Interface modul implementálása

Ebben a fejezetben a FlexRay Interface BSW modul implementációját mutatom be. A fejezet elején összefoglalom a modul AUTOSAR szabvány által meghatározott feladatait, és ismertetem a modul általam felépített adatstruktúráit. Bemutatom a modul absztrakciós szerepét megvalósító függvényeinek működését, valamint az ütemezési feladatokat ellátó algoritmusokat. Ismertetem a FlexRay Interface-en keresztül zajló adatküldés és -fogadás folyamatát és az ezt megvalósító algoritmusokat. Végül becslést adok a modul erőforrásigényére különböző konfigurációk esetén.

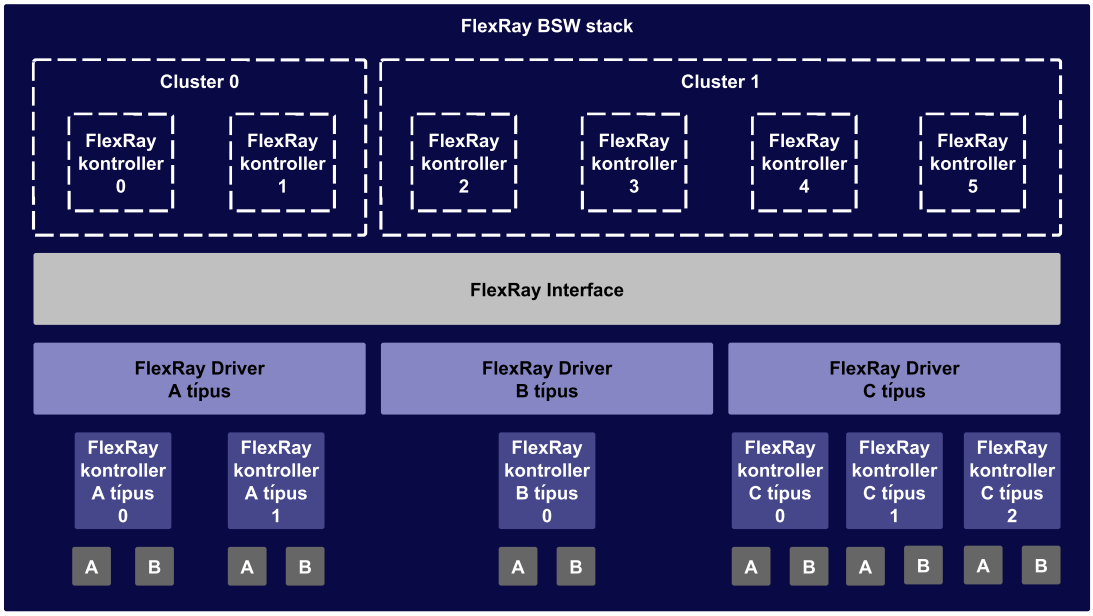
3.1. A szabvány értelmezése, a modul feladata

A FlexRay Interface modul az architektúra ECU Absztrakciós Rétegében helyezkedik el. Feladata a felette lévő modulok számára a vezérlőegység felépítésétől független, egységes felületet biztosítani a FlexRay driverekhez való hozzáféréshez, valamint a kommunikáció megvalósításához.

Az FrIf működése két jól elkülöníthető részre osztható. Egyrészről demultiplexer funkciót valósít meg a felső modulok – ebben az esetben jellemzően a Network Management és State Manager modulok – és a driverek API függvényei között. Másrészről pedig a FlexRay kommunikációs ciklusainak megfelelően ütemezi a kommunikációs folyamatokat. Eközben elvégzi a továbbítandó PDU-k (továbbiakban csak PDU) L-PDU-kba rendezését, valamint a fogadott L-PDU-k feldolgozását.

3.1.1. A FlexRay Interface demultiplexer funkciója

Egy ECU-ban több különböző típusú FlexRay kontroller és transceiver is helyet kaphat, melyek mind különböző drivereket igényelnek. Egy adott driver több azonos típusú hardver kezelését teszi lehetővé. Az eszközök a driverek API függvényein keresztül, minden driver esetén nullától kezdődő, szorosan fűzött indexeléssel érhetők el. Az FrIf feladata ezen kontroller és transceiver driverek elrejtése, és az indexek összefűzése egy nullától kezdődő folytonos sorozattá.



3.1. ábra. *A FlexRay Interface által nyújtott absztrakció*

Néhány, a későbbiekben használt fogalom AUTOSAR szerinti definíciója [14]:

ControllerIndex: A ControllerIndex egy absztrakt, egyértelmű, nullától kezdődő, folytonos index, mely absztrakciót valósít meg a kontrollerek felett, függetlenül a típusuktól, helyüktől és elérésük módjától.

ClusterIndex: A ClusterIndex egy absztrakt, egyértelmű, nullától kezdődő, folytonos index, mely absztrakciót valósít meg a clusterek[1] felett, függetlenül a típusuktól, helyüktől és elérésük módjától.

ChannelIndex: A ChannelIndex-nek két értéke lehet: `FR_CHANNEL_A` és `FR_CHANNEL_B`. A ControllerIndex és a ChannelIndex együtt egyértelműen azonosítanak egy transceiver-t.

A FlexRay Interface absztrakciót megvalósító API függvényei mind megfeleltethetők az egyes driverek által megvalósított függvényeknek. Az FrIf feladata a kapott ControllerIndex-ből vagy Controller- és ChannelIndex kombinációból a konfigurációja alapján kiválasztani a megfelelő driver meghívandó függvényét valamint az adott kontroller vagy transceiver driveren belüli indexét, elvégezni a függvényhívást a kapott paraméterekkel, majd a driver függvényének visszatérési értékét átadni a hívó modulnak.

3.1.2. Kommunikációs műveletek végrehajtása

Az Interface másik feladata az adatok küldésének és fogadásának FlexRay ciklushoz ütemezett vezérlése. Ezt úgynevezett JobList-ek végrehajtásával valósítja meg, melyek előre

[1] Ha az ECU gateway-ként is szolgál, több clusterhez is kapcsolódhat, valamint előfordulhat olyan eset is, mikor egy ECU-nak több kontrollere is azonos clusterhez kapcsolódik.

definiált műveleteket – úgynevezett Job-okat – tartalmazó listák. A FlexRay ciklus előre meghatározott időpontjaiban az időponthoz rendelt Job végrehajtásra kerül. Minden cluster saját, külön konfigurálható JobList-el rendelkezik, hiszen a FlexRay ütemezése (ciklusainak, szegmenseinek hossza) clusterenként eltérő lehet. Egy Job több kommunikációs műveletből áll, melyek mindegyikéhez egy-egy L-PDU van rendelve, amivel kapcsolatosan az adott műveletet végre kell hajtani. Ezek a következők lehetnek:

DECOUPLED_TRANSMISSION: A hozzárendelt Tx L-PDU által tartalmazott decoupled buffer access (függetlenített buffer hozzáférés) metódussal továbbítandó PDU-kat elkéri a felső modultól, majd az L-PDU-t összeállítja és továbbítja a drivernek.

TX_CONFIRMATION: A hozzárendelt, előzőekben elküldött Tx L-PDU állapotát lekérdezi a drivertől, majd ha az átvitel sikeresen megtörtént, az L-PDU által tartalmazott, még nem visszaigazolt PDU-k elküldését visszaigazolja a megfelelő felső modulnak.

RECEIVE_AND_STORE: Lekérdezi a drivertől az ehhez a művelethez rendelt Rx L-PDU állapotát, majd amennyiben érkezett új adat, az összes, az L-PDU által szállított és frissített PDU-t eltárolja egy-egy helyi bufferben. Az adott PDU-k vételéről csak később, az `RX_INDICATION` műveletben értesíti a felső modulokat.

RX_INDICATION: A `RECEIVE_AND_STORE` műveletben frissített tartalmú PDU-k vételéről értesíti a felső modulokat, majd továbbítja számukra az adott PDU-kat.

RECEIVE_AND_INDICATE: A `RECEIVE_AND_STORE` művelethez hasonló, de itt egyből megtörténik a felső modulok értesítése és az adatok továbbítása.

PREPARE_LPDU: Semmi mást nem végez, mint meghívja a hozzárendelt L-PDU továbbítására kijelölt driver `Fr_PrepareLPdu(...)` függvényét. A driver ezáltal felkészül az adott L-PDU továbbítására vagy fogadására. (Különböző erőforrás-optimalizációs vagy bufferfoglalási műveleteket hajthat végre.)

FREE_OP_A, FREE_OP_B: Szabadon felhasználható műveletek. Ezekben valósulhatnak meg az esetleges hardver- vagy implementáció-specifikus műveletek.

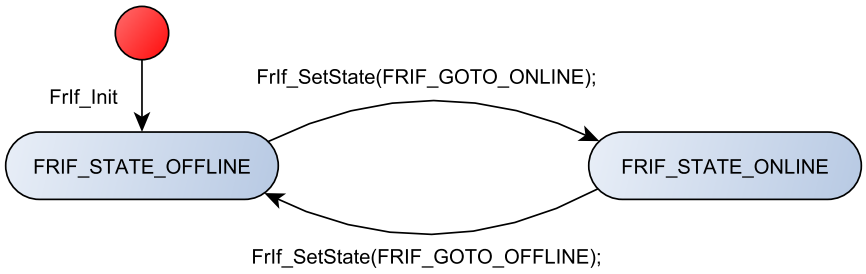
A műveleteket az `FrIf_JobListExec_<ClstIdx>(void)` függvény (JLEF) hajtja végre. A FlexRay clusterek eltérő időzítései miatt a függvénynek minden cluster számára külön léteznie kell. A függvény hívása megszakításból történik, melyet a kontroller driverek által felügyelt, úgynevezett AbsoluteTimer-ek váltanak ki. Minden JLEF-höz ki van jelölve egy dedikált AbsoluteTimer, mely megvalósítja a függvény időzített hívását. A JLEF feladata lefutása után a következő Job végrehajtási idejére – a driver API függvényein keresztül – „felhúznia” a hozzárendelt AbsoluteTimer-t.

Előfordulhat, hogy egy cluster JobList-jének végrehajtása kiesik a FlexRay időzítésével való szinkronitásból. Ezt a JLEF-nek fel kell ismernie, majd fel kell függesztenie a műveletek végrehajtását. A modul `FrIf_MainFunction_<ClstIdx>(void)` függvényének a feladata a

Job-ok végrehajtásának felügyelete és adott esetben szinkronizálása. A MainFunction egy operációs rendszer task által ciklikusan hívott függvény, melynek az előzőekben megismert okokból minden cluster számára külön léteznie kell.

3.1.3. A FlexRay Interface állapotai, inicializálása

A FlexRay Interface modul kívülről lekérdezhető és vezérelhető állapotai a következő állapotgéppel írhatók le:



3.2. ábra. *A FlexRay Interface modul állapotai*

FRIF_STATE_OFFLINE: Semmilyen kommunikációs művelet nem történik.

FRIF_STATE_ONLINE: A kommunikációs műveletek végrehajtódnak.

Az állapotgépnek minden cluster esetére egymástól függetlenül kell léteznie. Az ECU bekapcsolása vagy reset után az állapotgép inicializálatlan, kezdeti állapotban van. Ebből az állapotából csak az inicializálást követően léphet ki, és kerülhet `FRIF_STATE_OFFLINE` állapotba. Inicializálatlan állapotban a modul API függvényei nem hívhatók, hiszen a belső változók értéke ismeretlen, és a megfelelő futásidejű konfiguráció sem áll rendelkezésre. A modul inicializálása az `FrIf_Init(FrIf_ConfigType* FrIf_ConfigPtr)` szignatúrájú függvény meghívásával történik, mely paraméterében átadja a modul számára a post-build time konfigurációs struktúrára mutató pointert. Az Interface ezt követően elvégzi változóinak alaphelyzetbe állítását, és minden cluster állapotgépének `FRIF_STATE_ONLINE` állapotba vezérlését. Inicializált állapotban már hívhatók a modul API függvényei, melyeken keresztül elvégezhető a driverek, majd a hardver eszközök inicializálása is. Az `FRIF_STATE_OFFLINE` és `FRIF_STATE_ONLINE` állapotok között a clusterek az `FrIf_SetState(...)` függvénnyel vezérelhetők. A clusterek aktuális állapota pedig a `FrIf_GetState(...)` függvényen keresztül kérdezhető le.

3.2. Konfiguráció

A modul implentációját a konfigurációs adatstruktúra felépítésével kezdtem. A szabvány a teljes BSW rétegre specifikál egy összefüggő konfigurációs adatmodellt, mely a későbbiek során a konfigurálást végző PC-s generátor szoftver rendelkezésére áll. A feladatom egy, a

modell alapján generálható, az implementációmhoz legjobban illeszkedő, konkrét C nyelvű konfigurációs állomány tervezése volt.

A FlexRay Interface specifikációja a konfigurációs paraméterek egy részét pre-compile time osztályba sorolja, egy másik részükre pedig megengedi mindhárom osztály alkalmazását. (pre-compile time, link time, post-build time) A pre-compile time paraméterek, mivel értékük mindenképpen eldől még fordítás előtt, a legjobb optimalizáció érdekében preprocesszor makrókként lettek megvalósítva. Azok a paraméterek melyek bármelyik osztályba sorolhatók – megegyezés szerint – a legnagyobb rugalmasságot biztosító post-build time osztályba kerültek. Megvalósításuk C struktúrák és elemi típusok formájában történt.

Először megvalósítottam a modul specifikációja által megadott teljes adatmodellt, majd a modul implementációja során ezen a modellen finomításokat, egyszerűsítéseket végeztem. A továbbiakban ismertetem a végleges konfigurációs struktúrát.

3.2.1. A szabvány által specifikált pre-compile time paraméterek

Az engedélyező/tiltó funkciót megvalósító paramétereknek két lehetséges értéke lehet: `STD_ON` (bekapcsolt) és `STD_OFF` (kikapcsolt). A specifikáció a következő pre-compile time paramétereket definiálja:

FRIF_DEV_ERROR_DETECT: A fejlesztési hibadetektálás funkció engedélyezésére/tiltására szolgál.

FRIF_<függvénynév>_SUPPORT: Különböző API függvények engedélyezésére/-tiltására szolgál. Ha a driverek, vagy az implementáció nem támogatja valamelyik függvény megvalósítását, a makró értéke `STD_OFF`. Ebben az esetben az adott függvény nem fordul bele a programba, így ha esetlegesen felülről mégis meghívnák, a linkelés során hibaüzenetet kapunk.

FRIF_VERSION_INFO_API: A modul verziószámának lekérdezését megvalósító függvény engedélyezésére/tiltására szolgál. Ez a függvény az integrációt segíti, gyártási fázisban ki van kapcsolva.

3.2.2. A jobb optimalizáció érdekében létrehozott pre-compile time paraméterek

A FlexRay Interface modul a szabvány szerint számos olyan funkcióval kell rendelkezzen, amelyek rögzített architektúra és az alkalmazási terület (determinisztikus működésű valós idejű rendszer) esetén várhatóan nem relevánsak, pl. az üzenetküldés a dinamikus szegmensben. Ennek megfelelően létrehoztam néhány paramétert, amelyek segítségével a kódban adott esetben egyszerűsítések végezhetők, így memória és futásidő takarítható meg.

FRIF_USER_DEFINED_COMOP_x: Az implementáció specifikus kommunikációs műveletek engedélyezésére szolgál. Az **x** értéke lehet **A** illetve **B**. Amennyiben ezen műveletek nincsenek megvalósítva, érdemes a makrónak `STD_OFF` értéket adni, így a fordító ezeket figyelmen kívül hagyja.

FRIF_BYTESWAPPING_ENDIANNESS: Mikroprocesszor-specifikus paraméter. Azt adja meg, hogy a big vagy a little endian bájtsorrendű PDU-k bájtjait kell-e a felső réteghez való továbbítás előtt felcserélni. Két lehetséges értéke: `BIG_ENDIAN` és `LITTLE_ENDIAN`.

FRIF_IMMEDIATE_PDU_SUPPORT: Mivel valós idejű rendszerekben célszerűnek tűnik idővezérelt üzenetekkel kommunikálni, várhatóan nem fognak a rendszerben immediate buffer access metódussal – a dinamikus szegmensben – elküldendő PDU-k előfordulni. Ennek a paraméternek az `STD_OFF` értékre állításával forráskód szinten tiltható ez a szolgáltatás, így csökkentve a kódméretet.

FRIF_SINGLE_FRDRV és FRIF_SINGLE_TRCVDRV: Szintén várhatóan, az alkalmazások nagy részében egy FlexRay kontroller és azonos típusú transceiverek kerülnek felhasználásra. Ezek a makrók `STD_ON` értékükkel azt jelzik, hogy a rendszerbe egy-egy kontroller és transceiver driver kerül integrálásra. Ennek függvényében a kód egyszerűsödhet.

3.2.3. A post-build time konfigurációs adatok

A konfigurációs adatszerkezet az ábrán látható. *(3.3. ábra)* A következőkben a struktúrák adattagjait (nagy számuk miatt, és mivel a nevük utal a funkciójukra) külön nem részletezem, csak a struktúrák szerepét mutatom be.

FrIf_ConfigType: Ez a post-build time konfigurációs adatok gyökere. A modul `FrIf_Init` függvénye ilyen típusra mutató pointert vár paraméteréül, melyet a modul eltárol, és a későbbiekben kizárólag ezen keresztül fér hozzá a konfigurációs adatokhoz.

FrIf_ClstConfigType: Egy cluster konfigurációját tartalmazza (kontrollerek, JobList, időzítések).

FrIf_JobListConfigType: Ez a struktúra egy JobList konfigurációját tartalmazza.

FrIf_JobConfigType: Egy Job reprezentálására szolgál. Megadja a végrehajtásának idejét és a végrehajtandó műveleteket.

FrIf_ComOpConfigType: Ez a struktúra egy kommunikációs műveletet ír le. Megadja a végrehajtandó műveletet és azt, hogy melyik L-PDU-n kell azt végrehajtani.

FrIf_ComOpType: `enum` típus, melynek lehetséges értékei: `DECOUPLED_TRANSMISSION`, `TX_CONFIRMATION`, `RECEIVE_AND_INDICATE`, `RX_INDICATION`, `RECEIVE_AND_STORE`, `PREPARE_LPDU`, `FREE_OP_A` és `FREE_OP_B`.

FrIf_DemConfigType: Egy adott cluster diagnosztikai hibáinak detektálását konfiguráló struktúra.

FrIf_CCConfigType: Egy kontroller adatait tartalmazza. Megadja, hogy a kontroller melyik clusterhez tartozik, a hozzá tartozó drivert, a kontroller driveren belüli indexét, hogy melyik csatornái vannak használatban és a csatornákhoz tartozó transceiver-eket.

FrIf_FrDrvType: Egy driver reprezentálására szolgál. Ebben a struktúrában egy driver API függvényeire mutató függvénypointerek vannak.

FrIf_TrcvConfigType: Egy transceiver konfigurációját tartalmazza. Megadja a transceiver-hez tartozó drivert, és a transceiver driveren belüli indexét.

FrIf_TrcvDrvType: Az `FrIf_FrDrvType`-hoz hasonlóan egy transceiver drivert reprezentál.

FrIf_TxPduConfigType: Egy Tx PDU konfigurációs adatait tartalmazza. Megadja, hogy Immediate vagy Decoupled Buffer Access metódussal kell-e elküldeni a PDU-t, kell-e a driver visszaigazolására várni, melyik L-PDU-k tartalmazzák a PDU-t és a megfelelő felső modul függvényeket.

FrIf_RxPduConfigType: Egy Rx PDU konfigurációját tartalmazza. Megadja a PDU fogadása esetén felhasználható buffer helyét és méretét, valamint az értesítendő felső modul `RxIndication` függvényét.

FrIf_TxLPduFrameConfigType: Egy Tx L-PDU konfigurációját tartalmazza. Megadja az L-PDU által szállított PDU-kat, az L-PDU hosszára valamint elküldésének módjára vonatkozó adatokat és a küldésre kijelölt kontroller indexét.

FrIf_TxPduInFrameConfigType: Egy Tx PDU L-PDU-n belüli elhelyezkedésére vonatkozó adatokat tartalmaz. Megadja a PDU helyét, a PDU ID-ját, és az Update Bit adatait.

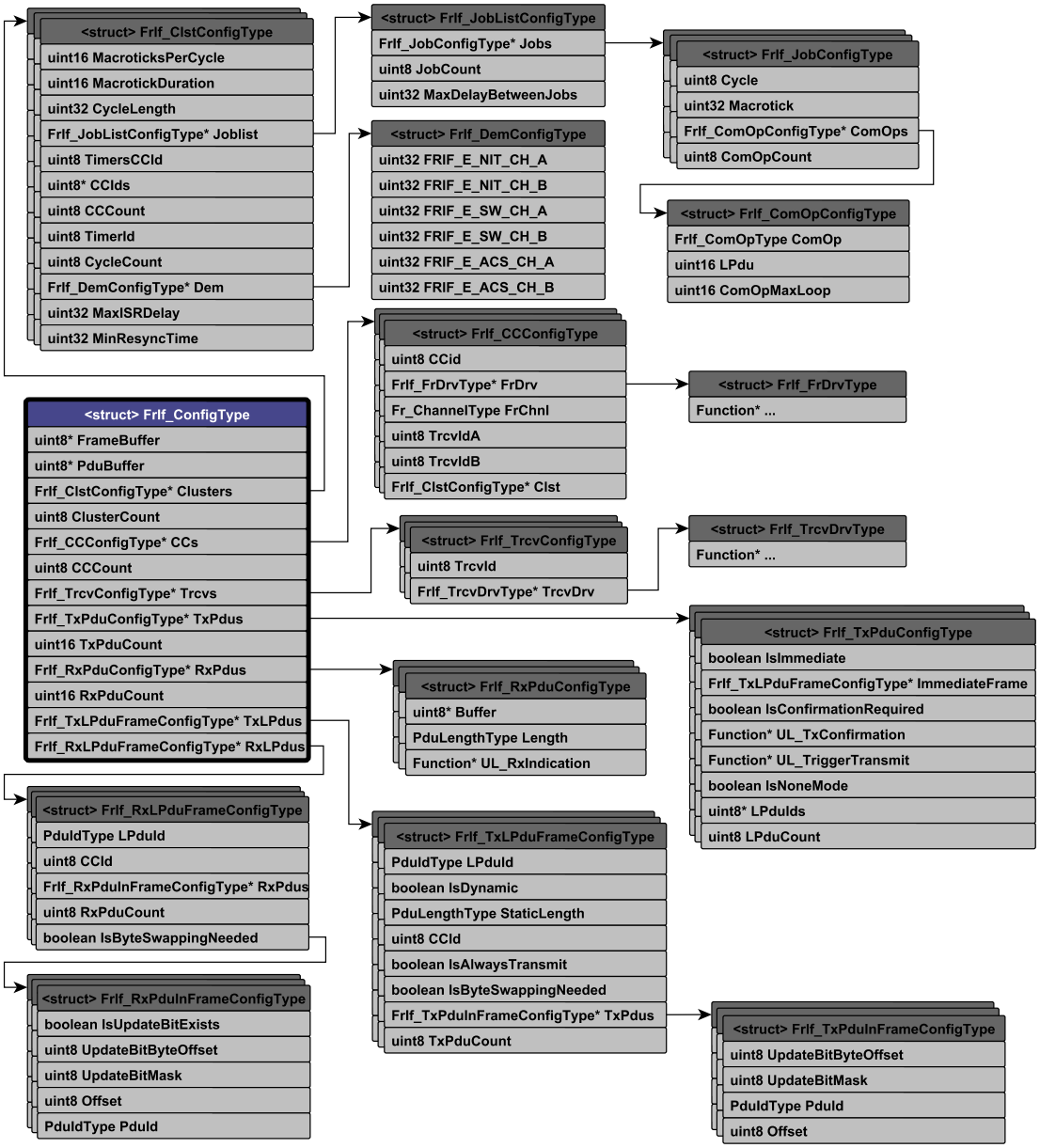
FrIf_RxLPduFrameConfigType: Egy Rx L-PDU konfigurációját tartalmazza. Az által tartalmazott PDU-król, a fogadásra kijelölt kontrollerről és az L-PDU bájtsorrendjéről tartalmaz információkat.

