Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Krucsai Gergő

2023

Szegedi Tudományegyetem
Informatikai Intézet

Általános szoftverfrissítő kliensalkalmazás autóipari beágyazott rendszerekhez

Szakdolgozat

Készítette:
Krucsai Gergő
üzemmérnök-informatikus
szakos hallgató

Témavezető:
Dr. Alexin Zoltán
egyetemi docens

Szeged
2023

Feladatkiírás

SZAKDOLGOZAT TÉMATERV

Név: **Krucsai Gergő**, üzemmérnök-informatikus BSc., nappali tagozat

Neptun kód: J9RMAR

Külső konzulens: **Molnár Tamás (villamosmérnök_MSc)**

Meghirdető: **Robert Bosch Kft.**

Belső konzulens: **Dr. Alexin Zoltán, egyetemi adjunktus**

Tanszék: **TTIK, Szoftverfejlesztés Tanszék**

ÁLTALÁNOS SZOFTVERFRISSÍTŐ KLIENSALKALMAZÁS AUTÓIPARI BEÁGYAZOTT RENDSZEREKHEZ

GENERIC FLASHTOOL FOR AUTOMOTIVE EMBEDDED SYSTEMS

A munkám lényege hogy, C++ nyelven megvalósításra kerüljön egy olyan kliensalkalmazás melynek a célja, hogy az autókban található elektronikus irányító egységeket (ECU) CAN csatornán keresztül tudjuk frissíteni.
Elsődleges fejlesztési irányelv, hogy sebességre legyen optimalizálva. A sebességet egy szimulált környezeten keresztül tudjuk mérni, ennek segítségével szeretném megközelíteni a CAN-FD buszon a lehetséges maximális frissítési sebességet. A legfontosabb szempont pedig, hogy kliensalkalmazás képes legyen példaprojekten szoftvert frissíteni, illetve könnyen adaptálható legyen új projektekhez. A programkódnak modulárisnak kell lennie, mellyel lehetővé válik más autóipari buszok későbbi használata.
A feladatmegoldás tartalmazni fogja a fordítókörnyezetet, mely a make eszközzel kerül megvalósításra Ennek az eszköznek a tulajdonsága, hogy automatikusan lefordítja a futtatható programkódokat és könyvtárakat a forráskódból.
Az időtervben a fejlesztési folyamatokat nagyobb mérföldkövekre bontom, ez alapján követem a munka folyamatát és fogalmazom meg a nagyobb követelményeket, amelyekkel biztosítom az alkalmazás határidő előtti befejezését. Az architektúrális tervezéshez PlantUML-t, a programkód teszteléséhez a Google Test nevű unit teszt könyvtárat fogom használni. A tesztekhez tartozó dokumentációban a tesztekhez felhasznált eszközök, valamint az elvárt és a kimeneti érték összehasonlítása fog szerepelni. A szimulációs tesztelési folyamatot egy szimulált targettel és egy példaprojekten keresztül tesztelem, ezzel reprezentálva az elvárt működést. A feladat megoldáshoz végül készítek egy fejlesztői és egy felhasználói dokumentációt.

Ütemterv:

1. hónap *(feb 1-feb 28.)*: Követelmények megfogalmazása
2. hónap *(márc 1-márc 31.)*: Időterv készítése
3. hónap *(ápr 1-ápr 30.)*: Irodalomkutatás
4. hónap *(máj 1-máj 31.)*: Architekturális tervezés és unit tesztek megírása a feladathoz
5. hónap *(jún 1-jún 30.)*: Implementáció

Szünet (júl.1-aug.31.)

6. hónap *(szept. 1-szept. 30.)*: Implementáció
7. hónap *(okt. 1-okt. 31.)*: Szimulációs tesztelés példaprojekten és szimulált targettel
8. hónap *(nov. 1-nov. 30.)*: Tesztelés dokumentáció/Bugfix
9. hónap *(dec. 1-dec. 31.)*: Végső dokumentáció létrehozása
10. hónap *(jan. 1-jan. 31.)*: Bemutatás

Szeged, 2023. február 24.

Tartalmi összefoglaló

- ***A téma megnevezése:***

A szakdolgozatom témája egy szoftverfrissítő kliensalkalmazás létrehozása C++ nyelvben, amely kommunikálva a Vector gyártó CAN interfészének könyvtárával.

- ***A megadott feladat megfogalmazása:***

A feladat elkészítéséhez szükség lesz egy CAN kommunikációs eszközre és egy diagnosztikai szoftverre.

A cél egy olyan program megírása, amely képes az ISO szabványoknak eleget téve fájlból adatot beolvasni CAN csatornán üzenetet küldeni és fogadni.

- ***A megoldási mód:***

A feladatnak megfelelően Vector XL Library külső könyvtár és a Visual Studio Code használata. Az Editor-on belüli különböző bővítményeket használata.

- ***Alkalmazott eszközök, módszerek:***

A szakdolgozatom során kifejlesztett parancssori alkalmazás C++ 20 szabvány szerint készült.[10] A forráskód fordításához a MINGW 8.2.0 fordítóprogramot használtam.

A hardveres konfigurációhoz a Vector VN1610 eszközét használtam fel. Az eredmények diagnosztizálására CANoe szoftvert használtam.

- ***Elért eredmények:***

Létrehoztam egy parancssorból indítható futtatható állományt, amely szoftverfrissítési protokollokat részben teljesíti. Mindezek mellett rendelkezik egy demonstrációs frissítési szekvenciával, ami egy hardvert szimuláló szerver alkalmazással kommunikál.

- ***Kulcsszavak:***

C++, CAN interfész, Vector Hardver, Szoftver frissítés

Tartalomjegyzék

BEVEZETÉS

A szakdolgozatom a járműipar egyik kulcsfontosságú területét, a beágyazott rendszerek szoftverfrissítéseinek alapos vizsgálatára összpontosít, különös tekintettel a Vector technológiák és a Controller Area Network (CAN) protokoll alkalmazásaira. Napjainkban az autóipar egyre inkább a szoftveralapú megoldások felé mozdul el, amely megköveteli a járművek elektronikus vezérlőegységeinek (ECU-k) folyamatos frissítését és karbantartását. A szoftverfrissítések célja, hogy javítsák a járművek teljesítményét, biztonságát és kényelmét, valamint, hogy új funkciókkal lássák el őket. Ezen frissítések jelentősége különösen azokban az esetekben növekszik, ahol a szoftverek közvetlen hatással vannak a jármű biztonsági rendszereire. Az autóiparban elterjedt CAN protokoll egy olyan hálózati kommunikációs rendszer, amely lehetővé teszi az ECU-k közötti adatcsere gyors és megbízható megvalósítását.

1. TERÜLETI ÁTTEKINTÉS

Ebben a szekcióban áttekintést nyújtok a CAN hálózatok alapvető jellemzőin, szerepüket a járműkommunikációban, és különös tekintettel a szoftverfrissítési folyamatokra. Kiemelten foglalkozom a Vector hardverekkel és szoftverekkel, mint például a VN1610 interfész és a CANoe diagnosztikai szoftver, amelyek nélkülözhetetlen eszközök a járműelektronikai fejlesztésben.

1.1. CAN

A CAN, vagyis a Controller Area Network, egy járművekben használt hálózati protokoll. A szoftverfrissítések tekintetében a CAN hálózat lehetővé teszi az autógyártók számára, hogy távolról frissítsék a járművek fedélzeti szoftverét. Ez különösen fontos az olyan modern járműveknél, amelyek nagymértékben támaszkodnak a szoftverre a funkcióik működéséhez. A CAN hálózatok üzenet alapú kommunikációt használnak. A hálózat minden csomópontja képes üzeneteket küldeni és fogadni.[9] A kommunikáció középpontjában az úgynevezett üzenetazonosítók (message identifiers) állnak, amelyek meghatározzák az üzenet prioritását is minél alacsonyabb az azonosító értéke, annál magasabb a prioritás. A CAN protokollt széles körben alkalmazzák olyan kritikus rendszerekben, ahol az adatok pontos és időben történő továbbítása elengedhetetlen. Ilyen területek például a járműipar, ahol a motorvezérléstől kezdve az ablakemelőkön át a légzsákrendszerekig sok különböző egység kommunikál egymással a CAN buszon keresztül.

A CAN busz egy kétvezetékes fél duplex rendszer, ami azt jelenti, hogy az adatátvitel egy időben csak egy irányban történhet, de a rendszer képes az adatok továbbítására és fogadására is. A CAN busz lehetővé teszi, hogy több tucat eszköz közvetlenül kommunikáljon egymással anélkül, hogy szükség lenne egy központi számítógépre. A CAN protokollban többszintű hibafelismerési mechanizmus van jelen. Amennyiben egy eszköz hibát észlel, megpróbálja megszakítani az üzenet továbbítását, jelezve ezzel a hibát a többi csomópont felé.

1.2. Vector hardver

A Vector egy vezető cég az autóipari mérnöki és elektronikai eszközök területén, amely különféle hardver- és szoftvermegoldásokat kínál a járműkommunikációhoz és diagnosztikához. A Vector eszközei, mint például a VN1610, elsősorban a járművekben használt hálózatok, mint a CAN protokoll vizsgálatára és diagnosztizálására szolgálnak. A VN1610 egy kompakt, nagy teljesítményű hálózati interfész, amely kifejezetten a járműipari alkalmazások számára lett tervezve. Két CAN csatornával rendelkezik, amelyek lehetővé teszik a felhasználó számára, hogy egyszerre két különböző CAN hálózathoz csatlakozzon. A VN1610 továbbá támogatja a nagy sebességű adatátvitelt és kompatibilis a legtöbb járműipari kommunikációs protokollal. Az eszköz szoftveres integrációja is kiemelkedő, hiszen kompatibilis a Vector szoftvereszközeivel, mint például a CANoe diagnosztikai eszközzel, amely lehetővé teszik a felhasználó számára, hogy részletesen elemezze és szimulálja a hálózati kommunikációt.

