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

Kanyó Bálint

DIAGNOSZTIKAI SZOFTVER AUTÓIPARI VEZÉRLŐEGYSÉGEKHEZ

KÜLSŐ KONZULENS

Dr. Balogh András

TANSZÉKI KONZULENS

Dr. Sujbert László

BUDAPEST, 2014

Tartalomjegyzék

Hallgatói nyilatkozat

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

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

Kelt: Budapest, 2014. 12. 17.

...…….
Kanyó Bálint

Összefoglaló

Az autóiparban mára már elengedhetetlen az elektronikus vezérlőegységek használata, és ezen egységek kontrollálása. Az egységeket fel kell tudni konfigurálni, ellenőrizni a konfiguráció helyes működését és diagnosztikát végezni rajtuk. A diagnosztikai szolgáltatásokat szabványokban foglalták össze, ilyen az UDS és az ODX. Az UDS ([1] ISO14229-1 2006 Road vehicles Unified Diagnostic Services) leírja az alapvető kommunikációs paramétereket, meghatározza az egyes diagnosztikai szervizek felépítését kérés és választ külön-külön. Megtalálhatók benne az általános parancsok konkrét értéke, aminek minden autóipari vezérlőegységen működnie kell. Az ODX ([2] ISO2291-1-ODX) szabvány egy leírási struktúrát határoz meg az UDS-ben leírt szolgáltatásokhoz. Ez a leírás XML formátumban történik. A feladatom egy olyan szoftver készítése, amely képes az XML leírás alapján diagnosztikai üzenetek létrehozására és válaszok értelmezésére.

Ezek a szabványok csak az üzenet összeállításáról gondoskodnak, a kommunikációs réteget is be kell mutatnom. A kommunikációhoz használt gateway (TKP Fieldbusz Gateway) egy Ethernet-terepbusz illesztő, mely leírását a dolgozatban részletesen bemutatok. A kommunikáció fizikailag CAN buszon történik. A CAN kommunikációs szabványát az [3] ISO 15765-2 2011 EN Road vehicles - Diagnostic communication over CAN Part 2 tárgyalja. Ez tartalmazza az elküldendő és fogadható CAN üzenetek felépítését.

A dolgozat második felében bemutatom az általam létrehozott szoftvert, azt ODX fájl feldolgozását az üzenetek összeállítását, és a fogadott válaszok értelmezését a kapott mintafájl alapján.

Abstract

Nowadays in the automotive industry has been necessary to the use of electronic control units and control of these units. The units should be able to configure, verify the correct operation of the configuration and diagnostics performed on them. The diagnostic services have been summarized in standards, these standards are UDS and ODX. The UDS ([1]ISO14229-1 2006 Road Vehicles Unified Diagnostic Services) describes the essential communication paremeters, determines the structure of each diagnostic service, request and response separately. It contains the concrete value of the general commands what works in all automotive industrial control units. The ODX ([2] ISO2291-1-ODX) standard determines a descriptor structure for the UDS's services. This description is written in XML format. My job is to make a software what could be able to create messages and interpret answers from diagnostic description.

These standards just describe about the collocation of messages, I have to present the communication layer. The gateway (TKP Fieldbus Gateway) what is used to the communication is an Ethernet-fieldbus gateway, what I present in details in the dissertation. The communication physically happens over CAN. The communication standard of CAN discussed in the [3] ISO 15765-2 2011 EN Road Vehicles – Diagnostic Communication over CAN Part 2. It contains the structure of sendable and receivable CAN messages.

In the second part of the dissertation I will present the software what I created, the processing of ODX file, the collocation of messages and the interpretation of the received answers from the gotten sample file.

1 Bevezető

A feladataim közé tartozik az UDS, ODX szabványok megismerése, a ThyssenKrupp Presta által fejlesztett Ethernet-Fieldbusz Gateway és ehhez tartozó API megismerése és egy grafikus felhasználói felülettel rendelkező szoftver elkészítése Eclipse fejlesztői környezetben, amely képes egyszerű, a kapott ODX leírásban megtalálható diagnosztikai egységek végrehajtására. A kommunikáció fizikailag CAN buszon történik. A CAN kommunikációs protokollját is meg kell ismernem, be kell építenem a programba. A szoftver implementálása Java nyelven történik. A szoftver tulajdonképpen egy plug-in, amelyet egy Eclipse platform futtat. A plugin egy beépülő szoftverkomponens, amely magában foglal egy komplexebb megvalósított szoftvert. A plugin tartalmazza a szoftver vázást és a grafikus felhasználói felület kezelését. A pluginon belül két nézetet kell létrehozni, egyet a diagnosztikai szervizek kiválasztásához és egyet a kiválasztott szerviz paramétereinek beállításához és a válasz megjelenítéséhez. Az ODX fájl feldolgozásához a konzulenstől kapott ODXLib_2.0.0 nevű apit használtam, ez megkönnyíti az XML formátum értelmezhetőségét. Az api osztályok és ezek függvényeinek gyűjteménye. Ez megtalálható a melléklet fájljai között. Első lépéseim között volt az előbb említett api megismerése is.

1.1 UDS szabvány

Az autóiparban a diagnosztikai funkciókat legjobban leíró szabvány az Unified Diagnostic Services szabvány. A mai autóipari fejlődést segítve hozták létre, hogy egységesítse az autóipari diagnosztika funkcióit és szolgáltatásait. A szabvány az OSI modell alapján osztja hét rétegre a diagnosztikát, egészen a fizikai rétegtől haladva az applikációs rétegig, ezeket a rétegeket is egy-egy szabványban rögzítették. Feladatom az applikációs réteg megismerése, a szabvány alapján.

1.1.1 A szabvány bemutatása

Az UDS szabvány definiálja a diagnosztika applikációs rétegét, ezen belül definiálja az alapszolgáltatásokat. A főbb szolgáltatásokat funkcionális egységekre szétválasztva definiálták. A főbb egységek közé tartoznak a szervizeket leíró egység, a diagnosztikai és kommunikációért felelős egység, adat átvitelért felelős egység, tárolt

adatok átviteléért felelős egység, I/O kontroll egység és egyéb adat fel és letöltésért felelős egység. Az egységek pontosan definiálják az összeállítandó és fogadandó üzenetek felépítését, még bájtokra, bitekre lebontva is. A szabvány engedi a főbb egységeken belüli üzenetek lényegi tartalmának szabad alakítását. Saját eszköz-specifikus parancsok is leírhatóak a segítségével. Ilyen parancsok lehetnek a vezérlőegységek belső állapotainak, szenzorok jeleinek kinyeréséhez vagy beállításához szükséges parancsok, ilyenek lehetnek még fordulatszámra, feszültségértékekre, kontaktusok állapotaira, időértékek, belső regiszterek tartalmára vonatkozó kérések. Az UDS-ben definiálnak biztonsági szinteket is, ezek a vezérlőegységek védelmére, a bennük lévő szoftver működésének védelmében hozták létre. Három biztonsági réteget definiáltak, ezek az alap (Default), programozási (Programming) és kiterjesztett (Extended). A vezérlők alapállapotban az alap biztonsági szinten vannak, ebben az állapotban, csak az alap diagnosztikai funkciók érhetőek el, ilyenek a hibakódok, regiszterek, szenzorok olvasása. A programozási szinten, már módosítható a vezérlő szoftvere, erre a szoftverfrissítések és újraprogramozások miatt van szükség. A kiterjesztett részben a programozás mellett más funkciók is elérhetőek lehetnek. Fontos megjegyezni, hogy az UDS szabványon belül az értékek hexadecimális, míg az ODX leírásban decimálisan vannak megadva.

1.1.2 UDS szolgáltatásai

A szolgáltatásokat a fent említett egységeken belül fogom bemutatni.

A szervizeket leíró egység:

Ez az egység írja le a kérések és válaszok általános felépítését, milyen adatokat kell tartalmazniuk, melyik bit és bájt mit tartalmazhat, mit jelent. Egy PDU két féle lehet alfunkcióval rendelkező vagy alfunkcióval nem rendelkező. Tartalmazza a forrás címét, a cél címét, a szerviz nevét, alfunkcióját, azonosítóját és paramétereit. Az alábbi ábra bemutatja egy alfunkcióval rendelkező PDU (Protocol Data Unit) felépítését. A szerviz alapfunkcióját a kérés azonosító, az első paraméter (Request Service Id) adja meg, ez határozza meg a kérés alapfunkcióját. A második paraméter lehet sima adatparaméter vagy a kérés alfunkcióját hordozó paraméter is.

A paraméterek alapján ismeri fel a vezérlőegység vagy a szoftver, hogy milyen kérdés/válasz érkezett, az tartalmaz-e alfunkciót és hogy melyek a fogadott paraméterek lényegi információi. A pozitív és negatív válaszok különböznek, a negatív válasz csak

egy vagy több NRC-t (negativ response code-negatív válasz kód) tartalmaz, ezek alapján lehet eldönteni a negatív válasz okát. A negatív válasz azonosítója mindig 0x7E, ez után következik a kérdés azonosítójú, majd a NRC. A visszaküldött kérésazonosító alapján mindig tudni fogjuk, hogy melyik kérdésre kaptuk a negatív választ. A pozitív válaszok és a kérdések felépítése hasonló, az adatok kinyerése és összeállítása hasonlóan megy végbe.

A_PDU parameter	Parameter name	Cvt	Hex value	Mnemonic
SA	Source Address	M	xx	SA
TA	Target Address	M	xx	TA
TA_type	Target Address type	M	xx	TAT
RA	Remote Address	C	xx	RA
A_Data.A_PCI.SI	<Service Name> Request Service Id	M	xx	SIDRQ
A_Data. Parameter 1	sub-function = [parameter]	S	 xx	LEV_ PARAM
Parameter 2 : Parameter k	data-parameter#1 : data-parameter#k-1	U : U	xx : xx	DP_...#1 : DP_...#k-1
C: The RA (Remote Address) PDU parameter is only present in case of remote addressing.				

ábra 1.1. PDU általános felépítése

ISO14229-1 2006 Road vehicles Unified Diagnostic Services

Diagnosztikai és kommunikációs menedzsment:

Ez a csoport az alapvető kommunikációs és diagnosztikai parancsokat foglalja össze. Az itt definiált szervizek az alábbi táblázatban olvashatóak. Az első oszlopban a szervíz kérés azonosítója (request service id), második oszlopban az elnevezése olvasható.

Hex	service
0x10	Diagnostic Session Control
0x11	ECU Reset
0x27	Security Access
0x28	Communication Control
0x3E	Tester Present
0x83	Access Timing Parameter
0x84	Secured Data Transmission
0x85	Control DTC Settings
0x86	Response On Event
0x87	Link Control

ábra 1.2 Diagnosztikai és kommunikációs menedzsment szervizei

Ez a rész definiálja a benne lévő szervizek azonosítókódját és az alfunkciók értékeit is. Ezek segítségével lehet nagyobb diagnosztikai rutinokat lebonyolítani, az ECU az utasítások egy részét csak biztonsági szintváltással oldható meg, egy jelszó megadásának segítségével lehetséges, ez a művelet is ezekkel a kérésekkel lehetséges. A paraméterek nevéből következtethetünk a szerviz funkciójára. Vegyük az ECU Reset parancsot, ez a parancs a vezérlőegység alaphelyzetbe állítására való, alfunkciók között lehet például a soft, hard reset vagy a gyújtáskapcsoló kikapcsolt állapotának szimulálása.

Adatátvitel:

Az adatátviteli csoport tartalmazza az adatok ki- és beviteléhez szükséges parancsokat, ezek a parancsok segítségével írhatunk, olvashatunk adatokat cím vagy azonosító alapján. A szabványban definiált parancsok az alábbi táblázatban olvashatóak. Ezen kérések nem rendelkeznek alfunkcióval, az alfunkciók helyett memóriacímek, azonosítók kerülnek az üzenetekbe, a válaszokba a kért című, azonosítójú adatok kerülnek. Ezen üzenetek szolgálnak az ECU-ban lévő szoftver kiolvasására, újraírására és a vezérlő egyes szenzorjainak vagy belső regisztereinek kiolvasására. A műveletek végrehajtásánál fontos, hogy a megfelelő biztonsági szinten kell lennie a felhasználónak, mert ellenkező esetben a kérésekre csak negatív válaszokat fogunk kapni.

hex	service
0x22	Read Data By Identifier
0x23	Read Memory By Address
0x24	Read Scaling Data By Identifier
0x2A	Read Data By Periodic Identifier
0x2C	Dynamically Define Data Identifier
0x2E	Write Data By Identifier
0x3D	Write Memory By Address

ábra 1.3 Adat átviteli csoport szervizei

Tárolt adatok átvitele:

Ebben a csoportban található a diagnosztikai hibakódok kiolvasása és törlése definiálva. Kettő alapvető üzenet típus van benne, egyik a kiolvasásra, másik a törlésre vonatkozóan. A kiolvasásnál meg lehet adni, hogy mely kódok kiolvasása szükséges, a hibakódok kiolvasása ennél fogva bonyolulttá válhat a lekérendő kódok szűrése miatt. A diagnosztika egyik legfontosabb egysége ez, mivel a diagnosztika fontos része a

hibák felderítése. A hibák kiderítéséhez a hibakódok kinyerése a kulcs, mivel a hibakódokból közvetlenül tudunk következtetni a hiba okára. A csoporton belüli két szerviz, csak az alfunkciók segítségével tudja lefedni a hibakódokra vonatkozó összes igényt. A hibakódok kiolvasásánál egy státuszbájt paraméter is definiálva van a hibakódok olvasáshoz. Ez a státuszbájt határozhatja meg a lekérendő kódok egyes csoportjait. A státuszbáj minden egyes bitjének funkciója definiálva van. A felhasználónak csak alap biztonsági szinten kell lennie ezeknek a parancsoknak a végrehajtásához, tehát szinte bárki törölhet hibakódot a gépjárműjéből a megfelelő eszköz segítségével.

hex	service
0x14	Clear Diagnostic Information
0x19	Read DTC Information

ábra 1.4 Tárolt adatok átviteli csoport szervizei

I/O kontroll:

Ez a csoport csak egy szervizt tartalmaz ez az Input Output Control By Identifier (0x2F). Ez a ki- és bemenetek vezérlését teszi lehetővé, a parancsazonosítók alapján tudja állítani a ki- és bemeneti portokat. A portokat egyszerre vagy külön-külön is lehet állítani az igényektől függően és itt is alfunkciók segítségével lehet váltani a parancsok módjai között.