FrIf_RxPduInFrameConfigType: Egy Rx PDU L-PDU-n belüli elhelyezkedéséről, és Update Bit-jéről tartalmaz információkat.

3.3. A futásidejű adatszerkezet

A futásidejű – vagy más néven runtime – adatszerkezet felépítése az ábrán látható. *(3.4. ábra)* Az adatszerkezet összes eleme a kiindulási `FrIf_DataType` típusú struktúrán keresztül érhető el. A következőkben bemutatom az egyes struktúrák szerepét.

FrIf_DataType: Ez az adatszerkezet gyökere. Ezen keresztül érhetőek el a clusterek, a Tx PDU-k és az Rx PDU-k futásidejű adatai, valamint a modul inicializáltságát is jelzi.



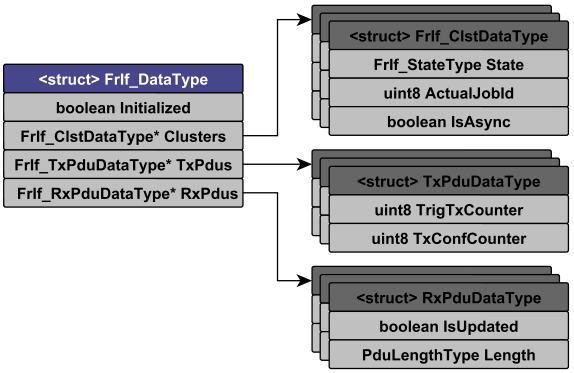
3.3. ábra. *A post-build time konfigurációs adatszerkezet*

FrIf_ClstDataType: Ez az adatszerkezet a clusterekre vonatkozó futásidejű adatokat tartalmazza. Megadja az adott cluster állapotát, az aktuálisan végrehajtandó Job indexét, valamint a cluster JobList végrehajtásának szinkronizáltságát.

FrIf_StateType: `enum` típus, melynek lehetséges értékei: `FRIF_STATE_ONLINE` és `FRIF_STATE_OFFLINE`.

FrIf_TxPduDataType: Az egyes Tx PDU-k elküldésére vonatkozó kéréseket és a várandó visszaigazolások számát számon tartó számlálók helyezkednek el benne.

FrIf_RxPduDataType: Az Rx PDU-k fogadására fenntartott bufferek állapotát tartja számon (új adatokat tartalmaz-e, mennyi adat van benne).



3.4. ábra. *A runtime adatszerkezet*

3.4. Az absztrakciót megvalósító függvények

Az absztrakciót megvalósító függvények felépítése és működése nagyon hasonló. A különbséget csak az jelenti, hogy az adott függvény a kontrollerek vagy a transceiverek driver függvényeit hívja-e tovább. Ebben a részben egy-egy példán keresztül bemutatom a kontroller és a transceiver driverek feletti absztrakciót megvalósító API függvények működését.

3.4.1. Hibakezelés

Ezekben a függvényekben két dolog vezethet hibás működéshez:

- Valamelyik – ebben a függvényben feldolgozandó – bemenő paraméter (ControllerIndex, ChannelIndex) nincs megfelelő tartományban.
- A modul nem lett inicializálva, így nem állnak rendelkezésére a megfelelő konfigurációs adatok.

A szabvány szerint amennyiben az `FRIF_DEV_ERROR_DETECT` pre-compile time konfigurációs paraméter be van kapcsolva, ezeket a lehetőségeket meg kell vizsgálni, az esetleges hibát jelenteni kell a Development Error Tracer modulnak, majd a függvény futását meg

kell szakítani, és a hibát jelző `E_NOT_OK` értékkel vissza kell térni belőle. Csak azokat a paramétereket kell vizsgálni, melyeket ebben a modulban kell feldolgozni. Azon paramétereket, melyeket a továbbhívott driver függvénye dolgoz fel, a driver modulnak kell ellenőriznie (illetve a hibát jelentenie).

A hibák jelentése a Development Error Tracer (DET) modul a `Det_ReportError(...)` függvényén keresztül történik. Ennek a függvénynek át kell adni a modult azonosító, szabvány által rögzített, úgynevezett ModuleId, az adott szoftverben érvényes InstanceId, a hiba keletkezésének helyére utaló, az adott függvényt azonosító ApiId, valamint a hiba okát azonosító ErrorId paramétereket.

3.4.2. A kontrollerek absztrakciós függvényei

A kontroller driverek feletti absztrakciót megvalósító függvények általános szignatúrája a következő:

```
Std_ReturnType FrIf_<függvénynév>(uint8 FrIf_CtrlIdx, ...);
```

A függvények visszatérési értéke `Std_ReturnType` típusú, mely hiba esetén `E_NOT_OK` értéket, sikeres lefutás esetén pedig a driver megfelelő függvényének visszatérési értékét veszi fel (ami a driverben történő hiba esetén `E_NOT_OK`, egyébként `E_OK`). Első paraméterük `FrIf_CtrlIdx` az absztrakt kontroller index, majd ezt követik a driver függvényének átadandó paraméterek. A driver függvények általános szignatúrája a következő (Az „ID” és „típus” mezők – több driver esetén – az adott driver azonosítására szolgálnak):

```
Std_ReturnType Fr_<ID>_<típus>_<függvénynév>(uint8 Fr_CtrlIdx, ...);
```

A függvények működése

A FlexRay kontroller driverek absztrakcióját megvalósító függvények működését egy konkrét függvény az `FrIf_EnableAbsoluteTimerIRQ(...)` példáján mutatom be. Két paramétere van: az `FrIf_CtrlIdx` és az `FrIf_AbsTimerIdx`. A függvény feladata engedélyezni a paraméterében megadott indexű kontroller, megadott időzítője által kiváltott megszakításokat.

Amennyiben a fejlesztési hibadetektálás engedélyezve van, a függvények végrehajtása a paraméterek ellenőrzésével, az 1. ponton kezdődik. Ha a fejlesztési hibadetektálás ki van kapcsolva, a függvények elején lévő paramétervizsgálat (a folyamatábrán a szaggatott vonallal határolt rész) nem fogja a forráskód részét képezni (a fordítás során az előfeldolgozó kiveszi a kódból). A MISRA szabvány előírásai szerint egy függvénynek csak egy kilépési pontja lehet. Ezt egy `ret` nevű lokális változó segítségével oldottam meg, melynek kezdeti értéke `E_NOT_OK`.

1. A modul inicializáltságának vizsgálata:
   IF `(FrIf_Data -> Initialized == TRUE)`

   THEN: Amennyiben a modul inicializált állapotban van, a végrehajtás folytatódik.

ELSE: Ha a modul még nem lett inicializálva a hibát jelenteni kell a Development Error Tracer modulnak: `Det_ReportError(...);` A végrehajtás a 4. ponton folytatódik, ahol a `ret` változó értéke kerül visszaadásra (`E_NOT_OK`).

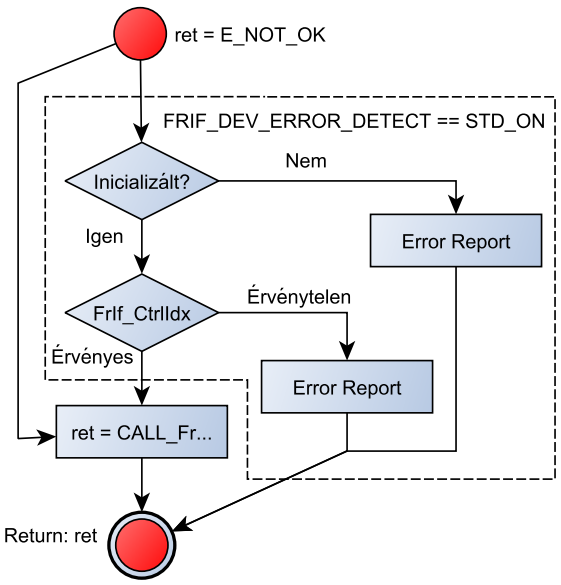
2. A ControllerIndex tartományának vizsgálata:
   IF (`FrIf_Config -> CCCount > FrIf_CtrlIdx`))

   THEN: Amennyiben a ControllerIndex a megfelelő tartományban van, a végrehajtás folytatódik.

   ELSE: Ha a ControllerIndex nincs a megfelelő tartományban, a hibát jelenteni kell a Development Error Tracer modulnak: `Det_ReportError(...);` A végrehajtás a 4. ponton folytatódik, ahol a függvény a `ret` változó értékét adja vissza (`E_NOT_OK`).

3. A megfelelő FlexRay driver függvény meghívása, és a visszatérési érték elmentése a `ret` változóba:
   `ret = CALL_Fr_EnableAbsoluteTimerIRQ(FrIf_CtrlIdx, FrIf_LPduIdx);`

4. Visszatérés a `ret` változó értékével (hiba esetén `E_NOT_OK`, egyéb esetben a FlexRay driver függvényének visszatérési értéke)
   `return ret;`



3.5. ábra. *A kontrollerek absztrakciós függvényeinek folyamatábrája*

A `CALL_Fr_EnableAbsoluteTimerIRQ(FrIf_CtrlIdx, FrIf_LPduIdx)` függvény takarja el az igazi működést. Ez igazából egy függvénymakró, mely a pre-compile time konfigurációs header állományban van definiálva. Minden FlexRay Controller Driver modul API függvényeihez definiálva van egy-egy `CALL_Fr_<függvénynév>(...)` függvénymakró. Ha

az optimalizálást növelő FRIF_SINGLE_FRDRV direktíva STD_ON (bekapcsolt) állapotban van (tehát az egész rendszerben csak egy FlexRay Controller Driver van), a makró definíciója az egyetlen driver megfelelő függvénye. Ebben az esetben nem szükséges az adott kontroller indexének transzformálása, hiszen egyetlen driver esetén az absztrakt ControllerIndex megegyezik a driveren belüli kontroller indexxel. Ha a rendszer több drivert tartalmaz, akkor a makró a post-build time konfigurációs struktúrán keresztül a megfelelő függvény pointer segítségével hivatkozik a driver függvényére. Ennek a függvénynek – szintén a konfiguráció alapján – az absztrakt ControllerIndex által kiválasztott, driveren belüli kontroller indexet, valamint a kapott FrIf_AbsTimerIdx paramétert adja át.

```
#if (FRIF_SINGLE_FRDRV == STD_ON)
#define CALL_Fr_EnableAbsoluteTimerIRQ(A, B) \
    Fr_<ID>_<típus>_EnableAbsoluteTimerIRQ(A, B)
#else
#define  CALL_Fr_EnableAbsoluteTimerIRQ(A, B) \
     (*FrIf_Config->CCs[A].FrDrv->EnableAbsoluteTimerIRQ) \
     (FrIf_Config->CCs[A].CCId, B)
#endif
```

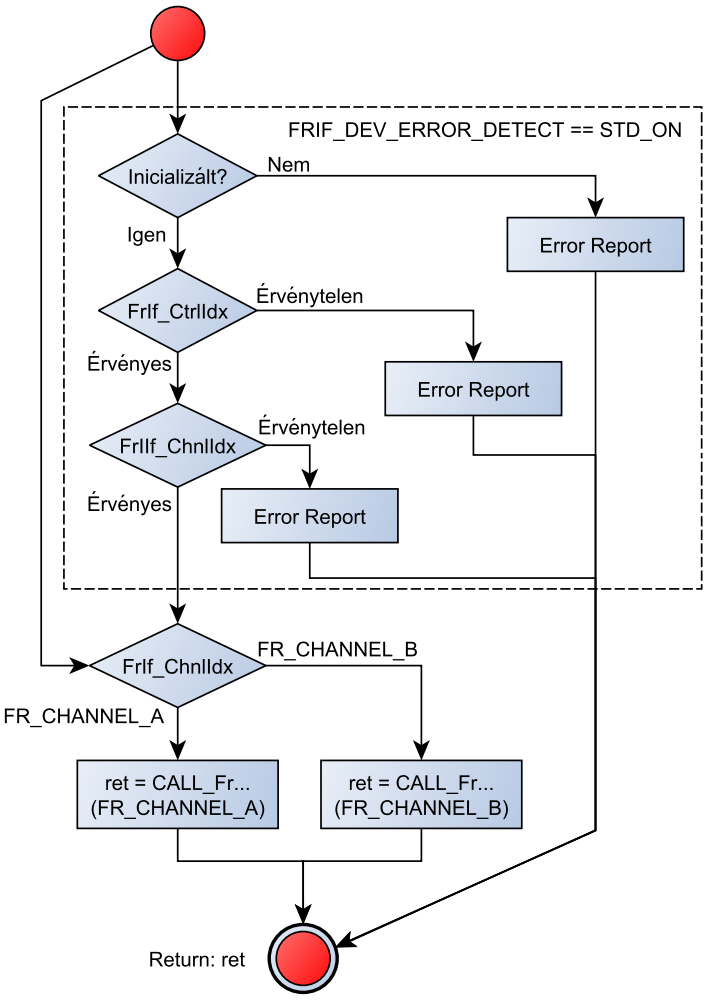
3.4.3. A transceiver-ek absztrakciós függvényei

A transceiver driverek fölötti absztrakciót megvalósító függvények analógok a kontroller driver függvényekkel. Egyedül annyi különbség jelentkezik, hogy az FRIF_DEV_ERROR_DETECT makró bekapcsolt állapota esetén a ChannelIndex érvényességét is vizsgálni kell. A ChannelIndex értéke nem veheti fel a FR_CHANNEL_AB értéket (hiszen egyszerre nem férhetünk hozzá adott esetben két különböző driverhez), valamint nem vehet fel olyan értéket, melyhez tartozóan nincs transceiver driver konfigurálva (például a kontroller adott csatornája nincs használatban).

A driver függvényének meghívása is kis mértékben eltér az előző példában ismertetettől. Mivel a ChannelIndex értéke futásidőben más és más lehet, mindenképpen meg kell vizsgálni az értékét, és annak megfelelően kiválasztani a konfigurációs struktúrából a megfelelő driver azonos függvényét. A vizsgálatokat követően a már előzőleg megismert módon történik a driver függvényének meghívása.

3.5. A JobList-ek végrehajtása

A szabvány előírásai szerint a FlexRay Interface modul API-ja annyi különálló FrIf_JobListExec_<ClstIdx>(void) függvényt kell szolgáltasson, ahány cluster van a rendszerben. Mivel a JLEF ugyanúgy működik minden cluster esetén, egy paraméterezhető FrIf_JobListExec(uint8 ClstIdx) függvényt hoztam létre. A szabvány által elvárt különálló függvényeket függvénymakrókkal valósítottam meg. Ezek a makrók a konfiguráció során az adatmodell alapján egyszerűen generálhatók. Ezzel a megoldással látszólag több különálló JLEF jött létre. Egy ilyen makró definíciója a 0. cluster esetén a következőképpen



3.6. ábra. *A transceiver-ek absztrakciós függvényeinek folyamatábrája*

néz ki:

```
#define FrIf_JobListExec_0() FrIf_JobListExec(0);
```

A JLEF által végrehajtandó kommunikációs műveleteket csak a modulon belül látható, (statikus) függvények formájában valósítottam meg. Részletezésükre a későbbiekben kerül sor.

3.5.1. A JLEF működése

A függvény a paraméterében kapott `ClstIdx` alapján az adott cluster JobList-jét hajtja végre. A modul a runtime adatstruktúrájában tárolja minden cluster számára a következő végrehajtandó Job indexét, melyet a végrehajtás után a soron következő Job-ra állít.

1. Az adott cluster állapotának ellenőrzése:
   IF `(FrIf_Data.Clusters[ClstIdx].State == FRIF_STATE_ONLINE)`

   THEN: Ha a cluster online állapotban van, folytatódik a JobList végrehajtása.

ELSE: Ha a cluster offline állapotban van, nem hajtódik végre a következő Job, a függvény végrehajtása az 5. ponton folytatódik.

2. A szinkronitás ellenőrzése:

   (a) A FlexRay ciklus aktuális időpontjának lekérdezése a cluster egyik – konfigurációjában, erre a célra kijelölt – kontrollerétől:
   `CALL_Fr_GetGlobalTime(..., &GlobalCycle, &GlobalMacrotick)`

   (b) A csúszás kiszámítása az aktuálisan végrehajtandó Job konfigurációja alapján, majd a konfigurációban megadott, megengedhető maximális késés figyelembevételével a szinkronitás ellenőrzése:
   IF `(Delay <= MaxISRDelay)`

   THEN: Ha a csúszás a megengedhető határon belül van, az aktuális Job-ot végre kell hajtani.

   ELSE: Ha a csúszás nagyobb a megengedhető maximumnál, a függvény jelzi a cluster JobList-jének kiesését a szinkronból, majd tiltja a clusterhez konfigurált AbsoluteTimer megszakítását. A függvény végrehajtása az 5. ponton folytatódik.
   `IsAsync = TRUE;`
   `CALL_Fr_DisableAbsoluteTimerIRQ(...);`

3. A Job által tartalmazott kommunikációs műveletek végrehajtása (ciklus).

4. A következő Job megjelölése a runtime adatstruktúrában.

5. Visszatérés a JLEF-ből: `return`

3.6. A MainFunction

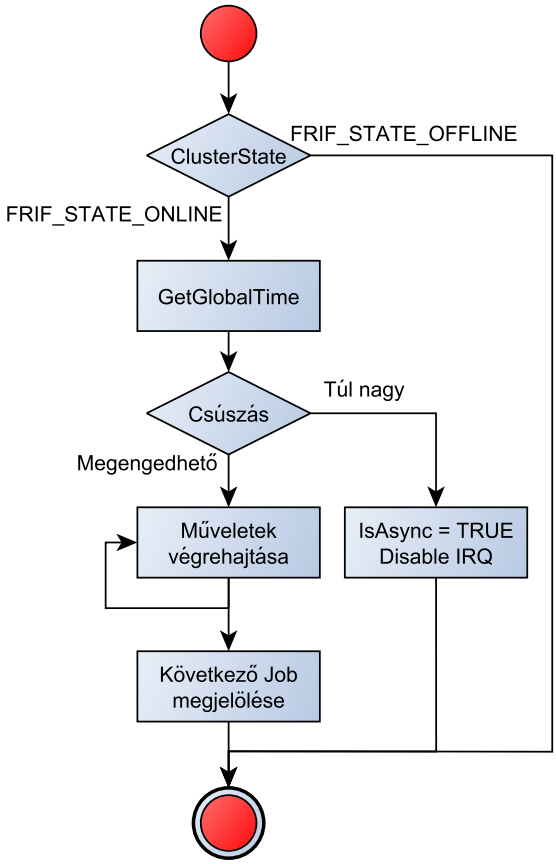
A modul Main függvénye, egy operációs rendszer task által periodikusan hívott függvény, melynek a JLEF-höz hasonlóan minden cluster számára külön léteznie kell. Ezt a már ismertetett függvénymakrók alkalmazásával oldottam meg:

```
#define FrIf_MainFunction_0() FrIf_MainFunction(0);
```

Feladata felügyelni a JLEF megfelelő működését, és szinkronizálatlanság esetén biztosítani a Job-ok végrehajtásának újraszinkronizálását. Emellett, ha az aktuális cluster esetén a konfiguráció előírja, diagnosztikai feladatokat is ellát, és az esetleges hibákat jelenti a Diagnostic Event Manager modulnak.

3.6.1. A MainFunction működése

1. Az adott cluster szinkronitásának ellenőrzése:
   IF `(FrIf_Data.Clusters[ClstIdx].IsAsync == TRUE)`



3.7. ábra. *A JobList-ek végrehajtásának folyamatábrája*

THEN: Ha a cluster Job-jainak végrehajtása kiesett a szinkronból, újra kell szinkronizálni.

ELSE: Ha a cluster JobList-je szinkronizált állapotban van, a függvény végrehajtása a 3. ponton folytatódik.

2. A JobList újraszinkronizálása:

   (a) A FlexRay ciklus aktuális időpontjának lekérdezése a cluster erre a célra kijelölt kontrollerétől:
   `CALL_Fr_GetGlobalTime(..., &GlobalCycle, &GlobalMacrotick);`

   (b) A konfigurációban megadott idő-buffer hozzáadása az aktuális időhöz. Erre azért van szükség, hogy adott esetben ne következhessen be túl hamar – még az újraszinkronizáció befejezése előtt – az új Job időpontja.
   `NextJobTime = GlobalTime + TimeBuffer;`

   (c) A következő – a kiszámított időhöz legközelebb eső – Job megkeresése. A keresés a cluster runtime adatstruktúrája által mutatott következő Job-bal kezdődik, mivel várhatóan ennek a közelében található a legközelebbi végrehajtandó Job.
   IF `(NextJobTime <= Job[i].Time)`

THEN: Ha a feltétel teljesül, megvan a következő végrehajtandó Job. A modul beállítja a cluster következő Job-ot jelző változóját erre az indexre, majd törli az aszinkronitást jelző flag-et. Felhúzza a megfelelő timer-t a Job végrehajtási idejére, majd engedélyezi a timer megszakítását. A végrehajtás a 3. ponton folytatódik.

```
NextJob = i;
IsAsync = TRUE;
CALL_Fr_SetAbsoluteTimer(...);
CALL_Fr_EnableAbsoluteTimerIRQ(...);
```

ELSE: Ha a feltétel nem teljesül, folytatódik a keresés.