1.3. Szoftver frissítés

Ez a folyamat azt jelenti, hogy adatokat továbbítunk a célrendszerre, például egy ECU-ra (Electirc Control Unit). Célja kizárólag az adatátvitel. A folyamat lényegében egy sorozat üzenetküldésből áll, melyek az UDS protokoll által definiált keretek között történnek. A CAN busz biztosítja az adatcsomagok megbízható szállítását a forrástól a célhardverig.

1.4. CANoe

A CANoe egy szintén Vector Informatik GmbH által fejlesztett szoftvereszköz, amelyet széles körben használnak az autóiparban a járművek elektronikus vezérlőegységeinek (ECU-k) fejlesztéséhez, hibakereséséhez és teszteléséhez. Ez a szoftver különösen hasznos a CAN hálózatok, valamint más járműipari kommunikációs protokollok teszteléséhez és elemzéséhez. A CANoe számos funkciót kínál, beleértve a hálózati kommunikáció szimulációját és tesztelését, amely segít a mérnököknek a hálózati problémák diagnosztizálásában és a rendszer viselkedésének elemzésében. A CANoe intuitív, grafikus felhasználói felületet kínál, amely megkönnyíti a hálózati forgalom megfigyelését, az üzenetek elemzését és a teszteredmények vizualizálását.

1.5. UDS

Az UDS üzenetek (Unified Diagnostic Services) az On-Board Diagnostics szabvány részét képezik, amelyet járművek diagnosztizálására és konfigurálására használnak. Ezek az üzenetek számos szolgáltatást foglalnak magukban, mint például a hibakódok olvasását, az ECU-k programozását, tesztelését és egyéb karbantartási funkciókat.[5] Az UDS protokoll szabványosítja a járművek és a diagnosztikai eszközök közötti kommunikációt, lehetővé téve a különféle gyártók és eszközök közötti kompatibilitást. A szakdolgozatban leírt rendszer az UDS üzeneteket használja az elektronikus vezérlőegységek frissítési folyamatában. Az UDS protokoll szolgáltatásainak implementálása lehetővé teszi a rendszer számára, hogy a különböző ECU-kkal hatékonyan kommunikáljon, frissítéseket hajtson végre, és diagnosztikai adatokat gyűjtsön.

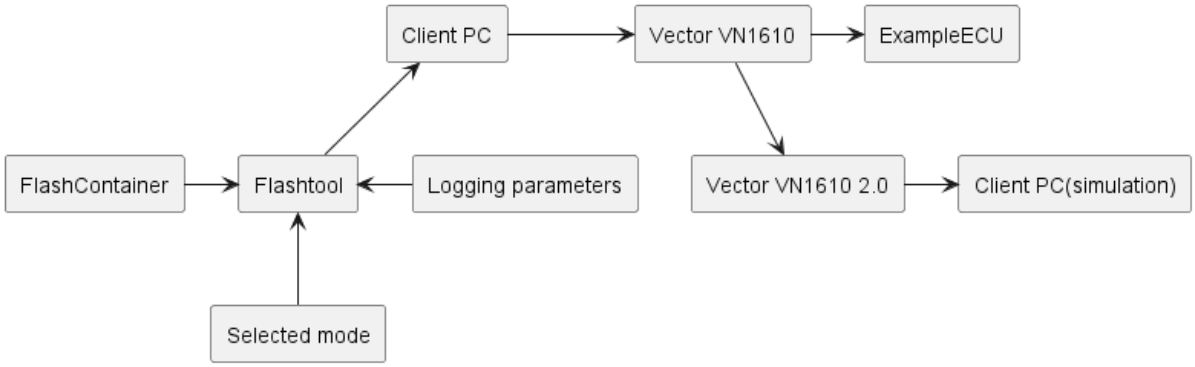
2. TERVEZÉSI FÁZIS

A tervezési szakaszban végzett munka alapvető jelentőséggel bír az egész projekt sikere szempontjából. Ennek a fázisnak a célja, hogy meghatározzam a projekt egyes fejlesztési fázisait, architektúráját, a komponensek közötti kapcsolatokat, valamint az adatfolyamatot és kommunikációt. Az elkészített rendszervázlat alapján létrehozott kommunikációs kapcsolat lehetővé teszi, hogy a számítógép és a hardvereszközök közötti adatáramlás zökkenőmentes és hatékony legyen.

2.1. Időterv készítése

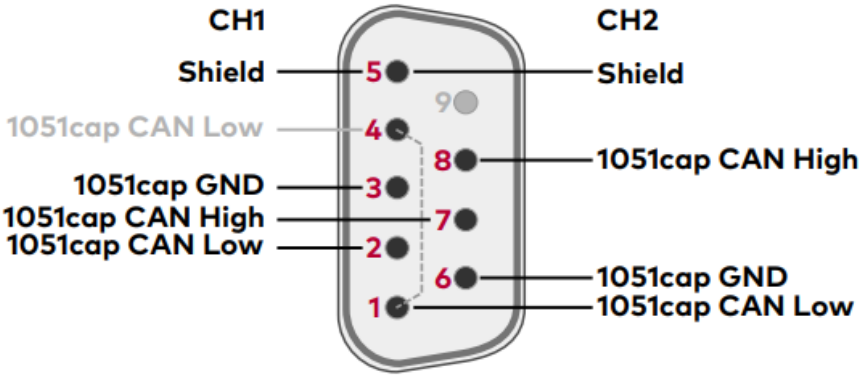
A projekt időtervét strukturált módon, mérföldkövekre bontva készítettem el, melyek a Rational Team Concert (RTC) szoftver segítségével kerültek definiálásra. Az RTC eszközt használva kialakítottam a fejlesztési folyamatot jellemző feature-öket, amelyeket Program Incrementekbe (PI-okba) rendeztem. Ezzel a módszerrel a projekt egyes szakaszainak előrehaladását, a fejlesztés dinamikáját biztosítottam. Minden egyes PI szakaszban részletesen meghatároztam a fejlesztési folyamat során végrehajtandó user story-kat. Ezek a story-k az adott PI szakasz fejlesztési célkitűzéseit és feladatait foglalják magukban, elősegítve a projekt átláthatóságát, és egyértelmű struktúrát adva a fejlesztés során.

2.2. Rendszer felépítése



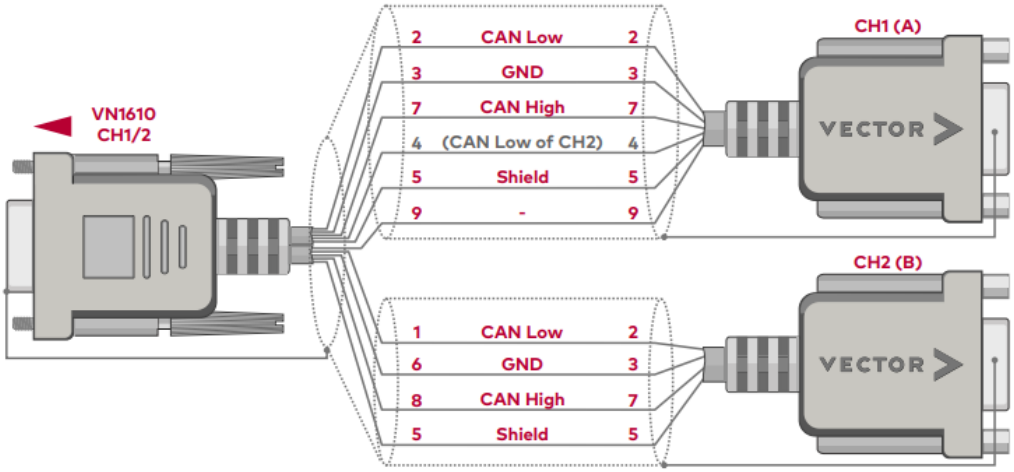
2.2. a ábra Rendszer diagram

A projekt korai hardverfejlesztési szakaszában elkészítettem egy rendszervázlatot, amely részletesen bemutatja a számítógép és a használt hardvereszközök közötti kapcsolatot. A 2.2. a ábrán látható, hogy a számítógép közvetlenül csatlakozik a Vector VN1610 interfészhez. A VN1610 ebben az esetben úgy van konfigurálva, hogy párhuzamosan van csatlakoztatva a fogadó pinjei és a küldő pinje, ezzel lehetővé téve számára, hogy egyszerre funkcionáljon adatküldő és adatfogadó eszközként. Ez a konfiguráció kritikus fontosságú a kommunikációs folyamat során, mivel lehetővé teszi a számítógépen keresztüli adatok átfogó vizsgálatát, legyen szó adatküldésről vagy adatfogadásról.



2.2. b ábra: VN1610 pin elrendezés [8]

A 2.2. b ábrán az egyes csatorna a bal oldalon, a kettes csatorna pedig jobb oldalon helyezkedik el. A feladatom az volt, hogy párhuzamosan összekössem a két csatorna megfelelő pinjeit. A CAN High vonalak esetében az egyes csatorna CAN High pinjét, a kettes csatorna CAN High pinjévelkötöttem össze, hasonlóképpen a CAN Low vonalak esetében is, az egyes csatorna CAN Low pinjét a kettes csatorna CAN Low pinjével. Ezen kapcsolatok kialakításához Y-kábelre volt szükségem, hogy külön-külön tudjam kezelni mindkét csatornát.