Fel- és letöltés:

A fel- és letöltés utasításai az 1.1.5-ös ábrán láthatóak. Az utasításokkal lehet feltölteni új szoftvert a vezérlőegységre, az adott átvitelt menedzselni, elindítani, megállítani. Ezeknek a parancsoknak magasabb programozási vagy kiterjesztett biztonsági szintre van szükségük, hogy ne tudjon bárki belepiszkálni a rendszer működésébe.

hex	service
0x34	Request Download
0x35	Request Upload
0x36	Transfer Data
0x37	Request Transfer Exit

ábra 1.5. Fel és letöltés szervizei

1.1.3 Megvalósítása CAN-en

A kommunikáció fizikai csatornája jelen esetben CAN-busz. Az itt történő kommunikáció szabályait is egy szabvány, a CAN Transport Protocol (*ISO 15765-2 2011*) tartalmazza. A szabvány négy keretet definiál (Single Frame, First Frame, Consecutive Frame és a Flow Control Frame). A keretek használata az elküldendő vagy a fogadandó üzenetek hosszától függ. Egy CAN üzenet legfeljebb nyolcbájtnyi lehet. Ha csak egyetlen rövidebb üzenetet kell kiküldeni, vagy fogadni, akkor csak egy szimpla keretet (Single Frame) kell küldeni vagy fogadni. A hosszabb üzenetek kommunikációja bonyolultabb, valamilyen kontrollt és formát követel meg a küldőtől.

1.1.3.1 Transzport protokoll

A protokoll jól definiált és viszonylag egyszerű. A szabvány két lehetőséget hoz fel, egyet a rövidebb és egyet a hosszabb üzenetek küldésére, és kontrollálására. Ennek lebonyolítására négy keret áll rendelkezésre, ezek használatát mutatom be.

A szimpla keret (Single frame) egy rövidebb maximum hétbájtnyi üzenetet tartalmazhat, mivel az első bájtja adott. A keret első bájtja tartalmazza a keret azonosítóját (N_PCItype = 0) és a keretben lévő adatok mennyiségét. A többi maximum hét bájt magát az üzenetet tartalmazza.

Az első keret (First Frame) egy hosszabb üzenet első adatcsomagját tartalmazza, a lényegi tartalom ebben az esetben maximum hat bájt, mivel az első és második bájt tartalmazzák a keret azonosítóját és a küldendő adatok mennyiségét, ez maximum 4095 bájt. Az első bájt alsó fele és a teljes második bájt tartalmazz a hosszt, és az első bájt első fele a keret azonosítóját, ami jelen esetben 1. A többi adat Consecutive Frame sorozatos kiküldésével valósítható meg. A Consecutive Frame hétbájtnyi adatot tartalmazhat, az első bájtja a keret típusát (2) és a keret sorszámát tartalmazza, ez az elveszett adatok detektálására használható A küldés ütemét és a küldhető adatok mennyiségét mindig a fogadó fél szabja meg, ezt egy Flow Control üzenet elküldésével valósítja meg. A Flow Control egy hárombájtos üzenet, tartalmazza a keret azonosítóját és az adatok küldésének paramétereit. A paraméterek meghatározzák az egymás után küldhető blokkok számát, a keretek küldése közötti időszünetet és a folyamat státuszát, ezzel lehet jelezni, ha hiba volt. Az alábbi ábrán egy összefoglaló található a keretek felépítéséről.

N_PDU name	N_PCI bytes			
	Byte #1		Byte #2	Byte #3
	Bits 7 – 4	Bits 3 – 0		
SingleFrame (SF)	N_PCItype = 0	SF_DL	N/A	N/A
FirstFrame (FF)	N_PCItype = 1	FF_DL		N/A
ConsecutiveFrame (CF)	N_PCItype = 2	SN	N/A	N/A
FlowControl (FC)	N_PCItype = 3	FS	BS	STmin

ábra 1.6 Összefoglaló az üzenet keretekről

ISO 15765-2 2011

N_PCItype: network protocol control information type

SF_DL: single frame data length

FF_DL: first frame data length

SN: sequence number

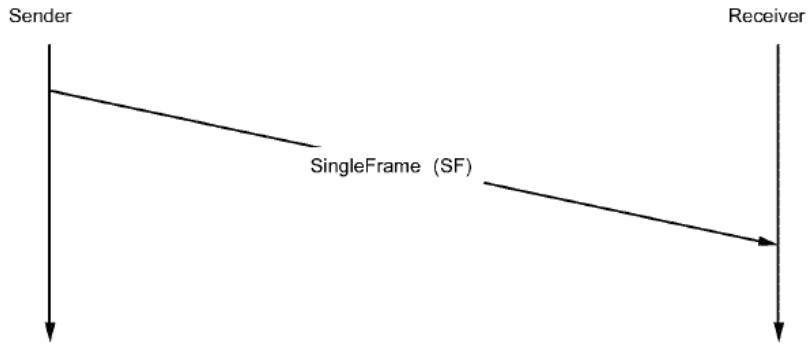
FS: flow status

BS: block size

Stmin: separation time minimum

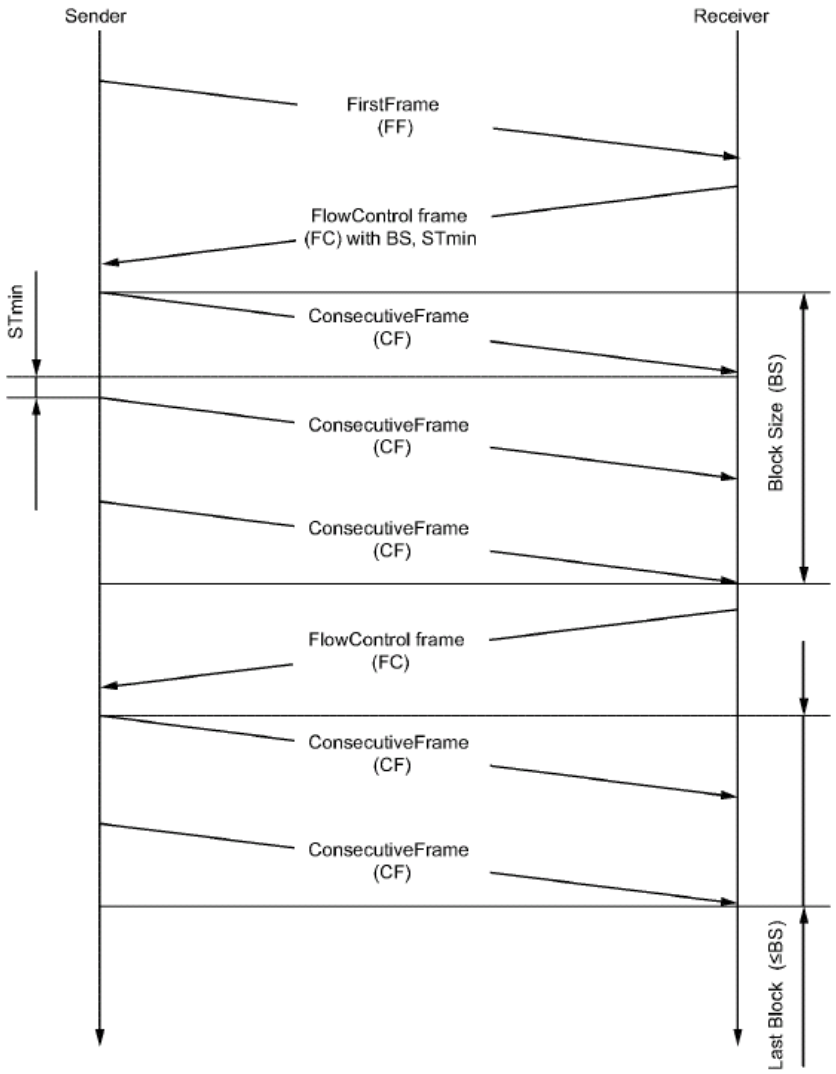
A kommunikáció kereteit már bemutattam, most az adatcsere folyamatát részletezem. Ha az üzenet hossza maximum hét bájt, akkor csak egy szimpla keretet kell kiküldeni, majd várni a választ. A válasz első üzenete vagy Single Frame vagy First Frame. A Single Frame után nincs más teendőnk, csak a kapott információ feldolgozása, ha First Frame jött, akkor ki kell küldeni egy Flow Controlt, amelyben megadjuk az egymás után küldendő üzenetek számát és a két keret között várandó minimális időt, ez után kapjuk a Consecutive Frameket a megadott Flow Control alapján. Ha bejött a kívánt mennyiségű Consecutive Frame, akkor újra Flow Controlt kell küldeni.

A szabványban diagramok is ábrázolják a kommunikáció menetét:



ábra 1.7 Single frame küldése

ISO 15765-2 2011



ábra 1.8 Szegmentált üzenet küldése és fogadása

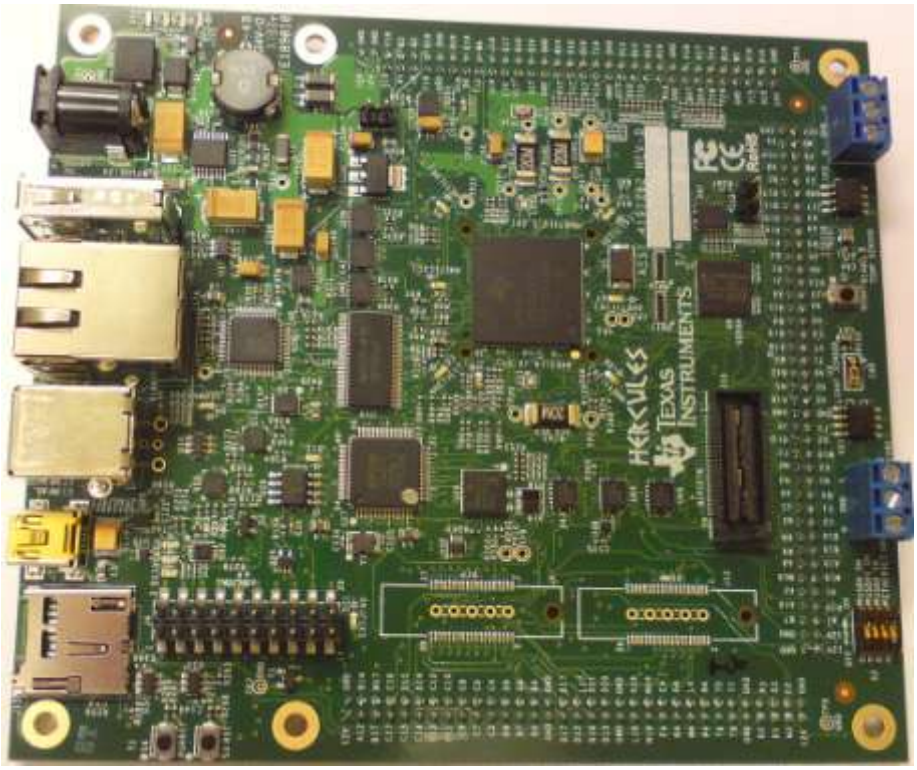
ISO 15765-2 2011

Az ábrákon láthatóak a folyamatok, és a keretek egymás utáni definiált sorrendje, az egyes paraméterek jelentése, jól látszik a szeparációs idő szerepe. A szeparációs idő a gyors küldő és lassú fogadó miatt kell a rendszerbe, mivel a túl hamar küldött üzenetet nem tudná a fogadó fél teljes egészében venni.

1.1.3.2 TKP Fieldbus Gateway

A gateway egy, a Texas Instruments által gyártott Hercules nevű fejlesztői kártyán fejlesztették, én is ezt a kártyát használtam, habár már létezik saját fejlesztésű panel is. A kártya tartalmaz egy TMX570LS típusú mikrokontrollert, ami biztonság specifikus kialakítású, a biztosabb működés érdekében, ez az autóiparban és egyéb biztonság specifikus alkalmazásokban fontos, emiatt két különálló ARM architektúrájú feldolgozóegységet tartalmaz. Három csatornás CAN és két csatornás FlexRay vezérlőt

tartalmaz. Számomra a CAN csatornák voltak fontosak, ezen kívül 10/100 Mbps Ethernet portot, SD kártya olvasót és más sok, az alkalmazás szempontjából nem fontos perifériát is tartalmaz. A fejlesztői panel már felprogramozva került hozzám, a rajta futó kódot nem ismerem, viszont a driver használatát később ismertem. A processzor adatlapja a www.ti.com oldalon megtalálható.



