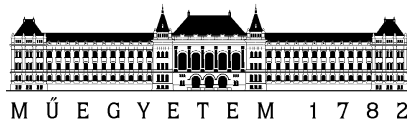


SZAKDOLGOZAT-FELADAT

Szentpéteri Szabolcs (HCAQUD)
villamosmérnök hallgató részére

AUTOSAR memóriastack tesztelése hibainjektálással

Egy modern gépjármű biztonsági és komfort funkcióit számos *beágyazott vezérlőegység* (ECU) támogatja. Az ezen számítógépeken futó szoftver komplexitása gyakran összemérhető a desktop alkalmazásokéval, pl. egy elektronikus kormányrendszer kb. 150 szoftverkomponensből, több ezer kapcsolatból és félmillió kódsorból áll. Annak érdekében, hogy az ECU a jármű teljes élettartamán át elviselje a fellépő *jelentős fizikai igénybevételt* (szélsőséges hőmérséklet, rázkódás, páratartalom, ingadozó tápellátás stb.), a desktop rendszerektől eltérően adattárolásra nem merevlemezeket, hanem tipikusan EEPROM vagy flash alapú tárolókat használnak. Ezen tárolók kezelőprogramjainak képesnek kell lenni a memóriát érő sérülések detektálására és javítására adatvesztés és a szolgáltatás időleges kiesése nélkül, hiszen akár biztonsági szempontból jelenős funkciók függhetnek tőle. Fontos követelmény tehát, hogy a memóriát kezelő rutinokról tapasztalati úton tudjuk bizonyítani, hogy képesek detektálni és túlélni egyes flash cellák sérülését. Ezen EEPROM vagy flash alapú tárolók kezelő szoftverei az autóiparban, a vezető autógyártók által 2002-ben életre hívott AUTOSAR konzorcium által definiált szoftver modulokból épülnek fel. A konzorcium célja az, hogy ezen szakterületi szabványokra építve specifikáljon egy (i) *alapvető szolgáltatásstruktúrát,* amely eltakarja a hardver sajátosságait és támogatja az alkalmazási szoftver hordozhatóságát (base software stack, BSW), (ii) egy *modellezési nyelvet* az ECU-kon futó alkalmazási szoftver szabványos leírására (software component template), és (iii) az alkalmazások és BSW-k ECU-n belüli és ECU-k közti *transzparens kommunikációját* lehetővé tevő elosztott, futásidejű szolgáltatást (RTE):

- A *base software stack* magában foglalja az alacsony szintű eszközmeghajtókat (pl. EEPROM és Flash driverek), az ezeket eltakaró absztrakciós rétegeket (pl. memória absztrakciós felület) és az ezekre ültetett magas szintű funkciókat (pl. perzisztens adattárolás).
- A *modellezési nyelv* lehetővé teszi, hogy precízen specifikáljuk az adattípusokat, illetve az alkalmazást alkotó komponensek interface-eit és belső felépítését.
- Az *RTE* egy generált glue kód réteg, amely eltakarja az alkalmazáskomponensek elől, hogy az általuk fogadott vagy küldött információ pontosan hogyan jut el a forrástól a célig.

A jelölt feladata egy hibainjektor szoftver kifejlesztése (egy – már létező – teszt keretrendszert felhasználva), amellyel a flash memória különböző sérüléseit lehet szimulálni, ezzel vizsgálva a flash memóriára épülő állománykezelő rendszer hibatűrő képességeit, esetleg a későbbiekben javaslatot adva a hibadetektáló és túlélő képességek javítására. A megvalósítandó tesztszoftver a *bitinverzión alapuló hibainjektálás* módszerét fogja felhasználni (melyet a későbbiekben részletezünk). A feladatot a következő lépésekben hajtsa végre:

- *A szabvány kapcsolódó részeinek megismerése:* (i) ismertesse az AUTOSAR rétegzett BSW struktúráján belül a *nem felejtő memória alacsony szintű kezeléséért (Flash Driver)* és a *nem felejtő memóriában tárolandó adatblokkok kezeléséért felelős modulok (Flash EEPROM Emulation* és *NVRAM Manager) szerepét (*gyűjtőnevükön: *Memory Stack*

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék
1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

modulok*),* különös tekintettel az adatblokkok kezelését megvalósító modulra (*NVRAM Manager*), majd (ii) vázolja ezen modulok együttműködését egy olyan *forgatókönyv bemutatásával*, melyben egy (vagy több) adatblokk flash memóriába kiírásának és visszaolvasásának menetét láthatjuk.

- *Memóriastack hibatűrésének analízise:*(i) ismertesse a teljes AUTOSAR memóriastack *hibamodelljét* - az egyes szoftver rétegek hiba detektáló képességein keresztül -, ill. ismertesse, hogy a memóriastack mely hibadetektáló képességeit tudja majd ellenőrizni a megvalósítandó teszt, (ii) fogalmazzon meg követelményeket a *tesztelési eljárásokra, a teszt környezet* és a *teszt eredményeinek kiértékelésére* a konzulens segítségével.
- *A megvalósítás erőforrásigényének vizsgálata:* (i) adjon becslést, az Ön által fejlesztett teszt teljes futási idejére, (ii) adjon becslést a célprocesszorban található beágyazott flash memória várható meghibásodására vonatkozóan, (iii) ismertesse, milyen környezetben érdemes a teszteket futtatnia, hogy az költséghatékony legyen.
- *Szoftvertervezés és -megvalósítás:* (i) modellezze UML-ben a statikus felépítést és a dinamikus viselkedést, (ii) készítsen – a teszt számára kiindulási alapot szolgáltató – különböző adattartalmú, hibamentes flashmemória-képeket (flash image), melyeken a bitinverziós hibainjektálás elvégezhető, (iii) módosítsa a rendelkezésre álló Memory Stack Integration Test projekt szoftver elemeit a megvalósítandó tesztnek megfelelően, (iv) készítsen vizuálisan is könnyen értelmezhető hibariportot a teszteredményekről (pl. HTML formátumban). A feladat megvalósítása a C és JAVA nyelv ismeretére támaszkodik, így a hallgatónak ezen nyelvek ismeretét is – a feladatnak megfelelő szinten – el kell sajátítania.
- *A megvalósítás tesztelése:* A memóriastack hibatűrésének analízise alapján készítsen olyan jellegű teszteseteket – a megvalósított teszthez –, amelyekkel a teszt hibadetektáló képessége az analízisben meghatározott esetekre bizonyítható.

Tanszéki konzulens: Dr. Sujbert László, docens
Külső konzulens: Teveli Zoltán (ThyssenKrupp Presta Hungary Kft.)

Budapest, 2017. október 7.

……………………
Dr. Dabóczi Tamás
tanszékvezető

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

Szentpéteri Szabolcs

AUTOSAR MEMÓRIASTACK TESZTELÉSE HIBAINJEKTÁLÁSSAL

BELSŐ KONZULENS

Dr. Sujbert László

BME MIT

KÜLSŐ KONZULENS

Teveli Zoltán

Thyssenkrupp Presta Hungary Kft.

BUDAPEST, 2017

Tartalomjegyzék

Összefoglaló ... **5**

Abstract ... **6**

Bevezető ... **7**

1 Elméleti háttér ... **8**

1.1 A flash memória ... 8

1.2 Az AUTOSAR 4.0 ... 8

1.2.1 Az AUTOSAR rétegzett szoftverarchitektúrája ... 9

1.3 A memóriastack ... 11

1.3.1 Az NVRAM Manager ... 12

1.3.2 Memory Hardware Abstraction Layer ... 17

1.3.3 A Flash EEPROM Emulation ... 17

1.3.4 A Flash Driver ... 18

1.3.5 A modulok együttműködése ... 18

1.3.6 A memóriastack hibamodellje ... 19

1.4 Hibainjektálással történő tesztelés ... 20

1.4.1 A hibainjektálás ... 20

1.4.2 A memóriastack tesztelése bitinverzión alapuló hibainjektálással ... 20

2 A tesztkörnyezet ... **21**

2.1 Rendszerterv ... 21

2.2 A tesztkörnyezet kialakítása ... 21

2.3 A teszt futási idejének becslése ... 22

2.4 A tesztkörnyezet megvalósítása ... 22

2.4.1 A mikrovezérlő ... 23

3 Tervezés ... **24**

3.1 A teszttel szemben támasztott követelmények ... 24

3.2 A flashmemóriaképek megválasztása ... 26

3.2.1 A flashmemóriaképek számának és tartalom elhelyezkedésének megtervezése ... 26

3.2.2 A megtervezett flashmemóriaképek ... 26

3.2.3 A flashmemóriaképek tartalmának megtervezése ... 29

3.3 A tesztfuttató szoftver megtervezése ... 30

3.3.1 A meglévő rendszer ismertetése 30
3.3.2 A Junit tesztelési keretrendszer 31
3.3.3 A hibainjektor tesztfuttató megtervezése 32
3.4 A hibainjektor szoftver megtervezése 33
3.5 A Serial Transport Protocol modul 34
3.5.1 A Serial Transport Protocollal szemben támasztott követelmények 35
3.5.2 A Serial Transport Protocol megtervezése. 37
4 Megvalósítás 41
4.1 A flashmemóriaképek előállítása 41
4.1.1 A pszeudovéletlenszám-generátor megvalósítása 41
4.1.2 Az NvM blokkok megválasztása 41
4.1.3 A képek létrehozása 41
4.2 A tesztfuttató szoftver megvalósítása 42
4.2.1 A tesztfuttató szoftver megvalósítása Junit parametrizált tesztként 42
4.2.2 A tesztfuttató szoftver megvalósítása Junit nem parametrizált tesztként 43
4.3 A hibainjektor szoftver megvalósítása 43
4.3.1 A hibainjektálás 45
4.3.2 Az NvM blokkok olvasása. 45
4.3.3 Az NvM blokkok írása 47
4.4 A Serial Transport Protocol megvalósítása 48
4.4.1 A fogadott üzenetek feldolgozása 48
4.4.2 Az üzenetküldés folyamata 50
4.4.3 Az STP használata a teszt szoftverben 51
4.5 A tesztriport előállítása 52
4.5.1 A tesztvégrehajtó válaszüzenete 53
4.5.2 A tesztriport megvalósítása 53
4.6 A megvalósítás tesztelése 54
Összefoglalás 55
Ábrák jegyzéke 56
Irodalomjegyzék 57

HALLGATÓI NYILATKOZAT

Alulírott **Szentpéteri Szabolcs**, 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, 2017. 12. 08.

...…………………………………………….

Szentpéteri Szabolcs

Összefoglaló

Egy mai modern gépjárműben egyre több funkciót elektronikus vezérlőegységek valósítanak meg. Ezekben a vezérlőegységekben adattárolásra leggyakrabban flash memóriát használnak. Biztonságkritikus funkciók esetén elengedhetetlen a megfelelő adattárolás biztosítása. Az AUTOSAR egy autóipari szoftver szabvány, mely definiálja a memóriát kezelő biztonságkritikus szoftvermodulokkal szemben támasztott követelményeket.

Feladatom egy hibainjektor szoftver kifejlesztése volt, amellyel a flash memória különböző sérüléseit lehet szimulálni, ezzel vizsgálva a flash memóriára épülő állománykezelő rendszer hibatűrő képességeit.

Szakdolgozatom első fejezetében bemutatom a flash memória sajátosságait, az AUTOSAR architektúrájának felépítését, különös tekintettel a memóriát kezelő modulok felépítésére, valamint a hibainjektálással való tesztelés sajátosságait.

A dolgozat második fejezetében bemutatom a tesztkörnyezet kialakításánál figyelembe vett megfontolásokat, becslést adok a teszt futási idejére, majd részletezem a megvalósított tesztkörnyezet felépítését.

Az ezt követő fejezetben részletezem az általam definiált követelményekből kiindulva, milyen előre létrehozott flash memória tartalmakon kell a hibainjektálást elvégezni. A hibainjektálást követő memóriaműveleteket végző algoritmus, valamint a tesztkörnyezet kialakítása miatt szükséges hibatűrő átviteli protokoll tervezési folyamatát is ebben a fejezetben részletezem.

A dolgozat utolsó fejezetében ismertetem a flash memória tartalmak létrehozásának menetét, a hibainjektor szoftver algoritmusainak működését, az implementált átviteli protokoll algoritmusokat és a teszt során keletkező tesztriport megvalósítását.

A dolgozatot a megvalósított tesztszoftver algoritmusainak megfelelő működésének bizonyításával zárom.

Abstract

In today's modern vehicles, more and more functions are implemented by electronic control units. In these controllers, flash memory is most commonly used for data storage. In the case of safety critical features, it is essential to ensure proper data storage. AUTOSAR is an automotive software standard that defines the requirements for the memory handling safety critical software modules.

My task was to develop a fault injector software that can simulate different types of damage to the flash memory, thus examining the fail safe capabilities of the flash memory-based journaling file system.

In the first chapter of my thesis, I present the features of the flash memory, the architecture of the AUTOSAR standard, with special attention to the structure of memory management modules, and the nature of fault injection testing.

In the second chapter of the thesis, I present the considerations that were taken in the development of the test environment, I give an estimation of the run time of the test, and then detail the structure of the implemented test environment.

In the following chapter, I detail what kind of pre-defined flash memory content should be used for the fault injection. I also outline the design process of the algorithm that handle the memory operations after the fault injection. I also detail the design process of the transmission protocol which ensure fault tolerant transmission.

In the last chapter of the thesis, I present the process of creating the flash memory contents, the behavior of the fault injection algorithms, the implemented transfer protocol algorithms and the implementation of the test report generation.

I conclude my thesis by demonstrating the proper functioning of the test software algorithms.

Bevezető

Témaválasztás

Szakdolgozatom a Thyssenkrupp Presta Hungary Kft.-nél készítettem. A céggel az Önálló laboratórium című tárgy során ismerkedtem meg, majd a nyári szakmai gyakorlatomat is a cégnél töltöttem. A szakmai gyakorlat során kezdtem el foglalkozni az AUTOSAR memóriastack hibainjektálással való tesztelésével, amely egyben a szakdolgozatom témája.

A feladat értelmezése

A feladat elvégzéséhez szükséges volt elsajátítanom az AUTOSAR architektúrájának felépítését, az AUTOSAR memóriastack felépítését és hibadetektáló képességét, valamint a hibainjektálással való tesztelés sajátosságait, így az első fejezetben ezeket a megszerzett elméleti ismereteket mutatom be.

A második fejezetben ismertetem a megvalósított tesztkörnyezet felépítését és kialakítását. A tesztkörnyezetnél egy a cégnél már meglévő tesztkörnyezetből indultam ki, ezt alakítottam át és fejlesztettem tovább, úgy, hogy a tesztet a lehető legköltséghatékonyabban tudjam futtatni.

A harmadik fejezetben ismertetem a teszttel szemben támasztott követelményeket. Ezen követelményekből kiindulva mutatom be a hibainjektor szoftver tervezését. A megvalósításhoz szükségesnek találtam egy UART feletti hibatűrő protokoll létrehozását, így a protokollal szemben támasztott követelményeket és a protokoll tervezési folyamatát is itt mutatom be.

A negyedik fejezetben ismertetem a megtervezett tesztszoftver, valamint a megtervezett átviteli protokoll megvalósítását. Ezen kívül bemutatom, hogy a teszt milyen tesztriportot generál, valamint, hogy a megvalósítást hogyan teszteltem.

1 Elméleti háttér

1.1 A flash memória

A flash memória egy nem felejtő típusú memória, amely elektronikusan törölhető és újraprogramozható. A flash memória két fő típusra osztható: NAND és NOR. Az autóiparban leginkább elterjedt memória a NAND flash. Előnye, hogy költséghatékony, és el tudja viselni azokat a komplex körülményeket, amelyeknek egy autóban meg kell felelnie. A flash memóriák kisebb blokkokban olvashatók és írhatók. A legkisebb írási egység a lap, ami általában nagyobb a processzor szóhosszúságánál. Írás csak törölt területre történhet. A flash memória legkisebb törölhető egysége a szektor, melynek mérete kilobájt nagyságrendű, természetesen ez eszközönként változó.[1]

1.2 Az AUTOSAR 4.0

Az AUTOSAR egy autóipari szoftver szabvány. Igen elterjedt, a legnagyobb autógyártók és az autóipari beszállítók nagy része AUTOSAR szabvány szerint fejleszt. A szabvány a hagyományos beágyazott rendszerekre fókuszál. Ezek alatt a processzorvezérelt irányító egységeket, a valós idejű rendszereket és az autóipari kommunikációs rendszereket értem. A szabvány nem fókuszál az autóban lévő grafikus felületekre, képfeldolgozó rendszerekre stb.

A szabvány 3 fő részből áll:

- Komponensorientált modellezési nyelv
- Gazdag alapszoftver könyvtár
- Fejlesztési folyamat leírása, megfelelőség, alkalmazási interfészek

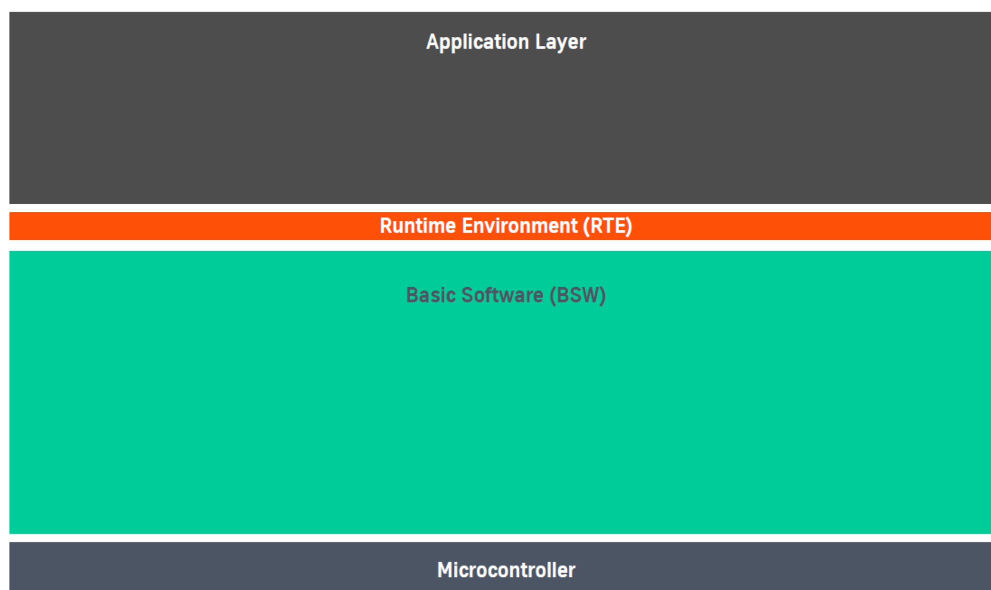
A fejezet a [2], [3], [4], és [5] dokumentumok alapján készült.

1.2.1 Az AUTOSAR rétegzett szoftverarchitektúrája

A legmagasabb absztrakciós szinten az AUTOSAR három réteget különböztet meg:

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

Az egyes rétegek elhelyezkedését az 1.1. ábra szemlélteti.