3. Diagnosztikai teendők elvégzése (ha az aktuális cluster esetén szükséges):
   IF `(ActualCluster->Dem->IsNeeded == TRUE)`

   THEN: Ha szükséges a függvény elvégzi az esetleges hibák lekérdezését a cluster összes kontrollerétől.

   (a) A driver állapot-lekérdező függvényének meghívása:
   `CALL_Fr_GetChannelStatus(...)`
   A függvény az argumentumában megadott kontroller mindkét csatornájának állapot adatait visszaadja.

   (b) A függvény által szolgáltatott adatok elemzése (meghatározott bitek vizsgálata) és az esetleges hibák jelentése a DEM modulnak.
   `Dem_ReportErrorStatus(...)`

   ELSE: Ha nem szükséges, a végrehajtás a 4. ponton folytatódik.

4. Visszatérés a MainFunction-ből: `return`

3.7. Adatküldés a FlexRay Interface modulon keresztül

A következőkben az Interface adatküldéssel kapcsolatos függvényeit mutatom be. Az eddigiek során ismertetett, függvényekkel kapcsolatos általános megoldásokat itt már nem részletezem, csupán a függvények funkcionalitására koncentrálok.

3.7.1. A Transmit függvény

A függvény szignatúrája a következő:

```
FrIf_Transmit(PduIdType FrIf_TxPduId, PduInfoType* FrIf_PduInfoPtr)
```

Ezen függvényen keresztül valósul meg az adatok küldése. Két bemenő paramétere van:

FrIf_TxPduId: A továbbítandó PDU azonosítóját tartalmazza.

FrIf_PduInfoPtr: `PduInfoType` típusú struktúra, mely a PDU adatbájtjainak kezdőcímét és a PDU hosszát tartalmazza.

A függvény – amennyiben a Development Error Detect funkció engedélyezve van – elvégzi a modul inicializáltságának, valamint a bemenő paramétereknek a vizsgálatát. A bemenő paraméterekkel szemben a következőket várjuk el:

- A `FrIf_TxPduId` a konfigurált tartományban van.
- Az `FrIf_PduInfoPtr` nem null-pointer.
- Az `FrIf_PduInfoPtr` által tartalmazott adat pointer nem null-pointer.

Ha valamelyik paraméter nem megfelelő, vagy a modul nem lett inicializálva, a hibát a függvény jelenti a DET modulnak, és `E_NOT_OK`-kal tér vissza.

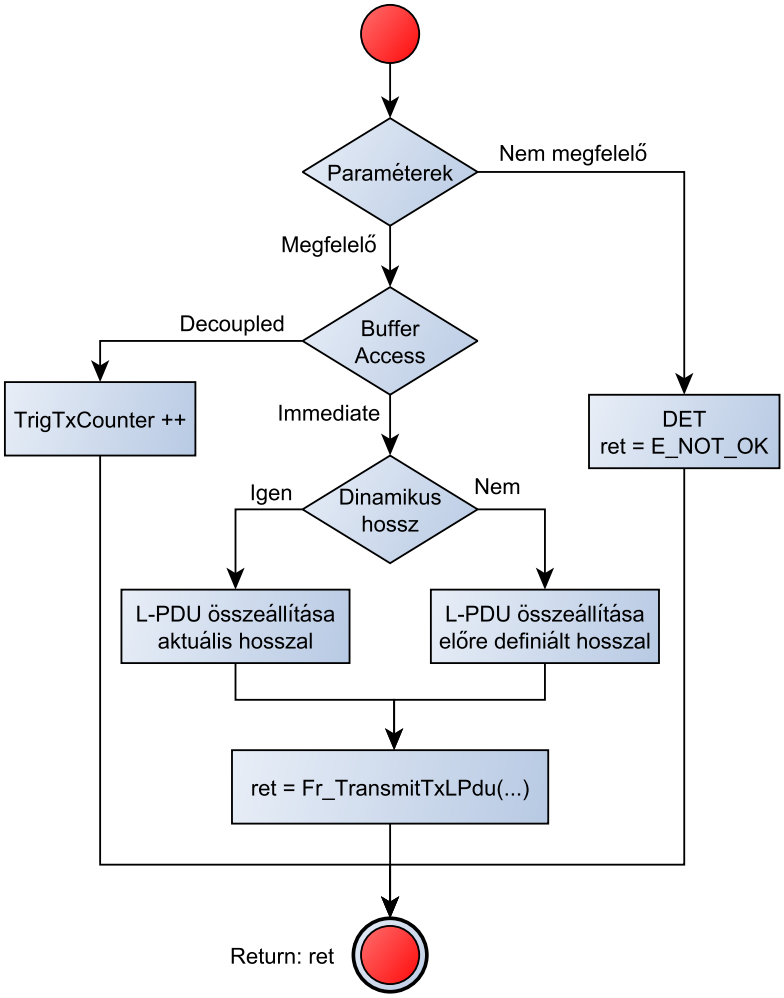
Ezek után a végrehajtás két módon folytatódhat. Amennyiben a kapott PDU az Interface konfigurációjában decoupled buffer access továbbítással van konfigurálva, a modul csupán megjegyzi a runtime adatstruktúrájában a PDU továbbítási kérését. Ezt egy számláló (`TrigTxCounter`) segítségével valósítja meg, melyet a PDU továbbítására érkező minden kérés esetén eggyel növel. A PDU küldését később, a JLEF ütemezi a `DECOUPLED_TRANSMISSION` kommunikációs művelet megfelelő időben történő végrehajtásával. Ha a PDU immediate buffer access-el van konfigurálva az Interface azonnal továbbítja a driver felé (`Fr_TransmitTxLPdu(...)`) egy olyan L-PDU formájában, melyben ez az egyetlen PDU foglal helyet. Az L-PDU elküldése után az PDU-hoz tartozó `TxConfCounter` értéke növelésre kerül.

Mivel ebben az esetben a PDU a FlexRay dinamikus szegmensében kell elküldeni, érdemes a lehető legrövidebb keretben elküldeni. Ez fix méretű PDU-k esetén előre konfigurálható, változó méretű PDU-k esetén pedig a „Dinamikus Hosszúság” (Dynamic Length) opció ad erre lehetőséget. Ha az adott L-PDU esetében a konfiguráció engedélyezi, az L-PDU összeállításánál az L-PDU mérete alkalmazkodhat a PDU méretéhez, így az elküldött keret hossza minimálisra csökkenthető. Ha ez a beállítás nem engedélyezett, az előre konfigurált L-PDU hosszt kell használni a küldésnél, függetlenül az PDU aktuális méretétől. Ebben az esetben a kihasználatlan biteket előre megadott értékre kell beállítani.

3.7.2. Adattovábbítás Decoupled Buffer Access esetén

Az adatok továbbítására függetlenített buffer hozzáférés esetén a JLEF által, ütemezetten végrehajtott `DECOUPLED_TRANSMISSION` kommunikációs művelet szolgál. A műveletet a `ComOp_DecoupledTransmission(FrIf_TxLPduFrameConfigType* FrIf_LPdu)` függvény valósítja meg. A függvény egyetlen bemenő paramétere az elküldendő L-PDU konfigurációjára mutató pointer.

A függvény egy `for` ciklussal végigmegy az L-PDU által tartalmazott összes PDU-n, és a runtime adatstruktúrában megvizsgálja, hogy az adott PDU továbbítására előzőleg érkezett-e kérés az `FrIf_Transmit` függvényen keresztül: `(TrigTxCounter > 0)`. (A konfigurációtól függően előfordulhat, hogy a `TrigTxCounter` vizsgálata nélkül, minden ciklusban továbbítani kell egy adott PDU-t.) Amennyiben az PDU-t el kell küldeni, az Interface elkéri azt a megfelelő felső modul `TriggerTransmit(...)` függvényén keresztül.



3.8. ábra. *A Transmit függvény folyamatábrája*

Ezek után a kapott PDU az L-PDU számára fenntartott buffer megfelelő (konfigurációtól függő) helyére kell, hogy kerüljön az PDU-hoz konfigurált bájtsorrend figyelembevételével. Mivel ehhez hasonló – esetlegesen bájtsorrend cserét igénylő – másolási műveletekre a modul működése során gyakran van szükség, erre egy külön függvényt hoztam létre:

```
FrIf_Copy(uint8* Source, uint8* Destination, uint8 Length, boolean Swap)
```

A függvény a `Source` által jelölt helyről a `Destination` által jelölt helyre másolja a `Length` által megadott számú a bájtot, a `Swap` által jelölt esetleges bájtsorrend figyelembevételével.

A másolás után a konfiguráció által az adott PDU-hoz kijelölt update bitet megfelelő értékűre kell állítani:

```
*(LPdu_Start + UpdateBitByteOffset) |= UpdateBitMask;
```

Ez a bit jelzi a fogadó oldalon lévő FlexRay Interface modulnak, hogy az PDU adatai érvényesek.

Amennyiben az PDU elküldését a `(TrigTxCounter > 0)` feltétel teljesülése váltotta ki, a számláló értéke csökkentésre kerül: `TrigTxCounter;` Ezután ha az PDU sikeres továb-

bításának visszaigazolása szükséges a felső modul számára (konfigurációs paraméter) az PDU-hoz tartozó `TxConfCounter` értékét a modul eggyel növeli.

Miután az L-PDU által tartalmazott összes PDU-t feldolgozta, az Interface dönt az L-PDU elküldésének szükségességéről. Az adott L-PDU-t továbbítani kell, ha legalább egy általa tartalmazott PDU frissítésre került, vagy ha a konfiguráció minden esetre előírja a továbbítását.

Mielőtt az L-PDU-t az Interface átadja a drivernek, a Transmit függvény esetén már megismert dinamikus hossz opció szerint dönt annak hosszáról. (Amennyiben a Dinamikus Hossz opció engedélyezve van, az L-PDU hosszát a benne elhelyezkedő utolsó frissítésre került PDU hossza, vagy az L-PDU utolsó elhelyezett update bitje határozza meg.) A nem használt bitek értékét az Interface ebben az esetben is előre definiált értékűre állítja be.

3.7.3. Visszaigazolás

A korábban elküldött L-PDU által szállított PDU-k sikeres továbbításának visszaigazolására a `TX_CONFIRMATIION` kommunikációs művelet szolgál. A művelet végrehajtását a JLEF ütemezi a megfelelő időpontra (a FlexRay ciklusnak arra az időpontjára, amikor az adott L-PDU-t a driver már továbbította a hálózaton). A művelet a `ComOp_TxConfirmation(FrIf_TxLPduFrameConfigType* FrIf_LPdu)` függvény valósítja meg.

A függvény először lekérdezi a drivertől a bemenő paraméterében kapott L-PDU állapotát: `CALL_Fr_CheckTxLPduStatus(...)`. Amennyiben a driver jelezte az L-PDU sikeres továbbítását, a függvény az L-PDU által tartalmazott összes PDU-n végigmegy és a `(TxConfCounter > 0)` feltétel teljesülése esetén – mely a visszaigazolás szükségességét jelzi – a sikeres továbbítást visszaigazolja a felső modulok számára a megfelelő modul `TxConfirmation(...)` függvényének meghívásával. Ezek után a modul csökkenti az adott PDU-hoz tartozó `TxConfCounter` értékét.

3.8. Adatfogadás a FlexRay Interface modulon keresztül

Az adatok fogadása az adatok küldéséhez hasonlóan történhet közvetlen (`RECEIVE_AND_INDICATE` kommunikációs művelet) és közvetett (`RECEIVE_AND_STORE` valamint `RX_INDICATION` kommunikációs műveletek) módon is.

3.8.1. Az adatok közvetlen feldolgozása

Az adatok közvetlen fogadását a JLEF által ütemezett `RECEIVE_AND_INDICATE` kommunikációs művelet végzi. A műveletet a következő privát függvény valósítja meg:

```
ComOp_ReceiveAndIndicate(FrIf_RxLPduFrameConfigType* FrIf_LPdu)
```

A függvény a paramétereként kapott L-PDU-t elkéri a drivertől (`CALL_Fr_ReceiveRxLPdu(...)`), majd amennyiben az L-PDU fogadása előzőleg a driver által sikeresen megtörtént, az Interface megkezdi annak feldolgozását. Végigmegy az összes általa tartalmazott PDU-n, a konfiguráció alapján megvizsgálja az update biteket

és amennyiben egy PDU érvényes adatot tartalmaz, a már ismertetett `FrIf_Copy(...)` privát függvény segítségével – az esetleges bájtsorrend konverziót is elvégezve – egy átmeneti bufferbe másolja azt. Ezek után meghívja a megfelelő felső modul `RxIndication(...)` függvényét, melyen keresztül átadja az PDU-t a megfelelő felső modulnak.

3.8.2. Az adatok közvetett feldolgozása

Az adatok közvetett fogadását a `RECEIVE_AND_STORE`, valamint a `RX_INDICATION` kommunikációs műveletek valósítják meg. Mindkettőt – az előzőekhez hasonlóan – a JLEF időzíti a kommunikációs ciklus megfelelő időpontjára.

RECEIVE_AND_STORE

A `RECEIVE_AND_STORE` műveletet szintén egy privát függvény valósítja meg:

```
ComOp_ReceiveAndStore(FrIf_RxLPduFrameConfigType* FrIf_LPdu)
```

A `RECEIVE_AND_STORE` a `RECEIVE_AND_INDICATE` művelettel szinte teljesen megegyezik, azzal a különbséggel, hogy az átmeneti, lokális bufferekbe másolt PDU-kat nem adja át egyből a felső rétegbeli modulnak. Ehelyett a runtime adatstruktúrában az PDU-t „frissítettnek” jelöli (`IsUpdated = TRUE`), és a felső modulok értesítését egy későbbi időpontban az `RX_INDICATION` kommunikációs művelet végrehajtása során teszi meg.

RX_INDICATION

A `RECEIVE_AND_STORE` művelet során eltárolt PDU-kat továbbítja a felső moduloknak. Az `RX_INDICATION` műveletet a következő függvény valósítja meg:

```
ComOp_Rx_Indication(FrIf_RxLPduFrameConfigType* FrIf_LPdu)
```

A függvény a paraméteréül kapott L-PDU által szállított PDU-k bufferein iterál végig, és amennyiben a bufferek valamelyike „frissítettnek” van jelölve (`IsUpdated == TRUE`), meghívja a felső modul `RxIndication(...)` függvényét, melyen keresztül a megfelelő felső modul számára átadja az PDU-t. Ezek után a buffer állapotát visszaállítja (`IsUpdated = FALSE`).

3.9. Erőforrásigény

A modul erőforrásigényének szempontjából a konfigurációnak két esetét különböztethetjük meg:

- Egy transceiver vagy kontroller driver van a rendszerben
- Több transceiver vagy kontroller driver van a rendszerben

Az erőforrásigény vizsgálatánál az `FRIF_DEV_ERROR_DETECT` – fejlesztési hiba ellenőrzést engedélyező – makrót `STD_OFF` (kikapcsolt) értékűnek feltételezem, hiszen az integrációt és tesztelést követően, a végleges szoftverben sem lesz engedélyezve. A fejlesztési hibák

detektálásából származó overhead (fölösleges többletterhelés) nem jelentkezik a végtermék esetén.

3.9.1. Erőforrásigény egy driver esetén

A legtöbb esetben az ECU-ban egy FlexRay kontroller van. Ennek megfelelően egy FlexRay Driver modul lesz integrálva a rendszerbe.

Ebben az esetben a pre-compile time konfiguráció `FRIF_SINGLE_FRDRV` makrójának bekapcsolása a forráskódban egyszerűsítéseket hajt végre. Ilyenkor a modulra az absztrakció megvalósításának szempontjából nincsen szükség, hiszen egyetlen driver esetén az absztrakt ControllerIndex megegyezik a kontrollerek driveren belüli indexével. Az erre a célra létrehozott makrók bekapcsolásával a modul működése egyszerűsödik azáltal, hogy eliminálhatunk egy indirekciót a driver függvényének hívásakor. Az FrIf absztrakciót megvalósító API függvényei ebben az esetben függvénymakrók, így a függvények meghívásának helyére a fordítás során az Interface konfigurációjában megadott driverfüggvények hívásai kerülnek. A fejlesztési hibadetektálás kikapcsolásával az absztrakciót megvalósító függvények törzse csupán a paraméterek egy másik függvénynek történő átadásából áll. A fordító az optimalizáció során ezt végül – inline függvényként megvalósítva (a függvény hívásának helyére a függvény törzsének behelyettesítése) – odáig egyszerűsíti, hogy az FrIf API függvényének hívási helyére a driver megfelelő függvényének hívása kerül. Ezzel a megoldással az absztrakciót megvalósító függvények esetén nem jelentkezik overhead.

Ez sajnos a transceiver függvények esetében nem ennyire egyszerű. Az `FRIF_SINGLE_TRCVDRV` bekapcsolása csupán azt eredményezi, hogy a megfelelő driver függvény egy makró alapján lesz meghívva, nem pedig a konfiguráció alapján kiválasztott függvénypointeren keresztül. A driveren belüli transceiver indexet azonban nem lehet egyből átadni a driver függvényének, mivel a transceivereket az Interface felett a ControllerIndex és a ChannelIndex azonosítja, így a drivernek átadandó indexet mindenképpen a modul konfigurációja alapján kell meghatározni. A transceiverek absztrakciós függvényei által jelentett overhead egy többszörös indirekción keresztül történő memóriaolvasás és egy függvényhívás.

3.9.2. Erőforrásigény több driver esetén

Több driver esetén a fenti behelyettesítéses megoldás nem alkalmazható, hiszen mindenképpen a konfigurációs struktúrából kell a modulnak – futásidőben – meghatároznia a hívandó függvényt és annak paramétereit. A jelentkező overhead a FlexRay Driver és a FlexRay Transceiver Driver modulok függvényeinek hívása esetén is egyaránt az előző esetben már ismertetett többszörös indirekción keresztül történő memóriaolvasás és függvényhívás.

Ez az overhead azonban onnantól kezdve, hogy legalább két driver van a rendszerben, nem függ az alkalmazott driverek számától, mivel a konfigurációban minden adat a megfelelő szempontok szerint rendezve helyezkedik el. A paraméterekben kapott indexek minden esetben használhatók a konfigurációs adatok tömbjeinek indexelésére.

3.9.3. A konfigurációs és futásidejű adatstruktúrák memóriaigénye

A konfigurációs és futásidejű adatstruktúrák memóriaigénye minden esetben az adott objektum (melynek adatait a struktúra tartalmazza) számának lineáris függvénye, hiszen ahány kontroller van a rendszerben, annyi a kontrollerek paramétereit leíró struktúrának kell elhelyezkednie egy tömbben a konfigurációban is. Ez igaz a clusterek, transceiverek és PDU-k esetére is, mind a konfigurációs mind a futásidejű adatok tekintetében.

3.9.4. A JLEF és a MainFunction erőforrásigénye

A JLEF erőforrásigényét egyedül a futásidő tekintetében van értelme vizsgálni. A függvény futásideje minden cluster esetében egyedül attól függ, hogy az éppen végrehajtandó Job hány kommunikációs műveletet tartalmaz, és a kommunikációs műveletek milyen bonyolultságú és méretű L-PDU-hoz vannak rendelve. Ezektől a paraméterektől a futásidő lineárisan függ.

A MainFunction futásideje – normál esetben – a diagnosztikai ellenőrzések és értesítések szükségességétől függ. Amennyiben a diagnosztikai ellenőrzések be vannak kapcsolva, ez a legrosszabb esetben akár hét függvényhívást is jelenthet.

Ha újraszinkronizálásra is szükség van, akkor ehhez még hozzájön a FlexRay globális idejének lekérdezése majd a Job megkeresése a konfigurációban, valamint az AbsoluteTimer beállítása és a megszakítás engedélyezése. A Job megtalálása nagy valószínűséggel néhány lépésben megtörténik, azonban az idő lekérdezése és a Timer kezelése három függvényhívásba kerül.

4. fejezet

A FlexRay Transport Layer modul implementálása

Az AUTOSAR két Transport Layer modult definiál. Az egyik a konzorcium által kidolgozott protokoll szerint működő FlexRay AUTOSAR Transport Layer modul, a másik pedig az ISO 10681-2 „Road vehicles – Communication on FlexRay – Part 2: Communication layer services" szabványban [19] definiált kommunikációs protokollt megvalósító FlexRay ISO Transport Layer modul. Feladatom az ISO szabványnak megfelelő Transport Layer implementálása volt.

Ebben a fejezetben először bemutatom a protokoll ISO szabvány által definiált működését, majd az AUTOSAR által az implementációra vonatkozó előírásokat. Ismertetem az általam létrehozott konfigurációs és futásidejű adatszerkezeteket, valamint a modul működését megvalósító algoritmusokat. Utóbbi esetében – a protokoll bonyolultsága és a modul összetett működése miatt – az adatküldés megvalósítására szorítkozom, mivel az adatok fogadását végző algoritmusok működésének alapelve nagyon hasonló. A fejezet végén becslést adok a modul erőforrásigényére.

A modul tárgyalása során már nem térek ki az implementáció általános – a FlexRay Interface esetében – már tárgyalt kérdéseire, helyette a működés bemutatására koncentrálok. A szükség szerint engedélyezhető illetve tiltható fejlesztési hibadetektálás az Interface-hez hasonlóan, a bemenő paraméterek és a modul inicializáltságának vizsgálatával, majd a Development Error Tracer (DET) modul értesítésével történik.

