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

Kotnyek Bence

AUTOSAR XCP MODUL MEGVALÓSÍTÁSA

TÉMAVEZETŐ

Dr. Sujbert László

KONZULENS

Szikszay László (thyssenkrupp Components Technology Hungary Kft.)

BUDAPEST, 2020

MŰEGYETEM 1782

DIPLOMATERVEZÉSI FELADAT

Kotnyek Bence (AEUCLQ)
szigorló villamosmérnök hallgató részére

AUTOSAR XCP modul megvalósítása

A korszerű járművek összetett funkcióit egymással kommunikációs kapcsolatban álló vezérlőegységek (ECU) valósítják meg. A kommunikáció különböző szabványos autóipari protokollokon (pl. CAN, FlexRay, LIN) zajlik. Az ECU-kon megvalósított (elsősorban szabályozó jellegű) algoritmusok hangolására szolgál az XCP protokoll, amely lehetővé teszi egyes ECU-k változóinak egyszeri vagy periodikus olvasását, külső eszközön (pl. egy a járműbuszra csatolt laptop képernyőjén) való megjelenítését, illetve a változók kívülről való írását. Az XCP több kommunikációs protokollra is ráültethető, a szervezetnél jelenleg használt megvalósítás a hagyományos CAN és a FlexRay buszt támogatja.

A vezető autógyártók által 2002-ben életre hívott AUTOSAR konzorcium célja az, hogy specifikáljon egy (i) alapvető szolgáltatásstruktúrát, amely eltakarja a hardver sajátosságait és támogatja az alkalmazási szoftver hordozhatóságát (base software stack, BSW), (ii) egy modellezési nyelvet az ECU-kon futó alkalmazási szoftver szabványos leírására (software component template), és (iii) az alkalmazások és BSW-k ECU-n belüli és ECU-k közti transzparens kommunikációját lehetővé tevő elosztott runtime szolgáltatást (RTE). Az AUTOSAR egy élő, aktívan fejlesztett szabvány, amelynek a közelmúltban jelent meg a 4.3-as verziója. A jelölt feladata egy régebbi AUTOSAR Base Software Stack megvalósítás (4.0 szabvány verzió) XCP funkcióinak továbbfejlesztése az újabb szabványnak megfelelően az alábbiak szerint:

- A szabvány kapcsolódó részeinek megismerése: (i) Ismerje meg az ASAM és AUTOSAR szabvány XCP-re vonatkozó részeit, (ii) foglalja össze az XCP modul 4.0 és 4.3 AUTOSAR szabványban tapasztalható különbségeit, (iii) ismerje meg az AUTOSAR kommunikációs moduljainak struktúráját és releváns interfészeit.
- Szoftvertervezés és megvalósítás: (i) Ismerje meg a szervezet XCP moduljának felépítését, (ii) tervezze meg az XCP modult érintő szükséges változásokat, majd (iii) valósítsa meg a szükséges módosításokat az XCP modul statikus (kézzel írt, minden konfigurációban azonos) részében illetve a dinamikus (konfigurációtól függő) részét előállító kódgenerátorban.
- A megvalósítás tesztelése: A modul helyességének vizsgálatához (i) készítsen modulteszt-infrastruktúrát, melyben tervezzen és valósítson meg teszteseteket, (ii) demonstrálja a megvalósított modul működését a célhardveren, illetve (iii) szükség esetén javítsa a megvalósítást.

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

Budapest, 2020.03.15.

……..
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár

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

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2020. 12. 20.

Bence Kotnyek

Digitally signed by Bence Kotnyek
Date: 2020.12.20 09:16:00 +01'00'

.....

Kotnyek Bence

Összefoglaló

Az autóiparban a szabványcsaládok változásai lekövetik a felmerülő új igényeket és funkciókat. Az újabb és újabb verziójú szabványok egyre jobban optimalizálják a működést. A változtatásokat az autóipari alkatrészgyártó cégeknek is követni kell.

A diplomamunkám során az AUTOSAR szabvány szerint implementáltam az XCP diagnosztikai protokollnak a nemrég megjelent 4.3.0-ás szoftververzióját. Megismertem a modul működését, a modullal kapcsolatban álló protokollokat és az előző 4.0.3-as verziót is. A két verzió közötti változtatásokat megkerestem és a módosításait megterveztem.

A tervezés befejeztével az implementációban a statikus és dinamikus kódot úgy módosítottam, hogy az új verzió szerint is megfelelően működjön. Az implementáció végeztével teszteltem az elkészült modult. Erre a funkcionális modulteszben teszteseteket készítettem úgy, hogy az implementált modul minden részét meghajtotta. A teszt során felismert hibákat javítottam az implementációban illetve esetlegesen a tesztben.

A célplatformon való tesztelés menetét megismertem. A célplatformon való tesztelést a vírushelyzet miatt nem tudtam elvégezni, viszont a diplomamunka enélkül is teljes értékű tud lenni. Ezt egyeztettem a konzulenssel és a témavezetővel is.

Abstract

In the automotive industry, changes in standard families are keeping pace with emerging new needs and features. These newer and newer versions of standards increasingly optimize performance. Changes must also be followed by automotive supplier companies.

During my dissertation, I implemented the recently released 4.3.0 software version of the XCP diagnostic protocol according to the AUTOSAR standard. During my task, I also learned about how the module works, the protocols associated with the module, the protocols associated with the module and the previous 4.0.3 version. I looked for changes between the two versions and planned the changes.

After completing the design, I modified the static and dynamic code in the implementation so that it would work properly according to the new version. After the implementation, I tested the completed module. I created test cases by driving all parts of the implemented module according to the functional module test. I corrected the errors detected during the test in the implementation and possibly in the test.

I got acquainted with the process of testing on the target platform. I was not able to perform the testing on the target platform due to the virus situation, but the thesis can be complete without it. I also discussed this with the consultant and the supervisor.

1 Bevezetés

Az autóipar és az igények növekedése szükségessé teszik a szabványok fejlődését is. A szabványok fejlesztése elengedhetetlen, hiszen így lehet az autókkal szemben támasztott egyre nagyobb igényeket megvalósítani. Mivel az autóban található vezérlőegységek száma is megnőtt, ezért meghibásodás esetén célszerű a vezérlőegységek adatait kiolvasni a járműből, hogy ezek a hibák kijavíthatók legyenek. Ebből a célból hozták létre az XCP protokollt, amellyel diagnosztikai és kalibrációs műveleteket lehet végezni.

A diplomatervezési feladatom az autóiparban használt XCP kalibrációs protokollnak a megvalósítása és tesztelése volt. Először elengedhetetlen az XCP modulnak a megismerése, a vele kapcsolatban lévő protokolloknak tanulmányozása és a vele szemben támasztott követelmények megismerése. A rendelkezésre álló 4.0.3-as verziójú XCP modulnak a továbbfejlesztése a feladat.

A működés megismerése után meg kellett tervezni a változtatások implementációját. A régi és az új verziójú XCP modul összehasonlítása a tervezési fázist segíti.

Miután megismertük a modulnak a működését, elkezdődhet az implementáció. A statikus és a dinamikus kódot is implementálni kell a 4.3.0-ás AUTOSAR szabvány szerint. A kód elkészülte után különböző konfigurációkat kell készíteni, amelyek segítségével az implementáció során megvalósított funkciókat lehet meghajtani a későbbiekben.

Az implementáció után fontos tesztet készíteni a megvalósított modulhoz. A követelmények alapján teszteket kell létrehozni, amelyek segítségével funkcionális modulteszttel és kódfedettség vizsgálatokkal sok hibát ki lehet szűrni.

A tesztek után a célplatformon való ellenőrzésre kerül sor, ahol hiba esetén még módosítani lehet az implementáción.

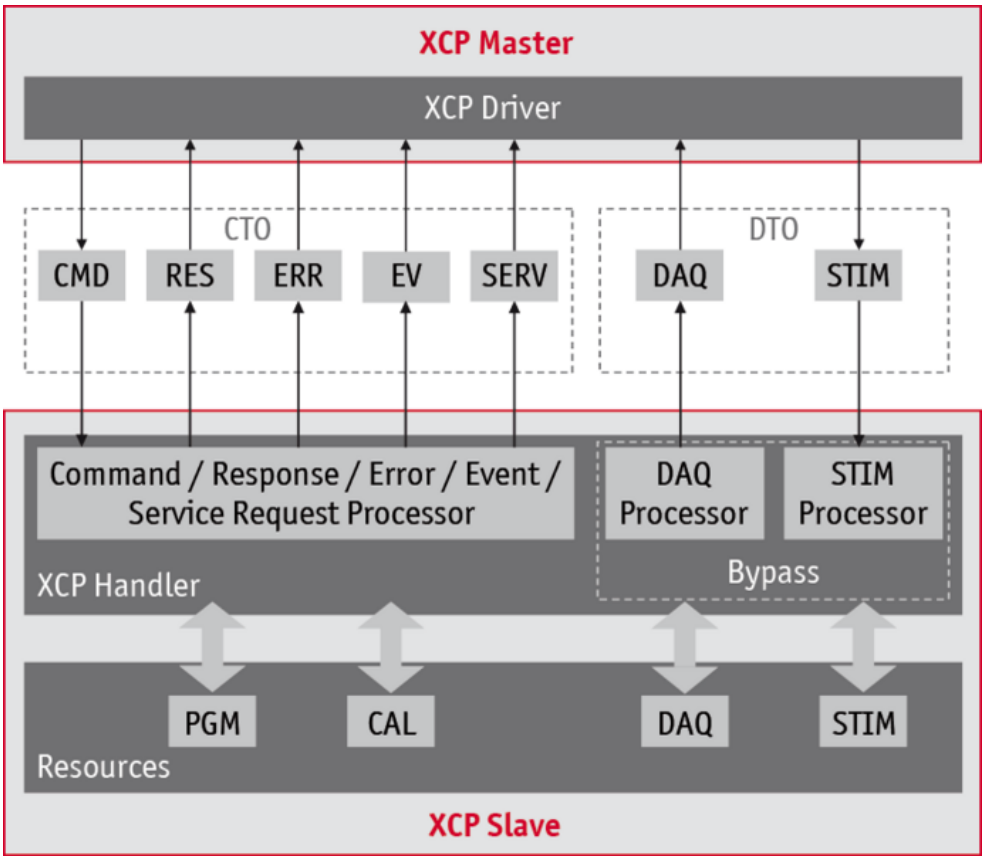
2 Műszaki háttér

2.1 ASAM szabványcsalád

Az ASAM (**A**ssociation for **S**tandardisation of **A**utomation and **M**easuring Systems) egy nonprofit szervezet, amely elősegíti a toolchain-ek szabványosítását az autóipar fejlesztésében és tesztelésében. Ahogy az AUTOSAR esetén, az ASAM tagjai is nemzetközi autógyártók, beszállítók, szerszámgyártók, mérnöki szolgáltatók és autóipari kutatóintézetek. Az ASAM szabványokat munkacsoportokban dolgozzák ki, amelyek a tagvállalatok szakértőiből állnak. Lehetővé teszik az adatok vagy eszközök egyszerű cseréjét a toolchain-en belül. A piacon található autók több mint 90 százaléka kompatibilis az ASAM szabványokkal. Az ASAM szabványok között van meghatározva az XCP protokoll is [7].

2.1.1 XCP protokoll

Az XCP protokoll (e**X**tensible **C**alibration **P**rotocol) egy adatgyűjtő és kalibrációs protokoll. A CCP (**C**AN **C**alibration **P**rotocol) protokollra építve többfajta protokollal való működést támogat. A CCP protokoll is ASAM szabvány. Az XCP protokoll CAN, FlexRay, Ethernet, USB és soros portnak a kommunikációját támogatja. Az XCP felépítése a 1. ábrán látható [6], [9], [10].



1. ábra: Az XCP master-slave elrendezése [9]

Az XCP protokoll egy master slave kapcsolatot valósít meg. A master az egy fejlesztői számítógép egy XCP meghajtóval és szoftverrel, a slave pedig az ECU. A diplomatervezés során a slave eszközön kell a protokollt megvalósítani.

2.1.1.1 XCP csomagok típusai

Az XCP protokoll több fajta csomagon keresztül valósítja meg a felek közötti kommunikációt. Az egyik ilyen kommunikációs csomag a CTO csomag (**C**ommand **T**ransfer **O**bject). A CTO csomagok vezérlési üzeneteket tartalmaznak:

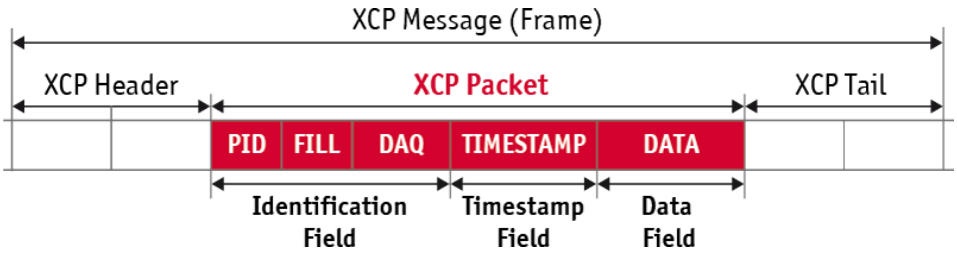
- CMD (**C**o**M**man**D**),
- RES (command **RES**ponse packet),
- ERR (**ERR**or Packet),
- EV (**EV**ent Packet),
- SERV (**SERV**ice Request Packet)

A CTO segítségével parancsokat (CMD) küldhet a master, illetve ezekre választ (RES) a slave. Ezenkívül a slave küldhet hibaüzeneteket (ERR), esemény értesítéseket (EV), és kiszolgálási kéréseket (SERV) [6], [10], [11].

A CTO csomagok specifikus hexadecimális értékek alapján különböztethetők meg. A 0xC0 - 0xFF hexadecimális értékeket vehetik fel a CTO parancsok. A csatlakozás (connect) a 0xFF, a lecsatlakozás (disconnect) a 0xFE, az állapot lekérdezés a 0xFD hexadecimális érték és így tovább [6], [9].

A DTO csomagok (**D**ata **T**ransfer **O**bject) adatok továbbítására szolgálnak. A slave egység kezeli és állítja össze a DTO csomagokat. A DTO csomagok kezelését a slave-ben megvalósított DAQ (**D**ata **A**c**Q**uisition) adatgyűjtő modul állítja össze. A master irányból érkező csomagokat a slave-ben megvalósított STIM (Data **STIM**ulation packet) feldolgozó modulja kezeli. A fent részletezett XCP csomagok felépítése a 2. ábrán látható [6], [10].

2.1.1.2 XCP csomag felépítése

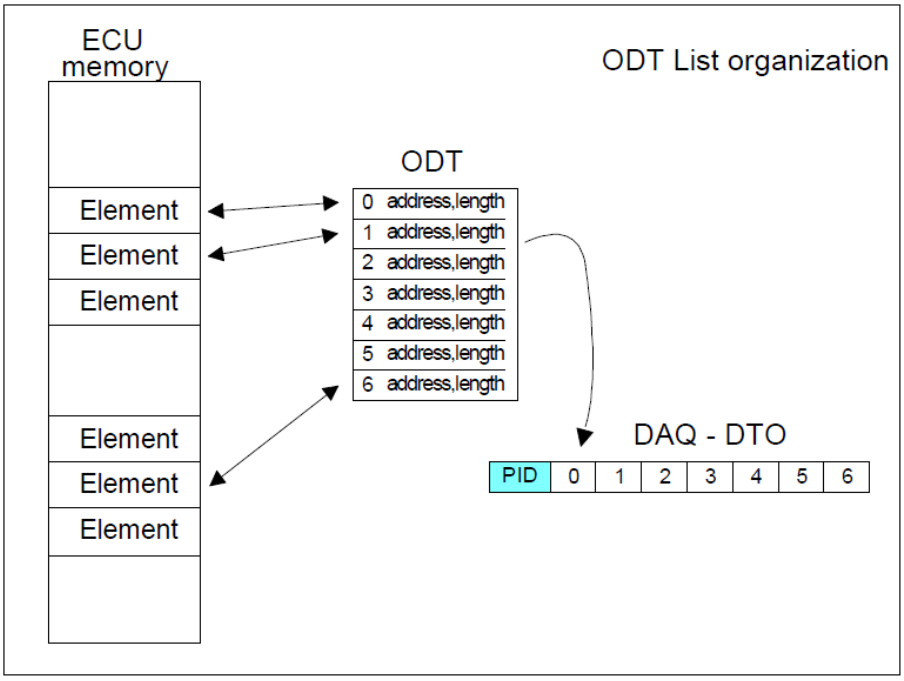


2. ábra: Az XCP üzenet felépítése [9]

Az XCP csomagban a header és a tail része protokollfüggő. A header és a tail-ben helyezkedik el általában a csomag azonosítója, az üzenetszámlálója és a checksum. A PID (**P**acket **ID**) az XCP csomag azonosítója. CTO esetén ennek a parancstól függően előre meghatározott értéke van, DTO esetén pedig a DAQ listában található azonosítót tartalmazza. A DAQ mező a PID mezővel meghatározza a mintavételi listát. A FILL mező akkor szükséges, ha a PID értéket más bájt értékre szeretnénk igazítani. A PID, a FILL és a DAQ mezőket közösen identifikációs mezőnek hívják. A TIMESTAMP mező az időbélyeget tartalmazza arra az esetre, ha mintavételi időt kell meghatározni és továbbítani. A DATA mezőben az adatokat helyezik el. CTO csomag esetén a CTO parancsok paramétereit, DTO csomag esetén a DAQ, vagy a STIM adatait tartalmazza az adatmező [6], [10].

