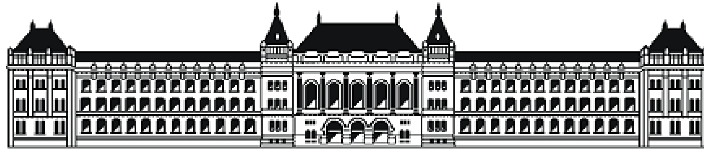


Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

Firmware-frissítő alkalmazás fejlesztése különböző típusú processzorokhoz

DIPLOMATERV

Készítette

Dajka Attila Norbert

Konzulensek

Molnár Károly, Orosz György

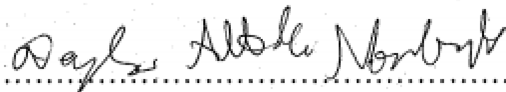
2013. május 17.

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2013.05.17.



Dajka Attila Norbert

Tartalomjegyzék

Kivonat 7
Abstract 8
1. Bevezetés 9
2. Specifikáció és teljes rendszerterv 11
3. Hoszt oldali szoftverkomponensek 15
3.1 Qt-ban rejlő lehetőségek 16
3.1.1. GUI fejlesztés 17
3.1.2. Szálkezelés 22
3.1.3. Signal és slot mechanizmus 26
3.1.4. Szkript használat 29
3.2 A szoftver rendszerterv részletei 37
3.2.1. Alkalmazói réteg 37
3.2.2. Kapcsolati réteg 50
3.2.3. GUI réteg 58
4. Eszköz oldali szoftverkomponensek 63
4.1 Futtatást végző hardver eszközök 65
4.1.1 Blackfin 537 DSP 66
4.1.2 ATmega 128 mikrokontroller 67
4.2 Nem portolandó részek 69
4.2.1 Protokollértelmező 69
4.2.2 Parancsértelmező 70
4.2.3 Verzióinformáció és konzisztenciaellenőrző 71
4.3 Portolt részek 73
4.3.1 Hardver interfész 73
4.3.2 Parancsvégrehajtó 78
5. Eredmények bemutatása 83
6. Összefoglalás 89
7. Irodalomjegyzék 90

Kivonat

A diplomatervben a beágyazott eszközök alkalmazói programjának egyik frissítési lehetőségét mutatom be. Kifejtem, hogy miért hasznos a frissítéssel foglalkozni, milyen gyakorlati előnyökkel jár a frissítés, ezután bemutatom a frissítést végző szoftverek megkívánt tulajdonságait. Ennek alapján kiválasztok a frissítő fejlesztéséhez egy olyan fejlesztőkörnyezetet, amely jelentősen támogatja a frissítő tulajdonságainak implementálását, majd ismertetem a fejlesztőkörnyezetet, és a fejlesztést segítő legfontosabb részeit kiemelem. Ezt követően a frissítő PC oldali rendszertervét részletezem és az ott szereplő egyes modulok funkcióit tisztázom. Ennek végeztével a frissítő beágyazott oldali rendszertervének tárgyalásába kezdek. Ebben elkülönítem a különböző eszközökre azonosan megírt modulokat a módosítást igénylő moduloktól. Bemutatom, hogy a modulok milyen feladatokat látnak el, és ezt milyen működéssel érik el. Ehhez kapcsolódóan kitérek a frissítés tesztelésére szolgáló hardver eszközök: egy ATmega128-as mikrokontroller és egy Blackfin 537-es DSP ismertetésére. Ezután teszteredményeket mutatok. Végül összegzem az elvégzett munkát, és továbbfejlesztési lehetőségeket is felvázolok.

Abstract

In this thesis, I present a possibility for the firmware upgrade of embedded systems. I explain the motivation for firmware upgrade, what practical benefits does it come with, then I introduce the requirements for the firmware loader software. Based on that I have chosen a software development kit that supports the most of the firmware loader's features, review its properties, and I emphasize its features, which were most helpful during the development. Then I go into details about the PC side software plan of the firmware loader, I unravel its modules, and the function of its modules. After that I start to explain the embedded side software plan of the firmware loader. I differentiate between the modules according to their portability. I show the functions of these modules, and the underlying operations. I present the hardware used for the testing of the firmware loader: an ATmega 128 microcontroller and a Blackfin 537 DSP. Then I show test results. Last, but not least, I summarize the my work, and suggest a few directions of improvement.

1. Bevezetés

Firmware-nek nevezzük beágyazott rendszerek alkalmazói programját. Ez az eszköz működéséhez szükséges legalapvetőbb feladatokat látja el. Beágyazott rendszerek a fizikai-technológiai környezetükkel intenzív információs kapcsolatban álló, bizonyos speciális feladat ellátására tervezett számítógépes eszközök. Ezek száma napjainkban exponencálisan nő, becslések szerint 2020-ra 40 milliárd beágyazott eszköz fog üzemelni, jelenleg pedig a legyártott mikroprocesszorok 98%-a beágyazott rendszerbe kerül [1]. Beágyazott rendszer vezérlőegysége lehet például mikrokontroller, DSP vagy FPGA. Jelen diplomaterv eltekint a változtatható hardver megvalósítást lehetővé tevő FPGA-k firmware-frissítésének leírásától. Helyette a konvergens fejlődési utat bejáró jelfeldolgozó processzorok és mikrokontrollerek szemszögéből vizsgálja a témát.

Az eszközök technológiájának és komplexitásának növekedési ütemét késve követi a szoftver eszközök és verifikációs lehetőségek fejlődése [2]. Ráadásul az esetek többségében a beágyazott rendszer idő hiányában és a piacra kerülési idő lerövidülése miatt nem tesztelhető teljes körűen, emiatt üzembe helyezése után is szükség van a rajta futó szoftver javítására, újraprogramozásra.

Az intenzív információs kapcsolat miatt az eszközt körülményes előkészíteni a firmware-frissítésre. A szükséges előkészületek közé tartozhat a vezérelt vagy vizsgált rendszer leállítása, új firmware eljuttatása az eszközhöz, esetleg annak programozó üzemmódba kapcsolása.

A fenti feladatok automatizálása céljából egységes megközelítést alkalmaztam a firmware-frissítéshez. Az információs kapcsolat megszakítását feleslegessé teszi, ha az eszköz üzem közben használt protokolljával történik a frissítés. Ez egyúttal a firmware eljuttatását is lehetővé teszi az üzem közben használt buszon keresztül. Megfelelő parancsokat definiálva a protokollban, a programozó üzemmódba kapcsolás is automatizálható. Bár ekkor a mérést vagy vezérlést meg kell szakítani a frissítés idejére,

de a frissítő funkciót egy bootloaderbe programozva ez akár történhet a beágyazott rendszer indítása után közvetlenül is. Ilyen működés megvalósítása nemcsak egy alkalmas bootloader meglétét feltételezi a beágyazott oldali részen, hanem az új firmware-t küldő eszköz, gazdagép (hoszt) részéről is feltételez az új firmware letöltését lehetővé tevő szoftverkomponenseket. Ez lesz a frissítő hoszt oldali része. Diplomamunkám során hosztként PC-t használtam.

Dolgozatom felépítése a következő:

Az ezután következő, második fejezetben bemutatom a kifejlesztett rendszerrel szemben támasztott követelményeket, és az ebből származó teljes rendszertervet, a főbb szoftverkomponensek kapcsolatát.

A harmadik fejezetben a hoszt oldali firmware-frissítő létrehozásához használt fejlesztőkörnyezetet ismertetem a hoszt oldali szoftvermodulok részleteivel együtt. Kifejtem, hogy a fejlesztőkörnyezet mely tulajdonságai a modulok fejlesztését hogyan befolyásolta.

A negyedik fejezetben a frissítő eszköz oldali szoftverkomponensei mellett az alkalmazott hardver eszközök jellemőit is részletezem. Ezek az eszközök a frissítés tesztelésére szolgáltak.

Az ötödik fejezetben bemutatom az elért eredményeket, a frissítés lépéseit.

A hatodik fejezetben összefoglalom az elvégzett munkát, és a továbbfejlesztési lehetőségeket is felvázolom.

2. Specifikáció és teljes rendszerterv

A specifikáció elkülönül hoszt oldali részre (ez maga a firmware-frissítő) és beágyazott vagy eszköz oldali részre (ez egy bootloader). A hoszt oldali rész PC-n fut, a beágyazott oldali rész mikrokontrolleren vagy DSP-n. A két részt összekötő fizikai réteg az RS-232.

A specifikáció szerint az eszközön történő resetelés után a bootloader vár a hoszt program parancsaira. Új firmware-t ír az eszköz memóriájának bizonyos címére, majd ezt a firmware-t el is indítja, ha a hoszt program olyan paranccsal látja el. A bootloader védve van a felülírástól, helytelen hoszt parancs nem képes módosítani.

Az 1. ábra a beágyazott eszközön futó bootloader rendszertervét szemlélteti. Ez a specifikáció most következő, eszköz oldali leírása alapján készült, mely a szükséges modulokat taglalja:

Parancsvégrehajtó: Meghatározott bootloader funkciókhoz kapcsolódó parancsokat hajt végre, melyek az alábbiak:

- törlés (cím, hossz szerinti memóriatartományon végezve)
- írás (cím, hossz szerinti memóriatartományon végezve, meghatározott adatokat)
- resetelés
- olvasás(cím, hossz szerinti memóriatartományon végezve)
- memória művelet befejeztének lekérdezése

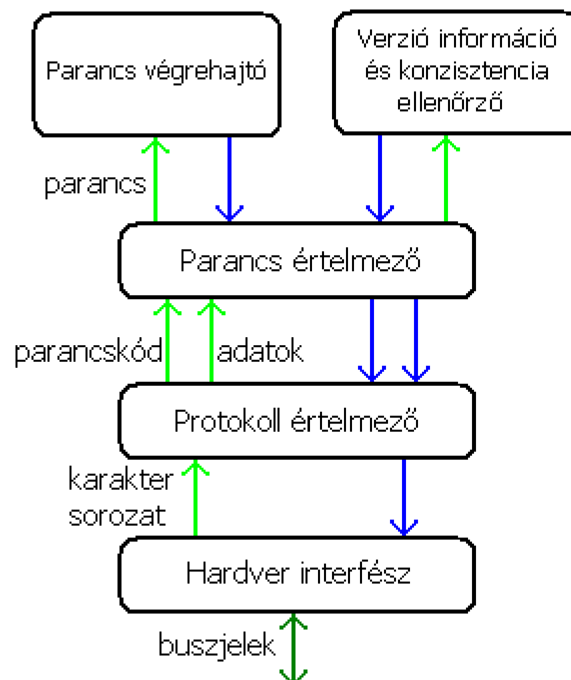
Parancsértelmező: Parancsokká alakítja a megkapott üzeneteket, és ezt a műveletet fordított irányban (parancs → üzenet) is elvégzi.

Protokollértelmező: A bejövő üzenetből kinyeri a hasznos információt, és a kimenő üzenet megfelelő felépítéséről gondoskodik. Pszeudokódja:

```
Protokollértelmező ( ) {
      // várakozás bejövő üzenetekre
      // bejövő üzenet részekre bontása
      // üzenet hibaellenőrzése
      // parancsértelmező meghívása
      // kimenő üzenet összeállítása, elküldése
}
```

Hardverinterfész: A kommunikáció alacsony szintű interfészét realizálja. Függvényei:

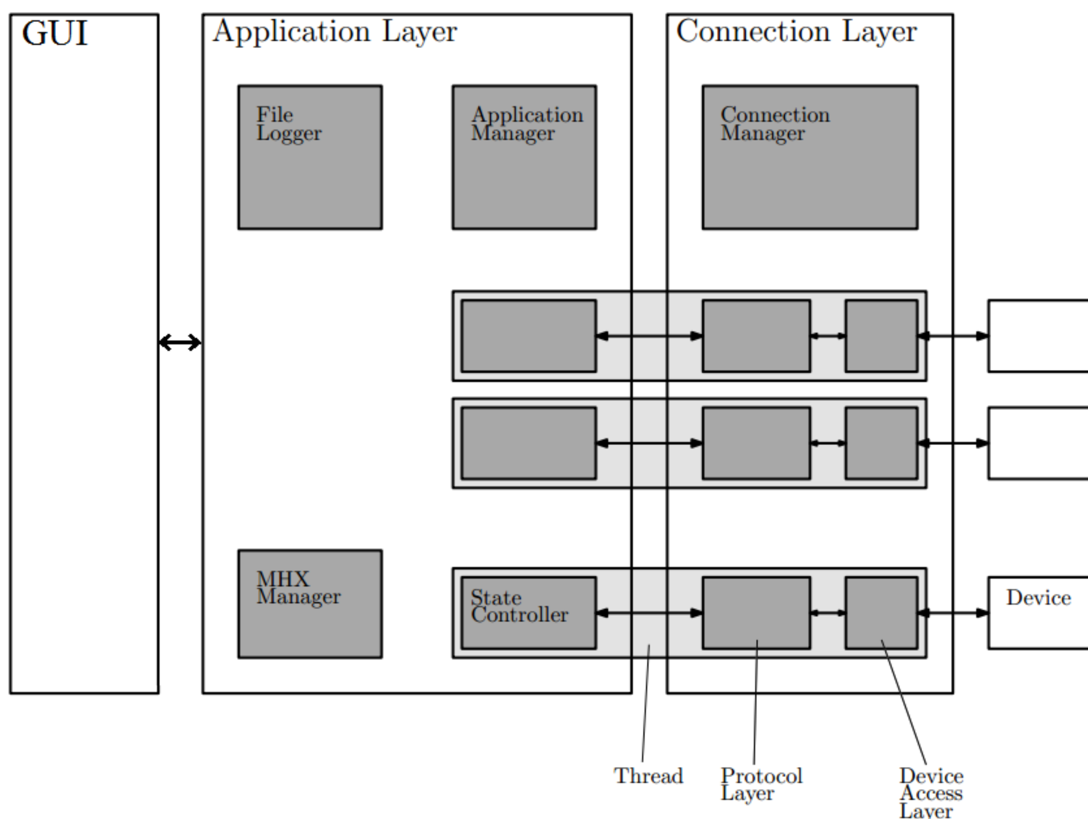
```
kommunikáció inicializálás(...)
bájt küldés(...)
bájt fogadás(...)
küldő fifo állapotlekérdezés(...)
fogadó fifo állapotlekérdezés(...)
üzenet keret hiba(...)
keret hiba törlés(...)
```



1. ábra: Eszköz oldali rész rendszerterve

Az eszköz oldali rész rendszertervéből jól láthatóan elkülöníthetőek processzorfüggetlen részek, melyek pusztán C-ben íródtak. Ide tartozik a protokollértelmező és a parancsértelmező. Ugyanakkor vannak processzorspecifikus részek is amik megszakításkezelést, memóriakezelést végeznek. Ide a hardver interfész, a parancsvégrehajtó, és a verzióinformációt kezelő modul tartozik. Konkrétan a hardver interfész felelős a kommunikációval kapcsolatos megszakításokért. A memóriakezelésért pedig a parancsvégrehajtó. Az utóbbiakat más processzorra történő portolásnál át kell írni. A verzióinformációt kezelő modulnál mindössze a header fájlban definiált memóriacímeket kell módosítani, annak a portolása a legkönnyebb. A processzorfüggetlen részeket pedig értelemszerűen egyáltalán nem kell módosítani portolásnál.

A 2. ábra a PC vagyis hoszt oldali rendszertervet mutatja, ami a hoszt oldali specifikáció része. Ez 3 fő rétegből áll, és ezen keresztül avatkozhat be a kezelő a firmware-frissítés folyamatába.



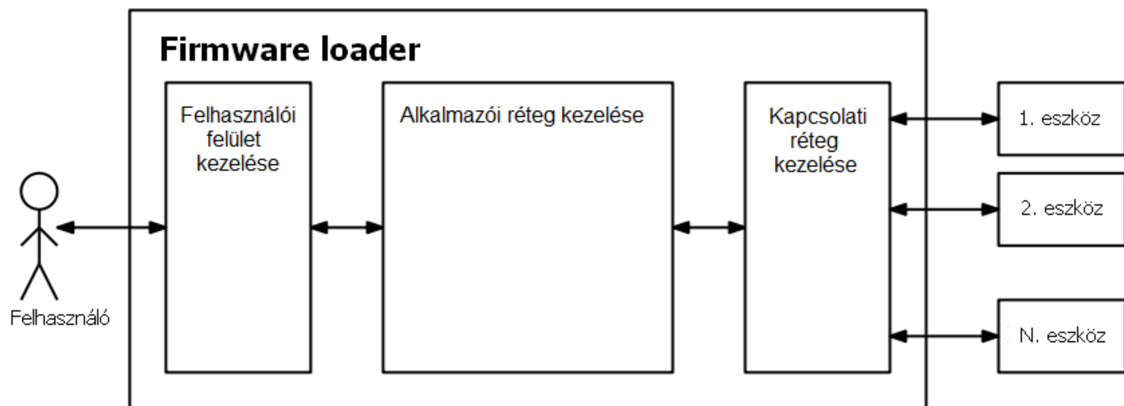
2. ábra: Hoszt oldali firmware-frissítő

A hoszt program (vagy firmware-frissítő) felelős az alkalmazói programok beágyazott rendszerre (eszközre) való letöltéséért. A program implementálására vonatkozó főbb megfontolások az alábbiak:

- Modularitás: a hoszt program újrahasznosítható, viszonylag független komponensekből áll, melyek Qt/C++ osztályok formájában vannak megvalósítva. Ezen osztályokat a lehető legkevesebb interfésszel érdemes ellátni (csak néhány tagfüggvény vagy Qt signal/slot jól meghatározott feladatok végrehajtására), melyek maximálisan függetlenek a mindenkori implementációtól.
- Konfigurálhatóság: a hoszt programot olyan módon kell felépíteni, hogy annak működése állítható, újrakonfigurálható legyen viszonylag könnyen, anélkül, hogy a programot újra kellene fordítani. A header fájlokon kívül deklarált, a programszerkezetbe épített numerikus megkötéseket, és bonyolult #ifdef struktúrákat kerülni kell. A hoszt programnak a lehető leginkább kompatibilisnek kell lennie jövőben alkalmazott eszközökkel is, azaz a működésének könnyen kibővíthetőnek kell lennie szkript fájlok módosításával. Ez különösen a hoszt kapcsolatainak vezérlésénél kritikus.
- Többszálúság: a hoszt programnak képesnek kell lennie egyszerre több csatlakoztatott eszköz kezelésére úgy, hogy minden egyes kapcsolatnak egy külön szálat (thread) hoz létre.
- Esemény és hiba loggolásra a hoszt programnak képesnek kell lennie log fájlok írására, melyek részletes információt tartalmaznak a felmerült hibákról és különböző eseményekről.

3. Hoszt oldali szoftverkomponensek

A program magasszintű felépítése a 3. ábrán látható. Az ott szereplő rétegek feladatai a következők:



3. ábra: Firmware-frissítő magasszintű felépítése

- Kapcsolati réteg: ez a réteg kezeli a kapcsolatokat és az adatforgalmat a hoszt program és az eszközök között. Egységes felületet biztosít az eszközökhöz való hozzáféréshez (kapcsolódás, olvasás, írás), mely nagyban független a kapcsolódás módjától. Az alacsony szintű részleteket elfedi (például az alkalmazott protokoll vagy fizikai réteg fajtáját, ellenőrzőösszeg számítást) a többi rétegtől.
- Alkalmazói réteg: ezen réteg feladatai közé tartozik:
  - a programozó fájlok (*.hex) kezelése,
  - a hoszt program és az eszköz közti handshake folyamat kezelése,
  - esemény- és hibaloggolás,
  - leíró és inicializálófájlok kezelése.

A programozó interfészét a felhasználói felülettel kapcsolatos felesleges feltételezések mellőzésével kell implementálni úgy, hogy az utóbbit könnyen le lehessen cserélni (tehát például szöveges terminálra egy grafikus felhasználói felület helyett).

- Felhasználói felület: grafikus felhasználói felület, mely felhasználói parancsok továbbküldéséért felelős az alsóbb rétegek felé, továbbá a program állapotát is kijelzi a felhasználónak

3.1 Qt-ban rejlő lehetőségek

Az előzőekben ismertetett rendszertervből kitűnik, hogy cél egy széles felhasználási körrel bíró hoszt oldali firmware loader megalkotása, ugyanis erőforrásigényes minden alkalmazásra, minden beágyazott eszközre külön hoszt programot írni. Az általános alkalmazhatóság követelménye maga után vonja, hogy:

- lehetőleg legyen platformfüggetlen a program,
- legyen újrakonfigurálható, lehetőleg anélkül, hogy módosítani kellene a forráskódját és újra le kellene fordítani,
- támogassa a szálkezelést és különféle hálózati kommunikációt.

Ezeknek a kívánalmak a kielégítéséhez az ún. Qt alkalmazás-keretrendszert választottam. Eredetileg a norvég Trolltech cég hozta létre a Qt-ot. Később ezt bekebelezte a Nokia, és folytatta a projektet [3].

A Qt jelenleg a 4.8-as verziójánál tart, komolyan dokumentált minden osztálya, függvénye, és funkciója. Példakódok és minták (design pattern) is rendelkezésre állnak a használatához, így egy megbízható fejlesztőkörnyezetnek tűnik.

Egyetlen hátránya van: közvetlenül a Qt-ba nincs beépítve a soros port kezelés. Viszont több Qt-on kívüli projekt is kínál függvénykönyvtárat ennek a hiányosságnak az áthidalására. Ilyen például a qextserialport függvénykönyvtár. A Qt-ban szereplő

IODevice osztályból származtatott *QextSeriaPort* osztály már az 1.2-es verziónál tart, és az elérhető információk alapján megbízhatónak bizonyult.

A Qt használata mellett szól, hogy GNU licensszel rendelkezik, ingyenes és nyílt forráskódú. Működik Windows, Linux és az apple OS X operációs rendszer alatt is. Standard C++-ra épül, de bele van építve egy ún. signal-slot mechanizmus. Ezzel robusztusabb kapcsolat létesíthető különböző objektumok között. Erősen támogatja a GUI fejlesztést, akár még grafikus felhasználói felületet is igénybe lehet venni ilyen jellegű feladatokra. Támogatja több szálon futó programok írását, ezáltal adott erőforrások jobb kihasználását segíti. A konfigurálhatóságot pedig a Qt Scriptből való programfuttatás teszi lehetővé. Most pedig részletesen fogom bemutatni ezeket a funkciókat.