4.1. A protokoll ISO szabvány szerinti működése

A protokoll a következő átviteli módokat támogatja:

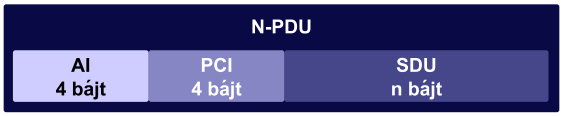
- Nem-szegmentált, nem-visszaigazolt adatátvitel ismert hosszal
- Nem-szegmentált, visszaigazolt adatátvitel ismert hosszal
- Szegmentált, nem-visszaigazolt adatátvitel ismert hosszal
- Szegmentált, visszaigazolt adatátvitel ismert hosszal

- Szegmentált, nem-visszaigazolt adatátvitel ismeretlen hosszal
- Szegmentált, visszaigazolt adatátvitel ismeretlen hosszal

Ennek megvalósításához különböző típusú, szabvány által definiált PDU-kat használhat fel.

4.1.1. Az ISO szabvány által definiált PDU-k

A PDU-k általános felépítése az ábrán látható. *(4.14. ábra)*



4.1. ábra. *Az N-PDU-k felépítése*

AI - Address Information
Ez a 4 bájtos mező szolgál az átvitel során a kommunikációs folyamat azonosítására. Az első két bájt az úgynevezett Source Address (SA) vagy forrás cím, a második két bájt pedig a Target Address (TA) vagy cél cím.

PCI - Protocol Control Information
Tartalmát a későbbiekben részletezem.

SDU - Service Data Unit
Ez a mező tartalmazza a PDU által szállított adatbájtokat.

4.1. táblázat. *A PCI bitjei*

A PCI bitjei					
A PDU típusa	**1. bájt**		**2. bájt**	**3. bájt**	**4. bájt**
	7-4. bit	3-0. bitek			
Start Frame (STFU)	4	0	FPL	ML	
Start Frame ACK (STFA)	4	1	FPL	ML	
Consecutive Frame 1 (CF_1)	5	SN	FPL		
Consecutive Frame 2 (CF_2)	6	SN	FPL		
Consecutive Frame EOB (CF_EOB)	7	SN	FPL		
Flow Control (FC_CTS)	8	FS=CTS	BC	BfS	
Flow Control (FC_ACK_RET	8	FS=ACK_RET	ACK	BP	
Flow Control (FC_WT)	8	FS=WT			
Flow Control (FC_ABT)	8	FS=ABT			
Flow Control (FC_OVFLW)	8	FS=OVFLW			
Last Frame (LF)	9	0	FPL	ML	

A PCI első bájtjának felső négy bitje a PDU típusát határozza meg, az alsó négy bit pedig a típuson belüli funkcióját pontosítja. A következő három bájt egyéb információkat közöl. *(4.1. táblázat)*

Start Frame - STFU, STFA

Minden adatátviteli szekvencia ezzel a PDU-val kell kezdődjön. Két típusa van: Az STFU a nem-visszaigazolandó (unacknowledged), az STFA pedig a visszaigazolandó (acknowledged) átvitel kezdetét jelzi. Ebben a keretben már megkezdődik az adatok első részének továbbítása. A PCI FPL (Frame Payload Length) mezője az ebben a frame-ben átvitt adatbájtok számát mutatja. Az ML (Message Length) a teljes átviendő üzenet hosszát jelzi. Amennyiben a kettő megegyezik – vagyis ebben a frame-ben megtörténik a teljes üzenet továbbítása – nem-szegmentált átvitel valósul meg. Ha az ML mező által jelzett érték nagyobb mint az FPL, további keretekre kell számítani, tehát szegmentált átvitel kezdődik.

Consecutive Frame - CF_1, CF_2, CF_EOB

Szegmentált adatátvitel esetén, az STF PDU-t követően ezekben a PDU-kban valósul meg az adatszegmensek továbbítása. Három típusa van:

CF_1: A CF_1 szolgál normál esetben az adatszegmensek átvitelére.

CF_2: A CF_2 hiba esetén a szegmensek megismétlésére szolgál (ez jelzi a fogadó fél számára, hogy a keret által tartalmazott adatok már megismételt adatok).

CF_EOB: A CF_EOB (End Of Bytes) az adott átviteli blokk (lsd. később) utolsó keretét jelzi.

Az SN (Sequence Number) mező az adott keret átviteli blokkon belüli sorszámát (lsd. később), a már ismertetett FPL mező pedig a keret által tartalmazott adatbájtok számát mutatja.

Flow Control - FC_CTS, FC_ACK_RET, FC_WT, FC_ABT, FC_OVFLW

Ezeket a PDU-kat az átviteli folyamat közben a címzett küldi vissza a feladó számára. Az FC PDU-kkal a címzettnek lehetősége van az adatok áramlásának befolyásolására, illetve hibák jelzésére, valamint visszaigazolások küldésére. Az FS (Flow Status) mező az FC PDU-k öt típusát határozza meg.

FC_CTS: Az FC_CTS (Continue To Send) PDU kéri a feladót az adattovábbítás (CF PDU-k küldésének) folytatására a BfS (Buffer Size) mezőben megadott számú adatbájt erejéig. STFU, STFA vagy CF_EOB frame-ek után küldi vissza a címzett. BfS = 0 esetén nincs limit az egy blokkban elküldhető adatbájtok számára.

A BC (Bandwidth Control) mező a sávszélesség szabályozására szolgáló információkat közöl a feladóval. A felső öt bitje az egy FlexRay ciklusban küldhető PDU-k maximális számát megadó MNPC (Maximum Number of PDUs Per Cycle) mezőt, az alsó három bitje pedig a PDU csoportok elküldése között kivárandó FlexRay ciklusok kiszámítására szolgáló SCexp (Separation Cycle Exponent) mezőt tartalmazza. $SC = 2^{SCexp} - 1$

FC_ACK_RET: Az FC_ACK_RET (Acknowledge/Retry) PDU szolgál az adattovábbítás befejeztével – visszaigazolt átvitel esetén – a visszaigazolásra, vala-

mint adattovábbítás közben, hiba esetén az adatok újraküldésének kérésére. Az ACK (Acknowledge) mező jelzi, hogy a PDU visszaigazolást (ACK) vagy újraküldést (RET - Retry) kér. Újraküldés esetén a BP (Byte Pointer) mező jelzi, hogy az adott blokk hányadik bájtjától kéri a fogadó egyég az adatok újraadását.

FC_WT: Az FC_WT (Flow Control Wait) keret az adatok küldésének ideiglenes felfüggesztését kéri. A küldő fél nem küldhet további PDU-kat, amíg nem kap egy FC_CTS (adattovábbítás folytatását kérő) PDU-t.

FC_ABT: Az FC_ABT (Flow Control Abort) PDU-ban a fogadó egység az átvitel megszakítását kérheti a küldő féltől.

FC_OVFLW: Az FC_OVFLW (Flow Control Overflow) PDU-t STFU vagy STFA PDU után küldheti vissza a címzett, azt jelzve, hogy a Start Frame-et, vagy a Start Frame ML mezőjében megadott hosszúságú üzenetet nem tudja fogadni. A küldő fél ennek hatására fel kell függessze az átvitelt.

Last Frame - LF

Szegmentált átvitel esetén az átvitel lezárását jelző PDU. Lényegében ez az utolsó CF PDU. Ennek hatására – ha visszaigazolás szükséges – az adatokat fogadó egység visszaigazolja az átvitel sikerességét a küldő egység számára, majd az összeállított I-PDU-t átadja a felső modulnak. FPL mezője az ebben a PDU-ban szállított utolsó adatbájtok számát tartalmazza. Az ML mező pedig az átvitel folyamán összesen elküldött adatbájtok számát tartalmazza.

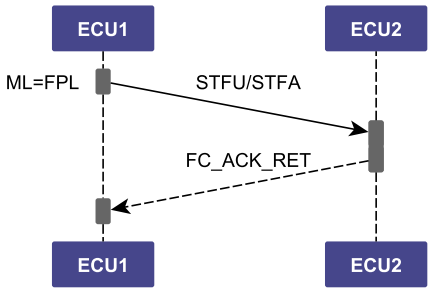
4.2. ábra. *Az SC és MNPC mezők jelentése*

4.1.2. Nem-szegmentált adatátvitel ismert hosszal

Ha az átviendő teljes I-PDU elfér egy PDU-ban, akkor az adatok szegmentálás nélkül továbbíthatók. Ebben az esetben funkcionálisan nem is lenne szükség a Transport Layer-re,

csupán az esetleges visszaigazolás szükségessége indokolhatja a modulon keresztül történő adattovábbítást.

A folyamat egyetlen Start Frame-ből (STFU vagy STFA) – mely a teljes I-PDU-t tartalmazza – és opcionálisan a fogadó féltől érkező visszaigazolásból áll. *(4.3. ábra)* A Start Frame-ben az ML és az FPL mezők értéke megegyezik, jelezve, hogy a keret a teljes üzenetet szegmentálás nélkül tartalmazza. Amennyiben az átvitel sikerességének visszaigazolása szükséges, a fogadó egység egy FC_ACK_RET Flow Control üzenetben az átvitelt visszaigazolhatja, vagy kérheti annak megismétlését.



4.3. ábra. *Nem-szegmentált adatátvitel*

4.1.3. Szegmentált adatátvitel ismert hosszal

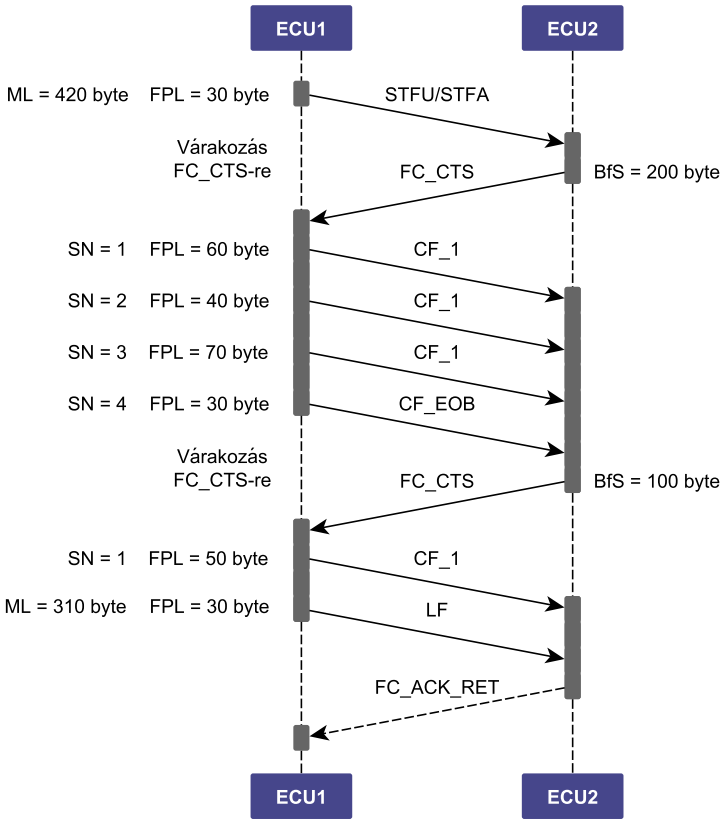
Ha a továbbítandó I-PDU nem fér el egyetlen PDU-ban, szegmentálásra – vagyis az üzenet darabokban való átvitelére – van szükség. *(4.4. ábra)* Ebben az esetben a Start Frame-et követően blokkokban történik az adatok átvitele.

Az első lépésben a küldő egység továbbítja a Start Frame-et (STFU vagy STFA), melyben az ML > FPL jelzi a fogadó fél számára, hogy szegmentált átvitelre kerül sor. Ezek után a feladó vár a címzett válaszára. A címzett az FC_CTS üzenetben kéri a feladót az átvitel folytatására, és egyben megadja az üzenet BfS mezőjében az egy átviteli blokkban küldhető bájtok számát.

Ezek után a feladó megkezdi az első adatátviteli blokkot. Addig folytatja az adatok továbbítását CF_1 keretekben, amíg el nem éri a fogadó fél által az FC_CTS üzenet BfS mezőjében megadott, egy átviteli blokkban elküldhető bájtok számát. A blokk utolsó kerete CF_EOB keret, mely jelzi a címzett számára, hogy az adott blokk befejeződött. A feladó ismét a címzett válaszára vár, hogy folytathassa az adatátvitelt a következő blokkban.

A címzett fél ismét kéri a feladót az átvitel folytatására egy FC_CTS üzenet elküldésével, melyben ismét megadja a következő blokkban elküldhető bájtok számát. A címzett folytatja a CF_1 keretek továbbítását. A blokkok addig követik egymást, míg az összes adatbájt továbbításra nem került. Az utolsó bájtokak és az átvitel végét az LF (Last Frame) keret jelzi. Az LF keret ML mezője ismét az üzenet teljes hosszát tartalmazza.

Az LF keret vétele után a címzett – ha szükséges (STFA kerettel kezdődött az átvitel) – visszaigazolja az üzenet átvitelét a küldő számára, majd továbbítja az összeállított, teljes üzenetet a felette lévő rétegnek.



4.4. ábra. *Szegmentált adatátvitel*

4.1.4. Adatátvitel ismeretlen hosszal

A FlexRay ISO Transport Layer lehetőséget biztosít előre nem ismert hosszúságú üzenetek átvitelére. Erre például akkor lehet szükség, ha az ECU gateway-ként üzemelve, egy másik ECU-tól érkező üzenetet továbbít egy harmadik ECU-felé. Ha az üzenet nagyon hosszú, ezzel a megoldással nem szükséges a teljes üzenet megérkezésének megvárása és bufferelése, hanem az első részlet megérkezését követően egyből megkezdhető annak továbbítása.

Ebben az esetben a Start Frame-ben (mely az üzenet első részletét is tartalmazza) szereplő ML mező értéke 0. Ez jelzi a fogadó egység számára, hogy a továbbítandó üzenet hossza előre nem definiált. Ezek után a már ismert módon, szegmentált átvitellel folytatódik az üzenet továbbítása. Miután az egész üzenet átvitele megtörtént (elfogyott a küldendő adat), a Last Frame (LF) ML mezőjében a küldő egység közli az összesen átvitt adatbájtok számát. Ezt a fogadó egység összehasonlítja az általa vett adatbájtok számával, és amennyiben a két érték megegyezik, az adatátvitel sikeres volt. Ezután az ismert módon – ha szükséges – megtörténhet a visszaigazolás.

4.1.5. Időzítési paraméterek

Az ISO szabvány a kommunikációs folyamat minden részletére szigorú időtúllépési paramétereket határoz meg. *(4.5. ábra)* (A paraméterek végén az „s” illetve „r” a teljes folyamat

szempontjából nézve, a küldő illetve a fogadó oldalt jelenti.) Időtúllépés esetén a kommunikációs folyamatot meg kell szakítani. A következő paramétereket kell figyelembe venni:

As: Maximális idő, amin belül egy keret elküldése sikeresen meg kell, hogy történjen.

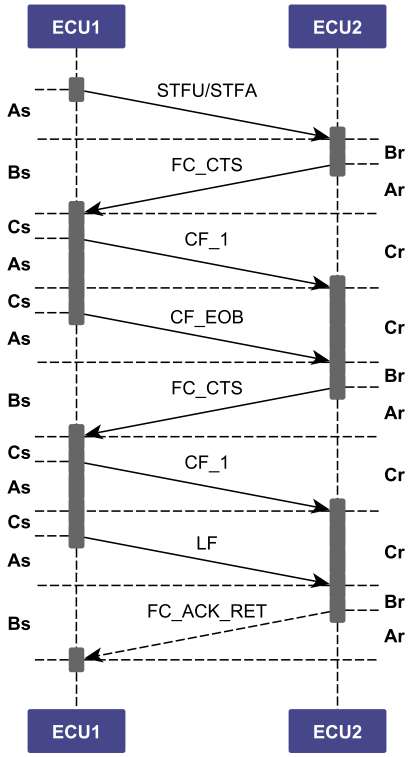
Ar: Maximális idő, amin belül egy keret elküldése sikeresen meg kell, hogy történjen.

Bs: Maximális idő, amin belül egy válasz keretnek meg kell érkeznie.

Br: Maximális idő, amin belül meg kell kezdeni egy válasz keret küldését.

Cs: Maximális idő, amin belül meg kell kezdeni egy következő keret elküldését.

Cr: Maximális idő, amin belül meg kell érkeznie a következő keretnek.



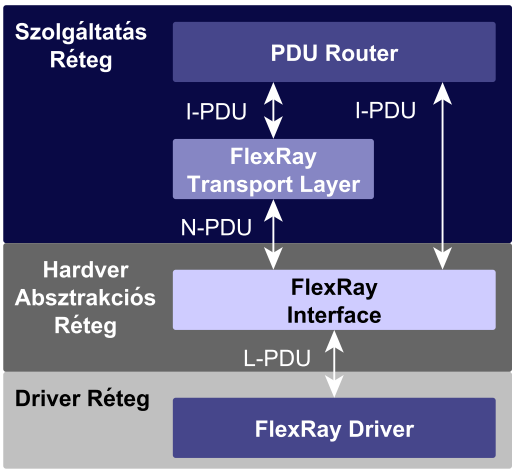
4.5. ábra. *Az előírt időzítések*

4.2. A modul AUTOSAR szerinti működése

Az AUTOSAR ennek a modulnak csupán a többi BSW modulhoz való kapcsolódását, interfészeit és konfigurációs lehetőségeit írja elő, valamint ajánlást tesz a modul implementációjára és az erőforrások kezelésére. Segítséget nyújt az ISO szabvány által definiált nyelv- és implementáció független szolgálati primitívek AUTOSAR API függvényeknek való megfeleltésében. Minden egyéb kérdésben az ISO szabványra [19] támaszkodik.

A FlexRay ISO Transport Layer BSW modul a PDU Router és a FlexRay Interface között helyezkedik el. *(4.6. ábra)* Feladata a FlexRay Interface-en keresztül maximálisan

elküldhető hosszúságot meghaladó I-PDU-k darabolása, majd a túloldalon való összeállítása. Mivel képes megbízható kapcsolat kiépítésére az üzenetet fogadó ECU Transport Layer-ével, használható olyan kisebb adatok továbbítására is, melyek esetében fontos megbizonyosodnunk arról, hogy a fogadó ECU valóban megkapta és feldolgozta azt.



4.6. ábra. *A FlexRay Transport Layer környezete*

A PDU Router a konfigurációja alapján dönt arról, hogy melyik I-PDU-t kell közvetlenül a FlexRay Interface-en keresztül, és melyiket a FlexRay Transport Layer közbeiktatásával továbbítani. A Transport Layer által előállított PDU-k neve N-PDU. (A továbbiakban a PDU szó alatt minden esetben N-PDU értendő.) A modul az összeállított PDU-kat az Interface-nek továbbítja, ahol megtörténik azok L-PDU-ba csomagolása majd továbbítása a hálózaton. Adatfogadás esetén a hálózaton érkező PDU-kat – az L-PDU-ból való kibontás után – az Interface további feldolgozásra közvetlenül a Transport Layer-nek továbbítja, ahol megtörténik az I-PDU összeállítása, majd továbbítása a felső rétegek felé.

Channel

A Channel-ök (csatornák) a modul futásidejű erőforrásai. Jelképesen ezeken keresztül valósul meg az adatátvitel. A csatornák feladata az átviteli folyamatok vezérlése és az adatok továbbítása, illetve feldolgozása.

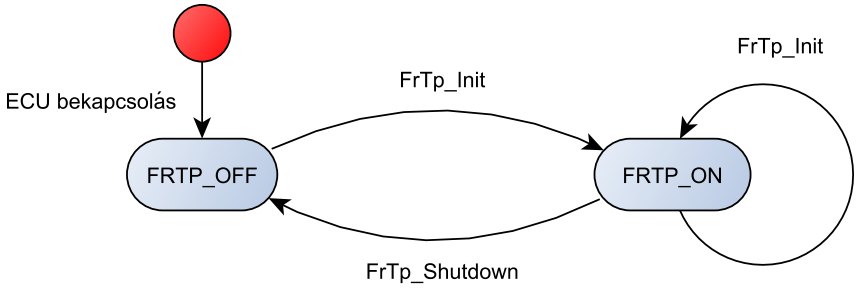
Számuk limitált (konfigurációs paraméter), így az egyszerre kezelhető kommunikációs folyamatok számát is megszabja. A csatornák valósítják meg a folyamatokat vezérlő állapotgépeket, időzítőket és számlálókat, valamint számon tartják az elküldött PDU-k állapotait. Fontos, hogy a Channel-ök nem azonosak a FlexRay fizikai csatornáival.

MainFunction

A modul `FrTp_MainFunction(void)` API függvénye egy operációs rendszer taszk által ütemezett, periodikusan meghívott függvény. Ez a függvény vezérli az adattovábbítási és adatfogadási folyamatokat.

4.2.1. A modul állapotai, inicializálása

A modul az ábrán látható alapvető állapotokkal rendelkezik. *(4.7. ábra)* Bekapcsolás, vagy reset után az inicializálatlan `FRTP_OFF` állapotba kerül. Ebben az állapotban API függvényei – az inicializálást végző `FrTp_Init(...)` függvényen kívül – nem hívhatók. Innen az inicializáció során léphet `FRTP_ON` állapotba, melynek következtében a modul minden belső változója alaphelyzetbe kerül, valamint a post-build time konfigurációra mutató pointert is ekkor kapja meg az FrTp. A modul másik konfigurációval bármikor újrainicializálható az `FrTp_Init(...)` függvényének ismételt meghívásával. Az `FRTP_ON` állapotból a modul ismét `FRTP_OFF` állapotba kerül az `FrTp_Shutdown(...)` függvény hatására.



4.7. ábra. *A FlexRay Transport Layer modul állapotai*

4.2.2. A modul által elvárt API függvények

Az FrTp a felette lévő modul – PDU Router – következő API függvényeit hívhatja adatküldés esetén:

PduR_FrTpCopyTxData

```
BufReq_ReturnType PduR_FrTpCopyTxData(
    PduIdType id,
    PduInfoType* info,
    RetryInfoType* retry,
    PduLengthType* availableDataPtr
);
```

Ez a függvény egy I-PDU megadott számú bájtjának felső modultól való elkérésére szolgál. Az `id` paraméterében megadott ID-val rendelkező I-PDU az `info` paraméter által meghatározott számú adatbájtját a szintén az `info` paraméter által meghatározott bufferbe másolja. A `retry` által mutatott `RetryInfoType` típusú struktúrának a `TpDataState` eleme szerint három jelentése lehet:

TP_CONFPENDING: Az eddig kimásolt adatok és az ebben a műveletben elkért adatok továbbra is maradjanak a felső modul bufferében az esetleges újrakérések miatt.