2.1.1.3 Szinkron adatátvitel

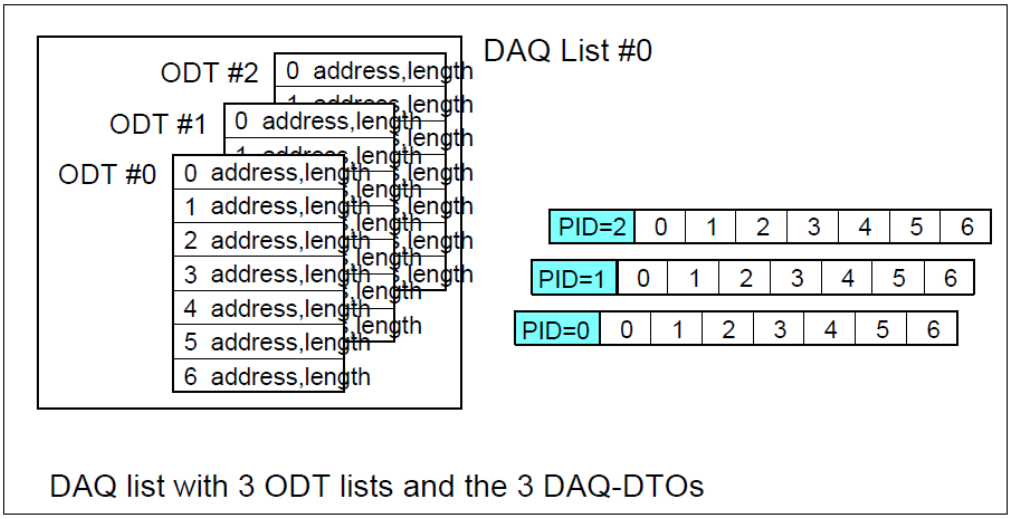
Az XCP modul szinkron adatküldéssel továbbít adatot. A slave-ből induló adatokat DAQ csomagok küldésével lehet megvalósítani. A slave adatküldése ODT listák (**O**bject **D**escription **T**able) segítségével valósítható meg. Az ODT leképezi a slave (ECU) memóriáját a szinkron adatküldéshez. A szinkron adat küldéskor a DTO-ban található PID mező segítségével határozható meg az ODT lista, amelyben az adat található. Az 3. ábra megmutatja az ODT felépítését [6], [10].



3. ábra: Az ODT lista felépítése [17]

Az ODT adatrészekre hivatkozik a memória címük, cím kiterjesztésük és méretük alapján. Egy ODT listában maximum annyi bejegyzés lehet, amennyi belefér egy DTO csomagba [6].

Az XCP modulban több ODT is található. Ezeket a DAQ listákba rendezve lehet megtalálni. Erre egy példa a 4. ábrán látható, amely egy 3 ODT-ből álló DAQ listát mutat be.



4. ábra: DAQ lista [17]

Az ODT listákat DAQ listába lehet szervezni. Bár a DAQ listában található összes ODT független egymástól, minden esemény, amely a DAQ listát triggereli, az a benne található összes ODT listát is triggerelni fogja [10].

A DAQ listából két típust különböztethetünk meg. A statikus DAQ listát és a dinamikus DAQ listát. Statikus DAQ lista esetén az ODT listák maximális száma meg van határozva, így nem lehet hozzáadni működés közben. Dinamikus DAQ esetén nincsen meghatározva ilyen érték [6].

A STIM a DAQ üzeneteknek az ellenkező irányú megfelelője. A STIM csomagok csak a master-ből indulhatnak. Az XCP modulban található irányjelző flag segítségével a DAQ listákat be lehet állítani szinkron adatstimulációs módba. A stimulációs adatokat szintén DTO csomagok segítségével lehet továbbítani. Az ODT megadja a leképezését a DTO és a slave eszköz memóriája között. A STIM processzor puffereli a bejövő adatstimulációs csomagokat. Amikor egy olyan esemény érkezik, amely a DAQ listát triggereli, akkor a pufferelt adatokat átküldi a slave eszköz memóriájába [17].

2.1.1.4 Timeout kezelés

Timeout lép fel, ha a slave egy meghatározott időn belül nem válaszol a master-től a slave felé küldött parancsra.

Parancs küldésekor a master eszköznek el kell indítania egy időzítőt. Minden parancsnál az időzítő által elérhető maximális méretét a Time-Out Value tx érték adja meg. Ha a master választ kap, mielőtt az időzítő eléri a maximális értéket,

akkor a master-nek vissza kell állítania az időzítőt. Ha az időzítő eléri a maximális értéket anélkül, hogy a master választ kapna a slave-től, akkor a master-nek ezt timeout-ként kell észlelnie. Az XCP protokoll 7 különböző időhosszt határoz meg az időtúllépés számára. A működés négy különböző kommunikációs modell szerint lehetséges [12].

- Standard kommunikációs modell [12]:

  Ha a master időtúllépést detektál a standard kommunikációs modellben, akkor a master-nek el kell végeznie az Pre-Action és az Action műveletet.

  Ezt a szekvenciát (Pre-Action, Action) kétszer kell kipróbálni. Ha a master ezek után is kimutat egy időtúllépési hibát, a master eszköz eldöntheti a szükséges reakciót.

  Általános esetben egy SYNCH parancsból áll, amely újraszinkronizálja a parancs végrehajtását a master és a slave között, majd a parancs meg lesz ismételve.

  Bizonyos speciális parancsok esetében az előintézkedés egy jól definiált állapotba hozza a slave-t, például a SET_MTA vagy a SET_DAQ_PTR küldésével, mielőtt megismételné a parancsot [17].

- Blokk kommunikációs modell [12]:

  Blokk kommunikáció esetén a master egymás után küld csomagokat a slave irányába és akkor indítja a master a timeout-ot, amikor az utolsó csomag kerül kiküldésre.

  Ha a master egy timeout értéket jelez a blokk kommunikációs modellben, akkor a master-nek ugyanazt a hibakezelést kell végrehajtania, mint a standard kommunikációs modellnél.

  A blokk kommunikációs módban a master-nek el kell indítania az timeout észleléséhez használt időzítőt, amikor elküldi a parancsot felépítő blokk utolsó frame-jét [17].

- Interleaved kommunikációs modell [12]:

A slave eszköz egyszerre több csomagot is fogadni tud egymás után és minden csomagnak külön timeout időzítője indul. Mivel mindegyik csomaghoz külön időzítő tartozik, ezért minden csomagra külön választ kell adni a slave eszköznek.

Ha a master kap egy időzítő lejárt üzenetet az interleaved kommunikációs modellben, akkor a master-nek ugyanazt a hibakezelést kell végrehajtania, mint a standard kommunikációs modell esetén [17].

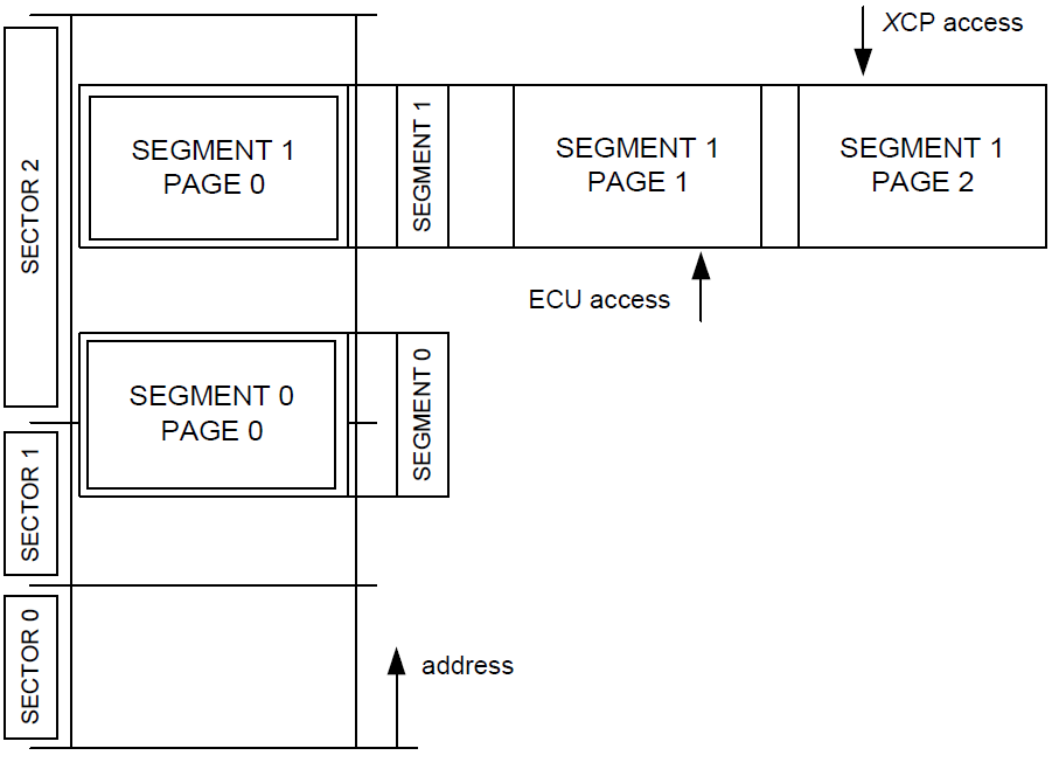
- Időtúllépés manipuláció: A master megkapja az alapértelmezett timeout értékeket t1 és t6 között az ASAM MCD 2MC leíró fájlból.

  Speciális esetben az XCP lehetővé teszi ezen időkorlátok felülbírálását.

  Az EV_CMD_PENDING használatával a slave felkérheti a master-t, hogy indítsa újra az időkorlát észlelését [17].

2.1.1.5 Online kalibráció

Az XCP online kalibrációs modell adat elrendezése az 5. ábrán látható.



5. ábra: A kalibrációs adatok elrendezése [12]

A slave memória-elrendezése egy folyamatos fizikai térként írható le. A memóriában található elemekre 40 bites érték hivatkozik, 32 az XCP cím és 8 az XCP cím kiterjesztése [12].

A memóriát szektorokra (SECTOR) bontja fel az XCP slave eszköz. A szektorok különböző hosszúak lehetnek. A szektorok méretei és a limitációi flasheléskor fontos információk.

A memória logikai elrendezése szegmens (SEGMENT) objektumokkal írhatók le. A szegmensek azt írják le, hogy a kalibrálható adatobjektumok hol helyezkednek el a slave memóriájában. A szegmensnek nem kell figyelembe vennie a szektor által megadott méreteinek korlátozásait, azaz a szegmens kezdőcímét és méretét úgy is meg lehet választani, hogy azok másik szektorba is átlóghatnak [12].

A szegmenseknek több oldala (PAGE) lehet. A szegmensek oldalai ugyanazokat az adatokat ugyanazon címeken írják le, de eltérő tulajdonságokkal, például különböző értékeket vehetnek fel, vagy read/write hozzáférés beállításban térhetnek el. Minden szegmensnek legalább egy oldallal kell rendelkeznie. A slave eszköznek az összes szegmens összes oldalát inicializálnia kell [12].

Abban az esetben, amikor a vezérlő algoritmus adatokat keres a slave eszközben, akkor a slave egyszerre csak egy oldalt tud megnyitni a szegmensektől függetlenül. Amikor a master eszköz parancsok segítségével a slave eszköz memóriájában található adatokra hivatkozik, akkor egyszerre szintén csak egy oldalhoz férhet hozzá. Ezek a slave és master által hivatkozott oldalak egymástól függetlenül kapcsolhatók. Az aktív oldal függetlenül kapcsolható bármely szegmens esetén [12].

A page switching funkció csak akkor használható, ha a SET_CAL_PAGE és a GET_CAL_PAGE parancsok meg vannak valósítva. A master rendelkezik az oldalváltás teljes ellenőrzésével. A slave nem tudja önállóan váltani az oldalait. A master az összes szegmenst szinkron módon is átkapcsolhatja ugyanarra az oldalra [12].

Kalibrációs adatoldalak fagyasztását is kérheti a master eszköz parancsok segítségével. Ebben az esetben a FREEZE üzemmódba kapcsolt oldal tartalma nem változik. A FREEZE kérését a SET_SEGMENT_MODE parancs segítségével jelezheti a master a slave eszköznek. Minden FREEZE üzemmódban lévő szegmens

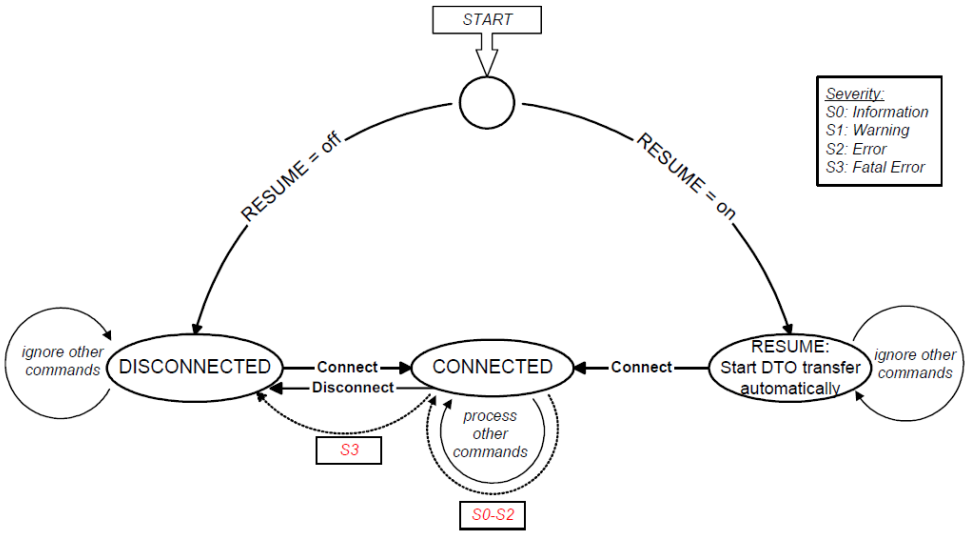
számára a slave eszköznek el kell mentenie a szegmens aktuálisan aktív oldalát az init szegmensének a 0. oldalába [12].

2.1.1.6 XCP állapotgépe

Az ASAM XCP protokoll master – slave kommunikáció alapján működik. Az AUTOSAR szabvány a slave eszközt írja le, amíg a master egy számítógép egy XCP meghajtóval és megfelelő szoftverrel. Az XCP protokoll slave eszköze többfajta üzemmódot vehet fel. Az állapotgépnek üzemmódjai a következők:

- DISCONNECTED
- CONNECTED
- RESUME

Az egyik üzemmódból a másik üzemmódba való átlépést egy állapotgép segíti, amelyben meg van határozva, hogy melyik állapotból hova léphet. Az állapotgép a 6. ábrán látható [6], [10].



6. ábra: Az XCP modul állapotgépe [17]

Az indítás után a slave eszköz megvizsgálja, hogy van-e olyan DAQ lista, amelyet automatikusan indítani kell. Amikor ilyen lista nincs konfigurálva, akkor a DISCONNECTED üzemmódba kapcsol. Ebben az üzemmódban a Connect parancson kívül más parancsra nem reagál. Ha létezik ilyen automatikusan indítandó DAQ lista, akkor a RESUME üzemmódba lép. Ebben az üzemmódban indítja el a mintavételezést. A RESUME üzemmódban is csak a Connect paranccsal lép át a CONNECTED üzemmódba, viszont a Connect parancs hatására nem áll meg a

folyamatban lévő mintavételezése. A CONNECTED üzemmódban minden parancsot fogadni képes [6], [10].

2.1.1.7 XCP parancsai

Az XCP protokoll parancsaival lehet csatlakozni, kapcsolatot bontani, le lehet kérdezni a slave aktuális állapotát, azonosítani lehet a slave csomópontját, a DAQ és CAL konfigurációt elmenteni a nemfelejtő memóriába, lehet kérni egy véletlen számot az azonosításhoz, visszaküldhet a seed alapján generált kulcsot, beállíthatja a cím mutatóját, adatokat tölthet fel és le a slave-től a master-be és így tovább [6], [10].

Ha a master eszköz egy olyan opcionális parancsot küld, amely a slave eszközben nincsen implementálva, akkor a slave eszköz az ERR_CMD_UNKNOWN hibaüzenetet küldi vissza, és nem történik az eszközben semmilyen módosítás. A hibaüzenet segítségével a master észlelni tudja, hogy az adott opcionális parancs nincs implementálva.

2.1.1.7.1 EV csomagok (EV) parancsai

A csomagok lehetnek esemény parancsok (EV). Az EV csomagok megvalósítása opcionális, azaz a vevői igények szerint van leghatározva, hogy a modulban mely parancsok vannak megvalósítva. Az EV parancsok aszinkron időben küldhetők a slave oldalról és nem kerülnek nyugtázásra. Mivel nincs nyugta a küldés után a master felől, ezért az átvitelük nem garantált [17].

EV_RESUME_MODE paranccsal jelzi a slave a master felé, hogy a RESUME módot elindította [17].

Az EV_CLEAR_DAQ, EV_STORE_DAQ és az EV_STORE_CAL parancsokkal a memóriába való írást és törlést lehet jelezni a master számára. Az EV_CLEAR_DAQ paranccsal lehetőség van jelezni a master számára, hogy a nemfelejtő memóriából a master által megadott DAQ konfigurációja törölve lett. Az EV_STORE_DAQ paranccsal a slave azt jelzi, hogy a master által küldött DAQ konfiguráció el lett mentve a nemfelejtő memóriába és az EV_STORE_CAL parancs azt jelzi, hogy a mastertől fogadott kalibrációs adat szintén el lett mentve a nemfelejtő memóriában [17].

Az EV_CMD_PENDING paranccsal a slave eszköz kérheti a master-t, hogy indítsa újra az időzítést [17].

Az EV_DAQ_OVERLOAD paranccsal jelzi a slave, hogy a DAQ processzor túl van terhelve [17].

EV_SESSION_TERMINATED jelzi, ha a slave befejezte a munkamenetet [17].

EV_TIME_SYNC parancs a slave eszköz időbélyegét továbbítja, amelyet a master kért egy előző paranccsal [17].

EV_STIM_TIMEOUT paranccsal lehet jelezni a master eszköznek, hogy a STIM időkorlátja lejárt [17].