3.1.1. GUI fejlesztés

A Qt egyik specialitása a grafikus felhasználói felület (Graphical User Interface, a továbbiakban GUI) programozásának egyszerűsége. Számos osztály van definiálva benne, amelyek egy-egy GUI elemet (widget) reprezentálnak. Ezeket deklarálva, majd a közöttük és a főprogram között lévő kapcsolatokat definiálva lehet felépíteni egy felhasználói felületet. Ezt a témát a későbbi könnyebb tárgyalás miatt a GUI elemek megjelenítésével folytatom. A GUI támogatás kidolgozottságának érzékeltetésére felsorolok néhányat a teljesség igénye nélkül a legalapvetőbb widgetek közül [4]:

1) QCheckBox jelölőnégyzet szöveges címmel
2) QComboBox legördíthető lista nyomógombbal
3) QDial kör alakú értékbeállító
4) QLabel szöveg ill. kép megjelenítés
5) QLineEdit egysoros szövegszerkesztő

6) QMenu menü elem menüsorokhoz, felugró menükhöz

7) QProgressBar függőleges vagy vízszintes állapotjelző

8) QPushButton nyomógomb

9) QScrollBar gördítősáv

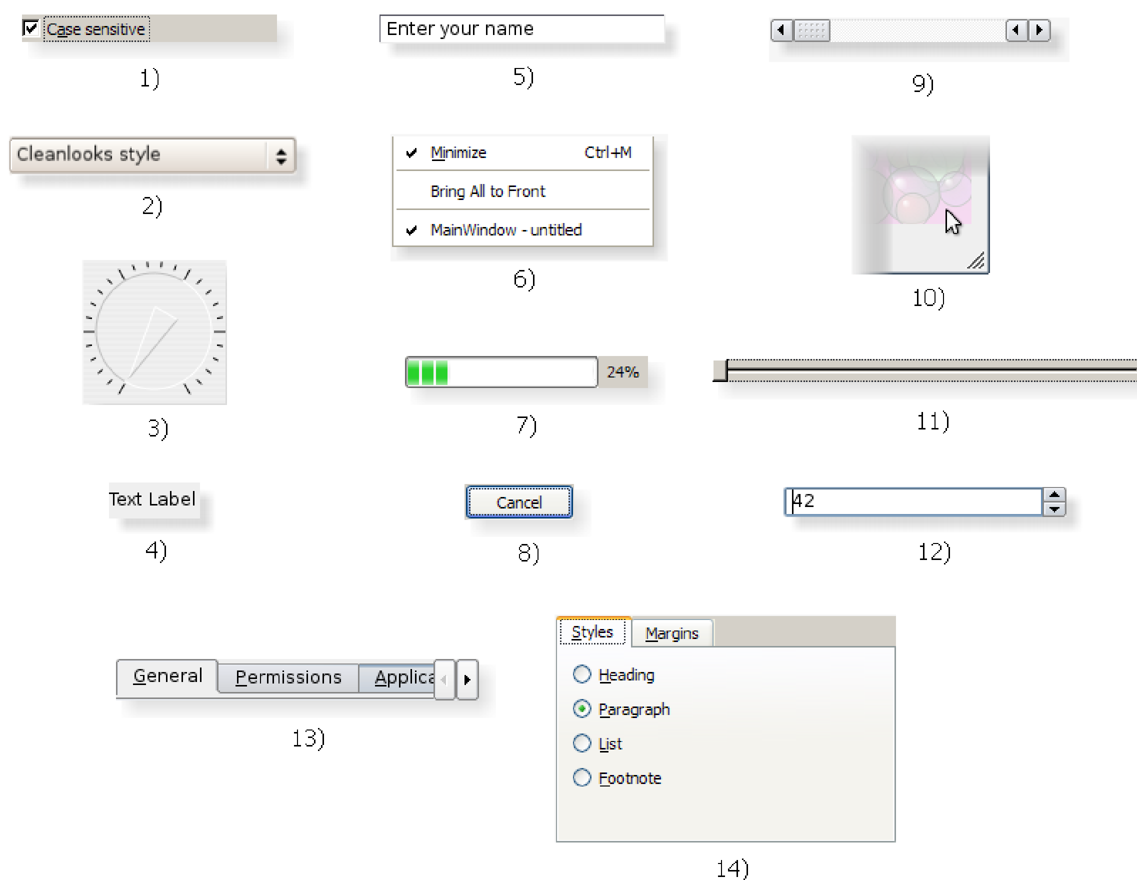
10) QSizeGrip ablakok átméretezéséhez használható

11) QSlider csúszka

12) QSpinBox értékválasztó

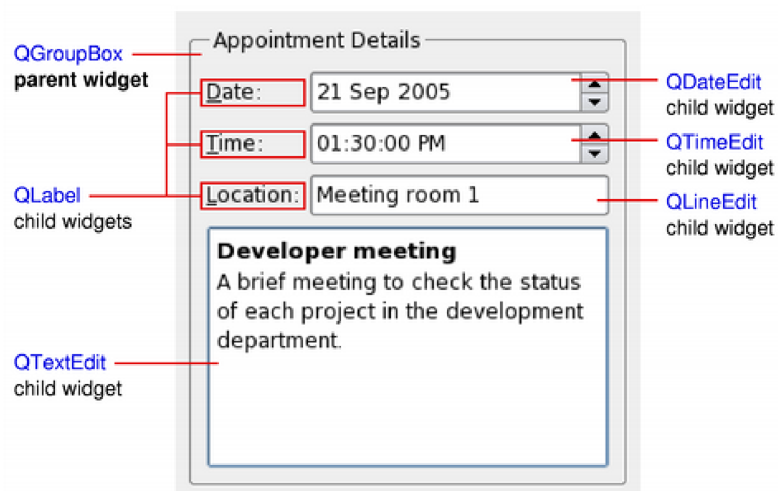
13) QTabBar kiválasztható fül

14) QTabWidget kiválasztható fülön lévő elemek



4. ábra: Alapvető widgetek

A widgetek a felhasználói felületek létrehozásának elsődleges eszközei Qt-ban. Meg tudnak jeleníteni adatokat és státuszinformációkat, tudnak felhasználótól jövő adatokat kezelni, interaktívan használhatók a program működésének megváltoztatásához, plusz funkcionalitás implementáláshoz.

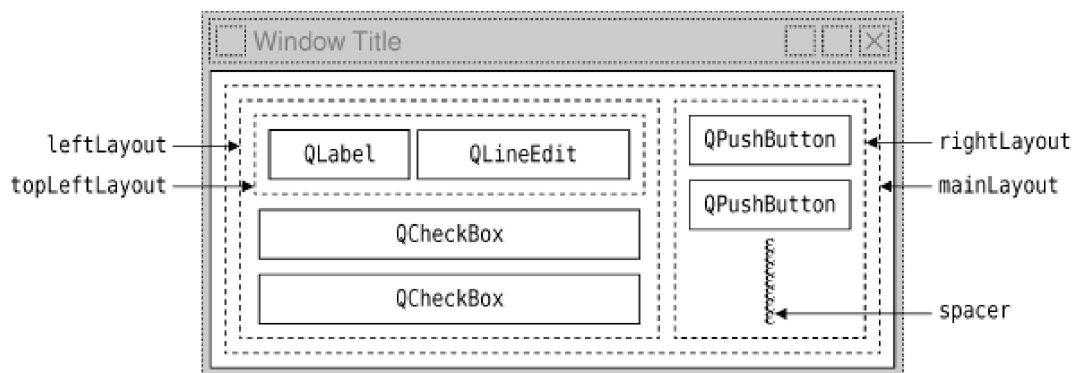


5. ábra: Widgetek azonosítása GUI-ban

Egyúttal más widgetek elhelyezésére alkalmas tárolóként funkcionálnak, feltéve, hogy azok egy csoportba tartoznak, és egymás mellé kell elhelyezni őket. Az a widget, mely nem egy másik widgetben található, az maga az ablak.

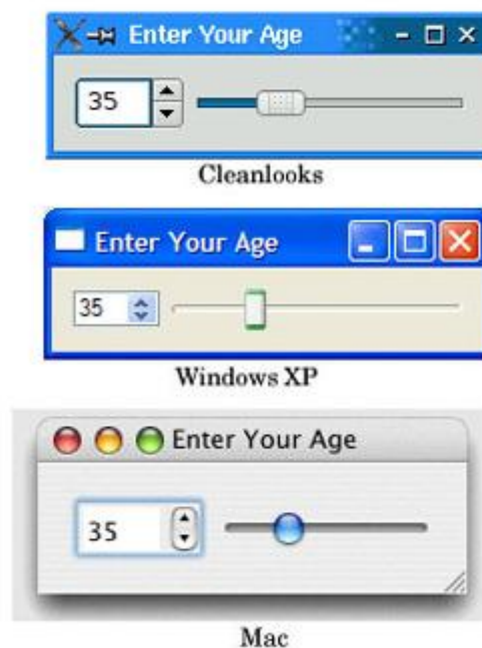
Adott GUI elem vagy megoldás megtalálását segíti a következő osztálystruktúra: A legfontosabb megjelenítéssel kapcsolatos osztály a *QWidget*, mely felhasználótól jövő információk kezelésére is alkalmas. A Qt-hoz tartozó minden felhasználói felület (UI) elem vagy a *QWidget*ből származtatott osztály, vagy kapcsolatban áll egy ilyen származtatott osztállyal. Tetszőleges widget megvalósítása is lehetséges, csak arra kell figyelni, hogy annak szülőosztálya a *QWidget* vagy egy abból származtatott osztály legyen. Ezután már csak a virtuális eseménykezelést kell újra implementálni.

A Qt egyszerűen és flexibilisen közelíti meg az UI létrehozást. A leginkább elterjedt módszer a szükséges widgetek példányosítása, majd tulajdonságaik módosítása. Ezután a widgetek elhelyezését (layout) érdemes beprogramozni, amely ezután automatikusan beálltja a méretet és pozíciót [5]. Végül az UI működését a Qt signal és slot mechanizmusával könnyen meg lehet határozni.



6. ábra: GUI felépítés layoutok segítségével

A layout kezelők megszabadítanak az ablakbeli pozíciók és méretek fix programozásától. Gondoskodnak arról, hogy az ablak átméretezésekor az egyes widgetek egymáshoz és az ablakhoz viszonyítva is esztétikusan helyezkedjenek el. Ezt úgy érik el, hogy mindegyik widget tudatja a layouttal a helyigényét bizonyos adattagokon keresztül, és a layout a rendelkezésre álló helyet arányosan osztja el.

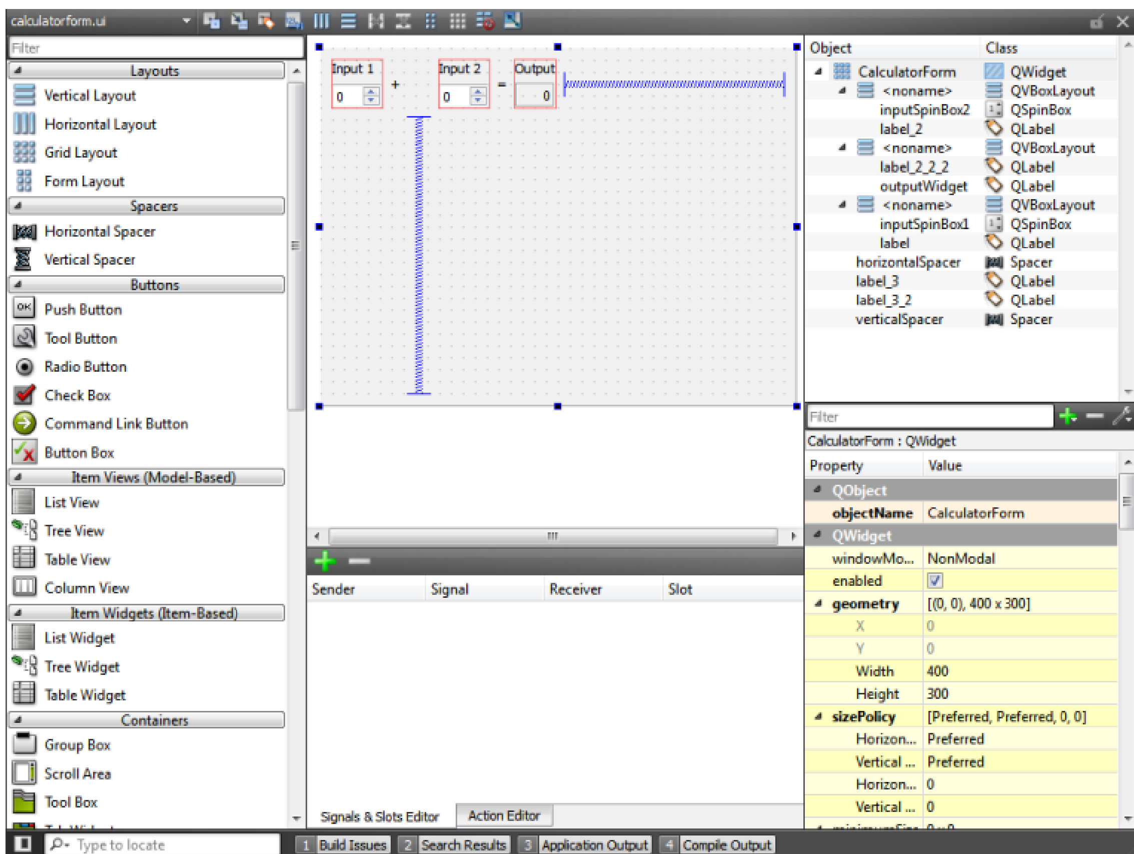


7. ábra: Megjelenítés Linux, Windows és Mac OS környezetben

Az esztétikához hozzájárul még, hogy a Qt az adott operációs rendszer stílusának megfelelő GUI elemeket jeleníti meg. Ez adott elem kinézetének részleteit,

megjelenítési módját jelenti. Mindezt automatikusan teszi a Qt, de tetszőleges stílus (style) beállítása is lehetséges.

Egyszerű GUI-k létrehozása programkód írás formájában nem vet fel jelentős problémákat, és nem is túlzottan időigényes. Viszont látványosabb felületek alkotása ilyen módszerrel sok időbe kerül [6]. Ennek a problémának a megoldására szolgál a Qt Creator design üzemmódja, mellyel grafikus úton lehet GUI-t fejleszteni. Különösen megkönnyíti változtatások implementálását, ha számos GUI elem található a felületen.



8. ábra: Design üzemmód kezelői felülete [7]

A Qt GUI programozás szemléltetésének céljából néhány rövid pszeudokód következik, köztük először egy megjeleníteni kívánt ablakhoz tartozó tipikus header:

```
// szükséges deklarációk include-olása
// prototípusok leírása
```

```
// megjelenítést végző osztály deklarációja
      // konstruktor deklarálása
      // signalok deklarálása
      // slotok deklarálása
      // GUI elemek deklarálása privát láthatósággal
```

Ablakhoz tartozó tipikus forráskód:

```
// szükséges deklarációk include-olása
// konstruktor definiálása
      // GUI elemek példányosítása
      // GUI elemek alaptulajdonságainak beállítása
      // elemek kapcsolatainak meghatározása
      // Layout beállítása
// slot metódusok definiálása
// signal metódusok kapcsolódásának meghatározása
```

Megjelenítést végző main függvény:

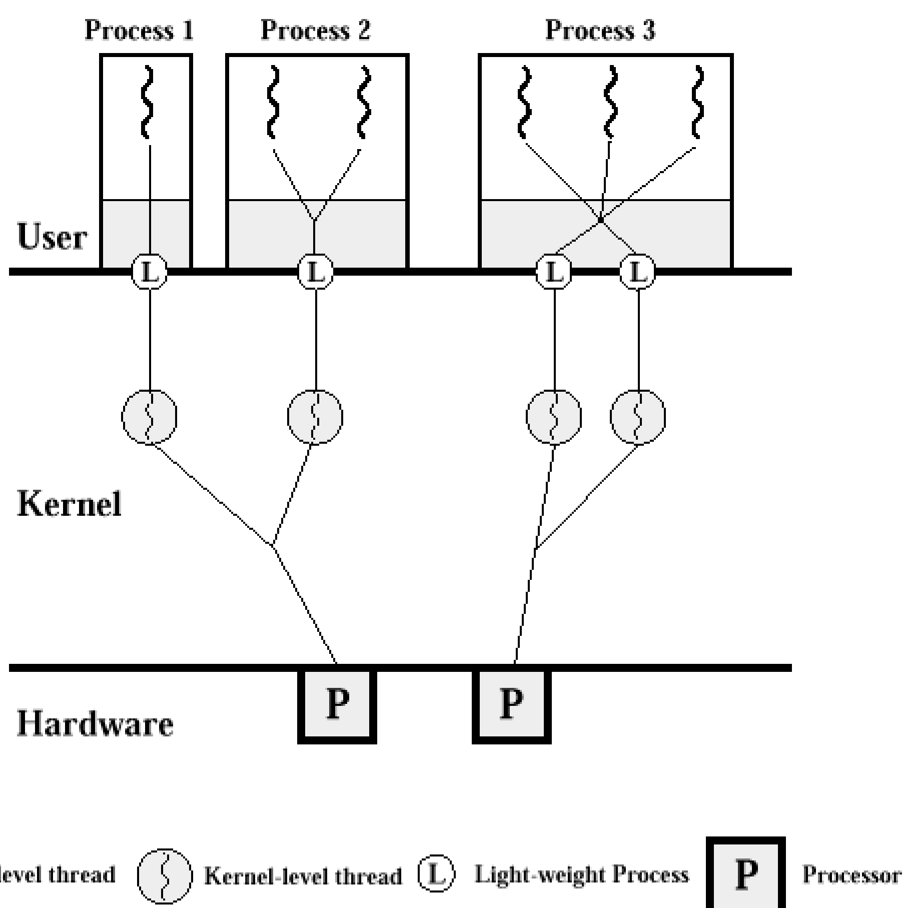
```
// szükséges deklarációk include-olása
// main függvény megvalósítása
      // alkalmazás példányosítása
      // megjelenítést végző osztály példányosítása
      // osztály megjelenítése
      // alkalmazás végrehajtása
```

3.1.2. Szálkezelés

A számítástechnikában a számítások elvégzésének legkisebb ütemezési egysége a végrehajtási szál (thread). Az ütemezést az operációs rendszer végzi. Szálnál nagyobb ütemezési egység a folyamat (process). Bár operációs rendszerenként változik a szálak és folyamatok implementációja, de a hoszt program futása a célnak tekinthető operációs rendszereken egyetlen folyamatnak minősül. Mivel a hoszt programnak egyszerre több eszköz programozását is lehetővé kell tennie, ezért a hozzá rendelt folyamatban több végrehajtási szálnak kell futnia. A hoszt program tehát például a 9. ábra második process-ének felel meg.

Szálak azok az ütemezési egységek, melyek egy folyamatban vannak és megosztják egymás között az erőforrásokat, például memóriát (a különböző folyamatok

viszont külön memória területet foglalnak le). Egy folyamat szálai hozzáférnek a folyamat utasításaihoz, kódjához és a kontextushoz is, ezen kívül ugyanazt a címtartományt látják. Az elmondottak lehetővé teszik párhuzamosan futó kód számára, hogy egyszerű maradjon a szálak közti információcsere. Nem érdemes külön folyamatokat létrehozni az egyes eszközök frissítésére, hiszen a folyamatok kontextusváltásának erőforrásigénye a szálakénál nagyobb. Hátrány viszont a szálak részéről, hogy olyan adatok esetén, melyekhez több szál is hozzáfér kialakulhatnak kritikus versenyhelyzetek. Ezek kijavítása nehéz, ezért megelőzésük kulcsfontosságú, erről szól a következő rész.



9. ábra: Szálak és folyamatok [8]

Biztonsági problémák szálaknál

Adott programkód akkor biztonságos, ha az úgy módosít adatokat, hogy egyszerre több szálon végrehajtva se okozzon hibát. Tehát a hoszt program végrehajtásakor egyszerre több szálnak is úgy kell futnia, hogy a közös címtartományt látva ne zavarják meg egymás működését. Gyakorlatilag ez azt jelenti, hogy a program állapotait kell egymással szinkronizálni valamilyen módszerrel:

- Reentráns programkód

  Olyan módon kell implementálni funkcionalitást, hogy azt egy szál részlegesen végrehajthassa, ugyanaz a szál újra futtathassa, és ezzel egyidejűleg akár egy másik szál is futtathassa. Mindezt úgy, hogy helyesen végrehajthassa a kódot az eredeti szál. Ehhez az állapotinformáció mentésére van szükség a végrehajtás számára lokális változókban, a memória ún. stack részén, statikus vagy globális tárolóhelyek nem használhatók.

- Kölcsönös kizárás

  Megosztott adatokhoz való hozzáférés időben egymás után, sosem párhuzamosan történik. Ez biztosítja, hogy egyszerre csak egy szál módosítja az adatokat. Itt is ügyelni kell azonban a szálak kiéheztetésének elkerülésére, és deadlock-ok kiiktatására.

- Szálhoz tartozó adatok lokális tárolása

  A változók úgy vannak tárolva, hogy mindegyik szálnak megvan róla a saját másolata. Ezek a változók megőrzik értéküket egy szál végrehajtásakor, hiszen egy párhuzamos másik szál kizárólag az ő saját változóit módosítja.

- Atomi műveletek

  Megosztott adatokhoz való hozzáférés atomi műveletekkel történik, melyeket nem szakíthat meg másik szál. Ehhez speciális gépi utasítások megléte

szükséges. Mivel a műveletek oszthatatlanok, az adatok mindvégig megfelelőek, nem lehetséges, hogy más szál módosítsa őket a hozzáférés alatt.

Szálak támogatása különböző programnyelvekben

Számos programnyelv támogatja a szálkezelést valamilyen formában. A C és C++ legtöbb implementációja nem biztosít direkt támogatást önmagában, inkább hozzáférést biztosít az operációs rendszerek által használt natív szálkezelő Application Programming Interface-hez (röviden: API). Más programnyelvek próbálják elvonatkoztatni a fejlesztőt a párhuzamosság és a szálkezelés szintjéről, mint például a Cilk vagy OpenMP. Megint más nyelveket párhuzamosság megvalósítására hoztak létre, például Ateji PX vagy CUDA.

Szálak támogatása Qt-ban

Szálkezelés a *QThread* osztályon keresztül történik. Lényegében egy osztályt kell származtatni belőle. A származtatott osztályban pedig felül kell definiálni a *QThread run( )* függvényét, majd a run függvényt *start( )*-tal futtatva elkezd futni a szál. A szálak létrehozásának egyszerűsített folyamatát mutatja be az alábbi példa[9]:

```
// szálakban futtatott metódusok osztályainak példányosítása
ClassA instanceA;
ClassB instanceB;


// példányok közötti kapcsolatok definiálása
instanceA.connect(instanceB, SIGNAL(signalB()), SLOT(slotA()));
instanceB.connect(instanceA, SIGNAL(signalA()), SLOT(slotB()));

// szálak példányosítása
QThread threadA;
QThread threadB;

// szálak osztálypéldányokhoz rendelése
instanceA.moveToThread(&threadA);
instanceB.moveToThread(&threadB);

// szálak indítása
threadA.start();
threadB.start();
```

A Qt dokumentáció szerint[9]:

Sem *QObject*, sem belőle származtatott osztály nem minősül szálbiztosnak. (Ez azért roppant kellemetlen, mert egy Qt programban szinte minden osztály a *QObject*ből származik.) Ez az egész eseménytovábbító rendszerre igaz. Fontos szem előtt tartani, hogy események érkezhetnek egy *QObject*ből származtatott osztálynak, amíg egy másik szál azon műveleteket végez. Ha függvényhívás történik egy *QObject*ből származtatott osztályra, amely nincs jelen az aktuálisan futó szálban, és az adott osztály eseményeket fogadhat, akkor mindenképpen meg kell védeni az osztály adattagjaihoz való hozzáférést egy mutex[1]-szel. Ellenkező esetben nem kívánt viselkedés vagy rendszerösszeomlás léphet fel műveletek végrehajtásánál.

Ezért ahol szükséges volt, ott én is mutexeket alkalmaztam a programban.

3.1.3. Signal és slot mechanizmus

A Qt egyik fő attribútuma az egyszerű GUI fejlesztés, ahogy az egyik előző alfejezetből kitűnik. Ehhez hozzájárul a vele használható ún. signal-slot mechanizmus. Szignálokat és slotokat objektumok közötti kommunikációra lehet használni. Más GUI fejlesztő keretrendszerektől leginkább emiatt tér el a Qt.

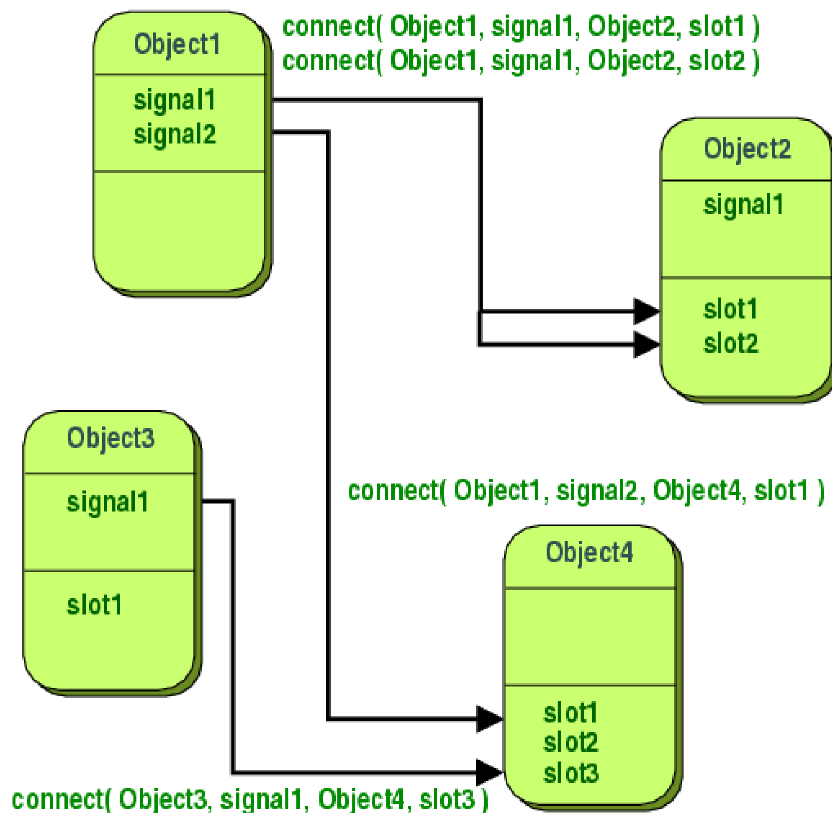
Függvénypointerek és hibáik

Fejlesztés során kívánatos, hogy tetszőleges GUI elemek között lehessen kapcsolatot létrehozni. Az egyik elem (például gördítősáv) változásakor egy másik elemet kell a változásról értesíteni. Ezt máshol függvényre mutató mutatókkal érik el. Ekkor a végrehajtást végző függvény kap egy pointert egy másik függvényre, és így meghívja azt a függvényt, ha szükséges.