TP_DATACONF: Az eddig elkért adatok vissza lettek igazolva, törölhetők a bufferből, de az ebben a műveletben elkért adatok továbbra is a bufferben kell maradjanak.

TP_DATARETRY: Hibás átvitelből fakadóan – újrakérés miatt nem a soron következő bájtokat, hanem az üzenet elejétől számítva a `retry` paraméter `TxTpDataCnt` eleme által mutatott bájttól kezdődően kell a megadott számú adatbájtot átadni.

A függvény az `availableDataPtr` paraméterében adja vissza a bufferben a kért számú bájt átadása után elérhető, még továbbítandó adatbájtok számát. Ha a függvényen keresztül nullaszámú bájtot próbálunk elkérni, felhasználhatjuk a bufferben tárolt adatbájtok számának lekérdezésére.

A függvény visszatérési értéke `BufReq_ReturnType` típusú, melynek ebben az esetben három lehetséges értéke van:

BUFREQ_OK: A függvény sikeresen végrehajtódott. A kért adat bemásolódott a megadott helyre.

BUFREQ_E_BUSY: A kért mennyiségű adat jelenleg nem elérhető a bufferben. (A buffer foglalt.) A függvény későbbi újrahívása szükséges.

BUFREQ_E_NOT_OK: A kérés nem teljesült. (Egyéb hiba miatt.)

PduR_FrTpTxConfirmation

```
void PduR_FrTpTxConfirmation(
    PduIdType id,
    NotifResultType result
);
```

Egy I-PDU átviteli folyamatának sikeres vagy sikertelen befejeztéről informálja az átvitelt indító modult. A `result` paraméternek számos értéke lehet, mely az átvitel lezárásának okáról és körülményeiről tájékoztatja a felső modult.

Az FrTp az alatta lévő modul – FlexRay Interface – következő függvényeit hívhatja adatküldés esetén:

FrIf_Transmit

```
Std_ReturnType FrIf_Transmit(
    PduIdType FrIf_TxPduId,
    const PduInfoType* FrIf_PduInfoPtr
);
```

A már ismertetett függvény szolgál az összeállított PDU-k elküldésére. A függvény hívása után az FrIf vagy azonnali vagy függetlenített buffer hozzáféréssel az `FrTp_TriggerTransmit(...)` függvényen keresztül elveszi a PDU-t az FrTP-től.

FrIf_CancelTransmit

```
Std_ReturnType FrIf_CancelTransmit(
    PduIdType FrIf_TxPduId
);
```

Egy már elküldött, de még az `FrTp_TxConfirmation(...)` függvénnyel nem visszaigazolt PDU elküldésének visszavonására szolgál, az átvitel időtúllépés, fentről jövő abortálási kérés, vagy egyéb hibák miatt történő megszakítása esetén.

4.3. Konfiguráció

Ebben az esetben is a konfigurációs paraméterek definiálásával kezdtem az implementációt. A paraméterek egy része pre-compile time, egy másik része pedig lehet pre-compile és post-build time osztályba sorolható is. Az utóbbi csoportot a nagyobb rugalmasságot megengedő post-build time konfigurációs osztályba soroltam.

4.3.1. A pre-compile time konfigurációs adatok

A következő pre-compile time paraméterek preprocesszor direktívákként lettek megvalósítva. (A funkciókat engedélyező/tiltó paraméterek lehetséges értékei: `STD_ON` és `STD_OFF`)

FRTP_HAVE_ACKRT: A visszaigazolás, vagy újrakérés funkció engedélyezésére/tiltására szolgál.

FRTP_CHAN_NUM: A használható csatornák számát definiáló paraméter.

FRTP_CHANGE_PARAMETER_API: A modul API-ja által szolgáltatott, nem minden esetben szükséges `FrTp_ChangeParameter(...)` függvény engedélyezésére/tiltására szolgál.

FRTP_DEV_ERROR_DETECT: A fejlesztési hibák detektálását és jelentését engedélyező/tiltó paraméter.

FRTP_FULL_DUPLEX_ENABLE: A full-duplex módot (lsd. később) engedélyező/tiltó paraméter.

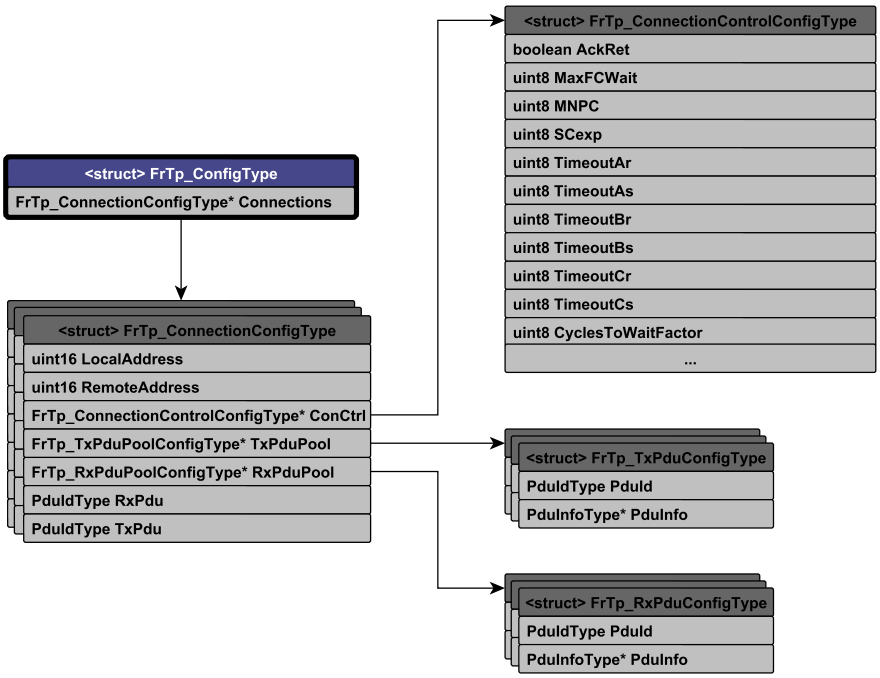
FRTP_HAVE_TC: A modul API-ja által szolgáltatott, nem minden esetben engedélyezett `FrTp_CancelTransmit(...)` függvény engedélyezésére/tiltására szolgál. A FlexRay Interface `FRIF_CANCELTRANSMIT_SUPPORT` konfigurációs paraméterével összehangolva állítandó.

FRTP_UNKNOWN_MSG_LENGTH: Az előre nem definiált hosszal történő adatátvitelt engedélyező/tiltó paraméter.

FRTP_VERSION_INFO_API: A modul API-ja által nyújtott, modul verziószámait szolgáltató `FrTp_GetVersionInfo(...)` függvény engedélyezésére/tiltására szolgál.

4.3.2. A post-build time konfigurációs adatok

A konfigurációs adatszerkezet felépítése az ábrán látható. *(4.8. ábra)* A következőkben bemutatom a konfiguráció által reprezentált objektumok feladatát.



4.8. ábra. *A post-build time konfigurációs adatszerkezet*

Connection: A Connection egy kommunikációs kapcsolatot definiáló adatszerkezet. Megvalósítása `FrTp_ConnectionConfigType` típusú struktúraként történt. A Connection tartalmazza a kommunikációs kapcsolatot azonosító `uint16` típusú `LocalAddress` (helyi cím) és `RemoteAddress` (távoli cím) mezőket, melyek az átvitel során az ISO szabvány szerinti PDU-k AI (Address Information) tartományának SA (Source Address) és TA (Target Address) mezőinek felelnek meg. Minden kapcsolathoz tartozik egy TxSdu és egy RxSdu, melyek a kapcsolaton keresztül küldésre vagy fogadásra kijelölt I-PDU-t azonosítják a PDU Router és az FrTp közötti átvitel során. Az I-PDU-k és a kapcsolatok közti hozzárendelés egy-egy értelmű.

ConnectionControl: Minden kapcsolathoz tartozik egy ConnectionControl adatszerkezet, mely `FrTp_ConnectionControlConfigType` típusú struktúraként lett megvalósítva. A ConnectionControl a kapcsolathoz tartozó egyéb paramétereket definiálja. Ez határozza meg a különböző időzítési paramétereket (As, Ar, Bs, Br, Cs, Cr), különböző hibák esetén a lehetséges újrapróbálkozások számát, a visszaigazolás szükségességét és a sávszélességre vonatkozó adatokat. Egy ConnectionControl több Connection-höz is tartozhat, hiszen két kapcsolatnak lehetnek azonos paraméterei.

TxPduPool: Minden kapcsolathoz tartozik egy megfelelő TxPduPool is. Ezek lényegében az adott kapcsolathoz tartozó kommunikációs folyamat során felhasználható N-PDU-kat tartalmazó tárolók. Egy keret elküldése előtt a modul innen választ egy szabad PDU-t, majd összeállítja a frame-et a hozzárendelt memóriaterületen, ezek után elküldi a PDU-t az FrIf-en keresztül. Egy TxPduPool több Connection-höz is hozzá lehet rendelve, de egy Connection-höz kizárólag egy TxPduPool tartozhat.

Ezzel a megoldással dinamikus sávszélesség szabályozás valósítható meg, hiszen a TxPduPool által tartalmazott N-PDU-k száma véges. Ha a TxPduPool által tartalmazott összes PDU el lett indítva egy vagy több Connection-ön keresztül, az FrTp nem küldhet újabbakat addig, amíg a már elindított PDU-k továbbítását az FrIf vissza nem igazolta.

A TxPduPool-okban az N-PDU-k a FlexRay buszon való továbbításuk sorrendjében vannak elhelyezve. A keretek küldése során az FrTp-nek mindig a legelső szabad PDU-t kell lefoglalnia, így elkerülhető a keretek sorrendjének felcserélődése a későbbiek folyamán.

RxPduPool: A TxPduPool-hoz hasonlóan ez is a modul által felhasználható N-PDU-kat tartalmazza a FlexRay buszon való küldés sorrendjében. Az RxPduPool által tartalmazott N-PDU-k adatfogadás esetén a Flow Contol üzenetek összeállítására és elküldésére vannak fenntartva.

4.4. A futásidejű adatszerkezet

A futásidejű adatszerkezet valósítja meg a csatornákat, és tárolja a PDU-k, valamint kapcsolatok állapotait jelző adatokat. *(4.9. ábra)* A következőkben bemutatom az adatszerkezet struktúráit.

FrTp_DataType: Ez az adatszerkezet gyökere. Minden futásidejű adat ezen keresztül érhető el. A modul állapotát is ez a struktúra tartalmazza.

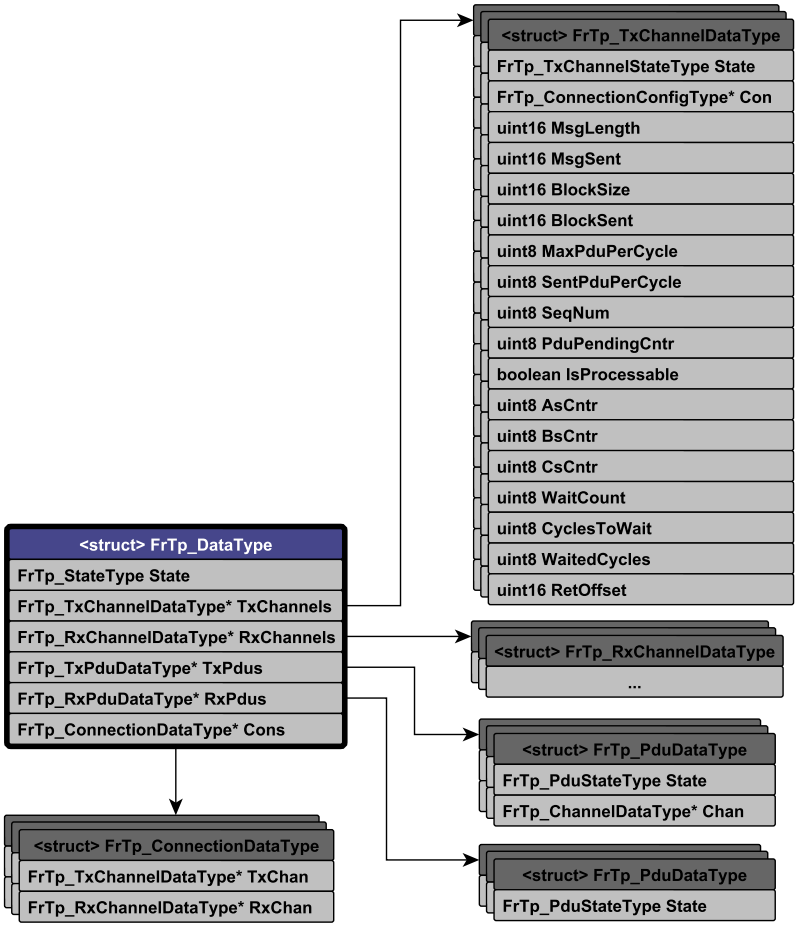
FrTp_StateType: `enum` típus, melynek lehetséges értékei: `FRTP_OFF` és `FRTP_ON`.

FrTp_TxChannelDataType: Ez a struktúra egy adatküldésre használható csatornát valósít meg. Tartalmazza a csatorna állapotát, a csatornához rendelt kapcsolatot, az elküldött bájtok számát, blokkok számát, PDU-k számát, ebben vannak megvalósítva az időzítéseket számon tartó számlálók, valamint a kapcsolat egyéb futásidejű paraméterei.

FrTpTxChannelStateType: `enum` típus, melynek lehetséges értékei: `IDLE`, `START`, `WAIT_CONF`, `WAIT_FC` és `CF_LF`.

FrTp_TxPduDataType: A Tx PDU-k adatait tartalmazza (foglalt vagy szabad, melyik csatorna foglalja).

FrTp_TxPduStateType: `enum` típus, melynek lehetséges értékei: `FREE`, `SENT` és `STUCKED`.



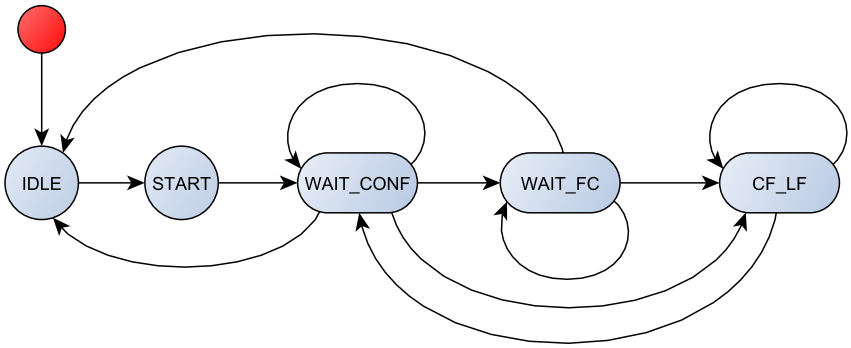
4.9. ábra. *A runtime adatszerkezet*

FrTp_ConnectionDataType: Egy kapcsolat állapotát tartja számon. Megadja, hogy a kapcsolatot melyik csatornák dolgozzák fel. (A `TxChan` vagy `RxChan` értéke lehet `NULL_PTR` is, mely azt jelzi, hogy az adott kapcsolaton épp nem zajlik adatátvitel.)

4.5. Adatküldés az FrTp modulon keresztül

A modul egyes csatornáinak működését adatküldés esetén az ábrákon látható folyamatábra *(4.13. ábra)* és állapotgép *(4.10. ábra)* írja le.

Inicializálás után minden csatorna állapotgépe `IDLE` (üresjárati) állapotban van. Az `IDLE` állapotban lévő csatornák lefoglalhatók adatküldési, vagy -fogadási célra. Az adatátvitel kezdetén a modul kiválasztja a megfelelő Connection-t és keres egy szabad csatornát, amit hozzárendel a kapcsolathoz. A későbbiek folyamán ciklikusan, ezt a csatornát dolgozza fel, és végrehajtja a soron következő műveleteket.



4.10. ábra. *A csatornák állapotgépei*

4.5.1. A Transmit függvény

Az adatküldés folyamata a modul Transmit függvényével kezdődik:

```
Std_ReturnType FrTp_Transmit(
    PduIdType FrTpTxSduId,
    const PduInfoType* FrTpTxSduInfoPtr
);
```

A függvény átadja az FrTp-nek a továbbítandó PDU ID-ját (`FrTpTxSduId`), valamint egy `PduInfoType` típusú struktúrára mutató pointert (`FrTpTxSduInfoPtr`). Ebben az esetben a struktúrának csak a PDU hosszát megadó eleme (`SduLength`) tartalmaz érvényes adatot.

A következő lépésben az FrTp a konfigurációjában megkeresi a kapott `FrTpTxSduId`-hoz tartozó, előre konfigurált Connection-t. Ha megtalálta a megfelelő kapcsolatot, keres egy szabad csatornát, átállítja az állapotát `START` állapotba, hozzáköti a Connection-t, és beállítja a csatorna `MsgLength` változójának értékét.

```
    FIND i: (FrTp_Config->Connections[i].TxSduId == FrTpTxSduId)
    FIND j: (Channels[j].Tx.State == IDLE)

    Channels[j].Tx.State = START;
    Channels[j].Tx.Con = &FrTp_Config -> Connections[i];
    Channels[j].Tx.MsgLength = FrTpTxSduInfoPtr -> SduLength;
```

Ha nem találta meg a megfelelő kapcsolatot, vagy nem volt szabad csatorna, a függvény – ha szükséges – jelenti a hibát a Development Error Tracer modulnak, majd `E_NOT_OK` értékkel tér vissza. Ha a futása közben nem ütközött hibába, `E_OK` értékkel tér vissza. A folyamat a `MainFunction`-ben folytatódik.

4.5.2. A MainFunction

A `MainFunction` minden lefutása során csak azokat a csatornákat dolgozza fel, melyek használatban vannak. A feldolgozandó csatornákat a csatornákhoz tartozó `IsProcessable` flag jelzi. A függvény lefutásának elején összegyűjti a feldolgozandó csatornákat:

```
FOR (i = 0; i < FRTP_CHAN_NUM; i++)
    IF (Channels[i].Tx.State != IDLE)
        Channels[i].Tx.IsProcessable = TRUE;
    ELSE
        Channels[i].Tx.IsProcessable = FALSE;
    END
END
```

Ezután a függvény egy ciklusban végigmegy az összes csatornán, és amennyiben egy csatorna feldolgozható, állapotától függően elvégzi a megfelelő műveleteket. Az iteráció addig zajlik, amíg van még ebben a `MainFunction` hívásban feldolgozandó csatorna:

```
WHILE (LÉTEZIK i: Channels[i].Tx.IsProcessable == TRUE)
    FOR (i = 0; i < FRTP_CHAN_NUM; i++)
        IF (Channels[i].Tx.IsProcessable == TRUE)
            SWITCH (Channels[i].Tx.State)
                ...
                ...
            END
        END
    END
END
```

START

Ha a csatorna `START` állapotban van, az azt jelzi, hogy meg kell kezdeni egy I-PDU átvitelét a Start Frame összeállításával, majd továbbításával. A modul a csatornához rendelt Connection TxPduPool-jából kikeresi az első szabad Tx PDU-t. (Ha nem talált szabad PDU-t, ebben a periódusban a `MainFunction` nem dolgozza fel a csatornát, az FrTp kiveszi a feldolgozandók listájából: `FrTp_Data.Channels[i].IsProcessable = FALSE;`)

Ezután a talált PDU méretét összeveti az `FrTp_Transmit(...)` függvényben kapott I-PDU hosszával, figyelembe véve a 4 bájtnyi AI és 4 bájtnyi PCI mezőket, majd dönt a szegmentálás szükségességéről.

```
IF (Channels[i].Tx.MsgLength > (PDU hossza - 8))
```

Ha az elküldendő üzenet nem fér el a talált PDU-ban, szegmentált átvitelre lesz szükség. Ha viszont az elküldendő üzenet mérete kisebb, mint a PDU mérete (figyelembe véve az AI és PCI mezőket), nem-szegmentált átvitellel – egyetlen Start Frame-ben – átvihető. Előre nem ismert hosszal történő adattovábbítás esetén (`FrTp_Data.Channels[i].Tx.MsgLength == 0`) mindenképpen szegmentált átvitelre kerül sor. Az FrTp a PDU Router `PduR_FrTpCopyTxData(...)` függvényén keresztül elkéri – az előző döntéstől függően – vagy a PDU-ban aktuálisan továbbítható bájtokat, vagy a teljes üzenetet.

Miután az adatok bekerültek a PDU SDU területére, az FrTp összeállítja a Connection-höz konfigurált LocalAddress és RemoteAddress paramétereknek megfelelően a PDU első négy bájtján az AI mezőt. Ezután a PCI mező összeállítása következik. A Connection-höz konfigurált ConnectionControl `AckRet` paraméterének függvényében (STFA, vagy STFU keret összeállítása szükséges) beállítható a PCI első bájtja. (STFU esetén 0x40, míg STFA keret esetén 0x41.) A PCI második bájtja az SDU adatbájtjainak számát (FPL), a 3-4. bájtok pedig a teljes üzenet hosszát (ML) tartalmazzák. (Nem-szegmentált átvitel esetén ezek megegyeznek. Ismeretlen hosszal történő adatküldés esetén pedig az ML mező értéke 0.)

Miután a teljes keret kész, az FrTp továbbítja azt a FlexRay Interface modulon keresztül (`FrIf_Transmit(...)`). A sikeres küldés után a PDU állapotát `SENT`-re állítja, és hozzárendeli a PDU-hoz a csatornát.