EV_SLEEP paranccsal jelzi azt az eseményt a slave eszköz, hogy SLEEP módba kapcsolt. Az EV_WAKE_UP paranccsal pedig az SLEEP mód végét lehet jelezni a master számára [17].

Az EV_USER paranccsal jelezheti a slave eszköz, hogy a felhasználó által definiált esemény megtörtént [17].

Az EV_TRANSPORT egy transport layer specifikus eseményt jelez a master számára [17].

2.1.1.7.2 Szolgáltatást kérő csomagok (SERV) parancsai

A SERV csomagokban küldött parancsok is aszinkron időben küldhetők a master számára a slave eszközből. A SERV csomagok implementációja is opcionális a slave eszköz számára, viszont az összes parancs fogadását kötelező megvalósítani a master eszköznek. A master eszköz nem küld nyugtát a SERV üzenetekre, így annak átvitele nem garantált [17].

A SERV_RESET paranccsal a slave eszköz kérheti a vezérlőegység újraindítását a master eszköztől.

A SERV_TEXT paranccsal a slave küldhet egy egyszerű ASCII karakterekből álló üzenetet. A hosszabb szövegek egymást követő csomagokban továbbíthatók. A sorokat elválasztó karakter az LF vagy a CR/LF. A teljes szöveg végét a Null végződésű karakterláncot tartalmazó utolsó csomag jelzi.

2.1.1.7.3 Parancs csomagok (CMD) parancsai

Parancs csomagok (CMD) esetén vannak kötelezően implementálandó és opcionálisan implementálandó parancsok. CMD csomagok esetén csoportosítani lehet a kategóriák szerint a parancsokat:

- Standard parancsok
- Calibration parancsok (CAL)
- Page switching parancsok (PAG)
- Adatgyűjtő és adatstimulációs parancsok (DAQ)
- Nemfelejtő memória programozási parancsok (PGM)

A standard parancsok az alap működést biztosítják a slave eszköz és a master eszköz számára.

A CONNECT paranccsal egy folyamatos, pont-pont kapcsolatot hozhat létre a master egy slave eszközzel. Ha egy működő kapcsolat során érkezik a CONNECT parancs, akkor a slave eszköz figyelmen kívül hagyja. Amíg ez a parancs nem érkezik meg és nincsen fennáló kapcsolat a master és a slave eszközök között, addig a slave minden más parancsot figyelmen kívül hagy [17].

DISCONNECT paranccsal lehet a kapcsolatot bontani. A kapcsolat bontása után nincs XCP kommunikáció, és csak a CONNECTED parancs hatására folytatódhat a kommunikáció a master és slave között [17].

A GET_STATUS parancs segítségével tudja a master lekérdezni az aktuális slave státuszt. A parancs hatására a slave elküldi az összes aktuális állapotinformációját a master számára, magába foglalva az erőforrás védelem státuszát, a függőben lévő kérelmeket, valamint az adatgyűjtés és stimuláció általános állapotát [17].

A SYNCH parancsot a parancsok szinkronizálásához használják időtúllépés esetén. A SYNCH parancs mindig negatív választ fog adni az ERR_CMD_SYNCH hibakóddal, így a parancs válasza egyszerűbben megkülönböztethető a többi parancsra érkező választól a master számára [17].

A többi CMD csomagnak a standard parancsai opcionálisan megvalósíthatók. A GET_COMM_MODE_INFO parancs opcionális információt ad vissza a kommunikációs módokról [17].

A GET_ID parancsot a slave eszköz azonosítására és a munkamenet automatikus konfigurálásához használja a master [17].

A SET_REQUEST parancs használatával a master kérheti kalibrációs adat tárolást, a DAQ lista tárolást, vagy a DAQ konfiguráció törlését [17].

A GET_SEED parancs segítségével a master elkérheti a slave eszköz seedjét. Két mód áll hozzá rendelkezésre. Az első mód esetén a master elkéri a slave seedjét és a slave eszköz továbbítja a seed első részét és seed teljes hosszát is küldi. A másik mód esetén a master kéri a fennmaradó többi seed részt, ha az hosszabb volt, mint a csomag adathossz mérete [17].

Az UNLOCK parancs segítségével a master feloldhatja a slave eszköz védelmét a GET_SEED parancs első módja által kapott seed részből és a seed teljes hosszából számolt kulccsal. Ha a GET_SEED parancs implementálva van, akkor az UNLOCK parancsnak is kötelező az implementálása [17].

A SET_MTA parancs segítségével inicializálni lehet a memóriának a mutatóját az utána következő memória átviteli parancsoknak. A SET_MTA parancs a később definiált BUILD_CHECKSUM, UPLOAD, DOWNLOAD, DOWNLOAD_NEXT, DOWNLOAD_MAX, MODIFY_BITS, PROGRAM_CLEAR, PROGRAM, PROGRAM_NEXT és PROGRAM_MAX parancsok mindegyike előtt ki lesz küldve [17].

Az UPLOAD parancs segítségével a megadott hosszúságú, a SET_MTA paranccsal beállított aktuális MTA-tól kezdődő adatblokkot továbbítja a masternek a slave. Az MTA értéke utólagosan növekedni fog a megadott számú adatelem méretével [17].

A SHORT_UPLOAD parancs adattovábbításra szolgál. Egy előző parancsban megkapott hosszúságú és című adatot küld vissza a slave eszköz. A hibakezelése és a válaszának a felépítése az UPLOAD paranccsal megegyezik. A SHORT_UPLOAD parancs nem támogatja a blokkátvitelt, és nem használható blokkátviteli szekvencián belül [17].

A BUILD_CHECKSUM parancs hatására a slave visszaküldi a checksum értékét annak a memóriablokknak, amely meg van határozva az előtte megkapott MTA és memóriablokk méretéből [17].

A TRANSPORT_LAYER_CMD parancsot a szállítási réteg határozza meg. A szállítási réteg specifikus műveletek végrehajtására szolgál [17].

A USER_CMD parancs egy felhasználó által meghatározott parancs. Nem használható más szolgáltatások által végzett funkciók megvalósítására [17].

A CMD csomagok esetén öt darab kalibrációs parancs van meghatározva az XCP protokolljában. Ezek közül mindegyik opcionálisan megvalósítható, kivéve a DOWNLOAD parancsot.

A DOWNLOAD paranccsal adatokat lehet letölteni a master eszközről a slave eszközre. Amikor a slave fogadni akar egy DOWNLOAD parancsot, akkor ellenőrzi, hogy van-e elegendő erőforrás a letöltés kéréséhez. Ha nem rendelkezik a megfelelő méretű erőforrással, akkor hibaüzenetet küld a masternek és nem hajtja végre a letöltési kérelmeket. A memória címet egy előzőleg elküldött SET_MTA paranccsal lehet beállítani [17].

A DOWNLOAD_NEXT parancs a DOWNLOAD parancs egymás utáni adatelemeinek továbbítására szolgál. A parancs felépítése is pontosan megegyezik a DOWNLOAD parancséval. Tartalmazza a továbbítandó adatelemek fennmaradó számát és a slave eszköz felhasználja ezeket az információkat az elveszett csomagok a detektálására [17].

Ha a slave eszköz fogad egy DOWNLOAD_MAX parancsot, akkor ellenőriznie kell, hogy rendelkezik-e elegendő méretű erőforrással a teljes adatblokk fogadásához. Ha nincs elegendő hely a slave memóriájában, akkor hibaüzenetet küld a master-nek. A megfelelő memóriacímet a master egy SET_MTA paranccsal tudja megadni a slave-nek. Ez a parancs nem támogatja a blokkátviteli módot, és nem használható blokkátviteli szekvencián belül, így a DOWNLOAD_NEXT parancs nem használható a DOWNLOAD_MAX paranccsal [17].

SHORT_DOWNLOAD paranccsal a kapott méretű és megadott memória címen található adatblokkal tölthet le adatot a slave eszköz a master eszköztől. Egy előzőleg küldött SET_MTA paranccsal a master beállítja a memóriacímet, ahol a megfelelő adat található. A SHORT_DOWNLOAD parancs fogadása előtt a slave

ellenőrzi, hogy rendelkezésre áll-e a megfelelő méretű terület, ha nem, akkor nem lesz fogadva a parancs és hibaüzenetet küld válaszként. Ez a parancs sem használható blokk átviteli szekvencián belül [17].

A MODIFY_BITS paranccsal egy 32 bit méretű értékkel lehet AND vagy XOR műveleteket végrehajtani. Ez az érték egy memória címen van, amelyet egy SET_MTA paranccsal adtunk meg neki [17].

A CMD csomagok másik nagy csoportja a page switching parancsok.

Ha a slave eszköz támogatja a SET_CAL_PAGE és GET_CAL_PAGE parancsokat, akkor a page switching funkció is támogatva lesz.

A SET_CAL_PAGE parancs a hozzáférés módját állítja be a kalibrációs adatszegmensnek abban az esetben, ha a slave eszköz támogatja a page switching funkciót. Ezt a PAG flaggel lehet jelezni a resource availability maszkban. A kalibrációs adatok szegmensét és oldalait logikai értékekkel lehet megadni [17].

A GET_CAL_PAGE parancs küldésekor válaszként megadja a megadott hozzáférési módhoz és adatszegmenshez jelenleg aktivált kalibrációs adatoldalnak a logikai számát. A GET_CAL_PAGE segítségével a master megkaphatja a slave XCP és ECU hozzáférésének aktuális aktív adatoldalát. Abban az esetben, ha a SET_CAL_PAGE parancs implementálva van, akkor a GET_CAL_PAGE parancsnak kötelező az implementálása [17].

A GET_PAG_PROCESSOR_INFO parancs általános információkat ad vissza a master számára a paging funkcióról [17].

A GET_SEGMENT_INFO parancs információt nyújt az előre megadott szegmensről. Ha a hivatkozott szegmens nem érhető el, akkor hibaüzenettel tér vissza. A GET_SEGMENT_INFO parancs kétféleképpen kérhet adatot a slave eszköztől. Egyik esetben a válasz általános információkat tartalmaz a szegmensről. Másik esetben a GET_SEGMENT_INFO parancs előre jelzi, hogy milyen típusban kéri a kért leképezési információkat az előre meghatározott szegmensről. Ezt egy MAPPING_INDEX-szel lehet jelezni a slave felé [17].

A GET_PAGE_INFO parancs egy adott PAGE adatait adja vissza. Ha ez a meghatározott PAGE nem érhető el, akkor a slave eszköz hibaüzenettel válaszol [17].

A SET_SEGMENT_MODE parancs segítségével be lehet állítani egy szegmensnek a módját. Ezt a FREEZE flag állításával lehet megtenni, amely 1-es értékre állítva azt jelzi, hogy a szegmens nincs használatban. Ha a módosítani kívánt szegmens nem érhető el, akkor hibaüzenettel tér vissza a slave eszköz [17].

A GET_SEGMENT_MODE visszaadja az egyik szegmensnek az aktuális állapotát. Ha ez a keresett szegmens nem érhető el, akkor hibaüzenettel tér vissza a slave eszköz [17].

A COPY_CAL_PAGE paranccsal a master kényszerítheti a slave eszközt, hogy az egyik kalibrációs PAGE értékét átmásolja a másik PAGE értékére. Ezt a parancs csak akkor elérhető, ha egynél több kalibrációs PAGE van konfigurálva [17].

A CMD csomag negyedik nagy csoportja az adatgyűjtő és adatstimulációs parancsoknak a csoportja. A SET_DAQ_PTR parancs segítségével a master inicializálni tudja a DAQ listamutatót egy későbbi művelethez a WRITE_DAQ, vagy a READ_DAQ használatával. Ha a megadott lista nem áll rendelkezésre, hibaüzenettel tér vissza [17].

A WRITE_DAQ parancs küldésével a master egy ODT bejegyzést ír a DAQ listamutató által meghatározott slave DAQ listájába. A mutatót a SET_DAQ_PTR paranccsal lehet beállítani. A WRITE_DAQ parancsot csak konfigurálható DAQ listák elemeinek írásához lehet használni [17].

A SET_DAQ_LIST_MODE parancs használható az előre definiált és konfigurálható DAQ listáknak a beállításához. Ha a megadott lista nem áll rendelkezésre, akkor hibaüzenettel tér vissza. A DIRECTION flag segítségével DAQ, vagy STIM módba állítható a DAQ lista [17].

A START_STOP_DAQ_LIST parancs szintén használható az előre definiált és konfigurálható DAQ listákhoz. Ezt a parancsot a megadott DAQ lista szám elindításához, leállításához vagy előkészítéséhez használják [17].

A START_STOP_SYNCH parancs a kiválasztott DAQ listák továbbításának szinkronizált indításának, vagy leállításának végrehajtására szolgál. Miután a parancs sikeresen végre lett hajtva, a kiválasztás lekerül az küldött DAQ listáról [17].

A WRITE_DAQ_MULTIPLE paranccsal egymás után következő ODT bejegyzéseket lehet írni a DAQ listamutató által definiált DAQ listára. A listamutatót a SET_DAQ_PTR paranccsal kell beállítani a WRITE_DAQ_MULTIPLE parancs előtt. A WRITE_DAQ_MULTIPLE nem használható az ODT határokon túlnyúló írásra. A hibakezelés megegyezik a WRITE_DAQ parancséval, viszont hiba esetén nem lehet tudni, hogy melyik ODT bejegyzés okozta a hibát [17].

A READ_DAQ paranccsal ki lehet olvasni egy DAQ listamutató által meghatározott ODT bejegyzést. A READ_DAQ parancs az előre definiált és a konfigurálható DAQ listákban lévő elemeket is kiolvashatja [17].

A GET_DAQ_CLOCK paranccsal szinkronizálható a slave eszköz szabadon futó DAQ órája a master eszköz STIM órájával. Opcionálisan megvalósítható, ha a slave eszköz nem támogatja az időbélyeggel ellátott adatgyűjtést [17].

A GET_DAQ_PROCESSOR_INFO parancs válasza általános információkat tartalmaz a DAQ listákról [17].

A GET_DAQ_RESOLUTION_TIME parancs információt ad vissza a DAQ listák felbontásáról [17].

A GET_DAQ_LIST_MODE parancs visszaadja a megadott DAQ lista aktuális módjának információit. Ez a parancs használható a előre definiált és a konfigurálható DAQ listákhoz. Ha a DAQ lista nem létezik, akkor válaszként hibaüzenet érkezik [17].

A GET_DAQ_EVENT_INFO parancs információt ad vissza egy adott eseménycsatornáról. Ha a megadott eseménycsatorna nem érhető el, akkor hibaüzenetet tartalmaz a válasz [17].

A CLEAR_DAQ_LIST törli a megadott DAQ listát. Konfigurálható DAQ lista esetén az összes ODT bejegyzés visszaáll az eredeti értékekre. Az előre definiált és konfigurálható DAQ listák esetén a listán futó adatátvitel leáll és az összes DAQ listaállapot visszaáll [17].

A GET_DAQ_LIST_INFO parancs információt biztosít egy előre meghatározott DAQ listáról. Ha a megadott DAQ lista nem létezik, akkor hibaüzenettel tér vissza [17].

A FREE_DAQ parancs kitörli az összes DAQ listát és felszabadítja az összes dinamikusan kiosztott DAQ listát, ODT listát és ODT bejegyzést [17].

A dinamikus DAQ lista konfigurációs szekvencia indítása elején a masternek először mindig el kell küldenie egy FREE_DAQ parancsot [17].

Az ALLOC_DAQ paranccsal számos DAQ listát hozzá lehet rendelni a slave eszközhöz. Ha nincs elegendő memória a kért DAQ listák kiosztásához, akkor hibaüzenetet küld a válaszként a master eszköznek [17].

Az ALLOC_ODT paranccsal ODT-ket lehet hozzárendelni a megadott DAQ listákhoz. Ez a parancs csak konfigurálható DAQ listákhoz használható [17].

Az ALLOC_ODT_ENTRY paranccsal ODT bejegyzéseket lehet lefoglalni. Ha nincs elegendő memória a kért ODT bejegyzések lefoglalásához, akkor hibaüzenetet ad válaszként [17].

A CMD parancsok utolsó nagy csoportja a nemfelejtő memória programozási parancsai. A PROGRAM_START parancs a nemfelejtő memória programozási szekvencia kezdetének jelzésére szolgál. Ha a slave eszköz nincs olyan állapotban, amely lehetővé tenné a programozást, akkor hibaüzenettel tér vissza. Amíg a PROGRAM_START parancsot sikeresen végre nem hajtják, addig a memória programozási parancsok: PROGRAM_CLEAR, PROGRAM, PROGRAM_MAX vagy PROGRAM_NEXT nem engedélyezettek. A nemfelejtő memória programozási szekvencia végét egy PROGRAM_RESET parancs jelzi [17].

A PROGRAM_CLEAR paranccsal törölhető a nemfelejtő memória egy része az újra programozás előtt [17].

A PROGRAM paranccsal a slave belsejében lévő adatok programozását lehet végrehajtani. A hozzáférési módtól két különböző koncepció támogatott, az abszolút hozzáférési mód és a funkcionális hozzáférési mód [17].

A PROGRAM_RESET parancs jelzi a nemfelejtő memória programozási szekvencia végét. Minden esetben a slave eszköz leválasztott állapotba kerül. Ez a parancs felhasználható a slave eszköz visszaállításának kényszerítésére más célokra [17].

A GET_PGM_PROCESSOR_INFO parancs segítségével általános információkat lehet kapni a programozásról. Jelezheti, hogy törlés, vagy programozás módot használ, mely tömörítési állapotot képes kezelni, esetleg milyen titkosítást képes feldolgozni, vagy képes-e másfajta sorrendben érkező bemeneti adatot feldolgozni [17].

A GET_SECTOR_INFO parancs egy adott szektor adatait adja vissza. Ha a megadott szektor nem érhető el, akkor hibaüzenetet lesz a válasz [17].