ábra 1.9. Texas Instruments Hercules panel

1.2 ODX szabvány

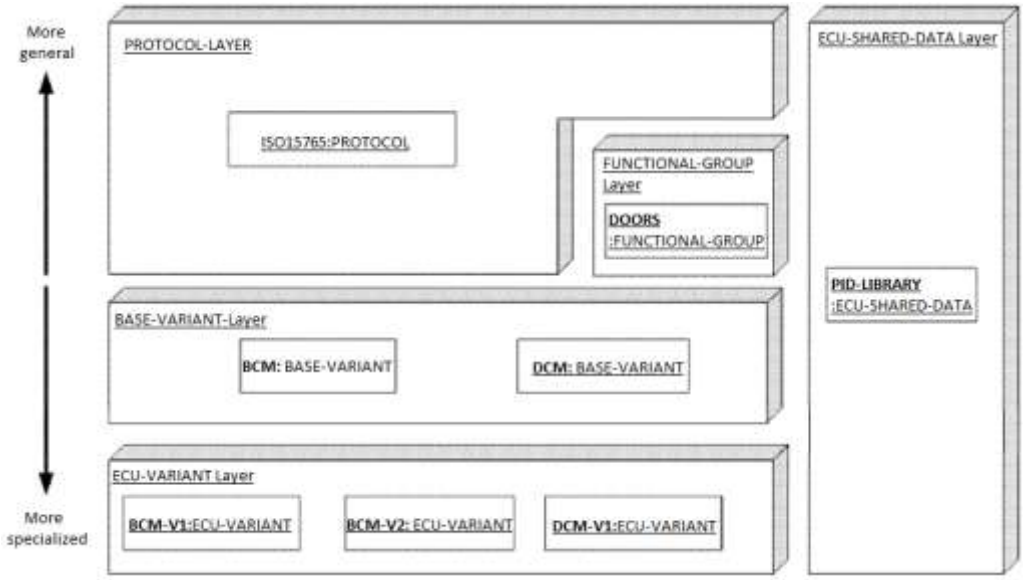
Az ODX szabvány az UDS szabványban definiált szolgáltatások leírására és egyéb diagnosztikához szükséges adatok tárolására szolgál. A szabványban rögzítik a leírás formátumát és az adatszerkezetét. A leírás maga XML formátumban történik. [4] Az XML (**Extensible Markup Language**, *Kiterjeszthető Jelölő Nyelv)* a W3C által ajánlott általános célú leíró nyelv, speciális célú leíró nyelvek létrehozására. Az SGML egyszerűsített részhalmaza, mely különböző adattípusok leírására képes. Az elsődleges célja strukturált szöveg és információ megosztása az interneten keresztül. Az XML-en alapuló nyelvek leírása formális, így lehetővé téve a programok számára a dokumentumok módosítását és validálását a formátum előzetes ismerete nélkül. Az XML leírás lehetővé teszi a diagnosztikai információk hordozhatóságát. A leírás egy hierarchikus leírás, azonosítókon keresztül hivatkozhatnak egymásra a benne lévő

egységek. Egy egység információit külön egységekbe rendezve találhatjuk meg. A hierarchikus rendszeren belül felülről lefelé haladva jutunk el a tényleges adatokhoz, struktúrákhoz. A diagnosztikai szoftverek különböző módon működhetnek, de az ODX leírást mindegyiknek ugyan úgy kell tudnia értelmeznie. Egy ODX fájlon belül több diagnosztikai egység is lehet, esetleg más-más vezérlők leírása is, erre azért lehet szükség, mert egy autóban több egyforma ECU is lehet, de ezek különbözhetnek paramétereikben.

1.2.1 ODX adatszerkezet

Az adatszerkezet hierarchikus, ezért kisebb és nagyobb egységekből áll. A leírásban szabványos neveket használnak, ezentúl ezeket fogom használni. A főbb egységek közül egyesek a kommunikáció leírására (pl.: PROTOCOL LAYER), míg mások a diagnosztika leírására (pl.: BASE-VARIANT, ECU-VARIANT) valók. A leírás tartalmazza a diagnosztikai rutinok mellett kommunikáció paramétereit is. A kommunikációs paraméterek között értem a kommunikáció fizikai megvalósítását (CAN, FlexRay, stb.), a kommunikáció sebességét, címezés módját és esetleges azonosítókat a sikeres kommunikációhoz. Az 1.10-es ábra tartalmazza a leírás blokkvázlatát. Az én projektemben a BASE-VARIAN és ECU-VARIANT rétegek a döntő fontosságúak. Az ezekben lévő információk kinyerése és felhasználása a cél.

Az általam használt leírásban az ECU-SHARED-DATA réteg nem volt benne és a FUNCTIONAL-GROUP információit sem tudtam hasznosítani. A számomra legfontosabbak a BASE-VARIANTS és az ECU-VARIANTS által hordozott információ, ezek tartalmazzák a diagnosztikai rutinok leírását, DIAG-SERVICE-ek formájában. A DIAG-SERVICE-ek hivatkozásokat tartalmaznak, melyek a hozzá tartozó kérés (REQUEST), válaszok (POSRESPONSE, NEGRESPONSE) azonosítóját tartalmazzák, ezek is a variánson belül találhatóak meg. Az XML leírás önmagában nem elegendő a leíráshoz, referenciák kellenek az egyes elemek kapcsolatához. Ezek a referenciák azonosítók, ezeken keresztül érhetjük el egy variáns hivatkozásait.



ábra 1.10 ODX leírás blokkvázlata

Egyes variánsok nem tartalmaznak szervizeket, csak egy hivatkozást a BASE-VARIANT szervizeire, tehát egy olyan variánsról van szó, amely olyan vezérlőt ír le, amely csak az alap tulajdonságokkal rendelkezik. Léteznek olyan variánsok is, melyek definiálnak saját szervizeket, és használják a BASE-VARIANT szervizeit is. Ez az autóban lévő egyforma, de más-más szolgáltatásokkal rendelkező vezérlőegységek miatt van így. Az autónak több ajtaja van, az első és hátsók különböznek és még a vezető oldali is különbözik a többitől, de az alap funkcióik egyformák. Az alap funkciók vannak a BASE-VARIANT szervizeiben leírva.

A DIAG-SERVICE azonosítókon keresztül hivatkozik a hozzá tartozó kérésre és a hozzá tartozó pozitív és negatív válaszokra, a variánsból kikeresve megkaphatjuk ezeket. A kérések és válaszok is paramétereket tartalmaznak.

A paraméterek között van előre meghatározott konstans értékű és olyanok, amelyek értékeit bizonyos határok között mi állíthatjuk be. Egyes paraméterek felépítése bonyolult, összetett, az ilyen paraméterek formáját struktúrában írják le, ezekre is azonosítókon keresztül hivatkoznak és a struktúrák is a variánsokon belül találhatóak meg. A paraméterek értékeinek meghatározása a DATA-OBJECT-PROP és DTC-DOP azonosítójú objektumok tárolják. A DOP-k a sima paraméterek leírását, a DTC-DOP-k a hibakódok tulajdonságait és értelmezéséhez szükséges adatokat tárolják

1.2.1.1 A leírás bemutatása egy példán keresztül

Ebben a részben bemutatok egy variánst (BASE-VARIANT), felépítését és tartalmát. Minden variánsnak van neve (LONG, SHORT-NAME), azonosítója és esetleg hivatkozás egy másik szülő variánsra. A lényegi rész a szülő és a saját adatai között található meg. Ezek az objektumok tartalmazzák a szervizeket, a pozitív, negatív válaszokat, a paraméterek beállításait.

A variáns rengeteg adatot tárol, ezek külön egységeken belül találhatóak meg. A variánson belül lévő lehetséges egységek az alábbiak:

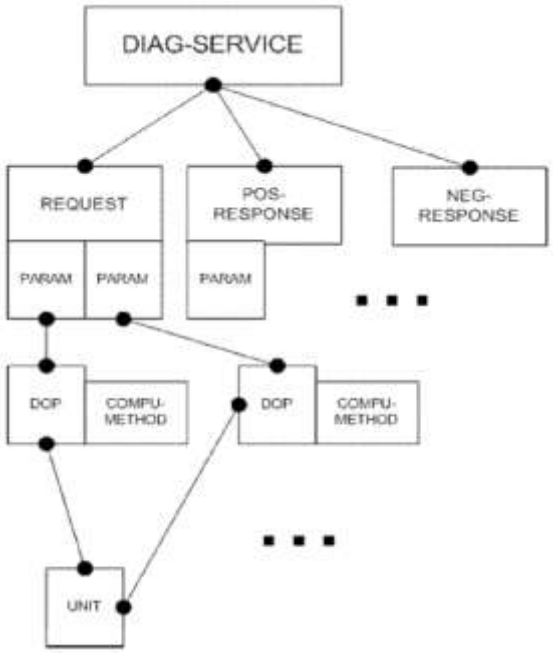
SHORT-NAME	rövid név
LONG-NAME	hosszú név
DESC	leírás
FUNCT-CLASSS	funkcionális osztályok
DIAG-DATA-DICTIONARY-SPEC	adatok tulajdonságai
DIAG-COMMS	diagnosztikai szervizek
REQUESTS	kérések
POS-RESPONSES	pozitív válaszok
NEG-RESPONSES	negatív válaszok
GLOBAL-NEG-RESPONSES	globális negatív válaszok
COMPARAM-FERS	kommunikáció paraméterei
PARENT-REFS	szülő referenciái

ábra 1.11. Variánsban lévő egységek

A fenti táblázat egyes elemei általános információkat tartalmaz, mint rövid név, hosszú név, leírás. A fontosabb diagnosztikai szervizeket a DIAG-COMMS sorolja fel, a diagnosztikai szervizek kérései és válaszai a REQUESTS, POS-RESPONSES, NEG-RESPONSES és a GLOBAL-NEG-RESPONSES tárolja.

Az 1.2.2-es ábrán egy szerviz felépítését láthatjuk. A szerviz három fő egységből áll kérés (REQUEST), pozitív válasz (POS-RESPONSE), negatív válasz (NEG-RESPONSE). A szerviz fő elemei is a variánson belül találhatóak meg a fenti táblázatban felsorolt, az elemnek megfelelő egységben. A szerviz maga nem, csak a variáns tartalmazza ezeket. A szerviz referenciákon keresztül hivatkozik ezekre. A fő elemek belső felépítése általában egyforma, csak tartalmi különbségek lehetnek. Az összes kérés, válasz paramétereket, azonosítókat tartalmaz. A paraméterek vagy, tartalmazzák az értéküket vagy egy azonosítón keresztül hivatkozik a tartalom leírására, ezek a DOP-k és DTC-DOP-k. A DOP-kon belül megtalálható a paraméterek lehetséges értékei vagy egy számítási mód (COMPU-METHOD). A számítási mód leírja, hogyan

kell belső programbeli, vagy külső kijelzendő értékeket kiszámolni egymásból. A DOP-k hivatkozhatnak UNIT-okra, ezek a kijelzendő érték SI mértékegységét tárolják. A felépítés ezen módon történik, a program egyik fontos része ezeknek a pontos feldolgozása.



ábra 1.12 Diagservice felépítése

ISO2291-1-ODX

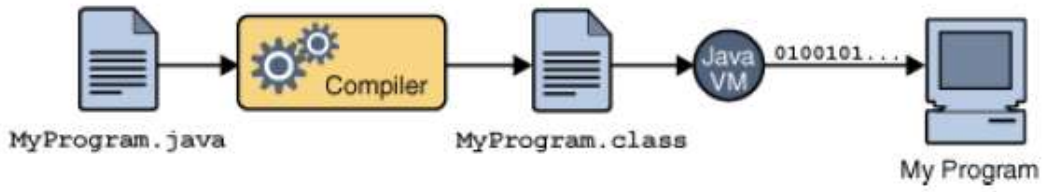
1.3 Eclipse

[5] Az Eclipse egy több platformon (Windows mellett Linux, Solaris, AIX, HP-UX, Mac OSX) rendelkezésre álló, több programozási nyelvet (Java, PHP, C/C++, stb.) támogató és többfajta fejlesztői környezetben (asztali alkalmazásfejlesztés, web fejlesztés, mobil alkalmazásfejlesztés, UML2 szerkesztés, vizuális szerkesztés stb.) alkalmazható nyílt forrású szoftverfejlesztő projekt.

Az Eclipse egy teljesen a felhasználó igénye szerint kialakítható felülettel rendelkezik (melyet perspektívának hívnak), egyszerű (fogd és vidd) felületen átpakolhatjuk a megnyitott ablakokat (nézeteket). Több felületet is kialakíthatunk magunknak az éppen aktuális munkától függően (programozás, HTML szerkesztés, stb.), melyek között a rendszer automatikusan is tud váltani, illetve mi is váltogathatunk. http://hu.wikipedia.org/wiki/Eclipse

1.3.1 A Java programozási nyelv:

[6] A Java egy magas szintű nyelv a következő főbb tulajdonságokkal: egyszerű, objektumorientált, előfordított, robosztus, biztonságos. hordozható, nagy teljesítményű, stb. A nyelv sajátja a hordozhatóság, tehát bármely platformon futtatható. A platformnak csak rendelkeznie kell a futtatáshoz szükséges keretrendszerrel a Java VM (Virtual Machine). Ez a konstrukció teszi lehetővé a kód hordozhatóságát a platformok között. A program futásához a Java kódból először Java bájtkódot generál a fordító, majd a bájtkódot a Java VM futtatja. A bájtkód értelmezése minden futtatáskor megtörténik.



1.13 Java kód futtatása

1.3.2 Alap tulajdonságok:

[5] Számos nézet áll rendelkezésünkre alapkiépítésben, illetve ezek listája beépülő modulokkal tovább bővíthető. Ilyen a szerkesztőmező és a projekt fájlok listája (navigator) mellett a problémák, tennivalók listája, a kivonat doboz (az aktuális állomány váza: XML szerkezet, vagy a függvények listája, stb.).

A beépített szerkesztő főleg Java és kapcsolódó kódszínezéseket támogat, de ezt a Colorer plugin segítségével kiegészíthetjük: így rengeteg nyelv áll rendelkezésünkre, illetve ha nem találunk egyet, XML leírók segítségével kibővíthetjük a rendszert.

Az Eclipse grafikus felhasználói felülete a Standard Widget Toolkitre (SWT) épül, ami rendhagyó, hiszen a legtöbb Java alkalmazás az Abstract Window Toolkitet (AWT) vagy a Swingeket használja. Az Eclipse felülete használja továbbá a JFace nevű köztes GUI réteget is, amely leegyszerűsíti az SWT alkalmazások készítését.

Az alkalmazásom fejlesztésénél én is az SWT-re támaszkodtam. Az Eclipse számomra a leghasználhatóbb fejlesztőkörnyezet, az iparban is ez az elterjedt fejlesztői környezet, a fejlesztők saját kiegészítőket pluginokat és nézeteket készítenek a saját munkájuk megkönnyítésére.

1.3.3 Alkalmazás fejlesztése

Eclipse alkalmazásfejlesztés elején el kell dönteni azt, hogy milyen alkalmazást akarunk létrehozni, ez nagyban függ a specifikációtól és a céltól is. Eclipse-ben több féle módon lehet programot létrehozni, lehet sima konzol applikációt, RCP applikációt, plugin applikációt stb. Az Eclipse tartalmaz minden programtípushoz segédanyagokat, egyes esetekben legenerálja a kód vázát, és nekünk csak a tényleges funkció megvalósításával kell foglalkoznunk. Ez nagy segítség lehet nagy és komplex programok létrehozásánál. A programozóknak van lehetőségük saját kódgenerátor létrehozására is.

1.3.4 Új funkciók bevitele

Ha egy program elkészült, de további lehetőséget akarunk a bővítésére, akkor pluginokkal, szkriptekkel vagy újabb osztályok implementálásával hozhatunk létre új funkciót. A programnak olyannak kell lennie, hogy képes is legyen az új funkciók befogadására. Az új funkciót csak hozzá kell csatolni a projekthez és minimális kódkiegészítéssel használatba kell azt venni.