Függvényre mutató mutatóknak két alapvető hibája van. Az első, hogy nem lehet velük biztonságosan kezelni a hivatkozott függvénynek átadott paramétereket. Ha hibás

[1] mutex (mutual exclusion): kölcsönös kizárást lehetővé tevő nyelvi elemek

paraméterekkel hívja meg a végrehajtó függvény a hivatkozott függvényt, arra nem figyelmeztet semmi. Másodsorban a végrehajtó függvénynek pontosan le kell kezelnie, hogy adott esetben melyik függvényre hivatkozzon. Ez túlságosan szoros kapcsolatot eredményez a végrehajtó függvény, és a hivatkozott függvény között. Ezeknek a problémáknak a kiiktatására Qt-ban a függvénypointerek kiváltására signal-slot párok szolgálnak, lásd 10. ábra.



10. ábra: Signal-slot kapcsolatok [10]

Adott signalt bizonyos esemény vagy események váltanak ki. Ekkor, ha adott slottal össze van kötve a signal, akkor a slot lefut. Az összekötést a *QObject* osztály *connect( )* függvénye végzi, a szétcsatolást a *disconnect( )*. A slot tulajdonképpen egy speciális tagfüggvénynek felel meg. A Qt-ban használt osztályoknak számos előre definiált signalja van. Ettől függetlenül egy ilyen osztályból leszármaztatva egy másik, saját osztályt, a programozó maga is tud definiálni új signalokat. Ez a slotokra is igaz.

A signal-slot mechanizmusnál az átadott paraméterek típusa mindig megfelelő, erről az ún. meta-object system gondoskodik. A signalokhoz, csakúgy mint a slotokhoz

tartozik egy szignatúra. Csak azonos szignatúrájú signal-slot, illetve signal-signal köthető össze. Így a fordító hibát fog jelezni, ha nem megfelelő signal-slot kapcsolatot hoz létre a programozó.

Egy signal lazán kapcsolódik egy slothoz: a signalt kibocsátó osztálynak nem számít, hogy melyik slot fut le a signal hatására. Bármennyi és bármilyen típusú paramétere lehet a signaloknak és slotoknak. Ezzel az objektum-orientált programozás egyik alapelve, az egységbezárás válik még inkább kézzelfoghatóvá, hiszen egymással kommunikáló objektumok így már nem kapcsolódnak olyan szorosan egymáshoz, mint például függvénypointerek esetén.

A meta-object compiler

Ez a működés a C++ egyfajta kibővítésének tekinthető. A Qt-ban ezt egy beépített meta-objektum rendszer (meta-object system) viszi véghez. Ez egyrészt a signal-slot mechanizmus működését, másrészt introspekciót biztosít. Introspekcióra van szükség a signal-slot kapcsolatok létrehozásához, és futási időben az ún. "meta-információ" kinyeréséhez. Ilyen információ például egy adott osztályhoz tartozó felsorolások (enum) listája, de ezekre épül például a Qt script modul működése is.

A standard C++ nem támogatja a meta-információ lekérdezését futási időben, erre pedig szüksége lenne a meta-objektum rendszernek. Ehhez a Qt egy külön fordítót használ, a meta-object compilert, röviden moc-ot. A moc feldolgozza a *QObject* osztálydefiníciókat, és a belőlük kinyerhető információt C++ függvényeken keresztül elérhetővé teszi. Mivel a moc minden funkciója C++-ban van implementálva, a Qt meta-objektum rendszere bármelyik C++ fordítóval használható.

A meta-információ kinyerésének a lépései a következők:

- A Q_OBJECT makró (amelynek szerepelnie kell minden *QObject*-ből származtatott osztálydeklarációban) deklarál néhány introspekcióhoz szükséges függvényt (*metaObject( ), tr( ), qt_metacall( )*... stb).
- A Qt moc eszköze implementációkat generál a Q_OBJECT által deklarált függvényekhez, és az összes signalhoz.

- A *QObject* tagfüggvényei (mint például *connect( )* és *disconnect( )*) felhasználják az introspekciós függvényeket a saját feladataik elvégzéséhez.

Ezeket a lépéseket a qmake, a moc és a *QObject* osztály tartja az irányítása alatt, ezért a programozónak ritkán kell ezekkel foglalkoznia.

Megemlítendő, hogy kapcsolat nem csak 1 signal és 1 slot között jöhet létre. Bármennyi signal kapcsolódhat 1 slothoz. De ennek a fordítottja is igaz: bármennyi slothoz kapcsolódhat 1 signal.

Sőt, akár signalokat is össze lehet egymással kötni. Ha az összekötés típusa signal-signal, akkor az elsőhöz tartozó esemény megtörténtekor az első signal kibocsátása után bekövetkezik a második signal kibocsátása, azonnal.

3.1.4. Szkript használat

Szkriptnyelv használata mellett szól, hogy kifejezett fordítás és linkelés nélkül végrehajthatóak a bennük lévő utasítások. A szkriptnyelveket általában egy értelmező hajtja végre a forráskódjukból. Ezzel szemben egy C vagy C++ programot fordítani és linkelni kell, hogy gépi kóddá legyen átalakítva, és az végrehajtható legyen. A lefordított program felhasználójának nem feltétlenül áll rendelkezésére a forráskód, és ekkor a program módosítására sincs módja. Szkripteknél viszont a felhasználó létrehozhatja és módosíthatja a végrehajtandó kódot. Az értelmező többnyire gépi kódra fordított programként fut (ez a QtScript esetében sincs másként). A szkriptek hátránya a lefordított programokkal szemben a lassú végrehajtás. Tízszeres végrehajtási idő sem minősül szokatlannak.

Mindezekből következik, hogy a hoszt program rugalmasságát, konfigurálhatóságát leginkább szkript használatával érdemes megoldani, hiszen így a végső felhasználó kezében maradhat az alkalmazásvezérlés. Meglehetősen alacsony

szinten tud majd a felhasználó utasításokat megadni, egy bizonyos feltétellel. A feltétel az, hogy alacsony szintű legyen a szkript és a C++ program közötti interfész. Tehát a C++ program alapvető információkat tudjon közölni a szkripttel, és a szkript a C++ program alapvető funkcióit tudja meghívni. Így egy komplex GUI létrehozását mellőzni lehet. Egyúttal a hoszt oldali firmware-frissítő jövőbeli alkalmazhatósága is sokkal inkább biztosított.

Viszont hátrányokkal is jár ez a megoldás: vagy hozzáértő felhasználónak kell kezelnie a szkriptet, vagy szigorúan jelölni kell, hogy a szkriptben milyen beállításokat kell vagy lehet változtatni, és hogy mire lehet változtatni. Mivel a kezelő végeredményben beágyazott rendszer firmware-frissítését fogja végezni, ezért feltételezhető valamiféle affinitás ehhez a technológiához. Valószínűleg képes lesz megbirkózni a feladattal, ha gondosan dokumentálva van a szkript működése.

Qt Script

A Qt 4.3 verziója óta van lehetősége a felhasználóknak alkalmazásaikat szkriptből vezérleni, amelynek hivatalos neve Qt Script. A Qt Script az ECMAScript nyelvre épül, melyet az ECMA-262 standard definiál. A Microsoft JScriptje, és a Netscape JavaScriptje is az ECMAScript standardra épül [11]. Ebből kifolyólag számos forrásból lehet tájékozódni a témával kapcsolatban. Sok könyv, interneten található leírás ad támpontot a szkript helyes használatára.

A Qt Scriptről általában

A leggyakrabban használt vezérlési struktúrák (*if*: feltételes utasításvégrehajtás, *for* és *while* ciklusok) ugyanazok, mint a C++ esetében, ráadásul az értékadó, relációs és aritmetikai operátorok is többnyire egyezőek [5].

Egy C++ programozó szemszögéből nézve a Qt Script lényeges tulajdonsága, hogy a változók típusa nincs explicit módon deklarálva. A *var* kulcsszóra van egyedül szükség egy változó deklarálásához. Csak olvasható változók a *const* kulcsszóval vannak deklarálva. Egy másik említésre méltó jellemző, hogy nincs *main( )* függvény. Bármely, függvényen kívül eső kód azonnal végrehajtásra kerül, fentről lefele haladva.

A C++-szal ellentétben nem szükséges pontosvesszővel lezárni egy utasítást. Az értelmező összetett szabályokat alkalmazva maga ki tudja találni, hogy hol végződik egy utasítás. Ennek ellenére mégis ajánlott a pontosvessző használata, hogy könnyebben elkerülhetőek legyenek a szintaktikai hibák.

A Qt Script fenti fundamentális jellemzőinek tisztázása után részletesebb bemutatásra is lehetőség nyílik. Először a leggyakrabban használt típusok, osztályok, majd egy C++-belihez hasonló osztálystruktúra ismertetése következik.

Adattípusok (konstruktorok)	
Object	Az osztályra jellemző funkcionalitást biztosít
Function	Egy Qt Script függvényt tartalmaz
Array	Elemekből álló átméretezhető vektor
String	Unicode string-et tárol
Boolean	Kétértékű logikai változót tárol (*true* vagy *false*)
Number	Lebegőpontos számot tárol

1. táblázat: Qt Script adattípusok

Ha nincs biztosítva egy kezdeti érték egy változó deklarálásakor, akkor alapbeállításként *undefined* értékű lesz a változó. Ez az *Undefined* típus speciális értéke. Az értékadó utasítást (=) használva később bámilyen típushoz tartozó bármilyen értéket rendelhetünk ehhez a változóhoz. A szkriptben ellenőrizhető a típus a *typeof* operátor segítségével. Ez szövegesen jeleníti meg a hozzárendelt adattípust. Qt script-ben öt primitív adattípus van meghatározva: Undefined, Null, Number és String. Az Undefined és Null típusok az undefined és null konstansok speciális típusai. A többi primitív adattípust az előző táblázat részletezi.

Változók ugyancsak tárolhatnak objektum (*object*) típusokat a primitív adattípusokon túl. Ezek közé tartozik például az *Object*, *Array* és *Function* típus. A primitív adattípusok az objektum típusoktól elkülönülnek: a primitív adattípusok tekinthetők úgy, mint a C++ adattípusai. Ezek a *new* operátor nélkül vannak létrehozva, és érték

szerint másolódnak. Ezzel szemben az objektum típusokat new operátorral kell létrehozni. Az ilyen típusú változók pedig csak az objektumra mutató referenciát tárolják. Az allokált objektumok felszabadításáról nem kell gondoskodni, a garbage kollektor megteszi ezt.

Szót kell ejteni még az *Array* (tömb) típusú változók néhány tulajdonságáról. Ilyen például, hogy nem feltétlenül szükséges megadni inicializálásnál a tömb méretét, ugyanis az automatikusan változik, ha az adott tömbhöz új elem lesz definiálva. Egy tömbben bármilyen elem tárolható, amit értékül lehet adni változónak. Tehát lehet primitív adattípusú elem, objektum, függvény vagy akár másik tömb. Ha egy elem ki van hagyva, akkor a típusa *undefined* lesz. A tömb egyik fontos tulajdonsága (*property*) a hossza. Ez megkapható például a *var arrayLength = arrayName.length* utasítással. Egy *property* egy asszociációt képvisel egy név és egy objektumhoz tartozó érték között. Más szavakkal a *property*-k halmaza az objektum.

Most pedig a C++-belihez hasonló osztálystruktúra bemutatása következik. Egy ilyen struktúra felépítése azért volt kívánatos, mert én C/C++ tapasztalattal rendelkezek, és így könnyebben tudtam fejleszteni a kód Qt Script részét. Ez a struktúra számottevően függvényekre (*function*) épül (itt megjegyzem, hogy Qt Script-beli függvények paramétereinek típusa nincs deklarálva, és nincs explicit típusa a visszatérési értéknek sem).

A Qt Script egy objektumokon alapuló, objektum orientált nyelv, szemben a C++-szal vagy Java-val, melyek osztályokat alkalmaznak. Az osztályok helyett a Qt Script alacsonyabb szintű mechanizmusokat biztosít, amivel ugyanaz az eredmény érhető el. Az egyik ilyen mechanizmus, amely osztályok létrehozását teszi lehetővé, a konstruktor. A konstruktor egy olyan függvényt jelöl, melyet new operátorral lehet meghívni. Például egy *Body* objektum konstruktora lehet a következő:

```
function Body(mass, volume) {
        this.mass = mass;
        this. volume = volume;
}
```

A *Body* konstruktornak két paramétere van, és inicializálja a *mass* és *volume property*-ket, a konstruktornak adott paraméterek alapján. A *this* kulcsszó hivatkozik a létrehozandó objektumra. Objektum *property* hozzáférésnél szükséges a *this* kulcsszó használata. A Qt Script-ben egy objektum alapvetően *property*-k halmaza, melyeket hozzá lehet adni, el lehet távolítani, vagy módosítani lehet. Egy *property* az első definiálásakor lesz létrehozva. Tehát amikor *this.mass* és *this.volume* értéket kap, akkor jönnek létre a *mass* és *volume property*-k. A *Body* objektum példányosításához a *new* operátort kell használni a következőképpen:

```
var body = new Body(10, 20);
```

A body változó besorolását szemléltetendő: A body változón alkalmazva a *typeof* operátort eredményül *Object*-et kapunk, nem *Body*-t, hiszen az operátor a típust adja vissza. Ha azt kell meghatározni, hogy egy objektumot a *Body* konstruktora hozta-e létre, akkor az *instanceof* operátort kell használni. A *body instanceof Body;* logikai *true* értékkel fog visszatérni.

A Qt Script-ben bármely függvény használható konstruktorként. Ha a függvény viszont nem módosítja a *this* objektumot, akkor nincs értelme meghívni konstruktorként a függvényt. Egyúttal az is leszögezhető, hogy meg lehet hívni egyszerű függvényként egy konstruktort, azonban többnyire ennek sincs értelme.

Eddig kiderült, hogy hogyan érdemes definiálni egy konstruktort, és hogyan lehet *property*-ket ("tagváltozókat") adni a létrehozott objektumhoz. Általában tagfüggvények hozzáadására is szükség van. Mivel a függvények kezelését nagymértékben támogatja a Qt Script, ezért ez meglehetősen egyszerű. Ezt szemlélteti a következő példa:

```
function Body(mass, volume) {
        this.mass = mass;
        this.volume = volume;
        this.expand = function(dV){
                this.volume = this.volume+dV;
        };
}
```

A *Body* konstruktorban ezúttal már egy *expand( )* tagfüggvény is szerepel. Ezt már a C++-ban megszokott módon lehet meghívni:

```
var body = new Body(10,20);
body.expand(5);
```

Ezzel a megközelítéssel minden *Body* példánynak megvan a saját *expand* tulajdonsága, *property*-je. Mivel célszerű, hogy ez a tulajdonság azonos legyen minden Body példányra, ezért érdemes lenne csak egyetlen helyen tárolni, nem pedig példányban külön-külön. A Qt Script-ben ezt úgy lehet elérni, hogy prototípusként (*prototype*) definiáljuk a függvényt. A prototípus egy olyan objektum, ami más objektumok számára szolgál támpontként. Tulajdonságok meghatározott halmazát definiálja. Az ilyen megközelítés előnye, hogy a prototípus objektumot változtatva a változások azonnal tükröződnek minden objektumnál, amik adott prototípussal lettek létrehozva. A Body "osztály" ezzel a megközelítéssel:

```
function Body(mass, volume) {
        this.mass = mass;
        this.volume = volume;
}
Body.prototype.expand = function(dV) {
        this.volume = this.volume+dV;
};
```

Így már az expand tulajdonság a konstruktoron kívül lett létrehozva, a Body.prototype objektumban. A Body példányosításakor az új objektum egy belső mutatót tartalmaz a Body.prototype-ra. Ha a Body objektumban nincs definiálva egy adott tulajdonság, akkor az objektum a prototípusára hivatkozik. Ezért felel meg a prototípus tagfüggvények tárolására. Csábító lenne a tagváltozókat is a prototípusban tárolni, viszont tagváltozó írásánál nem a prototípusban változna meg az adott változó, hanem létrejönne magában a Body objektumban. Ez pedig (mivel már az objektumban is létezik az adott nevű tulajdonság) beárnyékolná a prototípusbeli tulajdonságot. Hiszen az csak akkor hivatkozható, ha az objektumban nincs deklarálva.

Osztályokon alapuló nyelvben az öröklést felhasználva lehet speciális objektum típusokat definiálni. Ezt Qt Script-ben ugyancsak prototípusokkal, és a *call( )* függvénnyel lehet elérni. A *call( )* minden függvény objektumra definiálva van (konstruktorokat is beleértve). Ezért alkalmas a szülőosztály konstruktorának meghívására.

```
function Body(mass, volume) {
        this.mass = mass;
        this. volume = volume;
}
Body.prototype.impulse = function( ) { return 0; };

function Cart(mass, volume, speed) {
        Body.call(this, mass, volume);
        this.speed = speed;
}
Cart.prototype = new Body;
Cart.prototype.impulse = function( ) { return mass*speed; };
```

A fenti példakód publikus konstruktorral, tagváltozókkal és virtuális függvénnyel rendelkező Body szülőosztályt és abból származtatott *Cart* osztályt realizál, amelynek van virtuális *impulse( )* függvénye és *speed* tagváltozója. A fenti megközelítéssel az öröklés mellett a polimorfizmus is megvalósítható. Strukturált szkript kód kapható így, mely segít a program C++ részével való letisztult interfészek létrehozásában. Most pedig a szkript és a C++ kód összekapcsolásának részletei következnek.

Qt Script és C++ összekapcsolása

A bevezetőben említett szkript értelmező a szkript használat kulcsszereplője. Ez a *QScriptEngine* C++ osztálynak egy példánya. Ennek meg kell hívni a kiértékelést végző függvényét (*evaluate( )*), amelynek argumentuma maga a szkript, célszerűen *QString* formátumban. A kiértékelés visszatérési értéke a kiértékelés eredménye lesz, amely a *QScriptValue* egy példánya. Ez aztán C++-beli típusokká konvertálható.

Tetszőleges tulajdonságot (*property*) lehet a szkript értelmezővel az eredményül kapott *QScriptValue*-ban beállítani. Ez a szkript környezetbe helyezi az adott tulajdonságot, tehát szkriptből elérhetővé válik a tulajdonság. Tömbök, függvények, vagy akár *QObject*ből származtatott objektumok is átadhatók a szkriptnek a Qt-ban definiált változók mellett.

```
QScriptEngine engine;
QScriptValue qscriptVal = engine.evaluate("var multiply = 2*1");
QObject *myObject = new MyObject;
QScriptValue myObjectForScript = engine.newQObject(myObject);
qscriptVal.setProperty("myObject", myObjectForScript);
```

A fenti kód kiértékel egy szkriptet, majd továbbadja annak a *myObject* nevű *QObject* objektumot. Ez lehetőséget nyújt információ átadására a C++ kódból a szkript felé. A szkript már egy *QScriptValue*-t kap, konvertáláson esik keresztül minden adat. A program megírásánál a C++-ból szkriptbe, és szkriptből C++-ba konvertálás eredményeinek típusára is ügyelni kell.

Információ közlésére függvényeken keresztül is lehetőség van. Így már az információ a szkriptből a C++ oldal felé is közlekedhet. A kommunikáció ilyen formája nem magától értetődő, hiszen a szkript dinamikusabban viselkedik, mint a C++ kód, és ezért a standard C++ nem is biztosít eszközöket C++-beli tagfüggvények meghívására. Ez csak a C++ egy kibővített változatában, a Qt-ban található meta-object compiler (moc) segítségével vihető végbe. A előző alfejezetben bővebben esett szó erről az eszközről. A moc segítségével a Qt-ban létrehozott, *QObject*ből származtatott osztályok publikus függvényeinek speciális változatait, az ún. slotokat meg lehet hívni szkriptből. Azok pedig megfelelő paraméterekkel és visszatérési értékkel ellátva hatékony interfésznek bizonyulnak, melyek a szkript oldalról is használhatók.

A fejlesztést segítő legfőbb összetevők ismertetése után rátérek a hoszt oldali szoftver részleteire a következő alfejezetben.

3.2 A szoftver rendszerterv részletei

A 3. fejezet bevezetőjében már kitértem a hoszt oldali szoftver 3 rétegére: azok legfőbb feladatait ismertettem. Azonban a Qt funkcióira hivatkozva most már meg is indokolhatom, hogy miért van szükség egyáltalán rétegekre, és hogy miért pont olyanok lettek a rétegek, amilyenek.

Egy különálló GUI réteg magyarázható a következőképpen. A Qt-ban viszonylag könnyű a GUI fejlesztés. Ráadásul elképzelhető, hogy különböző környezetekben futtatva más-más felhasználói felület a kívánság. Ezért valószínű, hogy egyszer fel fog merülni az igény a lecserélésére. Emiatt érdemes a GUI-t egy könnyen leváltható, elkülönülő rétegként definiálni.

Egy különálló kapcsolati réteget célszerű kialakítani a kommunikációban felmerülő problémák könnyű kijavítására, ugyanis egy központi, a számítások zömét végző réteget a kapcsolati réteggel egybeépítve nehezebbé válik a program hibáinak pontos felderítése. Egy leválasztott kapcsolati réteg könnyebben fejleszthető, utólag könnyebb új protokollokat hozzáadni.

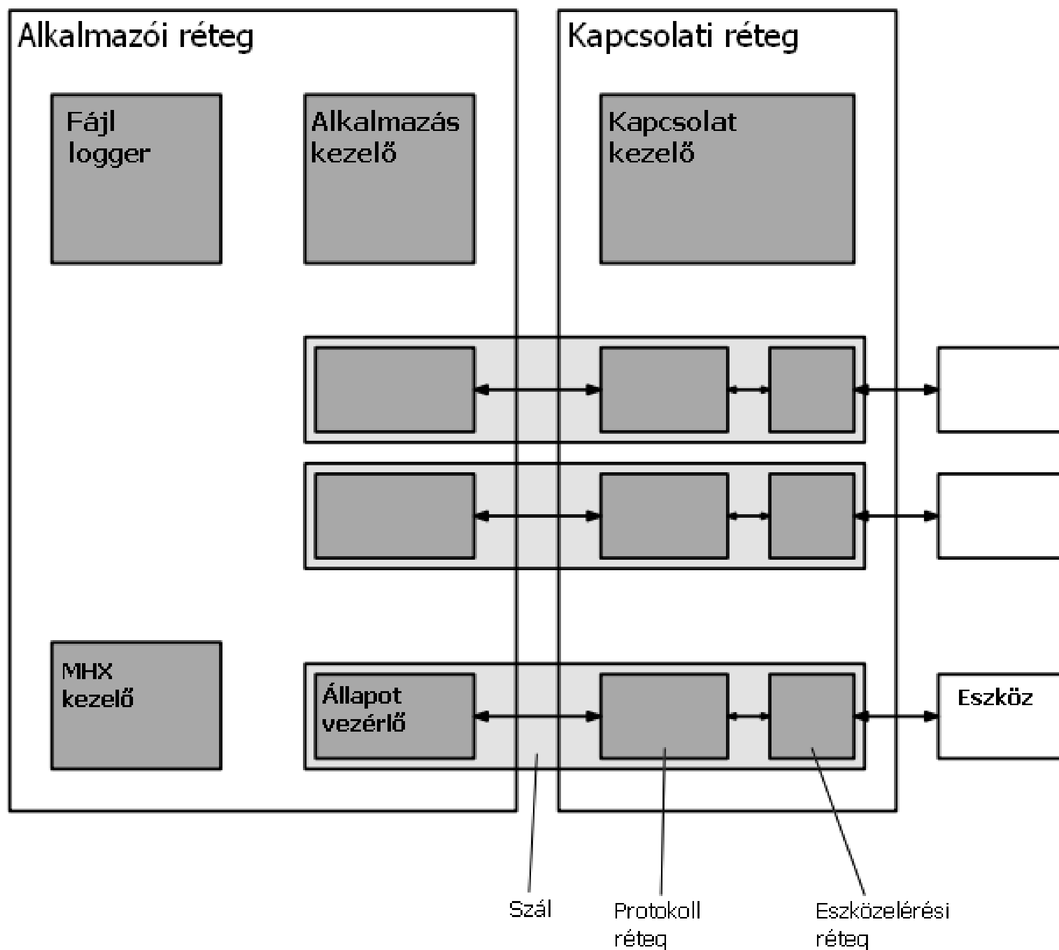
Az előző két réteg leválasztása miatt már csak a számítások zömét végző központi rész, az alkalmazói réteg marad. Az alkalmazói réteg önmagában is bonyolult, így néhány további modulra szétbomlik. A modulok közti adatcsere szerteágazó, így feltétlenül megéri egy rétegben hagyni őket.