1.1. ábra. Az AUTOSAR rétegzett szoftverarchitektúrája

1.2.1.1 Az Alkalmazás réteg

Az Alkalmazás Réteg a rétegzett architektúra legfelső, teljesen hardverfüggetlen része. Ebben a rétegben helyezkednek el az AUTOSAR szoftverkomponensek. A szoftverkomponensek a rendszer valamely moduláris részének megvalósításai.

1.2.1.2 A Futtató Környezet

A Futtató Környezet (RTE) valósítja meg a szoftverkomponensek egymás, és a BSW réteg által nyújtott magasszintű szolgáltatások közötti kommunikációját, így a szoftverkomponensek megvalósítása teljesen függetlenné válhat az őket hordozó vezérlőegység jellemzőitől.

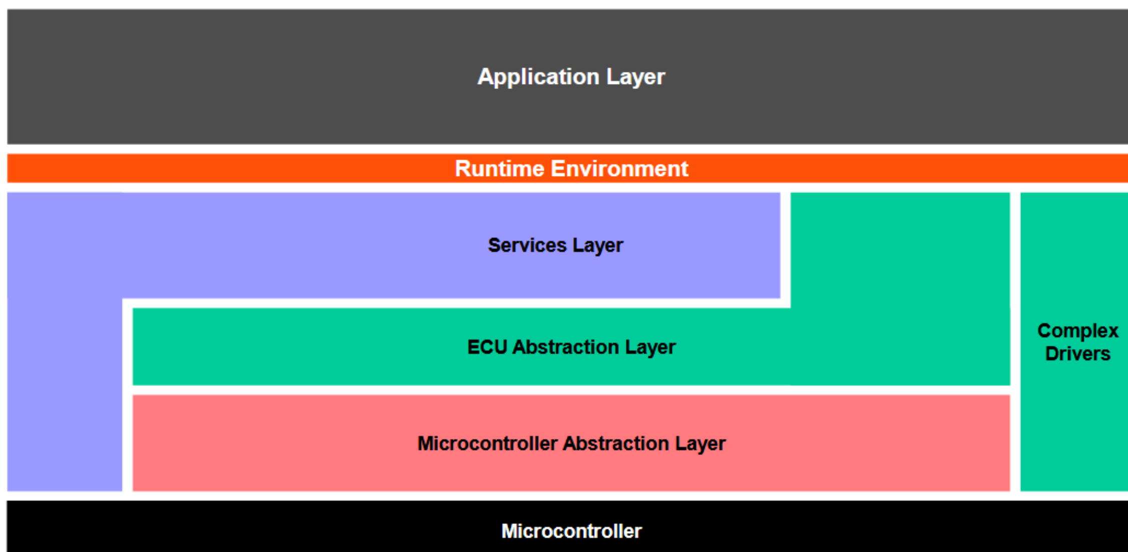
1.2.1.3 Az Alap Szoftver

Az AUTOSAR Alap Szoftver rétege lényegében egy szabványos interfészekkel rendelkező, moduláris felépítésű függvénykönyvtár, melynek feladata a mikrovezérlő szolgáltatásainak elérhetővé tétele az alkalmazás számára. Gazdagon konfigurálható modulok alkotják, melyeknek működése az adott vezérlőegység jellemzőinek és a vevői követelményeknek megfelelően testre szabható.

Az Alap Szoftver további négy rétegből áll:

- Szolgáltatás Réteg – Services Layer
- ECU Absztrakciós Réteg – ECU Abstraction Layer
- Mikrovezérlő Absztrakciós Réteg - Microcontroller Abstraction Layer
- Komplex Driverek – Complex Drivers

Az egyes rétegek elhelyezkedését az 1.2. ábra szemlélteti.



1.2. ábra. Alap Szoftver rétegek

A Szolgáltatás Réteg

A Szolgáltatás Réteg a BSW legfelső alrétege. Feladata, hogy alapvető szolgáltatásokat nyújtson az RTE-n keresztül a szoftverkomponensek számára. Ilyen szolgáltatások az operációs rendszer funkciók, a vezérlőegységek közötti kommunikáció, a hálózatmenedzsment, a memória menedzsment (nem felejtő memória kezelése), a

diagnosztikai funkciók (hibakódok kezelése, tárolása, diagnosztikai kommunikáció), és a vezérlőegység állapotmenedzsmentje.

Az ECU Absztrakciós Réteg

Az ECU Absztrakciós Réteg a driverek szolgáltatásaira támaszkodva nyújt hozzáférést a vezérlőegység funkcióihoz, függetlenül attól, hogy az adott funkciót megvalósító periféria a mikrokontroller belső vagy külső perifériája. Ez a réteg tartalmazza az esetleges külső perifériákat meghajtó driver modulokat is. Feladata, hogy a Szolgáltatás Réteg moduljai számára a vezérlőegység felépítésétől független interfészt biztosítson annak funkcióihoz.

A Mikrovezérlő Absztrakciós Réteg

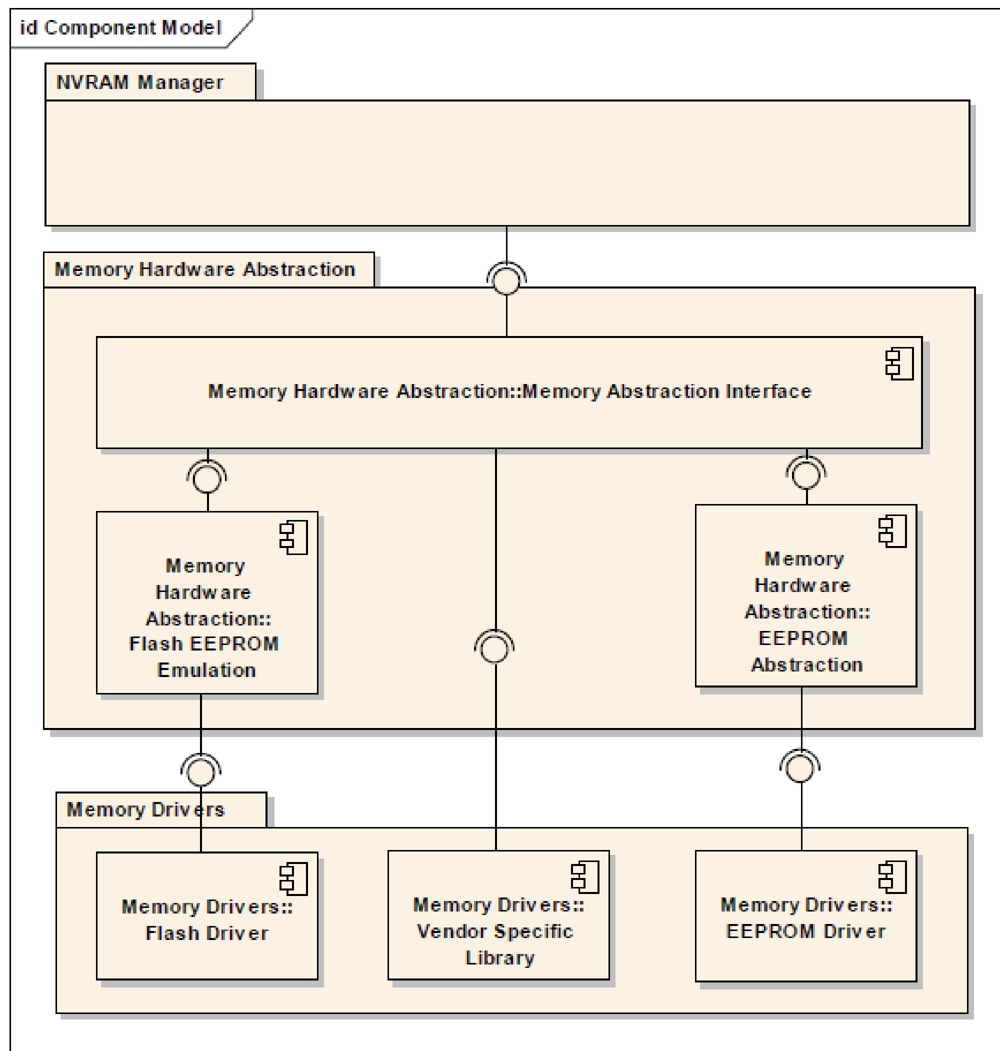
A Mikrovezérlő Absztrakciós Réteg a BSW stack legalsó rétege, mely a mikrokontroller perifériáihoz való hozzáférést biztosító driver modulokból áll. Feladata, hogy az ECU Absztrakciós Réteg moduljai számára mikrokontrollertől független hozzáférést biztosítson annak funkcióihoz, perifériáihoz.

Komplex Driverek

A Komplex Driver olyan BSW modul, mely az AUTOSAR által nem definiált funkcionalitást valósít meg. Az AUTOSAR a Komplex Driverek segítségével teremti meg a lehetőséget arra, hogy a speciális (nem szabványosított, vagy nem szabványosítható) funkciókat megvalósító perifériák szolgáltatásaihoz is hozzáférhessen az alkalmazás. A Komplex Driverek a rétegzett architektúrát átugorva felül közvetlenül az RTE-vel, míg alul közvetlenül az adott perifériával állnak kapcsolatban.

1.3 A memóriastack

A memóriastack az ECU memória műveleteiért felelős szoftvermoduljainak halmaza. A memóriastack a BSW réteg felépítése szerint felbontható Driver, Absztrakciós és Szolgáltatás alrétegekre. A Driver réteg a Flash Driver modulból, az Absztrakciós réteg a Memory Hardware Abstraction Layer, Flash EEPROM Emulation, EEPROM Abstraction modulokból, a Szolgáltatás réteg pedig az NVRAM Manager modulból áll. A memóriastack felépítését a 1.3. ábra szemlélteti. A fejezet a [6], [7], [8], [9], [10], és [11] dokumentumok alapján készült.



1.3. ábra A memóriastack felépítése

1.3.1 Az NVRAM Manager

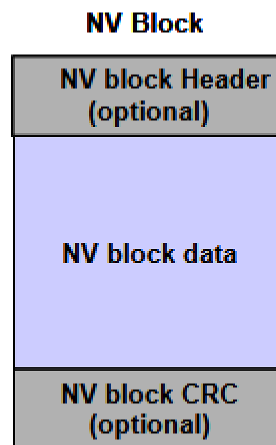
Az NVRAM Manager (NvM) az interfész az alkalmazások felé. Az NvM modul olyan szolgáltatásokat nyújt, amelyek biztosítják a NV (nem felejtő) adatok tárolását és karbantartását az egyéni igényeknek megfelelően egy autóipari környezetben. Ez a modul biztosítja a szükséges szinkron vagy aszinkron szolgáltatásokat a nem felejtő memória kezelésére és fenntartására. Az NvM egy EEPROM vagy FLASH EEPROM emulációs eszköz nem felejtő adatait kezeli. Megoldja, hogy szükség esetén az adatokat CRC-vel (Cyclic Redundancy Check) védjük, vagy több példányban tároljuk a rendelkezésreállás növelése érdekében. A továbbiakban részletesen ismertetem a különféle NVRAM blokk típusokat, hogy ezek a blokktípusok milyen építőelemekből állnak, valamint a teszt által használt NvM API függvényeket.

1.3.1.1 Az alapvető tárolási objektumok

Az NVRAM blokkok alapvető tárolási objektumokból állnak. Ilyen objektumból négy típust definiál az AUTOSAR szabvány.

NV blokk

Az NV blokk olyan alapvető tárolási objektum, amely egy memóriaterületet reprezentál, melynek részei a nem felejtő memóriában tárolt felhasználói adat, opcionális CRC, valamint opcionális NV blokk fejléc. Egy NV blokk felépítését a 1.4. ábra szemlélteti.



1.4. ábra. NV blokk szerkezete

RAM blokk

A RAM blokk olyan alapvető tárolási objektum, amely egy RAM területet reprezentál, melynek részei a felhasználói adat, opcionális CRC, valamint opcionális NV blokk fejléc. CRC csak akkor elérhető, ha a hozzá kapcsolódó NV blokknak van CRC-je, valamint a CRC-k típusának ilyenkor meg kell egyeznie.

ROM blokk

A ROM blokk olyan alapvető tárolási objektum, amely a ROM-ban található, és alapértelmezett adatokat szolgáltat üres vagy sérült NV blokkok esetén.

Adminisztratív blokk

Az Adminisztratív blokk a RAM-ban található. Ez a blokk az alkalmazások számára láthatatlan, kizárólag az NVM modul RAM és NVRAM blokkjainak biztonsági és adminisztratív feladatát látja el.

1.3.1.2 NVRAM blokk struktúra

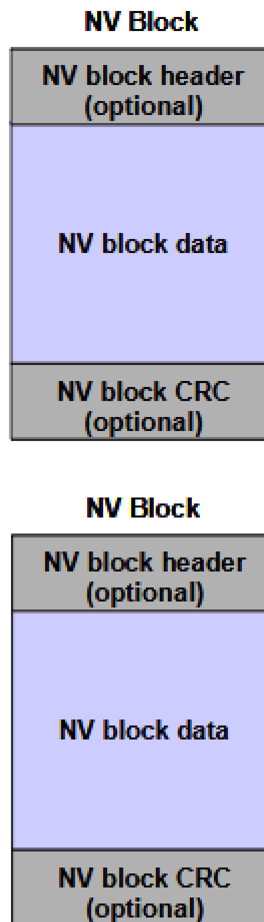
Az NVRAM blokkok alapvető tárolási objektumokból épülnek fel. Minden NVRAM blokk tartalmaz egy NV, RAM és Adminisztratív blokkot. Minden egyes NVRAM blokkhoz tartozik egy egyedi blokkazonosító, mellyel az adott blokk kiválasztható. A továbbiakban ismertetem a három NVRAM blokktípust.

Native NVRAM blokk

A Native NVRAM blokk a legegyszerűbb blokktípus, egy NV blokkból, egy RAM blokkból, valamint egy Adminisztratív blokkból áll.

Redundáns NVRAM blokk

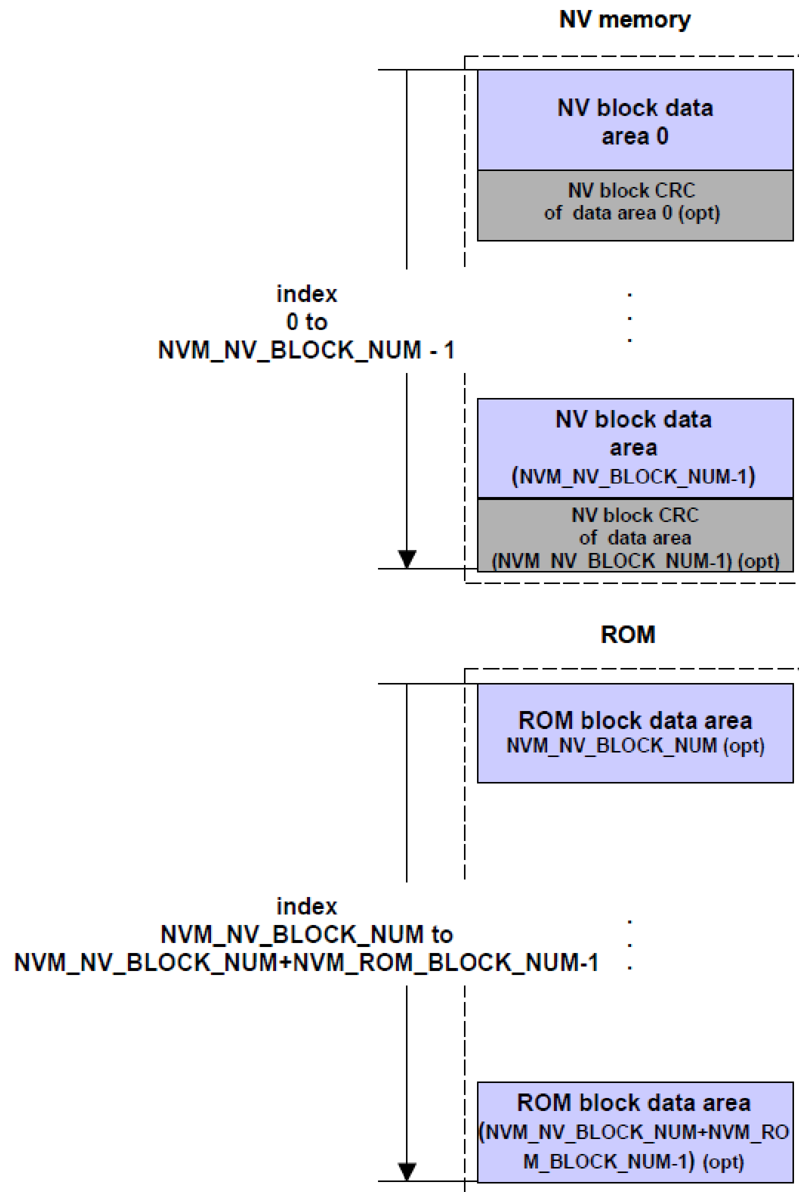
A Redundáns NVRAM blokk, két NV blokkból, egy RAM blokkból, valamint egy Adminisztratív blokkból áll. Mivel Redundáns NVRAM blokk esetén ugyanaz a felhasználói adat kétszer van letárolva, így egy Redundáns blokk sokkal nagyobb megbízhatóságot és hibatűrést biztosít, mint egy Native blokk. A blokk felépítését az 1.5. ábra szemlélteti.



1.5. ábra. Redundáns NVRAM blokk felépítése

Dataset NVRAM blokk

A Dataset NVRAM blokk egy olyan tömb, amely egyenlő méretű adatblokkokból áll (NV). Az alkalmazás egyszerre csak egy ilyen tömb elemet tud elérni. A Dataset NVRAM blokk több NV blokkból, egy RAM blokkból, valamint egy Adminisztratív blokkból áll. A tömb indexét az Adminisztratív blokk egy mezője tartalmazza. A blokk felépítését az 1.6. ábra szemlélteti.



1.6. ábra. A Dataset blokk felépítése

1.3.1.3 A teszt által használt NvM API függvények

NvM_Init

Az *NvM_Init* függvény alaphelyzetbe állítja a modul belső változóit és állapotgépeit. A függvény jelzi, ha az inicializálás megtörtént, így engedélyezve a modul belső folyamatainak működését.

NvM_SetDataIndex

Az *NvM_SetDataIndex* függvény beállítja egy Dataset NVRAM blokk tömb indexét a megadott értékre, egy olvasási, vagy írási művelethez. A függvény bemenő paraméterei egy Dataset blokkhoz tartozó blokkazonosító, valamint a beállítani kívánt index.

NvM_GetDataIndex

Az *NvM_GetDataIndex* függvény visszaadja a megadott Dataset NVRAM blokk tömb indexét. A függvény bemenő paramétere egy Dataset blokkhoz tartozó blokkazonosító, valamint egy 8 bites változó címére mutató pointer, amely címen a visszaadott index lesz elérhető.