2.2. c ábra: Y kábel pin elrendezés

A 2.2. c ábárn a VN1610 eszköz pinjeinek eloszlását láthatjuk az Y-kábel esetében. A CAN Low kapcsolathoz a 2-es pint kellett összekötni az 1-es pinnel, ami az A kimenet 2-es pinjének és a B kimenet szintén 2-es pinjének felel meg. A CAN High esetében pedig a 7-es pint kellett összekötni a 8-as pinnel, ami az A kimenet 7-es pinjének és a B kimenet 7-es pinjének felelt meg. Így a két csatorna mind a High, mind a Low vonalon összeköttetésre került.

Ez az összekötési metódus lehetővé teszi, hogy egy eszközzel, a VN1610-nel egyidejűleg két CAN csatornát monitorozzunk és szimuláljunk, így növelve a tesztelési és diagnosztikai műveletek hatékonyságát és rugalmasságát.

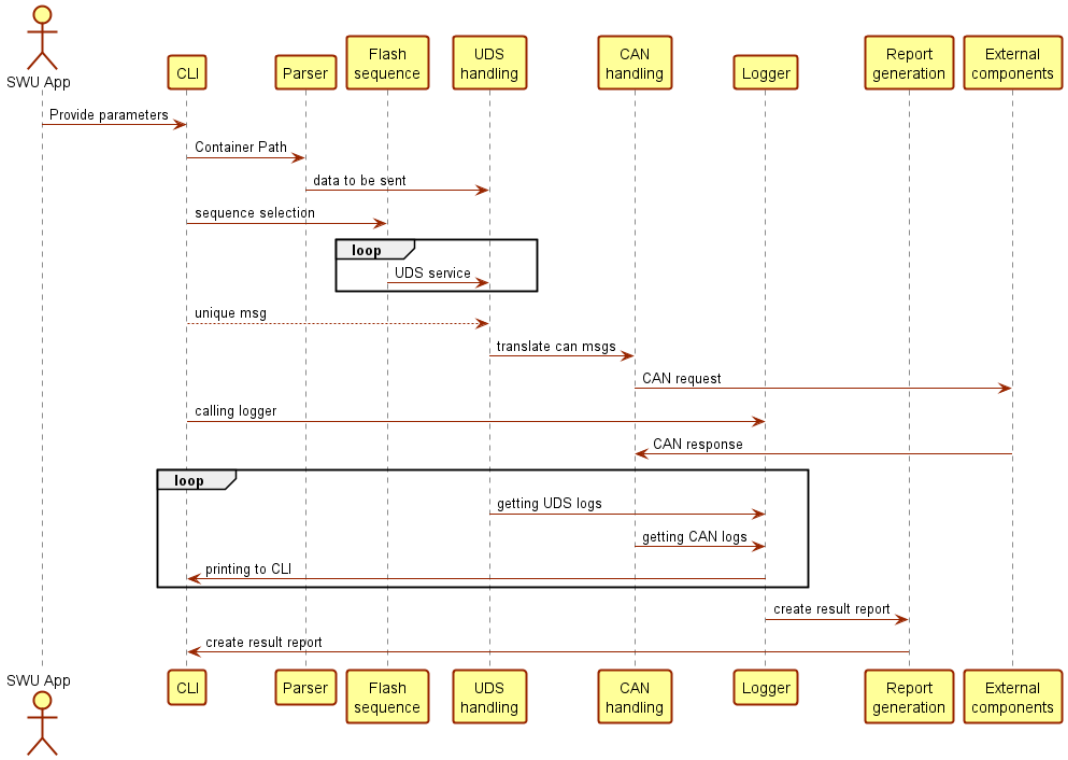
2.3. Adatfolyam

A szakdolgozatom keretében kifejlesztett szoftverfrissítési rendszer egy komplex, többrétegű architektúrára épül, amelynek célja a járművek elektronikus vezérlőegységeinek hatékony és megbízható frissítése. A rendszer központi eleme egy felhasználóbarát parancssori interfész (CLI), amelyen keresztül a felhasználók interaktív módon adhatják meg a frissítéshez szükséges paramétereket.

A rendszer további jellemzője a külső komponensekkel való kommunikáció képessége, amely lehetővé teszi a frissítési eredmények integrálását és további feldolgozását. A folyamat záró lépéseként a Report generation modul generál részletes jelentéseket, amelyek áttekintést nyújtanak a teljes frissítési folyamatról.

Az aktivitásdiagram részletesen mutatja be a rendszer működési logikáját, a döntési pontokat és a különböző tevékenységek közötti átmeneteket, lehetővé téve a rendszer működésének mélyebb megértését.

2.3.1. Szekvencia diagram

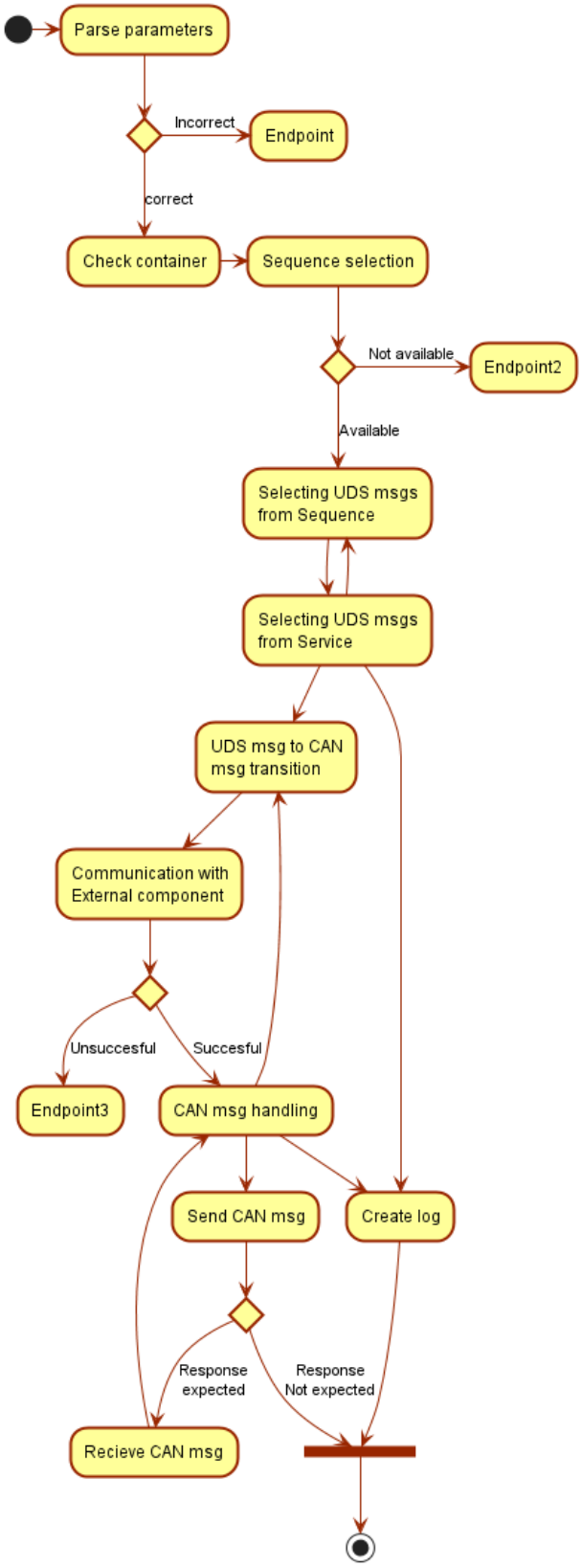


2.3.1. ábra: Szekvencia diagram

A szakdolgozatomban bemutatott szoftverfrissítési alkalmazás egy összetett komponensekből álló rendszer, amely a kommunikáció és adatfolyamat szabályozását végzi, ezt mutatja be a 2.3.1 ábra. A felhasználó által megadott paraméterek a parancssori interfész (CLI) által kerülnek fogadásra, majd továbbításra az Elemző (Parser) komponens felé, amely az adatokat feldolgozza és továbbítja a Flash sequence részére. A Flash sequence komponens az egyes frissítési szekvenciákat válogatja meg és kezdeményezi az UDS szolgáltatások iteratív hívásait, amelyek során az adatokat a CAN protokoll által értelmezhető formátumba alakítja át.

A CAN hálózati kommunikációt a CAN handling komponens bonyolítja le, amely az üzeneteket továbbítja, és fogadja. A Logger komponens a kommunikációs tevékenységek naplózásáért felelős, biztosítva ezzel a folyamatok átláthatóságát és ellenőrizhetőségét. A Logger által generált naplók ciklikusan frissülnek, tartalmazva a CAN üzeneteket és az UDS naplókat, amelyek létfontosságú információkat nyújtanak a frissítési folyamat jelenlegi állapotáról.

2.3.2. Aktivitás diagram



2.3.2. ábra**:** Aktivitás diagram

A szoftverfrissítési folyamatot bemutató 2.3.2 ábrán látható aktivitásdiagram a paraméterek elemzésével kezdődik, ahol az alkalmazás ellenőrzi a bemeneti paraméterek helyességét. Amennyiben a paraméterek helytelenek, az alkalmazás véget ér. Ha a paraméterek helyesek, a folyamat folytatódik a konténer állapotának ellenőrzésével, ahol a szükséges szekvencia kiválasztásra kerül. Abban az esetben, ha a kívánt szekvencia nem érhető el, a folyamat véget ér.

Az elérhető szekvencia esetén a rendszer kiválasztja az UDS (Unified Diagnostic Services) üzeneteket a szekvenciából, majd azokat szolgáltatásonként. Az UDS üzeneteket követően az alkalmazás átalakítja azokat CAN üzenetekké, és megkísérli a kommunikációt egy külső komponenssel. Amennyiben ez a kommunikáció sikertelen, a folyamatszintén véget ér. Sikeres kommunikáció esetén az alkalmazás továbbhalad a CAN üzenetkezeléshez, ahol a CAN üzeneteket elküldi és naplózza a tevékenységeket.