A rétegek feladatai mentén érveltem eddig a tagoltság mellett. Azonban maga az objektum-orientált programozási paradigma, annak egyik elve, az egységbezárás is indokolja rétegek kialakítását. Bár ez az egységbezárás az osztályszintű egységbezárásnál magasabb. Így lesz a program moduláris és újrafelhasználható.

3.2.1. Alkalmazói réteg

A program kulcsfontosságú részei itt futnak le, ezért a másik két réteg funkciói ennek a rétegnek az igényeihez lettek igazítva. Emiatt kezdem a hoszt program részletes

bemutatását ezzel a réteggel, így nem kell előreutalásokba bocsátkoznom a többi réteg tárgyalásakor.



11. ábra: Alkalmazói és kapcsolati réteg

Az alkalmazói réteget négy fő modul alkotja. Ezek közül három C++-ban van implementálva: a fájl loggoló, MHX kezelő (hex fájl kezelő modul) és az alkalmazás kezelő. Qt Script-ben valósítottam meg az állapotvezérlő modult. Ezek gyakran átadják egymásnak a processzor vezérlését, szorosan együttműködve futnak.

Alkalmazáskezelő

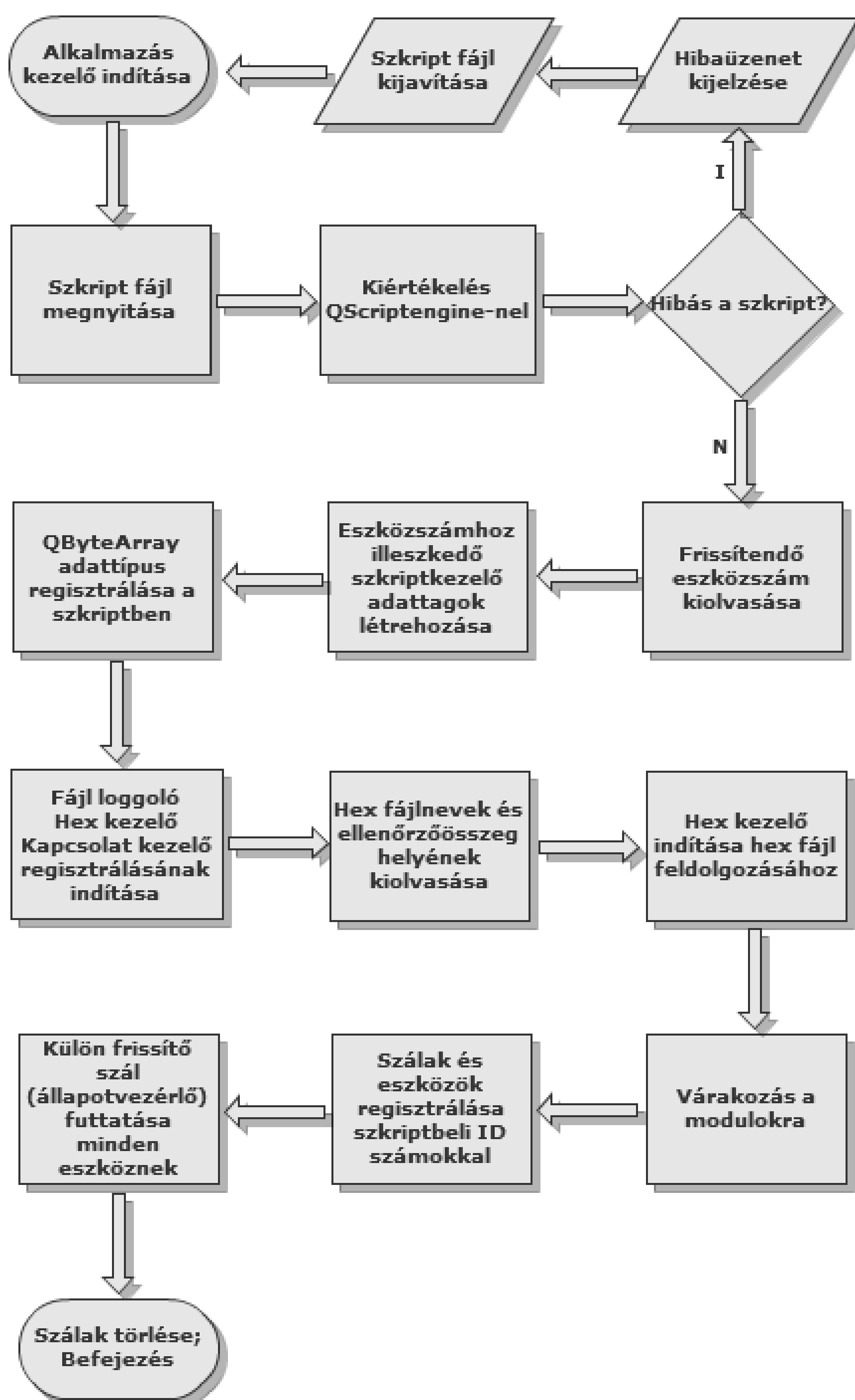
Ez a modul az alkalmazói réteg és a hoszt program fő koordinátora. A feladatai közé tartozik:

- szkript indítás,
- futó szálak kezelése,
- felhasználói parancsok végrehajtása.

A szkript indítás menete a következő. A felhasználó beírja a GUI-ba egy használható Qt script fájl elérési útját és fájlnevét. Ezután rákattint a Set gombra. A GUI ekkor ellenőrzi, hogy tényleg létezik-e olyan fájl, és hogy meg lehet-e nyitni. Ha igen, akkor klikkelhetővé válik az addig kikapcsolt Execute gomb. Erre rákattintva egy signalt kap az alkalmazáskezelő. A signal továbbadja a szkriptfájl nevét, elérési útját az *executeScript* publikus slotnak.

Az *executeScript* alaphelyzetbe állítja az alkalmazáskezelő privát bool adattagjait, amelyek jelzik számára, hogy mikor csatlakozott a többi modul a szkripthez. (Ugyanis az MHX kezelő, fájl logger, kapcsolat kezelő és protokoll réteg példányait a szkript számára elérhetővé tettem, ott regisztráltam, hogy az meghívhassa ezek publikus slotjait.) Ezután beolvassa a szkript fájlt, kiértékeli egy *QScriptEngine* példánnyal, és az eredményt egy *QScriptValue* típusú változóban (az egyszerűség kedvéért erre később *ScriptResult* néven hivatkozok) eltárolja. Ha hiba lép fel kiértékeléskor, akkor arra egy felugró ablak figyelmeztet, kiírva a hiba sorát és az értelmező hibaüzenetét (pl.: undefined symbol). A *ScriptResult*ból ki lesz olvasva a frissítendő eszközök száma. Ugyanis a többszálú futás miatt van szükség az eszközönként elváló szkriptkezelésre. Hiszen a *QScriptEngine* példányok nem csak olvassák, hanem írják is a szkriptet, amit egyazon *QScriptValue*-n végezni egyidejűleg nem lehet. Mert ez segmentation faulthoz vezetne. Ezért a régi *QScriptEngine* és *ScriptResult* törölve lesz, és frissítendő eszközök számával egyező *QScriptEngine* és *ScriptResult* lesz létrehozva. A kiértékelést végrehajtó *QScriptEngine*-eket mindegyik modul megkapja, amely regisztrálja magát a szkriptben. Gyakorlatilag ez azt jelenti, hogy a moduloknak saját *ScriptResult* és *QScriptEngine* 2 dimenziós (2D) pointere van. Ezek a pointerek az alkalmazáskezelő *ScriptResult* pointer tömbjére (2D pointer adattagjára) és *QScriptEngine* pointer tömbjére fognak mutatni, ugyanis az alkalmazáskezelő kiad egy signalt, melyet a modulok azzal kezelnek le, hogy a 2db 2D pointerüket az alkalmazás kezelő megfelelő adattagjaira állítják. Ezt követően a modulok a *newQObject( )* metódussal először az

értelmezőhöz tartozó *QScriptValue* értékké alakítják magukat, aztán a *setProperty( )* függvénnyel hozzáadják magukat a *ScriptResult*-hoz.



12. ábra: Folyamatábra az alkalmazáskezelő futásáról

Ezután a *ScriptResult*ból kiolvasásra kerül, hogy milyen hex fájlok tartoznak az egyes frissítendő eszközökhöz. Ha esetleg azonos hex fájl tartozik több eszközhöz, akkor is csak egyszer kell feldolgozni az adott hex fájlt. Ezután létrejönnek a különálló firmware-frissítési szálakat futtató modulok, amelyek az általam definiált *Thread* osztály példányai.

Végül, de nem utolsó sorban az alkalmazáskezelő kiad egy *initHexProcessing* signalt, mely paraméterként szolgáltatja a firmware-frissítéshez felhasználni kívánt fájlokat, redundancia nélkül. Ez jelzi az MHX kezelőnek, hogy megkezdheti a kijelölt hex fájlok feldolgozását.

Az alkalmazáskezelő ezután vár a többi modul jelzésére. Az MHX kezelő akkor jelez, ha a szkriptben elérhetővé tette a hex fájlok adatait (összefüggő memóriablokk tartalmak, azok kezdőcíme). A fájl logger akkor jelez, ha regisztrálta magát a szkriptben. A kapcsolatkezelő jelez a protokollréteg helyett is: saját magát és a frissítendő eszközök számával megegyező számú protokollréteget regisztrál a szkriptben. Ha mindhárom jelzés beérkezett, akkor az alkalmazás kezelő meghívja az összes *Thread run( )* függvényét. Ezzel elindul a firmware-frissítés.

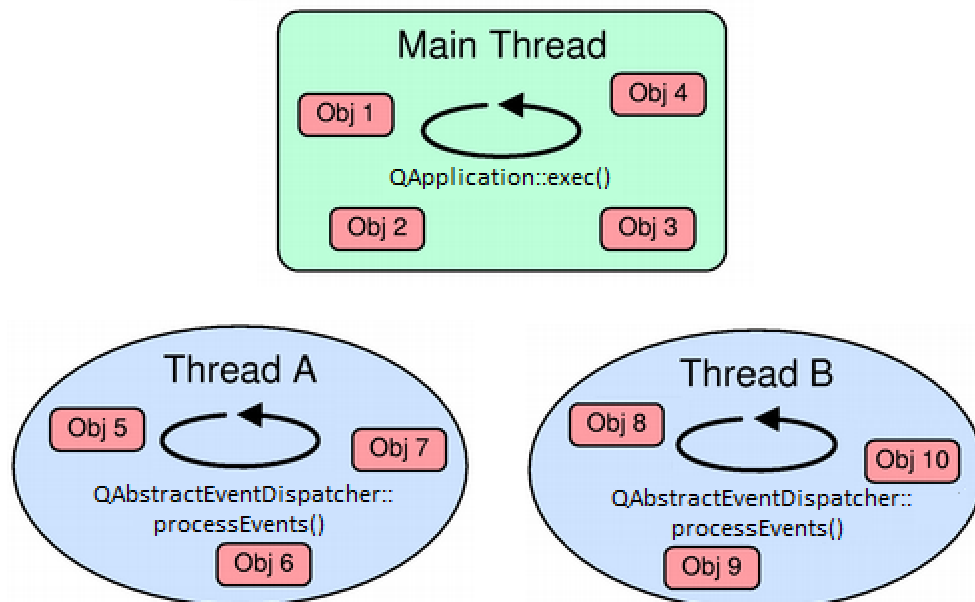
Az alkalmazáskezelő modul dolgozza föl a felhasználói parancsokat is. Ez kimerül a frissítés elindításának és megállításának lekezelésében. Az ezekhez szükséges két jel (signal) a GUI réteg felől érkezik. Ott két gomb (az Execute és a Stop gomb) adja ki a jeleket. Az Execute gomb indítja az *executeScript*et, ahogyan azt az előző részben leírtam. A Stop gomb pontos funkcióját viszont még nem ismertettem. A Stop megnyomása után lefut a *stopScript* slot az alkalmazáskezelőben. Ez az összes eszközhöz tartozó frissítési szálat leállítja, ha azok még futnának. Ezután kibocsátja a *stopEventOccurred( )* jelet. Ez többek között jelez a kapcsolat kezelőnek, ami ennek hatására meghívja az eszközelérési rétegek és a protokoll rétegek destruktorát.

Most már nagy vonalakban bemutattam a program központi részének működését. Ezek során bizonyos funkciók tisztázásánál említésre szorult a többi modul működése is, amelyet részletesen a következő szakaszban mutatok be.

Szálak

A *Thread* osztály példányai gondoskodnak az egyes szálak futtatásáról. Az osztály a *QThread*-ből származik azzal a különbséggel, hogy annak virtuális *run( )* függvényét felüldefiniáltam. Ily módon a *run( )* képes elindítani a szkriptben mindegyik eszközre definiált firmware-frissítő függvényt, ugyanis minden *Thread* példány konstruktora megkapja az alkalmazás kezelő *ScriptResult*-jára és *QScriptEngine*-jére mutató pointereket. Ezen kívül az adott szál és a hozzá tartozó eszköz azonosítására szolgáló ID számot (N darab eszköz esetén az ID 0-tól N-1-ig tart). A *Thread run( )* függvénye így már hivatkozhat a *ScriptResult* megfelelő elemének *run( )* függvényére, és így elindíthatja azt.

A szkriptbeli *run( )* függvény futtatása után ez a modul is jelzi a hibákat, akár csak az alkalmazás kezelő. Erre azért van szükség, mert az alkalmazáskezelő nem tudja ellenőrizni, hogy a többi modul *ScriptResult* módosítása után hibamentesen végrehajtható marad-e a szkript.



13. ábra: Várakozási sémák a program különböző szálaiban

Szálban várakozásra is ki kell itt térnem: A program fő szála rendelkezik event loop-pal. Azonban a frissítést végző szálak nem. Egy event loop felelős azért például, hogy a

program csak akkor záródjon be, ha a felhasználó bezárja. Event loop nélkül a futás után magától bezáródna a program. Event looppal egy várakozási séma valósul meg. A program utasításokra vár, és végrehajtja azokat egészen addig, amíg meg nem lesz hívva az *exit( )* függvény. Egy event loop a *QApplication::exec( )* függvényével hozható létre.

Persze itt felmerül a kérdés, hogy a szálakban akkor mégis milyen módon várakoztam, például az eszközöktől érkező válaszra. Ezt végeredményben a *processEvents( )* függvénnyel valósítottam meg, amivel összetettebb várakozási sémák is lekezelhetővé válnak:

Az adott szál várakozási sémáját részletesen a protokoll réteg tárgyalásánál mutatom be, ugyanis ott egy QTimerrel együtt használtam a várakozás implementálására. Ezért sokkal szemléletesebb ott részletesen ismertetni.

Itt viszont szükséges megemlíteni az egyes kritikus programrészek védelmi mechanizmusát többszálú futás esetén. Ezt a védelmi mechanizmust mutexekkel valósítottam meg. Ez azt jelenti, hogy adott szál mutexszel biztosítja saját maga számára egy védett programrészlet futtatásának kizárólagos jogát. A többi szálnak ekkor várnia kell egészen addig, amíg a foglaló szál nem végez a programrészlet futtatásával. Ha végez, akkor következhet a többi szál. Egyszerre csak egy szál futtathatja a védett programrészletet. Ehhez elegendő a védett programrészlet elején a *QMutex::lock( )*, a végén pedig a *QMutex::unlock( )* függvény meghívása.

MHX kezelő

Az MHX kezelő a hex fájl feldolgozását végzi. Ehhez segítséget nyújt az alkalmazás kezelő jele, mely az alábbi adatokat továbbítja számára:

- eszközök száma,
- különböző hex fájlok száma,
- fájl-eszköz párosító tömb,
- fájlnevek és elérési utak 2 dimenziós tömb formátumban,

- ellenőrzőösszeg memóriacíme.

Ez a jel az MHX kezelő megfelelő slotjába (*inithexProcessing(...)*) csatlakozik. Ott példányosul a különböző hex fájlok számával egyező számú *OnehexFile* objektum. Ezek konstruktora megkapja az objektumokhoz rendelt fájlnevet az elérési úttal. Ezután meg van hívva minden objektumra a tényleges fájlfeldolgozást végző függvény (*processhexFile( )*). Miután ez lefutott, az MHX kezelő regisztrálja magát a szkriptben, majd a programozó hex fájlok által kijelölt memóriablokk tartalmakat, és a memóriablokkok címét is átadja a szkriptnek a kiszámolt ellenőrzőösszegek mellett. Így a szkriptet módosító személy teljes irányítást kap a frissítés menetére vonatkozóan. Ugyanakkor nem kell túlságosan részletekbe menően foglalkoznia a hex fájl feldolgozással.

Ha a fenti folyamat véget ér, akkor az MHX kezelő küld egy jelet (*AllhexFileProcessed( )*) az alkalmazáskezelőnek. Ezzel tudatja, hogy beleegyezik a szkript *run( )* függvényeinek az indításába, azaz a firmware-frissítés lefuttatásába. Ezzel az MHX működésének lényegét kifejtettem. Innen látszik, hogy a fájlfeldolgozás elemi feladatait a OnehexFile objektumok végzik, így ennek az osztálynak a leírásával folytatom.

OnehexFile

Az osztály konstruktorában ellenőrzésre kerül, hogy létezik-e a paraméterben kapott hex fájl egyáltalán. Ha nem, akkor hibaüzenet lesz kiadva. Ha igen, akkor a *processhexFile( )* függvény fel tudja dolgozni a hex fájlt. A feldolgozás menetének megértését nagyban könnyíti, ha tisztázom a hex fájl körül felmerülő kérdéseket, mielőtt bemutatom a *processhexFile( )* függvényt.

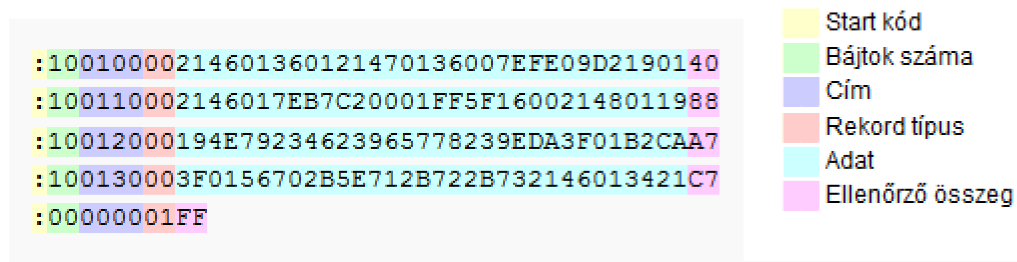
Maga a hex fájl kifejezés intel HEX formátumú fájlokat takar. Ilyen fájlokat mikrokontrollerek, DSP-k és EPROM-ok programozásához használnak. A C++, C vagy assembly kódot egy fordítóprogram gépi kóddá alakítja, ezt pedig intel HEX formátumú fájlban tárolja el. Ezt egy programozó alkalmazás később importálhatja, és felhasználhatja. Használata az 1970-es évekre nyúlik vissza.

Egy intel HEX fájlban az adatok rekordokban helyezkednek el. Egy rekord 6 mezőből áll:

Start kód	ASCII kettőspont ':'
Bájtok száma	Az adatmezőben levő bájtok száma
Cím	Megadja az adatok kezdőcímét a memóriában 16 biten
Rekord típusa	értéke *00*-tól *05*-ig adható meg
Adat	Adatsor, ami az eszköz memóriájába programozandó
Ellenőrző összeg	Az 1. és a 6. mező kivételével az összes mező 2-es komplemensben vett összege.

2. táblázat: Intel HEX rekordok felépítése [12]

A rekord típusa hatféle lehet. Az adatrekord adatokat és 16 bites címet tartalmaz. A fájl vége rekordból fájlonként csak egyetlen engedélyezett. Adatot nem tartalmaz, és ez a fájl utolsó sora. A maradék négy rekordtípus a 16 bittel megcímezhető 64kByte-ot növeli meg. A legfontosabb ezek közül a kiterjesztett szegmenscím rekord, amivel 20 bites lesz a megcímezhető tartomány, és a kiterjesztett lineáris cím rekord, amivel 32 bitessé válik a címzés.



14. ábra: Példa intel HEX rekordokra [12]

A *processhexFile( )* első lépésben QString formátumúvá konvertálja a beolvasott fájlt, és eltárolja a hosszát. Azután karakterenként kiolvassa a sztringet, és ha start kódot talál, akkor meghívja a *processRowOfhexFile(int*)* függvényt, az aktuális kiolvasott karakter helyét, azaz a start kód helyét átadva paraméterként. Ez a függvény egyszerre

egy rekord feldolgozására képes. Módosítja a *OnehexFile*-hoz tartozó memóriablokk tartalmat vagy kezdőcímet az adott rekord alapján. A *processRowOfhexFile(int*)* végül módosítja az aktuális karakter helyét, miután végzett a rekorddal. Ezt minden rekordra elvégezve az egész hex fájl fel lesz dolgozva.

Fájl loggoló

Diagnosztikai jellegű üzenetek létrehozásában és kijelzésében van szerepe ennek a modulnak. Gyakorlatilag a GUI rétegen kívül minden más modultól tud fogadni hibaüzeneteket, figyelmeztetéseket vagy csak egyszerűen információt. Ezzel az üzenetek súlyosságát 3 szintre bontja. A GUI rétegtől azért nem fogad üzeneteket, mert az saját maga is képes a megfelelő hibaüzenetek kijelzésére, ráadásul a program hibára hajlamos részei, funkciói nem a megjelenítéshez kapcsolódnak.

A GUI réteghez viszont egyoldalúan kapcsolódik, mert üzeneteit egy GUI elem jeleníti meg(egy *reportDialog* objektum). Ehhez két jelet definiál, a *showReport(...)* és *updateReport(...)* jeleket. Ezek a jelek 3 dinamikusan foglalt 2 dimenziós tömböt továbbítanak a *reportDialog* felé. Egy-egy tömb képviseli a különböző súlyosságú üzeneteket. A tömbön belül pedig az egyik dimenzió a különböző eszközökhöz, és a fő szálhoz tartozó üzeneteket képviseli. Egy diagnosztikai üzenet tartalmazza az alábbiakat:

- időbélyeg,
- súlyosság,
- szöveges leírás,
- a kijelzett üzenet helye implicit megmondja, hogy melyik szál küldte azt, de ez akár beírható a szöveges leírásba is.

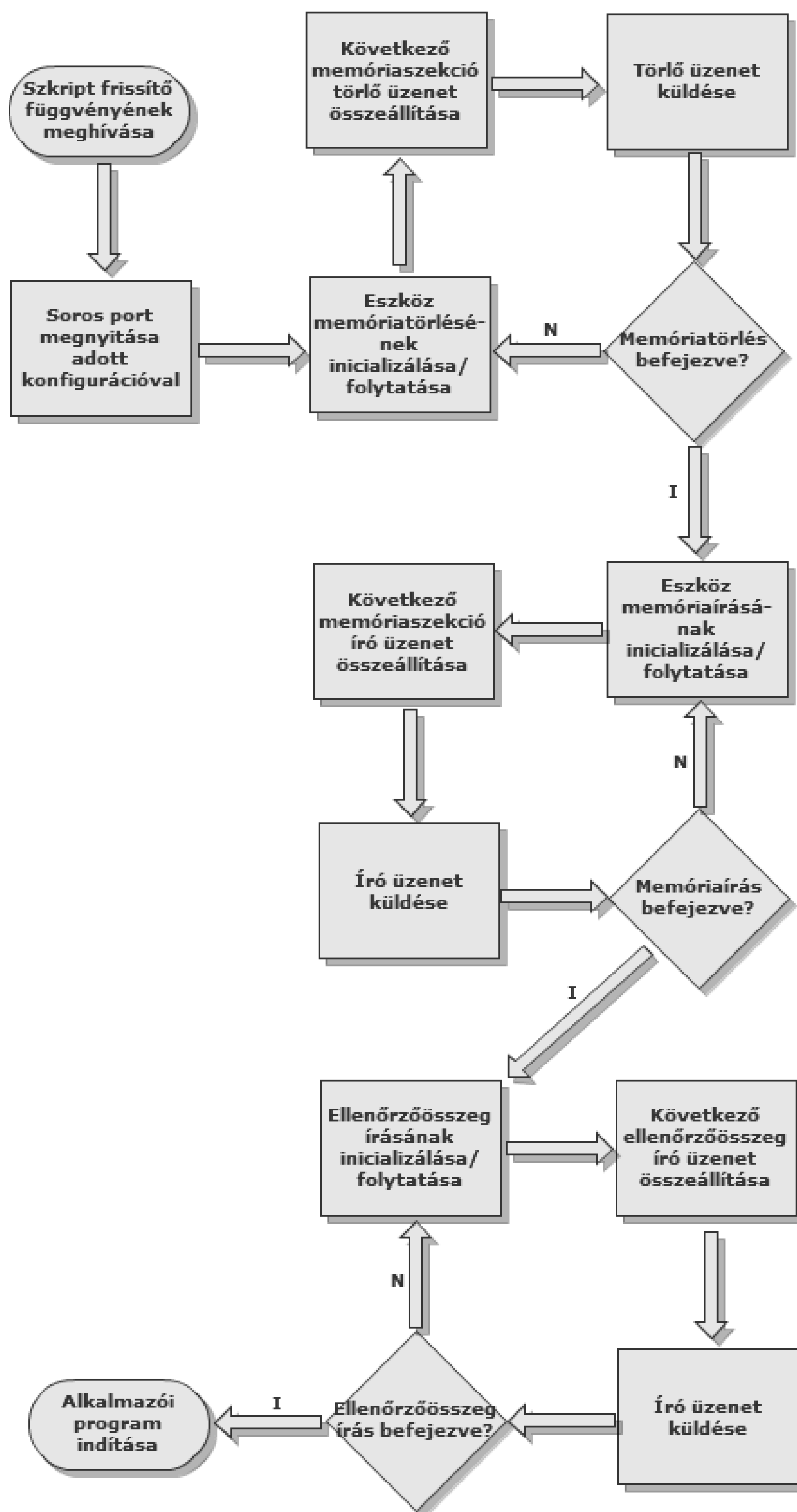
Állapotvezérlő szkript

A Qt-ban rejlő lehetőségeket kifejtő alfejezetben felvázoltam egy C++-belihez hasonló osztálystruktúrát Qt scriptben megvalósítva. Az állapotvezérlő szkript teljes mértékben ilyen struktúrát képvisel.