Mivel ebben a MainFunction ciklusban több műveletet ezen a csatornán nem kell végrehajtani, a csatornát kiveszi a feldolgozandók listájából, és az állapotgépet `WAIT_CONF` állapotba lépteti. Ezután növeli a `PduPendingCounter`-t (mely az elküldött, de még nem visszaigazolt PDU-kat számolja), majd nullázza az `AsCntr` számlálót, mely a `MainFunction` periódusidejében méri a protokoll As időzítését.

```
TxPdus[PduId].State = SENT;
TxPdus[PduId].Channel = &Channels[i];
Channels[i].Tx.PduPendingCntr++;
Channels[i].Tx.IsProcessable = FALSE;
Channels[i].Tx.State = WAIT_CONF;
Channels[i].Tx.AsCntr = 0;
```

WAIT_CONF

A csatorna ebben az állapotban marad, amíg az összes előzőleg elküldött Tx PDU továbbítását az Interface vissza nem igazolja. Ebben az állapotban a modul eggyel növeli a csatornához tartozó `AsCntr` számlálóját, ezzel mérve a PDU-k elküldése óta eltelt időt, majd kiveszi a csatornát a feldolgozandók sorából. Ebből az állapotból a csatorna állapotgépe csak az `FrTp_TxConfirmation(...)` függvény hatására léphet ki, amikor minden PDU-jának továbbítása vissza lett igazolva.

```
Channels[i].Tx.AsCntr++;
Channels[i].Tx.IsProcessable = FALSE;
```

Ha az `AsCntr` értéke elérte a csatornához kötött Connection-höz rendelt ConnectionControl konfigurációjában megadott maximumot, akkor időtúllépés történt. Az elküldött PDU-k továbbításáról nem érkezett meg időben a visszaigazolás az Interface-től. Az előírásoknak megfelelően ebben az esetben meg kell szakítani az átvitelt. Az összes, a csatorna által lefoglalt PDU felszabadítására meg kell hívni az `FrIf_CancelTransmit(...)` függvényt, majd a PDU Router `PduR_FrTpTxConfirmation(...)` függvényén keresztül értesíteni kell az átvitelt indító modult a hibáról.

WAIT_FC

A csatorna ebben az állapotban marad, amíg nem érkezik meg a megfelelő Flow Control keret. A csatornához tartozó `BsCntr` minden `MainFunction` periódusban eggyel nő, ezzel mérve a következő Flow Control üzenetig eltelt időt, és felügyelve a csatorna Bs időtúllépését. A számláló növelése után a csatorna kikerül a feldolgozandók listájából. Ebből az állapotból a csatorna csak a megfelelő Flow Control keret megérkezését követően, az `FrTp_RxIndication(...)` függvény hatására léphet ki.

```
Channels[i].Tx.BsCntr++;
Channels[i].Tx.IsProcessable = FALSE;
```

Ha a `BsCntr` számláló értéke elérte a csatornához rendelt Connection-höz tartozó ConnectionControl konfigurációjában megadott maximumot, akkor időtúllépés történt. A várt Flow Control keret nem érkezett meg a megadott időn belül, így ebben az esetben is az `AsCntr` időtúllépésénél ismertetett módon meg kell szakítani az átvitelt.

CF_LF

Ebben az állapotban valósul meg a CF és LF keretek küldése. Az FrTp úgynevezett löketekben (burst-ökben) küldi el a kereteket. Ez úgy valósul meg, hogy egy `MainFunction` hívás során addig folytatja egy csatornán a keretek küldését, amíg:

- a csatornához rendelt Connection-höz tartozó TxPduPool tartalmaz szabad PDU-t,
  `LÉTEZIK j: (TxPdus[j].State == FREE && TxPdus[j] ∈ Channels[i].Tx.Connection.TxPduPool)`
- a legutóbbi Flow Control keret BC (sávszélesség vezérlés) mezője által megszabott egy ciklusban küldhető keretek számát nem értük el,
  `IF (Channels[i].SentPduPerCycle < Channels[i].MaxPduPerCycle)`
- az ebben az átviteli blokkban elküldhető bájtok számát az adott csatornán még nem értük el,
  `IF (Channels[i].BlockSent < Channels[i].BlockSize)`

- az üzenet összes adatbájtját át nem küldtük,
  `IF (Channels[i].MsgSent < Channels[i].MsgLength)`
- nincs kivárandó ciklus a következő keretek elküldése előtt.
  `IF (Channels[i].CyclesToWait == Channels[i].WaitedCycles)`

Ha időközben bármelyik feltétel sérül, a csatorna kikerül a feldolgozandók listájából, és az állapotgép a `WAIT_CONF` állapotba lép, ahonnan csak az összes eddig elküldött PDU visszaigazolása után (`FrTp_TxConfirmation(...)`) léphet tovább.

```
Channels[i].IsProcessable = FALSE;
Channels[i].State = WAIT_CONF;
```

Annak érdekében, hogy a közös TxPduPool-on osztozó csatornák egyikénél se alakulhasson ki „éhezés”, a csatornák kiszolgálását round-robin ütemezéssel oldottam meg. A `MainFunction` minden lefutása során addig folytatja a csatornák feldolgozását, amíg van feldolgozható csatorna. Ha egy csatorna `CF_LF` állapotban van, az adott iterációs ciklusban csak egy keret elküldését végzi el a függvény, azután a következő csatornát szolgálja ki. Amikor ismét az előző csatornához ér – és a csatorna továbbra is feldolgozható – ismét lefoglal számára egy PDU-t, majd elküld egy újabb keretet.

Fontos, hogy amikor az FrTp egy TxPduPool-ban szabad PDU-t keres, mindig az első megfelelő PDU-t kell lefoglalnia. Mivel egy TxPduPool-on belül az N-PDU-k a FlexRay buszon való továbbításuk sorrendjében vannak elhelyezve, ezzel a megoldással biztosítható, hogy a CF keretek sorrendje az Interface-ben ne cserélődjön fel. *(4.11. ábra)*

Egy csatorna kiszolgálása (a `MainFunction` egy iterációs ciklusa) – a fentiek alapján – a következőképpen néz ki:
`IF (Channels[i].IsProcessable == TRUE)`)

1. A modul megvizsgálja, hogy van-e hátra kivárandó ciklus a következő keretek elküldése előtt. Ha igen, csökkenti a kivárt ciklusokat számláló `WaitedCycles` változót, és kiveszi a csatornát a feldolgozandók listájából. Ezután a következő csatorna kiszolgálására lép.

   ```
   IF (Channels[i].WaitedCycles < Channels[i].CyclesToWait)
       Channels[i].WaitedCycles++;
       Channels[i].IsProcessable = FALSE;
   END
   ```

2. Ha ebben a ciklusban küldhetők keretek, az FrTp megkeresi az első szabad PDU-t a csatornához tartozó TxPduPool-ban.
   `KERES j: (TxPdus[j].State == FREE &&`
   `TxPdus[j] ∈ Channels[i].Tx.Connection.TxPduPool)`
   (a) Ha nem talált szabad PDU-t, kiveszi a csatornát a feldolgozandók listájából, növeli a keretek elküldése közötti időt számláló `CsCntr`-t, és a következő csatorna kiszolgálásával folytatja a végrehajtást.

```
Channels[i].Tx.IsProcessable = FALSE;
Channels[i].Tx.CsCntr++;
```

Ha a `CsCntr` számláló elérte a csatornához kötött Connection-höz tartozó ConnectionControl konfigurációjában megadott maximumot, nem küldtük el időben a következő keretet, az átvitelt meg kell szakítani.

```
Channels[i].Tx.IsProcessable = FALSE;
```

(b) Ha talált szabad csatornát (az első szabad csatornát), lefoglalja azt a következő keret elküldésére.

3. Ezután kiszámolja a következő keretben elküldendő bájtok számát a következő képlet alapján:

```
MIN: (MsgLength - MsgSent, BlockSize - BlockSent, PduLength - 8)
```

A keretben elküldhető bájtok száma a fenti mennyiségek (üzenetből hátralevő bájtok száma, az adott blokkban elküldhető bájtok száma, a talált PDU hossza, figyelembe véve az AI és PCI mezőket) minimuma.

4. Ha kiszámolta a küldendő bájtok számát, a megfelelő mennyiségű adatot elkéri a felső modultól a PDU bufferébe, és a bájtok számát hozzáadja a csatorna `MsgSent` és `BlockSent` változójához, majd az adott ciklusban elküldött PDU-k számát eggyel növeljük:

```
PduR_FrTpCopyTxData(...);
Channels[i].MsgSent += ByteCount;
Channels[i].BlockSent += ByteCount;
Channels[i].SentPduPerCycle++;
```

5. Az adatok elkérése után, beállítja a PDU AI mezőjét, a csatornához rendelt Connection `LocalAddress` és `RemoteAddress` paramétere alapján.

6. Ezután összeállítja a megfelelő PCI mezőt, és elküldi a keretet a FlexRay Interface-en keresztül:

LF: Ha a csatorna `MsgSent` változója megegyezik a `MsgLength` változóval, az azt jelenti, hogy ebben a keretben küldjük el az üzenet utolsó bájtjait, tehát ez egy Last Frame (LF) keret. Ebben az esetben ennek megfelelő PCI mezőt *(4.1. táblázat)* kell összeállítani. Az FPL az aktuálisan küldött bájtok számát, az ML pedig a teljes üzenet hosszát tartalmazza. Ezek után megtörténik a keret továbbítása, a PDU állapotának foglaltra állítása és a csatorna `PduPendingCntr`-ének növelése.

Mivel ez egy LF keret volt, biztosan nem kell több keretet küldenünk. Az állapotgép `WAIT_CONF` állapotba vezérelhető (a visszaigazolásig eltelt időt számoló `AsCntr`-t törölni kell), a csatorna pedig kivehető a feldolgozandók listájából:

```
FrIf_Transmit(...);
TxPdus[PduId].State = SENT;
```

```
Channels[i].Tx.PduPendingCntr++;
Channels[i].Tx.State = WAIT_CONF;
Channels[i].Tx.IsProcessable = FALSE;
Channels[i].Tx.AsCntr = 0;
```

CF_EOB: Ha a `BlockSent` változó értéke egyezik meg a `BlockSize` változó értékével akkor ez az adott kommunikációs blokk utolsó kerete, tehát egy CF_EOB keret. Ennek megfelelően kell a PCI mezőt összeállítani. *(4.1. táblázat)* Az SN mező értéke a csatorna `SeqNum` változójának értékét kapja, az FPL mező pedig az aktuálisan elküldött bájtok számát. Ezután megtörténhet a frame elküldése.

Mivel ez a keret egy blokk utolsó kerete, mindenképpen meg kell várnunk az összes elküldött PDU visszaigazolását, majd a következő Flow Control üzenetet az átvitel folytatásához. Az állapotgép az `WAIT_CONF` állapotba vezérelhető, és a csatorna kivehető a feldolgozandók listájából.

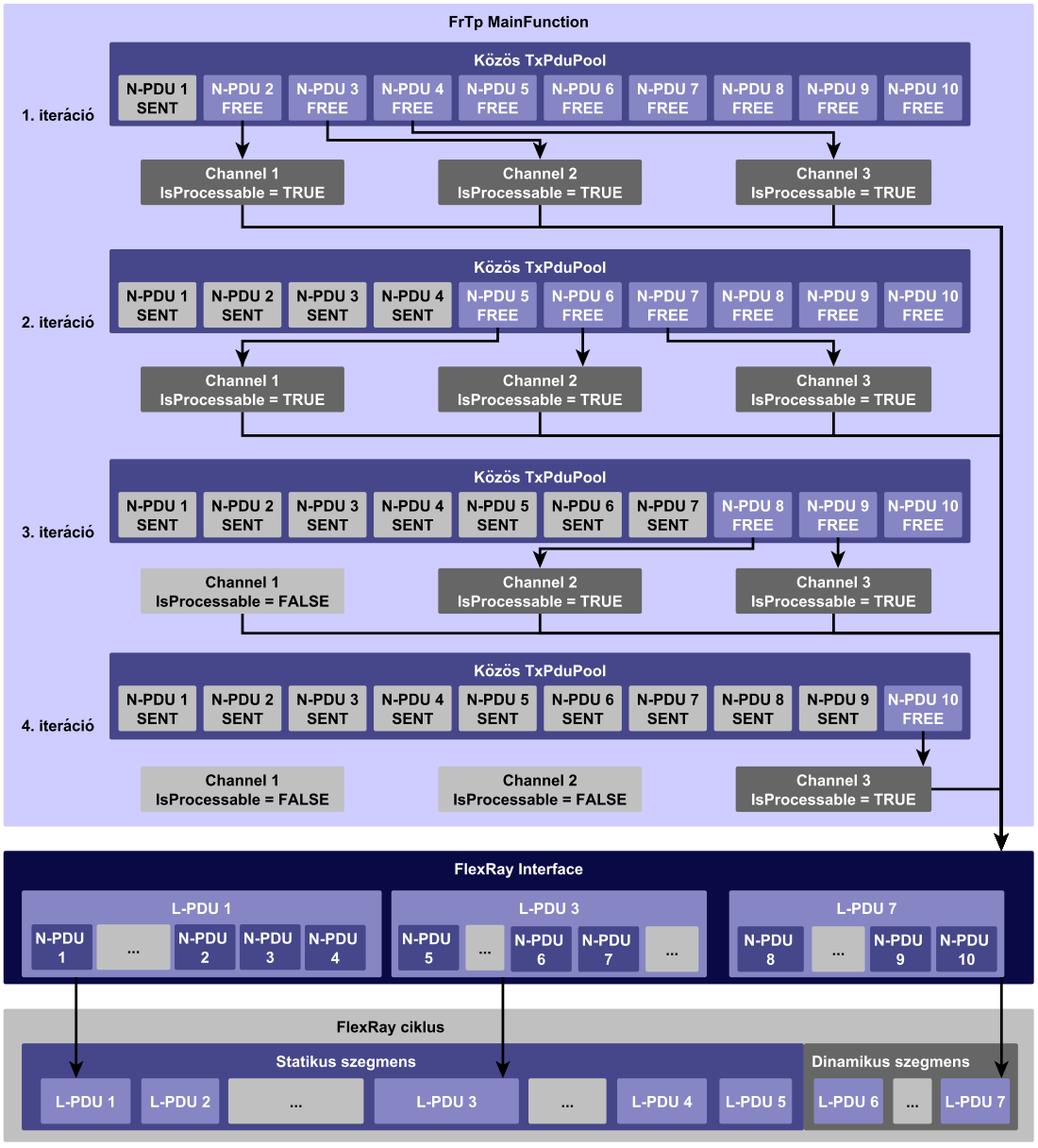
```
FrIf_Transmit(...);
TxPdus[PduId].State = SENT;
Channels[i].Tx.PduPendingCntr++;
Channels[i].Tx.State = WAIT_CONF;
Channels[i].Tx.IsProcessable = FALSE;
Channels[i].Tx.AsCntr = 0;
Channels[i].Tx.SeqNum = 0;
Channels[i].Tx.SentPduPerCycle = 0;
```

CF_1: Amennyiben a csatornához tartozó `BlockSent` változó értéke kisebb, mint a `BlockSize` változó értéke, az elküldendő keret egy CF_1 keret. Ennek megfelelően össze kell állítani a PCI mezőt. *(4.1. táblázat)* Az SN mező értéke a csatorna `SeqNum` változójának aktuális értékét, az FPL mező értéke pedig az ebben a PDU-ban küldött adatbájtok számát tartalmazza.

A PCI mező összeállítása után a keret elküldhető a FlexRay Interface-en keresztül. Az FrTp lenullázza a `CsCntr` számlálót, majd dönt a csatorna feldolgozásának folytatásáról. Amennyiben a csatorna `SentPduPerCycle` változója még nem érte el a legutóbbi Flow Control üzenet BC mezőjében megadott `MaxPduPerCycle` értéket, a csatornán ebben a `MainFunction` periódusban még küldhetünk kereteket. Ha elküldtük az egy ciklusban küldhető összes keretet, a csatorna `WAIT_CONF` állapotba kerül, és ebben a `MainFunction` periódusban már nem dolgozható fel többet. Ebben az esetben az elküldött PDU-k visszaigazolásáig eltelt időt mérő `AsCntr` számlálót, valamint az adott ciklusban eküldött keretek számát nyílvántartó `SentPduPerCycle` változót törölni kell.

```
FrIf_Transmit(...);
TxPdus[PduId].State = SENT;
Channels[i].Tx.PduPendingCntr++;
Channels[i].Tx.SeqNum++;
```

```
Channels[i].Tx.CsCntr = 0;
IF (Channels[i].Tx.SentPduPerCycle == Channels[i].Tx.MaxPduPerCycle)
    Channels[i].Tx.State = WAIT_CONF;
    Channels[i].Tx.IsProcessable = FALSE;
    Channels[i].Tx.AsCntr = 0;
    Channels[i].Tx.SentPduPerCycle = 0;
END
```



4.11. ábra. *A csatornák kiszolgálása*

4.5.3. Az FrTp_TxConfirmation függvény

Az elküldött PDU-k visszaigazolását a FlexRay Interface az `FrTp_TxConfirmation(...)` függvény meghívásával végzi el. A függvény a paraméterében kapott PDU ID alapján felszabadítja a Tx PDU-t, majd csökkenti a hozzá rendelt csatorna `PduPendingCntr`-ét.

```
TxPdus[PduId].State = FREE;
TxPdus[PduId].Channel -> PduPendingCntr--;
```

Ha a `PduPendingCntr` értéke elérte a nullát, a csatornához tartozó összes eddig elküldött PDU vissza lett igazolva, a csatorna állapotgépe kiléphet a `WAIT_CONF` állapotból. A következő állapot a Channel változóitól függ.

Amennyiben a csatorna `MsgSent` paramétere megegyezik az `MsgLength` paraméterrel, a teljes üzenet átvitele megtörtént. Ebben az esetben a csatornához tartozó Connection-höz rendelt ConnectionControl `AckRet` paramétere dönt a következő állapotról. Ha szükséges a visszaigazolás, akkor a csatorna következő állapota a `WAIT_FC` állapot, ahol az FrTp vár a visszaigazolás megérkezésére. A `WAIT_FC` állapotba lépéskor a csatorna `BsCntr` változója ki lesz nullázva. Ha nincs szükség visszaigazolásra, a csatornát fel lehet szabadítani (minden változó alaphelyzetbe áll), és az állapotgép az `IDLE` állapotba kerül.

Abban az esetben, ha az `MsgSent` és az `MsgLength` paraméter nem egyezik meg, az átvitel nem fejeződött be. A `BlockSize` és a `BlockSent` változók döntenek az állapotgép következő állapotáról. Ha a két paraméter megegyezik, az azt jelenti, hogy az adott blokkban elküldhető teljes adatmennyiséget elküldtük, tehát a folytatás előtt meg kell várnunk a következő Flow Control üzenetet. Az állapotgép a `WAIT_FC` állapotba lép és a `BsCntr` számláló törlésre kerül. Ha a két paraméter nem egyezik meg, az aktuális blokkban még küldhetünk kereteket, tehát az állapotgép a `CF_LF` állapotba lép. Ebben az esetben a protokoll Cs időzítéseit felügyelő `CsCntr` számlálót kell törölni.

Mivel a `PduPendingCntr` nulla értéke mindenképpen egy ciklus összes PDU-jának elküldését jelzi, a csatorna `SentPduPerCycle` és `WaitedCycles` változója nullázódik.

4.5.4. Az FrTp_RxIndication függvény

Az `FrTp_RxIndication` függvény szolgál az FrTp értesítésére FrIf által fogadott N-PDU-król.

```
void FrTp_RxIndication(
    PduIdType RxPduId,
    PduInfoType* PduInfoPtr
)
```

A függvény a kapott PDU PCI mezője alapján meghatározza a PDU típusát. *(4.1. táblázat)* Ha ez egy Flow Control PDU, az FrTp az AI mező által tartalmazott Source Address és Target Address alapján a konfigurációs struktúrájában megkeresi megfelelő Connection-t (`LocalAddress` és `RemoteAddress` mezők). A csatornák közül kiválasztja azt, amelyikhez az adott Connection hozzá van kötve. Amennyiben az adott Connection jelenleg egyik csatornához sem tartozik, az FrTp figyelmen kívül hagyja a kapott keretet.

Ha megvan a megfelelő csatorna, a modul a PCI mező alapján megkezdi a PDU feldolgozását:

CTS: Ha a kapott PDU egy FC_CTS keretet tartalmaz, és az adott csatorna `WAIT_FC` állapotban van, az FrTp beállítja a csatorna `BlockSize` változóját a PDU BfS mezőjének megfelelően, majd nullázza a `BlockSent` változó értékét. A keret BC (sávszélességre vonatkozó) paramétere alapján beállítja a csatorna `MaxPduPerCycle` és `CyclesToWait` paramétereit. A `CyclesToWait` változó értékét a Connection-höz tartozó `CyclesToWaitFactor` és a kiszámolt FlexRay ciklusok számának szorzata adja. Ezzel a módszerrel számítható át a FlexRay ciklusok száma `MainFunction` ciklusok számára. Ezután a csatorna állapotgépét `CF_LF` állapotba állítja, és törli a következő keret elküldéséig eltelt időt felügyelő `CsCntr` számlálót.

```
IF (State == WAIT_FC)
    BlockSize = BfS;
    BlockSent = 0;
    MaxPduPerCycle = MNPC;
    CyclesToWait = 2 << SCexp - 1;
    State = CF_LF;
    CsCntr = 0;
END
```

Ha a csatorna állapota nem `WAIT_FC`, az FrTp figyelmen kívül hagyj a keretet.

ACK: Ha a kapott PDU egy átvitelt visszaigazoló `FC_ACK_RET` keretet tartalmaz, és a csatorna `WAIT_FC` állapotban van, valamint az üzenet összes bájtja el lett küldve, akkor ez a PDU az átvitel visszaigazolása. A csatorna állapotgépe `IDLE` állapotba vezérelhető, és felszabadítható. Az átvitel sikerességéről az FrTp értesíti az átvitelt indító felső modult.

```
IF (State == WAIT_FC && MsgLength == MsgSent)
   State = IDLE;
   ...
   PduR_FrTpTxConfirmation(...);
END
```

RET: Ha az adott Connection `AckRet` konfigurációs paramétere nem engedélyezi a újraküldés mechanizmusát, az FrTp figyelmen kívül hagyja a PDU-t. Ha a me-

chanizmus engedélyezett, a modul beállítja a csatorna `RetOffset` mezőjét a BP mező által megadott értékre, ezzel jelezve, hogy az adatok újraküldése szükséges. Az újraküldési kérés feldolgozása az adott blokk befejezése után, a következő blokkban történik. A blokkból hátralevő kereteket a fogadó fél figyelmen kívül hagyja.

```
RetOffset = BP;
```

WT: Ha a Connection-höz tartozó csatorna állapota nem `WAIT_FC` az FrTp figyelmen kívül hagyja keretet. Ha a csatorna állapotgépe `WAIT_FC` állapotban van, ez az üzenet a keretek küldésének átmeneti felfüggesztésére kéri a modult. A Flow Control keretek érkezéséig eltelt időt mérő `BsCntr` számláló törlésre kerül. Az FC_WT keretek számát nyílvántartó `WaitCount` számláló pedig eggyel növelve lesz. Ha túllépte a Connection konigurációjában megengedett maximális értéket, az átvitelt meg kell szakítani.