A CAN üzenetek küldését követően a rendszer eldönti, hogy vár-e választ az üzenetre vagy sem. Amennyiben választ vár, a rendszer fogadja a CAN üzeneteket. Ha nem vár választ, a folyamat lezárul. A diagram átfogó képet ad a szoftverfrissítési folyamatról, kiemelve az egyes döntési pontokat és az alkalmazás által végrehajtott fő tevékenységeket.

3. TECHNOLÓGIÁK

A szakdolgozatom során számos modern technológiát és módszertant alkalmaztam, amelyek hozzájárultak a projekt sikeres megvalósításához és a szoftverfejlesztés hatékony menedzseléséhez. A fejlesztőcsapat fejlesztési módszerét, illetve az általuk használt technológiák kezelését is a szakdolgozatom során felhasználtam és arra törekedtem, hogy ezeknek a technológiáknak- a segítségével a fejlesztési folyamatot megkönnyítsem.

3.1. Agilis módszertan

Az agilis módszertan keretében iteratív és inkrementális fejlesztési ciklusokat folytattam a csapattal, amelyek lehetővé tették a gyors és rugalmas reagálást a projekt követelményeinek változásaira. Az agilis módszertan alapelveinek megfelelően a fejlesztés során két hetente tartott sprintek és napi stand-up találkozók során haladt a munka.

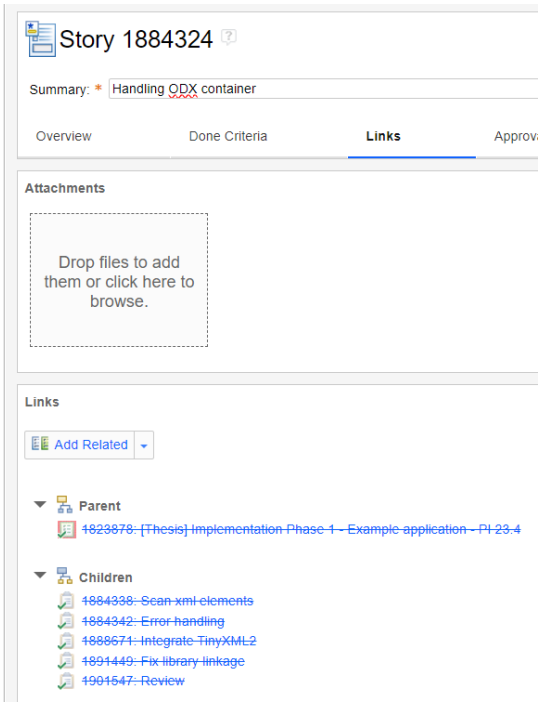
A fejlesztési ciklusokat követően minden ötödik sprint végén Program Increment (PI) tervezési szakasz került megtartásra, ahol az elvégzett munka és a további tervek egyaránt lettek megbeszélve. Ezek a találkozók kulcsfontosságúak voltak a projekt irányának meghatározásában, lehetővé téve, hogy az eddigiekben szerzett ismeretek alapján pontosíthassam a következő lépéseket.

A bemutató alkalmak különös hangsúlyt kaptak, hiszen ezeken keresztül mutattam be részletesen a fejlesztési eredményeket, az esetleges akadályokat és az azokra talált megoldásokat. Ezek az események biztosították, hogy a szakdolgozat írása során mindvégig szem előtt tarthassam a végcél elérését, miközben biztosítottam a munka folyamatos előrehaladását.

3.2. RTC tábla

A szakdolgozatom során az IBM Rational Team Concert (RTC) szoftvert és az Agile módszertant alkalmaztam a projektmenedzsment és a szoftverfejlesztés területén. Az RTC egy kollaboratív szoftverfejlesztő eszköz, amely integrálja a változás- és konfigurációkezelést, így támogatva az Agile elveket és gyakorlatokat. Az eszköz lehetővé teszi a feladatok és a kiadások nyomon követését.

A projekt tervezési fázisában előre definiáltam a mérföldköveket, amelyek az RTC-n belül 'feature'-öként jelennek meg. Ezek részletes leírásokkal készültek, amelyek az elfogadási követelményeket is magukban foglalták. A leírásokban válaszoltam az alapvető kérdésekre, mint például "Miért van szükség erre a funkcióra?", "Mit kell a funkció teljesítésével elérni?", "Van-e valamilyen függősége a funkcióknak?". Ezek a mérföldkövek voltak az alapjai a részletesebben kidolgozott felhasználói történeteknek, vagyis 'user story'-knak, amelyeket tovább bontottam feladatokra, vagyis 'task'-okra.



3.2. ábra

A 'user story'-k a fejlesztési folyamat központi elemei voltak, amelyek konkrét felhasználói igényeket és célokat fogalmaztak meg. Mindegyik 'user story' egy-egy kisebb, kezelhető méretű feladatsorozatra bontódott, amelyek specifikus követelményeket és célokat határoztak meg számomra, ezt a kapcsolatot szemlélteti a 3.2-es ábra.

3.3. PlantUML

A szakdolgozatomban használt diagramok elkészítéséhez a PlantUML eszközt választottam, amely egy nyílt forráskódú eszköz diagramok generálására. A PlantUML széles körben elterjedt gyakorlat a Bosch-nál, amelynek köszönhetően a szoftverfejlesztési folyamatokat és rendszerek tervezését nagyfokú pontossággal és egyszerűséggel lehet leírni és vizualizálni. Az eszköz használata a szabványos UML (Unified Modeling Language) szintaxist követi, ami lehetővé teszi a fejlesztők számára, hogy gyorsan és hatékonyan hozzanak létre különféle diagramokat, mint például osztály-, aktivitás-, vagy szekvenciadiagramokat. PlantUML használatának előnyei közé tartozik, hogy a diagramokat nem vizuális tervezés útján hozzuk létre, hanem kód írásával. A kód alapú megközelítés lehetővé tette, hogy a precízen és gyorsan definiáljam a komplex struktúrákat és folyamatokat anélkül, hogy kézzel kellene rajzolnom minden egyes elemet.

```
@startuml
skinparam componentStyle rectangle

[FlashContainer] -> [Flashtool]
[Flashtool] <- [Logging parameters]
[Flashtool] <-- [Selected mode]

[Client PC] <-- [Flashtool]

[Client PC] -> [Vector VN1610]

[Vector VN1610] --> [Vector VN1610 2.0]

[Vector VN1610 2.0] -> [Client PC(simulation)]

[Vector VN1610] -> [ExampleECU]
@enduml
```

3.3 kódrészlet

Az 3.3 kód részlet a System diagramhoz tartozó .puml kiterjesztésű forráskódot tartalmazza.

4. MODULOK

Dolgozatom készítéséhez a Visual Studio Code (VSCode) fejlesztőkörnyezetet használtam, amely egy széleskörben elterjedt, nyílt forráskódú, platformfüggetlen szövegszerkesztő és fejlesztői eszköz. A VSCode kiterjeszthető funkciói és könnyű használhatósága tette azt ideális választássá a dolgozat írása és a kód fejlesztése során egyaránt.

A fejlesztési folyamat során a MinGW (Minimalist GNU for Windows) 8.2.0 verzióját választottam, mint fordító és eszközlánc, amely lehetővé teszi a GNU fejlesztőeszközök Windows operációs rendszeren történő használatát. A MinGW-t a VSCode integrált terminálján konfiguráltam, így közvetlenül a fejlesztőkörnyezetből indíthattam a fordítási és hibakeresési folyamatokat.

A VSCode konfigurációját tasks.json és launch.json fájlok segítségével végeztem, amelyek meghatározzák a fordítási feladatokat és a hibakeresés beállításait. A tasks.json tartalmazza a fordításhoz szükséges parancsokat, míg a launch.json a hibakereső eszköz konfigurációját írja le. Ezeket a fájlokat a VSCode automatikusan generálja és kezeli, biztosítva a fejlesztési folyamat egyszerűségét és gördülékenységét.

- A MinGW használata a VSCode-ban számos előnyt biztosított a fejlesztés során, többek között:
- A GNU fejlesztőeszközök teljes körű támogatása, beleértve a GCC fordítót és a GDB hibakeresőt.
- A VSCode által nyújtott intelligens kódszerkesztési funkciók, mint a kódkiemelés, kódkiegészítés és refaktorálás.
- Az integrált terminál és hibakereső interfész, amely lehetővé teszi a kód fordítását és futtatását anélkül, hogy elhagynánk a fejlesztői környezetet.
- A VSCode által kínált kiterjesztések használata, mint például a C/C++ kiterjesztés a Microsoft-tól, amely további szintaktikai elemzést és fejlett hibakeresési lehetőségeket biztosít.

A dolgozatom fejlesztési folyamatában ez a konfiguráció biztosította a szükséges flexibilitást és hatékonyságot, lehetővé téve, hogy a szoftverfrissítési folyamatot a legmodernebb eszközökkel és technikákkal valósítsam meg.