NvM_ReadBlock

Az *NvM_ReadBlock* függvény kezdeményezi az olvasási műveletet, melynek során az NvM átmásolja letárolt NV blokk tartalmát a függvény bemenő paraméterként megadott RAM területre, vagy ennek hiányában a blokkhoz rendelt NVRAM blokkba. Az olvasási művelet lehetséges eredményei közül kettőt emelek ki, mivel a teszt során ezt a két visszaadott hibakódot veszem figyelembe, a következő esetekre, melyek (i) sikeres olvasás, (ii) redundancia elvesztése, (iii) bármilyen egyéb olvasási hiba. A következő két hibakód rendre a sikeres olvasás illetve a redundancia elvesztését jelzi:

- NVM_REQ_OK
- NVM_REQ_REDUNDANCY_FAILED

NvM_WriteBlock

Az *NvM_WriteBlock* függvény kezdeményezi az írási műveletet, melynek során az NvM az *NvM_WriteBlock* függvény bemenő paraméterként megadott RAM terület adatának felhasználásával, vagy ennek hiányában a blokkhoz rendelt NVRAM blokk felhasználásával az elkészített NV blokkot kiírj a flash memóriába.

1.3.2 Memory Hardware Abstraction Layer

A Memory Abstraction Interface (MemIf) feladata, hogy az FEE (Flash EEPROM Emulation) és EA (EEPROM Abstraction) modulok számától függetlenül egy virtuálisan szegmentált, egybefüggő címzési tartományt hozzon létre a felsőbb réteg számára. Ez a modul lehetővé teszi az NVRAM Manager számára, hogy elérjen számos memória-absztrakciós modult (FEE vagy EA modul).

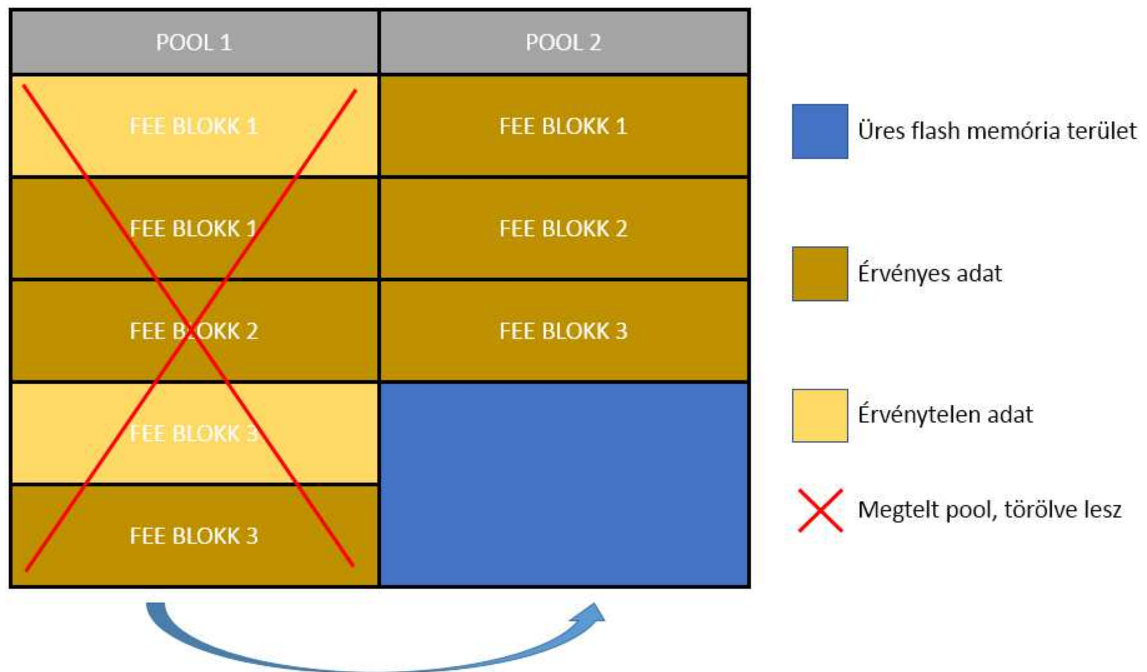
1.3.3 A Flash EEPROM Emulation

A Flash EEPROM Emulation (FEE) feladata, hogy az eszköztől függetlenül egy virtuális címzési tartományt hozzon létre a felsőbb réteg számára.

A Flash EEPROM Emulation egy naplózó fájlrendszert alakít ki a Flash Driver használatával, vagyis megoldja, hogy egy adatblokk virtuálisan felülírható legyen, tehát bejegyzi, hogy a régi adatblokk már nem érvényes, és az újat odaírja a szabad terület elejére. Az FEE-ben alkalmazott algoritmusok az adatblokkokat általában kettő vagy több összefüggő virtuális adatterületen tárolja, amelyekre a későbbiekben poolként hivatkozok. A definiált poolok közül mindig csak egy aktív, tehát ezen az aktív területen operál az algoritmus, addig amíg az be nem telik. Amikor az aktuális pool megtelik, akkor átmásolja az érvényes adatokat egy új poolba, és törli a régit. Egy pool állapotát(mely lehet aktív, nem aktív) a poolleíró tartalmazza.

A Flash EEPROM Emulation FEE blokkokat tárol le. Egy blokk újbóli kiírásával új példány keletkezik az adott blokkból. Minden egyes FEE blokkhoz tartozik egy implementáció specifikus blokk menedzsment leíró, mely tárolja a blokk aktuális állapotát. Az FEE blokk az NvM által letárolni kívánt adatot tartalmazza.

A naplózó fájlrendszert egy példán keresztül mutatom be, amit az 1.7. ábra szemléltet. Feltételezzük, hogy 2 poollal dolgozik az FEE. Először az FEE BLOKK 1 nevű blokk kerül kiírásra a memóriába. Ezután ugyanez a blokk újból kiírásra kerül, így az előzőleg kiírt blokk érvénytelen lesz, az új pedig érvényes. A későbbiekben két másik blokk is kiírásra kerül, az FEE BLOKK 2 és az FEE BLOKK 3. Az FEE BLOKK 3 újból kiírásra kerül, így az előzőleg kiírt FEE blokk érvénytelen lesz, az új pedig érvényes. Mivel az előző írási művelet után a POOL 1 megtelt, így az érvényes FEE blokkok átmásolódnak a POOL 2-be, a POOL 1 pedig törlődik.



1.7. ábra. FEE naplózó fájlrendszere

1.3.4 A Flash Driver

A Flash Driver a flash memória olvasására, írására és törlésére szolgál. A modul a Microcontroller Abstraction Layer-ben található, és közvetlenül hozzáfér a mikrovezérlő hardveréhez.

1.3.5 A modulok együttműködése

A modulok együttműködését egy írási és visszaolvasási forgatókönyvvel mutatom be. Írási művelet az NVRAM Manager NvM_WriteBlock nevű API függvényének meghívásával kezdeményezhető. A folyamat során NVRAM blokk típustól függően különböző számú FEE blokk kerül kiírásra. Native blokk esetén egy FEE blokk, Redundáns blokk esetén két FEE blokk, Dataset blokk esetén szintén egy FEE blokk. Az írni kívánt Dataset blokk egy tömb elemét az NvM NvM_SetDataIndex API függvénye segítségével lehet kiválasztani. Egy FEE blokk írása során az FEE ellenőrzi, hogy az adott FEE blokkhoz tartozik-e érvényes példány a flash memóriában. Ha tartozik, akkor érvényteleníti ezt a példányt, létrehoz az új példánynak egy leírót, és egy adat mezőt, majd a Flash Driver segítségével kiírja ezeket a flash memóriába.

Az olvasási művelet, az NVRAM Manager NvM_ReadBlock nevű API függvényének meghívásával kezdeményezhető. Native blokk esetén egy FEE blokk, Redundáns blokk esetén két FEE blokk, Dataset blokk esetén szintén egy FEE blokk kerül

felolvasásra. Az FEE a blokk menedzsment információt felhasználva a Flash Driver segítségével olvassa fel az FEE blokk adattartalmát.

1.3.6 A memóriastack hibamodellje

1.3.6.1 Az NVRAM Manager hibadetektáló képessége

Egy NV blokkhoz az előző fejezetben említettek alapján opcionálisan CRC rendelhető. A CRC egy hibadetektáló eljárás, mellyel az adat integritása vizsgálható. Ezt a CRC értéket NVRAM Manager kiszámolja a blokk kiírásakor az NVRAM blokk adatra, és le is menti azt. Az NVRAM blokk felolvasásakor az NvM kiszámolja a felolvasott NVRAM blokk adatára a CRC értéket, és ezt összehasonlítja a lementett értékkel. A számított CRC érték az egyes blokkokra konfigurálható, ez lehet 8 bites, 16 bites, vagy 32 bites. Az NVRAM Manager tehát képes a számított CRC értékekkel detektálni az NVRAM blokk felhasználói adatán fellépő hibákat, de ha egy adott blokkhoz nincs CRC rendelve, vagy a hiba olyan, hogy a számított CRC értékek egyeznek, akkor a fellépő hibát nem detektálja.

Az NvM konfigurációja során az egyes NVRAM blokkokhoz Static Block Id rendelhető. A Static Block Id az NVRAM blokk fejlécében található, és a blokk minden egyes írásakor eltárolódik. A blokk olvasásakor a blokkhoz tartozó Block Id és a felolvasott Block Id összehasonlításra kerül. Az NvM a Static Block Id segítségével tehát képes detektálni, ha rossz NVRAM blokk kerül felolvasásra.

1.3.6.2 A Flash EEPROM Emulation hibadetektáló képessége

A Flash EEPROM Emulation, mint már az 1.3.3. fejezetben ismertetésre került, FEE blokkokat és az ezekhez tartozó leírókat tárolja. Az FEE blokkok tartalma az a tartalom, amit az NVRAM Manager segítségével tárolunk le, így az itt fellépő hibák detektálása az NvM dolga. Az FEE a blokk menedzsment információk alapján képes detektálni a blokk státuszát, de ez az információ is megsérülhet, így az FEE-nek képesnek kell lennie a blokk menedzsment információ sérülésének detektálására is.

1.3.6.3 A Flash Driver hibadetektáló képessége

A Flash Driver a flash memória hibadetektáló képessége miatt tudja az írási, és olvasási műveletek hibáját detektálni.

1.4 Hibainjektálással történő tesztelés

1.4.1 A hibainjektálás

A hibainjektálás a hibakezelés tesztelésére szolgáló technika, mellyel egy rendszer hibakezelési funkcióinak megvalósítása ellenőrizhető. A módszer lényege, hogy olyan hibát idézünk elő a rendszerben, amely a valóságban várhatóan bekövetkezik, majd figyeljük és naplózzuk a rendszer válaszát az adott hibára. A hibainjektálás lehet célzott vagy véletlenszerű. Célzott hibainjektálás során adott hibakezelési funkciót lehet tesztelni, míg véletlenszerű hibainjektálással az adott rendszer megbízhatóságát, rendelkezésre állását lehet felmérni. [12]

1.4.2 A memóriastack tesztelése bitinverzión alapuló hibainjektálással

A memóriastack bitinverzión alapuló hibainjektálásával a flash memória élettartama során fellépő véletlenszerű hibákat lehet szimulálni, ezzel vizsgálva a flash memóriára épülő állománykezelő rendszer hibatűrő képességeit. A hibainjektálás, tehát egy bit invertálása, a Flash Driver segítségével történik, az FEE és NvM modulok inicializálása előtt, így a Flash Driver feletti modulok úgy érzékelik, mintha a flash memóriában keletkezett volna véletlenszerű hiba. Fontos megjegyezni, hogy teszt nem valós meghibásodást idéz elő, tehát a flash memória beépített hibadetektáló képessége által a hiba nem detektálható, így a Flash Driver nem fog hibát jelezni. Az injektált hibát csak a Flash Driver feletti modulok érzékelik, amelyek a naplózó fájlrendszer működéséért felelnek. A hibainjektálást a flash memória minden egyes bitjén szükséges elvégezni, tehát a poolleírókon, az FEE blokk leírókon, valamint az FEE blokkok adatain is. A teszt tehát képes ellenőrizni az FEE naplózó fájlrendszerének hibadetektáló képességét, az FEE blokk kezelő hibadetektáló képességét, valamint az NvM hibadetektáló képességét.

2 A tesztkörnyezet

2.1 Rendszerterv

Ebben az alfejezetben bemutatom a tesztrendszerhez milyen szoftver és hardver elemekre van szükség, valamint, hogy ezek az elemek milyen kapcsolatban állnak egymással.

A flash memória szoftveres hibainjektálásakor szükséges a flash memória fizikai írása, így a teszteléshez szükséges egy mikrovezérlő, amelyen a hibainjektor szoftver fut, valamint egy flash memória. A mai modern mikrovezérlőkben már található integrált flash memória, így elegendő egy ilyen célprocesszor beszerzése. A tesztelés során fontos, hogy a hibainjektálást végző tesztszoftver környezete hasonlítson egy ECU-ban futó szoftver környezetéhez, így szükséges, hogy a cég által használt mikrovezérlőn fusson a teszt. Az AUTOSAR memóriastack teszteléséhez szükséges a memóriastack összes szoftvermoduljának megléte. Ezek a modulok C nyelven vannak implementálva, így ezeket a modulokat összeintegrálva az általam írt hibainjektor szoftverrel a cég által használt mikrovezérlőn a hibainjektálás elvégezhető. Az 1.4.2. fejezetben említettek alapján az NvM és FEE modulokat szükséges a hibainjektálás után inicializálni. Mivel a modulok deinicializálása nem lehetséges, így szükséges minden egyes hibainjektálás után a mikrovezérlőt resetelni.

Célszerű lenne, egy olyan szoftvert implementálni, amely irányítja a hibainjektálás folyamatát a mikrovezérlőn, tehát hibainjektálás és reset parancsokat küld a mikrovezérlőn futó szoftvernek, majd a hibainjektálás után a hibainjektor szoftver által visszaküldött eredményt kiértékeli és egy tesztriportot generál belőle.

Az említett megfontolások miatt a tesztrendszer két elemből fog állni, egy tesztfuttatóból, amely a hibainjektálás folyamatát irányítja, valamint egy tesztvégrehajtóból, amely a hibainjektálást végzi.

2.2 A tesztkörnyezet kialakítása

A tesztkörnyezet kialakításánál a már rendelkezésre álló, a cég által használt célzott hibainjektáló tesztkörnyezetből indultam ki. A tesztkörnyezet két elemből állt, egy tesztfuttatóból és egy tesztvégrehajtóból. A tesztfuttató egy PC-n futó JAVA nyelven írt alkalmazás volt. Az alkalmazás egy tesztlépéseket leíró fájlt dolgozott fel, majd ez alapján

küldött parancsokat a tesztvégrehajtó alkalmazásnak. A tesztvégrehajtó egy C nyelven írt beágyazott szoftver volt, amely egy mikrovezérlőn futott, és a memóriastack célzott hibainjektálását végezte. A tesztkörnyezet elsősorban azért módosult, mert alkalmassá kellett tenni a bitinverziós tesztlépések futtatására. A tesztkörnyezet optimális kialakítása, a két legfontosabb szempont, (i) a teszt futási ideje, valamint (ii) a költséghatékonyság alapján lett meghatározva.

2.3 A teszt futási idejének becslése

A teszt futási idejét azért fontos megbecsülni, mert a kiinduló projekt tesztfuttató szoftvere PC-n futott, viszont egy több napig, vagy akár több hétig futó teszt esetén nem lenne költséghatékony a tesztfuttató szoftvert PC-n futtatni. Az 2.1-es fejezetben említettek alapján minden egyes hibainjektálás után szükséges a mikrovezérlőt resetelni. A mikrovezérlő resetelése után szükséges egy másodpercet várni, mire a bootloader újra elindítja a programot a mikrovezérlőben. Egy tesztlépés végrehajtása nagyjából két másodpercet vesz igénybe. Egy flashmemóriakép mérete, amelyen a hibainjektálást végezzük 32, kilobájt. A méret bit egységre való átváltását a következő számítás adja meg:

$$32 * 1024 * 8 = 262144$$

Egy bithiba előállítására 2,5 másodpercet számítva, egy flashmemóriaképre a teszt futási idejét másodpercben mérve a következő számítás adja meg:

$$262144 * 2{,}5 = 655360$$

A futási időt napokban is megadom, érzékeltetve ezzel a teszt futási időigényét:

$$\frac{655360}{24 * 3600} = \sim 7{,}5$$

A 3.1-es fejezetben említett követelmények alapján a tesztet minimum 2 flashmemóriaképen szükséges elvégezni, így a teszt futási ideje 15 napra adódik. Mivel a tesztet minimum 5 flashmemóriaképre szeretném elvégezni, így a teszt futási ideje nagyjából 38 napra adódik.

2.4 A tesztkörnyezet megvalósítása

A teszt futási idejéből adódóan nem lenne költséghatékony a tesztfuttató szoftvert PC-n futtatni, érdemes lenne inkább valamilyen mikroszámítógépen. Az eszköznek

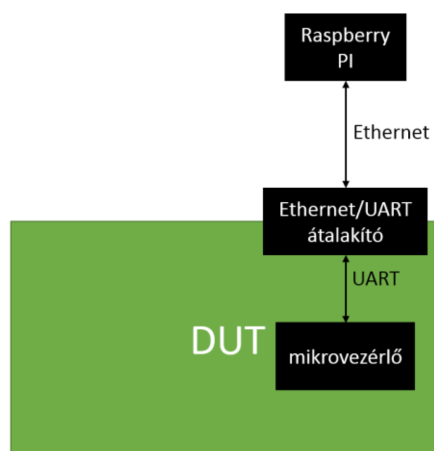
képesnek kell lenni futtatni egy olyan operációs rendszert, amelyen egy JVM (Java Virtual Machine) futtatható. Az eszköznek elegendő RAM-mal kell rendelkeznie a teszt futtatásához, valamint rendelkeznie kell Ethernet porttal. A felsorolt szempontok miatt a választás egy Raspberry Pi-ra esett. A tesztkörnyezet így a következőképp lett megvalósítva:

A tesztvégrehajtó egy mikrovezérlőn fut, mely mikrovezérlő a DUT (Device Under Test) része. A DUT-on található egy Ethernet/UART átalakító, így a mikrovezérlő UART-on keresztül fog kommunikálni ezzel az átalakítóval. A tesztfuttató egy Raspberry Pi-on fog futni. A DUT és a Raspberry Pi Etherneten keresztül fog kommunikálni.

A megvalósított tesztkörnyezet felépítését a 2.1. ábra szemlélteti.

2.4.1 A mikrovezérlő

A mikrovezérlő gyártóját, és típusát jogi okokból nem írhatom le, de a teszt számára fontos tulajdonságait ismertetem. A mikrovezérlő 32 bites és tartalmaz integrált flash memóriát. A mikrovezérlő adatlapja szerint a flash memória 250000–szer biztosan újraírható hibamentesen. Ezután a memória írása során léphetnek fel hibák. A 2.3-as fejezetben levezetett számítások alapján a teszt egy flashmemóriaképre 262144 tesztlépésből áll. Minden egyes tesztlépésben szükséges a flash memóriát újraírni, így egy flashmemóriakép teljes tesztelése esetén, a flash memória 262144-szer újraíródik. Ez a szám ~5%-kal lépi túl azt az írási számot, ami alatt garantáltan nem lép fel hiba. Előzetes tesztek alapján úgy döntöttem, hogy nem szükséges 250000 írás után cserélni a mikrovezérlőt, elegendő minden flashmemóriakép után, mivel a hiba valószínűsége elenyésző.