2 A diagnosztikai szoftver

A fő feladat egy olyan rendszer létrehozása, amely képes felhasználói felületen keresztül diagnosztikai feladatok végrehajtására valamilyen autóipari elektronikus vezérlőegységen. A rendszer felhasznál egy ODXLib_2.0.0 nevű csomagot, amely képes az ODX fájl XML leírását feldolgozni, a programozók munkáját megkönnyítve. Az egyes ODX elemek külön osztályban elhelyezkedve lettek implementálva, a paraméterek egyszerű függvények használatával elérhetőek. A rendszer egy keret az API használatához. Tartalmaz a könyvtár egy komfort API-t is, ezt csak a szervizek megjelenítésére használt fa nézet felépítéséhez használtam. A többi funkciót magam készítettem el.

2.1 A szoftver főbb funkciói

A rendszer alapvetően több elemből épül fel. A rendszer része egy fa nézet, amely kilistázza az ODX fájlban fellelhető diagnosztikai szervizeket és társítja őket a hozzájuk tartozó vezérlőhöz. A fa nézet valamely elemét kiválasztva adjuk tovább a választott szerviz paramétereit egy szerkesztőnek.

A szerkesztő jeleníti meg a szerviz paramétereit az ODX fájl segítségével. A megjelenített paramétereket tudjuk szerkeszteni, ha szerkeszthető paraméterről van szó. Egyes paraméterek értékét listából kiválasztva adhatjuk meg.

Fontos feladat a kérés pontos összeállítása, ez is az ODX leírás alapján történik. Az ODX fájlban található bitpozíció, bájtpozíció és bithossz alapján. A kérés összeállítása nehézkes, bonyolult feladat. Később részletesen kifejtem.

A kész üzenet kiküldése a fentebb **Transzport protokoll** néven ismertetett fejezet alapján megy végbe. A keretek kiküldése, fogadása és összeállítása több osztályban lett implementálva, szétválasztva ezzel a fontosabb részeket. Az üzenet küldés és fogadás felügyelete szabványban definiált módon megy végbe.

A kommunikáció befejezése után a vett értékes információkat fel kell dolgozni és meg kell jeleníteni, ez is ugyanolyan komplex folyamat, mint a kérdés összeállítása. A kérdés és válasz egy szerkesztőn belül jelenik meg, baloldalon a kérdés, a jobboldalon a válasz.

A rendszer tartalmaz egy konzolt, amelyre az éppen kiválasztott, futtatott szerviz információit írathatjuk ki, itt megjeleníti a kimenő és bejövő üzeneteket. A konzol az Eclipse rendszerkonzoljához kapcsolódik.

A rendszer fontos része az Ethernet-terepbusz átjáróval való kommunikáló. A rendszerbe egy driver beimportálásával és használatával valósítottam meg. A driver a TI Herkules kártyával kommunikál Etherneten keresztül. A kommunikáció és a driver használata részletezve a 2.2 fejezetben található.

A saját kényelmem érdekében hoztam létre egy MyODX nevű osztályt. Az osztályba olyan függvényeket írtam meg, amelyek gyakran használt műveleteket valósítnak meg. Az osztály függvényei között van olyan, ami csak a variánsokat kérdezi le, de sok függvény végez kereső vagy lekérdező funkciót. A keresések általában azonosító alapján mennek, tehát az egyes szervizek és a bennük lévő paraméterek hivatkozásai által jelölt elemek könnyen, egy-egy függvény meghívásával lekérdezhetjük. A lekérdezendő elemek általában ne egyformák, így a különböző elemek lekérdezése külön függvényekben lett megvalósítva. Az osztályban lévő függvények felhasználják az ODXLib_2.0.0 nevű csomagot.

Az ODXLib_2.0.0 csomag a 2.0.0-ás verziójú ODX fájlok feldolgozására hozták létre. A könyvtár tartalmazza az összes, a diagnosztikai leíróban lévő elemek feldolgozásához szükséges osztályt és a hozzájuk szükséges függvényeket. A könyvtár elemei általában a szabványban megadott neveket kapták, eltérés a Java szintaktikai megkötések miatt lehet. A használata egyszerű a megfelelő elem lekérdezéséhez a megfelelő elemeken és függvényeken keresztül lehetséges.

2.2 Kommunikáció a vezérlőegységgel

A kommunikáció a számítógép és a vezérlőegység között egy köztes eszköz, úgynevezett átjáró segítségével megy végbe. Az átjáró felépítését már részleteztem, itt a hozzá tartozó driver használatának bemutatása a célom, utána bemutatom a CAN kommunikáció kereteinek létrehozását és a tényleges adatküldés és fogadás menetét. A tesztelés során egy USB-Ethernet átalakító is bekerült a rendszerbe, erre azért volt szükség, mert a számítógép saját Ethernet csatolójára szükség volt egyéb feladatok miatt és az USB-s átalakítót beállítva és felhasználva elérhető maradt a gép számára az internetkapcsolat.

A driver tartalmaz egy példakódot (CANExample.java), amellyel egyszerűen tesztelhető az átjáró működése. Ebben a kódban az átjáró két csatornáját fizikailag össze kell kötni, mivel itt önmagával folytat kommunikációt az egység. Első lépésem a kód értelmezése volt. A vezérlő egyszerű üzeneteket küld és fogad periodikusan, és kiírja ezeket a konzolra. Így lehet ellenőrizni a hardver működését. Az átjáró fix IP címet és portot használ (192.16.1.1:1234), így a számítógép IP címét ennek megfelelően kell beállítani. A gépi IP cím bármi lehet ebben az alhálózati címben, csak az átjáró címe nem, mivel egy hálózaton belül egy címet csak egy eszköz birtokolhat.

A rendszerben az előző példakód alapján hoztam létre egy saját osztályt (BlockingCommunication), amely felelős a fizikai kommunikáció lefolytatásáért. Az osztály tartalmazza a kommunikáció fizikai paramétereit (sebesség, CAN címek). A cimek fixen vannak beállítva, de egy egyszerű függvénnyel átállíthatóak lennének. Az osztály inicializációs függvényében csatlakoztatjuk a drivert, beállítjuk az egyes csatornát a megfelelő paraméterekkel, amelyek kellenek a kommunikáció végrehajtásához. A fogadást és küldést külön kell beállítani. A kommunikáció sebesség a jelen esetben 500 kbaud/s küldő és fogadó oldalon is, mivel a küldés és fogadás a CAN protokollnál egy csatornán megy végbe. A CAN busz adatszórásos technológia, mindenki tud venni minden adatot, de felhasználni nem kötelező azokat.

A kommunikációhoz két függvényem van, egyik a választ megvárva visszaadja azt, a másik csak kiküldi az adatot. Az eső megoldás szemafort használva állítja meg a program továbblépését, de ha nem érkezik válasz időben, akkor folytatja futását és nullát ad vissza. Az előbbi megoldás egyszerű és hasznos, mivel nem kell külön foglalkozni az adat fogadásával, egy függvény elintézi a kettőt. A vezérlő folyamatosan küld üzeneteket, ezét fontos az, hogy a számunkra fontos üzenet kerüljön a kimenetre. Az üzenet fogadásához implementálni kell a driver egy interfészét (CANDriverListener), a driver példakódja ezt is bemutatja. Az interfész tartalmazza a tényleges adat küldés és fogadás függvényeit, a saját függvényeim ezeknek a függvényeknek a segítségével fogadják az adatokat. Az üzenetet a CANRxIndication függvény nyeri ki és felszabadítja a szemafort, ha az megérkezett az adott címen. A szemafor lefoglalásánál fontos tudni, ha lefoglaljuk a szemafor, akkor a szál futása megáll. A szálat túl sok ideig lefoglalni nem szabad, mert ekkor a program további futása bizonytalan lehet, ezért használtam egy idő után önmagát feloldó szemafort, erre azért van szükség, mert a túl sűrűn küldött kéréseknél a vezérlő nem biztos, hogy tud

válaszolni minden kérésre, és a szemafor feloldatlan maradhat. A kiküldött adatot is lehet ellenőrizni, erre a CANTxConfirmation függvény való, ezt a funkciót nem használtam ki, nem volt rá szükségem.

A kommunikáció megkezdése előtt inicializálni kell a csatornát, itt kell megadni a sebességet, címeket, a cím típusát, a címekhez tartozó maszkot. Az alábbi kódrészlet mutatja be az egyes csatorna beállítását CAN üzenet fogadására a megadott címen standard CAN ID-val. A driver sok esetben saját változókat használ a megszokottak helyett, ezekre a szabványban leírt változófélék miatt van szükség, ilyen például az UINT32. A java nem használ előjel nélküli integereket, így létre kellet hozni az ezt kezelő típusokat.

```
CANRxMessageConfiguration rxMsgConfig = new CANRxMessageConfiguration();
rxMsgConfig.setParameter(CANRxMessageConfigurationEnum.canId,new
Uint32(RX_ID)); // CAN ID beállítása
rxMsgConfig.setParameter(CANRxMessageConfigurationEnum.canIdMask,new
Uint32(0x7ff)); //  CAN ID maszk beállítása
rxMsgConfig.setParameter(CANRxMessageConfigurationEnum.channel,
ECANChannel.CHANNEL_1); //csatorna beállítása
rxMsgConfig.setParameter(CANRxMessageConfigurationEnum.frameType,
CANFrameType.STANDARD); // standard CAN ID használata
rxMsgConfig.setParameter(CANRxMessageConfigurationEnum.rxIndication,
EBoolean.TRUE); // kérünk az üzenetről információt
```

Az inicializálás közben kapcsolódik a program az átjáróhoz, ezt a connect függvény valósítja meg. Itt kell megadni az átjáró címét és a portját. A párbeszéd lefolyása után le is kell zárni az átjárót, ez a close függvénnyel valósítható meg, ebben a függvényben a driver disconnect függvénye hívódik meg. A driver kapcsolás esetén hiba léphet fel, ekkor egy kivételt dob a connect függvény, a hibát kezelni kell. A hiba fellépésének több oka lehet, hogy már foglalt az átjáró, lefagyott esetleg nem található. A fejlesztés során is előfordultak ilyen kivételek, de az átjáró újraindításával megoldható volt a probléma.

A CAN kommunikáció a fent említett CAN Transport Protocol szabvány szerint megy végbe. A megfelelő kereteket össze kell állítani a kiküldendő adatokból. Az adatok mennyisége határozza meg, hogy milyen kereteket kell létrehozni és kiküldeni a vezérlőegységre. A hat bájtnál kevesebb adatot tartalmazó üzenetnél elég egy sima keretet (Single Frame) kiküldeni, de ha ennél több a kiküldendő adat, akkor már bonyolódik a helyzet. A protokoll szerint kell a megfelelő kereteket kiküldeni.

A keretek előállításához és az azokban lévő adatok kinyerésére egy osztályt (PDUFrame) hoztam létre. Ebben az osztályban olyan függvények találhatóak, amelyekkel létre lehet hozni a kereteket és ki lehet nyerni a fogadott keretek adatait, amelyeket az 1.1.3-as fejezetben ismertettem. Az osztály függvényeit más osztályok használják fel. A PDUFrame osztály csak függvényeket tartalmaz, a függvények a különböző keretek összeállítását és az adatok kinyerését végzik. A keretek összeállítására négy függvényem van a négy keretnek megfelelően. A függvények a keretnek megfelelő mennyiségű adatot (integer tömb formájában) és a keretben lévő további, a keret részét képző adatot várnak a bemenetükön. A fogadott keretek feldolgozása is az osztály függvényei végzik. A keretekből nem csak a lényeges adat, hanem a keretezéshez tartozó információk is kinyerhetőek.

A tényleges kommunikációt a CANcom osztály valósítja meg. Ez az osztály felhasználja a PDUFrame és a BlockingCommunication osztályokat, melyek tartalmazzák a kommunikációhoz szükséges egyes függvényeket. Elég beadni neki a nyers üzenetet, egy integer tömb formájában és létrehoz egy integer tömböket tároló listát, mely tárolja a kiküldendő kereteket, ezt az init függvény valósítja meg. Az inicializálás után ki lehet küldeni az üzenetet. Az üzenet kiküldése után egyből fogadja is a választ. A válasz hosszának megfelelően a CAN protokoll szerint fogadja az adatokat. A fogadott adatokat is egy integer tömböket tartalmazó listába helyezi. Az itt eltárolt adatokból még ki kell nyerni a lényeges, keret nélküli adatot. Az adatok kinyerésére szolgáló függvény (disFramer) is a PDUFrame osztályban található. A függvény megvizsgálja, milyen keretről van szó és a lényeges információval tér vissza, függvény leveszi a keret CAN kommunikációhoz szükséges részét.

2.3 Az üzenet

Az üzenet összeállítása az egyik legösszetettebb része a feladatnak. Ennél a feladatnál szinte az összes a diagnosztikai szervizeket leíró információra szükség lehet. Egyes információk készen definiálva vannak az ODX leírásban, ezek lehetnek konstansok, mint a szervizazonosító és a szerviz alfunkció (subfunction). De a többi információ vagy tetszőlegesen, vagy a meghatározott értékek között megadható. Néhány megadott érték szöveges formában értelmezett és ebben a formában is van a kijelzőn, ezt át kell konvertálni számszerű adatra.

Az adatok hossza is változhat, lehet néhány bit vagy néhány bájt hosszúságú is. A különböző hosszúságú adatokat össze kell fűzni bájtos hosszúságú adatokra. A szervizazonosító általában nyolcbites, de az alfunkció lehet nyolc vagy tizenhat bit hosszúságú is, ez a szerviz funkciójától függ. A tizenhatbites alfunkció általában egy regiszter vagy memória címet tartalmaz, ilyenkor valami belső információ lekérése megy végbe.

2.3.1 Az paraméterek és értékek tárolása

Az adatokat és a paramétereket egy Pbox nevezetű osztály elemei tárolják. A Pbox osztály tartalmaz egy Widget és egy MyParam nevű elemet. A Widget típusú elem tartalmazza a grafikus felhasználó felületen megjelenített Text vagy Combo nevű elemeket. A Text típusú elemekben az egyszerű szám értéket tartalmazó paraméterek kapnak értéket. A Combo típusúakban az előre definiált értékek közül választhatunk, általában szövegesen megadott értékek közül (pl.:true/false).