A PROGRAM_PREPARE parancsot a kód letöltésének kezdetének jelzésére használják. Ez a parancs a nemfelejtő memória programozásának előfeltétele. Ebben a parancsban kell elküldeni a letölteni kívánt kód méretét. A parancs használata előtt a slave eszköznek meg kell győződnie arról, hogy a cél memória területe rendelkezésre áll-e, és olyan működési állapotban van, amely lehetővé teszi a kód letöltését [17].

A PROGRAM_FORMAT parancs a következő, megszakítás nélküli adatátvitel formátumát írja le. Az adatformátumot közvetlenül a programozási szekvencia elején kell beállítani és az aktuális szekvencia végéig érvényesnek kell lennie [17].

A PROGRAM_NEXT parancsot az egymás után következő adatbájtok továbbítására használják a PROGRAM parancs számára a blokkátviteli módban. Ha az adatelemek száma nem egyezik meg a várt értékkel, akkor hibaüzenetet ad vissza [17].

A PROGRAM_MAX parancs is adat továbbítására szolgál. Ez a parancs nem támogatja a blokkátvitelt, és nem használható blokkátviteli szekvencián belül [17].

A PROGRAM_VERIFY parancs ellenőrzi a kiküldött adatokat. Az ellenőrzési mód meghatározása projektspecifikus. A master megkapja az ellenőrzési módot a programozási folyamatirányításból, és továbbítja azt a slave-nek. Kétfajta ellenőrzési mód használható. Az egyik, amikor a master megkérheti a slave-t, hogy indítsa el a belső teszt rutinokat annak ellenőrzésére, hogy az új flash tartalma illeszkedik-e a flash többi részéhez. Csak a parancs eredménye számít. A másik esetben a master megadhatja a slave-nek, hogy ellenőrzési értéket fog küldeni [17].

2.2 AUTOSAR szabványcsalád

Az elmúlt 100 évben jelentősen megnőtt az egyre jobb és biztonságosabb autók iránti igény. Ennek következménye volt, hogy az autókba egyre komplexebb rendszereket kellett megvalósítani.

Ezekhez a rendszerekhez rengeteg alkatrészgyártó igazodott. Mivel minden márka különböző, saját fejlesztésű protokollt hozott létre a fejlesztésre, ezért különböző szabvány szerint kellett az alkatrészek szoftverét fejleszteni.

A szoftverfejlesztés időigényes feladat, ezért ezzel jelentősen megnőtt fejlesztési idő. A különböző szabványok szerint készített elektronikai alkatrészek pedig jelentősen megnehezítették a fejlesztést, hiszen mindegyik szabványra külön kellett a szoftvereket implementálni.

Ezen okok miatt is jött létre az AUTOSAR (**AUT**OMOTIVE **O**pen **S**ystem **AR**chitecture) konzorcium, melynek célja az integrált E/E architektúrák komplexitáskezelésének javítása és a modellezési- és integrációs technológia egységesítése volt.

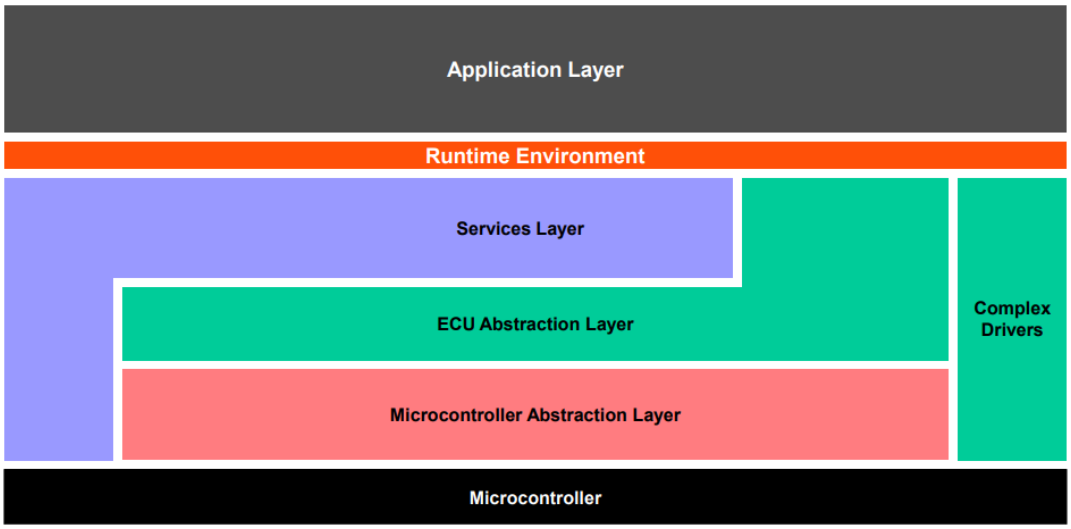
Az AUTOSAR a járműgyártók világméretű fejlesztési partnersége, beszállítók, szolgáltatók és az autóipari vállalatok elektronikai, félvezető- és szoftveripar résztvevői között. Az AUTOSAR szabványcsalád egy konzorcium, melyet hét nagy autóipari vállalat alapított 2003-ban. Azóta több mint száz cég tagja a konzorciumnak, főként Európából, de Észak-Amerikában is egyre jobban terjed a használata [1], [2].

A konzorcium tagjai az elektronikus vezérlőegységek (ECU) szoftver-architektúrájának egységesítésére törekednek azáltal, hogy követelményeket fogalmaznak meg a megvalósítandó szoftverrel szemben. Céljuk, hogy a modulok egyszerűen cserélhetők és újra felhasználhatók legyenek a megvalósítandó szoftverben, mind a beszállítók és a gyártók között [1].

2.2.1 AUTOSAR rétegzett architekúra

Az AUTOSAR architektúrájában három fő réteget különböztetnek meg: az Applikációs réteget (Application layer), az RTE-t (**R**un**T**ime **E**nvironment) és a BSW-t (**B**asic **S**oft**w**are). Ezt jól szemlélteti a 7. ábra. Az Applikációs réteg különféle alkalmazás-specifikus szoftverkomponenseket alkalmaz, amelyeket arra

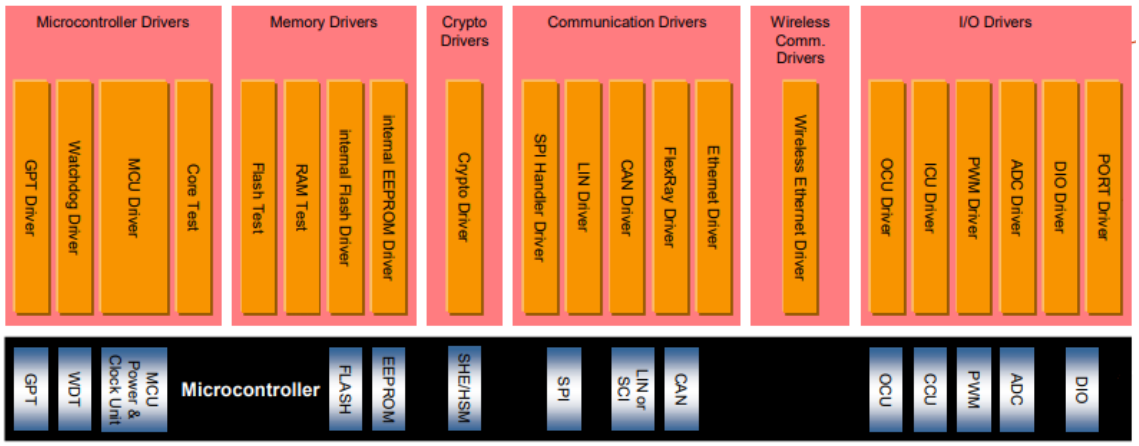
terveztek, hogy az adott felhasználási területhez tartozó feladatok meghatározott sorozatát végrehajtsák. Az RTE köztes szoftverként működik az Applikációs réteg és a BSW réteg között. Alapvetően ez a réteg kezeli a komponensek közötti kommunikációt, illetve elérhetővé teszi a BSW szolgáltatásait az Applikációs réteg számára [3].



7. ábra: Az AUTOSAR rétegzett architektúra felépítése [5]

A BSW réteg feladata, hogy a szoftverkomponensek számára alapvető szolgáltatásokat tudjon biztosítani az RTE rétegen keresztül. A BSW réteg három alrétegből áll össze, ez a szolgáltatás réteg, ECU absztrakciós réteg és a mikrokontroller absztrakciós rétege [15], [16].

A mikrokontroller absztrakciós réteg (Microcontroller Abstraction Layer) feladata az alacsonyszintű perifériák kezelése oly módon, hogy a felsőbb rétegek számára már hardverfüggetlen interfészt biztosítson. Ebben az alrétegben a mikrokontroller meghajtásáért felelős eszközmeghajtók találhatóak. A mikrokontroller absztrakciós réteg felépítése a 8. ábrán látható



8. ábra: A mikrokontroller absztrakciós réteg felépítése [5]

Négy csoportba lehet a meghajtókat bontani:

- Mikrokontroller meghajtók (Microcontroller Drivers)
- Memória meghajtók (Memory Drivers)
- Kommunikációs meghajtók (Communication Drivers)
- I/O periféria meghajtók (I/O Drivers)

A mikrokontroller meghajtók közé tartoznak a mikrokontroller alapvető egységeit kezelő modulok. A memória meghajtók közé a belső memória egységek meghajtó programjait lehet sorolni. A kommunikációs meghajtók csoportba az ECU-ban megvalósított kommunikációs periféria meghajtó moduljai tartoznak. Ezek a periféria meghajtó modulok az AUTOSAR-ban a következők lehetnek: CAN, FlexRay, Ethernet, Lin és SPI. Az I/O periféria meghajtókról beszélünk, ha a modulok kezelni képesek a bemenethez és kimenethez tartozó perifériákat [15], [16].

Az ECU absztrakciós réteg (ECU Abstraction Layer) feladata az ECU specifikus hardveres tulajdonságok elrejtése a felsőbb rétegek elől. Ebben a rétegben is csoportokra lehet bontani a modulokat működés szerint [15], [16].

Az ECU absztrakciós réteg egyik ilyen csoportja az I/O Hardver Absztrakció (I/O Hardware Abstraction). Ebbe a csoportba olyan modulok tartoznak, amelyek a bemeneti és kimeneti eszközöknek a megvalósítását elrejtik a felette elhelyezkedő rétegek elől. Az I/O Hardver Absztrakció csoportban elhelyezkedő modulok ECU

függők, az alkalmazás rétegben elhelyezkedő modulokkal közvetlenül tud kommunikálni [15], [16].

Az ECU absztrakciós réteg másik nagy csoportja a Kommunikációs Hardver Absztrakció (Communication Hardware Abstraction) csoport. Ebben a csoportban olyan modulok helyezkednek el, amelyek elfedik a felsőbb rétegek elől a kommunikációs hardver elemeket. Ebben a csoportban minden lehetséges hálózati protokollhoz (FlexRay, CAN, Ethernet, Lin) meg van valósítva egy interfész modul, valamint egy transceiver driver modul is. A transceiver driver modul feladata a buszokhoz tartozó meghajtóknak a kezelése. A transceiver driver modulokat az interfész modulon keresztül lehet elérni a felsőbb rétegekből. A Kommunikációs Hardver Absztrakció az alsóbb rétegek eszköz specifikus kommunikációját eltakarja [15], [16].

Az ECU absztrakciós réteg harmadik csoportja a Memória Hardver Absztrakció (Memory Hardware Abstraction) csoport. A csoportban található a nemfelejtő memória megvalósítása [15], [16].

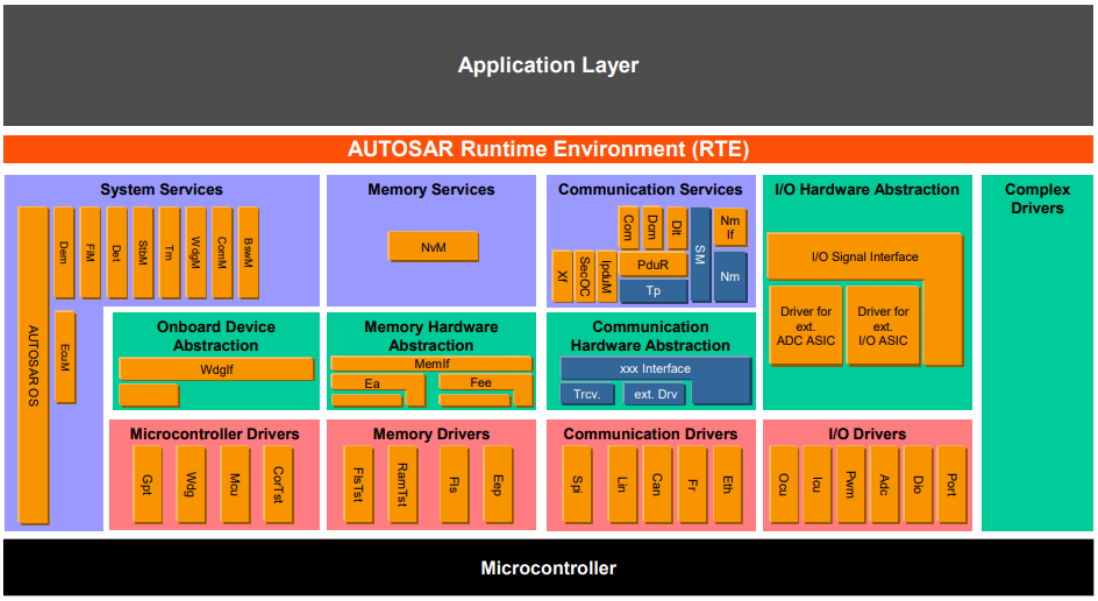
A ECU absztrakciós réteg negyedik csoportja az Belső Eszköz Absztrakció (Onboard Device Abstraction). Ebbe a csoportba az ECU legfontosabb absztrakciós moduljai tartoznak. A watchdog interfész és a külső watchdog meghajtók vannak definiálva a csoportban [15], [16].

A Szolgáltatás réteg 3 csoportból áll. Az egyik csoport a Kommunikációs szolgáltatások csoport. Ebben a csoportban a kommunikációval kapcsolatos magas szintű modulok helyezkednek el [15], [16].

A Szolgáltatás réteg egy másik csoportja a Memória Szolgáltatások (Memory Services). Ebbe a csoportba a nem felelejtő memóriakezelő (NvM) tartozik. A Memória Szolgáltatás csoportba ez az egyetlen modul tartozik. Az NvM feladata, hogy az alkalmazás rétegnek magas szintű működést biztosítson a memória interfész felhasználásával [15], [16].

A Szolgáltatás réteg utolsó csoportja a Rendszerszolgáltatások (System Services). A Rendszerszolgáltatások csoportba a platform műveleteket nyújtó modulok helyezkednek el. Ebben a csoportban helyezkedik el a beágyazott operációs rendszer (Operation System, OS), az ECU állapot menedzser (ECU State Manager, EcuM), a Kommunikációs menedzser (Communication Manager, ComM),

a Diagnosztikai esemény menedzser (Diagnostic Event Manager, DCM), a Funkció Engedélyezés Menedzser (Function Inhibition Manager, FIM), a Fejlesztési Hibakövető (Development Error Tracer, DET), a Diagnosztikai Napló és Követő (Diagnostic Log and Trace, DLT), a Szinkronizált Időalap Menedzsment (Synchronised Time Base Management, StdM) és a BSW üzemmód menedzsment (BSW mode Management, BswM). Az AUTOSAR BSW réteg részletes felbontása a 9. ábrán látható [15], [16].

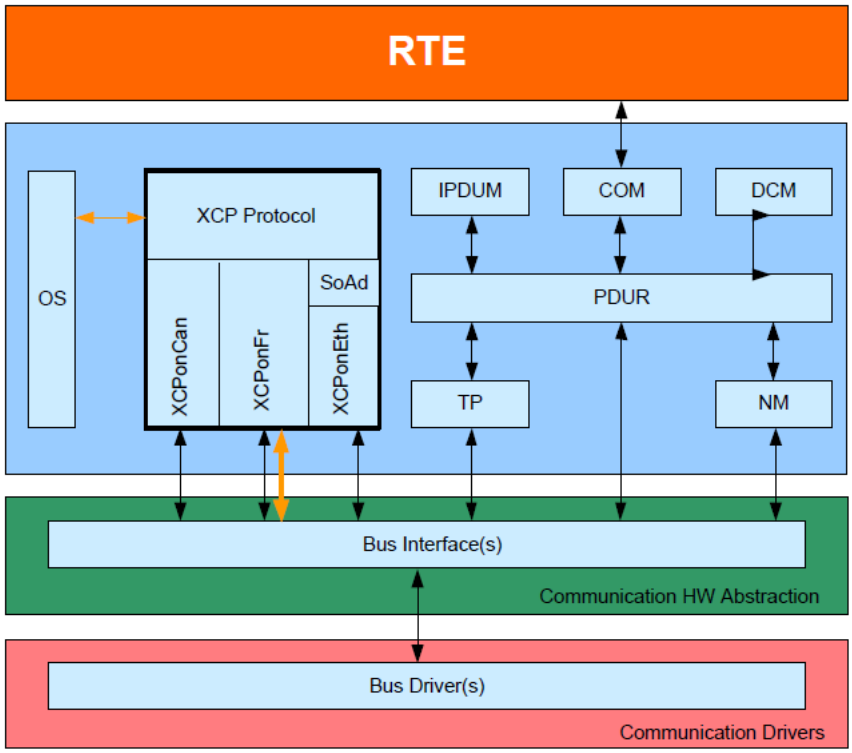


9. ábra: Az alapvető szoftvermodulok hozzárendelése az AUTOSAR BSW rétegekhez [5]

2.2.2 Kommunikációs stack felépítése

A munkám során a kommunikációs stack-ben található modulokkal kellett foglalkoznom. A kommunikációs stack több egymással kommunikáló modulból épül fel. A 9. ábrán található alrétegek közül a Szolgáltatási rétegből a Kommunikációs szolgáltatás csoport, az ECU Absztrakciós alrétegből a Kommunikációs Hardver Absztrakció és a Mikrokontroller Absztrakciós alrétegből a Kommunikációs meghajtó csoportok tartoznak hozzá.