ABT: Az FC_ABT keret az átvitel azonnali megszakítására kéri az FrTp-t. Ebben az esetben a csatornát fel kell szabadítani, a hozzá tartozó állapotgépet `IDLE` állapotba kell vezérelni, majd értesítani kell a felső modult az átvitel megszakításáról.

```
State = IDLE;
PduR_FrTpTxConfirmation(...);
```

OVFLW: Ez a keret kizárólag Start Frame után érkezhet, ha a fogadó egység jelenleg nem képes a Start Frame-ben küldött adatok feldolgozására és az átvitel megkezdésére. Ha nem a Start Frame után érkezik, az FrTp figyelmen kívül hagyja.

4.5.5. A PDU-k átvitele függetlenített buffer hozzáféréssel

Ha egy PDU átvitele függetlenített buffer hozzáféréssel (Decoupled Buffer Access) van konfigurálva, az `FrIf_Transmit(...)` függvény hatására az Interface nem veszi át egyből a PDU-t. A későbbiek folyamán az `FrTp_TriggerTransmit(...)` függvény meghívásával kéri az FrTp-t a PDU megfelelő bufferbe másolására.

```
Std_ReturnType FrTp_TriggerTransmit(
    PduIdType TxPduId,
    PduInfoType* PduInfoPtr
)
```

A kért PDU ID-ját a függvény `TxPduId`, az FrIf által szolgáltatott buffer paramétereit pedig a `PduInfoPtr` paramétere tartalmazza. Az FrTp feladata a `PduInfoPtr` által hivatkozott `PduInfoType` típusú struktúra `SduDataPtr` mezőjének megfelelő memóriacímre másolni a PDU adatbájtjait, majd beállítani az `SduDataLength` mezőjét a PDU hosszának megfelelően.

4.5.6. Adatküldés ismeretlen hosszal

Előre nem ismert hosszúságú üzenet átvitelét a felső modul `FrTp_Transmit(...)` függvény számára az üzenet hosszát tartalmazó paraméter nulla érékével jelzi. A Connection számára lefoglalt csatorna `MsgLength` változója is ennek megfelelően a nulla értéket veszi fel. A továbbiakban ez jelzi a modul számára, hogy az aktuális csatornán ismeretlen hosszúságú üzenet átvitele zajlik.

Az FrTp ezt figyelembe véve, az előzőkben leírtaktól kis mértékben eltérően végzi a csatorna kiszolgálását. Minden keret összeállítása előtt meghívja a `PduR_FrTpCopyTxData(...)` függvényt, az elkérendő adatbájtok számát jelző paraméternek nulla értéket adva. A függvény ennek hatására a paraméterlistján visszaadja a bufferben jelenleg elérhető adatbájtok számát. Az FrTp ennek az adatnak a segítségével a számíthatja az ebben a keretben elküldhető bájtok számát. A minimum meghatározásánál az üzenetből hátralevő bájtok száma helyett a bufferben aktuálisan elérhető bájtok száma szerepel.

```
MIN: (ActualBuffer, BlockSize - BlockSent, PduLength - 8)
```

A keretek elküldése ettől eltekintve azonos módon történik az ismert hosszúságú üzenetek kereteinek elküldésével. Az utolsó elküldendő kerethez (melyet LF keretként kell elküldeni) akkor értünk, mikor az adatbájtok elkérése során a `PduR_FrTpCopyTxData(...)` függvény azt jelzi, hogy a buffer kiürült. (A paraméterlistáján visszaadott, hátralevő bájtok számát jelző érték nulla.) Ilyenkor az LF keret ML mezője a csatorna `MsgSent` változója által számolt, eddig elküldött összes bájt számát tartalmazza.

4.5.7. A bufferkezelés és az újraküldés mechanizmusa

Az újraküldés szükségességét a csatorna `RetOffset` változójának nullától különböző értéke jelzi. Ezt a változót a `FrTp_RxIndication(...)` függvény állítja be, ha időközben újraküldést kérő Flow Control keret érkezett. Ha valamelyik átviteli blokk elején (tehát `SeqNum = 0`) az FrTp újraküldés szükségességét észleli, a `PduR_FrTpCopyTxData(...)` függvény `TP_DATARETRY` paraméterrel való meghívásával megkezdi a `RetOffset` változó által mutatott pozíciótól a már elküldött adatbájtok újrakérését, és az előző átviteli blokk adott pozíciótól való megismétlését. Ebben az esetben a blokk befejeztéig a keretek CF_2 keretként lesznek elküldve.

Egy adott átviteli blokkban elkért bájtok a felső modul bufferéből az FC_CTS keret megérkezése után üríthetőek ki. Mivel egy FC_CTS üzenet megérkezése egyben egy új blokk kezdetét is jelzi, ezért egy új blokk első PDU-jának összeállításakor a `PduR_FrTpCopyTxData(...)` függvény meghívása `TP_DATACONF` paraméterrel történik, melynek hatására az eddig elkért adatokat a bufferből ki kell üríteni, azonban az ebben a függvényhívásban elkért bájtok továbbra is a bufferben maradnak. A blokk további PDU-inak elküldésekor a `PduR_FrTpCopyTxData(...)` függvény

TP_CONFPENDING paraméterrel lesz hívva, melynek hatására az átviteli blokkban elkért adatok a bufferben maradnak a következő FC_CTS keretig – tehát egy új blokk megkezdéséig.

Az új blokk elején megint megtörténhet az eddig elküldött adatok ürítése, vagy – újraküldés esetén – ismételt elkérése.

4.5.8. Időzítések

A protokoll As, Bs és Cs időzítéseinek betartását a csatornákhoz tartozó AsCntr, BsCntr valamint a CsCntr felügyeli. A számlálók a MainFunction periódusidejében mérik a különböző események között eltelt időt. Ha bármelyik számláló eléri a csatornához rendelt Connection-höz tartozó ConnectionControl-ban konfigurált értéket, az adott csatornán zajló átvitel megszakad. Az összes, a csatornával kapcsolatban elküldött Tx PDU átvitele felfüggesztésre kerül az FrIf_CancelTransmit(...) fügvény meghívásával, majd az FrTp értesíti az adattovábbítást indító felső modult az átvitel megszakításáról a PduR_FrTpTxConfirmation(...) függvényen keresztül. Ezek után a csatorna felszabadítása és IDLE helyzetbe állítása következik. Végül a csatorna kivehető az adott MainFunction ciklusban feldolgozandó csatornák sorából.

A számlálók léptetését és törlését kiváltó események:

AsCntr

Léptetés: A MainFunction minden lefutásakor egyszer, ha az állapotgép WAIT_CONF állapotban van.

Törlés: Minden esetben, amikor az állapotgép WAIT_CONF állapotba lép.

BsCntr

Léptetés: A MainFunction minden lefutásakor egyszer, ha az állapotgép WAIT_FC állapotban van.

Törlés: Minden esetben, amikor az állapotgép WAIT_FC állapotba lép.

CsCntr

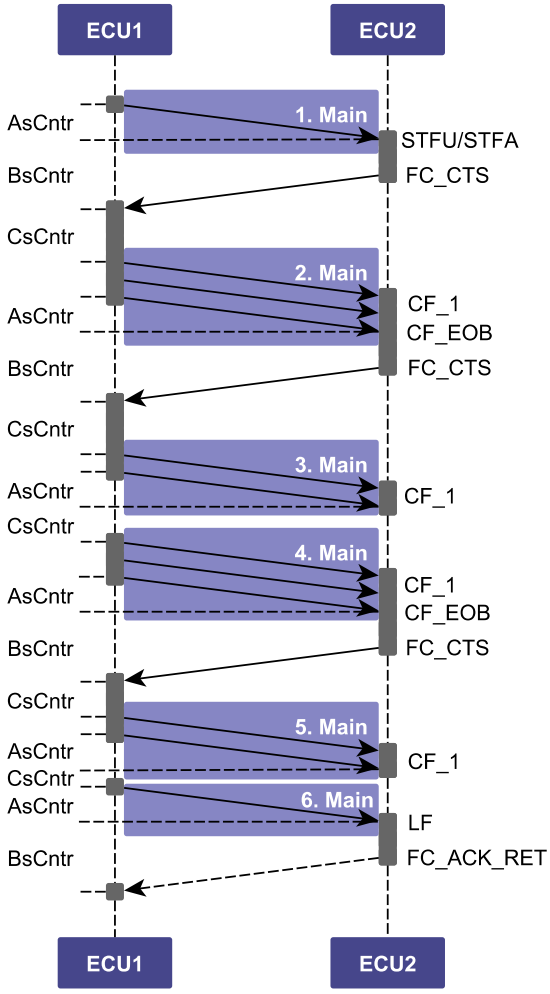
Léptetés: A MainFunction minden olyan lefutása esetén, ha az adott csatorna CF_LF állapotban van, de a modul nem tudott keretet küldeni.

Törlés: Minden esetben, amikor az állapotgép CF_LF állapotba lép.

4.6. Erőforrásigény

4.6.1. Memóriaigény

Az FrTp modul memóriaigényét lényegében a TxPduPool-ok és az RxPduPool-ok által tartalmazott PDU-k száma és mérete határozza meg. A konfigurációs és futásidejű adatstruktúrák méretei az egyéb paramétereknek (csatornák és PDU-k száma) szintén lineáris függvénye.



4.12. ábra. *A szoftver időzítéseket felügyelő számlálói*

4.6.2. Futásidő

A modul konfigurációja nem biztosított lehetőséget a FlexRay Interface esetében ismertetett jelentős futásidőbeli egyszerűsítésekre. A `MainFunction` futásának időtartama minden egyes lefutás alkalmával jelentősen eltérhet. Mivel ebben a függvényben zajlik az adatok feldolgozása és a csatornák kiszolgálása, ez a következő tényezőktől függ:

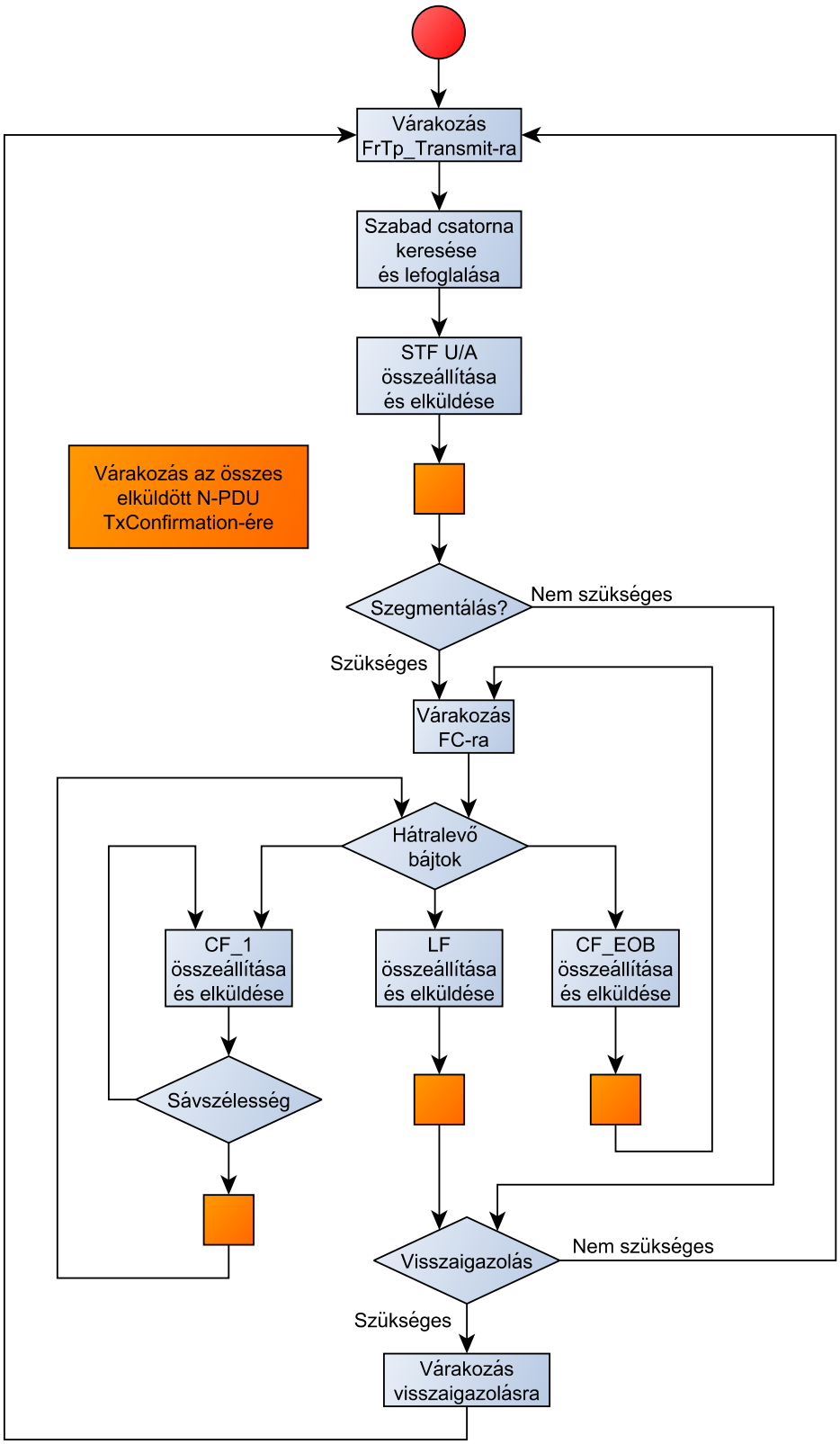
- az aktuális ciklus során feldolgozandó csatornák száma,
- a felhasználható N-PDU-k mérete,
- az adott ciklusban küldött vagy fogadott adatbájtok száma,
- az adott átviteli blokkban összesen küldhető, vagy fogadható adatbájtok száma,
- az egy ciklusban küldhető PDU-k száma

A többi API függvény esetében – mivel azok általában csak egyszerűbb műveleteket végeznek el, sőt, a legtöbb esetben csupán flag-eket és állapotváltozókat állítanak be

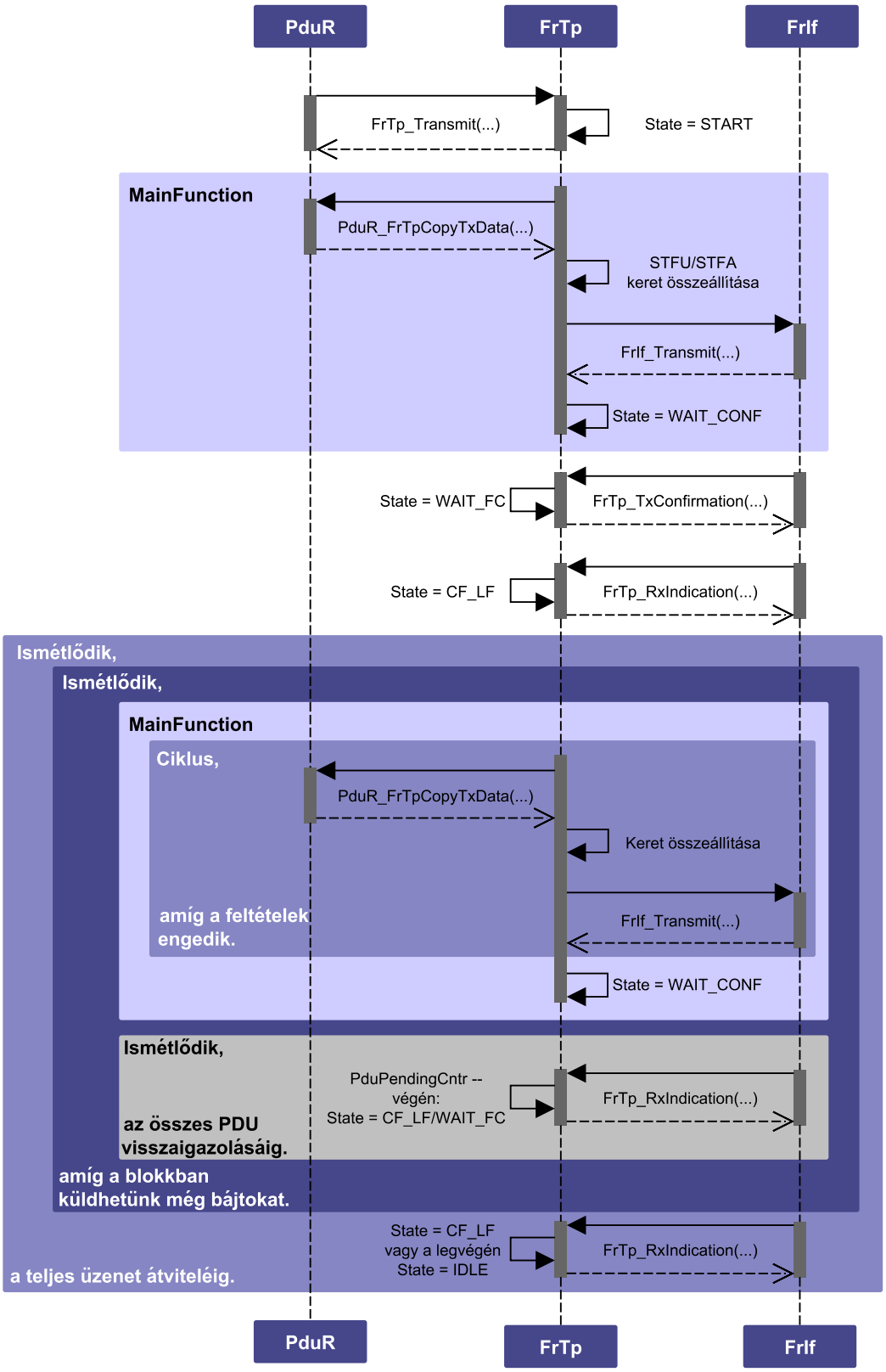
– a futásidő elhanyagolható.

A megfelelően felépített konfigurációs és futásidejű adatstruktúráknak köszönhetően nincs szükség bonyolult keresési algoritmusokra. A legtöbb esetben az adatok egyszerű indexeléssel elérhetőek.

Az egyetlen jelentősebb keresési feladat a PduPool-ok szabad PDU-inak keresése. Ilyenkor a `MainFunction` sorban végignézi a PduPool-ok által tartalmazott PDU-kat, majd az első szabad PDU-nál megáll. Ez a művelet a legrosszabb esetben annyi iterációt igényel, ahány PDU van az adott PduPool-ban.



4.13. ábra. *Az adatküldés folyamatábrája*



4.14. ábra. *Az adatküldés teljes folyamata*

5. fejezet

Tesztelés

Ahhoz, hogy meggyőződhessünk egy szoftver elvártaknak megfelelő működéséről, elengedhetetlen annak tesztelése. Ebben a fejezetben először a teszteléssel kapcsolatos néhány elméleti fogalmat tisztázok, majd ismertetem a tesztelés folyamatát CUnit környezetben, melyre egy példát is bemutatok.

5.1. Teszteléssel kapcsolatos alapfogalmak, mérőszámok

Az alapfogalmak definícióit a Magyar Szoftvertesztelői Tanács Egyesület Szoftvertesztelés egységesített kifejezéseinek gyűjteménye című dokumentuma [20] alapján írtam.

Tesztelés: Tesztelésnek nevezzük az összes szoftverfejlesztési életciklushoz kapcsolódó folyamatot, mely kapcsolatban áll a szoftvertermékek tervezésével, elkészítésével és kiértékelésével, hogy megállapítsa, hogy a szoftvertermék teljesíti-e a meghatározott követelményeket, megfelel-e a célnak. A tesztelés felelős a szoftvertermékkel kapcsolatos hibák megtalálásáért.

A tesztelésnek négy fokozatát különböztetjük meg:

Komponens teszt: (Component Test) A komponens teszt során a szoftver legkisebb önállóan tesztelhető egységének (komponens) működését teszteljük.

Integrációs teszt: (Integration Test) Több komponens együttes (integrált) tesztelése. Célja a komponensek közötti interfészekben, illetve kölcsönhatásokban lévő hibák megtalálása.

Rendszer teszt: (System Test) Integrált, teljes rendszer tesztje. Célja a követelményeknek való megfelelés vizsgálata.

Átvételi teszt: (Acceptance Test) A felhasználó vagy megrendelő által a végtermékeken végzett fekete doboz teszt. Célja eldönteni, hogy a termék valóban megfelel-e a megfogalmazott elvárásoknak.

Egy szoftver teszteltségének mérésére a következő kódlefedettségi mérőszámok szolgálnak:

Utasítás lefedettség: (Statement Coverage - SC) 100% utasítás lefedettség azt jelenti, hogy a teszt futása során a kód minden utasítása legalább egyszer lefutott.

Elágazás lefedettség: (Branch Coverage - BC) 100% elágazás lefedettség azt jelenti, hogy a teszt futása során a kódban előforduló minden ágon legalább egyszer végigment a program.

Döntési lefedettség: (Decision Coverage - DC) 100% döntési lefedettség azt jelenti, hogy a teszt futása során a kódban előforduló összes döntés minden lehetséges kimenetele előfordult.

Feltétel lefedettség: (Condition Coverage - CC) 100% feltétel lefedettség azt jelenti, hogy a teszt futása során a döntésekben szereplő minden egyes feltétel minden lehetséges értékre legalább egyszer kiértékelődött.

Döntési feltétel lefedettség: (Condition Decision Coverage - C/DC) 100% döntési feltétel lefedettség azt jelenti, hogy a teszt futása során a kódban szereplő összes döntés minden kimenetele előfordult, miközben a döntésekben résztvevő feltételek minden lehetséges értékre kiértékelődtek legalább egyszer.

Módosított döntési feltétel lefedettség: (Modified Condition Decision Coverage - MC/DC) Ez a legösszetettebb mérőszám. 100% módosított döntési feltétel lefedettség azt jelenti, hogy a teszt futása során a kódban szereplő összes döntés minden kimenetele előfordult, miközben a döntésekben résztvevő feltételek minden lehetséges értékre kiértékelődtek, és a döntés kimenete az abban résztvevő minden változótól legalább egyszer függött.

100% döntési feltétel lefedettség nem garantálja, hogy valóban minden eset előfordult, mivel lehetséges, hogy egy négyváltozós VAGY kapcsolat esetében például az egyik változó maszkolja a többi hármat, és azok hiába vesznek fel minden lehetséges értéket, a program végrehajtása ettől valójában nem függ. A módosított döntési feltétel lefedettség előírja, hogy minden változótól legalább egyszer függenie kell a program végrehajtásának.