4.1. Elemező

A szakdolgozatomban kifejlesztett Elemező (Parser) modul fejlesztése során két fő nyíltforráskódú könyvtár segítségét vettem igénybe: a tinyxml2 és a miniz. A tinyxml2 könyvtár lehetővé teszi XML fájlok hatékony elemzését és kezelését, míg a miniz a PDX fájlok tömörítésére és kicsomagolására szolgál. Ezek az eszközök különösen hasznosnak bizonyultak az autóiparban használt diagnosztikai adatformátumok, mint az ODX (Open Diagnostic data eXchange)[1] és a PDX (Package Data Exchange), kezelésében. Az ODX egy nyílt formátum, amely a járműdiagnosztikában használt adatok leírására szolgál,[3], míg a PDX egy csomagolt formátum, amely tömörített formában tartalmazza az ODX adatokat. Az Elemző modul a következő funkciókat látja el a szakdolgozatomban:

- PDX fájlok kicsomagolása miniz segítségével és az abban található ODX adatok kinyerése.
- Az ODX fájlok strukturált elemzése és a konfigurációs paraméterek kinyerése a tinyxml2 segítségével.
- Az így kinyert adatok FlashObject objektumokká alakítása, melyek magukban foglalják azokat a kulcsinformációkat, mint a flash adatok azonosítói, kezdőcímek és méretek, melyek a frissítési folyamat végrehajtásához szükségesek.

Az Elemző modul nélkülözhetetlen a járműszoftverek frissítéséhez, hiszen biztosítják, hogy a szoftverfrissítési folyamat során használt fájlok megfelelően kerüljenek beolvasásra és feldolgozásra. Az operációs rendszer független könyvtárak használata továbbá növeli a rendszer rugalmasságát és hordozhatóságát, lehetővé téve a különböző platformokon történő felhasználást is, amely kritériuma volt a szakdolgozatomnak.

4.2. Config

A Config modulban implementáltam egy XML konfigurációs osztályt, amelynek feladata a szoftverfrissítési folyamat során használt beállítások és paraméterek kinyerése. Ez az osztály felelős a konfigurációs adatok betöltéséért és kezeléséért, valamint a szükséges paraméterek eléréséért a szoftver különböző részei számára. A szakdolgozatom keretében kifejlesztett Config modulban egy XML konfigurációs osztályt implementáltam, amely a projekt specifikus beállítások kezeléséért felelős. Ez az osztály, amely a config.xml fájlból olvas be adatokat, kulcsfontosságú szerepet tölt be a szoftverfrissítési folyamat során, hiszen itt tárolódnak azok a konfigurációs paraméterek, amelyek a projekt specifikációjához szükségesek.

4.3. Kommunikáció

A szakdolgozatom során a Vector XL API-t (vxlapi) használtam a CAN hálózati kommunikáció megvalósításához. A vxlapi egy átfogó programozási interfész, amely lehetővé teszi a fejlesztők számára, hogy közvetlenül interakcióba lépjenek a CAN hálózattal és a hozzá kapcsolódó hardverekkel, mint például a Vector CAN interfész kártyák. Az API széleskörű funkcionalitást biztosít, többek között a CAN üzenetek küldését, fogadását és a hálózati események kezelését. A Vector XL API azért vált választásom tárgyává, mert stabil és megbízható megoldást kínál a járműipari kommunikációs feladatokhoz, illetve az API támogatja a CAN FD-t (Flexible Data-rate), ami lehetővé teszi nagyobb adatátviteli sebességeket és hatékonyabb hálózatikapacitás kihasználást.

A vxlapi használatával kapcsolatban az XLstatus érték kulcsfontosságú szerepet tölt be, hiszen ez jelzi, hogy az egyes API hívások sikeresen lefutottak-e. Az XLstatus egy adattípus, általában egy enumeráció, amelyet a használhatunk egy művelet státuszának jelzésére. Ez a módja annak kommunikálására, hogy egy adott funkció vagy parancs sikeresen végrehajtódott-e, vagy hiba történt. A könyvtárak függvényei egy XLstatus értéket adnak vissza az eredmény jelzésére. Például egy meghajtó inicializálásához, üzenet küldéséhez vagy kommunikációs csatorna megnyitásához kapcsolódó funkció visszatér egy XLstatus értékkel.

```
XLstatus              xlStatus;
if(XL_SUCCESS == xlStatus)
```

4.3. kódrészlet

A 4.3. kódrészlet demonstrálja a sikeres eredmény feltételét, melyben a XL_SUCCESS egy konstans, amelyet a könyvtárban definiálnak. Az XL_SUCCESS ellenőrzése alapvető a hibakezelésben. Amikor a program meghív egy hardverkommunikációs vagy adatátviteli függvényt, ellenőriznie kell, hogy a visszaadott státusz XL_SUCCESS-e. Ha nem, akkor ez azt jelzi, hogy valamilyen probléma történt a művelet során, és a program megfelelően reagál.

4.3.1. Driver inicializáció

A projekt kommunikációs részében az első lépés a Vector XL könyvtár által nyújtott driver inicializációja. A driver inicializáció folyamata során a rendszer konfigurálja a CAN csatornákat, ellenőrzi a CAN FD támogatását, és beállítja a szükséges kommunikációs paramétereket. Ez a lépés kritikus jelentőségű, hiszen biztosítja a hálózati kommunikáció stabil alapjait.

```
// open the driver
xlStatus = xlOpenDriver();

// get/print the hardware configuration
if(XL_SUCCESS == xlStatus) {
  xlStatus = xlGetDriverConfig(&g_xlDrvConfig);
}
```

4.3.1.a kódrészlet

A 4.3.1.a kódrészletben a driver inicializációs folyamat során az xlOpenDriver() függvény hívással megnyitjuk a drivert, amely lehetővé teszi a kommunikációt a hardvereszközökkel. Ezt követően az xlGetDriverConfig() függvény segítségével lekérdezzük a hardverkonfigurációt, ami fontos információkat nyújt a rendelkezésre álló csatornákról és a hálózati képességekről.

```
if(canFdSupport) {
    XLcanFdConf fdParams = {};
    // arbitration bitrate
    fdParams.arbitrationBitRate = 1000000;
    fdParams.tseg1Abr           = 6;
    fdParams.tseg2Abr           = 3;
    fdParams.sjwAbr             = 2;

    // data bitrate
    fdParams.dataBitRate = fdParams.arbitrationBitRate*2;
    fdParams.tseg1Dbr    = 6;
    fdParams.tseg2Dbr    = 3;
    fdParams.sjwDbr      = 2;

    if (g_canFdModeNoIso) {
    fdParams.options = CANFD_CONFOPT_NO_ISO;
    }

    xlStatus = xlCanFdSetConfiguration(g_xlPortHandle,
g_xlChannelMask, &fdParams);
    printf("- SetFdConfig.     : ABaudr.=%u, DBaudr.=%u,
%s\n", fdParams.arbitrationBitRate, fdParams.dataBitRate,
xlGetErrorString(xlStatus));
}
```

4.3.1.b kódrészlet

A 4.3.1.b kódrészlet a CAN FD támogatás ellenőrzését és beállítását szemlélteti, amellyel a rendszer képessé válik a magasabb adatátviteli sebességek kezelésére. Az XLcanFdConf struktúrában megadott paraméterek, mint például az arbitrációs és adatátviteli sebesség, a jelzésidők és a szinkronizációs ugrás szélessége, pontosítják a CAN FD üzenetek formátumát és sebességét. Az xlCanFdSetConfiguration() függvény meghívása után a visszatérési érték ellenőrzése biztosítja, hogy a konfiguráció sikeres volt-e, és ha nem, az xlGetErrorString() segítségével azonnal tájékoztatást kaphatunk a hiba természetéről.

4.3.2. CAN üzenetküldés

Ez a modul a CAN csatornán keresztüli kommunikációért felelős, amit a Vector XL könyvtár segítségével valósítottam meg. A Vector XL könyvtár kiemelkedő eszköz a CAN hálózati kommunikáció kezelésére, lehetővé téve a szoftver számára, hogy közvetlenül kommunikáljon a CAN hálózaton keresztül az ECU-kkal és egyéb hálózati eszközökkel.

```
XLstatus transmit(unsigned int txID, XLaccess xlChanMaskTx,
std::vector<uint8_t> data)
{
  XLstatus              xlStatus;
  unsigned int         messageCount = 1;
  static int               cnt = 0;

  if (canFdSupport) {
      XLcanTxEvent canTxEvt;
      unsigned int cntSent;

      memset(&canTxEvt, 0, sizeof(canTxEvt));
      canTxEvt.tag = XL_CAN_EV_TAG_TX_MSG;

      canTxEvt.tagData.canMsg.canId = txID;
      canTxEvt.tagData.canMsg.msgFlags =
XL_CAN_TXMSG_FLAG_EDL; // Always CAN FD
      canTxEvt.tagData.canMsg.dlc = 15; // DLC for CAN FD (64
bytes)

      for (size_t i = 0; i < data.size() && i <
XL_CAN_MAX_DATA_LEN; i++) {
            canTxEvt.tagData.canMsg.data[i] = data[i];
      }

      xlStatus = xlCanTransmitEx(g_xlPortHandle, xlChanMaskTx,
messageCount, &cntSent, &canTxEvt);
      std::cout << "Sent on CAN FD" << std::endl;
}
```

4.3.2. kódrészlet

A 4.3.2. kódrészletben a CAN FD (Flexible Data-rate) protokoll implementálása döntő jelentőséggel bír a kommunikációs sebesség növelésében. Ez a protokoll magasabb adatátviteli sebességet és hatékonyabb adatkezelést tesz lehetővé, amit a xlCanTransmitEx függvény hívása biztosít.