A szkriptben definiátam egy *main( )* függvényt, és egy tömböt (*maindata*). Ezt a *main( )* függvényt hívom meg a szkript legvégén, miután definiáltam a *maindata* tömböt. A *main( )* függvény visszaadja a *maindata* tömböt. A tömb elemei az egyes frissíteni kívánt eszközökhöz tartozó *Serial_device* szkript objektumok. Ezekre az objektumokra számos tulajdonság van definiálva a szkriptben. A teljesség igénye nélkül felsorolok néhányat:

- hex fájl neve és elérési útja,
- eszközön rendelkezésre álló memória mérete,
- ellenőrzőösszeg elhelyezéséhez szükséges memóriacímek,
- soros port beállításai,
- eszköz címe,
- eszközhöz tartozó szál ID-ja,
- üzenet felépítését meghatározó adatok,
- üzenetkódok (íráshoz, törléshez, olvasáshoz, firmware indításhoz),
- üzenetváltást meghatározó adatok (várakozási idő a válaszig, újrapróbálkozások száma ha nem érkezik válasz az eszköztől).

Természetesen definiálni kell függvényeket is a *Serial_device* objektumokra, hiszen ezek hajtják végre a firmware-frissítést. A tulajdonságok csak beállítják az ehhez szükséges paramétereket. Kulcsfontosságú függvény a *run( )*, ezt indítja el az alkalmazáskezelő minden eszközhöz tartozó szálban.



15. ábra: Folyamatábra a szkript futásáról

A *connect( )* függvény ezután a meghatározott soros port beállítások függvényében kapcsolódik egy eszközhöz. Ezután az *eraseAll( )* függvény letörli az eszközön rendelkezésre álló applikációs memóriában lévő adatokat, helyet teremtve az új firmware-nek. Az *eraseAll( )* az *assembleNextEraseMsg( )* függvényt meghívva gyárt flash törlést kiváltó üzenetet, majd ezt el is küldi. Vár maximum 500 ms-ig (ez tetszés szerint állítható). Ha addig érkezik törlést megerősítő válasz, akkor továbbhalad a következő törlő üzenet küldésére. Fix idejű várakozás helyett az eszköz oldalon megvalósított memóriaművelet befejeztét jelző lekérdezés is használható. Ehhez csak egy speciális lekérdező üzenet küldése szükséges az eszközhöz. A törlő üzenetek küldését addig teszi, amíg az egész applikációs szekciót le nem törölte. Ekkor a *run( )* függvény meghívja a *writeAll( )* függvényt. Ez az *assembleNextWriteMsg( )* függvényt meghívva gyárt flash író üzenetet, majd ezt el is küldi. Ugyancsak vár, és ha érkezik megerősítő válasz, akkor halad tovább a következő író üzenetre. Ezt addig teszi, amíg az egész firmware-t bele nem programozta az eszköz memóriájába.

Ezután néhány *readFlash( )* függvénnyel kiolvasható az eszköz flash tartalma, és az ellenőrizhető a PC segítségével. Ezt helyettesítheti az alkalmazói programra vonatkozó ellenőrzőösszeg beírása az eszköz memóriájába. Ezt a *sendConsistency( )* függvény hajta végre. Ekkor az ellenőrzést az eszköz bootloadere végzi. Az ellenőrzőösszeg eszközbe írása mellett döntöttem, és a szkriptben meghagytam a *readFlash( )* függvényt is, ezzel a felhasználónak meghagytam a választás lehetőségét. Helyes memóriatartalom esetén az új firmware a *startApp( )* függvény lefutásakor elindul. Ugyanis a PC ekkor egy megkülönböztetett üzenetet küld az eszköznek. Ha az eszköz ellenőrzőrutinja (*checkconsistency( )*) nem fedezett fel hibát, akkor elindul a firmware.

3.2.2. Kapcsolati réteg

A kapcsolati réteg elrejti a hoszt-eszköz kommunikáció alacsonyszintű részleteit, mint például üzenetek felépítése és CRC ellenőrzése. Az alkalmazói réteg, azon belül konkrétan az állapotvezérlő szkript és az alkalmazáskezelő felé biztosít interfészeket a következő feladatokkal:

- kapcsolatok inicializálása,
- kapcsolatok megszüntetése,
- adatok küldése eszközökhöz,
- adatok fogadása eszközöktől.

A kapcsolati réteg a következő almodulokból épül fel:

Kapcsolatkezelő

Ez a modul a kapcsolati réteg koordinátora, továbbá ez a modul teremt összeköttetéseket az alkalmazás kezelő és a fájl loggoló felé, és hozza létre a protokoll és eszközelérési rétegeket. Gyakorlatilag a hoszt program *main( )* függvényében a kapcsolati és alkalmazói rétegek példányosítása után a kapcsolat kezelő segítségével össze van kötve az alkalmazói réteg a kapcsolati réteggel:

Jel iránya	Jel szerepe
Kapcsolat kezelőnek	Kapcsolati réteg moduljainak regisztrálása a szkriptben
Alkalmazás kezelőnek	Kapcsolati réteg modulok regisztrálása sikeres
Kapcsolat kezelőnek	Soros vonali kapcsolatok megszüntetése
Fájl loggolónak	Diagnosztikai üzenetek küldése

3. táblázat: Jel összeköttetések az alkalmazói réteg és a kapcsolati réteg között

A kapcsolatkezelő működését érintőlegesen leírtam az előző alfejezetben, de nem árt, ha pontosítok az ott elhangzottakon. Az alkalmazáskezelő a szkript kiértékelése után jelet küld bizonyos moduloknak, köztük a kapcsolat kezelőnek, hogy regisztrálja magát és a kapcsolati réteg más objektumait is a szkriptben. Ekkor a kapcsolatkezelő számára már hozzáférhető a frissíteni kívánt eszközök száma (és ekkor lefut a kezelő *connectToScript( )* slotja). Azonban a szkript *run( )* függvényei még nem indultak el, pedig az ott meghívott *connect( )* függvények kezdeményezik a csatlakozást soros porton egy eszközhöz. Tehát nincs információ arról, hogy milyen beállításokkal szeretne soros portokon kapcsolatot létesíteni a felhasználó. Ilyen feltételek mellett kénytelen voltam azt feltételezni, hogy akár minden eszközhöz különböző soros port lehet később rendelve. Emiatt annyi eszközelérési réteget és protokoll réteget példányosítok, ahány eszköz a szkriptben szerepel, csak a nem használt vagy rosszul beállított eszközelérési rétegeket és protokoll rétegeket nem inicializálom. Létrehozásuk után a protokollrétegek regisztrálva lesznek a szkriptben, majd a kapcsolatkezelővel is ugyanez történik. Végül egy jel lesz küldve (*connecToScriptDone( )*) az alkalmazáskezelőnek, hogy a regisztrálások sikeresen megtörténtek.

Az alkalmazáskezelő a szkriptbe regisztráláson kívül a soros kapcsolatok megszakítására is kérheti a kapcsolatkezelőt. Ekkor meghívódnak az eszközelérési és protokoll réteg destruktorai, amik lekezelik a kapcsolatok bontását.

A fentiek mellett a kapcsolatkezelő küldhet diagnosztikai információt is a fájl loggolónak. Egyúttal megkapja a protokoll és eszközelérési rétegek ilyen jellegű üzeneteit is, és továbbítja azokat a fájl loggolónak. Így nem kell a kapcsolati réteg összes moduljának a jelét kivezetni a fájl loggolóhoz, elég csak egyetlen modul jelét.

A szkript is meg tud hívni egy slotot a kapcsolatkezelőből. Ez az *establishserialConnection( )*. Ez a függvény konfigurál egy adott protokoll réteget, eszközelérési réteget, vagy semelyiket sem. Ha több eszköz soros kapcsolata kompatibilis (ugyanazon a porton ugyanakkora baudrate-tel, paritással stb-vel szeretne kommunikálni), akkor csak az első eszköz esetében lesz új eszközelérési réteg konfigurálva. A többi eszköznél egy már meglévő eszközelérési réteghez lesz egy addig konfigurálatlan protokoll réteg felhasználva. Ha több eszköz soros kapcsolata nem kompatibilis (ugyanazon a porton különböző baudrate-tel, vagy paritással ...stb-vel

szeretne kommunikálni), akkor az első eszközre konfigurálva lesz a protokoll és eszközelérési réteg is, viszont a többire semelyik réteg sem. Ha rossz egy eszköz soros beállítása (nem létező port, nem használható baudrate), akkor semelyik réteg nem lesz konfigurálva ahhoz az eszközhöz.

Protokoll réteg

A protokoll réteg jelenlegi formájában támogatást nyújt Modbus-ASCII üzenetek küldésére és fogadására. Ez azt jelenti, hogy tetszőleges *QByteArray* típusban kapott üzenetet továbbít a hozzá tartozó eszközelérési réteg felé vagy az állapotvezérlő szkript felé. A továbbított üzenet szintén *QByteArray* típusú.

A működés leírását érdemes a Modbus-ASCII részletesebb bemutatásával kezdeni. A Modbus-ASCII a Modbus soros porton alkalmazott változatát takarja, azon belül is az ASCII átviteli módot. Soros porton ugyanis kétféle átviteli módot definiál a Modbus, az ASCII mellett létezik Modbus-RTU is. A Modbus-ASCII a 7 rétegű OSI modell 2. rétegének, az adatkapcsolati rétegnek felel meg. A master-slave protokollok közé tartozik.

Master-slave protokollok esetében egyetlen master létezik, mely különféle utasításokkal vezérli a többi slave eszközt, és feldolgozza a bejövő válaszokat. Slave eszközök általában nem küldenek adatokat a mastertől érkező kérés nélkül, és nem kommunikálnak más slave eszközökkel.

ASCII átviteli módban mindegyik átküldeni kívánt 8 bites bájt 2 hexadecimális karakterként van elküldve. Tehát például a 0x5B bájt a 0x35 = ’5’ és 0x42 = ’B’ ASCII karakterek formájában van továbbítva. Egy karakter az átküldendő bájt 4 bitjét tárolja. A küldő eszköz a Modbus üzenetet keretbe illeszti. Így könnyen felismerhető kezdete és vége van egy üzenetnek, ugyanis a keretet alkotó karakterek a kettőspont (start karakter), és a CR-LF dupla karakter (üzenet vége karakterek). Viszont a hexadecimális kódolás miatt az üzenet tartalmát alkotó karakterek a 0-9 és A-F ASCII karakterek lehetnek. A kezdő karakter után a céleszközt kijelölő 2 karakteres cím mező következik. Ezután a 2 karakteres parancskód mező jön, majd az 504 karakteres adatmező

következik. Ezután egy 2 karakteres ellenőrzőösszeg, az LRC áll. Végül a CR-LF üzenet vége karakterek következnek.

Start	Address	Function	Data	LRC	End
1 char :	2 chars	2 chars	0 up to 2x252 char(s)	2 chars	2 chars CR,LF

16. ábra: Modbus-ASCII üzenet felépítése [13]

A fenti bevezető után világosabbak lesznek a protokoll réteg feladatai:

- Kezdő és végkarakterek beszúrása \ eltávolítása
- ASCII-vé konvertálás \ abból dekódolás
- LRC kiszámolása és beszúrása \ LRC ellenőrzése
- Üzenet átvétele az eszközillesztő rétegtől \ továbbítása az eszközillesztő rétegnek

Most az eszköz irányába történő üzenetküldés folyamatát ismertetem a protokoll réteg szempontjából. Ekkor a szkript meghívja a hozzá tartozó protokoll réteg üzenetküldő slotját (*sendMessage(QByteArray)*). Erre képes, hiszen a kapcsolat kezelő elérhetővé tette a szkriptben az adott protokoll réteget, regisztrálta azt. A *sendMessage(...)* első lépésben kiszámolja az üzenethez tartozó LRC-t. Az LRC a start karakteren kívül az üzenet LRC-ig tartó bájtjaiból jön létre. Ezeket a bájtokat összeadva az eredmény legalsó 8 bitjét kell képezni. Ebből a 8 biten ábrázolható számból ki kell vonni egyet, majd bitenként negálni kell a kapott számot, így áll elő az LRC. Az LRC pedig az üzenet végére van illesztve. Ezután az üzenet ASCII formátumúra történő átalakítása, majd a kezdő és végkarakterek beillesztése történik. Utolsó lépésként meghívjuk az eszközillesztő réteg *sendBytes(QByteArray)* függvényét. Amint a neve is jelzi, ez bájtokat küld a soros portra.

Ha az eszköz felől üzenetet vár az állapotvezérlő szkript, akkor nem csak az üzenetfogadást kell lekezelni, hanem a várakozást is. A szkript ezért a protokoll rétegből meghívhat egy várakozó függvényt (*startWaitingFor(int)*), egy beérkezett

üzenetet visszaadó függvényt (*receivedMessage( )*), és egy függvényt, ami megmondja, hogy érkezett-e érvényes üzenet (*getrxMsgReceived( )*). A várakozást végző függvény a működéséhez segítségül hív egy QTimer objektumot. Első futás alkalmával egy ilyen időzítő objektum jön létre. Emellett még a protokoll réteg szálához tartozó eseménykezelő objektum is példányosul (ennek típusa *QAbstractEventDispatcher*), majd megtörténik az időzítő elindítása. Az a várakozó függvénynek átadott int paraméterben specifikált ezredmásodpercekig fut. Az időzítő leteltét jelzi a *QTimer::isActive( )* függvénye.

Az időzítő indítása után egy *while* ciklus fut addig, amíg le nem telik a várakozás ideje, vagy egy üzenet be nem érkezik. Ebben a ciklusban eseményekre várakozik a program (ezt a *QAbstractEventDispatcher::processEvents( )* függvényével teszi), és ha beérkeznek az események, akkor végre lesznek hajtva. Így a várakozó függvény ténylegesen addig fut, amíg letelik a várakozási idő, vagy egy üzenet érkezik az eszköz felől. Beérkező üzenetet jelez az eszközillesztő réteg egyik jele (*check4EndCharacters( )*). Ekkor a protokoll réteg ellenőrzi, hogy le van-e zárva az eszközillesztő réteg felől érkező bájtsorozat kezdő vagy végkarakterekkel. Ha igen, akkor ASCII formátumról átalakítja a bájtsorozatot szimpla bájtokká, majd ellenőrzi a beérkező üzenet LRC-jének helyességét. Ha az LRC helyes, akkor a beérkező üzenetet *QByteArray* formátumban tárolja (a *receivedMessage( )* ezt a *QByteArray*-t adja vissza), és beállítja az üzenet megérkezését jelző logikai változót. A *getrxMsgReceived( )* ezt a logikai változót adja vissza, így a szkript vagy az időzítő lejártáig, vagy üzenet beérkezéséig vár, majd a logikai változóval kiolvassa, hogy érkezett-e üzenet. Ha igen, akkor azt *QByteArray* típusként megkapja a *receivedMessage( )* függvénytől.

Persze itt felmerül a kérdés, hogy mégis hogyan lehetséges, hogy a szkript *QByteArray* típusokkal is tud dolgozni, hiszen a Qt Script ECMAScript-re épül, és az nem ismer ilyet. A *QByteArray* Qt Scriptbe "importálását" a következő szekció írja le.

ByteArray osztály implementálása Qt Script-ben

Tetszés szerinti szkript osztály implementálható Qt scriptben egy egyszerű API-val (Application Programming Interface), a *QScripClass* API-val. Mivel a hoszt program bájtsorozatokat küld és fogad soros porton, ezért előnyösnek tűnt, hogy implementálva

legyen a *QByteArray* osztály a szkriptben, így az eltárolt bájtok nem foglalnak nagyobb helyet, mint amire szükségük van. Ráadásul nem igényel annyi számítást a küldés vagy fogadás, mert nem kell időigényes konverziót a szkript és Qt típusok között végrehajtani. A most következő leírás egy ilyen implementáció megvalósítását mutatja be, és a forrása [14].

Saját szkript osztályt a *QScriptClass* osztályból kell származtatni. Annak virtuális metódusait pedig felül kell definiálni. Mindemellett biztosítandó egy konstruktor az új szkript osztályra, melyet a szkriptértelmező (*QScriptEngine* példány) felismer:

```
// konstruktor definiálása, származtatás QScriptclass-ból
      // konverziós függvények regisztrálása
      // tömbmérethez tartozó referencia (handle) beállítása
      // ByteArray prototípus inicializálása
      // értelmező belső konstruktorának ByteArray konstruktorra
      // állítása
```

Kétirányú konverziós függvényekre azért van szükség, hogy a C++ *QByteArray* objektumok, és a Qt Script *ByteArray* objektumok fennakadások nélkül tudjanak a C++ oldalról a szkript oldalra, és fordítva mozogni. Tehát például egy *QByteArray* paraméterű C++ slot meghívása egy ByteArray objektummal így már nem okoz gondot. Tömbmérethez tartozó referencia beállítására azért van szükség, mert így a tömbméret tulajdonságot gyorsabban ki lehet olvasni. A ByteArray prototípus inicializálást azért kell végrehajtani, hogy a Qt Script prototype tulajdonsága C++-ban implementálhatóvá váljon. Az értelmező belső konstruktorát pedig azért kell a ByteArray konstruktorra állítani, hogy az ténylegesen egy új ByteArray példányt létre is tudjon hozni.

Célszerű definiálni egy új, szkriptbeli ByteArray példányosításáért felelős függvényt is. Ez nem része a *QScripClass* API-nak. Ez egy C++ oldalon létező objektumból *QScriptValue* objektumot hoz létre, így könnyen generálható egy *ByteArray* objektum egy *QByteArray* objektumból. A függvény egyszerre funkcionál "konstruktorként", és hajt végre konverziót:

```
// konverziós konstruktor, QByteArray paraméterrel
      // memória allokálása egy új QScriptValue objektumnak
      // paraméter konvertálása QScriptValue objektummá
      // visszatérés a QScriptValue objektummal
```

Memória allokálásnál lefut a C++ oldal prototípus implementációja, amely gondoskodik arról, hogy az új objektum prototípusa *ByteArray* legyen. A fentieken kívül kulcsfontosságú szerepet töltenek be bizonyos felüldefiniált metódusok, amelyek az API részét képezik, és a következő feladatokat látják el:

- *construct( )* - ByteArray konstruktor a szkript oldalon
- *queryProperty( )* - szkriptbeli tulajdonság hozzáférhetőségét jelzi
- *property( )* - szkriptbeli tulajdonság értékével tér vissza
- *setProperty( )* - szkriptbeli tulajdonságot módosít
- *propertyFlags( )* - tömbméret tulajdonságot védi a törléstől

A fentieken kívül szükség van a C++ oldal prototípus implementációjára, amelyet már említettem korábban. Ez *ByteArray* kezeléshez tartozó függvényeket slotok formájában valósít meg. Részletesebben nem ismertetem, inkább a *ByteArray* szkript osztály használatát mutatom be ezután.

Gyakorlatilag elegendő a megfelelő deklarációk include-olása után egy *ByteArray* osztályt deklarálni. Majd a szkriptet kiértékelő értelmezőt példányosítani, és arra egy ByteArray nevű globális tulajdonságot definiálni a *ByteArray* osztály konstruktorával. Ezután *ByteArray* hozható létre a szkript oldalon a C++ oldalról a konverziós konstruktorral és a *setProperty( )* függvényhívással. A szkript oldalon ugyanerre használható a *new ByteArray( )*. A szkript oldalról *ByteArray* paraméterekkel meghívhatók a *QByteArray*-t paraméterként feldolgozó C++ függvények. Egyúttal a *QByteArray* visszatérési értékű C++ függvények a szkriptből meghívva *ByteArray* típussal térnek vissza.

Eszközillesztő réteg

A kapcsolatkezelő által generált eszközillesztő rétegek felelősek a soros port közvetlen vezérléséért. A protokoll rétegeken keresztül kötik össze az egyes eszközöket az állapotvezérlő szkript megfelelő firmware-frissítő függvényeivel. Megvalósításuknál a Qt-ba importálható QextSerialPort függvénykönyvtárra támaszkodtam. A

függvénykönyvtár használatához az 1.2 verzió óta elegendő a forrásfájlok letöltése után a projekt fájlban jelölni az elérési utat egy include-dal. A réteg konkrét feladatai közé tartoznak az alábbiak:

- port konfigurálása,
- port megnyitása,
- port bezárása,
- bájtok küldése,
- bájtok fogadása,
- küldés/fogadás jelzése a protokoll rétegnek.

A port megnyitása előtt be kell állítani a port jellemzőit, konfigurálni kell. E jellemzők közé tartozik a port neve, bitrátája, stop bitek száma, adatbitek száma, paritása és üzemmódja. Az üzemmód lehet szinkron és aszinkron. Az eszközillesztő réteg aszinkron üzemmódot használ mindig. A konfigurálás után már megnyitható az adott soros port.

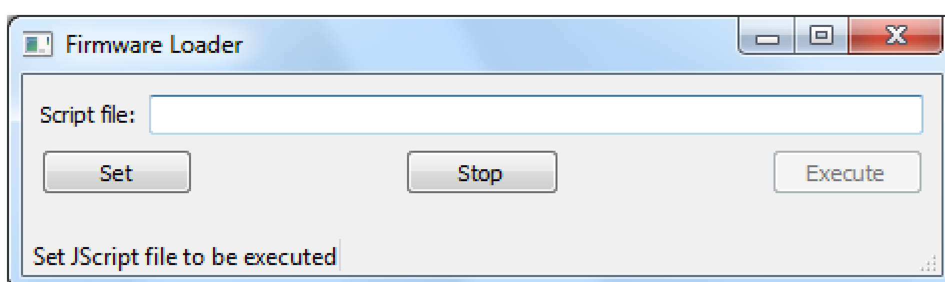
Ha a megnyitás sikeres, akkor bájtok fogadásáról egy signal tájékoztat. Ezt a jelet egy slotban kezeli le a réteg. Ebben először a fogadott bájtok hosszát olvassa ki, majd lefoglal egy ugyanakkora hosszúságú lokális *QByteArray* puffert. A pufferbe beolvassa a kapott bájtokat, és a puffer tartalmát illeszti egy *QByteArray* adattag végére. Az adattag tartalmát a réteg törli, ha a protokoll réteg értesíti arról, hogy érvényes üzenetet kapott, vagy ha megtelt a pufferhez maximálisan rendelhető 10 MB memória.

Bájtok küldését a port *write( )* függvényével lehet megoldani. Paraméternek meg lehet adni az elküldendő bájtokat reprezentáló *QByteArray*t, és a függvény a küldés sikerét jelző logikai változóval tér vissza.

A kapcsolati rétegnek ezzel minden fontos funkcióját ismertettem. Egyúttal azok megvalósítását is leírtam. Most a GUI réteg tárgyalására térek ki.