A MyParam nevű osztály magát a paramétert és a hozzá tartozó adatbeállításokat tartalmazza. Megtalálható benne a paraméter bit, bájt pozíciója és a bithossz is, ezek fontosak az üzenet összeállításánál, hogy a megfelelő paraméter a megfelelő helyen legyen az üzenetben. A Myparam minden egyes belső változóját írhatjuk, olvashatjuk tetszés szerint, így menet közben is módosíthatóak. A menet közbeni módosítás a bájpozíció szempontjából fontos, mert a paraméter struktúráján belül sokszor csak a nullás érték található, tehát ez nem használható. A paraméter bájtpozícióját a kérés paraméteréből kell venni, praktikusan a fő paraméter bájtpozícióját kell használni.

A paraméterek információinak tárolása egy listában történik, minden paraméter egy újabb eleme a listának. A lista a Pbox elemeit tárolja a feldolgozás sorrendjében. A feldolgozás sorrendje nem gond, ez az ODX leírás sorrendje. Az ODX leírás sorrendje nem biztos, hogy a kiküldendő üzenetnél is megfelelő, ezért kell a paraméter pontos behatárolása az üzeneten belül. A behatárolás a bitpozíció, bájtpozíció és bithossz alapján meg végbe, ezért fontos ezeknek a pontos tárolása.

2.3.2 Az üzenet összeállítása

Az üzenet összeállítása bonyolult és összetett feladat. Az üzenethez fel kell használni az ODX leírás adatait és a felhasználói felületen megadott információkat is.

Az előző részben leírtam a paraméterek tárolásának módszerét. A Pbox elemeket tároló lista segítségével állítom össze az üzenetet. A lista tartalmazza az ODX leírás fontos elemeit és a paraméter beállított értékét is. Tehát az információk felhasználásával kell létrehozni a kiküldendő üzenetet. Az üzenet összeállítása egy külön osztályban van implementálva, az osztály neve RequestBuilder.

A RequestBuilder osztályban lévő getMessage függvény segítségével lehet a kiküldendő konkrét üzenetet előállítani. A függvény bemenete a fentebb leírt Pbox lista. A lista elemein végigfutva kiszámolja az üzenet hosszát. A hossz alapján létrehoz egy karaktertömböt, a tömbben egyes és nullás értékek helyezkednek el a feltöltése után. A karaktertömböt a paraméterek bináris értékeivel töltöm fel, végigfutva a lista elemein. A feltöltés után a karaktertömbből egy integereket tartalmazó tömböt állítok elő. (Az átjáró drivere is integer tömböt használ.) Minden egyes bájtból egy integer lesz előállítva. A módszerem kissé kezdetleges, de ez volt számomra praktikusan használható. Az üzenetek összeállításánál fontos, hogy a paraméter értékét sok esetben át kell konvertálni külső értékről belső értékre. A szabvány leírása erről is gondoskodik, tartalmaz leírást az átváltásokhoz. A komfort API tartalmaz függvényt ennek megoldására, ez a függvény a DOP-t és a paraméter értékét várja a bemeneten. A paraméter átváltásának módja a DOP-ban van definiálva, a COMPU-METHOD tartalmazza az adatokat.

Az üzenet csak a lényeges, a kérés által definiált adatokat tartalmazza. Az adatok kiküldéséhez még kell a keretek összeállítása és a küldési protokoll végrehajtása, ennek módját már ismertettem. A függvény által használt függvények megtalálhatók az osztály függvényei között.

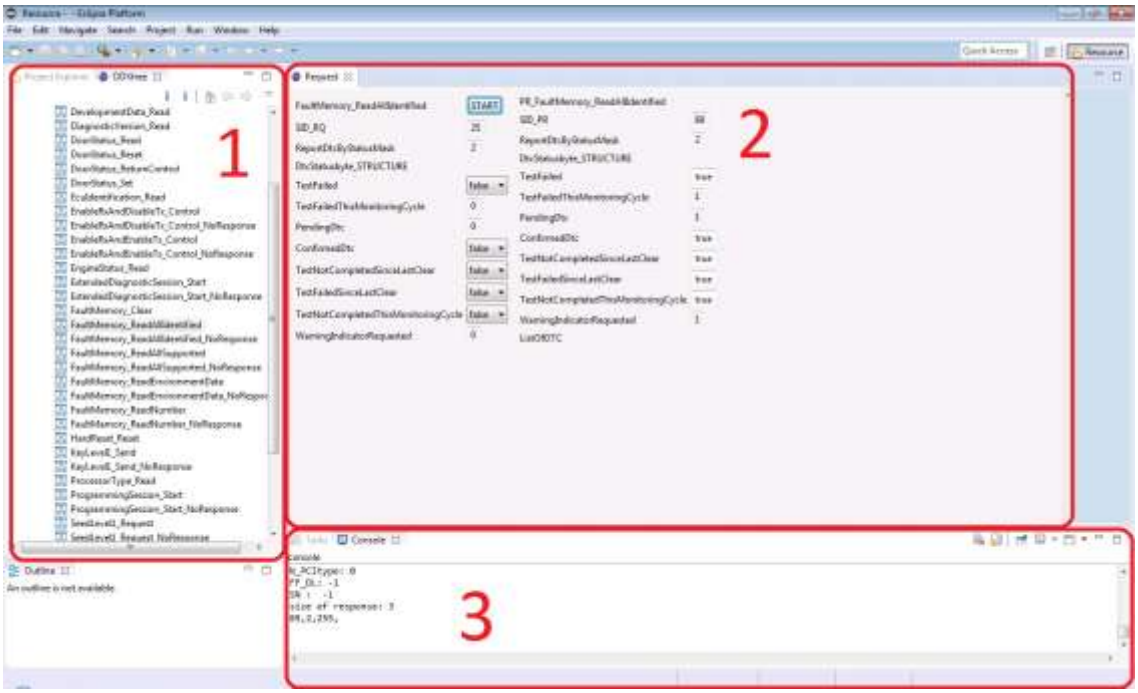
2.4 A GUI felépítése

A grafikus felhasználói felület fontos része a felhasználói programoknak, ezzel teremt kapcsolatot a felhasználó a programmal. A felhasználói felületen adhatunk meg paramétereket, indíthatunk el más programokat, szerkeszthetünk fájlokat, stb.

Az Eclipse nem a Swing-et vagy az AWT-t használja, hanem az IBM által fejlesztett SWT-t (Standard Widget Toolkit), ez nem része a szabvány Java API-nak. A saját grafikus felületem is az SWT-t használja és az Eclipse-ben megtalálható mintakódokat, melyekkel nézeteket és editorokat lehet létrehozni. Az SWT használatához rengeteg segítség található meg az Eclipse honlapján

(https://www.eclipse.org/swt/), itt leírást és mintakódot is találunk az SWT-ben található építőelemekhez.

A programom felhasználói felülete három részre osztható. Található benne egy fa nézet, amely az ECU-kat és a hozzá tartozó szervizeket listázza ki, tartalmaz még egy szerkesztőt, amely a szerviz kérését és a kapott választ jeleníti meg, és van benne egy konzol, amely információkat jelenít meg az éppen kiválasztott szervizről és a folyamatról. A felhasználói felület a **2.1. ábrán** látható.



ábra 2.1. Felhasználói felület: 1 fa nézet, 2 szerkesztő, 3 konzol

2.4.1 Fa nézet a szervizekhez

A fa nézet az ODX fájlban lévő variánsok szervizeit sorolja fel, az egyes vezérlők alatt. A nézetből lehet kiválasztani azt a szervizt, amelyet végre akarunk hajtani. A fa egyik elemén duplakattintásra nyitja meg a szerkesztőt. A fa nézet információkat ad át a szerkesztőnek, ami alapján megkeresi a diagnosztikai szervizt és felépíti a kérés felületét.

A fa nézet létrehozásánál az Eclipse-ben található fa nézet mintát használtam fel, ez létrehozza a kód vázát és egy kis példakód is található benne, ami segít a fa nézet használatának elsajátításában. A fa elemeit két osztály implementálja, egyik TreeObject, másik TreeParent, a TreeObject a TreeParent őse. A TreeObject a szervizeket, a TreeParent a vezérlőegységeket tartalmazzák. Minden TreeObject szülője egy

TreeParent, tehát minden TreeObject egy TreeParent gyermeke, de egy TreeParent lehet gyermeke egy másik TreeParentnek is. Kell lennie egy legfelső szintek, ez a gyökére a fának, a fa felépítésének végén a fát be kell állítani a gyökér gyermekének.

A fa felépítését a treeBuilder függvény végzi. A fa felépítésénél felhasználtam az ODXLib komfotAPI-ját. Az API használata egyszerűbbé teszi az ODX leírás könnyebb értelmezését, egyszerű függvények segítségével gyorsan ki lehet listázni a vezérlőegységeket és a szervizeiket.

A nézetnek információt kell átadnia a szerkesztőnek, ami a kiválasztott szerviz paramétereit listázza ki és teszi lehetővé a szerkesztését. A szerkesztő az egyik szervizen való duplakattintással nyitható meg. A duplakattintás funkcióját egy listener valósítja meg. A Java nyelvben minden eseményt listenerekkel kell lekezelni, ilyenek lehetnek az egér mozgatása, billentyű lenyomása vagy bármi más, ami eseményt generálhat.

Az információt az editor megnyitásakor adjuk át a szerkesztőnek. A szerkesztő bemenete egy objektum, ami tárolja a szerviz feldolgozásához szükséges információkat. Az objektum egy ODXEditorInput nevű osztály. Az osztály tartalmazza a szerviz nevét, a vezérlő nevét és az ODX leírást. Az ODXEditorInput osztály egy leszármazottja a NullEditorInput osztálynak, ez egy alapvető osztálya a szerkesztőknek, ez az ősosztálya az összes szerkesztő bemenetének.

```
public class ODXEditorInput extends NullEditorInput implements
IEditorInput {
     private String service = null;
     private ODXDocument ODX= null;
     private String parent = null;
     .
     .
     .
```

2.4.1.1 A fa felépítése

A fa felépítéséhez csak a vezérlőegységekhez tartozó szervizeket kell kilistázni. Egyes variánsok nem tartalmazzák az összes szervizt, amit használnak ezeket a szervizeket a BASE-VARIANT tartalmazza, tehát össze kell fűzni a variáns és a BASE-VARIANT szervizeit. Első lépés a variánshoz tartozó faelem létrehozása, majd a variánsban lévő szervizek listázása. A variáns feldolgozása után meg kell nézni, hogy van-e hivatkozás a BASE-VARIANT szervizeire. A BASE-VARIANT szervizei között ott lehetnek a vezérlőben definiált szervizek is, de ezeket már nem kell újra hozzáadni,

mivel a variánsban benne voltak az aktuális vezérlőhöz specifikusan létrehozott szervizek. Így a variánsok szervizei a saját és a BASE-VARIANT szervizeiből állhatnak össze.

A nézeten belül történik meg az ODX fájl megnyitása és átalakítása is az ODXLib elemei számára, a könyvtár függvényeinek segítségével. Az ODX fájlt az alábbi két sorban nyitom meg és alakítom át:

```
File ODX = new File("./workspace_ODX_se/UDS-ExampleEcu.XML");
ODXDocument docODX = ODXDocument.Factory.parse(ODX);
```

A docODX objektum típusa ODXDocument, ami a felhasznált könyvtár sajátja, ez az objektum tartalmazza a leírás struktúráját és a benne lévő összes információt. Az információk kinyerése az ODXLib függvényei segítségével megy végbe. Az ODX információ kinyerése egyszerű, itt egy példa:

```
docODX.getODX().getDIAGLAYERCONTAINER().getECUVARIANTS();
```

Ez a sor mutatja be, hogyan kell kinyerni az összes ECU-VARIANT objektumot az ODXDocument objektumból. A variánsok kinyerése lehet listában is, ilyenkor egy List típusú elemmel tér vissza, ami ECU-variánsokat tartalmaz.

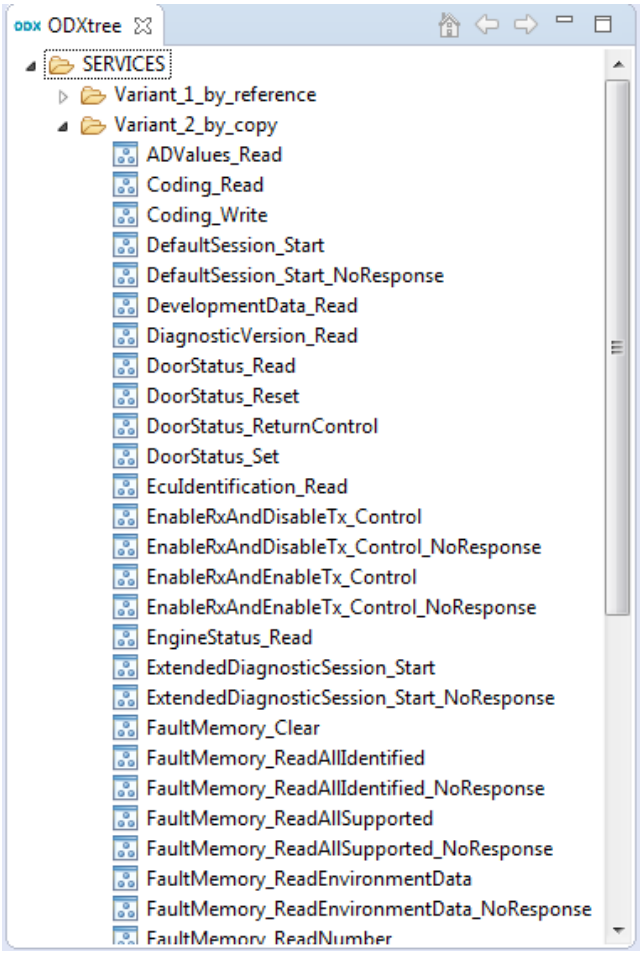
2.4.1.2 A kód:

A függvény első néhány sora egy ICoProject objektumot hoz létre, amihez hozzá adja az ODX leírást. A DIAGLAYER-ek tartalmazzák a variánsokat, ezek lekérdezése után egy ciklusban végigmegyünk az összes ilyen elemen. A ciklus belsejében teszteljük a variánst, hogy egy ECU leírását tartalmazza-e. A vezérlők variánsaiból kerülnek elő a szervizek, a ECU-VARIANT szervizei után a külső szervizek következnek. A külső szervizek a BASE-VARIANT szervizei között lehetnek. Az első lépés az, hogy megvizsgáljuk a variáns hivatkozik-e külső szervizekre. A külső szervizek esetén meg kell vizsgálni a szerviz meglétét az ECU-VARIÁNS szervizeiben, a fellelt szervizek nem kerülnek be a fába kétszer. A függvény lefutása után a beadott gyökér (root) fogja tartalmazni a fát, az a gyökér lesz a gyermeke a kirajzolt fa alapelemének. A fa kirajzolásával nem kellet foglalkoznom, mivel erről a fa nézet generált függvényei gondoskodnak.

```
private void treeBuilder(ODXDocument docODX, TreeParent root) throws
CoInvalidParameterException {
ICoProject proj = ICoProject.Factory.newInstance();
     proj.addODX(docODX);
     DIAGLAYER[] dlayers = proj.getDiagLayer();
```

```
        for (DIAGLAYER d : dlayers) {
             ICoDiagLayer dlay = proj.getICoProvider().get(d);
        if   (d.getClass().getSimpleName().equalsIgnoreCase("ECUVARIANTImpl"))
{
             TreeParent parent = new TreeParent(d.getSHORTNAME());
             root.addChild(parent);
             if (d.getDIAGCOMMS() != null) {
             for (DIAGSERVICE dc : d.getDIAGCOMMS().getDIAGSERVICEList()) {
                  TreeObject leaf = new TreeObject(dc.getSHORTNAME());
                  parent.addChild(leaf);
                            }
                       }
        if (dlay.getDiagLayerByInheritance(true, false) != null) {
             DIAGLAYER[] inher = dlay.getDiagLayerByInheritance(true, false);
             for (DIAGLAYER ind : inher) {
                  if (ind.getDIAGCOMMS() != null) {
for (DIAGSERVICE indc : ind.getDIAGCOMMS().getDIAGSERVICEList()) {
             if (isElemetOfTree(indc.getSHORTNAME(), parent) == false) {
             TreeObject leaf = new TreeObject(indc.getSHORTNAME());
             parent.addChild(leaf);
                                          }
                                     }
                                }
                           }
                      }
             }
        }
```

Az alábbi ábrán látszik a fa felépítve egy külön ablakban, nézetben. Az ODX leírás két ECU adatait tartalmazza, a fában látszik a két variáns. Egyik referenciával hivatkozik a szervizeire, a másikban csak a BASE-VARIANT szervizeinek egy része van bemásolva. A második variáns alatt találhatóak a benne és a BASE-VARIANT-ból kapott szervizek, mivel saját és hivatkozott szervizek is vannak benne. Az ODX leíróm egy mintaleíró, így minden lehetőséget fenntartva hozták létre. Az összes lehetséges kapcsolat meg volt benne, ami BASE-VARIANT és ECU-VARIANT között lehetséges.



ábra 2.2. ODX leírás fa nézete

2.4.2 Szerkesztő

A szerkesztőt is az Eclipse editor sablon segítségével hoztam létre, hozzá kellett adni a plugin projekt kiterjesztései közé. Az Eclipse egy sablon alapján legenerálja a szerkesztő vázát, ami a legszükségesebb osztályokat, implementációkat tartalmazza. A megfelelő működéshez meg kell adni a szerkesztő bemenetét is, ez a fa nézetben lesz átadva a szerkesztőnek. A szerkesztő bemenete az ODXEditorInput nevű objektum, ezt az objektumot már leírtam a fa nézetbél.

A szerkesztő jeleníti meg a kiválasztott szerviz kérését, és a kérés paramétereit. A paramétereket egyenként jeleníti meg egymás alatt. A struktúrával rendelkező paraméterek feldolgozása is nehéz feladat volt, ki kell keresni a megfelelő struktúrát és fel kell dolgozni azt és a paraméterekhez tartozik adatbeállítás, ezt is figyelembe kell venni.

Egyes paraméterek megjelenítése más és más lehet. Egyes paraméterek szám formában vagy szöveges formában is megadhatók. A szöveges paramétereknél meg van

határozva, melyek lehetnek a választható értékek. A szöveges paraméterek és ezek konkrét értékei egymás alatt, egy egységen belül foglalnak helyet. A numerikus és nem numerikus értékeknél az értéke korlátai is meg vannak adva.

A fogadott válasz megjelenítésekor a feladat ugyanaz, csak a kijelzett értéket nem a felhasználó, hanem program adja meg a fogadott adatokból kinyerve azt.

2.4.2.1 A szerkesztő grafikus felülete

A szerkesztő grafikus felületét úgy alakítottam ki, hogy az két fő részből álljon össze. Első lépés a felület főelemének úgynevezett composite-nak a létrehozása és beállítása. A composite-ok segítségével lehet struktúrát kialakítani a felhasználói felületen. Composite helyett más is használható lett volna például canvas vagy group, én a composite mellett döntöttem. A fő mainComposite-nak be kell állítani a layout-ját, ezzel lehet megadni a composite szerkezetét. A layout beállításnak több módja van, ezeket itt nem ismertetném, csak az általam használt gridlayout féle beállítást. A gridlayout az egyik legegyszerűbb módja a composite felosztásának, ezzel csak oszlopokra lehet felosztani a felületet, két oszlopra osztottam szét a főkomponenst. A két oszlopban újabb composite-ok lettek létrehozva hasonló layout-tal a fő komponenséhez. A szerkesztő tehát három composite alkotja, egy fő és azon belül egymás mellett két másik. A két belső composite közül az első lesz a kérés, a második a válasz felülete. A másodlagos compositok is két-két oszlopra lett felosztva, az első oszlop a paraméter nevét, a második az értékét tartalmazza. A grafikus felületek építésénél figyelni kell arra, ha ezt a layout-ot használjuk a hozzáadott komponensek a hozzáadás sorrendben fognak elhelyezkedni a grafikus felületen.

A kérés composite-ja az első sorban a kérés nevét és egy Button, gomb widgetet tartalmaz. A button, gomb lenyomásával lehet elindítani a kérés kiküldését. A gomb lenyomása után kimegy az összeállított üzenet, fogadja a program a választ és megjeleníti azt. A gomb lenyomása egy esemény, így azt egy listenerrel kell lekezelni. A listeneren belül hívódik meg az adat küldése, fogadása és a válasz feldolgozása.

A válasz feldolgozása közben újabb elemeket adunk a felhasználói felülethez, de ezek ettől még nem mindig jelennek meg. A grafikus felület felépítése után frissíteni kell a layoutot, ez után válnak láthatóvá a módosítások. A hosszabb adatok fogadása esetén nem fér el a képernyőn a composite, így görgethetővé kellet azt tenni.

2.4.2.2 Kérés feldolgozása

Az összes kérés tartalmaz egy azonosítót és egy alfunkciót. Az azonosító mindig nyolc, az alfunkció általában nyolc-tizenhat bites lehet, funkciótól függően. A további paraméterek sokféleképpen nézhetnek ki. Az egyes paraméterek hossza tág tartomány között mozoghat, lehetnek 1 bitesek és akár több bájtosak is. A szöveggel kijelzett paraméterek úgynevezett Combo-ban míg a többi paraméter Text boxban foglal helyet a grafikus felhasználói felületen belül. A paraméterek neve Label-ekben kerülnek a felhasználói felületre, a Label is szöveget tartalmazó doboz, de csak feliratok kijelzésére szolgál, a felhasználó nem módosíthatja azt.

A kérés a kiválasztott szervizhez tartozik, a szerviznek megfelelő kódokat, paramétereket tartalmazza. A paraméterek közül az első kettő általában előre definiált, nem megváltoztatható paraméter. A további paraméterek meghatározott értékek vagy felhasználó által megadott érték lehet. A paraméterek feldolgozása ennek megfelelően megy végbe. Az egyes paraméterek típusa háromféle lehet konstans (CODED-CONST), érték (CODED-VALUE) vagy foglalt (RESERVED). Az első kettő paraméter konstans, a többi az utóbbi kettő közül kerül ki. A feldolgozás sorrendje az ODX leírásban lévő sorrend, de az adatok a kiküldendő adatokban más lehet, erre oda kell figyelni.

```
<REQUEST ID="_191">
    <SHORT-NAME>RQ_SerialNumber_Write</SHORT-NAME>
    <LONG-NAME>RQ Serial Number Write</LONG-NAME>
    <PARAMS>
      <PARAM SEMANTIC="SERVICE-ID" xsi:type="CODED-CONST">
          <SHORT-NAME>SID_RQ</SHORT-NAME>
          <LONG-NAME>SID-RQ</LONG-NAME>
          <BYTE-POSITION>0</BYTE-POSITION>
          <CODED-VALUE>46</CODED-VALUE>
          <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32" xsi:type="STANDARD-
          LENGTH-TYPE">
           <BIT-LENGTH>8</BIT-LENGTH>
           </DIAG-CODED-TYPE>
           </PARAM>
           <PARAM SEMANTIC="ID" xsi:type="CODED-CONST">
           <SHORT-NAME>RecordDataIdentifier</SHORT-NAME>
             <LONG-NAME>Identifier</LONG-NAME>
             <BYTE-POSITION>1</BYTE-POSITION>
             <CODED-VALUE>146</CODED-VALUE>
             <DIAG-CODED-TYPE BASE-DATA-TYPE="A_UINT32"
             xsi:type="STANDARD-LENGTH-TYPE">
             <BIT-LENGTH>16</BIT-LENGTH>
              </DIAG-CODED-TYPE>
              </PARAM>
              <PARAM SEMANTIC="DATA" xsi:type="VALUE">
              <SHORT-NAME>SerialNumber</SHORT-NAME>
              <LONG-NAME>Serial Number</LONG-NAME>
              <BYTE-POSITION>3</BYTE-POSITION>
```

```
      <DOP-REF ID-REF="_19"/>
      </PARAM>
    </PARAMS>
  </REQUEST>
```

[1] UDS-Example.XML

A fenti kód egy példát mutat, ami egy kérés XML leírását tartalmazza. A leírás egy egyszerű, három paramétert tartalmazó kérést tartalmaz. A kérés azonosítója (SERVICE-ID) a 46, a második paraméter az alfunkció azonosítója 146, a harmadik paraméter egy szériaszámot rejt, melynek adatbeállításait egy DOP tartalmazza. A DOP azonosítója _19, ez alapján kell kikeresni a többi DOP (DATA-OBJECT-PROP) közül.

A kérés feldolgozása tehát egy ciklus végzi, amely a paramétereken fut végig és a paraméter tulajdonságainak megfelelően építi fel a grafikus felhasználói felületet és egy struktúrát, ami tárolja a GUI elemeit és a paraméter információit. Egyes különleges paraméterek tartalmazhatnak saját paramétereket is, az ilyen paraméterek a struktúrák. A struktúrák feldolgozása hasonló a kérés feldolgozásához, ugyanúgy végig kell menni a paramétereken és hozzá kell építeni a felhasználói felülethez. Az alábbi kódrészlet egy struktúra XML leírását mutatja be.

```
<STRUCTURE ID="_61">
            <SHORT-NAME>EnvironmentData</SHORT-NAME>
            <LONG-NAME>Environment Data [4]</LONG-NAME>
            <BYTE-SIZE>4</BYTE-SIZE>
            <PARAMS>
              <PARAM SEMANTIC="DATA" xsi:type="VALUE">
                <SHORT-NAME>Operation_Cycle_Counter</SHORT-NAME>
                <LONG-NAME>Operation Cycle Counter</LONG-NAME>
                <BYTE-POSITION>0</BYTE-POSITION>
                <DOP-REF ID-REF="_54"/>
              </PARAM>
              …
          </STRUCTURE>
```

[2] UDS-Example.xml

A struktúra leírása hasonló a kéréshez, így a feldolgozás is hasonló módon megy végbe.

A kérés feldolgozása a RequestBuilder osztály build függvénye végzi. A függvény a paramétereken végigfutva végzi el a kérés feldolgozását. A függvény bemenete egy composite és egy lista, ami a Pbox elemeket tárolja. A kérés feldolgozásában a nehézséget a paraméterek különbözősége okozza, a paramétert úgy kell feldolgozni, ahogy annak beállítása megköveteli. A három típusnak megfelelően kell feldolgozni a paramétereket. A kódban látszik a három fő irány a függvényen belül.

A kérés feldolgozásában a struktúra feldolgozása is fontos. A struktúra feldolgozását a structureBuilder függvény végzi, ennek részletezését mellőzném, mivel nagyban hasonlít a kérés feldolgozásához.

A struktúrán vagy a kérésen belüli paraméterek feldolgozása különböző lehet, a paraméterek eltérő tulajdonságai miatt. A paraméterben lévő referencia az adott paraméter beállításának címe lehet. A cím alapján kikeresett DOP vagy DTCDOP tartalmazza a paraméter tulajdonságait. A tulajdonságok adnak információt a méretéről, helyéről, hogy az üzenet megfelelő részére kerüljön. A képernyőre kiírt értéket, szöveget is meghatározhatja, vagy a kijelzés módját. A feldolgozás lényege tehát ezeknek az objektumoknak a feldolgozása. A feldolgozást végző kódokat bemutatom.

A DTC-t feldolgozó kód az előre definiált értékeket keresi és felhasználja azokat a grafikus felülethez, ellenkező esetben csak egy Text dobozt helyez el. Az előre meghatározott értékek a COMPU-SCALE nevű elemeken belül helyezkednek el, ezeken belül vannak megadva az alsó és felső korlátai a paraméter értékének, sokszor ezek megegyeznek. Itt van megadva a kiírandó szöveg a paraméterhez, ezt a VT nevű elem tárolja, tehát a grafikus felület Combo-ja ezeket a szövegeket kapja meg. COMPU-SCALE nem csak egy lehet, több is előfordulhat egy DOP-n belül, ezekben definiálják a választható értékeit a paraméternek. A kód az alábbi.

```
public Widget paramWithDOP(MyParam param, Composite parent) {
	if (param.hasCompuScale()) {
	List<COMPUSCALE> cs = param.getCompuScales();
	Combo combo = new Combo(parent, SWT.NONE | SWT.READ_ONLY);
	int[] val = new int[cs.size()];
	int i = 0;
	for (COMPUSCALE c : cs) {
		val[i] = Integer.parseInt(c.getLOWERLIMIT().getStringValue());
		combo.add(c.getCOMPUCONST().getVT().getStringValue());
		i++;
		}
		combo.select(0);
		param.setValues(val);
		return combo;
		} else {
		Text text = new Text(parent, SWT.BORDER);
		return text;
		}
		}
```

A függvény először megnézi, hogy van-e a paraméternek COMPU-SCALE paramétere. A függvény a vizsgálat után létrehoz egy Combo-t, majd el kezdi belepakolni a feliratokat, ha végzett visszatér a Combo-val vagy Text-boxal.