Több interfész protokoll implementálható benne, így megkönnyítve a többi mikrokontrollerrel a kommunikációt. A kommunikációs stack felépítése a benne található XCP diagnosztikai modullal a 10. ábrán látható. Az ábrán a nyilak mutatják, hogy mely modulok kommunikálnak egymással.



10. ábra: AUTOSAR kommunikációs stack [6]

A kommunikációs stack-ben PDU-kkal (**P**rotocol **D**ata **U**nit) kommunikálnak egymás között a modulok. A PDU egy adott protokoll által értelmezhető adategység. A kommunikációs stack-ben három fajta PDU-t különböztetünk meg egymástól. Az első az L-PDU, azaz az adatkapcsolati réteghez tartozó PDU. Ezt a driverek és az interfészek közötti kommunikációra használják és a küldött, valamint a fogadott adatokat tartalmazza. A második az N-PDU, amely a hálózati réteghez tartozó PDU. Ez egy olyan PDU, amely az interfészek és a szállítási rétegben található modulok közötti kommunikációra használják. A harmadik PDU az I-PDU. Ez az interakciós réteghez tartozó PDU és a COM modul (**Com**munication), vagy a DCM modul (**D**iagnostic **C**ommunication **M**anager) által értelmezhető adatok vannak ebben a csomagban [3], [4].

A kommunikációs szolgáltatás alrétegben található AUTOSAR COM modul feladata, hogy az alkalmazás felől érkező jeleket PDU csomagokba illessze, a hálózaton továbbítsa, valamint az onnan érkező jeleket kicsomagolja és tovább küldje az alkalmazás felé. Ennek a modulnak a működése protokollfüggetlen.

A DCM modul szintén protokollfüggetlen. A diagnosztikai jogosultságok és a kommunikáció irányítása a feladata.

A ComM modul (**Com**munication **M**anager) a kommunikációhoz szükséges erőforrásokat irányítja. Ez a modul az Sm (**S**tate **M**anager) és az Nm (**N**etwork **M**anager) modulok által irányítja az ECU és a hálózat kommunikációs állapotát.

A PduR modul (**PDU R**outer) feladata a PDU csomagok irányítása a kommunikációs stack-ben található modulok között.

A CanTp (**CAN T**ransport **P**rotocol) és az FrTp (**F**lex**R**ay **T**ransport **P**rotocol) modulok feladata a szegmentálás és összeillesztés attól függően, hogy fogadja, vagy küldi a csomagokat. CAN protokoll esetén 8 bájtnál hosszabb méretű adatokat szegmentál, CAN-FD esetén előre meghatározott értéktől függ a szegmentálás és összeillesztés. CAN-FD protokoll esetén csak a 8, 12, 16, 20, 24, 32, 48 és 64 bájt hosszak helyesek. FlexRay protokoll esetén előre meghatározottnál nagyobb I-PDU üzenetek kerülnek szegmentálásra [8].

A CanSm (**CAN S**tate **M**anager) és az FrSm (**F**lex**R**ay **S**tate **M**anager) modulok kezelik a hálózat állapotát. Ehhez tartozik a hálózat indítása, leállítása, és az alvó állapot kezelése stb. A CanSm és az FrSm modulok felelősek a támogatott CAN és FlexRay vezérlők és adóvevők üzemmódjainak vezérléséért.

A Generic Network Management Interface a felette elhelyezkedő modulok számára nyújt interfészt a protokollspecifikus Network Management funkciókhoz.

A CanNm (**CAN N**etwork **M**anagement) és az FrNm (**F**lex**R**ay **N**etwork **M**anagement) modul feladata a CAN és FlexRay protokollhoz illeszkedő hálózati menedzsment funkcióknak az elvégzése [4].

Az XCP modul is a kommunikációs stackben helyezkedik el.

2.2.3 AUTOSAR XCP modul specifikációja

Az XCP modul egy kalibrációs protokoll és feladata a diagnosztika. Az AUTOSAR szabványcsalád az XCP protokollt az ASAM szabványok szerint használja, a hozzá tartozó követelményeket elfogadja és új követelményeket támaszt vele szemben, hogy az AUTOSAR BSW modulok közé illessze. Az AUTOSAR 4.0.3-as és 4.3.0-ás verziója is az ASAM 1.1-es verzió implementálását kéri [6].

Ahogy a 10. ábrán látható az XCP modul kapcsolatban van több interfésszel (CAN, FlexRay, Ethernet stb.), a SoAd modullal (**So**cket **Ad**aptor), az OS modullal (**O**peration **S**ystem) és a DET (**D**efault **E**rror **T**racer) modullal. Az AUTOSAR XCP

modul a buszspecifikus interfészek felett helyezkedik el. Ethernet esetén az AUTOSAR XCP modult a Socket Adapter modul felett kell elhelyezni [6].

Az AUTOSAR XCP modul az XCP üzenetek fogadására és továbbítására egyedi PDU-ID-t használ [6].

Az AUTOSAR XCP modul néhány ASAM által nyújtott XCP protokoll funkciót nem használ. Az AUTOSAR RTE réteg nem specifikál olyan API-kat, amivel az XCP közvetlenül kommunikálni tudna vele [6].

CAN szállítás réteg protokoll esetén az AUTOSAR XCP modulnak nem része a SET_DAQ_ID parancs [6].

Az AUTOSAR XCP modul az alábbi funkciókat valósítja meg:

- Xcp_Init
- Xcp_GetVersionInfo

Call-back értesítések:

- Xcp_<Lo>RxIndication
- Xcp_<Lo>TxConfirmation
- Xcp_<Lo>TriggerTransmit
- Xcp_SetTransmissionMode

Ütemezett funkció:

- Xcp_MainFunction

Az Xcp_Init függvény inicializálja az AUTOSAR XCP változóit. Az Xcp_Init függvény eltárolja a kapott konfigurációs címet, hogy a későbbi API-hívások hozzáférhessenek a konfigurációhoz [6].

Az Xcp_GetVersionInfo függvény visszaadja a modul verzióinformációit [6].

Az Xcp_<Lo>RxIndication függvény jelzi az XCP modulnak az alacsonyabb szintű kommunikációs interfész modul felől érkező PDU-t. A függvényt a busz interfészek, az Ethernet Socket Adapter, vagy a CDD (**C**omplex **D**evice **D**river) modul hívja meg. Az Xcp_<Lo>RxIndication függvényt az Xcp modul hívja meg egy interrupt esetén [6].

Az Xcp_<Lo>TxConfirmation függvény jelzi az XCP modulnak, ha az alsó réteg kommunikációs interfész modulja jelzi, hogy a buszon a PDU továbbítása sikeres volt, vagy sikertelen. Az Xcp_<Lo>TxConfirmation call-back funkciót a busz interfészek, az Ethernet Socket Adapter vagy a CDD hívja meg. Az Xcp_<Lo>TxConfirmation funkció meghívása akkor lehetséges, ha az Xcp modult helyesen inicializálták [6].

Az Xcp_<Lo>TriggerTransmit funkción belül a hívott XCP modul ellenőrzi, hogy a függvényhíváskor kapott hossz belefér-e a pufferméretbe. Ha belefér, akkor átmásolja az adatokat és frissíti az adatok hosszát. Ezt a függvényt is a busz interfészek, az Ethernet Socket Adapter, vagy a CDD modul hívja meg.

Az Xcp_SetTransmissionMode funkció a használatban lévő kommunikációs buszcsatorna TX működésének a be- és kikapcsolására szolgál [6].

Az ütemezett funkciót közvetlenül az Basic Software ütemezője hívja meg. Az Xcp_MainFunction függvénynek nincs visszatérési értéke és paramétere. Az Xcp_MainFunction az XCP modul ütemezett funkciója. Az XCP_MainFunction függvényt ciklikusan kell meghívni [6].

A többi modul által nyújtott szükséges interfészek a következők:

- A CanIf_Transmit funkcióval a PDU továbbítását lehet kérni.
- A Det_ReportError funkció szolgáltatást nyújt a fejlesztési hibák jelentésére.
- A FrIf_DisableLPdu funkció becsomagolja a FlexRay illesztőprogram Fr_DisableLPdu funkcióját. Letiltja az LPdu hardver erőforrását az átvitelhez, vagy a fogadáshoz. Az XCP modul ezeket az adatokat lementi.
- A FrIf_ReconfigLPdu funkció meghívja a FlexRay illesztőprogram Fr_ReconfigLPdu függvényét. Ezzel a funkcióval lehetséges az LPdu újrakonfigurálása. Az XCP modul tárolni tudja ezeket az LPdu-kat és a tárolt értékeket később aktiválni is tudja a FrIf_ReconfigLPdu funkció hívásával.
- A FrIf_Transmit funkcióval egy PDU továbbítását lehet kérni.

- A GetCounterValue funkció leolvassa a számláló aktuális értékét. Az OS modul függvénye. Az OS modulban található számláló értékéről kap információt az XCP modul. A számláló megadja, hogy mely ütés értékeket kell olvasni. Ezeket az értékeket az XCP arra használhatja, hogy az opcionális óraszinkronizáció funkció megvalósuljon.
- A GetElapsedValue funkcióval megkapja az ütések számát az aktuális ütés és egy korábban leolvasott ütés között. Szintén az OS modul függvénye. Az ütés az OS modul által használt számláló mentett értéke. Az XCP modul a használatával lementi az ütések közötti idő értékét és továbbküldi a master számára az óraszinkronizáció miatt.
- A SoAd_IfTransmit funkcióval szintén egy PDU továbbítását lehet kérni.

Ezeket az interfészeket implementácótól, vagy konfigurációtól függően opcionálisan használhatja az XCP modul. Az AUTOSAR XCP modul nem specifikál kötelezően megvalósítandó interfészt [6].

3 Tervezés

Az AUTOSAR konzorcium az évek során sok verziót kiadott a szabványokból, minden új verzióval kibővítve, javítva és optimalizálva azokat. A diplomaterv feladatom is ezt a témakört dolgozza fel. Az XCP 4.3.0-ás szoftververziót kellett megvalósítanom. A thyssenkruppban az előző 4.0.3-as szabvány verziója készen állt, így célszerű volt a kész verzióból kiindulni.

3.1 AUTOSAR XCP modul követelményei

A tervezés első fontos teendője az XCP modulhoz tartozó AUTOSAR és ASAM követelmények értelmezése volt. Az AUTOSAR szabványok közül a régebbi és az új szoftververziót is tanulmányoztam az XCP modulhoz. Az AUTOSAR és ASAM szabványcsaládban a leírt modulokkal szemben követelményeket támasztanak, amelyeknek a fejlesztés és megvalósítás során meg kell felelniük.

A szoftververziók tanulmányozásakor a különbségek vizsgálatát kellett ellenőrizni. Az összehasonlított követelményeket ellenőrizni hatékonyabban lehet egy táblázatba rendezve és könnyebben kereshetőek az újonnan felvett, vagy a módosult követelmények. Az XCP 4.0.3-as és a 4.3.0-ás AUTOSAR szoftververzió is az 1.1-es verziójú ASAM követelményekre támaszkodott.

A tervezés során készítettem egy excel táblázatot. Ebben az AUTOSAR szoftververziók különbségeit osztályoztam. Ennek a táblázatnak egy része a 11. ábrán tekinthető meg.

idx	4.0.3 req	4.3.0 req	functional change	note	functionality
3	Xcp502	SWS_Xcp_00502	Y	Minor changes	Removed Dem.h
4	Xcp503	-	N	removed	
9	Xcp509	-	N	removed	
20	Xcp761	SWS_Xcp_00761	N	same	
21	Xcp766	SWS_Xcp_00766	N	same	
22	Xcp712	SWS_Xcp_00712	N	same	
23	-	SWS_Xcp_00852	Y	Minor changes	Page switching
24	-	SWS_Xcp_00853	Y	Minor changes	generate the A2L IF_DATA sectior
25	-	SWS_Xcp_00854	Y	Minor changes	RESUME MODE
26	-	SWS_Xcp_00855	Y	Minor changes	flash programming (PGM)
27	-	SWS_Xcp_00856	Y	Minor changes	PROGRAM_RESET
28	-	SWS_Xcp_00859	Y	Minor changes	Avoid overwriting. Wait for the Xcp
56	Xcp760	-	N	removed	
57	Xcp759	-	N	removed	

11. ábra: Az AUTOSAR verziók követelményeinek változásait tartalmazó táblázat

Ennek a táblázatnak a segítségével rendszereztem, hogy mely követelmények nem változtak a verziófrissítés után és melyek azok, amelyeket módosítani kell. Az új verziószámmal újonnan felvett követelmények is bekerültek. Belátható, hogy azokkal a követelményekkel kell foglalkoznom, amelyek változtak, vagy újonnan vannak felvéve a verzió frissítés során. Az összes követelménypárhoz megjegyzéseket írtam. A megjegyzés mezőben tömören leírtam, hogy változott-e és ha változott, akkor nagy, esetleg kisebb változásról van szó. Azok a követelménypárok, amelyek nem egyeznek meg, esetleg a 4.3.0-ás verziójú új követelmények esetén a funkcionalitás mezőt is kitöltöttem. Ebben a funkcionalitás mezőbe a témakör csoportját írtam bele.

Meg kell vizsgálni azokat a 4.0.3-as verziójú követelményeket is, amelyek a 4.3.0-ás verzióból ki lettek véve. Ezeket a követelményeket aszerint kellett megvizsgálnom, hogy tartalmaznak-e olyan szűkítést, amelyeket nem kell végrehajtani a 4.3.0-ás verziójú modulban.

A tervezés szakaszban érdemes követelménylistát készíteni. A követelménylista egy specifikációja egy adott modulnak, és a specifikáció elfogadásával a benne található összes követelménynek meg kell felelni. A követelménylistának az összes követelményt tartalmaznia kell az XCP modullal szemben, így a listát felhasználva könnyebb követni, hogy mely követelményeket kell még megvalósítani. Abban az esetben, amikor egy követelmény tartalma az adott vezérlőegység működéséhez biztosan nem szükséges, akkor nem lesz megvalósítva. Ilyenkor az adott követelményt el lehet utasítani. Amely követelményeket nem utasítanak el a tervezés során, azokat kötelező implementálni.

Az összes követelményt az elfogadás, vagy elutasítás után osztályozni kellett implementálhatóság, valamint tesztelhetőség szempontjából. Egy követelmény akkor implementálható, ha annak a tartalma olyan strukturális, vagy működésbeli teendőket ír le, amelyeket a forráskódban meg lehet valósítani. Akkor nem implementálható, ha például megértést segítő, a fejlesztőnek szóló információkat tartalmaz a követelmény. Egy követelmény akkor tesztelhető, ha a követelmény implementálva van, és az implementált kód funkcionális teszttel ellenőrizhető.

Ha a követelményt nem fogadjuk el, esetleg nem implementálható, vagy nem tesztelhető, akkor minden esetben kötelező megjegyzést írni. Ezek a megjegyzések

segítenek a kód tesztelőjének, valamint azoknak akik a modullal később foglalkoznak, hogy a fejlesztő milyen indokkal tette az elutasítást.

A listában az összes követelményt a forráskódban tageljük, hogy a követelményeknek a fedettségét vizsgálni lehessen. A Doxygen a programozási nyelvekből képes online dokumentációt generálni a forrásfájlokból kiolvasva azokat. Ennek segítségével, ha a forráskód megfelelően van kommentezve, jó dokumentációt kapunk a majdani szoftverhez. A Doxygen a thssyenkrupp házon belül fejlesztett követelmény analizáló eszközének eredményére támaszkodva képes kiolvasni a követelménylista tartalmát és a forráskódban való implementálás után elérést biztosítani a követelménylista dokumentációjából a megvalósított forráskódig. A Doxygen az analizáló eszköz segítségével képes kiolvasni a tervezés során létrehozott követelménylistából a követelmények osztályozását és ezeket a követelményeket forráskódban megkeresni. A Doxygen a követelmény keresésekor a kommentek között kulcsszavakat keres. Az implementációhoz tartozó kulcsszó a @reqimpl{}, a teszteléshez tartozó kulcsszó a @reqtest{} volt. Ezeket minden esetben a megvalósítás és tesztelés helyére kell írni, így az online dokumentáció jól kereshető és értelmezhető lesz [14].

3.1.1 AUTOSAR XCP modul szoftvertervezés

A követelmények listába szedése után a következő lépés a két verzió közötti változások alapján megtervezni, hogy milyen módon kell azokat megvalósítani, esetleg módosítani.

Ehhez azt a módszert választottam, hogy a funkcionalitások szerint összegyűjtöttem a követelményeket. Ezáltal a funkciók működését egységesen lehet látni, így könnyítve az implementációt.

A követelményeket még lehet az alapján is csoportba sorolni, hogy a dinamikus kódhoz tartozik-e, esetleg a statikus kódhoz.

Statikus kód esetén a követelmény tartalmát nekem kell implementálni a modulban, amíg a dinamikus kód esetén ez a modulhoz megvalósított kódgenerátor által generálódik. A statikus kódra vonatkozó követelmények minden konfigurációban egységesek.

A dinamikus kódra vonatkozó követelmények esetén a kódgenerátorra vonatkozik a követelmény. Ezekben a követelményekben általában a konfigurációk specifikációja van megszabva. Ezek a dinamikus kódra vonatkozó követelmények konfigurációtól függnek.

Az AUTOSAR XCP modul 4.0.3 és 4.3.0 követelményei közötti különbségek az alábbiak:

- A DEM (Diagnostic Event Manager) modul típusainak a mellőzése.
- Az XcpDaqlist tároló paraméter multiplicitásainak módosítása.
- Megváltozott az XcpEventChannelTriggeredDaqListRef multiplicitása.
- Az ECU fejlesztési célokra szolgáló Flash programozás korlátozásának eltávolítása.
- Az Xcp_RxIndication argumentum PduInfoType * -ról const PduInfoType * -ra változott.
- Paraméterek hozzáadva az új verzióban az eseménycsatorna és az időbélyeg konfigurációjához.
- Lehetőség van az új verzióban az ODT memóriafogyasztásának kiszámítására (DAQ & STIM).
- A statikus és dinamikus ODT konfigurációs paramétereinek átalakítása.
- Lehetőség van a deaktiválni az átviteli képességek támogatását.