4.3.3. CAN üzenetfogadás

A CAN üzenet fogadás modulja a bejövő adatok kezeléséért felelős. A xlCanReceive függvény segítségével a rendszer képes fogadni a CAN hálózaton érkező üzeneteket. A fogadási folyamat során az alkalmazás figyelemmel kíséri a hálózati forgalmat, és a beállított időkorláton belül érkező üzeneteket feldolgozza. Ez a modul kritikus szerepet tölt be a kommunikációs láncban, hiszen lehetővé teszi a rendszer számára, hogy reagáljon a hálózaton érkező adatokra és üzenetekre.

```
XLcanRxEvent reception(int timeoutInSeconds) {
    XLstatus xlStatus;
    XLcanRxEvent xlCanEvent = {};

    // Convert timeout from seconds to milliseconds
    int timeoutInMilliseconds = timeoutInSeconds * 1000;

    // Check if the timeout value is invalid
    if (timeoutInMilliseconds < 0) {
        // Set default timeout value
        timeoutInMilliseconds = 5000; // 5 seconds
    }

    auto start_time =
std::chrono::high_resolution_clock::now();
    do {
        xlStatus = xlCanReceive(g_xlPortHandle, &xlCanEvent);
        if (xlStatus != XL_ERR_QUEUE_IS_EMPTY) {
            break;
        }
    } while
(std::chrono::duration_cast<std::chrono::milliseconds>(
                std::chrono::high_resolution_clock::now() -
start_time).count() < timeoutInMilliseconds);

    return xlCanEvent;
}
```

4.3.3. kódrészlet

A 4.3.3. kódrészletben az XLcanRxEvent struktúra található, amely a CAN hálózaton keresztül érkező üzenetek információit tartalmazza, beleértve az üzenet azonosítóját, adattartalmát és egyéb attribútumait, mint például az üzenet hosszát és zászlóit. Az xlCanReceive függvény hívása a várakozási idő lejártáig (amelyet a timeoutInSeconds paraméter határoz meg) ciklikusan ellenőrzi az eseményváró sor állapotát a CAN kommunikációban. Amennyiben az eseményváró sor nem üres (azaz van fogadott üzenet), az XL_ERR_QUEUE_IS_EMPTY hibakód helyett más értéket kapunk vissza az xlStatus változóban, és az üzenetet tartalmazó XLcanRxEvent struktúra adatait elérhetjük. Amennyiben a függvény sikeresen észlel egy üzenetet, a ciklus megszakad és a függvény visszaadja az XLcanRxEvent struktúrát, ami lehetővé teszi számunkra, hogy feldolgozzuk a fogadott CAN üzenetet.

4.4. Services

A szakdolgozatomban részletesen foglalkoztam a "Services" modullal, amely számos kulcsfontosságú fájlt tartalmaz, melyek együttesen alkotják az üzleti logikát, valamint kezeli az ISO-TP (ISO 15765-2) protokollt és az UDS (Unified Diagnostic Services) szolgáltatásokat.

4.4.1. CAN

A CAN osztály a CAN kommunikációs protokoll kezelésére szolgál. A konstruktora inicializálja az osztály változóit, és biztosítja, hogy az osztályból csak egy példány létezzen (singleton pattern). Ez az egyedüli példány a getInstance metódus segítségével hozható létre. A Config metódus konfigurálja az osztályt egy XML konfigurációs fájl alapján (XML Config segítségével), beállítva az arbitrációs azonosítót és a CAN paramétereket.

A send metódus lehetővé teszi a CAN üzenetek küldését a hálózaton, míg a receive metódus a beérkező üzenetek fogadására szolgál. Mindkét metódus a CANBus osztállyal kommunikál, amely a tényleges alacsony szintű kommunikációt kezeli. A shutdown metódus leállítja a kommunikációt és felszabadítja az erőforrásokat.

4.4.2. CANBus

A CANBus osztály alapvetően fontos szerepet tölt be a CAN kommunikáció kezelésében. Ez az osztály felelős a CAN üzenetek küldéséért és fogadásáért, valamint a CAN hálózati interfész kezeléséért. Az alábbiakban bemutatom, hogy a CANBus osztály hogyan kapcsolódik a rendszer többi részéhez.

- Inicializálás: A konstruktorban a initDriver függvény meghívásával inicializálja a CAN hálózati illesztőt.
- Adatküldés: A send metódus segítségével küldhetőek CAN üzenetek a hálózaton keresztül.
- Adatfogadás: A receive metódus az érkező CAN üzeneteket fogadja.
- Leállítás: A shutdown metódus lezárja a CAN kommunikációs csatornát.

```
CANBus::CANBus(int& channel, char* app_name, bool& can_fd) {
    XLstatus status;
    unsigned int xlChannel = 0;

    try {
        status = initDriver(&g_xlChannelMask, &xlChannel,
app_name, can_fd);
    } catch (const std::exception& e) {
        std::cerr << "Exception occurred: " << e.what() <<
std::endl;
    }
}
```

4.4.2. kódrészlet

A 4.4.2. kódrészletben látható, hogy amikor egy CANbus példány létrejön, megtörténik az eszköz konfigurálása, az eszköz FD compabilitásának észlelése és a csatornák aktiválása, ami lehetővé teszi, hogy a CAN-en végzett műveleteket el lehessen végezni. Mindezek nélkül nem lehetne üzenetet se küldeni se fogadni, ezért is helyeztem el az osztály konstruktorában.

4.4.3. CANIsoTp

A CanIsoTp osztály a CAN hálózaton keresztüli ISO-TP (ISO 15765-2) protokoll alapú kommunikációt kezeli. Az osztály feladata a nagyobb méretű adatcsomagok szegmentálása és újraegyesítése a CAN hálózat korlátozott üzenetmérete miatt. Az osztály a CAN* típusú objektumot használja, amely közvetlen hozzáférést biztosít a CAN hálózati réteghez. A CanIsoTp egy singleton osztály, amely garantálja, hogy csak egy példánya létezzen a program futtatása során. A CanIsoTp::send metódus a kulcsfontosságú interfész a CAN hálózaton keresztüli adatküldésre. Ez a metódus a kapott adatcsomagot (payload) továbbítja a IsoTpHandler objektumnak, ami kezeli az ISO-TP protokoll szerinti adatátviteli folyamatot.

```
void CanIsoTp::send(const std::vector<uint8_t>& payload) {
    // Pass the payload to IsoTpHandler for processing
    isotpHandler.handleTransmission(payload);
}
```

4.4.3. kódrészlet

A 4.4.3. kódrészletben a send metódus felépítését ábrázolom, amikora ez a metódus meghívódik, az egy std::vector<uint8_t> típusú adatcsomagot (payload) kap paraméterként. A metódus ezután továbbítja ezt az adatcsomagot az IsoTpHandler osztály handleTransmission metódusának. A handleTransmission metódus feladata az adatcsomag ISO-TP protokoll szerinti feldolgozása. Ez magában foglalhatja az adatok szegmentálását kisebb részekre, a folytonosság biztosítását, és az adatcsomagok újraegyesítését a fogadó oldalon.

4.4.4. IsoTpHandler

A CAN ISO-TP (ISO 15765-2) kommunikációs protokoll, amely a CAN hálózaton történő adatátvitelt teszi lehetővé nagyobb adatmennyiségek esetén is. Az ISO-TP protokoll a hosszabb üzeneteket szegmentálja és több kisebb, könnyebben kezelhető üzenetre bontja, amelyeket a CAN hálózaton keresztül tudunk elküldeni.

```
void IsoTpHandler::initializeTransmission(const
std::vector<uint8_t>& payload) {
    currentPayload = payload;
    offset = 0;
    remaining = payload.size();
    sequenceNumber = 1;
    txState = payload.size() <= (MAX_FRAME_SIZE - 2) ?
TxState::TransmitSingleFrame : TxState::TransmitFirstFrame;
    std::cout << "Initializing Transmission." << std::endl;
}
```

4.4.4. a kódrészlet

A 4.4.4. a kódrészletben a program eldönti, a payload változó méretének segítségével, hogy ha a bejövő adat nagyobb mint 62 bájt akkor a State machine-t SingleFrameként kezdi, ha nagyobb mint 62 akkor FirstFrame státuszban kezd amely, aztán a tovább iterálhat a további státuszokon.[1]

```
void IsoTpHandler::processStateMachine() {
    switch (txState) {
        case TxState::Idle:
            // Default state
            break;
        case TxState::TransmitSingleFrame:
            sendSingleFrame(currentPayload);
            break;
        case TxState::TransmitFirstFrame:
            sendFirstFrame();
            break;
        case TxState::WaitFlowControl:
            // Handle wait for flow control state
            waitForCTS(1000);
            break;
        case TxState::TransmitConsecutiveFrame:
            sendConsecutiveFrames();
            break;
        case TxState::Finished:
            // Done state
            break;
    }
}
```

4.4.4. b kódrészlet

A 4.4.4. b kódrészletben látható az úgynevezett StateMachine, mely a bejövő adat hosszából eldönti, hogy szükséges-e feldarabolni az üzenetet.

Amennyi adatot egyszerre lehet küldeni CAN FD-n keresztül az 64 byte, ha ezt meghaladja a küldeni kívánt méret, akkor a StateMachine először is elküldi FirstFrame-t, amelynek az első két bájtja speciális jelentéssel bír. Az első bájt a Most Significant bájt, amely az üzenet típusát jelöli.[2]

CAN FD Frame	Rx	15	64	12 02 36 01 11 9D FF
CAN FD Frame	Rx	3	3	30 00 00
CAN FD Frame	Rx	15	64	21 B0 3F 1F 08 22 03
CAN FD Frame	Rx	15	64	22 DA F0 DA 49 66 05
CAN FD Frame	Rx	15	64	23 BD 53 EE 35 8A C3
CAN FD Frame	Rx	15	64	24 3B 77 0E 1A 6E 37
CAN FD Frame	Rx	15	64	25 94 1E 5A 4A 00 FE
CAN FD Frame	Rx	15	64	26 00 7E B8 9E 20 28

4.4.4. c ábra First Frame

A 4.4.4. c ábrán látható kijelölt adatrésznél az első két bájt (12 02) az ISO-TP First Frame hosszát jelöli. Ebben az esetben a '12 02' bájtpár a következő információt hordozza:

- Első bájt (12): Ez a bájt a PCI (Protocol Control Information) információt tartalmazza. Az ISO-TP First Frame esetében a PCI értéke '1' jellegű, ami az első adat framet jelzi, és az első bájt felső nibble-je (első négy bitje) tartalmazza ezt az információt. A '12' hexadecimális értékben az első négy bit ('0001') jelzi az első framet, míg a második négy bit (a '2', ami decimálisban '0010') az üzenet hosszának felső nibble-jét képezi.
- Második bájt (02): Ez a bájt az üzenet hosszának alsó bájtja. Az első bájt második nibble-je és a második bájt együtt adják meg az üzenet teljes hosszát.