2.1. ábra A tesztkörnyezet felépítése

3 Tervezés

3.1 A teszttel szemben támasztott követelmények

[TKP_FI_001]

A hibainjektálást minimum 2 flashmemóriaképen kell elvégezni. Ezen flashmemóriaképek tartalmának életszerű helyzetet kell tükröznie. Az egyik képnek csak az egyik pooljában kell adatnak lennie, a másik képnek mindkét pooljában kell adatnak lennie.

Magyarázat: Az FEE hibadetektálási képessége különböző a két flashmemóriaképre.

[TKP_FI_002]

A környezetnek biztosítania kell, hogy az NVRAM blokkok jól megkülönböztethetők, valamint, hogy minden NVRAM blokk típus használva van a teszt során. A tesztnek valamilyen tipikus konfigurációval kell futnia.

[TKP_FI_003]

A táblázat a hibainjektálás lépéseit mutatja be.

Lépés	Leírás
A	A hibainjektáló tesztnek az előre meghatározott flashmemóriaképeket be kell töltenie a RAM-ba, és ott kell a hibainjektálást elvégeznie, majd ezt a tartalmat vissza kell írnia a flashmemóriába a Flash Driver segítségével. Sikeres végrehajtás esetén a tesztnek a C lépéssel kell folytatódnia, egyébként a B lépéssel. Megjegyzés: A hibainjektáló tesztnek visszaírás előtt össze kell hasonlítania poolonként a flashmemóriaképet a RAM-képpel, és ha különböznek, csak akkor kell felülírnia azt.
B	A hibainjektáló tesztnek jelentenie kell, hogy a hibainjektálás sikertelen volt, és a tesztnek be kell fejeződnie.
C	A hibainjektáló tesztnek az NVRAM Manager segítségével fel kell olvasnia azokat az NVRAM blokk tartalmakat, amelyek olvasásra vannak

	konfigurálva. Sikeres végrehajtás esetén a tesztnek a D lépéssel kell folytatódnia, egyébként az E lépéssel.
D	A hibainjektáló tesztnek össze kell hasonlítania a felolvasott NVRAM blokk tartalmakat az előre meghatározott NVRAM blokk tartalmakkal. Sikeres végrehajtás esetén a tesztnek az F lépéssel kell folytatódnia, egyébként az E lépéssel.
E	A hibainjektáló tesztnek jelentenie kell, hogy az olvasás sikertelen volt, és a tesztnek be kell fejeződnie.
F	A hibainjektáló tesztnek írnia kell az összes olyan NVRAM blokkot, amely írásra van konfigurálva. Sikeres végrehajtás esetén a tesztnek a H lépéssel kell folytatódnia, egyébként a G lépéssel.
G	A hibainjektáló tesztnek jelentenie kell, hogy az írás sikertelen volt, és a tesztnek be kell fejeződnie.
H	A hibainjektáló tesztnek az NVRAM Manager segítségével fel kell olvasnia azokat az NVRAM blokk tartalmakat, amelyek olvasásra, vagy írásra vannak konfigurálva. Sikeres végrehajtás esetén a tesztnek a I lépéssel kell folytatódnia, egyébként a J lépéssel.
I	A hibainjektáló tesztnek össze kell hasonlítania a felolvasott NVRAM blokk tartalmak közül azokat, amelyek olvasásra voltak konfigurálva, de írásra nem, az előre meghatározott NVRAM blokk tartalmakkal. Sikeres végrehajtás esetén a tesztnek a K lépéssel kell folytatódnia, egyébként az J lépéssel.
J	A hibainjektáló tesztnek jelentenie kell, hogy az olvasás sikertelen volt, és a tesztnek be kell fejeződnie.
K	A hibainjektáló tesztnek jelentenie kell, hogy a tesztlépés sikeres volt, és a tesztnek be kell fejeződnie.

[TKP_FI_004]

A tesztlépés olvasási művelete, amelyet a TKP_FI_003C és a TKP_FI_003H pontok írnak le, sikeresnek tekinthető, ha (i) az összes API függvény az olvasás lépésben E_OK-kal tér vissza, és (ii) az összes NVRAM blokkot sikeresen olvastuk, vagy a sikertelenül

olvasott NVRAM blokkok száma 1, vagy a sikertelenül olvasott NVRAM blokkok száma megegyezik az olvasott NVRAM blokkok számával.

[TKP_FI_005]

A tesztlépés írási művelete, amelyet a TKP_FI_003D pont ír le, sikeresnek tekinthető, ha (i) az összes API függvény az írás lépésen belül E_OK-kal tér vissza, és (ii) az összes NVRAM blokkot sikeresen írtuk.

[TKP_FI_006]

A tesztlépés összehasonlítási művelete, amelyet a TKP_FI_003D és TKP_FI_003I pontok írnak le, sikeresnek tekinthető, ha az olvasott és a várható tartalmak egyeznek.

[TKP_FI_007]

A tesztlépés hibainjektálás művelete, amelyet a TKP_FI_003A pont ír le, sikeresnek tekinthető, ha a visszaírás a flash memóriába sikeres.

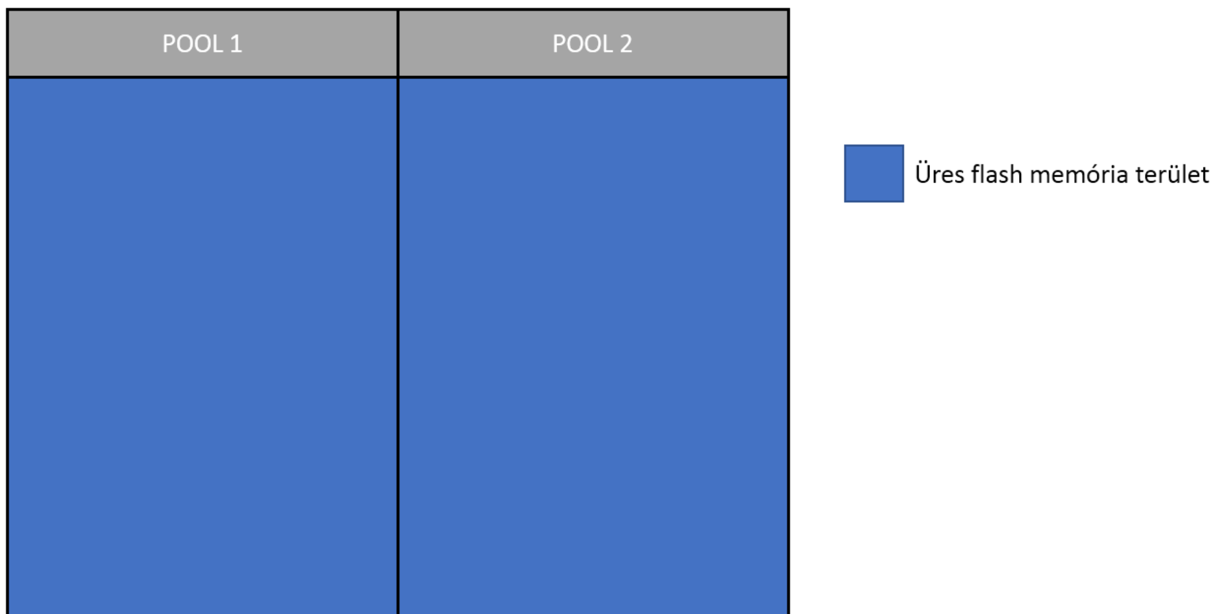
3.2 A flashmemóriaképek megválasztása

3.2.1 A flashmemóriaképek számának és tartalom elhelyezkedésének megtervezése

A flashmemóriaképek számának megválasztásánál elsődleges szempont a [TKP_FI_001] számú követelmény volt. A követelmény szerint minimum 2 flashmemóriaképet kell létrehozni, úgy, hogy egyikben csak az egyik pool, másikban pedig minkét pool adattal van feltöltve. Erre azért van szükség, mert ebben a két esetben az FEE hibadetektálási képessége eltérő. Az előző állításból viszont következik, hogy szükséges lenne kettőnél több flashmemóriaképet létrehozni, melyekkel az összes előforduló flashmemóriakép elrendezés realizálható.

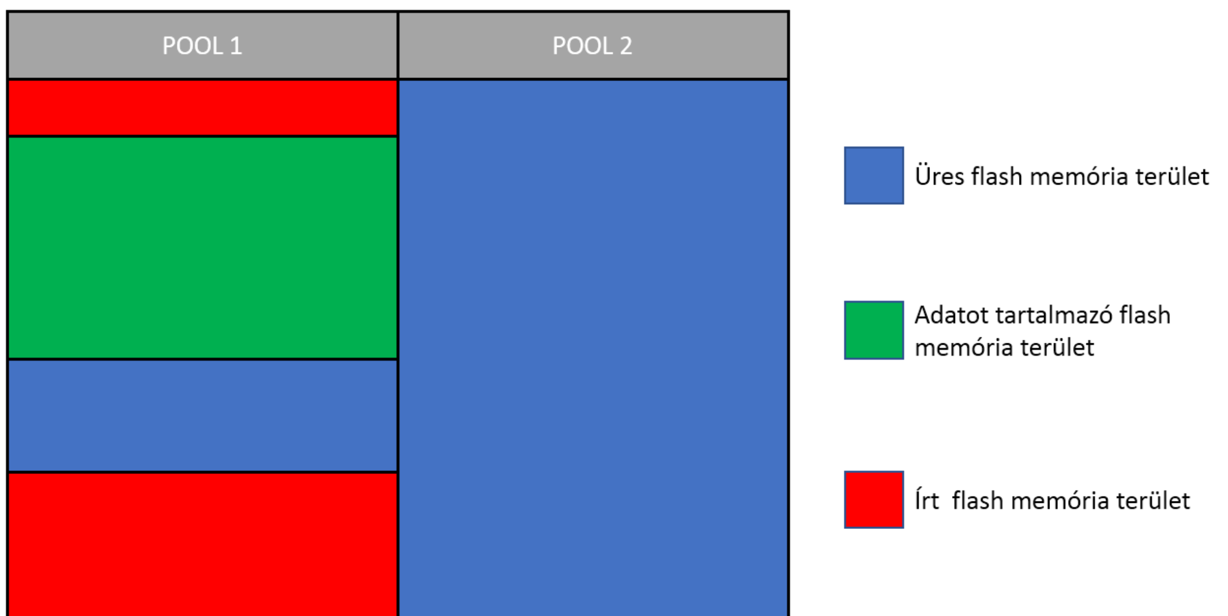
3.2.2 A megtervezett flashmemóriaképek

Az első flashmemóriakép egy teljesen üres kép, ahol mindkettő pool üres, ilyen a flash memória kezdeti állapota. A flashmemóriaképet a 3.1. ábra szemlélteti.



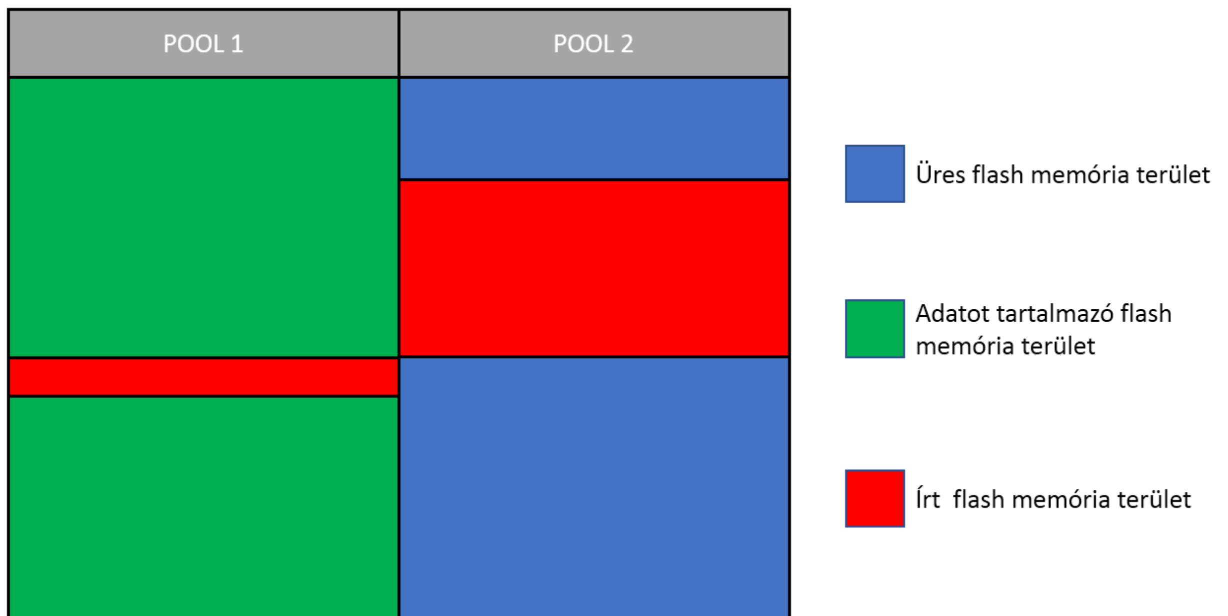
3.1. ábra. Üres flashmemóriakép

A második flashmemóriaképen csak az egyik poolban van adat, és ott is csak pool felénél kevesebb, így nem fog bekövetkezni poolváltás az írási művelet után sem. A flashmemóriaképet a 3.2. ábra szemlélteti.



3.2. ábra Második flashmemóriakép

A harmadik flash kép olyan, hogy az első pool majdnem tele van, a második viszont üres. Ebben az állapotban biztosan lesz poolváltás az írási művelet után. A flash képet a 3.3. ábra szemlélteti.

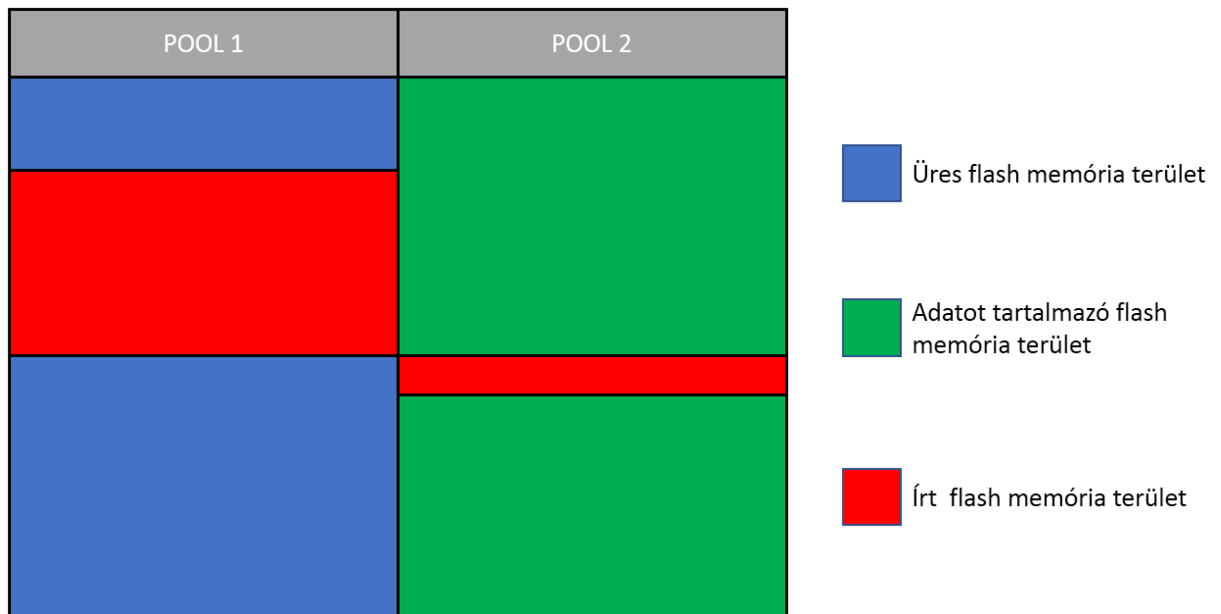


3.3. ábra. Harmadik flashmemóriakép

A negyedik flashmemóriaképen az első pool már tele van, viszont a második pool csak félig van. Ebben az esetben nem fog bekövetkezni poolváltás vissza az első poolra az írási művelet után. A flashmemóriaképet a 3.4. ábra szemlélteti.

3.4. ábra. Negyedik flashmemóriakép

Az ötödik flashmemóriaképen már mindkét pool tele van adattal, így biztosan lesz visszaváltás az egyes poolra az írási művelet után. A flash képet a 3.5. ábra szemlélteti.



3.5. ábra. Ötödik flashmemóriakép

3.2.3 A flashmemóriaképek tartalmának megtervezése

A flashmemóriaképek tartalmának megválasztásánál elsődleges szempont a [TKP_FI_002] számú követelmény volt. A követelmény szerint az egyes NVRAM blokkoknak jól megkülönböztethetőnek kell lennie, mivel, mint a [TKP_FI_003D] követelményben látszik, sikeres olvasás esetén az olvasott blokkok tartalmát össze kell hasonlítani a blokkok előre meghatározott tartalmával. Ehhez mindenképpen szükséges, hogy a blokk tartalmát újra elő lehessen állítani. Célszerű lenne, ha a blokkok tartalma valamilyen determinisztikus véletlenszerű érték lenne, mivel így biztosítható, hogy egy flashmemóriaképen belül a blokkok tartalma eltérő. A követelmény még előírja, hogy a mindegyik flashmemóriaképnek tartalmaznia kell legalább egy blokkot mind a három blokktípusból.

Az említett megfontolások miatt végül úgy döntöttem, hogy az egyes NVRAM blokkok tartalmát egy pszeudovéletlenszám-generátor segítségével hozom létre. A pszeudovéletlenszám-generátor olyan véletlenszám generátor, amely vár egy adott értéket (seed értéket), és az adott értékhez mindig ugyanazt a véletlenszerű számsorozatot generálja. Az egyes NVRAM blokk példányokhoz hozzá kell rendelni egy seed értéket, amellyel létrehozható a blokk véletlenszám tartalma. A seed érték segítségével ez a tartalom reprodukálható lesz.

A flashmemóriaképek létrehozásánál szükséges tehát implementálni egy pszeudo véletlenszám generátort, valamint meg kell valósítani az NvM egy olyan konfigurációját, melyben az összes NVRAM blokktípus szerepel.

3.3 A tesztfuttató szoftver megtervezése

A tesztfuttató feladata előállítani az elvégezni kívánt tesztlépéseket, és az egyes tesztlépések paramétereit. A tesztlépések előállítása után a tesztfuttató végighalad az egyes tesztlépéseken és továbbítja a tesztlépés paramétereit a tesztvégrehajtónak. A tesztvégrehajtó, miután végrehajtotta a tesztlépést, visszaküldi az eredményt a tesztfuttatónak, ami eldönti, hogy az adott tesztlépés sikeres vagy sikertelen volt. Ha sikertelen volt, a tesztnek le kell állnia, ellenkező esetben tovább kell haladnia a következő lépésre. Sikertelen tesztlépés esetén a tesztnek azért kell leállnia, mert ilyen esetben vagy a flash memória sérült meg, tehát a célprocesszor azonnali cseréje szükséges, vagy a memóriastack implementációban olyan jellegű hiba van, amit azonnal meg kell vizsgálni.