Ezeket az AUTOSAR verziók közötti különbségeket a fent leírtak alapján terveztem megvalósítani. Az ASAM szabványnak a követelményeit azért nem kellett belevennem a tervezésbe, mert mindkét AUTOSAR verzió ugyanazt az ASAM 1.1 verziót használja.

4 Implementáció

4.1 Előkészületek

Mielőtt a tényleges változtatások implementálásra kerülnek, előtte több feladatot is el kell végezni. Mivel a thyssenkrupp AUTOSAR BSW csoportjában build szerver segítségével történik a fejlesztés, ezért első körben az új verziójú XCP modulnak létre kellett hozni egy job-ot. Itt konfigurálni kellett, hogy mely modulokkal és pluginokkal forduljon, és az elérési helyét is meg kell adni az új XCP modulnak az SVN-en (**S**ub**V**ersio**N**). Ez a szerver képes együtt build-elni az összes fejlesztés alatt álló modult, valamint a build után statisztikákat lehet olvasni róla. Innen érhető el a későbbiekben a Doxygen online dokumentációja is.

Miután a build szerver job konfigurációja megtörtént, akkor a számára beállított SVN elérési helyére átmásoltam a 4.0.3-as XCP modulnak az implementációját, hogy az előző revíziók megmaradjanak benne.

Az SVN egy központosított verziókezelő rendszer. Az SVN-en könnyedén lehet keresni szoftververzióju k és revíziószámuk alapján. A fejlesztők a jelenlegi és a régi verziók kezelésére használják. Az SVN egy szerver, amely az összes eddigi verziót tartalmazza. Amit a fejlesztők a saját számítógépükön verzió kezelve használnak, az csak a lokális másolata a szerveren található fájloknak [13].

Mivel a 4.0.3 által a követelmények már tagelve vannak, ezért azokat a build szerver job indítása előtt el kell távolítani, mivel a módosított követelménylistával build-elve hibával térne vissza. A régi tagek eltávolítása után el lehet indítani a build szerver job-ot.

4.2 XCP 4.3.0 szoftververzió megvalósítás

Az előkészületek elvégzése után neki lehet látni a pluginok bővítésének. A definitions nevű pluginban már implementálva voltak az XCP 4.3.0-ás verziónak a konfigurációs paraméterei. Az XCP-vel szemben a thyssenkrupp is követelményeket támaszt. Ezeket a követelményeket azért támasztották a modullal szemben, mert a 4.0.3-as verziónak a fejlesztése során a vevői igényeket teljesítették, így érdemes implementálni a következő verzióban is.

4.2.1 Dinamikus kód változásainak megvalósítása

A dinamikus kód készítése kódgenerátorral zajlik. A dinamikus kód emiatt minden egyes bemeneti érték változáskor és a generátor indításakor újragenerálódik. A kódgenerátorban több konfigurációt fel lehet venni az XCP modul számára. Az XCP modul számára a tesztelés során készítenek konfigurációkat. Az AUTOSAR XCP 4.0.3-as verziójú modul számára 19 darab volt létrehozva. A 4.0.3-as verziójú konfigurációkat írtam át 4.3.0-ás verzióra. Mindegyik konfiguráció egyedileg beállított bemeneti paraméterekkel van ellátva.

A modul kódgenerátora és a teszt kódgenerátora is ezekből az adatokból készíti el a dinamikus kódot. Ezeket a bemeneti értékeket Container-ekbe lehet beírni és elmenteni. Az AUTOSAR Architect belső fejlesztésű szoftverrel lehet a generálást elvégezni. Az AUTOSAR Architect egy Eclipse alapú szoftver, amelyet az Eclipsen keresztül lehet elindítani debug módban.

A házon belül támasztott követelmények alapján kellett implementálni a Container-eket, amelyeket a kódgenerálásnál fogunk használni. Az itt megvalósított bemeneti mezőknek minden egyes konfigurációnál más paramétereket lehet megadni. Miután ezeket a Container-eket létrehoztuk, a konfigurációkban az új bemeneteknek értéket adva ellenőrizzük, hogy a generátor helyesen működik és létrehozza a dinamikus kódot a megadott paraméterek alapján.

A teszt generátor feladata a konfigurációkban megadott bemeneti paraméterek alapján az adatok generálása. Ugyanazokat a bemeneti paramétereket használja fel a generáláshoz mint a modul generátora, viszont a modulban bekövetkező változásoktól függetlenül jön létre a dinamikus kód a tesztelés számára. Ezzel kiküszöböltük azt, hogy a teszt alatt a modul által generált dinamikus kódot használva, a modul generátorának módosításai befolyással legyenek a tesztelésre.

Ezen változások után a build szerveren található modellezőeszközbe integráltam a modul generátorát és a teszt generátorát, hogy a build-ek során mindig újra generálódjon a dinamikus kód a konfigurációkban beállított adatok szerint.

A kódgenerátorok segítségével a szisztematikusan előállítható forráskódot hozzuk létre, ezzel a fejlesztőket megóvva a monoton munkától. A kódgenerátor implementációja során a generátorok megváltozott követelményeivel foglalkoztam,

amelyek a dinamikus kód létrehozásához szükségesek. Azok a követelmények, amelyek a paraméter multiplicitásának megváltozására vonatkoznak, azokat az AUTOSAR Architect szoftverrel lehetett módosítanom. Ebben a szoftverben a modulok számára elérhető Container-ek vannak felvéve. Ezekben a megváltozott követelményeknek a Container-eiben beállítottam a multiplicitást a követelmény szerint lefektetettek szerint. A multiplicitás azt jelenti, hogy egy konfigurációban az adott Container-nek mekkora a sokasága, azaz hányszor szerepelhet a konfigurációban. Ez bizonyos esetekben lehet akár 0 és 1 között, amikor opcionális funkciót lehet beállítani, viszont lehet olyan is, amikor kötelező létrehozni és azt jelöli a multiplicitás minimális 1 értéke.

A dinamikus kód generálásához egy másik lépés a java alapú kód módosítása. A generálás során az összes AUTOSAR Architectben létrehozott Container értékét lekérdezi és a kapott értékek szerint készíti el ezeket a dinamikus kód részeket. A java kódban a feladatom a generált funkciók paraméter változtatásainak lekövetése volt, valamint bizonyos változó típusoknak a megfelelő értékre való átírása. Az AUTOSAR sok követelményekben meghatározott típust ír le. Ezeket a típusokat módosítani kell, ha a verziók között a követelményben definiált típus felépítése megváltozik.

Ezeknek az elkészülte után a dinamikus kódnak a szükséges módosításait elvégeztem. Az AUTOSAR Architectben futtatva a konfigurációkat legenerálódik a dinamikus kód mindegyik konfigurációhoz. Ennek eredményeként a módosított kódgenerátor a konfigurációk segítségével a dinamikus kód már a 4.3.0-ás verzió szerint meghatározott követelményeknek megfelelően generál.

4.2.2 Statikus kód változásainak megvalósítása

A statikus kód esetén azokról a kódrészekről beszélünk, amelyek konfigurációnak a változásai során nem változnak. Ezeket az implementáció során nem kódgenerátorban készítik el, hanem a fejlesztő saját kezűleg valósítja meg. Ezeket a kódokat a cégnél használt az Eclipse fejlesztői környezetében valósítottam meg.

A statikus kód első lépéseként az XCP modul függvényeinek szignatúra változtatásait kellett elvégeznem a 4.3.0-ás szoftververzió követelményei alapján. A függvények több esetben is új bemeneti változót kaptak, esetleg a típusai változtak

meg. A változtatások a függvények visszatérési értékeire vonatkoznak. Ezek a visszatérési értékek adják meg a módosított értéket annak a függvények, amely meghívta. A módosult változók működésének elkészítése és a kódban a változó lekövetése volt a következő lépés. Ebben az esetben megkerestem azokat a kódrészeket, amelyek a verzió váltások során a módosított függvényeket meghívja. Itt a megváltozott visszatérési értékeknek az ellenőrzését kellett elkészítenem, esetlegesen a megfelelő továbbítását, ha azt az új változónak az értékét más funkció használja.

A szignatúraváltoztatások és a változtatások elkészülte után a következő lépés a 4.3.0-hoz tartozó követelményeknek a tagelése. Ezeket a követelmény tageket a statikus és dinamikus kódban is meg kell tenni, így nyomon lehet követni azokat a követelményeket is, amelyek esetlegesen kimaradnának. A követelmény tagelésnél a cél a 100 százalékos követelményfedés, ezzel biztosítva az AUTOSAR és ASAM szabványok követelményeinek betartását.

A követelmények tagelését a 3. fejezetben említett @reqimpl{} kulcsszóval kellett elvégezni. A zárójel közé a megfelelő követelmény sorszámot beírva és az implementációnak a megfelelő helyén tagelve biztosítható az adott követelménynek az elkészülte. Ezt a tesztelés során lehet majd ellenőrizni. A Doxygen a követelménylista és a tagelt követelmények segítségével ki tudja számítani, hogy a követelményfedésnek hány százaléka van elkészülve.

5 Tesztelés

Az utolsó lépés a megvalósított modul tesztelése. A tesztelés során a cél az, hogy a modulban megfigyelhető összes felmerülő hibát észlelni lehessen és ezeket ki lehessen javítani. Ahhoz, hogy a hibamentes modulról beszéljünk, a modulban implementált összes funkciót meg kell hajtani a teszt által. Ehhez modulteszt infrastruktúrát kellett készítenem, amivel a buildelés után ellenőrizni lehet különböző kódfedettség mérésekkel a helyes működést.

A modul teszteléséhez a tesztelőnek modulspecifikus konfigurációkra van szüksége. Ezek a konfigurációk az AUTOSAR Architect segítségével készítettem el. Az AUTOSAR Architect modulspecifikus konfigurációkat tárol egy arxml fájlban. A modul kiválasztott konfigurációjának előállításához a tesztelő használhatja az AUTOSAR Architect grafikus felhasználói felületét vagy a build környezet generáló parancsát.

A tesztek készítésekor az egyik szempont volt a letesztelt és a kódba beszúrt követelmények számának mérése. Minden tesztesetnél a Doxygen által használt @reqtest{} kulcsszót is be kellett szúrni, ahol az adott követelmény vizsgálata teljesült a teszt során. A cél az volt, hogy az összes olyan követelmény le legyen fedve tesztekkel, amelyek a követelménylistában tesztelhetőre voltak állítva. A @reqtest{} és @reqimpl{} kulcsszavak segítségével a Doxygen az analizáló eszköz használatával képes kiolvasni ezeket a modulból és a tesztből, és egy összegzést tud mutatni a követelmény fedettségéről.

5.1 Tesztelési elv

Az elkészült modult tesztelni kell, hogy megbizonyosodjunk arról, hogy az implementáció hibamentes. Tesztelés során a hibák jelenlétét tudjuk ellenőrizni. Gyakran a tesztkörnyezet mérete nagyobb, mint a modulé. Mivel csak a teszt során felmerült hibákat lehet javítani, ezáltal célszerű minél alaposabb tesztelést elvégezni a tesztelendő modulon.

5.1.1 Tesztelési metrikák

5.1.1.1 Követelmény alapú tesztelés

Követelmény alapú tesztelésnél a specifikációban található követelmények szempontjait vizsgálják meg. Alapértelmezés szerint minden modulhoz specifikáció tartozik, amely a modulra vonatkozó összes követelményt tartalmazza. A szabványcsalád által létrehozott dokumentumban meghatározott követelményeket meg kell valósítani és tesztelni kell. A tesztelés során az összes követelményt le kell fedni. Ha mindegyik implementált szoftver elemnek a működése a követelménynek megfelelő, akkor a tesztelt modul megfelel a specifikációban leírt működéshez.

5.1.1.2 Ekvivalencia osztály alapú tesztelés

Az ekvivalencia osztály alapú tesztelés során olyan teszt értékeket keresünk, amelyek azonosan járják be a vezérlési ágakat. Ekvivalencia osztály alapú tesztelést érdemes olyan esetekben használni, amikor nagyon sok bemeneti érték áll a tesztelő rendelkezésére. Ezeket az értékeket ekvivalencia osztályokba lehet rendezni. Az ekvivalencia osztályokat a hasonló témakörbe tartozó bemenetekből választja ki a tesztelő. Az ekvivalencia osztályokat úgy kell létrehozni, hogy azok elemei ne legyenek más ekvivalencia osztályban. Minden ekvivalencia osztályra igaz, hogy az osztályban szereplő véletlenszerűen választott bármelyik értékkel tesztelve feltehetően ugyanazt a vezérlési ágat járja be, mint az osztály többi tagja. Az ekvivalencia osztályokat a tesztelő választja ki a követelmények és a modul alapos megismerése után [18], [19].

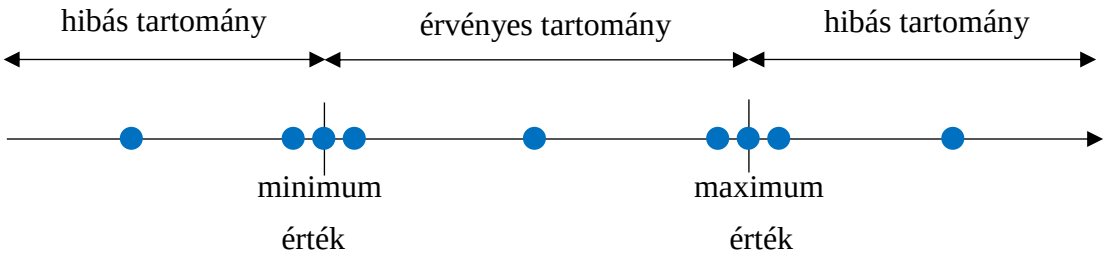
Például van egy döntésünk, ahol a 0 és 100 között beérkező számok közül csak a 30 és 40 közötti számokra ad vissza pozitív értéket, a többire negatív értékkel tér vissza. Ebben az esetben három ekvivalencia osztályt tudunk választani. Az egyik a pozitív kimenetekkel rendelkező 30 és 40 közti értékeket tartalmazó osztály, a másik az ennél az osztálynál kisebb 0-29-es számokat tartalmazó intervallum, valamint a 41-100 közötti intervallumot lehet ekvivalencia osztályba sorolni. Belátható, hogy megfelelő működés esetén a 0 és 29 közötti számok és a 41 és 100 közötti számok közül bármelyiket kiválasztva minden esetben negatív értékkel kell visszatérnie, míg a 30 és 40 közötti számok pozitív értéket adnak.

5.1.1.3 Határérték analízis tesztelés

A határérték analízis tesztelés (boundary value analysis) egy black box tesztelési mód. A black box tesztelés azt jelenti, hogy a tesztelő a specifikáció alapján végzi el a tesztesetek megvalósítását.

A határértékek az ekvivalencia osztályok határainál helyezkednek el. Az ekvivalencia osztály tartományának minimum és maximum értékénél is el kell végezni a határérték analízist. A fontos szempont alaposan tesztelni a tartományok határait.

A határérték analízis során ezeket a határértékeket, annál egyel kisebb és nagyobb értékeket kell tesztelni, valamint az ekvivalencia osztály tartományában egy közbenső elemet, amely kellően távol helyezkedik el az osztály határától és az ekvivalencia osztályon kívül egy-egy értéket. Ennek az értelmezésnek a segítségére a 12. ábra áll rendelkezésre. Az ábrán egy ekvivalencia osztály van vizsgálva, ezt nevezi érvényes tartománynak, amíg a többi ekvivalencia osztályt hibás tartománynak nevezi [18].



12. ábra: Határérték analízis tesztelendő pontjainak felvétele

Az ekvivalencia osztály alapú tesztelés példájára visszatérve ebben az esetben a 30 és 40 értékek vannak az ekvivalencia osztály határán, ezért ezeket mindenképpen ellenőrizni kell. Ezeknél egyel kisebb és nagyobb értékeket is vizsgálni kell, ezért a 29,31 és a 39,41 értékeket is tesztelni kell. Emellett az ekvivalencia osztályok olyan pontjait is tesztelni kell, amelyeket a tervezés során biztosan az osztályba sorolnánk, ezért a példában fel lehet venni a 14, 35, és a 70-es értékeket.

5.1.1.4 Code coverage

A code coverage (kódfedettség) egy százalékos eredményt ad a tesztelőnek. A fedettség ismeretében a tesztelő információt kap a teszt állapotáról. A nagyobb

érték azt jelenti, hogy jobban meghajtja a teszt a tesztelés alatt álló szoftvert és jobban garantálható a hibamentes működés is. A code coverage egy white box teszt. Coverage metrikákat a követelmény alapú tesztelés és határérték analízis után kiegészítésként használjuk. A code coverage megadja, hogy a modulnak mennyi sorát hajtották végre, hányszor hajtották végre kifejezéseit és az ágait. Minél jobban le van fedve a modul a teszt által, annál jobban növekszik a végrehajtási idő is.

5.1.1.4.1 Functional coverage

A functional coverage, azaz a funkcionális fedettség az implementált funkciók alapján vizsgálja a modult. A functional coverage szintén white box tesztelési módszer. A fedettség azt méri, hogy a modulteszt által lefedett modulnak mekkora arányban lettek tesztelve, azaz a funkcionalitást tesztelésének a mértéke [20].