A DTCDOP-val rendelkező paraméter a hozzá megfelelő DTC alapján kapja meg a felépítését, ilyen paramétereket általában csak fogadni kell. A DTC felépítése egyforma, csak a kiírt érték más és más. A DTCDOP-val rendelkező paraméterek nem sima DOP azonosítót tartalmaznak, hanem DTCDOP-ra mutató azonosítót. Az ilyen paraméterek által hordozott adat hibakódot takar, amit a vezérlőegység generál és tárol valamely rendellenes működés közben. A megtalált DTCDOP hasonló a DOP-hoz, de ez tartalmaz egy a DTC-ket tároló egységet a DTCS-t, ezen belül foglalnak helyet a DTC-k. A DTC-k az egyes hibakódok értelmezéséhez adnak segítséget, tartalmazzák a hibakódot és a hibakód szöveges leírását is.

A paraméterek összeállítása során nem csak a felhasználói felületet építjük fel, hanem egy adatstruktúrát is a widgetekből és a hozzájuk tartozó paraméterekből. Ez a struktúra kell, hogy az üzenetet össze tudjuk állítani. Az üzenet összeállításánál leírtam, hogy a struktúra hogyan néz ki, itt a felépítéséről is esett szó. A függvényekben a struktúra pboxes néven található meg.

2.4.2.3 Válasz feldolgozása

A programnak nem elég csupán kiküldenie egy üzenetet, fogadnia is kell egy választ. A válasz a vezérlő jelzése a programunk felé, hogy fogadta a választ és kielégítette azt, vagy nem volt lehetséges a végrehajtása. A válasz kétféle lehet pozitív vagy negatív. A válaszok mindegyike hordoz információt, némelyik csak tájékoztatást ad, egyesek ezzel ellentétben fontos adatokat is hordozhatnak. A pozitív válasz a kért információt tartalmazza, vagy művelet sikeres lefolyását jelzi. A negatív válasz jelzi, hogy a művelet nem lehetséges valamilyen oknál fogva. A negatív válasznak is két lehetséges változata van globális és egyszerű negatív válasz. A negatív válaszok azonosítója mindig 127, az alfunkció a kiküldött kérés azonosítója és lehet egy harmadik paraméter, amely a hiba kódját hordozza. A hibakódhoz az ODX leíráson belül tartozik egy szöveg, amiből megtudhatjuk a kérés elutasításának okát. A válasz feldolgozása hasonló, mint a kérésé, csak itt nekünk kell megadni a paraméterek értékét a fogadott válasz adataiból.

A válasz a szerkesztőnkön belül jobb oldalt, a kérés mellett helyezkedik el, hogy jól lássuk milyen kérésre érkezett a válasz. A válasz paramétereinek is vannak tulajdonságai, ezeket is ki kell nyerni az ODX leírásból és fel kell használni az adatok kijelzéséhez. Az adatok kijelzése hasonlóan megy, mint a kérés paramétereinél. A

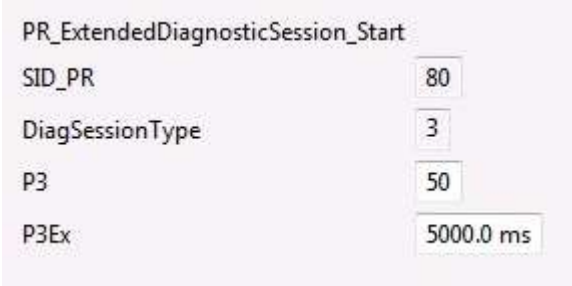
hasonlóságok és a feldolgozás menete az ODX leírás formája miatt van így, a leírás nem tesz különbséget kérésen és válaszon belül lévő leírásban. A válasz feldolgozásának első lépése az, hogy el kell dönteni a válasz milyen típusú pozitív vagy negatív. A feldolgozás csak ezután folytatható. A válasz kikereséséhez az első paraméter ad segítséget, ez alapján tudjuk a szervizhez tartozó válaszok közül kiválasztani a megfelelőt. A kiválasztás a válaszok első paraméterei és a fogadott adatok első bájtjának összehasonlítása az alapja, ha az összehasonlítás pozitív eredményt ad, akkor megvan a válasz, ez alapján dolgozható fel a bejövő üzenet.

A válasz feldolgozásánál vannak olyan esetek, mikor a válasz hossza nagyobb, mint egy normál válaszé, ekkor egy listát kapunk a vezérlőtől, vagy valami egyéb adatot. A lista általában a hibakódok listája. A hibakódok listájánál újra és újra a megadott struktúra alapján kell a szerkesztő felületére helyezni, ilyenkor egy sima for ciklust futtatok.

A válasz kijelzését a ResponseBuilder osztály függvényei valósítják meg. A válasz kijelzését a build függvény végzi, ez a függvény az osztály további függvényeit meghívva végzi feladatát. Külön függvények valósítják meg a struktúrával, DOP-val vagy DTCDOP-val rendelkező paraméterek feldolgozását. Az egyik legbonyolultabb a struktúra felépítése, mivel ez a másik kettő függvényt is meghívhatja.

A build függvény a megadott feltételek mellett, a megfelelő függvényt hívja meg, ezek a feltételek a paraméter megfelelő típusát tesztelik, és a típusnak megfelelő feldolgozó függvényt hívja meg. A meghívott függvény widgteket ad hozzá a felhasználói felülethez és írja ki ezekben az adatokat. A kiírt adatok a fogadott üzenetből származnak, az adatok kinyerésénél szükség van a bit- és bájtpozícióra, és a bithosszra, ezek alapján nyerhető ki az adat. Az adat kinyerését egy külön függvényben valósítottam meg. A kinyerő függvény a fogadott üzenetből nyeri ki a bit-,bájtpozíció és bithossz alapján a paraméter értékét. A függvénynek ennek megfelelően négy bemenő paramétere van. Fontos megemlítenem, hogy ez a függvény egy bináris Stringből nyeri ki az információt.

A kijelzésnél fontos, hogy nem mindig csak a bináris adatot kell kiírni, hanem hexadecimális, decimális vagy szöveges formában kell közölni az adatokat. Az adatok kijelzésének e módja a felhasználó számára fontos, csak így tudja értelmezni a kapott információt.



ábra 2.3. Egy válasz kijelezve

A 2.3-as ábrán látszik, hogy az 5000.0 ms lebegőpontos bináris számként kijelezve nem sokat mondana a felhasználónak, ezért van szükség az adatok konvertálására. Az adatok kijelzésének, számításának módja is az ODX leírásban található. Az adatok a DOP-n belül találhatóak meg a kijelzéshez. A mértékegységek a PUD-n belül találhatóak. A kijelzésénél tehát több adatot felhasználva kell létrehozni a végeredményt. Egyes értékekhez képlet is lehet megadva, de általában az ODX szabvány definiál egy alapképletet is. Az alapképlet egyszerű, csak olyan DOP esetén alkalmazható, amelyben a COMPU-METHOD típusa LINEAR. LINEAR típus esetén tartalmaz COMPU-NUMERATOR-t és ezen belül található két érték (pl.: <V>0</V><V>0.1</V>), ezek lesznek a képlet egyes állandói, a vett adat a változó. A képlet a példa alapján a következő: érték = 0 + vett adat * 0.1. A számítás módja más is lehet, ekkor le van írva egy képlet az ODX leíróban a számításokhoz.

2.4.3 konzol

A felhasználói felület része egy konzol felület, amire a program információkat tud kiírni. A kiírt információk hasznosak a felhasználó számára, tájékoztatást adnak a program működéséről. A programom által kiírt információk a szervizek nevei, a megnyitott szerviz tulajdonságai, kiküldött, és fogadott üzenetek. A konzol a tesztelésnél és fejlesztés közben is nagyon hasznos volt. A saját konzolra azért is szükség volt, mert a rendszerkonzol a futtató Eclipse konzoljára ír, és az olvasásához folyamatosan váltani kellett volna az ablakok között.

A konzol kezeléséhez egy MyConsole nevű osztályt hoztam létre, az osztály csak két függvényt tartalmaz. A függvények megkeresik a konzolt vagy létrehoznak egyet, ha nem találtak. A létrehozott objektum MessageConsoleStream nevű objektum lesz, és ezen az objektumon keresztül lehet üzeneteket kiírni a kijelzőre. A kiírás függvényei ugyanazok, mint a rendszerkonzol esetében, mivel a két objektum típusa

ugyanaz. A saját konzol használatához az Eclipse weboldalán találtam segítséget, ott egy példán keresztül bemutatják a módját.

http://wiki.eclipse.org/FAQ_How_do_I_write_to_the_console_from_a_plug-in%3F

A példányosítás egyszerű:

```
MessageConsoleStream out = new MyConsole().getMessageStream();
```

A példányosítás után a Javaban szokott módon kell használni a konzolt. Itt egy példa:

```
out.println("példa");
```

2.5 Tesztelés

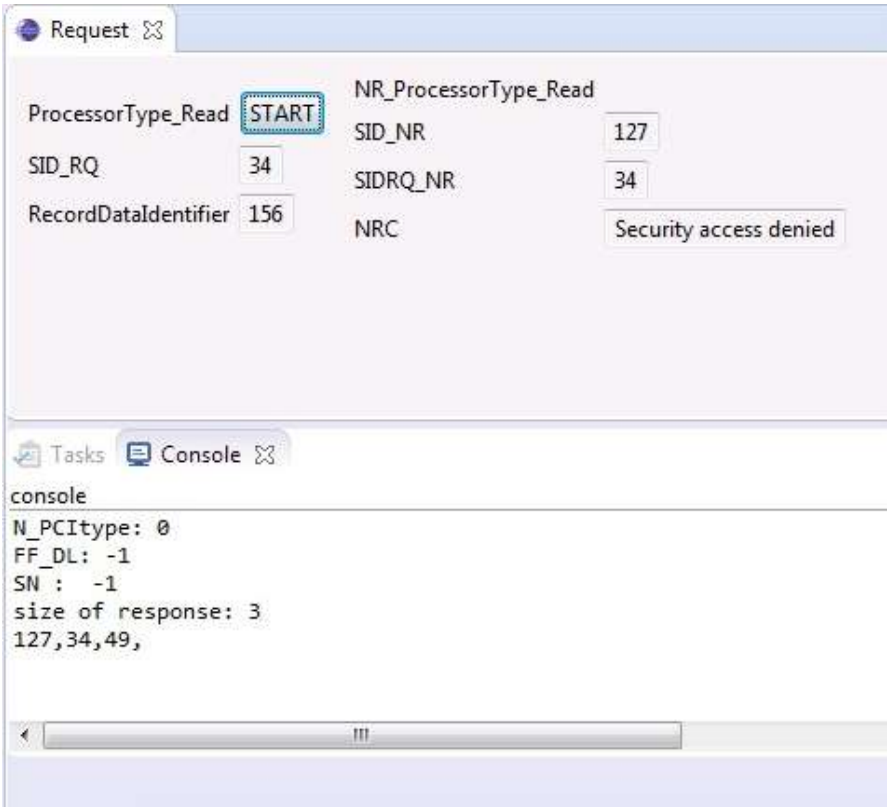
A tesztelés fontos fázisa a fejlesztésnek, mivel a tesztelés segítségével tudjuk ellenőrizni a helyes működést és a hibákat kiszűrni. A talált hibákat javítani kell. Fejlesztés során minden egyes új funkciót tesztelni kell, magában és a rendszerrel együtt is, ha hiba jelentkezik, azt javítani kell.

A végső tesztelésnél már az összeállított rendszert kellett vizsgálnom. A rendszer vizsgálatánál, az egyes funkciókat külön-külön és együtt is teszteltem. Először a fa nézetet, teszteltem. A fa nézetnél fontos volt, hogy csak a megfelelő szervizeket tegye be a fába. A kommunikáció és a driver tesztelése a kész rendszerben volt lehetséges. A felhasználó felület felépítése is nehéz feladat volt, ennek ellenőrzése is fontos volt. Az elemek külön-külön történő tesztelése a fejlesztés közben történt. A felhasználói felület felépítése nehéz és bonyolult, ezért ennek tesztelése és módosítása a fejlesztés közben folyamatos volt.

A rendszer teljes tesztje a szervizek kiküldése és fogadása közben történt. A tesztelés közbeni hibákat menet közben javítottam, a kódot módosítottam, finomítottam. Sokszor hiba volt, az üzenet összeállításában, a grafikus felület felépítésében, ezek voltak a bonyolultabb részei a rendszernek. A hibák javítása után már a működő rendszert tudtam tesztelni. A tesztelés folyamán kiválasztottam egy szervizt és az üzenetet kiküldtem egy vezérlőre, aminek azonosítóit megkaptam, tehát tudtam vele kommunikálni. A vezérlő minden kiküldött, választ váró szervizre válaszolt, tehát a kommunikáció megfelelően működött. A kijelzésnél voltak problémák, de ezeket kiküszöböltem. A helyes működést néhány példában bemutatom.