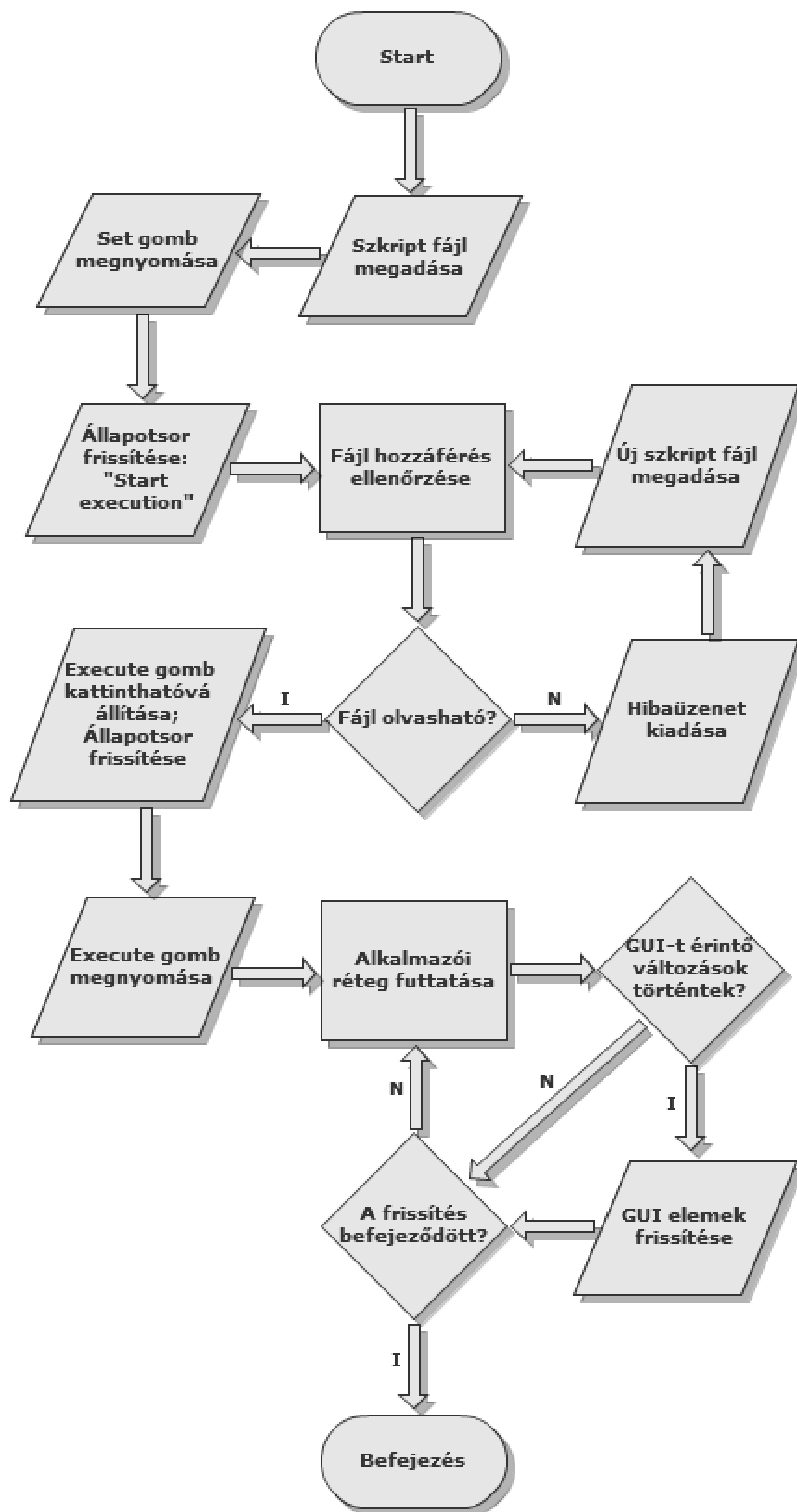
3.2.3. GUI réteg

A hoszt program leglátványosabb része értelemszerűen a GUI, azonban előre kell bocsátanom, hogy egy egyszerű GUI-ról van szó. Hiszen a firmware-frissítés beállításait a szkript fájlban kell rögzíteni, így egy összetett GUI-ra nincs is szükség a beállítások terén. A frissítés állapotával kapcsolatos megjelenítésnek viszont van értelme, és ilyen funkcióval bíró GUI elemek implementációjára helyeztem a hangsúlyt.



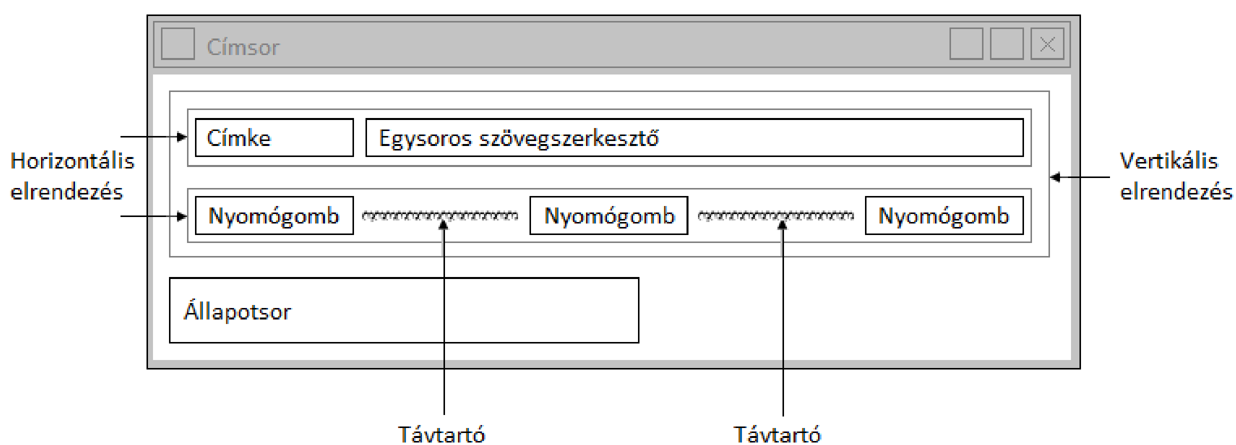
17. ábra: Hoszt program GUI

Az indítás után megjelenő GUI látható a fenti ábrán. Ebben található egy egysoros szövegszerkesztő, előtte pedig egy címke, utalva arra, hogy szkript fájlt kell megadni a szerkesztőben. Ezalatt találhatók gombok. Az első gomb (Set) a megadott fájlnév és elérési útvonal nyugtázására szolgál. Ha helyes, vagyis létezik olyan fájl, akkor az Execute gomb megnyomhatóvá válik. Ha nincs ilyen fájl, akkor megjelenik egy hibaüzenet, és az Execute gomb megmarad az alapbeállításának megfelelő állapotban.



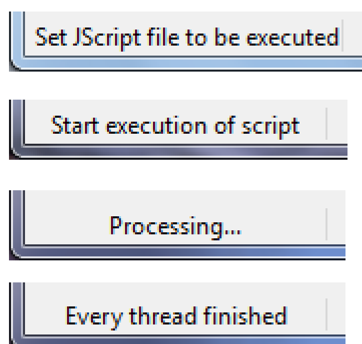
18. ábra: Folyamatábra a GUI működéséről

A Stop gombbal a firmware-frissítés megszakíthatóvá válik. Ha a felhasználó úgy ítéli meg, hogy még sincs szüksége a frissítése, akkor közbeavatkozhat. Ez különösen akkor bizonyult hasznosnak, ha az adott eszköz firmware-t tartalmazó memóriája lassan törölhető flash memória volt, és viszonylag nagy része törölve lett. A flash kijelölt kis részét törölve pár másodperc alatt lefuthat a firmware-frissítés, és ekkor a Stop gombnak gyakorlatilag nincs jelentősége.



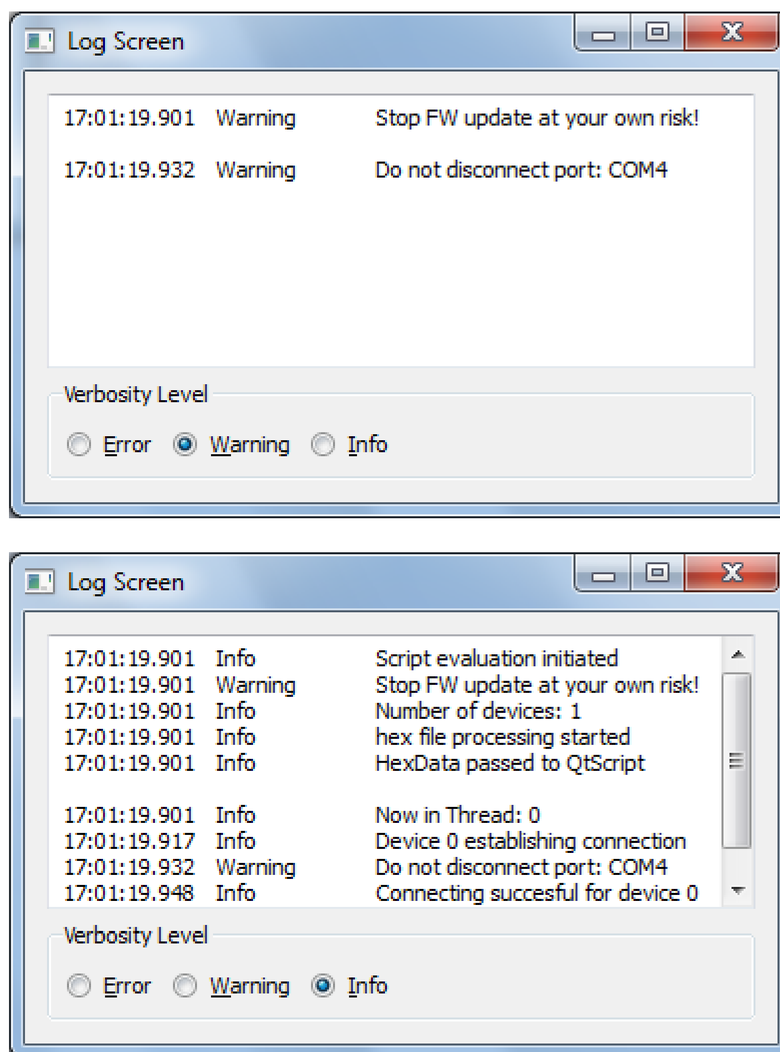
19. ábra: GUI elemek és elrendezésük

Az eddig taglalt GUI elemek a központi GUI részhez, egy ún. *CentralWidget*hez tartoznak, azon belül pedig különféle elrendezésben (layout) vannak elhelyezve. Ez látható a fenti ábrán.



20. ábra: Az állapotsor változása a frissítés során

A központi részen túl a GUI alján egy állapotsor található, mely alapvető információkat nyújt a program működésével kapcsolatban. Négy állapotba kerülhet jelenleg. Állapotváltozást idéz elő egy szkript fájl beolvasása, futtatása és futásának befejeződése. Azért nem implementáltam több állapotot, mert a hibaüzenettel szolgáló felugró ablakok mellett még az eddig nem említett dialógusablak is szolgáltat információkat.



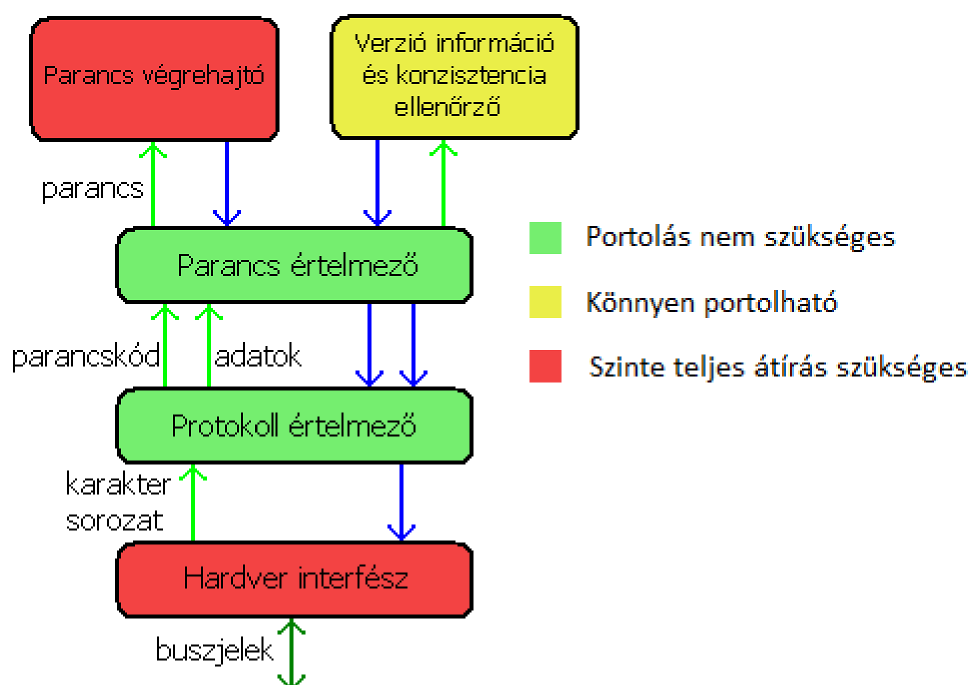
21. ábra: Dialógusablak üzenetek megjelenítésére

Az ábrán látható dialógusablak 3 szintű részletességgel tud megjeleníteni üzeneteket, amelyeket a hoszt program C++ részéből vagy a szkriptből kaphat. Alapbeállításként a hibákat jeleníti meg, de át lehet kapcsolni a figyelmeztetés üzemmódra és az összes

információt megjelenítő üzemmódra. Figyelmeztetés üzemmódban a hibák és a figyelmeztetések is megjelennek. Egy üzenet egy időbélyegből, súlyosságot jelző besorolásból, és az üzenet leírásából áll. Az üres sorral nem elválasztott üzenetsorok egy szálhoz tartoznak. A legelső ilyen sorozat a program főszálához tartozik. Az utána következő sorozatok pedig az egyre növekvő sorszámú szálakhoz tartoznak.

4. Eszköz oldali szoftverkomponensek

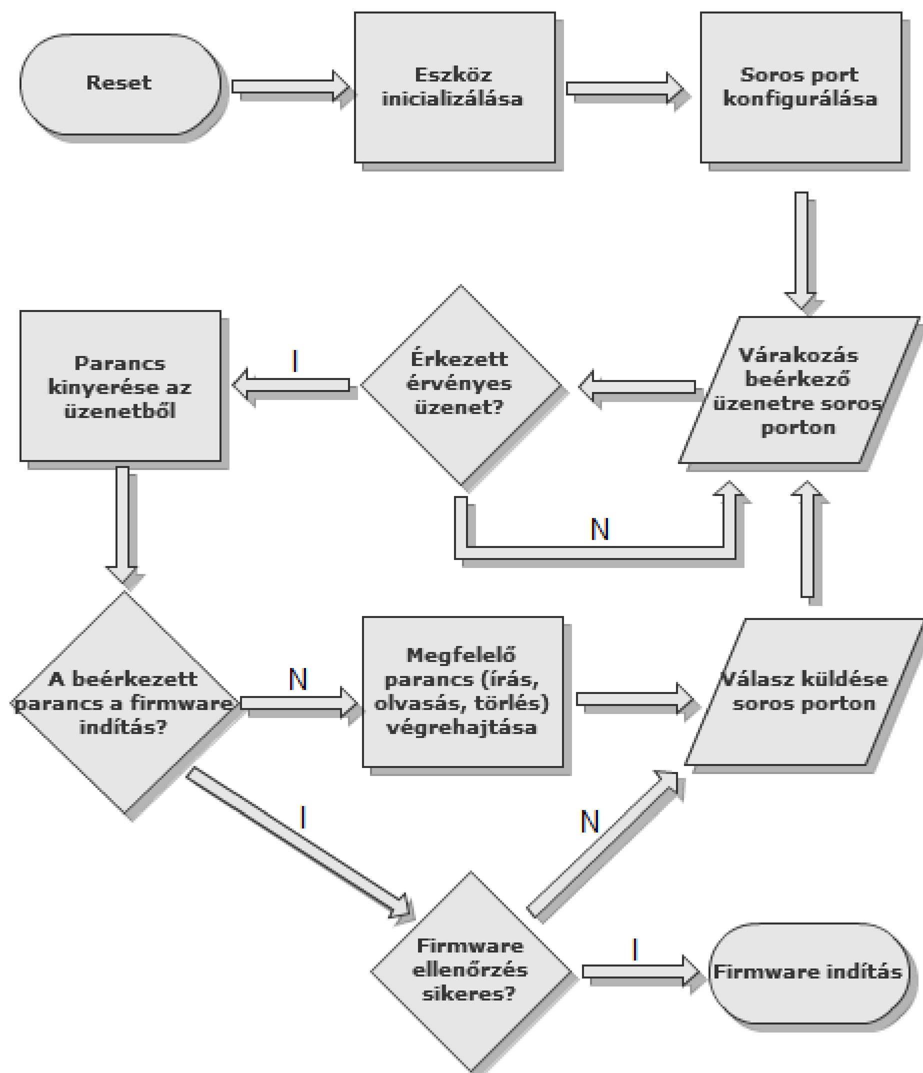
A teljes rendszerterv a hoszt oldali program mellett az eszköz oldalon is megkívánja a firmware-frissítés támogatását. Erre a támogatásra alkalmas egy bootloader, amely várja a hoszttól érkező üzeneteket, és az alapján indítja el az eszközön futó firmware-t, vagy frissíti azt. A bootloaderrel szemben támasztott követelmény, hogy lehetséges legyen a hatékony kód újrafelhasználása. Ezen felül viszonylag kis helyigényűnek kell lennie, és csak kevés erőforrást szabad felhasználnia (így kibővítve a bootloader futtatására alkalmas mikrokontrollerek, DSP-k számát).



22. ábra: A bootloader moduljai portolás szempontjából

A kód újrafelhasználását leginkább az könnyíti, ha a bootloader egyszerűen portolható más eszközökre. Emiatt a booloader modulok határait a funkciók mellett az adott modul portolhatósága is befolyásolta. Így öt modul lett elkülönítve, melyek a 22. ábrán

láthatók. Ezek a modulok jelentősen különböző energiabefektetéssel portolhatóak, mint ahogy az az ezután következő alfejezetekből kiderül.



23. ábra: A bootloader futásának folyamatábrája

Az egyes modulok ismertetése előtt a működés alapjait is leírom nagy vonalakban. A bootloader az adott eszköz nem felejtő memóriájában foglal helyet, ezen belül is egy olyan memóriablokkban, hogy reset után közvetlenül elindulhasson. Munkám során két eszközön valósítottam meg az eszköz oldali szoftvert:

- ATmega 128 mikrokontroller,
- Blackfin 537 DSP.

Az általam programozott ATmega128 mikrokontrolleren egy ilyen memóriablokk az eszköz on-chip flash memóriájának végén helyezkedett el. A Blackfin 537 DSP-t tartalmazó kártyán pedig a DSP-hez csatlakozó flash memória első részét jelöltem ki a bootloader számára. Az eszköz resetelése után a bootloader üzeneteket vár a hoszt programtól, és azok alapján felülírja a firmware-t, mely a memóriájában foglal helyet. A bootloader maga védve van a felülírástól. Ha sikerült a kívánt programot beírni, akkor a hoszt utasítja az eszközt, hogy indulhat a firmware. Ekkor be lesz állítva, hogy reset után a processzor a firmware kezdőcímére ugorjon, és onnan kezdje a végrehajtást. Majd egy reset kerül kiadásra. Ezzel elindul a firmware. Ez a folyamat a firmware-frissítés, tömören, amelynek folyamatábrája a 23. ábrán látható.

A frissítés nagy vonalakban történő bemutatása után ismertetem a futtatást végző hardver eszközöket, ugyanis az eszköz oldali szoftver komponensek működése erősen kötődik az eszközök hardveréhez, hiszen a frissítés gyorsaságát a nem felejtő memória hozzáférési ideje, a választott kommunikáció vagy a firmware indítás fajtáját a hardverben biztosított lehetőségek szabják meg. Beágyazott rendszerek esetén egyébként is nagy befolyása van a választott hardvernek a rajta futó szoftverre.

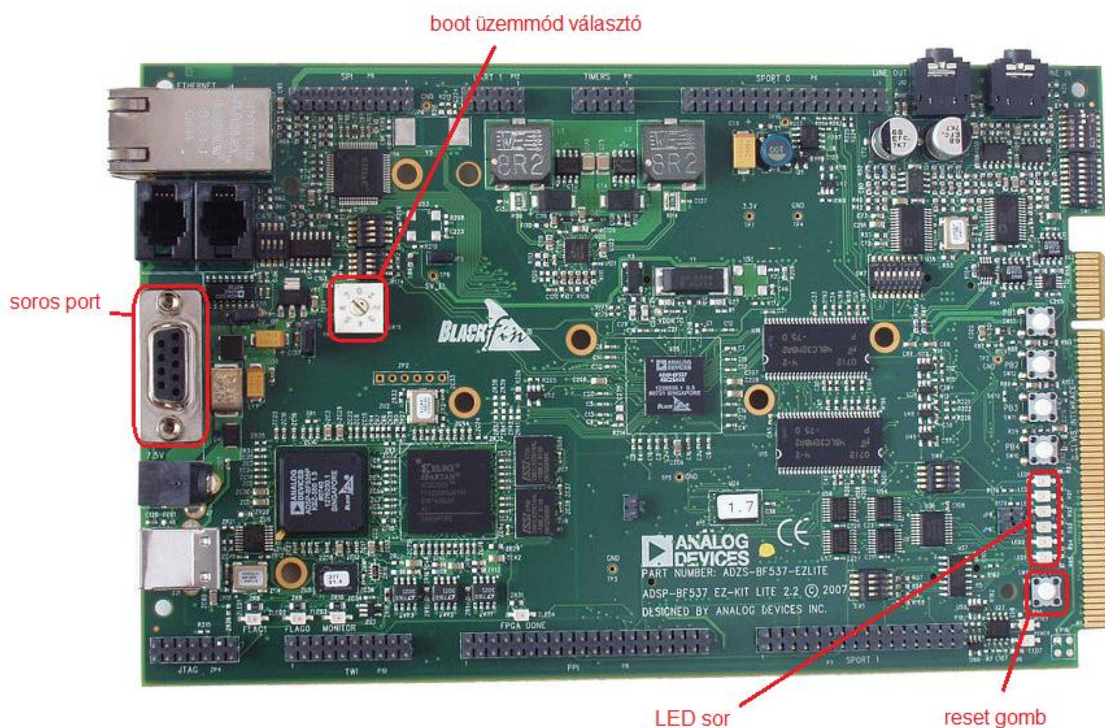
4.1 Futtatást végző hardver eszközök

Firmware-frissítésre leginkább a különféle mikroszámítógépeknek (mikrokontroller, DSP) lehet szüksége. Beágyazott rendszerek közé tartozik még az FPGA, viszont az egy ún. konfigurációs memóriából szerzi meg a belső huzalozását, és így (párhuzamos) működését leíró adatokat. A konfigurációs memória pedig a teljes működését leírja. Ráadásul boot és alkalmazói memóriaszekciók közötti ugrás nem értelmezhető az FPGA párhuzamos utasításvégrehajtási módja esetén. Ezek miatt a firmware-frissítés FPGA-n roppant komoly nehézségekbe ütközik, feltéve, hogy azt magán az FPGA-n futó bootloader végezze. Ezért a firmware-frissítés eszközeként egy DSP-t és egy mikrokontrollert választottam. Ezeknél a programvégrehajtás memóriából történik (nem

úgy, mint az FPGA-nál, ahol csak konfigurációs adatot tárol a memória). Így lehetővé válik a bootloader és firmware elkülönítése.

4.1.1 Blackfin 537 DSP

Egy Analog Devices gyártmányú, 16 bites ADSP-BF537 processzort választottam a DSP-k közül, ugyanis a tanszékemen ez az eszköz állt rendelkezésre a fejlesztéshez. Az architektúrája azt eredményezi, hogy DSP-k és mikrokontrollerek jellemzőit is magán viseli. 600 MHz-es mag frekvenciája van, amihez 120 MHz-es rendszer frekvencia társul. Az utasításait gyors hozzáférésű memóriában tudja eltárolni. Ez gyors végrehajtást eredményez, ez a DSP-k egyik jellemzője. Ugyanakkor számos perifériával rendelkezik pl.: CAN interfész, UART, 48 általános célú I/O, TWI controller... stb. Ez a mikrokontrollerekhez teszi hasonlatossá.



24. ábra: A DSP kártyán használt I/O eszközök

A DSP-t a fejlesztői kártyáján használtam (EZ-Kit-Lite), lásd 24. ábra. Itt egy különálló aszinkron flash memóriát tudott kezelni. Ezen tároltam a bootloadert és a frissített firmware-t is. A fejlesztés során a következőket tapasztaltam:

- Mivel a flash memória hozzáférési idői nagyok, így a firmware-frissítés is sok időbe kerül a DSP-n, annak ellenére hogy a DSP gyors.
- Bonyolult inicializálás, amiben érdemes segítségül hívni a processzor Boot-ROM-ját.
- Bonyolult architektúra, körülményes használat.