A fedettség vizsgálata során azt ellenőrzi, hogy az implementált funkciók közül mennyi van letesztelve. A functional coverage százalékos értéket ad vissza:

$$Functional\ coverage = \frac{modulteszt\ által\ lefedett\ funkciók}{a\ modulban\ implementált\ összes\ funkció}\ [\%]$$

5.1.1.4.2 Statement coverage

A statement coverage, azaz az utasítás fedettség is white box tesztelési módszer. A statement coverage szintén egy százalékos értékkel tér vissza. A teszt által végrehajtott és az modulban implementált összes utasítás hányadosa.

$$Statement\ coverage = \frac{Modulteszt\ által\ végrehajtott\ utasítás}{Modulban\ szereplő\ összes\ utasítás}\ [\%]$$

Az állítások lefedettségén keresztül azonosíthatjuk a végrehajtott utasításokat és azokat, ahol a feltételek miatt a kód nem kerül végrehajtásra. A statement coverage előnyei [22]:

- Ellenőrzi az írott kód működését az alapján, hogy milyen működést várnak el tőle
- Mérni tudja az implementált kód minőségét

A statement coverage hátrányai [22]:

- A teszt során nem tudja tesztelni a hamis feltételeket.

- Nem számol be arról, hogy a ciklus eléri-e a végét.
- A fedettség nem ismeri a logikai operátorokat.

5.1.1.4.3 Branch coverage

A branch coverage szintén white box tesztelési módszer, jelentése ág fedettség. A branch coverage célja annak biztosítása, hogy minden ág minden döntési feltétele legalább egyszer teljesüljön. A tesztelés során kialakult ágak a döntések eredményei, ezért egyszerűen azt méri, hogy mely döntési eredményeket tesztelték. Ez magában foglalja az igaz és a hamis feltételeket nem úgy, mint a statement coverage esetén [21], [23].

Az branch coverage kiszámításához használt egyenlet:

$$Branch\ coverage = \frac{Modulteszt\ során\ végrehajtott\ ágak\ száma}{Modulban\ található\ összes\ ág}\ [\%]$$

A döntés lefedettségének előnyei [23]:

- Annak ellenőrzése, hogy a kódban lévő összes ágat elérte-e
- Annak biztosítása, hogy egyetlen ág sem vezet a program működésének hibáihoz
- Megszünteti a statement coverage tesztelésével felmerülő kapcsolatos problémákat

A döntés lefedettségének hátrányai [23]:

- Ez a mutató figyelmen kívül hagyja a logikai kifejezések elágazásait, amelyek a logikai operátorok miatt fordulnak elő.

5.1.1.4.4 MC/DC coverage

Az MC/DC coverage (modified condition / decision coverage), azaz a módosított feltétel / döntés fedettség az eddig felsorolt fedettségek közül a legbiztonságosabb. Az MC/DC coverage is egy white box teszt, amely szintén százalékos értéket ad vissza eredményül.

Ez a fedettség biztosítja, hogy a biztonságkritikus szoftverek megfelelő szinten legyenek tesztelve. Az MC/DC coverage alapja hasonló, mint a branch coverage, azzal a különbséggel, hogy a döntések összes feltételét le kell tesztelni a 100 százalékos fedettség eléréséhez [24].

Az MC/DC coverage pontos ismeretéhez ismerni kell az összes olyan feltételt és a hozzá tartozó ágakat. Ezek ismeretében a teszt során végrehajtott feltételeknek az arányából adódik a fedettségnek az értéke.

Például van egy if feltétel, amely igaz hamis eredményt ad. Ennél úgy kell tesztelni az igaz és a hamis ágat is, hogy az egyéb feltételeknél ne legyen változás azoknak az igazságértékeiben. 100 százalékos MC/DC fedettség esetén a condition / decision fedettség is 100 százalékot vesz fel.

5.1.1.5 Error guessing

Az error guessing, azaz a hiba sejtése esetén a modul tesztelőjének nagyfokú ismerettel kell rendelkeznie a modul specifikációjából. Az error guessing black box tesztelési módszer. Ez egy olyan tesztelési technika, amely nagyfokú tesztelési ismeretet követel meg a tesztelőtől. Ez egy tapasztalat-alapú tesztelési technika, ahol a tesztelemző tudását felhasználva megbecsli a modul problémás területeit. A tesztelőnek ismernie kell a tesztelendő modult, vagy hasonló működésű modulokat ahhoz, hogy meg tudja határozni, hogy hol fordulhatnak elő a leggyakrabban a hibák [25].

Az error guessing technika nem követ semmilyen konkrét szabályt. A tesztelő a sejtésnek megfelelően választ teszt input értékeket. Ezeket az inputokat a specifikáció, valamint a követelmények alapján határozza meg. A tesztelő próbálja meghatározni, hogy a modulban hol fordulhatnak elő a hibák. A error guessing technikán alapuló tesztesetek tervezéséhez az elemző a múlt tapasztalatait felhasználhatja a feltételek azonosítására. Ennek a technikának a tesztelési minősége elsősorban a tesztelők képességeitől függnek [25].

Ez a technika a tesztelés bármely szintjén és a gyakori hibák kipróbálására használható, például [25]:

- Nullával való osztás,
- Null pointer exception.

- Invalid paraméterek

Az error guessing céljai lehetnek [25]:

- Az Error guessing beépítve van a tesztelésbe. Ennek a technikának a fő célja a lehetséges hibák megtalálása.
- Az Error guessing kezdetekor meg kell kapni egy átfogó tesztkészletet, amelynek nincsenek kihagyott területei.
- Ez a technika kompenzálja a határérték analízis és az ekvivalencia osztály alapú technikák jellemző gyengeségeit. Ez a gyengeség lehet az, hogy az összes eseten végig kell menni és csak az esetek végrehajtása közben derül ki a hiba, amíg az error guessing tesztnél, jó választás esetén egyből meg lehet találni a hibát.

Habár az error guessing a tesztelés egyik kulcsfontosságú technikája, nem nyújt teljes körű lefedettséget a modul számára. Nem garantálható az sem, hogy a modul szoftvere elérte az elvárt minőségi referenciaértéket, ezért fontos más tesztelési technikákkal együtt alkalmazni. Ennek a tesztelési technikának jelentős előnye, hogy feltárja a hibákat azokon a területeken, amelyeket egyébként más tesztelési technika kihagyhat a tesztelése során [25].

5.1.2 Teszt elrendezés

A modul a hozzá tartozó specifikáció követelményeiből jön létre. Emiatt a tesztet is ezek a követelmények alapján kell létrehozni.

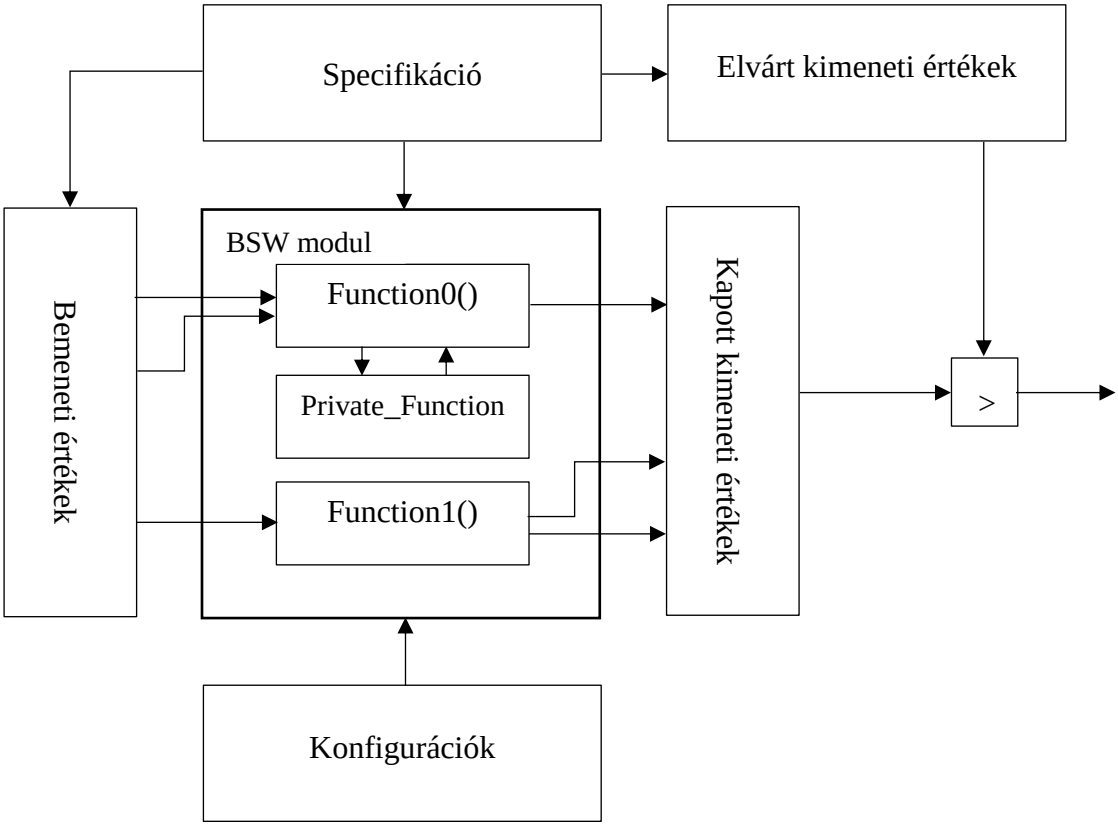
A tesztprojektben tesztcsomagok (test suites) vannak megvalósítva. Ezekben a tesztcsomagokban több teszteset (test case) szerepelhet, hiszen a eseteket lehet funkció szerint csoportosítani. A tesztesetek a modul specifikációjából készülnek. A tesztesetekben el kell helyezni speciális követelményekhez tartozó tageket, a futtatások során generálni lehet különböző dokumentációt a teszt aktuális állapotáról. Ilyen dokumentációt készít a Doxygen is, amelyhez cégen belül fejlesztett követelmény analizáló eszköz segítségével képes a tesztet vizsgálni és fedettséget számítani.

A tesztesetekből és a hozzájuk tartozó teszt scriptekből a futtatás után tesztnapló generálódik. A precompile értékek vizsgálatát, a code coverage vizsgálatot is ilyenkor hozza létre. A precompile értékek vizsgálata során

ellenőrizzük, hogy a teszt nem hagy ki feltételes kódelemet a precompile értékek beállításai miatt. Ehhez úgy kell a precompile értékeket megválaszani, hogy a preprocesszor direktívák minden ágába belemenjenek.

A tesztesetek és teszt scriptek segítségével lehet tesztjelentéseket készíteni. A tesztjelentésben szerepelnek a teszt adatai, valamint azok az értékek is, hogy a teszt mekkora arányban volt sikeres és milyen arányban bukott el. A tesztjelentés hibajelentés is egyben, hiszen a teszt a modul specifikációjából lett készítve, így ha bukik a teszt, akkor a modul eltérhet a specifikációban szereplő követelményektől. A tesztjelentések a modul tesztprojektjében generálódnak ki.

A tesztprojektben olyan vázat kell biztosítani a tesztelendő modulnak, amely helyettesíteni tudja a környezetét. Ez a környezet a vele kapcsolatban álló modulokból áll. A tesztelés során egy funkcionális modulteszt elvégzése a cél. A modulteszt felépítése a 13. ábrán látható.



13. ábra: Modulteszt elméleti blokkvázlat

A feladat során a modulra black box tesztelésként tekintünk, így a tesztprojekt létrehozása a specifikációban leírtakra hagyatkozik. Vannak olyan esetek, amikor a tesztprojekt előbb elkészül, mint maga a modul, így segítve a

modul folyamatos implementációjának a haladását. A specifikáció során olyan bemeneti értékeket lehet meghatározni, amelyek együttesen a modul összes funkcióját képesek meghajtani. Ezekhez a bemeneti értékekhez a specifikáció alapján elvárunk kimeneteket. Tehát a teszt alapján bemenet és elvárt kimenet párokat alkotunk.

A modullal kapcsolatban álló többi modulhoz stub fájlokat készítünk. Ezek a stub fájlok a tesztprojektben helyezkednek el és feladatuk a modullal kapcsolatban álló többi modulnak a szimulálása úgy, hogy a tesztelt modul által küldött összes paraméter értékét lementi. A stubok azok a függvények, amelyeket a tesztelt modul használ a környezetében elhelyezkedő többi modullal történő kommunikációjához. A teszt során a környező modulok függvényeit helyettesítjük. Ezekben a helyettesített függvényekben viszont nem a tényleges funkciónak az implementációját használjuk, hanem a hívás során kapott értékeket lehet lementeni a stub függvényekkel.

A tesztelés során több konfigurációt is létre lehet hozni. Ezekben a konfigurációkban olyan a teszthez tartozó bemeneti értékeket célszerű megvalósítani, melyek együttesen a legjobban meghajtják a tesztelendő modult.

Ezekkel a bemeneti értékekkel meghatjuk a tesztelt modult úgy, hogy a modul kimeneteit megfigyeljük a stubok segítségével. A modul által adott kimeneteket össze kell vetni a bemenetekhez tartozó elvárt kimenetekkel. Ha a kapott kimenet és az elvárt kimenet megegyezik, akkor a modul implementációja azokra az esetekre helyesnek tekinthetőek, viszont hiba esetén szükség van a modul vizsgálatára.

5.2 Funkcionális modulteszt

Az XCP modulnak a tesztelését a parancsok alapján bontottam fel, ezzel együtt a funkcióit is tesztelve. Az XCP modulnak küldött összes parancs segítségével megadható esetet tesztesetekbe rendeztem és parancsok küldésének és fogadásának minden lehetséges végkimenetelére teszteltem.

Minden teszteset futtatása során első lépésként inicializálni kell a modult. Ezek után a konfigurációktól kapott értékek segítségével elő kellett készíteni az aktuális tesztesetet, majd a parancs küldése után a kapott értékeket összevetettem az

elvárt kimeneti értékekkel, hogy minden követelmény által lefektetett szabály teljesül-e.

A teszt során 57 darab tesztesetet kellett létrehozni a megfelelő parancsok tesztelésére.

5.2.1 CONNECT parancs működése és vizsgálata

5.2.1.1 CONNECT parancs működése

A CONNECT parancsot a slave eszköz egy RxIndication függvény segítségével tudja fogadni a master eszköztől. A függvény fogadásakor a slave eszköz ellenőrzi, hogy milyen típusú a parancs és a megfelelő FIFO-ba belehelyezi a Pdu-t. A CONNECT parancs esetén a command FIFO-ba lett elmentve.

Miután el lett mentve a parancs és nem érkezik következő, a slave eszközben a MainFunction függvény indul el. Ebben a függvényben a slave eszköz ellenőrzi, hogy fogadott-e csomagot. Ezt a FIFO-k tartalmának vizsgálatával tette meg. Mivel a command FIFO-ban a CONNECT parancs helyezkedik el, ezért ennek a Pdu-ját kivette és a Pdu alapján meghatározta, hogy egy CONNECT parancs érkezett.

A CONNECT parancs feldolgozására a slave eszköz inicializálja a hiányzó FIFO-kat, a modul állapotát beállítja és a CONNECT parancsnak készít egy válaszüzenetet, amelyet az újonnan inicializált válasz FIFO-ba menti el.

A MainFunction függvényben a megfelelő busz protokollhoz tartozó main függvénybe lépett, ahol a válasz FIFO-t, a DAQ FIFO-t és az esemény FIFO-t ellenőrizte. A CONNECT válaszüzenetét megtalálva a busz protokollnak megfelelő header és tail elemekkel bővítette ki a csomagot, majd a Transmit függvénnyel válaszol a CONNECT üzenetre.

A Transmit üzenetre a slave modul egy TxConfirmation függvényt vár. Ha ez pozitív üzenet megérkezik, akkor a poolban elmentett Pdu értéket kitöröli, hiszen ilyenkor nem kell újra küldeni az üzenetet.

5.2.1.2 CONNECT parancs vizsgálata

A CONNECT parancs vizsgálatához össze kellett állítani egy CONNECT csomagot, amelyet elküldtünk a teszt során az XCP modulnak. A követelmények

alapján tudtam, hogy milyen válaszüzenet érkezik, így azokat az értékeket előre elmentettem.

Amikor a modul a Transmit függvényt hívja a teszt során a busz protokoll megfelelő stub Transmit függvényébe kerül. Itt a kapott értékeket összehasonlítottam az előre kigondolt válaszüzenettel. Ha ezek az értékek nem egyformák, akkor hibaüzenetet küld a teszt.

A teszt során a TxConfirmation parancsot küldtem a modulnak. Ezek alapján a CONNECT parancs fogadása és válaszüzenete meg lett vizsgálva.

5.2.2 Parancsok tesztelései

Mivel sok paranccsal rendelkezik az XCP protokoll, ezért nem egyesével parancsokként írom le a tesztelés fejezetben az összes folyamatát, hanem minden egyes tesztelési lehetőséget kifejtettem és írtam hozzá egy parancsot példának, ahol azt elvégeztem.

5.2.2.1 Alapműködés vizsgálata

Az ASAM szabvány által meghatározott összes parancsra készült teszteset. Minden parancs legelső tesztjének az alapműködésnek a tesztelését szántam. Ebben az esetben úgy kellett a parancsot meghívni, hogy a modul már a parancs fogadását el tudja végezni és a megfelelő visszatérési értékeket válaszként visszaküldeni, esetleg a memóriájában megtörténő változást végrehajtani. Például az ALLOC_DAQ, ALLOC_ODT_ENTRY és ALLOC_ODT esetén ez azt eredményezte, hogy felvette a megadott DAQ, ODT, vagy ODT bejegyzést az XCP modul. Ebben az esetben azt ellenőriztem, hogy a parancs végrehajtása hiba nélkül futott le és elmentette-e a csomag által küldött DAQ, ODT, vagy ODT bejegyzést. Az összes XCP követelmény által meghatározott parancsra el lehetett készíteni a hasonló ellenőrzést a helyes működés tesztelésére.