3.3.1 A meglévő rendszer ismertetése

A tesztfuttató szoftver tervezésénél a cég által használt célzott hibainjektálást végző tesztkörnyezetből indultam ki. A projekt tesztfuttató szoftvere JAVA nyelven volt implementálva, és Junit tesztelési keretrendszert használt. A program tesztlépéseket tartalmazó fájlokat dolgozott fel. Egy fájlban tesztlépések egy sorozata volt szövegesen megadva, a Junit környezet pedig ezeket a tesztlépéseken végighaladva végezte el a tesztet. Példa egy ilyen tesztlépésre:

```
Tesztparancs,Tesztparaméter,Tesztparaméter,Tesztparaméter,Várt_teszteredmé
ny, Tesztlépés_leírás
```

3.3.2 A Junit tesztelési keretrendszer

A későbbi megvalósítás miatt fontos megérteni, hogyan működnek a Junit parametrizált tesztek általánosságban, valamint a memóriastack teszt szempontjából, ezért ezeket részletesen ismertetem. Egy Junit parametrizált teszt 4 fő részből áll:

- Before
- Parametrized
- Test
- After

Before

A Before részben kell meghívni az egyes osztályok konstruktorait, valamint megadni, milyen test suitokon szeretnénk elvégezni a tesztet. Az Ethernet kapcsolatot a tesztvégrehajtóval ebben a részben építi ki a tesztfuttató.

Parametrized

A Parametrized részben kell a paraméterlistát létrehozni, tehát itt történik a tesztlépéseket tartalmazó fájl feldolgozása, a tesztparaméterek meghatározása, a paramétereket tartalmazó objektum létrehozása és feltöltése a tesztparaméterekkel.

Test

A Test lépésben hajtódnak végre az egyes tesztlépések. A tesz-futtató elküldi a paramétereket a tesztvégrehajtónak, majd megvárja a választ. A válasz alapján megállapítja a tesztlépés sikerességét. Fontos kiemelni, hogy a tesztek elvégzése előtt a Junit előre meghatározza hány tesztlépésből fog állni a teszt, és ezek paramétereit előre betölti a memóriába.

After

Az After részben kerül sor a tesztriport létrehozására, és az Ethernet kapcsolat bontására.

3.3.3 A hibainjektor tesztfuttató megtervezése

A hibainjektor teszt futtatójánál úgy döntöttem, hogy felhasználom a már meglévő szoftver elemeit, azokat megfelelően átalakítva. A hibainjektor szoftver esetén elegendő egyetlen tesztparancs, a tesztparaméterek pedig a következők:

- flashmemóriakép azonosító
- bájt pozíció
- 8 bájtos maszk
- elvárt teszteredmény

A flashmemóriakép azonosító szolgál a használni kívánt flashmemóriakép kiválasztására. Értéke a [0,4] zárt intervallumban értelmezett, mivel öt flashmemóriakép közül lehet választani. A bájt pozíció paraméter szolgál egy 8 bájtos memóriaterület kiválasztására az adott flashmemóriaképen. A paraméter a memóriaterület virtuális kezdőcímre mutat. Értéke a [0,32767] zárt intervallumban mozog, mivel egy flashmemóriakép 32 kilobájtos. A 8 bájtos maszkkal adható meg a bitinverzió pontos helye. A teszt egybites hibainjektálást végez, így a maszknak mindig pontosan egy bitje 1-es a többi pedig 0. A bitinverzió kizáró vagy (XOR) művelettel lesz megvalósítva. Az elvárt teszteredmény paraméterrel beállítható, hogy a tesztvégrehajtó által visszaküldött teszteredmény mely értékénél tekinthető a tesztlépés sikeresnek. Ezek alapján egy tesztlépés a következőképpen írható le szövegesen:

```
FI_BITINVERSION,0,32,0x0000000000000001,FI_OK
```

A tesztkörnyezetnél említettek alapján minden egyes hibainjektálás után szükséges a mikrovezérlő resetelése, valamint utána való várakozás. Ezekre a műveletekre szükséges még 2 parancs felvétele, így egy tesztlépés összesen 3 parancsból áll. Ez összesen 3*32728/8*64 = 786432 parancs flashmemóriaképenként. Mivel ez igen nagy fájlméretet eredményezne a build szerveren, ezért inkább egy újabb parancs felvételét tartottam szükségesnek. A felvett parancs után a parancsokat ciklikusan kell feldolgozni, a parancsban megadott számszor. Például:

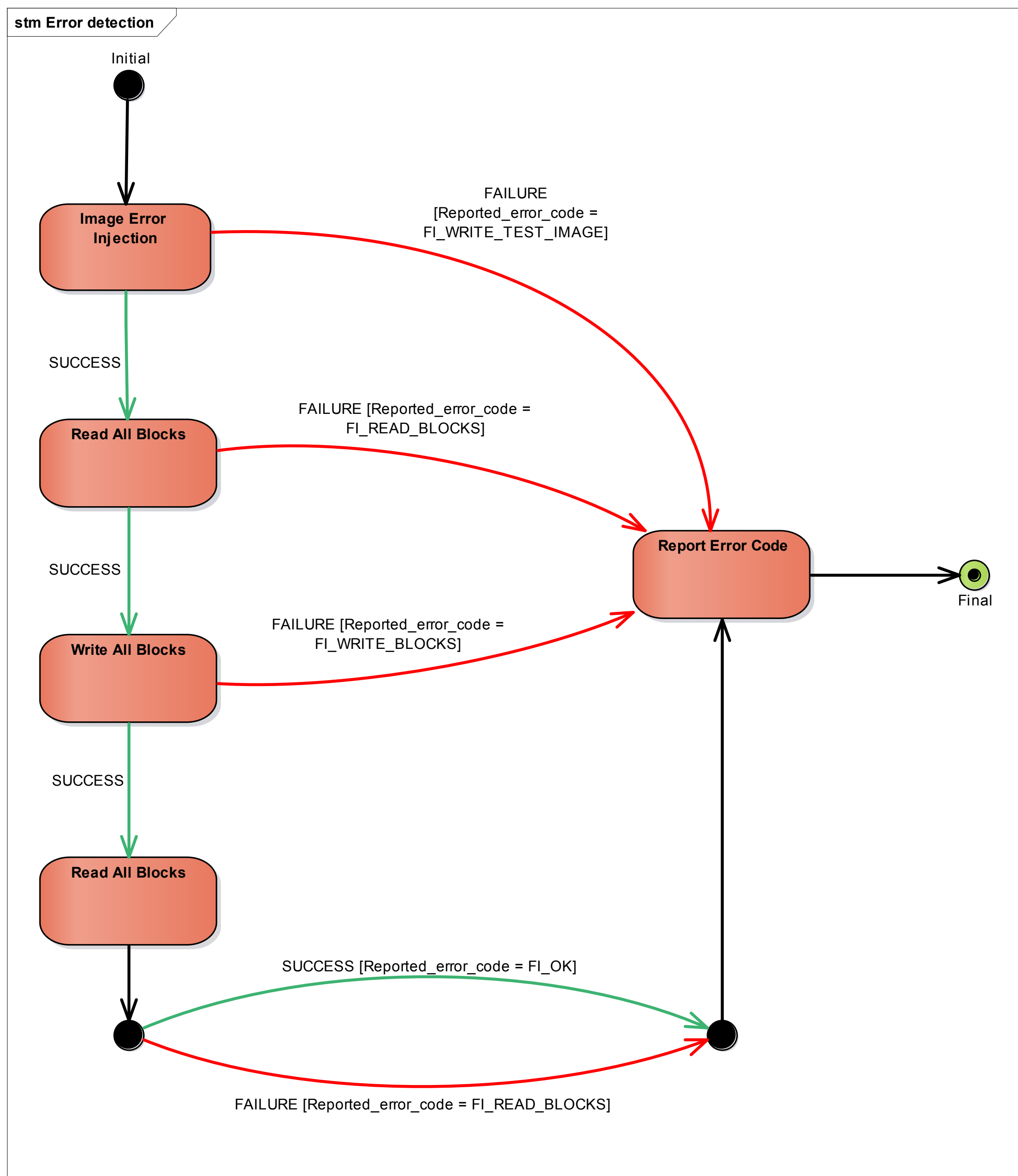
```
CP_FOR,4
FI_BITINVERSION,0,BYTEPOS,PATTERN,FI_OK
RESET
CP_WAIT,2500
CP_ENDFOR
```

A példában említett esetben a CP_FOR és a CP_ENDFOR utasítások közötti részt 4-szer kell feldolgozni.

3.4 A hibainjektor szoftver megtervezése

A hibainjektor szoftver tervezését a [TKP_FI_003] követelmény alapján végeztem. Látható, hogy ez a követelmény egy lépésről-lépésre leírás, amely meghatározza, hogy az adott flashmemóriaképen egy tesztlépés során milyen műveleteket kell végezni, valamint, hogy az egyes műveleteknek milyen kimenetelei lehetnek, és ezekben az esetekben mi a teendő. A követelmények alapján megtervezett szoftver állapot-átmeneti diagramját a 3.6. ábra szemlélteti.

A szoftver kezdeti állapotból Image Error Injection állapotba kerül. Ebben az állapotban valósul meg a hibainjektálás. Ha a hibainjektálás sikeres volt, akkor az állapotgép Read All Blocks állapotba kerül, ha nem, akkor a hibakódot FI_WRITE_TEST_IMAGE értékre állítva az állapotgép Report Error Code állapotba kerül. Read All Blocks állapotban kerül felolvasásra az összes flashmemóriakép által tárolt NVRAM blokk, valamint sikeres olvasás esetén a felolvasott értékek ebben az állapotban lesznek összehasonlítva az előre meghatározott értékekkel. Az összehasonlítás azért szükséges, mert az olyan NVRAM blokknál, melyekhez nincs CRC rendelve, az NVM a hibainjektálás után nem jelez hibát. Ha mindkét művelet sikeres volt, akkor az állapotgép Write All Blocks állapotba kerül, ellenkező esetben a hibakódot FI_READ_BLOCKS értékre állítva az állapotgép Report Error Code állapotba kerül. Write All Blocks állapotban kerül kiírásra az összes írásra konfigurált NVRAM blokk. Sikeres művelet esetén az állapotgép a második Read All Blocks állapotba kerül, egyébként a hibakódot FI_WRITE_BLOCKS értékre állítva az állapotgép Report Error Code állapotba kerül. A második Read All Blocks állapotban felolvasásra kerül az összes NVRAM blokk, ami olvasásra vagy írásra lett konfigurálva, majd még ebben az állapotban összehasonlítjuk azoknak a blokkoknak a tartalmát, amelyek nem lettek írva, az előre létrehozott blokktartalmakkal. Ha mindkét művelet sikeres volt, akkor a hibakód FI_OK értéket vesz fel, jelezve az egész tesztlépés sikerességét, ellenkező esetben a hibakód FI_READ_BLOCKS értéket vesz fel. Mindkét esetben a Report Error Code állapotba kerül az állapotgép. Report Error Code állapotban a tesztlépés során előállt hibakód kerül kiküldésre a tesztfuttató számára.



3.6. ábra. Hibainjektor állapot-átmeneti diagram

3.5 A Serial Transport Protocol modul

A 2. fejezetben a tesztkörnyezetnél már ismertetésre került, hogy a tesztfuttató és a tesztvégrehajtó Etherneten, és UART-on keresztül kommunikál. A teszt hosszú lefutási ideje, és a DUT-on lévő célprocesszor UART port-ja, és a Raspberry Pi Ethernet port-ja között átvitt adatok integritásának védelme érdekében, indokolt az átviteli közegek feletti hibatűrő átvitel. A DUT és a Raspberry Pi Ethernet-en keresztül kommunikál, itt a TCP megfelelő hibatűrést biztosít, így ezt a protokollt alkalmaztam. A DUT-on lévő célprocesszor UART port-ja és az UART/Ethernet átalakító között nincs biztosítva a

hibatűrő átvitel, így terveztem egy hibatűrő protokollt, amely a Serial Transport Protocol (STP) nevet viseli. A következő alfejezetekben bemutatom a protokollal szemben támasztott követelményeket, és a protokoll tervezési folyamatát.

3.5.1 A Serial Transport Protocollal szemben támasztott követelmények

[TKP_STP_001]

Az STP modul kizárólag inicializálás után végezhet műveleteket.

[TKP_STP_002]

A modul inicializálását az STP_Init függvénnyel kell végrehajtani.

[TKP_STP_003]

Minden szolgáltatásinterfésznek, amelyeket az STP a felsőbb rétegnek nyújt, függetlennek kell lennie a modul belső kommunikációjától, konfigurációjától, és implementálásától.

[TKP_STP_004]

Üzenetküldést a modul STP_Send függvényével kell kezdeményezni.

[TKP_STP_005]

Az STP_Send függvénynek 2 paraméterrel kell rendelkeznie, a küldött adatok kezdőcímére mutató pointerrel, és a küldött adatok méretével.

[TKP_STP_006]

Az STP_Send függvénynek jeleznie kell, ha nem tud újabb üzenetküldési kérést elfogadni, mivel még az előző kézfogásos átvitel folyamatban van. Ekkora a függvénynek E_NOT_OK értékkel kell visszatérnie.

[TKP_STP_007]

A küldött üzeneteknek a következő adatelemekből kell állnia: HEADER, PAYLOAD, és PAYLOADCRC.

[TKP_STP_008]

A HEADER résznek a következő paramétereket kell tartalmaznia: SequenceNumber, MessageType, Length, HeaderCrc.

[TKP_STP_009]

A SequenceNumber paraméternek tartalmaznia kell az üzenet sorszámát.

[TKP_STP_010]

A MessageType paraméternek jeleznie kell, hogy a küldött üzenet egy adat üzenet, vagy egy nyugtázó üzenet.

[TKP_STP_011]

A Length paraméternek a PAYLOAD méretét kell megadnia bájtokban.

[TKP_STP_012]

A HeaderCrc paraméter értékének a SequenceNumber, MessageType, Length paraméterekre számolt 8 bites CRC-nek kell lennie.

[TKP_STP_013]

A PAYLOAD résznek kell tartalmaznia az elküldött felhasználói adatokat.

[TKP_STP_014]

A PAYLOADCRC résznek, a PAYLOAD részre számolt CRC értéknek kell lennie. A számolt CRC hosszának konfigurálhatónak kell lennie.

[TKP_STP_015]

Az STP modulnak ellenőriznie kell, hogy az érkezett üzenet egy nyugtázó üzenet egy elküldött üzenetre, vagy egy adat üzenet, melyet nyugtázni kell.

[TKP_STP_016]

Egy fogadott üzenet sikeres feldolgozása esetén a modulnak értesítenie kell a felsőbb réteget a felsőbb réteg üzenetfogadást jelző értesítési függvénye segítségével.

[TKP_STP_017]

A felsőbb réteg üzenetfogadást jelző értesítési függvényének 2 paraméterrel kell rendelkeznie, a fogadott adatok kezdőcímére mutató pointerrel, és a fogadott adatok méretével.

[TKP_STP_018]

Ha nem érkezik helyes nyugtázó üzenet egy konfigurált időintervallumon belül, az STP-nek újra kell küldenie az előzőleg kiküldött üzenetet ugyanazzal a tartalommal, de csak

akkor, ha van még újraküldési lehetőség. Abban az esetben, ha nincs már újraküldési lehetőség, a modulnak a felsőbb réteg üzenetküldést nyugtázó értesítési függvénye segítségével negatív visszajelzést kell jeleznie a felsőbb réteg számára.

[TKP_STP_019]

Sikeres üzenetátvitel esetén az STP-nek a felsőbb réteg üzenetküldést nyugtázó értesítési függvénye segítségével pozitív visszajelzést kell jeleznie a felsőbb réteg számára.

[TKP_STP_020]

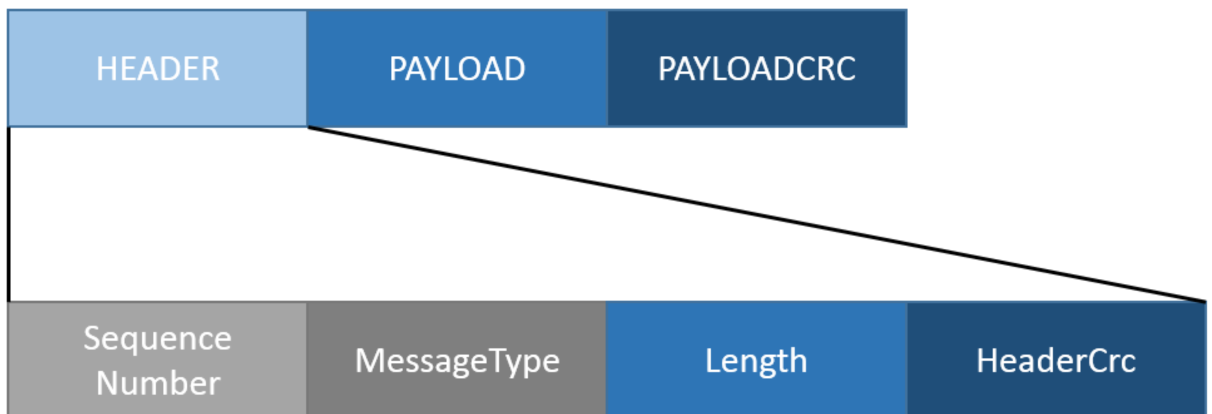
A felsőbb réteg üzenetküldést nyugtázó értesítési függvényének, egy paraméterrel kell rendelkeznie, amellyel az üzenetátvitel sikerességét jelzi.

[TKP_STP_021]

Az újraküldési lehetőségek számának, amely a [TKP_STP_018] követelményben kerül említésre, konfigurálhatónak kell lennie.

3.5.2 A Serial Transport Protocol megtervezése.

3.5.2.1 Csomagfelépítés



3.7. ábra. Csomagfelépítés

A követelmények részletesen leírják az STP által küldött üzenetek felépítését, ezt a 3.7. ábra szemlélteti. A követelmények nem térnek ki minden egyes paraméter méretére, így ezt ebben az alfejezetben ismertetem. A HEADER rész 5 bájtból áll. A SequenceNumber paraméter egy sorszám, amellyel ellenőrizhető a fogadott nyugtázó üzenet helyessége. A helyesség ellenőrzése során a fogadott nyugtázó üzenet SequenceNumber paramétere, valamint a küldött üzenet SequenceNumber paramétere kerül összehasonlításra. A két érték egyezése esetén a fogadott nyugtázó üzenet helyes.