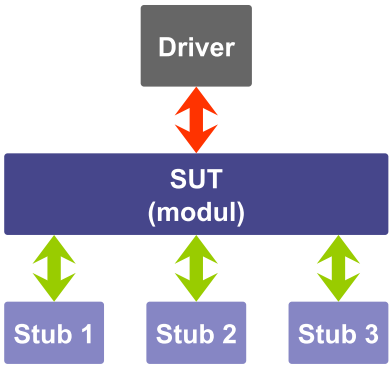
5.2. A tesztelés folyamata

A tesztelés úgynevezett fekete doboz teszteléssel történik. Ez azt jelenti, hogy a modul belső működését nem, csak a kívülről látható viselkedését vizsgáljuk. Ezt csakis az API függvényein keresztül valósíthatjuk meg.

Az AUTOSAR BSW modulok önállóan történő, komponens tesztelésének mechanizmusa az ábrán látható *(5.1. ábra)*. A tesztelt komponens vagy SUT (System Under Test) gerjesztését a meghajtó (driver) szolgáltatja. A driver látja el a kontextusából kiemelt komponens felett elhelyezkedő többi modul szerepét, ő hívja a SUT API függvényeit.

A SUT alatt elhelyezkedő szoftvermodulok feladatát a tesztelés alatt az úgynevezett stub-ok valósítják meg. Ezek általában üres, belső működés nélküli függvények, me-

lyek segítségével vizsgálhatók a tesztelt modul által hívott függvények paraméterei és ezen keresztül a modul válasza a driver által adott gerjesztésre.



5.1. ábra. *A tesztelés módszere*

5.3. A tesztelés folyamatának bemutatása CUnit környezetben a FlexRay Interface egy függvényén

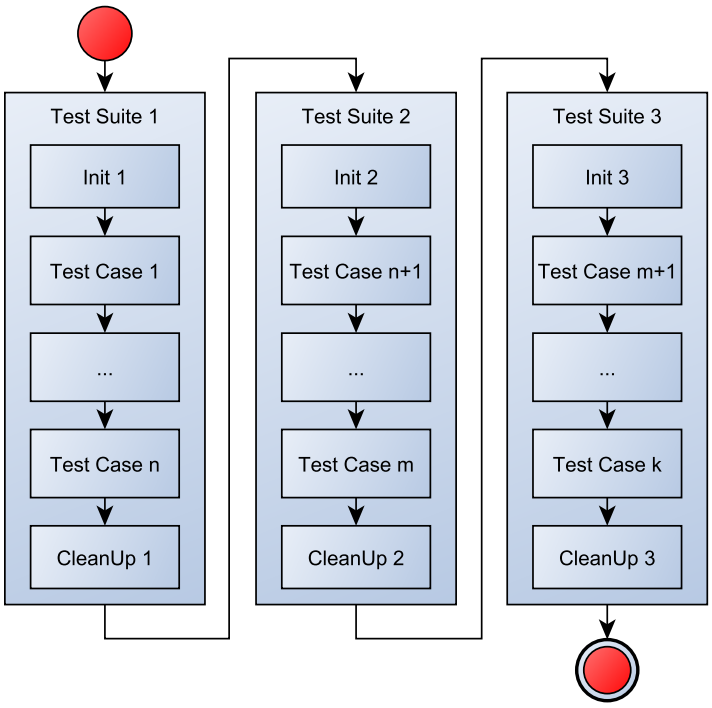
A CUnit egy C függvénykönyvtár, mely a tesztelést segítő előre definiált adminisztrációs és összehasonlító függvényeket tartalmaz. Egy CUnit teszt több tesztkészletet, úgynevezett test suite-ot tartalmazhat. A test suite-ok teszteseteket, úgynevezett test case-eket tartalmaznak, melyek már egy-egy függvényként vannak megvalósítva. Minden tesztkészlethez rendelhető egy `Init` és egy `CleanUp` függvény, mely a teszt készlet futtatása előtt és után fut le, elvégezve a megfelelő inicializálási és esetleges lezáró műveleteket. *(5.2. ábra)*

A CUnit-os teszt `main()` függvényének elején megtörténik a test suit-ok és test case-ek, valamint inicializáló és lezáró függvények beregisztrálása a futtató környezetbe. Ezután a tesztek egymás után lefutnak.

A nyílt forráskódú gcc toolchain gcov eszköze képes a teszt lefuttatása után a kódlefedettségi mérőszámok számítására. Ez akár make környezetből is megtehető, így az egyszer már megírt, tesztelésre alkalmas tesztkészletek automatizáltan, felhasználói beavatkozás nélkül, újra és újra lefuttathatók a modul forráskódján történő esetleges változtatásokat követően. A CUnit és a gcov együttes használatával egy ingyenes, nyílt forráskódú tesztkörnyezet áll rendelkezésünkre a modulok teszteléséhez.

5.3.1. Stub függvények, paraméter vizsgálatok

A tesztelt modul által hívott függvények stub-jait úgy valósítottam meg, hogy meghívásuk után hozzáférjünk a függvény számára átadott paraméterekhez. Ennek legegyszerűbb módja, ha a stub függvények csupán annyit tesznek, hogy a nekik átadott



5.2. ábra. *Egy teszt felépítése*

paramétereket globális változókba mentik, melyeken keresztül később a vizsgálható, hogy a tesztelt modul milyen paraméterekkel hívta az adott függvényt. Szintén globális változókkal oldható meg a hívott stub függvény visszatérési értékének előre beállítása.

A paraméterek vizsgálatára számos, a paraméter típusától függő függvény áll rendelkezésünkre. Az előre definiált, úgynevezett assert függvényekkel számos összehasonlítás végezhető. Az összehasonlítások eredményét a környezet adminisztrálja.

5.3.2. Példa egy tesztkészletre

Az ismertetendő példa során a következő előírásokat tesztelem:

- a modul kiválasztott API függvénye a modul inicializálása előtt nem hívható, hívása hibát okoz
- a modul `NULL` pointerrel történő inicializálása hibát okoz,
- a modul inicializálása után a kiválasztott függvény hívható,
- a kiválasztott függvény a konfigurációnak megfelelő driver függvényét hívja meg a megfelelő paraméterekkel

A tesztelt függvények:

```
void FrIf_Init(
```

```
    const FrIf_ConfigType* FrIf_ConfigPtr
);

Std_ReturnType FrIf_EnableAbsoluteTimerIRQ(
    uint8 FrIf_CtrlIdx,
    uint8 FrIf_AbsTimerIdx
);
```

A tesztkészlet:

Init: Az `Init` függvényben összeállítom a tesztelés során használt konfigurációs adatstruktúrát.

1. test case: Az első teszteset a FlexRay Interface `FrIf_EnableAbsoluteTimerIRQ(...)` függvénynek a modul incializálását megelőző meghívását teszteli. Az szabvány által előírt viselkedés szerint a függvénynek ebben az esetben értesítenie kell a Development Error Tracer modult a `Det_ReportError(...)` függvényen keresztül, és `E_NOT_OK` értékkel kell visszatérnie.

1. A függvény visszatérési értékének mentésére szolgáló `tmp` változó értékének `E_OK`-ra, a `Det_ReportError(...)` stub függvény által beállítandó globális változók értékének pedig 255-re állítása:
   ```
   tmp = E_OK;
   det_modId = 255;
   det_errId = 255;
   det_instId = 255;
   det_apiId = 255;
   ```
2. Meghívódik az `FrIf_EnableAbsoluteTimer` függvény:
   ```
   tmp = FrIf_EnableAbsoluteTimerIRQ(0, 0);
   ```
3. Visszatérési érték ellenőrzése:
   ```
   CU_ASSERT_EQUAL(tmp, E_NOT_OK);
   ```
4. A `Det_ReportError(...)` függvényhívás megtörténtének ellenőrzése:
   ```
   CU_ASSERT_EQUAL(det_modId, FRIF_MODULE_ID);
   CU_ASSERT_EQUAL(det_errId, FRIF_E_NOT_INITIALIZED);
   CU_ASSERT_EQUAL(det_instId, FRIF_INSTANCE_ID);
   CU_ASSERT_EQUAL(det_apiId, FRIF_ENABLEABSOLUTETIMERIRQ_API_ID);
   ```

2. test case: A következő tesztesetben az `FrIf_Init(...)` függvény `NULL` pointerrel történő meghívását vizsgálom. A szabvány előírásai szerint a függvénynek ebben az esetben értesítenie kell a DET-et és inicializálatlan állapotban kell maradnia.

1. Az előző esetben ismertetett változók alaphelyzetbe állítása.
2. Az `FrIf_Init(...)` függvény meghívása `NULL` pointerrel:
   ```
   FrIf_Init(NULL_PTR);
   ```

3. Az első esetben ismertetett módon a változók vizsgálata.
4. Az első teszteset ismételt meghívása az inicializálatlan állapotról való meggyőződés érdekében:

```
testcase1();
```

3. test case: Ebben a tesztesetben a modul előzőleg összeállított konfigurációs struktúrával való inicializálása után az `FrIf_EnableAbsoluteTimerIRQ(...)` függvény helyes működését vizsgálom. A konfigurációban a globálisan 4-es indexű kontrollerhez a 0-ás driver 2-es kontrollere tartozik. Az elvárt működés szerint az FrIf meg kell, hogy hívja a 0-ás driver azonos függvényét, a 2-es kontroller indexet, valamint a megadott időzítő indexét átadva neki. A függvénynek a driver által visszaadott visszatérési értéket kell visszaadnia.

1. A változók alaphelyzetbe állítása. (A drivert helyettesítő stub függvény a `ret_val` változó értékét fogja visszaadni.)

```
ret_val = E_OK;
drv0_cntIdx = 255;
drv0_tmrIdx = 255;
tmp = E_NOT_OK;
```

2. A modul inicializálása a létrehozott konfigurációval.

```
FrIf_Init(&config);
```

3. A függvény meghívása:

```
FrIf_EnableAbsoluteTimerIRQ(4,1);
```

4. Paraméter vizsgálat:

```
CU_ASSERT_EQUAL(tmp, E_OK);
CU_ASSERT_EQUAL(drv0_cntIdx, 2);
CU_ASSERT_EQUAL(drv0_tmrIdx, 1);
```

CleanUp: A `CleanUp` függvény törzse – mivel jelen esetben nincs semmilyen elvégzendő teendő – üres.

Az ismertetett módszerrel és a tesztek megfelelő megtervezésével a modulok működése teljes körűen vizsgálható.

Összefoglalás

A szakdolgozatom készítése során két szabványos AUTOSAR alapszoftver modul C nyelvű implementációját végeztem el. Az első megvalósított modul a FlexRay Interface volt, melynek feladata a FlexRay kontroller és transceiver driverek elrejtése, valamint a FlexRay időzítéseihez ütemezett kommunikációs műveletek végrehajtása. A második modul a FlexRay ISO Transport Layer volt, mely a nagyméretű üzenetek szegmentálását, valamint újraegyesítését végzi.

A modulok megvalósításához elengedhetetlen volt a FlexRay protokoll részletes megismerése, különös tekintettel annak időzítéseire. Munkámat a modulokra vonatkozó ISO és AUTOSAR szabvány megismerésével folytattam, melyhez számos egyéb modulokra és általános elvekre vonatkozó specifikáció elolvasása is szükséges volt. A megszerzett elméleti tudást a dolgozat első fejezetében foglaltam össze.

Az AUTOSAR kommunikációs moduljait és az adatcsere folyamatát a második fejezetben részletesen is ismertettem. Külön hangsúlyt fektettem a modulok együttműködésének bemutatására, ezt egy példán is szemléltettem.

Az implementációt mindkét esetben a konfigurációs valamint futásidejű adatstruktúrák megtervezésével, majd felépítésével kezdtem. Ezután megvizsgáltam a lehetséges konfigurációtól függő egyszerűsítési lehetőségeket, és elkezdtem a működést megvalósító algoritmusok megtervezését.

A FlexRay ISO Transport Layer esetében külön nehézséget okozott az ISO szabványban definiált protokoll AUTOSAR szabványban megfogalmazott módon való implementálása. Ez a modul csak az AUTOSAR jelenlegi, 4.0-ás verziójában jelent meg először, így a szabvány sajnos sok esetben még nem elég kiforrott. A specifikáció megértése sokszor csak a Transport Layer-t használó modulok működésének tanulmányozása alapján volt lehetséges, mely sok időt vett igénybe.

Az implementáció elvégzése után megkezdtem a modulok tesztelését. Ehhez elvégeztem a modulok stub függvényeinek megvalósítását, majd az egyszerűbb tesztkészletek megírását a szabványpontoknak való megfelelés igazolására.

A továbbiakban szükséges a tesztkészletek továbbfejlesztése. A dolgozatban leírt elvek alapján törekedni kell arra, hogy a tesztesetek minél nagyobb kódlefedettséget érjenek el. További lépés lehet a teljes FlexRay stack integrációs tesztelése a szabványhoz készülő acceptance tesztek alapján.

Ábrák jegyzéke

Rövidítések és kifejezések jegyzéke

Acceptance Test	átvételi teszt
ACK	Acknowledge – visszaigazolás
AI	Address Information – cím információ
API	Application Programming Interface – Egy szoftvermodul által nyújtott nyilvános (kívülről látható) függvényei és adattípusai.
ASIC	Application-Specific Integrated Circuit – alkalmazás-specifikus integrált áramkör
AUTOSAR	AUTomotive Open System ARchitecture – Autóipari Nyílt Rendszer Architektúra
BC	Bandwidth Control – sávszélesség szabályozás
BC	Branch Coverage – elágazás lefedettség
BfS	Buffer Size – buffer méret
BM	Bus Minus – a FlexRay busz logikai „1” érték esetén alacsonyabb potenciálon lévő vezetéke
BP	Bus Plus – a FlexRay busz logikai „1” érték esetén magasabb potenciálon lévő vezetéke
BP	Byte Pointer – bájt mutató
BSS	Byte Start Sequence – bájt kezdő szekvencia
BSW	Basic SoftWare – alap szoftver
burst	löket
C/DC	Condition Decision Coverage – döntési feltétel lefedettség
CAN	Controller Area Network
CC	Condition Coverage – feltétel lefedettség

CF	Consecutive Frame – „köztes keret”, szegmentált átvitel esetén az adatszegmensek átvitelére szolgáló keretek
CF_EOB	Consecutive Frame End Of Bytes – az adott átviteli blokk utolsó CF-e
Channel	csatorna
cluster	az egymással fizikai összeköttetésben álló FlexRay végpontok halmaza
COM	Communication BSW modul
ComM	Communication Manager BSW modul
Component Test	komponens teszt
CRC	Cyclic Redundancy Check – ciklikus redundancia vizsgálat
CSMA/CA	Carrier Sense Multiple Access with Collision Avoidance – a csatorna figyelésén alapuló többszörös hozzáférés, ütközés elkerüléssel
CSMA/CD	Carrier Sense Multiple Access with Collision Detection – a csatorna figyelésén alapuló többszörös hozzáférés, ütközés detektálással
DC	Decision Coverage – döntés lefedettség
DC	egyenáramú
DCM	Diagnostic Communication Manager BSW modul
Decoupled Buffer Access	függetlenített buffer hozzáférés
DET	Development Error Tracer BSW modul
DTS	Dynamic Trailing Sequence – dinamikus lezárószekvencia
ECU	Electronic Control Unit – elektronikus vezérlőegység
FC	Flow Control – az adatáramlást befolyásoló PDU
FC_ABT	Flow Control Abort – az átvitel megszakítását kérő PDU
FC_ACK_RET	Flow Control Acknowledge/Retry – az adattovábbítás visszaigazolására, vagy újraküldés kérésére szolgáló PDU
FC_CTS	Flow Control Continue To Send – adattovábbítás folytatását kérő PDU

FC_OVFLW	Flow Control Overflow – a buffer túlcsordulását jelző PDU
FC_WT	Flow Control Wait – várakozást kérő PDU
FES	Frame End Sequence
FPL	Frame Payload Length – keret adatbájtjainak száma
Fr	FlexRay Driver BSW modul
frame	keret
FrArTp	FlexRay AUTOSAR Transport Layer modul
FrIf	FlexRay Interface BSW modul
FrNm	FlexRay Network Management BSW modul
FrSM	FlexRay State Manager BSW modul
FrTp	FlexRay ISO Transport Layer BSW modul
FrTrcv	FlexRay Transceiver Driver BSW modul
FS	Flow Status – az FC PDU-k funkcióját meghatározó mező
FSS	Frame Start Sequence – keret kezdő szekvencia
FTDMA	Flexible TDMA – rugalmas TDMA
gateway	átjáró
Header	a FlexRay keretek fejléce
idle	üresjárati
Immediate Buffer Access	azonnali buffer hozzáférés
inline függvény	a függvény hívásának helyére fordítás során a függvény törzse kerül behelyettesítésre
Integration Test	integrációs teszt
IpduM	I-PDU Multiplexer BSW modul
LF	Last Frame – az átvitelt lezáró PDU
LIN	Local Interconnect Network
link time	Linkelési idejű konfigurációs paraméterek.
LocalAddress	helyi cím
MAC	Media Access Control – az ISO/OSI modell adatkapcsolati rétegének közeghozzáférési alrétege

MC/DC	Modified Condition Decision Coverage – módosított döntési feltétel lefedettség
MNPC	Maximum Number of PDUs Per Cycle – az egy FlexRay ciklusban küldhető PDU-k maximális száma
NIT	Network Idle Time – hálózati üresjárási idő
NmIf	Network Management Interface BSW modul
node	végpont
notification	jelzések, melyeken keresztül a COM az RTE-t különböző eseményekről értesíti
overhead	fölösleges többletterhelés
Payload	a FlexRay keretek adatbájtokat tartalmazó része
PCI	Protocol Control Information – protokoll vezérlő információ
PDU	Protocol Data Unit – protokoll adategység
PduR	PDU Router BSW modul
post-build time	Futásidejű konfigurációs paraméterek.
pre-compile time	Fordítás előtti konfigurációs paraméterek.
Remote Address	távoli cím
RET	Retry – újraküldés
RTE	Runtime Environment – futtató környezet
runtime	futásidejű
SA	Source Address – forrás cím
SC	Separation Cycle – a PDU csoportok elküldése között kivárandó FlexRay ciklusok száma
SC	Statement Coverage – utasítás lefedettség
SCexp	Separation Cycle exponent – a PDU csoportok elküldése között kivárandó FlexRay ciklusok száma számítható belőle az $SC = 2^{SCexp} - 1$ képlet alapján
SDU	Service Data Unit – szolgáltatás adategység
slot	időrés
SN	Sequence Number – az adott CF blokkon belüli sorszáma

SPI	Serial Peripheral Interface – soros periféria interfész
stack	Az architektúrának – funkcióját és függőségeit tekintve – jól elkülöníthető része.
STFA	Start Frame Acknowledged – a visszaigazolandó átvitel kezdetét jelző keret
STFU	Start Frame Unacknowledged – a nem-visszaigazolandó átvitel kezdetét jelző keret
SUT	System Under Test – tesztelt komponens
SW-C	Software Component – AUTOSAR szoftverkomponens
System Test	rendszer teszt
TA	Target Address – cél cím
TDMA	Time Division Multiple Access – időosztásos többszörös hozzáférés
Trailer	a FlexRay kereteket lezáró tag
transceiver	busz-illesztő
TSS	Transmission Start Sequence – átvitelindító szekvencia
TTCAN	Time Triggered CAN – idővezérelt CAN
update bit	a FlexRay L-PDU-kban egy I-PDU-hoz rendelt update bit az I-PDU frissített állapotát jelzi
UTP	Unshielded Twisted Pair – árnyékolás nélküli, csavart érpárú (vezeték)
VFB	Virtual Function Bus – Virtuális Függvénybusz

Irodalomjegyzék

[1] AUTOSAR Consortium. *AUTOSAR The Worldwide Automotive Standard for E/E Systems*, 2012. http://www.autosar.org/download/papersandpresentations/AUTOSAR_Brochure_EN.pdf.

[2] AUTOSAR Consortium. *Layered Software Architecture R4.0 Rev 3*, 2012. http://www.http://autosar.org/download/R4.0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf.

[3] FlexRay Consortium. *FlexRay Communications System Electrical Physical Layer Specification Version 3.0.1*, október 2010. http://www.flexray.com.

[4] FlexRay Consortium. *FlexRay Communications System Protocol Specification Version 3.0.1*, október 2010. http://www.flexray.com.

[5] AUTOSAR Consortium. *Specification of Communication R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_COM.pdf.

[6] AUTOSAR Consortium. *Specification of Diagnostic Communication Manager Interface R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_DiagnosticCommunicationManager.pdf.

[7] AUTOSAR Consortium. *Specification of Debugging in AUTOSAR R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_Debugging.pdf.

[8] AUTOSAR Consortium. *Specification of Communication Manager R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_COMManager.pdf.

[9] AUTOSAR Consortium. *Specification of Network Management Interface R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_NetworkManagementInterface.pdf.

[10] AUTOSAR Consortium. *Specification of I-PDU Multiplexer R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_IPDUMultiplexer.pdf.

[11] AUTOSAR Consortium. *Specification of PDU Router R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_PDURouter.pdf.

[12] AUTOSAR Consortium. *Specification of FlexRay AUTOSAR Transport Layer R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayARTransportLayer.pdf.

[13] AUTOSAR Consortium. *Specification of FlexRay ISO Transport Layer R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayISOTransportLayer.pdf.

[14] AUTOSAR Consortium. *Specification of FlexRay Interface R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayInterface.pdf.

[15] AUTOSAR Consortium. *Specification of FlexRay Driver R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayDriver.pdf.

[16] AUTOSAR Consortium. *Specification of FlexRay Transceiver Driver R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayTransceiverDriver.pdf.

[17] AUTOSAR Consortium. *Specification of FlexRay Network Management R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayNetworkManagement.pdf.

[18] AUTOSAR Consortium. *Specification of FlexRay State Manager R4.0 Rev 3*, 2012. http://www.autosar.org/download/R4.0/AUTOSAR_SWS_FlexRayStateManager.pdf.

[19] International Organization for Standardization. *ISO 10681-2 Road vehicles – Communication on FlexRay – Part 2: Communication layer services*, first edition, június 2010.

[20] Magyar Szoftvertesztelői Tanács Egyesület. *Szoftvertesztelés egységesített kifejezéseinek gyűjteménye R3.12*, február 2012.