5.2.2.2 Hibás csomagméret vizsgálata

Olyan teszteseteket is el kellett végezem, amikor a parancsok nem megfelelő csomag méret szerint érkeznek meg. Az ASAM szabványban meg vannak határozva, hogy a parancsok mekkora méretű csomagban vannak elküldve. Ebben az esetben, a modulban azt vizsgáltam, hogyha az elvárt méretű csomag helyett eltérő csomagmérettel érkezik a parancs, akkor milyen folyamat történik az XCP

modulban. Ilyenkor a követelmények alapján azt vártam a modultól, hogy hibaüzenetet küld válaszként. Ez a hibaüzenet az ERR_CMD_SYNTAX nevű hibaüzenet volt.

5.2.2.3 Parancs vizsgálata zárolt erőforrás esetén

Bizonyos parancsokat oly módon is tesztelnem kellett, hogy a teszt által küldött parancsot az XCP modul nem tudja végrehajtani, mivel a parancsban kért erőforrások zárolva voltak. Ebben az esetben is hibaüzenetet vártam az XCP modultól. Ilyenkor a követelmények szerint az ERR_ACCESS_LOCKED hibaüzenetet vártam a slave eszköztől. Például az ALLOC_ODT parancs esetén a kért DAQ erőforrás zárolva volt, így az ERR_ACCESS_LOCKED hibaüzenetet megkaptam a modultól.

5.2.2.4 Memória túlcsordulás vizsgálata

Azok a parancsok esetén, amelyek a modul memóriájába írnak, tesztesetet kell készíteni arra az esetre is, amikor a memória megtelt adatokkal és egy új memória allokációs parancs érkezik az XCP modul számára. Ilyenkor a modulnak az ERR_MEMORY_OVERFLOW parancsot kell küldenie amellett, hogy a kért adatnak a mentése nem történik meg. Ezt az ALLOC_DAQ, ALLOC_ODT_ENTRY és ALLOC_ODT esetén is vizsgálnom kellett.

5.2.2.5 Nem aktív funkció hívásának vizsgálata

A konfiguráció módosításával bekapcsolható és kikapcsolható opcionális funkciókhoz tartozó parancsokat is tesztelni kell. Ilyenkor a parancs olyan funkció használatát kéri, amely az adott konfiguráció mellett ki van kapcsolva. Ebben az esetben a ERR_CMD_UNKNOWN parancsot vártam hibaüzenetként a modultól. A modul ilyenkor a kikapcsolt funkció miatt azt az alapértelmezett hibaüzenetet adja, mintha nem is lenne a modulban megvalósítva a funkció. Erre az esetre egy példa a READ_DAQ, GET_DAQ_CLOCK, GET_DAQ_EVENT_INFO és még más parancsok is, amelynek kapcsolható funkciói vannak a konfigurációk létrehozásakor.

5.2.2.6 Hibás tartományra hivatkozás vizsgálata

Olyan esetet is vizsgálni kell, amikor a tesztelendő parancs által hivatkozott érték kiesik a helyes tartományból. Ebben az esetben az XCP modul hibaválaszt

küld ERR_OUT_OF_RANGE hibaüzenettel. Például ilyen eset az, amikor a GET_DAQ_EVENT_INFO parancs elküldésekor nem a helyes eseménycsatorna szám van küldve. Ebben az esetben nem a 0 és a maximális csatornaszám értéke közötti értéket vette fel.

5.2.2.7 Hibás memóriaolvasás vizsgálata

Lehet olyan tesztesetet készíteni, ahol az adott parancsokat úgy hívjuk meg, hogy nincs engedélyezve a memóriaterület olvasása. A memóriacím helyét az előzőleg küldött SET_MTA érték határozza meg. Ebben az esetben ERR_ACCESS_DENIED hibaüzenetet küld az XCP modul válaszként. Ilyen tesztesetet például a DOWNLOAD_MAX parancsnál kellett implementálnom a tesztben.

5.2.2.8 Érvénytelen címzés vizsgálata

Felléphet olyan teszteset bizonyos parancsok tesztelésénél, amikor a paraméter címkiterjesztése érvénytelen. Ilyenkor ERR_OUT_OF_RANGE parancsot kell kapni az XCP modul válaszaként. Ezt az esetet a SHORT_UPLOAD és a SET_MTA parancsoknál hajtottam végre. Például a SHORT_UPLOAD esetén az érvényes tartomány a nulla és a külső memóriák száma között kellene lennie, viszont más értéket kap a teszt során.

5.2.2.9 Üres DAQ lista vizsgálata

START_STOP_DAQ_LIST parancs esetén felmerül olyan egyedi teszteset is, amikor az adott DAQ listában egy ODT lista sincs allokálva. Ilyenkor a master keres egy ODT listát és hibaüzenettel tér vissza.

5.2.2.10 Parancs újra küldésének vizsgálata

Olyan tesztesetet is készíteni kellett, amikor az XCP modulban a WRITE_DAQ_COMMAND parancs érkezik, miután a DAQ lista elindult. Ebben az esetben az ERR_DAQ_ACTIVE hibaüzenettel válaszolt a modul. Ezt a tesztesetet két parancsnál kellett megvalósítani, az egyik a WRITE_DAQ_MULTIPLE, a másik pedig a WRITE_DAQ parancs esetén volt.

5.2.2.11 DAQ listamutató vizsgálata

A WRITE_DAQ parancs esetén készíteni kellett olyan tesztesetet, ahol a WRITE_DAQ parancsot az XCP modul fogadta és feldolgozta. A parancs hatására a DAQ listamutatóját megnövelte a következő ODT bejegyzéshez ugyanazon ODT-n belül. Abban az esetben kell ellenőrzést végezni, amikor egy ODT-nek az utolsó ODT bejegyzésére ér és utána nincsen meghatározva új DAQ mutató érték. Ha az XCP slave ebben az esetben WRITE_DAQ parancsot kap, és a DAQ mutató nincs meghatározva, akkor az XCP ERR_ACCESS_LOCKED hibaüzenetet küld a parancs válaszaként.

A DOWNLOAD_MAX és a BUILD_CHECKSUM parancsok tesztelésénél olyan teszteseteket kellett létrehoznom, amelyekben ellenőrizni lehet, hogy az MTA értéke a parancs küldése után növekszik-e a parancsban kapott blokkmérettel.

5.2.2.12 Hibás szintaxisú csomag vizsgálata

Abban az esetben, ha érvénytelen szintaxisú parancsot küldünk a modul számára. Ilyen esetekben az XCP modul az ERR_CMD_SYNTAX hibaüzenettel válaszol a parancsra. Ezt az esetet például ellenőriztem a COPY_CAL_PAGE, GET_CAL_PAGE, GET_PAGE_INFO, GET_SEGMENT_INFO, GET_SEGMENT_MODE, SET_CAL_PAGE és SET_SEGMENT_MODE parancsoknál.

5.2.2.13 Hibás mód vizsgálata

Hibás mód beállítására is készítenem kellett teszteseteket. Ezekben az esetekben a parancs az ERR_MODE_NOT_VALID hibaüzenetet kapta válaszul. Például a CONNECT parancs tesztelésekor tesztelni kellett ilyen esetet is.

5.2.2.14 Hibás szegmens érték vizsgálata

Tesztelni kellett azokat az eseteket is, amikor érvénytelen volt a szegmensnek az értéke. Ebben az esetben a parancsokra a modul az ERR_SEGMENT_NOT_VALID hibaüzenetet küldte válaszul. Például a GET_SEGMENT_INFO parancs tesztelésénél ezt az esetet is figyelembe kellett vennem. Ilyenkor a parancs során hivatkozott szegmens értéke nem létezik a modulban, ezért küldi a hibaüzenetet.

5.2.2.15 Hibás oldal érték vizsgálata

Teszteseteket kellett létrehozni olyan esetekben is, amikor az egyik modul egy olyan oldalszámra hivatkozik, amely érvénytelen. Ekkor a válasznak kapott ERR_PAGE_NOT_VALID hibaüzenetet kell kapni. Például a GET_PAGE_INFO parancs esetén szükséges volt ilyen tesztesetet létrehoznom.

5.3 Hardverelrendezés

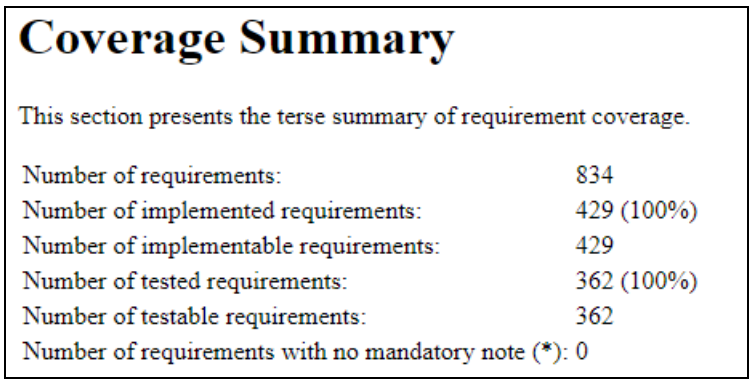
A mikrokontroller kalibrálását CANape szoftverrel kell végezni. A CANape elsődleges alkalmazási területe az elektronikus vezérlőegységek paraméterezésének kalibrálásának optimalizálása. A CANape segítségével a mérési folyamat során egyszerre lehet kalibrálni a paramétereket és rögzíteni a jeleket. A kommunikáció a CANape és az mikrokontrollerek között olyan protokollokon keresztül zajlik, mint az XCP, vagy mikrokontroller-specifikus interfészeken keresztül a VX1000 mérő- és kalibráló hardverrel [26].

A cél platformon való tesztelés során a mikrokontrollerre feltöltött 4.3.0-ás XCP modul vizsgálatát lehet elvégezni. A hardveres tesztelés azért fontos, mert ilyenkor valós környezetben lehet tesztelni a működését az AUTOSAR követelményei által megvalósított moduloknak. A memória és a parancsok kezelését lehet ellenőrizni.

Az ESD védelem miatt megfelelő ESD védelemmel ellátott munkalapra kell elhelyezni az ECU-t. A mérés során a PC-n futó CANape szoftverrel megfigyelhetjük az XCP modul működését ECU-n. A mikrokontrollert egy kiválasztott buszon keresztül tudjuk elérni a mérésre szolgáló PC-vel. Ilyen busz a CAN, FlexRay, Lin és Ethernet busz is lehet.

6 Eredmények

A diplomamunka során elkészültem az XCP modul 4.3.0-ás verziónak az implementálásával és a tesztelésével. A modul és a teszt is a követelmények szerint készült el és az ehhez szükséges tageket elhelyeztem a kódban. Az implementálható és a tesztelhető követelmények fedettségei is elérik a 100 százalékot, erről a 14. ábrán láthatjuk az eredményeket. Látható, hogy 834 darab követelmény volt meghatározva az XCP modul számára. Ezek közül 429 követelmény volt implementálhatónak és 362 darab tesztelhetőnek választva.



Coverage Summary

This section presents the terse summary of requirement coverage.

Number of requirements:	834
Number of implemented requirements:	429 (100%)
Number of implementable requirements:	429
Number of tested requirements:	362 (100%)
Number of testable requirements:	362
Number of requirements with no mandatory note (*):	0

14. ábra: A kódfedettség eredményei

A tesztelés során minden funkcióra és parancsra sikerült tesztet létrehozni. A tesztprojekt futtatása után az összes teszteset hiba nélkül zárult, így elmondható, hogy a kódfedettség mérése 100 százalékos eredménnyel zárult. A branch fedettség vizsgálata is 100 százalékos értéket ér el.

Ezek a metrikáknak értékei mellett elmondható, hogy a modul a teszt alapján hibamentesnek tekinthető, így elkészült a modulnak a tesztelése is.

7 Összefoglalás

A diplomatervezés során alapos ismereteket szereztem egy autóipari eszközöket fejlesztő vállalat fejlesztésének a menetéről, valamint a tesztelésről. Az XCP protokoll működését megértettem és elsajátítottam a hozzá tartozó ASAM, valamint az AUTOSAR szabványcsaládokat.

A kezdetben rendelkezésre álló 4.0.3-as verziójú XCP implementáció miatt célszerű volt annak a továbbfejlesztése a 4.3.0-ás AUTOSAR verzióra. Feladatom első eleme a 4.0.3-as és 4.3.0-ás XCP követelmények összevetése volt, majd az ezekből készült táblázat alapján az implementáció megtervezése.

Az implementáció során megismertem a kódgenerátorok és a szükséges szoftverek működését. A dinamikus és a statikus kódokat megvalósítottam a 4.3.0-ás verzióban. Ehhez az Eclipse fejlesztői környezet és az AUTOSAR Architect belső fejlesztésű szoftvereket vettem segítségül.

Az implementáció elkészülte után a tesztelés volt a következő lépés. Ennek során az ASAM és AUTOSAR követelményei által meghatározott parancsok által végeztem el a tesztelést. A tesztelés elkészült és a követelmények mindegyike sikeresen lefutott bukás nélkül. A követelmény és branch fedettség értékei elérik a 100 százalékos értéket.

A cél eszközön való tesztelést megismertem, viszont a fennálló vírushelyzet miatt azt elvégezni nem tudtam. Ezt a helyzetet a konzulenssel és a témavezetővel egyeztettem. A tesztelés nagyobb része a funkcionális modulteszt. A cél platformon való tesztelés nem ad olyan részletes információt, mint a funkcionális modulteszt. A cél platformon való tesztelés csak akkor jelez, ha a modulban olyan hiba merült fel, amely a funkcionális modulteszt nem szűrt ki. A cél platform teszt elméleti részének megismerésével a gyakorlati hasznát is megismertem.

8 Köszönetnyilvánítás

A diplomamunkámat a thyssenkrupp Components Technology Hungary Kft.-nél végeztem. Köszönetet szeretnék mondani konzulenseimnek, Sujbert Lászlónak és Szikszay Lászlónak, akikhez bármikor fordulhattam segítségért és tanácsért. Emellett köszönöm a thyssenkrupp AUTOSAR BSW csapatának is a segítséget, amit tőlük kaptam.

Irodalomjegyzék

[1] AUTOSAR: *AUTOSAR introduction,* https://www.autosar.org/fileadmin/ABOUT/AUTOSAR_EXP_Introduction.pdf (2020.05.14.)

[2] *AUTOSAR honlap*, https://www.autosar.org (2020.05.14.)

[3] Embitel: *Decoding the „Component Concept" of the Application Layer in AUTOSAR,* https://www.embitel.com/blog/embedded-blog/decoding-the-component-concept-of-the-application-layer-in-autosar (2020.05.04.)

[4] BME MIT szabadon választható tárgy: *AUTOSAR alapú autóipari szoftverrendszerek,* https://www.mit.bme.hu/oktatas/targyak/vimiav15/jegyzet (2020.05.17.)

[5] AUTOSAR: *Layered Software Architecture*, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf (2020.05.04.)

[6] AUTOSAR: *Specification of Module XCP*, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_XCP.pdf (2020.05.05.)

[7] ASAM: *Our vision*, https://www.asam.net/about-asam/our-vision/ (2020.05.09.)

[8] AUTOSAR: Specification of CAN Transport Layer, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_CANTransportLayer.pdf (2020.05.10.)

[9] ASAM: *ASAM MCD-1 XCP*, https://www.asam.net/standards/detail/mcd-1-xcp/wiki/ (2020.05.09.)

[10] Dunaújvárosi Főiskola, Informatikai Intézet: *Autóipari beágyazott rendszerek,* https://docplayer.hu/25114281-Autoipari-beagyazott-rendszerek-ccp-es-xcp.html (2020.05.06.)

[11] Eszenyi Andrea: *Diagnosztika és nyomkövetés*, https://adoc.tips/diagnosztika-es-nyomkvetes.html (2020.05.06.)

[12] Association for Standardisation of Automation and Measuring Systems: *XCP Version 1.1, Part 1 – Overview* (2008.03.31)

[13] Perforce: *What is SVN (Subversion)?,* https://www.perforce.com/blog/vcs/what-svn (2020.05.17.)

[14] Doxygen *honlap*, http://www.doxygen.nl/ (2020.05.17.)

[15] *Autoipari beágyazott rendszerek – AUTOSAR*, https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0035_autoipari_beagyazott_rendszerek/pdf/aut_4.pdfattachment (2020.11.25.)

[16] AUTOMOTIVE BASICS: *AUTOSAR BASICS*, https://automotivetechis.wordpress.com/autosar-concepts/ (2020.11.28.)

[17] Association for Standardisation of Automation and Measuring Systems: *XCP Version 1.1, Part 2 – Protocol Layer Specification* (2008.03.31)

[18] Horváth László: *Szoftvertervezés a gyakotlatban*, https://people.inf.elte.hu/holaci/Diploma/Szoftvertesztel%20a%20gyakorlatban.pdf (2020.11.10.)

[19] Ficsor Lajos, Kovács László, Kusper Gábor, Krizsán Zoltán: *Szoftvertesztelés*, https://www.tankonyvtar.hu/hu/tartalom/tamop425/0046_szoftvertesztelles/ch04s03.html (2020.11.12.)

[20] *Functional Coverage Part I*, http://www.asic-world.com/systemverilog/coverage1.html (2020.11.12)

[21] C*ode Coverage Tutorial: Branch, Statement, Decision, FSM*, https://www.guru99.com/code-coverage.html (2020.11.12.)

[22] *What is Statement coverage? Advantages and disadvantages*, http://tryqa.com/what-is-statement-coverage-advantages-and-disadvantages/ (2020.11.23)

[23] *What is Branch Coverage or Decision Coverage? Its advantages and disadvantages*, http://tryqa.com/what-is-decision-coverage-its-advantages-and-disadvantages/ (2020.11.12.)

[24] *Modified Condition / Decision Coverage (MC/DC)*, https://www.froglogic.com/coco/modified-condition-decision-coverage-mcdc/ (2020.11.15.)

[25] *What is Error Guessing Technique?*, https://www.softwaretestinghelp.com/error-guessing-technique/ (2020.11.15.)

[26] CANape: *ECU Calibration with CANape*, https://www.vector.com/int/en/products/products-a-z/software/canape/ (2020.11.25.)