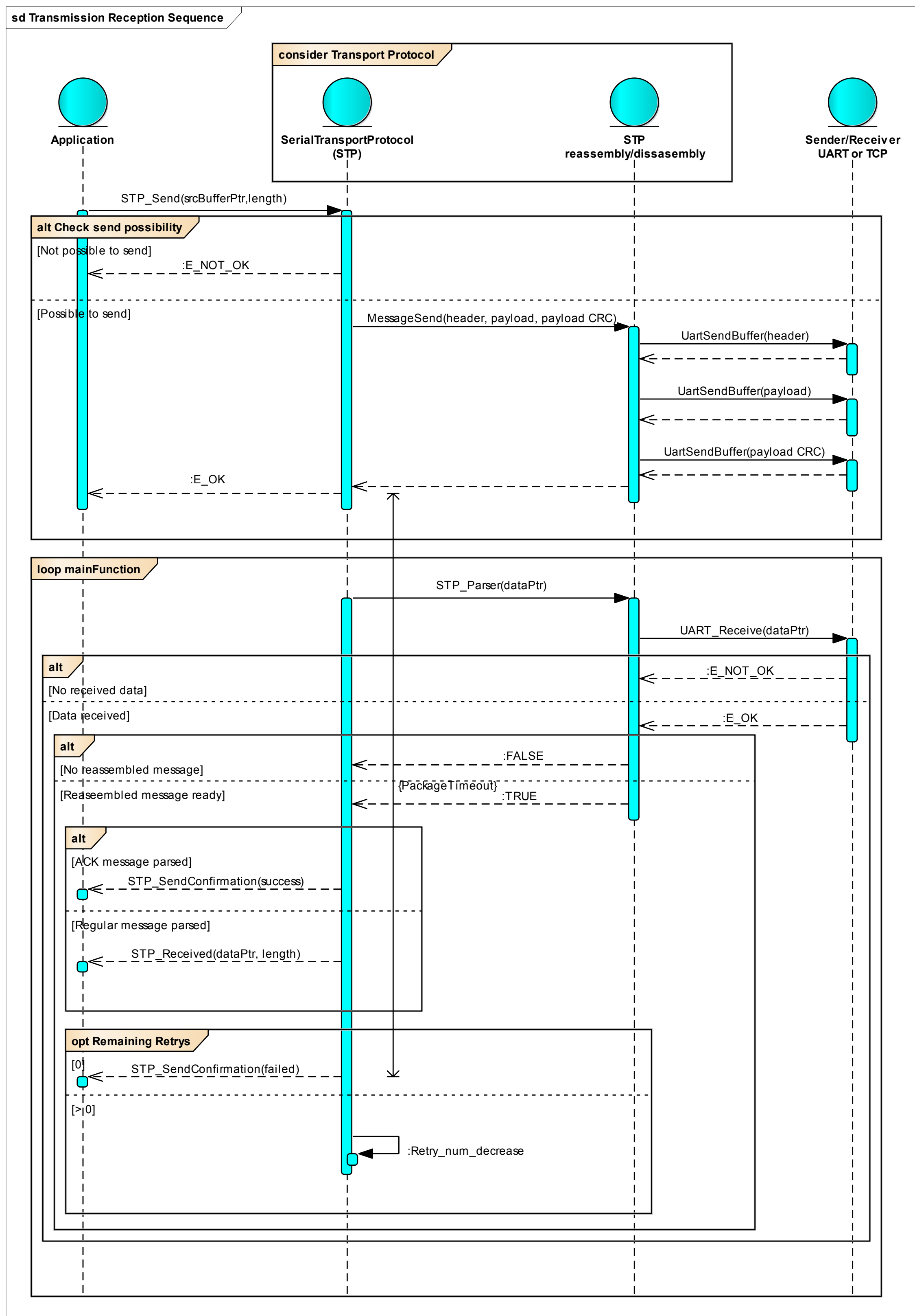
A paraméter méretét elegendőnek tartottam egy bájtosra felvenni, mivel a túlcsordulás nem okoz gondot a helyesség ellenőrzésénél. A MessageType paraméter az üzenet típusát tartalmazza. Kétféle típusú üzenetet definiál a követelmény: adat, és nyugtázó. A két típusra elegendő egy-egy bájtos értéket definiálni, így a MessageType paraméter mérete egy bájt. A Length paraméter a PAYLOAD rész méretét tartalmazza. A hibainjektálás során a küldött üzenetek adatának mérete nem fogja meghaladni a 2^8 értéket, de az STP modul tervezésénél figyelembe vettem a későbbi újrafelhasználás lehetőségét a cégen belül, így a paraméter mérete két bájt. A HeaderCrc paraméter mérete a követelményekből következően egy bájt. A PAYLOAD rész mérete változó, de a Length paraméter mérete egyértelműen behatárolja mekkora intervallumok között mozoghat ez a méret. Ebben az esetben a PAYLOAD mérete a [0,65535] zárt intervallumban mozoghat bájtban megadva. A követelményekből látható, hogy a PAYLOADCRC mérete konfigurálható, így ezt a 3.5.2.3. fejezetben ismertetem.

3.5.2.2 Az üzenetátvitel megtervezése

A 3.8. ábra szemlélteti az átvitel UML szekvenciadiagramját. Az ábrán végigkövethető az üzenetátvitel folyamata. Az ábrán az üzenetfogadást jelző értesítési függvény az STP_Received, az üzenetküldést nyugtázó értesítési függvény az STP_SendConfitmation.

Az üzenetváltást mindig az egyik fél kezdeményezi az STP_Send API függvénnyel, amelyben az elküldeni kívánt adatok kezdőcímére mutató pointert, valamint ezen adatok méretét kell megadni bájtban. Ha éppen folyamatban van egy átvitel az API függvény E_NOT_OK értékkel tér vissza. Ha nincs átvitel folyamatban, akkor elfogadja az üzenetküldési kérést, összeállítja a HEADER és a PAYLOADCRC részeket, és szinkron módon továbbítja az üzenet összes részét. Az STP bizonyos időzítéssel lekérdezi az alatta lévő rétegtől, hogy érkezett-e üzenet. Ha a helyes nyugtázó üzenet konfigurált időintervallumon belül nem érkezik meg, akkor az STP újraküldi az üzenetet, de csak ha van még újraküldési lehetősége. Üzenetfogadás esetén az STP megállapítja az üzenet típusát. Helyes nyugtázó üzenet esetén a modul a felsőbb réteg üzenetküldését nyugtázó értesítési függvényének meghívásával jelzi az üzenetátvitel sikerességét. Adatokat tartalmazó üzenet beérkezését a modul a felsőbb réteg üzenetfogadást jelző értesítési függvényének meghívásával jelzi. Az újraküldések száma, ha meghaladja a konfigurált értéket, akkor az STP nem próbálja meg többször elküldeni az üzenetet, hanem

visszajelez a küldő fél számára az üzenetküldést nyugtázó értesítési függvény meghívásával, hogy az átvitel sikertelen volt.



3.8. ábra. Az STP UML szekvenciadiagramja

3.5.2.3 A modul konfigurálhatósága

A modulnak négy paramétere konfigurálható, melyek a következők:

- Újraküldések száma
- Várakozási idő a válaszra - PackageTimeout
- Várakozási idő egy bájtra - ByteTimeout
- PayloadCrc mérete

Az újraküldések száma határozza meg, hogy helyes nyugtázó üzenet hiányában az adott üzenetet hányszor legyen újra elküldve, tehát a tervezésnél említett újraküldési lehetőségek kezdőértéke. A várakozási idő a válaszra meghatározza, mennyi idő telhet el maximálisan a nyugtázó üzenet megérkezéséig egy üzenet elküldése után, mielőtt újra küldenénk. A várakozási idő egy bájtra meghatározza, mennyi idő telhet el maximálisan a beérkezett üzenet feldolgozása során két egymást követő beérkező bájt között. Ha két bájt fogadása között hosszabb idő telik el, mint ez a konfigurált érték, akkor az üzenet feldolgozása sikertelen. A PayloadCrc mérete megadja, hogy az üzenet PAYLOAD részére hány bites CRC legyen számolva. Ez a méret lehet 8, 16, vagy 32 bit.

4 Megvalósítás

4.1 A flashmemóriaképek előállítása

4.1.1 A pszeudovéletlenszám-generátor megvalósítása

A pszeudovéletlenszám-generátort egy önálló mellékmodulként hoztam létre, mivel mind a flashmemóriaképek létrehozásánál, mind a hibainjektálás összehasonlító lépésénél alkalmaztam. A genetátor két adattal dolgozik, egy seed értékkel, és egy hossz értékkel. A hossz értéke határozza meg a véletlenszerű bájtsorozat hosszát. Az algoritmus a paraméterül kapott seed értéken végez műveleteket, főleg prímszámokkal való szorzást, valamint eltolást.

4.1.2 Az NvM blokkok megválasztása

A követelményeknek és a tervezésnek megfelelően minden egyes NVRAM blokk típust fel kell használni a flashmemóriaképek előállításánál. Ezek a típusok a következők: Native blokk, Redundáns blokk, Dataset blokk, írásvédett Native blokk, írásvédett Redundáns blokk, írásvédett Dataset blokk. Az NVRAM blokk példányok közül úgy érdemes választani, hogy a blokkok mérete egy széles skálán változzon, választani kell tehát kisebbméretű, 8-16 bájtos blokkokat, közepes méretű, ~1 kilobájtos blokkokat, és minimum egy nagyméretű, több mint 7 kilobájtos blokkot. Az NvM konfigurálásánál az összes lehetséges CRC méretet legalább egy blokknál használni kell.

4.1.3 A képek létrehozása

A flashmemóriaképeket egy meglévő, - a cég által használt - szoftver segítségével hoztam létre, mellyel lehetséges NVRAM blokkok kiírása a flash memóriába. A 3.2.3. fejezetben említett tervezésnek megfelelően az NVRAM blokkok véletlenszerű tartalma egy egyedi seed érték segítségével került létrehozásra, így biztosítva a blokkok tartalmának egyediségét.

A létrejött flashmemóriakép tartalmakat a flash memóriából egy debugger segítségével bináris formában kiolvastam, majd átkonvertáltam a hibainjektor szoftver C nyelvű forráskódjába beilleszthető tartalommá. A hibainjektor szoftver C nyelvű forráskódjában az említett tartalmak konstans bájttömbök. Mind az öt flashmemóriaképhez tartozik egy ilyen tömb. A teszt futása során a tesztfuttató által

elküldött flashmemóriakép azonosító megadja, hogy az öt tömb tartalma közül melyiket kell betölteni a RAM-ba. A hibainjektálás a RAM-ba betöltött flashmemóriakép tartalmon lesz elvégezve, majd ez a hibainjektált tartalom kerül kiírásra a flash memóriába.

Az NVRAM blokkok véletlenszerű tartalmának reprodukálhatónak kell lennie. Ehhez szükséges volt mindegyik flashmemóriaképhez eltárolni a hozzá tartozó NVRAM blokk azonosítókat, és a blokkokhoz tartozó seed értéket, mert így a blokk véletlenszerű tartalma a hibainjektor szoftver futása során újra előállítható.

4.2 A tesztfuttató szoftver megvalósítása

4.2.1 A tesztfuttató szoftver megvalósítása Junit parametrizált tesztként

A tervezésnek megfelelően a tesztfuttató szoftver a létrehozott tesztlépéseket tartalmazó fájlokat dolgozza fel. A tesztparancsok egy CP_FOR-CP_ENDFOR parancspár között vannak, így feldolgozásnál a bájt pozíció és maszk értékeknél a szoftver határozza meg a megfelelő, soron következő értéket. A parancsból feldolgozott egyes tesztparaméterek egy Junit objektumban vannak eltárolva. A Junit objektumtípusból szükséges létrehozni egy listát, amely tartalmazza az összes tesztlépést, és az egyes tesztlépésekhez tartozó összes paramétert. A Junit teszt ezután ezen az objektumlistán végzi el a tesztet. Mint már a 3.3.2-es fejezetben említettem, a Junit a teszt elején ezt az egész objektumlistát betölti a memóriába. Ennek az objektumlistának a mérete 32768*8*5 = 1310720 objektum. A tesztfuttató Raspberry Pi-on történő előzetes futtatása során a nagyméretű lista miatt a JVM túl kevés memóriára hivatkozva kivételt generált, így más megoldást kellett keresnem a tesztfuttató implementálására.

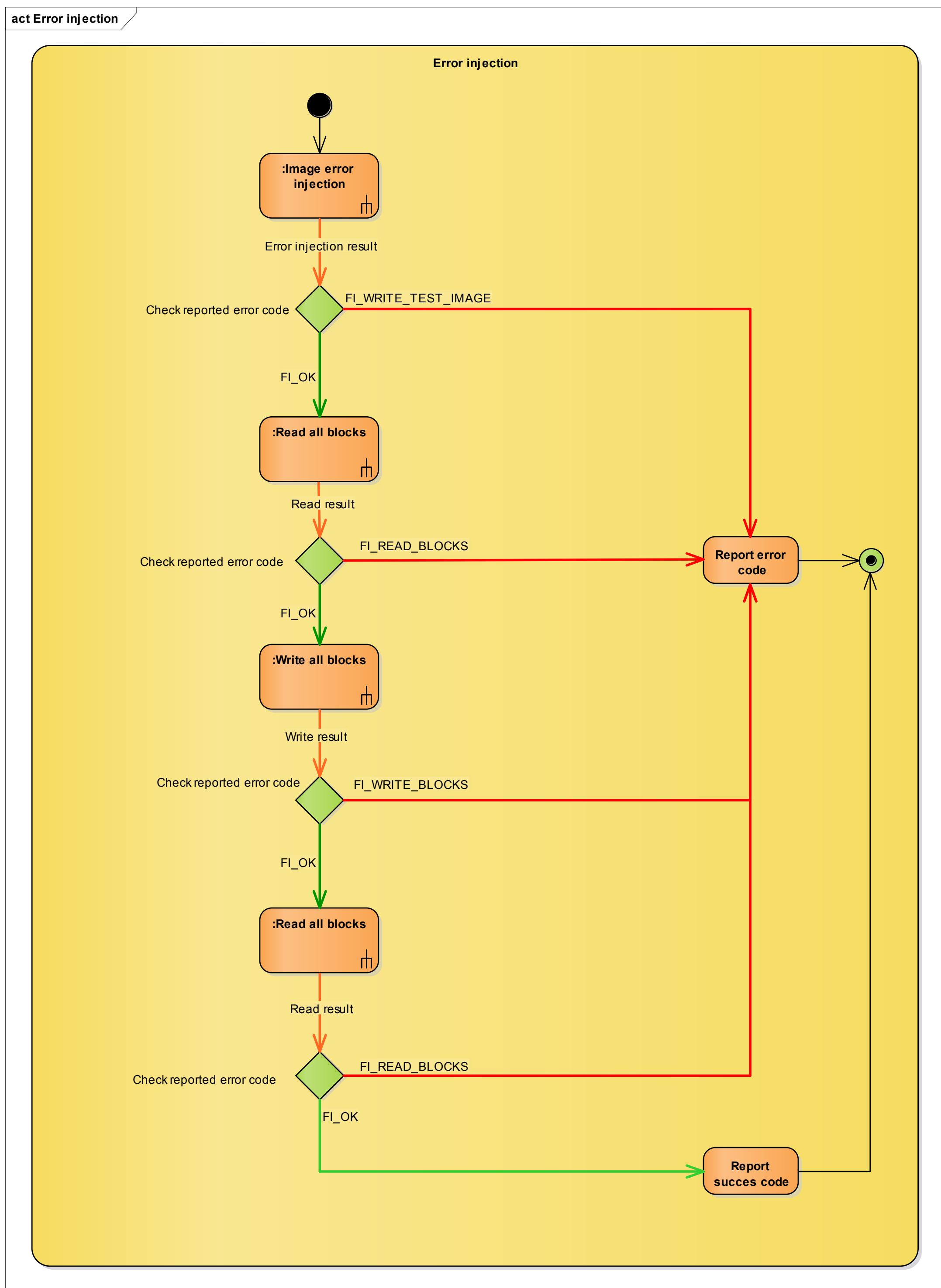
Egy másik megoldásként próbáltam a tesztet több tesztesetre bontani. A már megírt fájlfeldolgozó algoritmust felhasználva legeneráltam több olyan tesztlépéseket tartalmazó fájlt, amelyekbe CP_FOR-CP_ENDFOR parancspár nélkül vannak a tesztparancsok. A megoldás mögötti elgondolás az volt, hogy a tesztlépéslistát kisebb egyenlő részekre bontva a tesztlépéslistához tartozó Junit objektumlista már elfér a RAM-ban. Az ugyanabban a fájlban szereplő tesztlépések így egy teszt case-nek feleltek meg a Junit teszten belül. A futtatás során a JVM ugyanazt a kivételt generálta, mint az előző esetben, mivel a Junit nemcsak az egyes teszt case-eknek a paramétereit tölti be a memóriába, hanem az egész tesztét.

4.2.2 A tesztfuttató szoftver megvalósítása Junit nem parametrizált tesztként

A Junit nem parametrizált teszt előnye, hogy nem tölti be a RAM-ba az összes tesztlépés paramétereit, így nem foglal le előre memóriát. Hátránya, hogy így csak egyetlen Junit tesztlépésből áll az egész teszt. Parametrizált teszt esetén az egyes tesztlépések elkülönülnek, így a Junit tesztjelentésben láthatóak az egyes tesztlépéshez tartozó paraméterek és a tesztlépés sikeressége. Nem parametrizált teszt esetén a Junit jelentésben nem kapjuk meg az egyes tesztlépések sikerességére vonatkozó információt, mivel a Junit teszt egyetlen tesztlépésből áll. Az említett hátrány a teszt szempontjából nem okoz gondot, mivel a teszthez egy saját tesztjelentés is készül, így a továbbiakban nincs szükség a Junit által létrehozott jelentésre. A létrehozott tesztkörnyezet indokolttá tette, hogy a tesztfuttató Junit nem parametrizált tesztként legyen létrehozva, így ennek megfelelően implementáltam a tesztfuttatót.

4.3 A hibainjektor szoftver megvalósítása

A hibainjektor megvalósításánál a 3.4. fejezetben tervezett állapotgépből indultam ki. A megvalósított algoritmusnak nincsenek állapotai, sorrendi szoftverként lett implementálva. A modul folyamatábráját a 4.1. ábra szemlélteti.