Ebben az esetben a '12 02' kombináció azt jelenti, hogy az üzenet hossza '0202' hexadecimális, vagyis 514 decimális bájt.

Ha az üzenet megfelelően érkezik meg a fogadó félhez, egy Flow Control adat framet kapunk, (30 00 00) melynek első bájtja szintén a PCI-t jelzi, a következő bájt az időzítést, az utolsó pedig az adatfolyam engedélyezését. Ezt követően az állapotgép (StateMachine) átlép az adatok folyamatos küldésének fázisába, amely az ISO-TP protokoll működésének kulcsa.

CAN FD Frame	Rx	15	64	21 B0 3F 1F 08 22 03
CAN FD Frame	Rx	15	64	22 DA F0 DA 49 66 05
CAN FD Frame	Rx	15	64	23 BD 53 EE 35 8A C3
CAN FD Frame	Rx	15	64	24 3B 77 0E 1A 6E 37
CAN FD Frame	Rx	15	64	25 94 1E 5A 4A 00 FE
CAN FD Frame	Rx	15	64	26 00 7E B8 9E 20 28
CAN FD Frame	Rx	15	64	27 7D 3A 44 92 D8 26
CAN FD Frame	Rx	15	64	28 40 B0 0B AA AB 3C
CAN FD Frame	Rx	3	3	02 76 01

4.4.4. d ábra Consecutive Frames

A 4.4.4. d ábrán egy folyamatos küldés (Consecutive Frame) állapot látható, ahol a CAN réteg a '21' kezdetű PCI-vel jelölt bájtokat küldi egymás után. Az iterációs folyamat addig tart, amíg az összes, az első üzenetben meghatározott 514 bájt át nem kerül a hálózaton. Az ISO-TP protokoll ezen fázisa biztosítja, hogy a nagyobb adatcsomagok is megbízhatóan és hatékonyan továbbíthatók legyenek a korlátozott méretű CAN üzenetekben.

4.4.5. UDSonCAN

Az UDSonCAN osztály a CAN hálózaton keresztüli Unified Diagnostic Services (UDS) kommunikációt kezeli. Az osztály konstruktora egy CanIsoTp típusú objektumot fogad, amely a CAN-ISO-TP protokollra vonatkozó információkat tartalmazza, mint például az üzenetek várakozási idejét (rpTimeout), az üzeneteket fogadó azonosítót (rxid) és a CAN FD támogatásának állapotát (can_fd). Az osztály singleton mintát követ, így biztosítva, hogy egyszerre csak egy példánya létezhessen.

Az osztály konstruktora, amely inicializálja a tagváltozókat és beállítja a szükséges kommunikációs paramétereket. A CanIsoTp objektumot használja a CAN hálózati kommunikációhoz.[6] A Config metódus segítségével későbbi konfigurációt végezhetünk, ahol beállíthatjuk a különböző paramétereket, mint a projekt nevét (projectName) és a CAN példányt (can).

```
std::pair<bool, std::vector<uint8_t>> UDSonCAN::sendUdsMessage(const
std::vector<uint8_t>& udsData, bool waitForResponse) {
    if (projectName == "ProjectName") {
        size_t size = udsData.size();
        size_t raw_data_size = this->can_fd ? 62 : 7;

        std::vector<uint8_t> can_data;
        if (size <= raw_data_size) {
            // Prepend the size of the UDS data
            can_data.push_back(static_cast<uint8_t>(size / 256));
            can_data.push_back(static_cast<uint8_t>(size % 256));
            can_data.insert(can_data.end(), udsData.begin(),
udsData.end());

            // If size is small, adjust the data format to exclude
the first byte
            if (size <= 7) {
                canisotp->send(std::vector<uint8_t>(can_data.begin()
+ 1, can_data.end()));
            } else {
                canisotp->send(can_data);
            }
        } else {
            // For larger data, send it directly as ISO-TP frames
are meant to handle multi-frame messages
            canisotp->send(udsData);
        }
    }
    if (waitForResponse) {
        return waitForUdsResp(1);
    }
    return {false, std::vector<uint8_t>()}; // No response expected
}
```

4.4.5. a kódrészlet

A 4.4.5. a kódrészlet lehetővé teszi a sendUdsMessage metódus által az UDS üzenetek küldését a CAN hálózaton, és választ várhat a fogadó oldaltól. Az üzenetek küldése során a metódus figyelembe veszi az adatok méretét és a CAN FD támogatást.[7]

A waitForUdsResp metódus egy meghatározott időkereten belül várja a válaszüzeneteket. Ez a folyamat az üzenetek időzítését és formátumát is figyelembe veszi, mint például az egyszerű vagy többszörös kereteket (multi-frame messages).

A sendUdsMessage és a waitForUdsResp metódusok közötti kapcsolat kulcsfontosságú az UDS kommunikációs folyamatban. Az sendUdsMessage metódus egy UDS kérést küld el a CAN hálózaton keresztül. Ha a waitForResponse paraméter igaz, akkor a metódus meghívja a waitForUdsResp metódust, amely várakozik a válaszra. Ez lehetővé teszi, hogy szinkron módon kezeljük az UDS kommunikációt: egy kérés elküldése után azonnal megvárjuk a választ, mielőtt továbblépnénk.

A minimális szoftverfrissítéshez szükséges UDS üzenetek a következők, transferData, requestDownload és requestTransferExit amleyek külön metódusként vannak implementálva és különböző UDS szolgáltatásokat valósítanak meg, mint az adatátvitel, letöltés kérése és az adatátvitel befejezésének kérése. Ezek a metódusok lehetővé teszik a CAN hálózaton keresztüli komplex UDS kommunikációt, ahol az üzenetek strukturált formában kerülnek továbbításra.

```
std::pair<bool, std::vector<uint8_t>>
UDSonCAN::requestDownload(uint8_t dfi, uint32_t address, const
std::vector<uint8_t>& data) {
    std::cout << "Sending request_download" << std::endl;

    uint8_t alfid = 0x44; // addressAndLengthFormatIdentifier
    std::vector<uint8_t> udsData;
    udsData.push_back(0x34); // 'request download' Service ID
    udsData.push_back(dfi);
    udsData.push_back(alfid);

    for (int i = 3; i >= 0; --i) {
        udsData.push_back((address >> (i * 8)) & 0xFF);
    }
    uint32_t dataLength = data.size();
    for (int i = 3; i >= 0; --i) {
        udsData.push_back((dataLength >> (i * 8)) & 0xFF);
    }
    std::cout << "UDS Data: ";
    for (auto byte : udsData) {
        std::cout << std::hex << static_cast<int>(byte) << " ";
    }
    std::cout << std::endl;
    return sendUdsMessage(udsData, true);
}
```

4.4.5. b kódrészlet

A 4.4.5.b kódrészletben található függvény lehetővé teszi a szoftverfrissítési folyamat során a szükséges adatok letöltését a jármű vezérlőegységébe. Az implementáció a következő fontos lépésekből áll.

Első lépésként a funkció hozzáadja a 0x34 értéket az udsData vektorhoz, ami a "letöltési kérelem" szolgáltatást jelzi. A következő lépésben az Adatformátum Azonosító, amely a függvénynek átadott paraméter. Ezután a funkció hozzáadja a dfi értékét. Ez az érték amelyet a függvény paramétereként kapott és az adatformátumot határozza meg. A következő lépésben a 0x44 érték kerül hozzáadásra, ami az adatok címének és hosszának formátumát határozza meg.

Az adathossz a letöltendő adatok méretét jelöli, ami szintén big-endian formátumban kerül az üzenethez hozzáadására. Ebben az esetben az adatok mérete 9, így a hossz értéke 0 0 0 9 lesz, ahol a legjelentősebb bájtok értéke 0, mivel a hossz kevesebb mint 256 (egy bájt).

Az udsData vektor hexadecimális bájtok sorozataként kerül létrehozásra, ahol minden push_back művelet egy újabb bájtot ad a sorozathoz. Ennek eredményeképpen a requestDownload funkció a következő sorozattá alakul: 0b 34 00 44 12 34 56 78 00 00 00 09, ahol minden részlet jelentősége a fentiek szerint értelmezhető.

A következő diagnosztikai lépés a transferData amelyre szintén megvalósításra került egy metódus ami kulcsfontosságú funkcióval bír hiszen ez az a protokoll amely az adatátviteli kérelmet kezeli a CAN hálózaton keresztül.

```
std::pair<bool, std::vector<uint8_t>> UDSonCAN::transferData(uint8_t
sequenceNumber, const std::vector<uint8_t>& data) {
    std::cout << "Sending transfer_data" << std::endl;

    // Construct the UDS data with the 'transfer data' SID and
sequence number
    std::vector<uint8_t> udsData;
    udsData.push_back(0x36); // 'transfer data' Service ID
    udsData.push_back(sequenceNumber);
    udsData.insert(udsData.end(), data.begin(), data.end());
    std::cout << "UDS Data: ";
    for (auto byte : udsData) {
        std::cout << std::hex << static_cast<int>(byte) << " ";
    }
    std::cout << std::endl;

    // Send the UDS message and return the result
    return sendUdsMessage(udsData, true);
}
```

4.4.5. c kódrészlet

A 4.4.5. c kódrészlet a transferData metódust hivatott bemutatni. Ez a metódus egy std::pair objektumot ad vissza, amely tartalmaz egy bool értéket és egy byte vektort. A bool érték jelzi, hogy az adatátviteli kérelem sikerült-e, míg a byte vektor a kapott UDS válaszadatokat tartalmazza.