A fejlesztéshez az Analog Devices VisualDSP++ 5.0 fejlesztőrendszerét használtam. A bootloader alapvető (memóriakezelésen kívüli) funkcióit valósítottam meg először (amikor a mikrokontrolleres kódot portoltam DSP-re). Ezután a memóriakezelést végző részt is portoltam. Majd segítségül kellett hívnom a fejlesztőkörnyezet flash programozó tool-ját. Ez speciálisan egy bizonyos fejlesztői kártyához készült programot tölt be a DSP-be. Erre azért van szükség, mert kártyánként különbözhet a DSP-hez kapcsolt memória típusa vagy a kivezetések összekapcsolása. Ezzel a kártya flash memóriájába le tudtam tárolni a bootloadert. A flash programozása után be kellett állítani a kártyán és a DSP-n a bootolási folyamatot, és resetelni kellett a DSP-t. Ezután tudtam tesztelni a működést.

Az éppen futó programot (bootloader vagy firmware) a LED-ek jelezték. A 2. LED folyamatos világítása a bootloader futását jelentette (ez persze a soros porton küldött üzenetekből is kiderült). A 2. LED kikapcsolt állapota és a 4. LED villogása pedig a teszt firmwareben volt lekódolva, így annak futását jelezte.

4.1.2 ATmega 128 mikrokontroller

A mikrokontrollerek közül egy ATmega 128-at használtam fel.

Jellemzői:

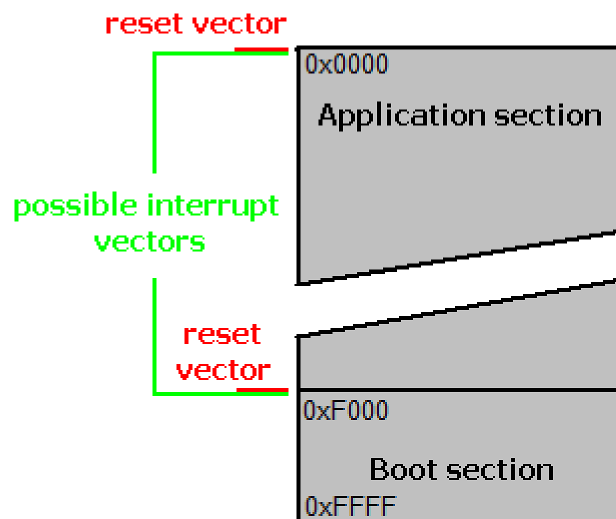
- 8 bites utasítások,

- 128 kByte integrált flash memória,
- 16 MHz maximális működési frekvencia,
- számos periféria: 2db UART, SPI, TWI, 4db időzítő ... stb.

Az ATmega fejlesztői kártyája egyedi gyártású, 3db LED-et, és 1 reset gombot hasznosítottam róla a teszteléshez a soros port vonalai mellett. A fejlesztés során a következőket tapasztaltam:

- Egyszerű architektúra, egyszerű használat.
- Az eszköz inicializálása nem igényel nagy tudást.
- A firmware-frissítés viszonylag gyorsan le tud futni az integrált flash gyors törlési ciklusa és a minimális törölhető adatméret kis volta miatt.

A fejlesztést a mikrokontrolleren AVR Studio 4 fejlesztői környezettel végeztem. Ez támogatást biztosított a bootloader konfigurálásánál, ugyanis a flash boot szekciója a memória végén helyezkedik el ennél az eszköznél:



25. ábra: ATmega 128 flash memória térkép

A projekt beállításainál ki lehetett választani a fejlesztőkörnyezetben, hogy melyik memóriacímtől kezdődjön a feltöltött program, és azt is, hogy mekkora legyen a boot szekció mérete pontosan.

A bootloader futását ennél a kártyánál szintén egy folyamatosan világító LED jelezte. A firmwareben pedig ennél is egy LED villogását és egy másik folyamatos világítását programoztam le, így a firmware és bootloader futása jól elkülöníthetővé vált.

A hardver ismertetése után rátérek az eszköz oldali bootloader moduljainak az ismertetésére.

4.2 Nem portolandó részek

A nem portolandó részek közé olyan modulok kerülhettek, melyek nincsenek közvetlen kapcsolatban perifériákkal vagy egyéb hardver erőforrásokkal. Két ilyen modul van: a protokollértelmező és a parancsértelmező. A protokollértelmezőt a hardver interfész választja el a perifériáktól. A parancsértelmezőt pedig a parancsvégrehajtó mentesíti az eszköz nem felejtő memóriájának és a resetelésnek a kezelésétől.

4.2.1 Protokollértelmező

A hardver interfész által kapott bájtokat, pontosabban karaktereket dolgozza fel ha fogadásról van szó. Küldésnél pedig a hardver interfész felé továbbít karaktereket a parancs értelmező felől. A Modbus-ASCII protokollt támogatja.

A beérkező karakterekből üzenetek azonosítására, részekre bontására, ellenőrzőösszeg számítására, és a parancsvégrehajtáshoz szükséges adatok kinyerésére képes. Meg tudja hívni a parancsértelmezőt, és az attól érkező üzenetdarabokat össze tudja rakni egy teljes üzenetté. Ez a komponens csak processzorfüggetlen C kódból áll.

A protokollértelmező egy üzenetet tartalmazó *struct* típust ad vissza a parancsértelmezőnek. A visszaadott *struct* típus magában foglalja:

- az eszköz címét, amelynek az üzenetet címezték,
- a végrehajtandó parancs kódját,
- memóriacímet és a hozzá tartozó adatokat, ha a parancs ilyet definiál.

Küldésnél a protokollértelmező nem csak a fenti adatokat képező karaktereket adja át a hardver interfésznek elküldésre, hanem az ellenőrzőösszeget, illetve a kezdő és végkaraktereket. A működés pszeudokódja a következő:

```
Protokollértelmező ( )
{
	// várakozás beérkező bájtokra
	// beérkezett bájtok karakterekké alakítása
	// üzenetrészek összerakása
	// hibaellenőrzés
	// parancs értelmező hívása
	// adatátvitel inicializálása, ha van kimenő üzenet
	// átküldendő bájtok karakterré alakítása
	// küldés
}
```

4.2.2 Parancsértelmező

Ez a modul a beérkező üzenetet paranccsá alakítja. Ezt az üzenetben található parancskód és adatok alapján teszi. A parancsvégrehajtóhoz és értelmezőhöz érdemes egy olyan struktúrát definiálni, amely a parancskódokat és a parancsokat tartalmazza. Ekkor az értelmezőnek elég összehasonlítania az általa argumentumként kapott parancskódot a létező parancskódokkal, majd meghívnia a parancsvégrehajtó megfelelő függvényét. A parancsértelmező gerincét egy ciklus adja, az értelmező maga pedig processzorfüggetlen C kód. A fent említett típus a következő elemeket tartalmazza:

- függvény pointer írást végző függvényre,
- írás parancs kódja,

- függvény pointer olvasást végző függvényre,
- olvasás parancs kódja,
- függvény pointer törlést végző függvényre,
- törlés parancs kódja,
- függvény pointer memóriaműveletre várakozást végző függvényre,
- várakozás parancs kódja.

Azért van szükség függvény pointerekre, hogy ha esetleg nem csak egy nem felejtő memóriája van az eszköznek, akkor a lehetőség adott legyen a különböző memóriák firmware-frissítésére. Ha a parancsértelmezőt a protokollértelmező meghívja, akkor az végignézi az összes parancskódot, és ha egyezést talál, akkor végrehajtja az ahhoz rendelt parancsot.

4.2.3 Verzióinformáció és konzisztenciaellenőrző

A fő funkciók mellett szükség van még néhányra kiegészítő jelleggel. Ezek a frissítés hatékony megvalósítását szolgálják. A feltöltött program helyességét például érdemes ellenőrizni újraindítás előtt. Ezt konzisztenciaellenőrzésnek hívják.

<table>
<tr><td>...</td><td rowspan="2">Boot flash</td></tr>
<tr><td>Bootloader</td></tr>
<tr><td>x, y, *konzisztenciaregiszer</td><td rowspan="5">Applikáció flash</td></tr>
<tr><td>...</td></tr>
<tr><td>konzisztenciaregiszter [1,x+y] dimenzióval</td></tr>
<tr><td>...</td></tr>
<tr><td>Program [x,y] dimenzióval</td></tr>
</table>

4. táblázat: A konzisztencia változónak helye a memóriában

A memória megfelelő elrendezése kulcsfontosságú a konzisztencia eléréséhez. Ez csak az applikáció memóriára vonatkozik. Ennek írását a PC vezérli. A beágyazott rendszer szoftverének feladata, hogy a memória tartalma alapján kiszámolja a konzisztenciát ellenőrző összegeket, és azt összehasonlítsa a PC által küldöttel.

A konzisztenciaregiszter két vektort tartalmaz, ezek az ellenőrzőösszegek, amelyek a programkódhoz tartoznak. Ha a PC által küldött és a beágyazott rendszer által számított ellenőrzőösszegek nem egyeznek, akkor az application firmware nem fog elindulni.

0x00		*fl_cons_x – 1*	*consistency_y [fl_cons_y]*
fl_cons_x		*2*fl_cons_x – 1*	
	....		
*(fl_cons_y-1)*fl_cons_x*		*(fl_cons_y-1)*fl_cons_x -1*	
consistency_x[fl_cons_x]			

26. ábra: Ellenőrzőösszegek számítása

A firmware szürkével van ábrázolva, és előjel nélküli char formátumú. Az ellenőrzőösszeget a következő képletekkel lehet kiszámítani:

consistency_y[y] = sum(flash_data[fl_cons_x*y + i]) // i=0..(fl_cons_x-1)

consistency_x[x] = sum(flash_data[x + j*fl_cons_x]) // j=0..(fl_cons_y-1)

A konzisztenciaellenőrzés implementálásánál figyelembe kellett venni, hogy a mikrokontrollernek nincs elég memóriája ahhoz, hogy egyszerűen beolvassa a teljes firmware-t és az alapján számítsa ki az ellenőrzőösszeget. A memória beolvasásához szükséges idő közelítőleg: 2 Mbit / 10 MHz => 0.2 másodperc. Ennyi késleltetés az alkalmazás indulásakor megengedhető, ez egy elég gyors módszer.

Verzióinformációt is érdemes lehet tárolni a flash memóriában az ellenőrzőösszeg mellett. Így a firmware és a bootloader verziója, létrehozásának dátuma és pontos ideje

is eltárolhatóvá válik a hardverre vonatkozó információ (revízió száma, típus ... stb.) mellett.

4.3 Portolt részek

A portolt részek közé olyan modulok kerülnek, melyeknek valamilyen szinten nem eszköztől független az implementációja. Ide tartozik a parancsvégrehajtó és a hardver interfész. A hardver interfész időzítőket és USART-ot vezérel. A parancsvégrehajtó pedig az eszköz nem felejtő memóriáját kezeli, és a megfelelő resetelésről gondoskodik. E két modul átírása komoly feladat portoláskor. A portolt részek közé tartozik egy verzióinformációt kezelő és memória helyességét (konzisztenciát) ellenőrző modul is. Ennek header fájlja tartalmaz eszközspecifikus memória címet. A portolás ennél a modulnál viszonylag egyszerű.

4.3.1 Hardver interfész

Processzorfüggő részek alkotják. Az üzenetküldés legalacsonyabb szintjének implementálása a fő feladata. A mikrokontroller vagy DSP megfelelő regisztereiből való olvasás és írás történik függvényei segítségével.

Megvalósításához szükséges funkciók, függvények:

kártya inicializálás – nem minden eszköznél szükséges lépés

Mikrokontroller esetén

használat: main függvény elején

- felhasznált GPIO (ez esetben 1 LED-hez tartozó) beállítása, nem használtak kikapcsolása,

- az alkalmazói firmware indításhoz szükséges interrupt vektor változtatás engedélyezése, konfigurálása.

DSP esetén

használat: Init_Flags és Init_PLL paraméter és visszatérési érték nélküli függvények felhívása

- felhasznált GPIO (ez esetben 1 LED-hez tartozó) beállítása, nem használtak kikapcsolása,
- PLL frekvenciaosztó beállítása,
- PLL-hez tartozó feszültség beállítása.

kommunikáció inicializálása (bitráta)

Mikrokontroller esetén

használat: main függvény elején

- interruptok globális kikapcsolása,
- megfelelő GPIO kivezetések (pinek) beállítása inputként ill. outputként,
- bitráta regiszter beállítása a bemeneti paraméter alapján,
- a kommunikációt végző UART regisztereinek konfigurálása (stopbitek, paritás... stb.),
- interruptok visszakapcsolása.

DSP esetén

használat: main függvény elején

- UART órajel engedélyezése,
- UART konfigurálás (8 adatbit, stopbitek, paritás),
- bemenő paraméter és PLL vezérlő regiszter alapján UART bitrátát meghatározó regiszter beállítása,
- megszakítások engedélyezése az UART-ra,
- megszakítás maszk kikapcsolása az UART-ra,

- UART fogadó és küldő megszakítások megszakítási csoporthoz rendelése, prioritás megadása,
- UART portok és megszakítás kezelő rutinok párosítása.

bájt küldés (adat)

Mikrokontroller esetén

használat: protokollértelmezőben felhívva

- küldő regiszterbe beírja az elküldendő bájtot, ez pedig küldő interruptot generál.

DSP esetén

használat: protokollértelmezőben felhívva

az elküldendő bájtot a küldő FIFO-ba teszi

- UART regiszter olvasása annak megállapítására, hogy van-e hátra még előző küldés,
- ha van, akkor várakozás,
- ha nincs, akkor küldő regiszterbe írással interrupt generálás.

bájt fogadás ( )

Mikrokontroller esetén

használat: protokollértelmezőben felhívva

- a fogadott bájtot a fogadó FIFO-ból kiolvassa.

DSP esetén

használat: protokollértelmezőben felhívva

- a fogadott bájtot a fogadó FIFO-ból kiolvassa.

küldő FIFO állapota( ) – küldő FIFO fel nem használt helyeinek számával tér vissza

Mikrokontroller esetén

használat: protokollértelmezőben felhívva

- visszaadja, hogy várakozik-e küldésre bájt.

DSP esetén

használat: protokollértelmezőben felhívva

- kiszámolja a küldő FIFO pointerei alapján a FIFO fel nem használt helyeinek a számát.

fogadó FIFO állapota( ) – fogadó FIFO olvasatlan karaktereinek számával tér vissza

Mikrokontroller és DSP esetén

használat: protokollértelmezőben felhívva

- kiszámolja a fogadó FIFO pointerei alapján a FIFO kiolvasatlan karaktereinek számát.

hibás üzenet ( ) – 1-gyel tér vissza, ha vétel során hiba lépett fel

Mikrokontroller és DSP esetén

használat: protokollértelmezőben felhívva

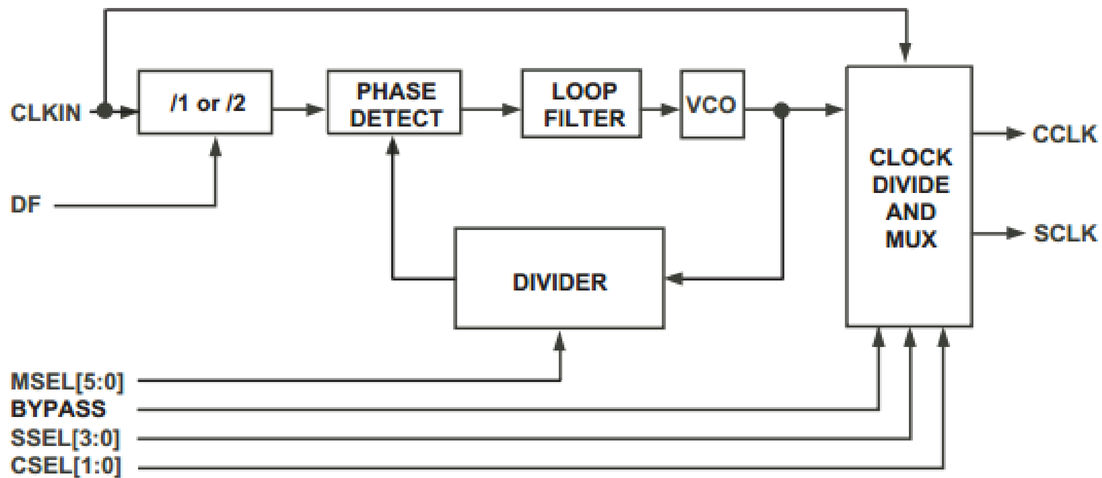
- visszaad egy globális változót, ami a fogadó megszakításban kap 0-tól különböző értéket(az UART egy regisztere alapján), ha vétel során hiba(pl.: kerethiba) lépett fel.

hibás üzenet törlése ( ) – hibát jelző változót nullázza

Mikrokontroller és DSP esetén

használat: protokollértelmezőben felhívva

- a hibát jelző globális változót is, és az UART hibajelző regiszterét is alaphelyzetbe állítja, törli.

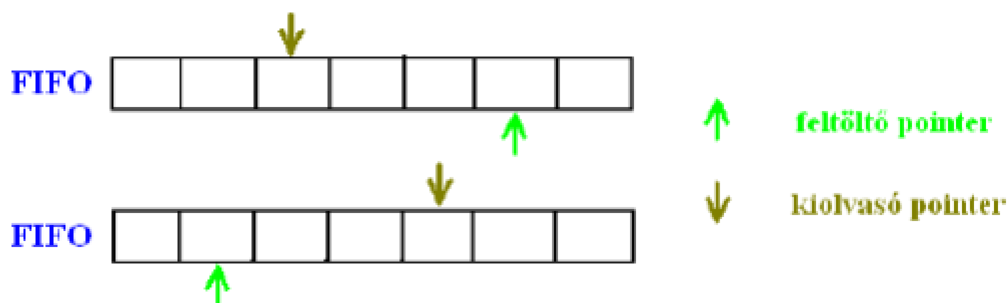


27. ábra: Blackfin 537 fáziszárt hurka [15]

A szükséges függvények listájából látszik, hogy az üzenetküldés mellett kártya inicializálásra is szükség lehet. Ez az ATmega128 kártyájánál nem hangsúlyos lépés (kimerült néhány parancsban, mint például bootloader üzemelését jelző LED bekapcsolása). Viszont a Blackfin 537 kártyán ez a lépés fontos, hiszen a processzormag órajele (CCLK) és a rendszerórajel (SCLK) is egy fáziszárt hurokkal (PLL) van többszörözve egy 25 MHz-es kvarc oszcillátorból. A többszörözést a 27. ábra szemlélteti. Az órajelek létrehozása mellett a megszakítások inicializálását is el kellett végezni.

Az első függvényen kívül gyakorlatilag üzenetküldéssel kapcsolatos függvények vannak. Ezek működése a következőképpen foglalható össze. Két külön FIFO-ban történik a kimenő és beérkező bájtok tárolása (mikrokontroller esetén a FIFO 1 elemű volt, DSP esetén nem). Ezek segítségével végzik műveleteiket az üzenetküldéssel kapcsolatos függvények. Értelemszerűen a fogadó FIFO az eszközhöz beérkező bájtokat tárolja, míg a küldő FIFO a kimenőket. Mindkét FIFO-hoz 2-2 pointer tartozik. A feltöltést mutató pointer megadja, hogy melyik helyre lett beírva a FIFO-ba legutoljára

beírt bájt. A kiolvasást mutató pointer pedig a FIFO-ból legutoljára kiolvasott bájt helyére mutat. A pointer párokra a feltöltés és kiolvasás közti késleltetés miatt van szükség. Ha a FIFO maximális hosszát túllépnék a pointerek, akkor lenullázódnak, és újra növelhetők.

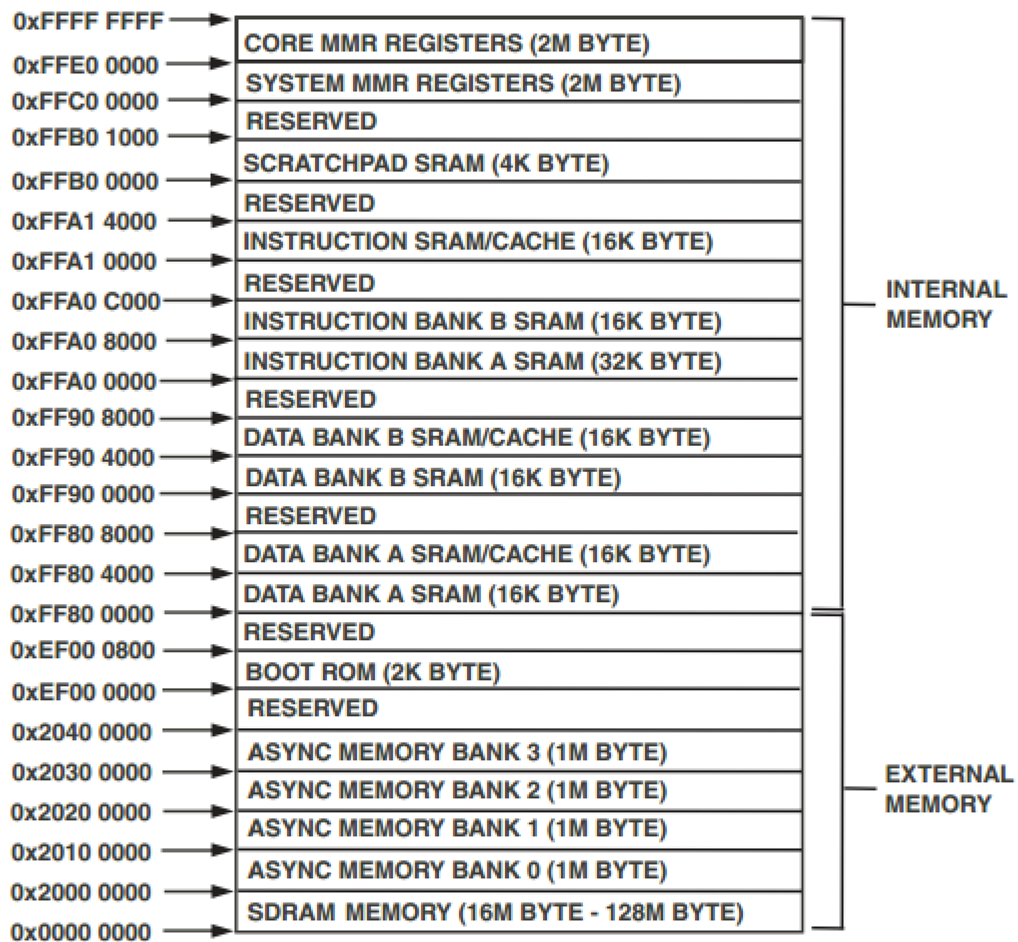


28. ábra: A feltöltő és kiolvasó pointerek a FIFO-ban

A fenti ábra felső része egy olyan állapotot mutat, amikor mindkét pointer ugyanannyiszor nullázódott. Az alsó része pedig egy olyat, amikor a feltöltő pointer eggyel többször nullázódott mint a kiolvasó.

4.3.2 Parancsvégrehajtó

Ez a modul memóriakezelést és resetelést végző függvényeket tartalmaz. Resetelő függvényekre azért van szükség, mert a hoszt program is resetelheti az eszközt. A firmware elindításához pedig egy speciális resetre van szükség. Az ATmega128 esetében AVR LibC-t használtam az implementáláshoz. Ez egy függvénykönyvtár, mely AVR eszközökhöz készült. A Blackfin 537-nél nem használtam függvénykönyvtárat, ott a memória adatlapja, a kártya és a DSP dokumentációja alapján meghatározott címre meghatározott adatot írtam. Gyakorlatilag ezzel adtam parancsokat a flash memóriának, mely a kártyán volt.



29. ábra: Blackfin 537 memóriatérképe [15]

A flash memória kezdő címe a DSP-nél az async memory bank 0 címe volt (0x20000000). Ennek a memóriának a speciális címeire kellett speciális adatokat írni, hogy a törlés és írás megtörténjen. Emellett a műveletek befejezéséig várakozó függvényt is implementáltam. A függvények részletesen:

Flash írás (memóriacím, hossz, *mutató_adatra) - visszaadja a művelet sikerességét

Mikrokontroller esetén

használat: a parancs értelmezőből meghívva

- memóriacím ellenőrzése a bootloader védelme érdekében (rossz címnél sikertelen írás),
- megszakítások kikapcsolása,
- várakozás függőben lévő memória író műveletekre,
- memória írás 16 bitenként,
- várakozás a művelet befejezéséig,
- írt memória szekció visszakapcsolása,
- megszakítások visszakapcsolása,
- visszatérés hiba függvényében.

DSP esetén

használat: a parancs értelmezőből meghívva

- megszakítások kikapcsolása,
- flash hibajelzés resetelése,
- várakozás függőben lévő memória műveletekre,
- memória írás 16 bitenként,
- várakozás a művelet befejezéséig,
- flash hiba lekérdezése,
- megszakítások visszakapcsolása,
- visszatérés a flash hibának megfelelően.