4.1. ábra. A hibainjektor folyamatábrája

A folyamatábrán látszik, hogy a hibainjektor 4 lépésből áll: a hibainjektálásból, az NVRAM blokkok felolvasásából, az NVRAM blokkok írásából, majd az NVRAM blokkok újbóli felolvasásából. Az egyes műveletek csak akkor végződnek el, ha az előttük lévő összes művelet sikeres volt, tehát ha a jelentett hibakód értéke FI_OK. A műveletek elvégzése előtt a jelentett hibakód értéke FI_OK. Ha a második olvasási művelet után a

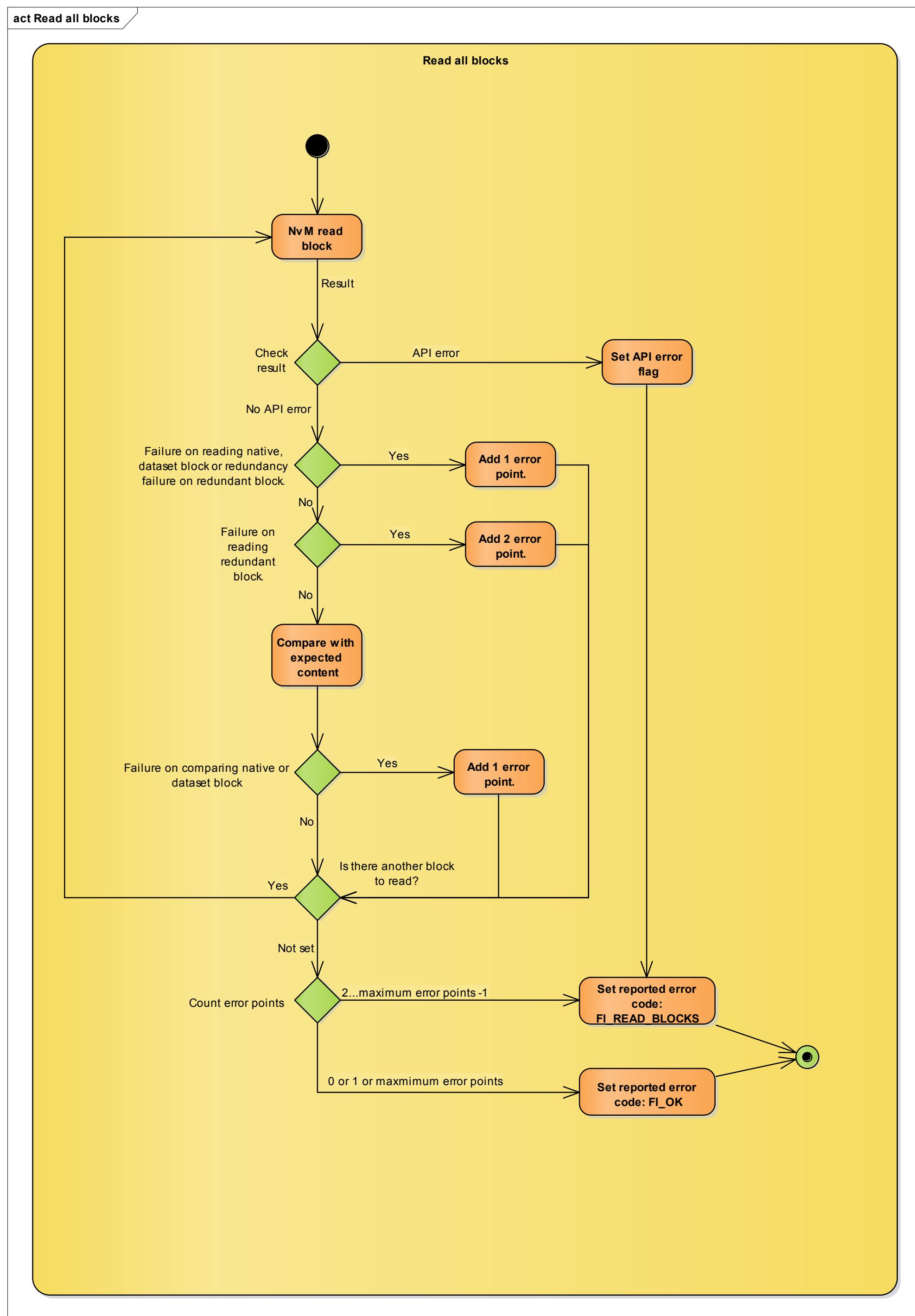
jelentett hibakód értéke FI_OK, akkor az összes művelet sikeres volt, tehát az adott tesztlépés sikeres. A következő alfejezetekben részletesen bemutatom az egyes lépéseket.

4.3.1 A hibainjektálás

A hibainjektálás a TKP_FI_003A követelmény alapján lett megvalósítva. A követelménynek megfelelően a hibainjektálás a RAM-ban lett elvégezve. A művelet során a paraméterül kapott flashmemóriakép azonosítóhoz tartozó memóriakép betöltésre kerül a RAM-ba. A betöltött kép kezdőcíme és a paraméterül kapott bájt pozíció összege megadja, melyik memóriacímtől kezdődően kell a következő 8 bájtot bitenkénti kizáró-vagy (XOR) kapcsolatba hozni a paraméterül kapott 8 bájtos maszkkal. A RAM-ban lévő flashmemóriakép hibainjektálása után a kép kiírásra kerül a flash memóriába. A követelmény megjegyzése alapján csak azt a poolt kell felülírni, amelyben változás történt, a másikat nem. Ehhez szükséges összehasonlítani a RAM-ban lévő flashmemóriakép pooljainak a tartalmát a flash memóriában lévő poolok tartalmával a Flash Driver segítségével. Az összehasonlítás után a nem egyező pool tartalma a Flash Driver segítségével törlésre kerül, majd az új tartalom a Flash Driver segítségével kiírásra kerül a flash memóriába. A függvény sikeres írási művelet esetén E_OK-kal, sikertelen írási művelet esetén E_NOT_OK-kal tér vissza. Sikertelen írási művelet esetén a jelentett hibakód értéke FI_WRITE_TEST_IMAGE lesz, egyébként nem változik.

4.3.2 Az NvM blokkok olvasása.

Az NvM blokkok olvasására a hibainjektálás során kétszer is sor kerül, de a két esetben különböző blokkokon kell elvégezni az említett műveleteket. Mivel az olvasás folyamata ugyanaz a blokkok írása előtt és után, így ez a két művelet egy függvényként lett megvalósítva. A megfelelő blokkok kiválasztása egy-egy utility függvényként lett implementálva, melyek az olvasási műveletek előtt hívódnak meg. Az első művelet során az adott flashmemóriakép létrehozásánál használt NVRAM blokk tartalmakat kell felolvasni és összehasonlítani a blokk előre definiált tartalmával. A második művelet során azokat az NVRAM blokk tartalmakat kell felolvasni, melyek használva voltak az adott flashmemóriakép létrehozásánál, vagy írásra van konfigurálva. Összehasonlítani azokat a blokkokat kell, melyek használva voltak az adott flashmemóriakép létrehozásánál, de nincsenek írásra konfigurálva Az olvasási és összehasonlítási folyamatot a 4.2. ábra szemlélteti.



4.2. ábra. Az olvasás és összehasonlítás művelet folyamatábrája

Az egyes NVRAM blokkok olvasására és összehasonlítására egy ciklusban kerül sor. Ha a művelet során bármely API függvény negatív visszajelzést ad, a ciklus megszakad, és a művelet sikertelen.

Sikertelen blokkolvasás esetén a blokk típusától függően növekszik egy hibapontszámláló. A hibapont számláló megadja, hogy az olvasás és összehasonlítás során hány blokk olvasása vagy összehasonlítása volt sikertelen. Nativ és Dataset blokk

esetén, ha az olvasás sikertelen, akkor egy hibapont adódik a számlálóhoz, mivel ilyenkor csak egy blokknyi adat felolvasása sikertelen. Redundáns blokk esetén, ha az olvasási művelet eredménye REDUNDANCY_FAILED, akkor egy hibapont kerül hozzáadásra a számlálóhoz, mivel ilyenkor a két NVRAM blokk közül csak az egyik felolvasása sikertelen. Redundáns blokk esetén, ha az olvasási művelet eredménye nem FI_OK vagy REDUNDANCY_FAILED, akkor 2 hibapont adódik a számlálóhoz, mivel akkor az egész redundáns blokkpár olvasása sikertelen volt.

Összehasonlítás csak sikeres olvasás után történik, és kizárólag Native és Dataset blokkokon. Redundáns blokk esetén nincs szükség összehasonlításra, mivel az NvM a redundanciát a két FEE blokk adatának összehasonlításával végzi. Az összehasonlított blokkok tartalmának eltérése esetén egy hibapont adódik a számláló értékéhez.

A művelet végeredménye a hibapontszámláló értékétől függ. A számláló 0 értéke esetén a művelet végeredménye sikeres. A számláló 1 értéke esetén a művelet sikeres, mivel az FEE blokkhoz tartozó leíró, vagy az FEE blokk adatának hibainjektálásakor a blokkolvasásnak sikertelennek kell lennie. Ha a számláló értéke megegyezik az olvasott blokkok számával a művelet sikeres, mivel a poolleíró hibainjektálása során a poolnak érvénytelen állapotot kell felvennie, és a poolban lévő blokkon végzett műveleteknek sikertelennek kell lennie. Az előzőekben felsoroltaktól eltérő hibaszámláló érték esetén a művelet eredménye sikertelen. Sikertelen olvasási művelet esetén a jelentett hibakód értéke FI_READ_BLOCKS lesz, egyébként nem változik.

4.3.3 Az NvM blokkok írása

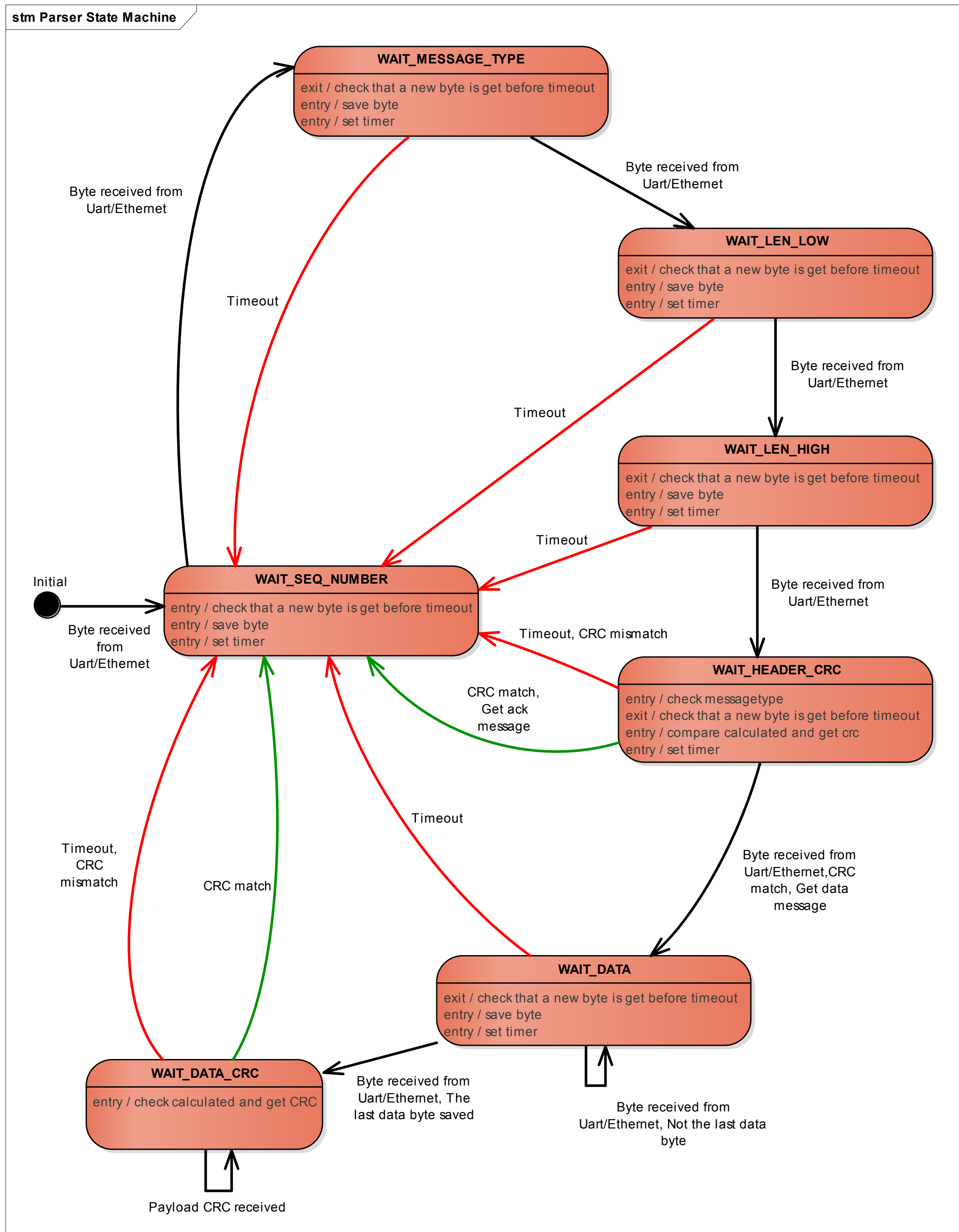
Az írásra konfigurált NVRAM blokkok írására egy ciklusban kerül sor, hasonlóan az olvasási művelethez. Ha a művelet során bármely API függvény negatív visszajelzést ad, a ciklus megszakad, és a művelet sikertelen. Az NVRAM blokk írása az NvM NvM_WriteBlock függvényével történik. Ha az írási művelet eredménye nem NVM_REQ_OK, tehát az adott blokk írása sikertelen, akkor az egész írási művelet sikertelen. Sikertelen írási művelet esetén a jelentett hibakód értéke FI_WRITE_BLOCKS lesz, egyébként nem változik.

4.4 A Serial Transport Protocol megvalósítása

A STP megvalósításánál a követelményeknek megfelelően implementáltam az inicializáló függvényt, az API függvényeket, és az értesítési függvényeket. A megtervezett kézfogásos protokollokhoz szükséges volt még egy üzenetküldő, valamint egy üzenetfeldolgozó állapotgép implementálása. A következő alfejezetekben részletezem a két állapotgép megvalósítását.

4.4.1 A fogadott üzenetek feldolgozása

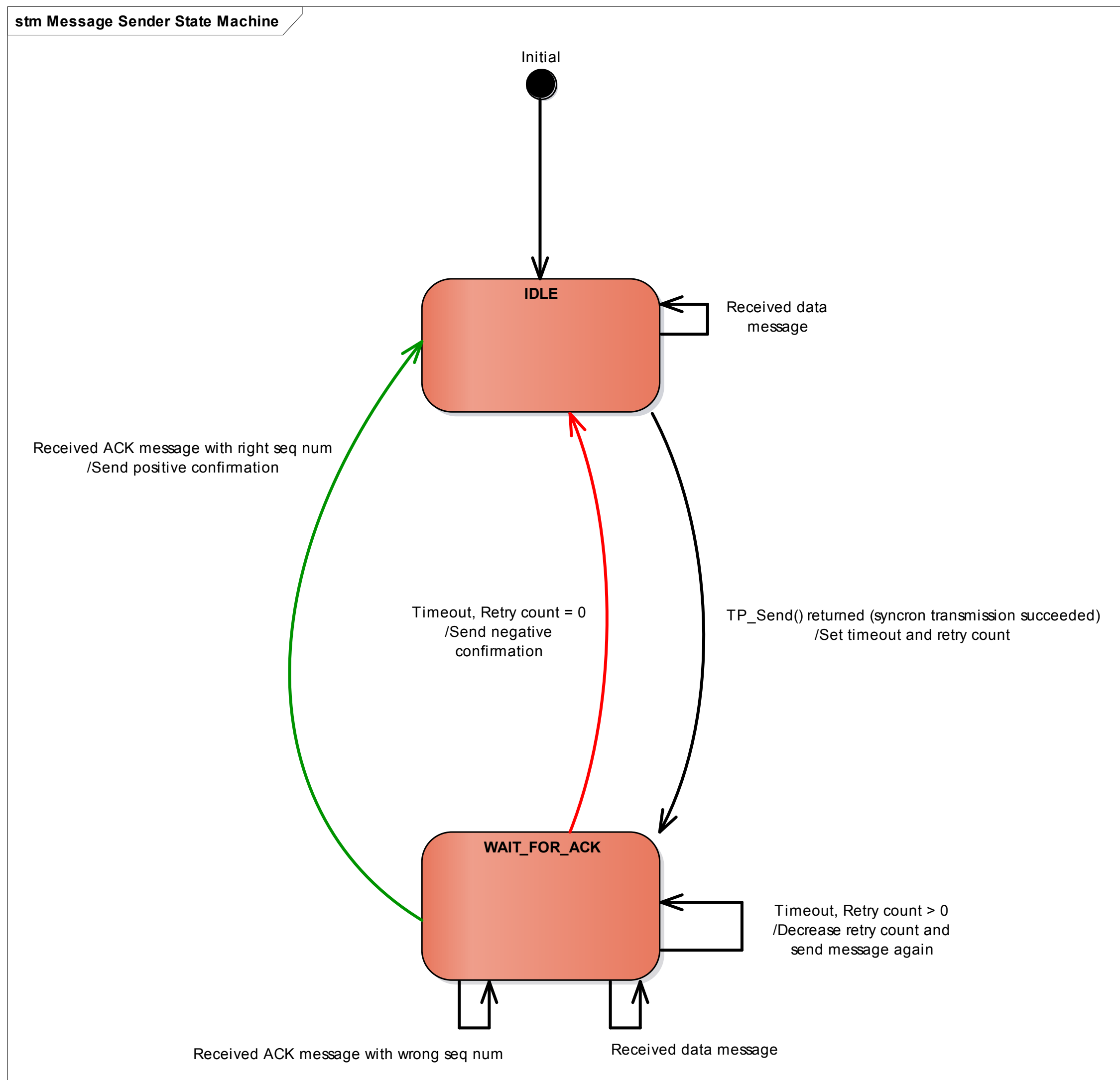
Az üzenetek feldolgozása egy állapotgéppel lett megvalósítva, amire a későbbiekben Parserként fogok hivatkozni. A Parser állapot-átmeneti diagramját a 4.3. ábra szemlélteti. Az állapotgép alapállapota a WAIT_SEQ_NUMBER állapot. Ebben az állapotban várakozik az állapotgép az első érkező bájtra, ami SequenceNumber paraméterként lesz lementve. Miután a fogadott bájt mentésre került, egy időzítő elindításra kerül. Látható, hogy az időzítő minden olyan állapotban indításra kerül, amikor a következő állapot nem az alapállapot. Ennek oka, hogy az időzítő egy byte timer, tehát az egyes bájtok érkezése között eltelt időt méri. Ha ez az idő túllépi a 3.5.2.3-es fejezetben említett ByteTimeout értéket, akkor az állapotgép alapállapotba kerül. Az állapotgép egy bájt fogadása után mindig egy következő állapotba lép, amíg el nem jut WAIT_HEADER_CRC állapotba. A WAIT_MESSAGE_TYPE állapotban mentésre kerül a fogadott üzenet MessageType paramétere, a WAIT_LEN_LOW és WAIT_LEN_HIGH állapotokban pedig a fogadott bájtokból összeállításra kerül a Length paraméter. WAIT_HEADER_CRC állapotban az eddig fogadott paraméterekre számolt 8 bites CRC érték, és a fogadott bájt összehasonlításra kerül. Ha az értékek nem egyeznek, az állapotgép alapállapotba kerül. Ha a CRC-k egyeznek, és a fogadott üzenet típusa nyugtázó üzenet, akkor az állapotgép alapállapotba kerül, és a Parser pozitív visszajelzést ad, hogy sikeresen fogadta az üzenetet. Ha a CRC-k egyeznek, és a fogadott üzenet típusa adat üzenet, akkor az állapotgép WAIT_DATA állapotba kerül. WAIT_DATA állapotban az állapotgép megvárja, míg a Length paraméternek megfelelő számú bájt érkezik, és a bájtokból összeállítja a fogadott üzenet PAYLOAD részét. Az adatbájtok sikeres fogadása után az állapotgép WAIT_DATA_CRC állapotba kerül, ahol összehasonlítja a fogadott CRC értéket az adatra kiszámolt CRC értékkel. Az állapotgép innen mindenképp alapállapotba kerül, de csak a CRC-k egyezése esetén ad a Parser pozitív visszajelzést.