A transferData metódus az UDS 'Transfer Data' (0x36) szolgáltatási azonosítóját (SID) használja az adatok átvitelére. Ez a metódus a következő lépéseket hajtja végre.

Először is a metódus hozzáadja a Transfer Data SID-t, majd a sorozatszámot (sequenceNumber) az UDS üzenethez. Ezután hozzáfűzi a küldeni kívánt adatokat. A metódus szintén kiírja az elküldendő adatokat, amely segít az üzenetkövetésben. Az így összeállított üzenetet már UDS üzenetnek nevezzük, amit elküldünk a sendUdsMessage metódus segítségével.

A végső lépés requestTransferExit metódus, amely az Unified Diagnostic Services (UDS) protokoll részeként az adatátviteli folyamat befejezését kezeli. A requestTransferExit metódus az UDSonCAN osztály egyik fontos funkciója, amely az Unified Diagnostic Services (UDS) protokoll részeként az adatátviteli folyamat befejezését kezeli a CAN hálózaton keresztül.

```
std::pair<bool, std::vector<uint8_t>>
UDSonCAN::requestTransferExit(const
std::optional<std::vector<uint8_t>>& data) {
    std::cout << "Sending request_transfer_exit" << std::endl;

    // Construct the UDS data with 'request transfer exit' ID
    std::vector<uint8_t> udsData;

    // If additional data is provided, insert it
    if (data.has_value()) {
        const auto& additionalData = data.value();
        udsData.insert(udsData.end(), additionalData.begin(),
additionalData.end());
    }

    // Insert the 'request transfer exit' Service ID
    udsData.push_back(0x37);

    // Log the UDS data
    std::cout << "UDS Data: ";
    for (auto byte : udsData) {
        std::cout << std::hex << static_cast<int>(byte) << " ";
    }
    std::cout << std::endl;
    // Send the UDS message and return the result
    return sendUdsMessage(udsData, true);
}
```

4.4.5. d kódrészlet

A requestTransferExit metódus parameter listájában opcionálisan hozzáadható további adatok byte-sorozata, amelyek szükség esetén részét képezik a kérelemnek. A metódus egy std::pair objektumot ad vissza, amely tartalmaz egy bool értéket és egy byte vektort. A bool érték jelzi, hogy a 'Transfer Exit' kérelem sikerült-e, míg a byte vektor a kapott UDS válaszadatokat tartalmazza. A requestTransferExit metódus az UDS 'Request Transfer Exit' (0x37) szolgáltatási azonosítóját (SID) használja a kommunikációs folyamat befejezésére. A metódus lépései a következők. Az opcionálisan megadott adatokat azaz, ha a data paraméter értéket tartalmaz, a metódus hozzáfűzi ezeket az adatokat a SID előtt. Ezután hozzáadja a 'Request Transfer Exit' SID-et a kérelemhez.

Végül a metódus kiírja az elküldendő UDS adatokat. Majd az így összeállított UDS üzenetet a sendUdsMessage metódus segítségével elküldi. A sendUdsMessage metódus választ vár vissza, így a requestTransferExit szintén szinkron módon fogja kezelni a kommunikációt. A requestTransferExit metódus szükséges a folyamatok megfelelő lezárásához az UDS kommunikáció során.

4.5. Tester

A Tester osztály a UDSonCAN osztállyal együttműködve végzi az UDS protokollon keresztüli frissítési tesztet. Az osztály konstruktora inicializálja az udsOnCan mutatót, amely az UDS kommunikációt kezeli.

A minimal_update metódus egy minimális frissítési folyamatot hajt végre. Első lépés a Kommunikációs Ellenőrzés, ahol Ellenőrzi, hogy az udsOnCan példány inicializálva van-e. Majd ezt követi a letöltési kérelem azaz a requestDownload metódus meghívása, amely az UDSonCAN objektumon hívódik meg és egy kérést küld a letöltéshez a cél felé. Sikertelen letöltési kérelem esetén a folyamat megszakad.

```
std::vector<std::vector<uint8_t>> Tester::parseHexFile(const
std::string& filename) {
    std::ifstream hexFile(filename);
    std::vector<std::vector<uint8_t>> dataBlocks;

    if (!hexFile.is_open()) {
        std::cerr << "Unable to open HEX file." << std::endl;
        return dataBlocks;
    }

    std::string line;
    while (std::getline(hexFile, line)) {
        std::vector<uint8_t> block;
        std::istringstream hexStream(line);
        std::string byteString;

        while (hexStream >> std::setw(2) >> byteString) {
            uint8_t byte =
static_cast<uint8_t>(std::stoul(byteString, nullptr, 16));
            block.push_back(byte);
        }

        if (!block.empty()) {
            dataBlocks.push_back(block);
        }
    }

    hexFile.close();
    return dataBlocks;
}
```

4.5. a kódrészlet

A 4.5. a kódrészletben a parseHexFile implementációja látszódik, amely HEX formátumú fájlokat olvas be, és azok tartalmát std::vector<std::vector<uint8_t>> formában adja vissza. Minden sor a fájlban egy adatblokkot képvisel.

Az adatblokkok átvitelére a transferData üzenet küldő metódus lesz segítségül, amely a beolvasott std::vector<std::vector<uint8_t>> típusú adatot feldarabolja soronként. Minden egyes sor egy adatblokkot képvisel, amit byte-ok vektoraként tárolunk és ezeket a blokkokat tudjuk isoTp üzenetekként elküldi. Minden adatblokk küldése után ellenőrzi a sikeres átvitelt.

```
std::vector<std::vector<uint8_t>> dataBlocks =
parseHexFile("Data.hex");

// Iterate over each block and send it
uint8_t sequenceNumber = 1;
for (auto& block : dataBlocks) {
    auto [transferSuccess, transferResponse] = udsOnCan-
>transferData(sequenceNumber++, block);
    if (!transferSuccess) {
        std::cerr << "Failed to transfer data chunk " <<
static_cast<int>(sequenceNumber) << std::endl;
        return;
    }
}
```

4.5. b kódrészlet

A 4.5. b kódrészlet egy for ciklus segítségével végigiterál az adatblokkokon. Minden iterációban növeljük a sequenceNumber változót, amely követi az egyes adatblokkok sorozatszámát. A udsOnCan objektum transferData metódusát hívjuk meg, amely a sorozatszámot és az aktuális adatblokkot használja az adatok elküldéséhez.

Az adatátviteli folyamat végén a requestTransferExit metódust hívja meg, amely jelzi a frissítési folyamat befejezését.

5. ÖSSZEFOGLALÁS

A C++ nyelv használata a szakdolgozat során különleges kihívást és tanulási lehetőséget jelentett számomra. A C++-ban való mélyebb elmélyülés révén fejleszthettem programozói készségeimet és jobban megértettem a komplex rendszerek működését. A Vector eszközökkel és könyvtárakkal való munka pedig tovább bővítette szakmai ismereteimet. Ezek az eszközök kritikus szerepet játszottak a CAN hálózati kommunikáció megvalósításában, és lehetővé tették számomra, hogy közelebbről is megismerjem a járműipari kommunikációs protokollokat. A Vector technológiák használata közben megtanultam, hogyan kell hatékonyan kezelni és elemezni a hálózati forgalmat.

Összességében a szakdolgozatom során szerzett tapasztalatok nemcsak a szakmai tudásomat bővítették, hanem a problémamegoldó készségeimet és kreativitásomat is fejlesztették.

Irodalomjegyzék

[1] DoCAN Standard: ISO 15765-4 2021 (https://www.iso.org/standard/78384.html)

[2] DoCAN Standard: ISO 15765-2 2016: (https://www.iso.org/standard/66574.html)

[3] ODX Standard: ISO 22901-1 2008: (https://www.iso.org/standard/41207.html)

[4] ODX Standard: ISO 22901-2 2011: (https://www.iso.org/standard/52669.html)

[5] UDS Standard: ISO 14229-1 2020: (https://www.iso.org/standard/72439.html)

[6] UDS Standard: ISO 14229-2 2021: (https://www.iso.org/standard/77322.html)

[7] UDS Standard: ISO 14229-3 2022: (https://www.iso.org/standard/77323.html)

[8] Vector VN1600 Manual: VN1600 Manual (vector.com)

[9] Controller Area Network: Introduction to the Controller Area Network (CAN) (Rev. B)

[10] Learning C++ eBook: c++ eBook (riptutorial.com)

Nyilatkozat

Alulírott Krucsai Gergő üzemmérnök-informatikus BProf. szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztési Tanszékén készítettem, üzemmérnök-informatikus BProf. diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

Dátum: Szeged, 2023. december 14.

Aláírás

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani mindazoknak, akik hozzájárultak a szakdolgozatom elkészítéséhez. Külön köszönet illeti Molnár Tamást, aki mint a külsős témavezetőm, rendkívüli szakértelmével és átfogó ismereteivel segítette elő a szakdolgozatom fejlődését. Hálás vagyok továbbá Dr. Alexin Zoltánnak, az egyetemi konzulensemnek, aki iránymutatásával és szakmai támogatásával hozzájárult a szakdolgozatom színvonalának emeléséhez.