Flash olvasás (memóriacím, hossz, *mutató_adatra) - visszaadja a művelet sikerességét

Mikrokontroller esetén

használat: a parancs értelmezőből meghívva

- memóriacím ellenőrzése (érvénytelen címnél sikertelen olvasás),
- megszakítások kikapcsolása,
- adatok kiolvasása,
- megszakítások visszakapcsolása,
- visszatérés hiba függvényében.

DSP esetén

használat: a parancs értelmezőből meghívva

- memóriacím ellenőrzése (érvénytelen címnél sikertelen olvasás),
- megszakítások kikapcsolása,
- flash hibajelzés resetelése,
- várakozás függőben lévő memória műveletekre,
- memória olvasás,
- megszakítások visszakapcsolása,
- visszatérés a flash hibának megfelelően.

Flash törlés (memóriacím) - visszaadja a művelet sikerességét

Mikrokontroller esetén

használat: a parancs értelmezőből meghívva

- memóriacím ellenőrzés a bootloader védelme érdekében (rossz címnél sikertelen törlés),
- megszakítások kikapcsolása,
- várakozás függőben lévő memória író műveletekre,
- memória page törlése,
- várakozás a művelet befejezéséig,
- törölt memória szekció visszakapcsolása,
- megszakítások visszakapcsolása,
- visszatérés hiba függvényében.

DSP esetén

használat: a parancs értelmezőből meghívva

- megszakítások kikapcsolása,
- flash hibajelzés resetelése,
- várakozás függőben lévő memória műveletekre,

- memória blokk törlése,
- várakozás a művelet befejezéséig,
- flash hiba lekérdezése,
- megszakítások visszakapcsolása,
- visszatérés a flash hibának megfelelően.

Reset ( ) - újraindítja az eszközön lévő bootloadert, és magát az eszközt

Mikrokontroller és DSP esetén

használat: a parancs értelmezőből meghívva

- bootloader LED-jének kikapcsolása,
- megszakítások kikapcsolása,
- watchdog timer indítása,
- megszakítások visszakapcsolása.

Firmware indítás ( ) - a memória alkalmazói szekciójára ugorva folytatódik a végrehajtás

Mikrokontroller esetén

használat: a parancs értelmezőből meghívva

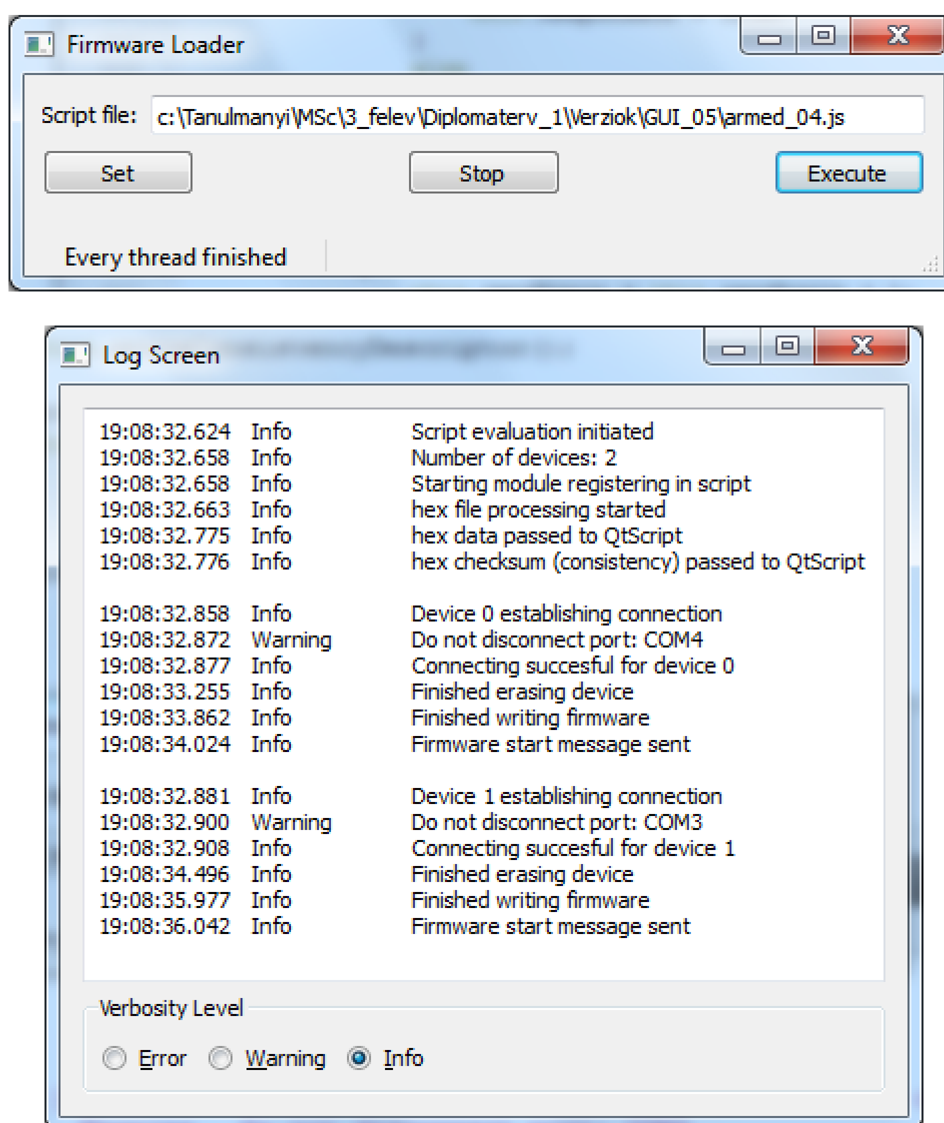
- bootloader LED-jének kikapcsolása,
- processzor alapbeállításainak visszaállítása,
- megszakítás vektor firmware rész (applikációs szekció) elejére állítása,
- megszakítás generálása időzítővel.

DSP esetén

- bootloader LED-jének kikapcsolása,
- processzor alapbeállításainak visszaállítása,
- a DSP boot ROM-jának MEMBOOT függvényét felhívva a firmware rész elejére ugrás.

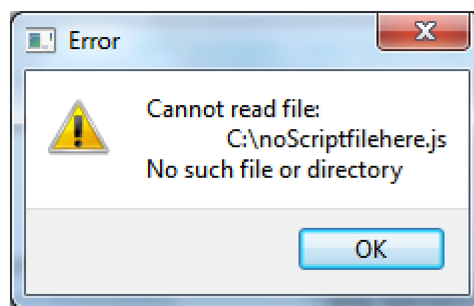
5. Eredmények bemutatása

Mivel a hoszt program képes üzenetek megjelenítésére, melyek támpontot adnak a firmware-frissítés állapotáról, ezért az eredmények bemutatását a hoszt program kimenetének ismertetésével végzem.

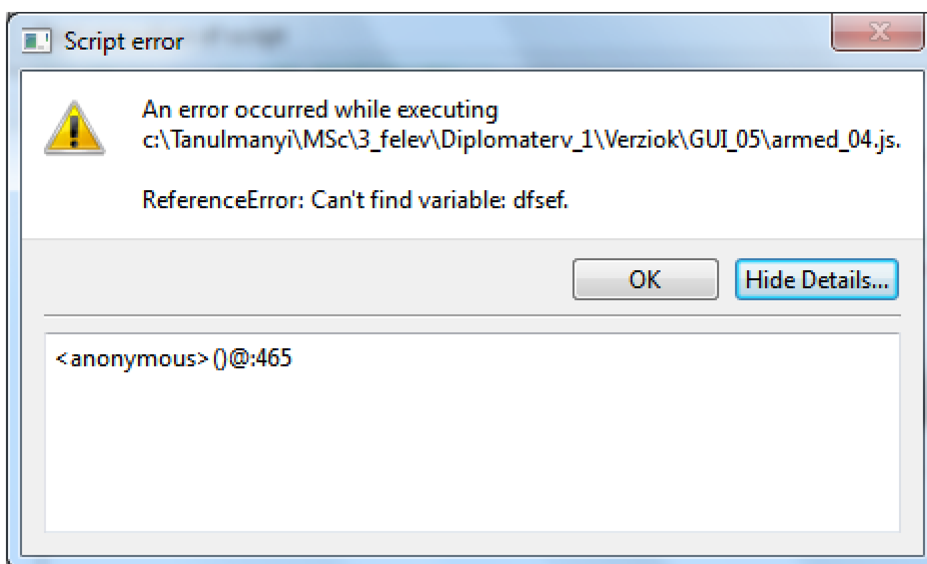


30. ábra: Firmware-frissítés eredménye

A 30. ábrán a sikeres firmware-frissítés után megjelenő GUI látható. Az ATmega128 kártyán és a DSP kártyán végeztem párhuzamosan egyidejűleg ebben az esetben a tesztet. A firmware-frissítés sikeres volt mindkét eszközre, az alkalmazói programok elindultak. Később a GUI hibajelző képességét is teszteltem. Először nem létező fájlt adtam meg szkriptfájlként. Majd egy létező szkriptfájlba csempésztem hibát. Mindkét esetben megjelent felugró ablak:



31. ábra: Nem létező fájlra figyelmeztető felugró ablak

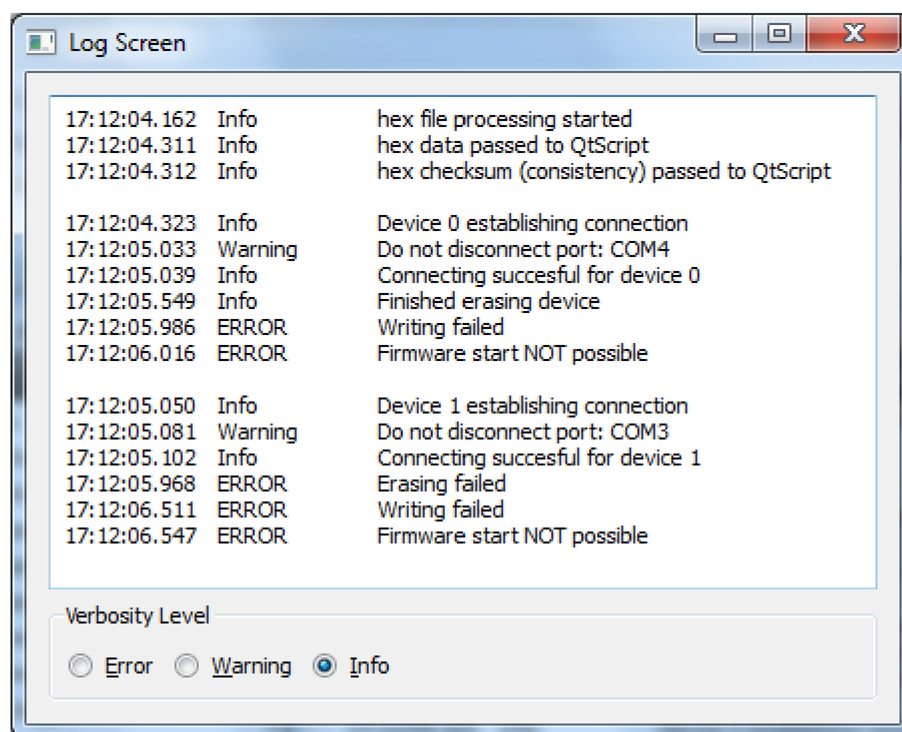


32. ábra: Hibás szkript fájl jelző felugró ablak

A szkript hibára figyelmeztető ablak megadja részletesen, hogy mi a hiba oka, és hogy hanyadik sorban van a hiba. Azt is megmondja, hogy melyik függvény hibás. Mivel

függvényen kívül helytelenül deklaráltam egy változót, ezért névtelen függvényként jelzi ki a forrást az értelmező. Egyébként magát az üzenetkijelző ablakot is lehet korlátozottan hibafelderítésre használni, hiszen a szkriptből is lehet üzenetet küldeni.

Továbbá teszteltem a firmware-frissítés menet közbeni leállítását is. Ekkor a Stop gomb megnyomása után megszakadtak a műveletek. Jól láthatóan az 1. eszköz memóriatörlése még sikeres volt, de a többi művelet már nem. Az alábbi üzenetek jelentek meg:



33. ábra: A firmware-frissítés leállításánál megjelenő üzenetek

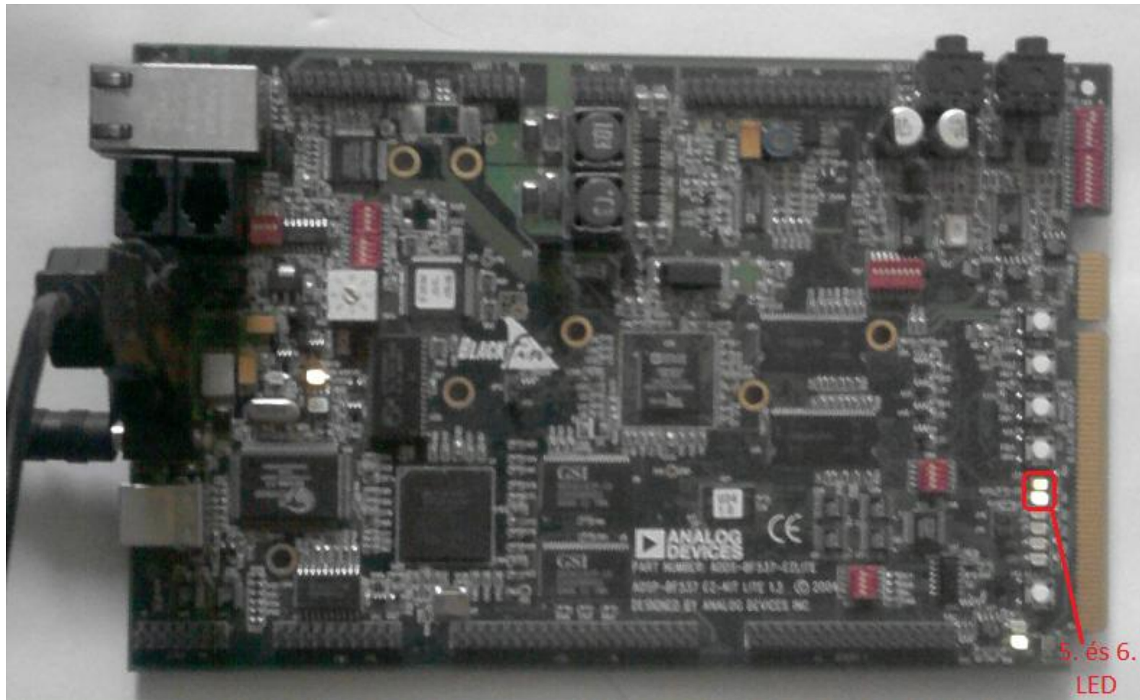
A firmware-frissítésről képeket is készítettem, a mikrokontrollerről és a DSP-ről egyaránt. Ezeken a képeken kiindulási állapotként egy firmware futása látható, utána a bootloader futása, majd a frissítés után elinduló másik firmware futása is megfigyelhető. Ugyanis különböző LED-ek vezérlését végzik a különböző programok. A bootloader futását mindkét eszköznél 1-1 LED folyamatos világítása jelzi. A firmware-frissítés után az ATmega 128-nál folyamatosan világít egy narancssárga LED, és villog egy

vörös. A DSP-nél pedig frissítés után egy másik LED villog, mint ami a bootloadernél folyamatosan világított, vagy ami a frissítés előtt világított:

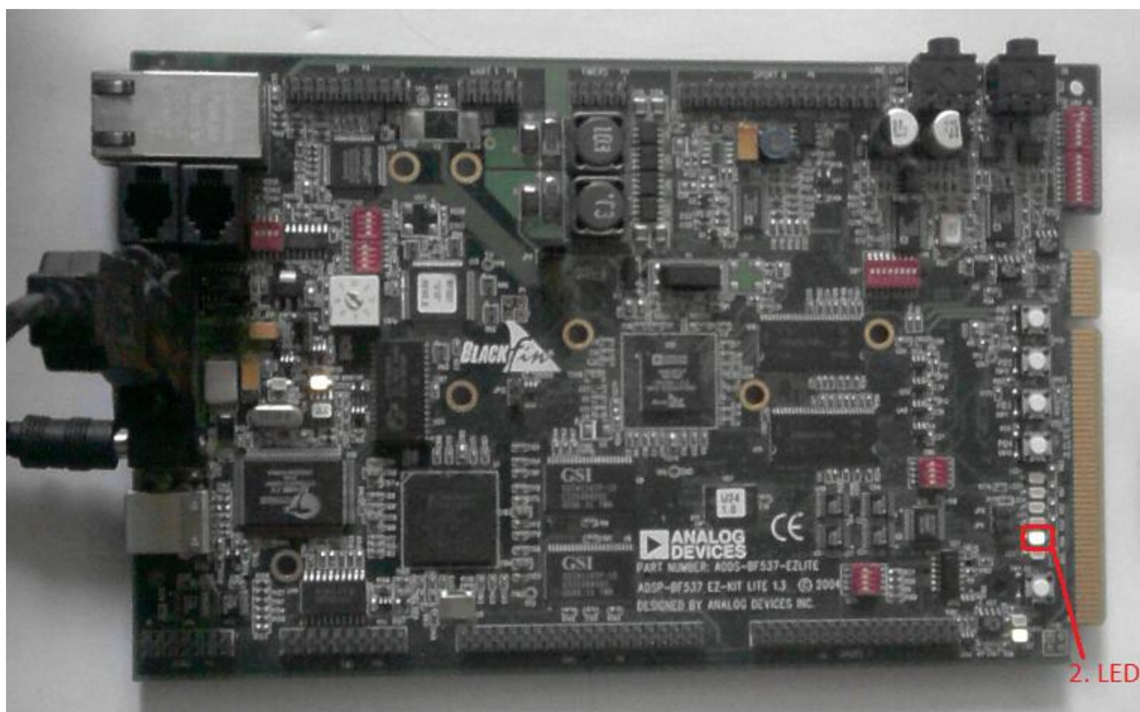
34. ábra: Az ATmega 128 a firmware-frissítés előtt futó firmware-rel

35. ábra: Az ATmega 128 a bootloader futása közben

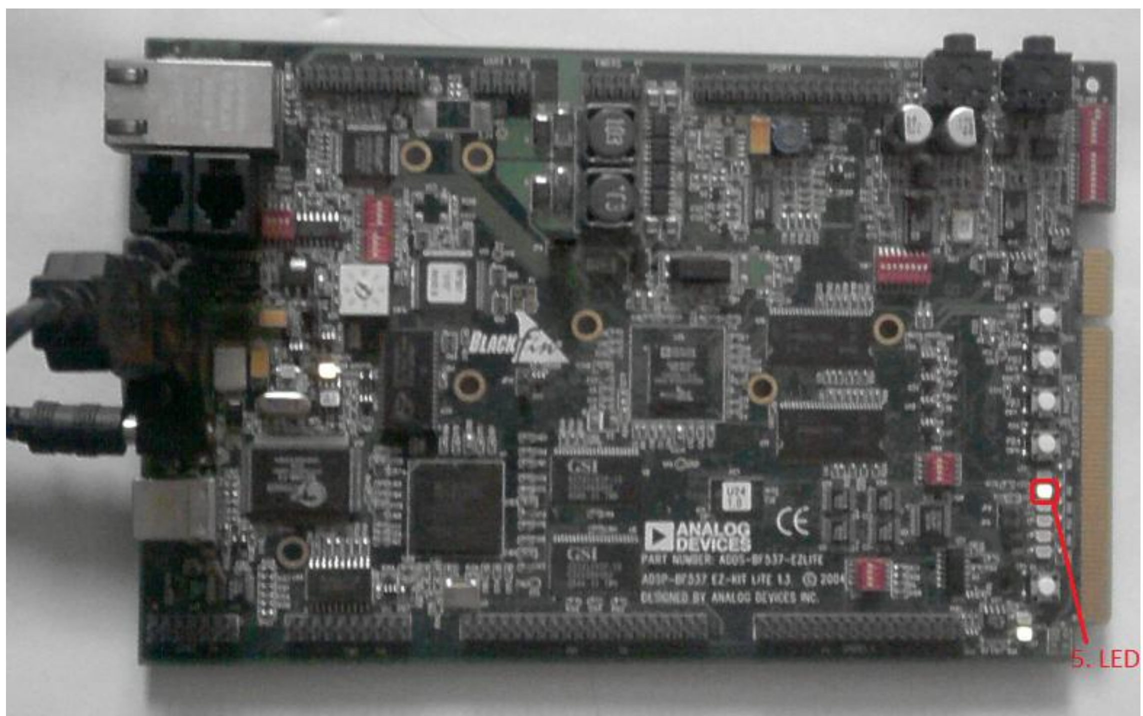
36. ábra: Az ATmega 128 firmware-frissítés után, az új firmware futása közben



37. ábra: A DSP firmware-frissítés előtt



38. ábra: A DSP a bootloader futása közben



39. ábra: A DSP firmware-frissítés után, az új firmware futása közben

6. Összefoglalás

A diplomatervezés során először megismerkedtem a firmware-frissítés problémakörével. A megvalósítani kívánt funkciók ismeretében pontosítottam a szoftver rendszertervet. Kiválasztottam a megvalósításhoz szükséges szoftver eszközöket. Illetve megismerkedtem az adott fejlesztőkörnyezettel, ha nem volt választási lehetőségem (a mikrokontroller esetében az AVR Studio, a DSP esetében a Visual DSP++ megkerülhetetlen volt).

Ezután elvégeztem az eszköz oldali szoftver, azaz bootloader implementálását, illetve portolását. Először egy ATmega128-hoz tartozó kártyát használtam ehhez. A portolást pedig egy Blackfin 537 DSP-t tartalmazó kártyára végeztem el.

Végül megvalósítottam a hoszt oldali programot, és teszteltem vele mindkét kártyát. A megfelelő hex fájlokat és szkript fájlt biztosítva a firmware-frissítés sikeres volt. A firmware elindult, erről a kártyákon villogó LED-ek is tanúskodtak.

Továbbfejlesztési lehetőség a verzióinformációt kezelő szkriptfüggvény megírása. Ezen túl a hoszt program tüzetes tesztelése is hasznosnak bizonyulhat az esetlegesen előforduló program bugok kiiktatására.

7. Irodalomjegyzék

[1]. Joseph A. Fisher, Paolo Faraboschi, Cliff Young: *Embedded Computing*, Elsevier, 2004

[2]. Wojciech Maly: *SIA Roadmap and Design & Test*, [Online]. http://cc.ee.ntu.edu.tw/~ywchang/Courses/Vlsi2k/SIA97.pdf. [Hozzáférés dátuma: 04 2013]

[3]. *Qt (framework),* [Online]. http://en.wikipedia.org/wiki/Qt_%28framework%29. [Hozzáférés dátuma: 01 2013]

[4]. *Qt reference documentation,* [Online]. http://doc.qt.nokia.com/4.7/widgets-and-layouts.html. [Hozzáférés dátuma: 03 2013]

[5]. Jasmin Blanchette, Mark Summerfield: *C++ GUI Programming with Qt 4, Second Edition,* Prentice Hall, 2008

[6]. Molketin: *The Book of Qt 4 - The Art of Building Qt Applications,* Open Source Press, 2007

[7]. *Qt Creator Manual,* [Online]. http://doc.qt.nokia.com/qtcreator-2.3/images/qtcreator-formedit.png. [Hozzáférés dátuma: 12 2012]

[8]. Dave Marshall: *Threads: Basic Theory and Libraries,* [Online]. http://www.cs.cf.ac.uk/Dave/C/node29.html. [Hozzáférés dátuma: 03 2013]

[9]. *Threading without the headache,* [Online]. http://labs.qt.nokia.com/2006/12/04/threading-without-the-headache. [Hozzáférés dátuma: 10 2012]

[10]. *Signals and Slots,* [Online]. http://doc.qt.digia.com/4.3/signalsandslots.html. [Hozzáférés dátuma: 01 2013]

[11]. *Making Applications Scriptable*, [Online]. http://qt-project.org/doc/qt-4.8/scripting.html. [Hozzáférés dátuma: 02 2013]

[12]. *Intel HEX,* [Online]. http://hu.wikipedia.org/wiki/Intel_HEX. [Hozzáférés dátuma: 02 2013]

[13]. The Modbus Organization: *MODBUS over serial line specification and implementation guide V1.02,* [Online].

http://cars9.uchicago.edu/software/epics/Modbus_over_serial_line_V1_02.pdf. [Hozzáférés dátuma: 04 2013]

[14]. *Custom Script Class Example,* [Online]. http://www.trinitydesktop.org/docs/qt4/script-customclass.html. [Hozzáférés dátuma: 05 2013]

[15]. *ADSP-BF 537 Blackfin Processor Hardware Reference*, [Online]. http://www.analog.com/static/imported-files/processor_manuals/ADSP-BF537_hwr_rev3.4.pdf. [Hozzáférés dátuma: 04 2013]