4.3. ábra A Parser állapotgépe

4.4.2 Az üzenetküldés folyamata

Az üzenetek küldése egy állapotgéppel lett megvalósítva, amelyre a későbbiekben MesssageSender-ként fogok hivatkozni. A MesssageSender állapot átmeneti diagramját a 4.4. ábra szemlélteti.



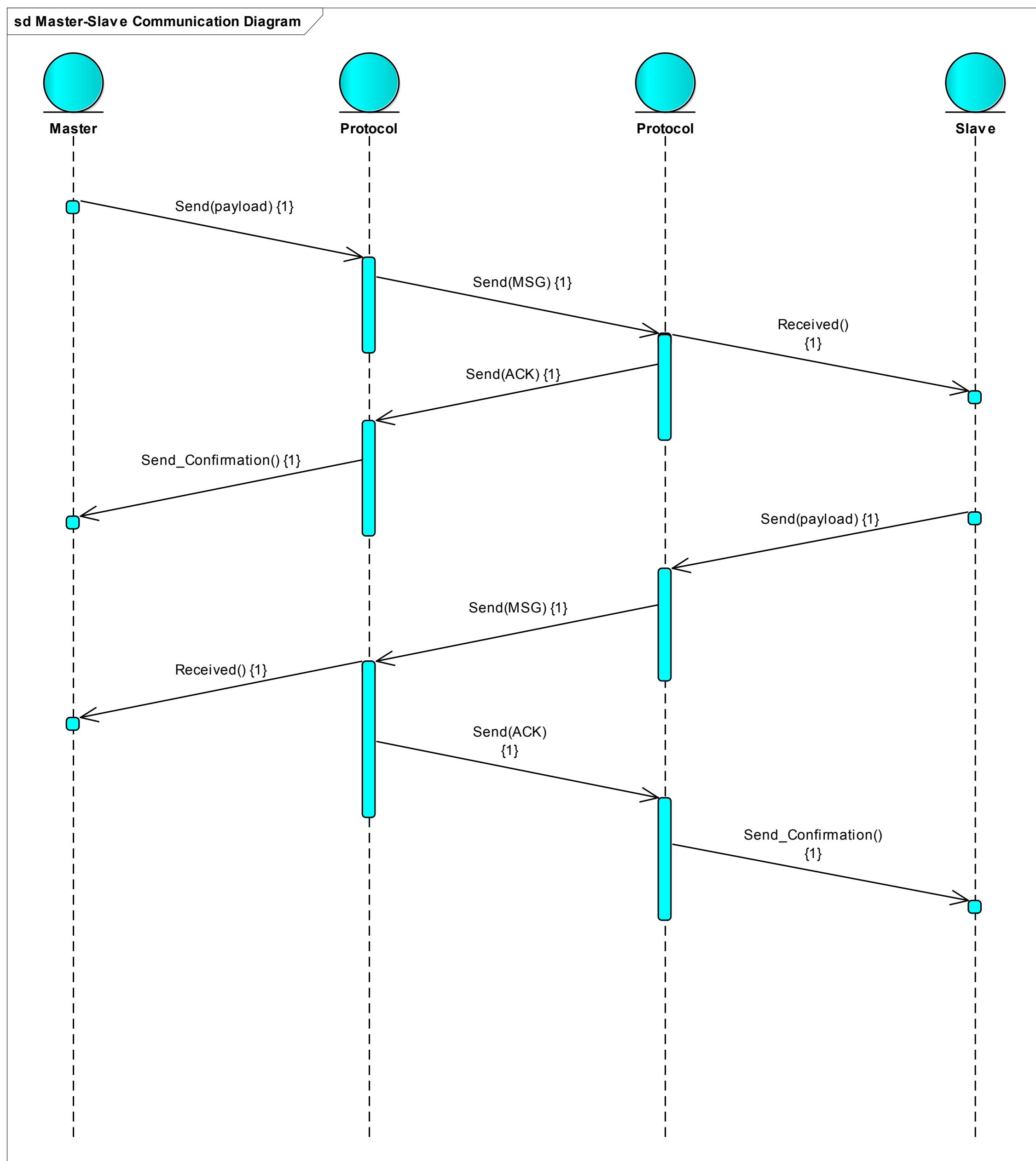
4.4. ábra A MessageSender állapotgépe

A MesssageSender-nek két állapota van, az IDLE, és a WAIT_FOR_ACK. Az állapotgép inicializálás után IDLE állapotba kerül. Az STP_Send függvény meghívása után az állapotgép WAIT_FOR_ACK állapotba kerül, beállítja a konfigurált újraküldési számot, és elindít egy számlálót. WAIT_FOR_ACK állapotban, ha a számláló eléri a 3.5.2.3. fejezetben említett PackageTimeout értéket, akkor timeout következik be. Ha még van újraküldési lehetőség, akkor újra elküldi az üzenetet, egyébként a felsőbb réteg üzenetküldést nyugtázó értesítési függvényének meghívásával jelzi a felsőbb réteg

számára a sikertelen üzenetátvitelt. Ha a timeout előtt adat üzenet érkezik, az üzenetfogadást jelző értesítési függvény meghívásával a modul jelzi a felsőbb réteg számára a sikeres adatüzenet fogadást. Helyes nyugtázó üzenet esetén a modul az üzenetküldést nyugtázó értesítési függvény meghívásával jelzi a felsőbb réteg számára a sikeres üzenetátvitelt. Helytelen nyugtázó üzenet esetén a modul nem végez műveletet, megvárja a timeout leteltét.

4.4.3 Az STP használata a teszt szoftverben

A teszt során a tesztfuttató és a tesztvégrehajtó master-slave módon fog kommunikálni, mivel mindig csak a tesztfuttató küld üzenetet, a tesztvégrehajtó csak az eredményt küldi vissza válaszüzenetként. A követelményekből kiindulva a következő master-slave kommunikációs procedúra játszódik le a teszt során: A master az STP_Send API függvény segítségével üzenetet küld a slave egységnek. Amikor a slave egység Parsere sikeresen feldolgozta az üzenetet, visszaküldi, hogy vette az üzenetet. Miután a slave (tesztvégrehajtó) elvégezte a tesztlépést, visszaküldi az eredményt az STP_Send API függvény segítségével. A master egység Parsere feldolgozza a vett üzenetet, és az előzőekhez hasonló módon nyugtázza ennek sikerességét a slave számára. A lejátszódó folyamatot a 4.5. ábra szemlélteti.



4.5. ábra. Master-slave kommunikáció

4.5 A tesztriport előállítása

A tesztriport előállításánál elsődleges szempont volt, hogy a tesztriport könnyen értelmezhető legyen. A tesztriport tervezésénél ezért egy táblázatos formát alakítottam ki, melyben szerepel az adott tesztlépés száma, a hibainjektálás paraméterei (flashmemóriakép, bájt pozíció, maszk), a hibainjektálás eredménye, valamint, hogy melyik NVRAM blokkon, melyik műveletnél történt hiba. A táblázatos forma megvalósítására a legegyszerűbb az volt, hogy a tesztfuttató létrehoz egy HTML fájlt, amelybe a riportot HTML formázással beleírja. A továbbiakban részletezem a

tesztvégrehajtó válaszüzenetének összeállítását, valamint hogy ebből az üzenetből hogyan keletkezik a tesztriport.

4.5.1 A tesztvégrehajtó válaszüzenete

A tesztvégrehajtó futása során, amikor egy NVRAM blokk olvasása, összehasonlítása, vagy írása sikertelen, akkor az adott NVRAM blokk azonosítója, valamint az aktuális művelet, amelynél a hiba történt, mentésre kerül. A válaszüzenet ezen adatok összessége, valamint a teszt végeredménye.

4.5.2 A tesztriport megvalósítása

A tesztriportot a tesztfuttató hozza létre a válaszüzenet alapján. A hibainjektálás paramétereit, eredményét, valamint a naplózott NVRAM blokk azonosítókat, és hozzájuk tartozó műveletet beleilleszti egy HTML táblázatba. A HTML fájlba ezek a táblázatok vannak sorban beszúrva egy sorszámfejléccel. Mivel az előzetes tesztek, majd azt követő számítások alapján úgy véltem, hogy a keletkező HTML fájl nagy méretű(~100 Mbyte), így megnyitásával problémák lehetnek, ezért találni kellett egy megoldást a fájlméret csökkentésére. A fájlméret úgy lett csökkentve, hogy azon hibainjektálások, amelyek egymást követik és eredményeiknek naplózott NVRAM blokkazonosítói, valamint a hozzájuk tartozó műveleteik egyeznek, össze vannak vonva. Az ilyen összevont bejegyzéseknél a bájt pozíció és a maszk kezdő- és végértékei vannak feltüntetve. A keletkező tesztriportot a 4.6. ábra szemlélteti.

STEP 1...32	
IMAGE ID	2
BYTE POSITION	0...0
BYTE PATTERN	0x8000000000000000...0x0000000100000000
FINAL RESULT	FI_OK
LOG	
	FIRSTREADALL failed on ALL
STEP 33...64	
IMAGE ID	2
BYTE POSITION	0...0
BYTE PATTERN	0x0000000080000000...0x0000000000000001
FINAL RESULT	FI_OK
LOG	

4.6. ábra Tesztriport

4.6 A megvalósítás tesztelése

A teszt lefuttatása előtt szükséges meggyőződni a megvalósított teszt helyességéről. Tesztelés céljából egy- és kétbites hibainjektálásokat végeztem, a 3.2.2-es fejezetben említett első flashmemóriaképen. A létrehozott flashmemóriakép alapján meghatároztam a poolleíró, az egyes FEE blokkleírók, valamint blokktartalmak elhelyezkedését a memóriában. A tesztelés során megvizsgáltam, hogy az egyes blokkok leíróinak és adatainak injektálásakor a megfelelő blokkok lesznek-e hibásnak jelezve. Kétbites hibainjektálásokat is végeztem: egy adott blokkleírón belül, két egymást követő blokkleírón, blokkadaton, valamint két egymást követő blokkadaton. Megvizsgáltam, hogy az injektált blokkok lesznek-e hibásak, valamint hogy 2 blokk elvesztése esetén a tesztlépés sikertelen lesz-e. A tesztelés során a poolleíró, az összes FEE blokkleíró, valamint mindhárom típusú NvM blokkhoz tartozó FEE blokk adata injektálva lett. A tesztelés során keletkező tesztriport egy részletét a 4.7. ábra szemlélteti.

STEP 8	
IMAGE ID	1
BYTE POSITION	184
BYTE PATTERN	0x0000000010000000
FINAL RESULT	FI_OK
LOG	
STEP 9	
IMAGE ID	1
BYTE POSITION	186
BYTE PATTERN	0x1000000000000001
FINAL RESULT	FI_READ_BLOCKS
LOG	
	FIRSTREADALL failed on REDUNDANTBLOCK5
	FIRSTREADALL failed on WPREDUNDANTBLOCK1
STEP 10...12	
IMAGE ID	1
BYTE POSITION	8448...8472
BYTE PATTERN	0x1000000000000000...0x0000000000000001
FINAL RESULT	FI_OK
LOG	
	FIRSTREADALL failed on NVBLOCK3
	SECONDREADALL failed on NVBLOCK3

4.7. ábra A megvalósítás tesztelésének tesztriport részlete

Az ábrán látható, hogy a 8. lépésben üres flashmemória-területet injektálva a tesztlépés eredménye FI_OK, a tesztriport LOG mezője pedig üres, mivel minden blokk olvasása sikeres volt. A 9. lépésben kétbites hibainjektálással két szomszédos FEE blokkleírót injektálva a tesztlépés eredménye FI_READ_BLOCKS, ami helyes, mivel 2 FEE blokk olvasásának sikertelennek kell lennie. A 10-12. lépésekben az egybites hibainjektálást egy blokk határain végeztem, így vizsgálva a bitinverzió helyének helyességét.

Összefoglalás

Az AUTOSAR memóriastack egy komplex, több modulból álló szoftver, amely egy hibatűrő naplózó fájlrendszert alakít ki a flash memóriában. Az említett szoftvert az ISO26262 szabvány kockázati szempontból ASIL (Automotive Safety Integrity Level) D kategóriába sorolja, amely megköveteli a termék legmagasabb szintű integritási megfelelőségét. A magas biztonsági követelmények miatt volt szükséges a memóriastack rendelkezésre állásának vizsgálata.

A szakdolgozatom készítése során egy AUTOSAR memóriastack tesztelésére alkalmas hibainjektor szoftvert implementáltam. Az általam megvalósított teszt egy szoftverteszt, amellyel memóriastack robosztusságának kísérleti vizsgálata lehetséges. A megoldás során először flashmemóriaképeket hoztam létre, amelyek a memória élettartama során előforduló memóriaállapotokat modellezik, a tesztet ezeken a memóriaképeken szükséges lefuttatni. A teszt futása során a tesztszoftver egybites hibát injektál a flash memóriába. A teszt nem valós meghibásodást idéz elő, tehát a flash memória beépített hibadetektáló képessége által a hiba nem detektálható, így a memóriastack legalsó modulja, a Flash Driver, nem fog hibát jelezni. Az injektált hibát csak a Flash Driver feletti modulok érzékelik, amelyek a naplózó fájlrendszer hibatűrő működéséért felelnek.

A hibainjektálás után a tesztszoftver különböző memóriaműveleteket végez, így vizsgálva, hogy egy egybites hiba esetén a naplózó fájlrendszer nem veszít el olyan adatot, amelybe nem lett hiba injektálva, valamint, hogy a memóriát kezelő szoftver egyszer sem került végtelen ciklusba. A tesztelés során egy tesztriport készül, amely tartalmazza a hibainjektálás pontos helyét, valamint a hibainjektálás utáni műveletek eredményét és az elvesztett blokkok azonosítóit.

Továbbfejlesztési lehetőség a teszt hosszú futási idejének csökkentése. A teszt során minden egyes tesztlépésben szükséges a célprocesszor resetelése, és az utána való várokozás. Ez a várakozási idő csökkenthető lenne a bootloader továbbfejlesztésével, mivel így a célprocesszoron futó program gyorsabban elindulna. Másik lehetőség esetleg az egyes flashmemóriaképek tesztelésének külön processzoron való futtatása. Ez nem lenne pazarló, mivel a processzort egy flashmemóriakép tesztelése után amúgy is cserélni kell. Ezzel a párhuzamos futtatással a teszt futási ideje ötödére csökkenthető.

Ábrák jegyzéke

1.1. ábra. Az AUTOSAR rétegzett szoftverarchitektúrája 9

1.2. ábra. Alap Szoftver rétegek 10

1.3. ábra A memóriastack felépítése 12

1.4. ábra. NV blokk szerkezete 13

1.5. ábra. Redundáns NVRAM blokk felépítése 14

1.6. ábra. A Dataset blokk felépítése 15

1.7. ábra. FEE naplózó fájlrendszere 18

2.1. ábra A tesztkörnyezet felépítése 23

3.1. ábra. Üres flashmemóriakép 27

3.2. ábra Második flashmemóriakép 27

3.3. ábra. Harmadik flashmemóriakép 28

3.4. ábra. Negyedik flashmemóriakép 28

3.5. ábra. Ötödik flashmemóriakép 29

3.6. ábra. Hibainjektor állapot-átmeneti diagram 34

3.7. ábra. Csomagfelépítés 37

3.8. ábra. Az STP UML szekvenciadiagramja 39

4.1. ábra. A hibainjektor folyamatábrája 44

4.2. ábra. Az olvasás és összehasonlítás művelet folyamatábrája 46

4.3. ábra A Parser állapotgépe 49

4.4. ábra A MessageSender állapotgépe 50

4.5. ábra. Master-slave kommunikáció 52

4.6. ábra Tesztriport 53

4.7. ábra A megvalósítás tesztelésének tesztriport részlete 54

Irodalomjegyzék

[1] Yu Cai, Erich F. Haratsch, Onur Mutlu, Ken Mai: *Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis*, 2012, https://pdfs.semanticscholar.org/5a04/b332441e2ff025313bfd303383e13050a274.pdf

[2] AUTOSAR Consortium: *Layered Software Architecture R4.0 Rev 3*, szabványdokumentum, 2012, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_EXP_LayeredSoftwareArchitecture.pdf

[3] Dr. Pintér Gergely: *Szoftverkomponensek*, előadásjegyzet, 2017, https://www.mit.bme.hu/system/files/oktatas/targyak/vedett/10727/AUTOSAR-Based_Automotive_Systems_-_Software_Components_-_annotated.pdf

[4] Faragó Dániel: *Basic Software – Kommunikáció*, előadásjegyzet, 2017 https://www.mit.bme.hu/system/files/oktatas/targyak/vedett/10727/02_bsw_communication_0.pdf

[5] Faragó Dániel: *AUTOSAR FlexRay kommunikációsmodulok megvalósítása*, szakdolgozat, BME-VIK, 2012

[6] AUTOSAR Consortium: *Specification of Flash Driver R4.0 Rev 3*, szabványdokumentum, 2011, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_FlashDriver.pdf

[7] AUTOSAR Consortium: *Specification of Flash EEPROM Emulation R4.0 Rev 3*, szabványdokumentum, 2011, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_FlashEEPROMEmulation.pdf

[8] AUTOSAR Consortium: *Requirements on Memory Hardware Abstraction Layer R4.0 Rev 1*, szabványdokumentum, 2009, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SRS_MemoryHWAbstractionLayer.pdf

[9] AUTOSAR Consortium: *Specification of NVRAM Manager R4.0 Rev 3*, szabványdokumentum, 2011, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_NVRAMManager.pdf

[10] AUTOSAR Consortium: *Specification of EEPROM Abstraction R4.0 Rev 3*, szabványdokumentum, 2011, https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_EEPROMAbstraction.pdf

[11] AUTOSAR Consortium: *Specification of EEPROM Driver R4.0 Rev 3*, szabványdokumentum, 2011,

https://www.autosar.org/fileadmin/user_upload/standards/classic/4-0/AUTOSAR_SWS_EEPROMDriver.pdf

[12] Majzik István, Micskei Zoltán: *A hibakezelés tesztelése: Hibainjektálás*, előadásjegyzet, https://inf.mit.bme.hu/sites/default/files/materials/category/kateg%C3%B3ria/oktat%C3%A1s/msc-t%C3%A1rgyak/szoftverellen%C5%91rz%C3%A9si-technik%C3%A1k/12/SZET-2012_EA12b_hibainjektalas.pdf

[13] AUTOSAR Consortium: *Overview of Acceptance Tests R1.0*, szabványdokumentum, https://www.autosar.org/fileadmin/user_upload/standards/tests/1-0/AUTOSAR_EXP_AcceptanceTestsOverview.pdf

[14] AUTOSAR Consortium: *Acceptance Test Specification of Memory Stack R1.0*, szabványdokumentum, https://www.autosar.org/fileadmin/user_upload/standards/tests/1-0/AUTOSAR_ATS_MemoryStack.pdf