2.5.1 Szervizek tesztje

A programba beadott ODX fájlban lévő szervizek egy része működött az adott vezérlővel, mivel ezeknek a formája mindig egyforma minden egyes vezérlőegység esetén, egyes szervizekre csak egy általános negatív válasz jött vissza, mivel ezek nem voltak támogatott szervizek a felhasznált vezérlőegységen. Az alapvető szervizek működtek, mint a hibakódok (DTC-k) kiolvasása, kitörlése, stb. A kérések nagy részénél a pozitív válasz hiányát a biztonsági szint váltásának hiánya okozta. A vezérlőegység biztonsági kódjai nem találhatóak meg az ODX leírásban, tehát nem tudtam biztonsági szintet váltani, a vezérlő az alapértelmezett alap biztonsági szinten működött. A biztonsági kódok ismeretében sokkal több funkciót tudtam volna megvalósítani és használni, ilyenek lettek volna a program kiolvasása, írása és további biztonsági hibával visszajövő kérések lebonyolítása. A működés tesztelése közben így csak a működő szervizek tesztelése volt célszerű, mivel csak ezek adtak értékelhető adatokat a felhasználó számára. Más kérések szimplán nem voltak támogatott szervizek a vezérlőben, ezért nem kaptam pozitív választ. A következőekben bemutatok néhány a program által végrehajtott szervizt.

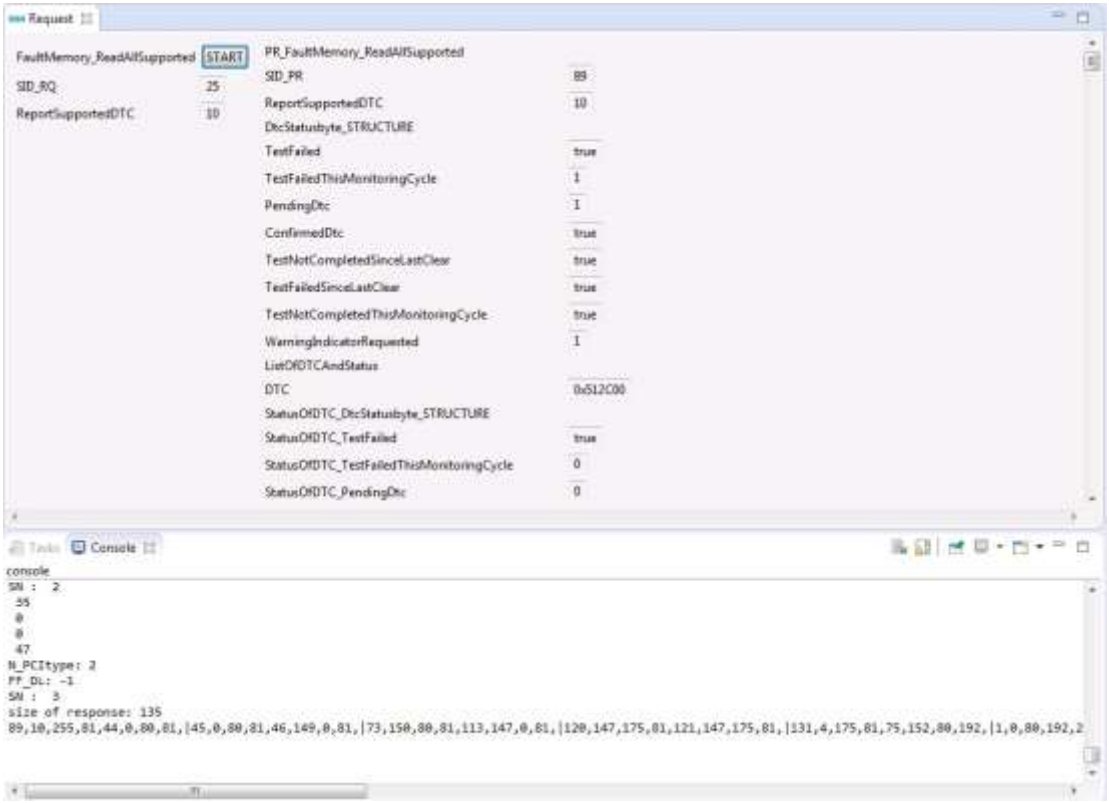


ábra 2.4. Processzor típusának kiolvasása

A processzor típusának kiolvasása egy egyszerű szerviz, ez csak a processzor típusát kéri le. A fogadott válasz egy negatív válasz, amely biztonsági hibára utal. A kapott üzenetből látszik, hogy biztonsági kód kell, biztonsági szintet kell váltani, úgymond rendszergazdának kell lenni a vezérlőn, hogy megkaphassuk a processzor típusát. A lényeg az, hogy ki tudtam küldeni egy az UDS által definiált, az ODX leírásból kinyert üzenetet és fogadni is tudtam a választ, igaz a válasz nem hordoz érdemi információt. A program tesztelésénél az ilyen típusú rövid válaszokat váró üzeneteket teszteltem le először. A későbbi tesztekben haladtam a bonyolultabb választ váró üzenetek felé. A bonyolultabb választ váró kérések is általában rövid egyszerű üzenetek.

Az első lépések során a kijelzés is csak primitívebb formában ment elsősorban a konzolon jelenítettem meg, miután már biztosan tudtam fogadni a rövidebb és hosszabb információkat, már a kijelzés is finomítottam, el kezdtem a grafikus felület felépítését. A kellő helyre a kellő módon kerültek ki a feliratok a felületre.

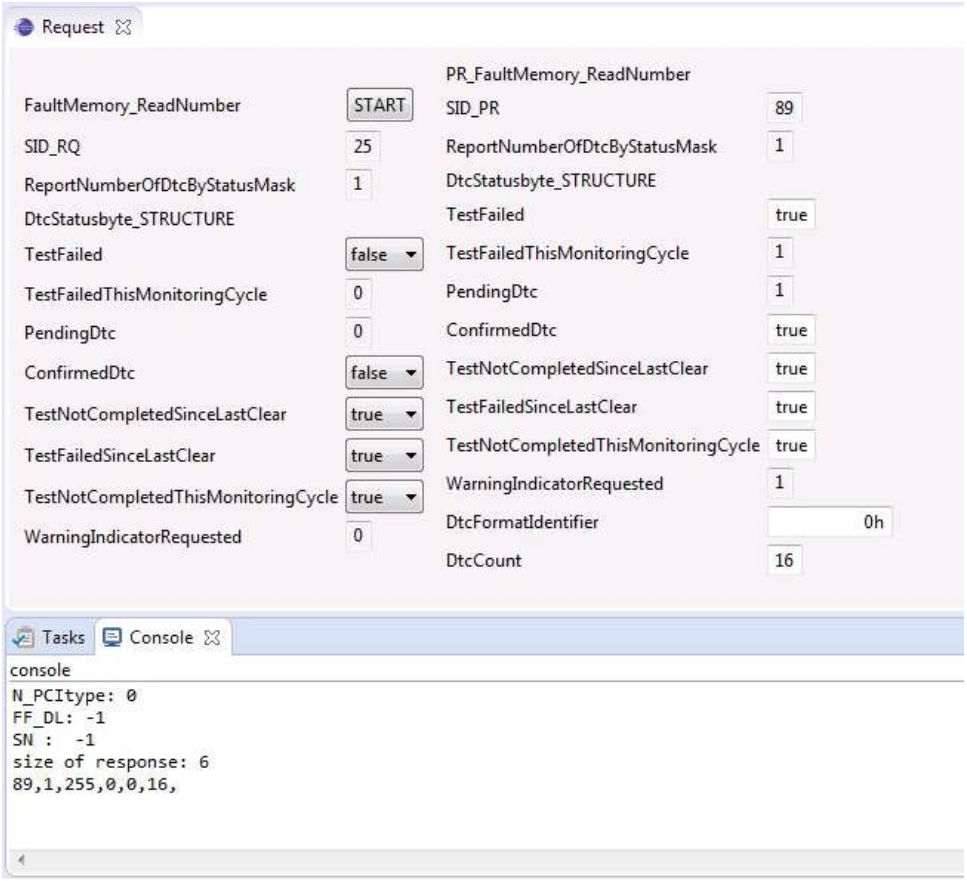
Egyik hosszú üzenetet váró szerviz a támogatott hibakódokat lekérő üzenet. Ez a kérés lekéri az összes a vezérlő által adható hibakódot, tehát egy hosszú üzenetről van szó. Az alábbi ábrán látszik a szerviz és a rá kapott információ.



ábra 2.5. Támogatott szervizek

A konzolon látszik, hogy a válasz hossza 135 bájtnyi, ez tartalmazza az összes hibakódot, amit támogat a vezérlőegység. A kijelzőn látszik a hibakód hexadecimális formában és a hibakódhoz tartozó státuszbájt. A státuszbájt minden bitje külön-külön ki van jelezve, az ODX leírás egy struktúrát társít a státuszbájthoz, amiben bitenként vannak a paraméterek definiálva. A nézeten belül lefelé görgetve látnánk sorban az összes hibakódot és azok státuszbájtját is. A vezérlő által támogatott hibakódok száma a válasz méretéből következtetve harminchárom. Az üzenet első három bájtja a pozitív válasz azonosítója, alfunkció és egy státuszbájt. A fennmaradó 132 bájtból a szervizek száma már könnyen számolható. A hibakód három, a státuszbájt egy bájt hosszúságú.

Lehetőség van a vezérlőegységben jelentkező hibák számának lekérdezésére is, ez is egy külön szerviz valósítja meg. A szerviz kérése három bájból áll az első kettő adott, az utolsó a státuszbájt módosítható. A módosítható státuszbájban adhatjuk meg, hogy milyen hibakódokra vagyunk kíváncsiak. A státuszbájt paraméterei egy struktúrában van megadva, minden bit jelentése más és más. A szerviz egy bizonyos státuszbájt beállítással az alábbi eredményt adta.



ábra 2.6. Hibakódok számának kiolvasása

A képen látszik a státusz beállítása és a hibakódok száma is. A hibakódokat le is lehet kérdezni, és a státusz is ugyanígy módosítható annál a szerviznél is. A hibakódok lekérdezése hasonló az összes hibakód lekérdezéséhez, csak ott beállíthatjuk a státuszbájttal a lekérdezendő hibakódok típusát.

A tesztelés során csak az ODX leírásban lévő szervizeket tudtam kipróbálni, ezek egy része működött, ezeket tudtam érdemileg ellenőrizni. Az ellenőrzött szervizek mindig érdemi adatot tartalmazó válasszal tértek vissza.

3 Összegzés

Az előző fejezetekben bemutattam a feladatot, a feladat értelmezését, magát a feladat megoldását és a megoldás tesztelését is. Létrehoztam és bemutattam a programot, ami képes kommunikálni elektronikus vezérlőegységgel, képes diagnosztikai üzenetek küldésére fogadására. A szoftver elég kezdetleges funkciókkal rendelkezik, de azok működnek, tesztelhetőek. A legegyszerűbb funkciók a szervizek listázása, szervizek kijelzése, üzeneteik összeállítása, kiküldése, válasz feldolgozása és értelmezése. A program a tesztelt vezérlőegységhez tartozó ODX leírás alapján sokkal több szervizt tudtam volna érdemben tesztelni, mivel a tesztelt szolgáltatások reprezentatív mintát adnak, az összes többi szolgáltatás is működőképes lehet, ha a vezérlőegység támogatja és az ODX leíróban szerepel.

3.1 Értékelés

A rendszer működik, az alapvető szervizek feldolgozása és kiküldése rendben lefolytatható. A felhasználói felület elég kezdetleges, a programnak ez a része fejleszthető, csinosítható. A kommunikáció egyes esetekben problémás, mivel a hosszabb üzenetek kiküldésénél a fogadott Flow Control üzenet nem azon a címen jön, mint amin a szerviz válaszát várjuk. A hosszabb üzenetek küldése, mint a szériaszám, program adatok és egyéb adatok kiküldése emiatt nem lehetséges. A hosszú adatok küldését nem tudtam tesztelni.

3.2 Kihívások

Számomra egyik legnehezebb pont a Java programozás megtanulása és a használatának elsajátítása volt. A Java hasonlít a többi általam ismert programozási nyelvhez, így lassan, de hozzá szoktam a használatához. A C vagy C++ nyelvhez képest egyszerűbb a használata, többnyire csak elvesz belőlük.

A program szempontjából nehéz pont az ODX leírás feldolgozása, az SWT használatának elsajátítása. Az ODX leírás feldolgozásában a szervizek és azok paramétereinek feldolgozás igen nehéz, sok odafigyelés igénylő feladat volt. Bonyolult és sok információ feldolgozását igénylő feladat az üzenetek összeállítása is.

3.3 További lehetőségek

A program csak az alapvető funkcióját valósítja meg a diagnosztikának, szervizüzeneteket küld és fogad. A programot bővíteni lehetne olyan funkciókkal, amelyekben diagnosztikai feladatok egy csoportját egyszerre lehetne végrehajtani. Bővítési lehetőséget látok még abban, hogy a diagnosztikai rutinokat és az eredményeiket naplózni is lehetne, így később vissza lehetne keresni a lefuttatott diagnosztikák eredményeit, ez hosszabb teszteknél az adatok tárolásához lenne nagy segítség. A kötegelt feldolgozás megvalósítása egy szkript segítségével megoldható lenne. A szkript egymás után több diagnosztikai feladatot hajtana végre és naplófájlt is készítene. A naplófájlt elég lenne egyszer áttekintve értékelni a lefuttatott tesztek eredményeit. Egyszerű, de fontos kiegészítés lehetne FlexRay támogatás beépítése, az átjáró rendelkezik FlexRay portokkal, csak egyszerűbb kiegészítések kellenének a programba a kommunikáció megvalósításához. Implementálni kellene a protokoll kezelését és az átjáró FlexRay driverét kellene felhasználni a programban.

4 Függelék

4.1 Köszönetnyilvánítás

Szeretnék köszönetet nyilvánítani Balogh Andrásnak, amiért segítséget nyújtott a munka elvégzéséhez és a Thyssenkrupp Presta Hungary kft. azon dolgozóinak, akikhez bátran fordulhattam kérdéseimmel.

Tisztaszívvel köszönöm szüleimnek a sok gondoskodást, ami elkísért a tanulmányaim során. Hálásan köszönöm kedvesem kitartó türelmét és a mindennapi feladatokban nyújtott segítségét.

4.2 Ábrajegyzék

4.3 Irodalomjegyzék

[1] ISO14229-1 2006 Road vehicles Unified Diagnostic Services

[2] ISO2291-1-ODX

[3] ISO 15765-2 2011 EN Road vehicles - Diagnostic communication over CAN Part 2

[4] http://hu.wikipedia.org/wiki/XML - XML

[5] http://hu.wikipedia.org/wiki/Eclipse - Eclipse

[6] Nagy Gusztáv Java programozás