

## FELADATKIÍRÁS

A piaci termékek előállításánál gyakran a költségek kétharmadát a szoftverfejlesztés teszi ki. Ezért a beágyazott alkalmazásoknál egyre inkább megszokottá válik beágyazott operációs rendszerek használata. Az összetett hardverek alkalmazása, a kód újrahasznosíthatósága, a csoportban történő fejlesztés és a multitask igénye szintén szükségessé teszi valamilyen operációs rendszer alkalmazását.

A feladat célja különböző operációs rendszerek megismerése, előnyei és hátrányaik kitapoztatása és egy ipari alkalmazáson keresztül való összehasonlítása. A hallgató feladatának a következőkre kell kiterjednie:

- Adjon áttekintést az elterjedt operációs rendszerek
  - o felépítéséről,
  - o működéséről,
  - o előnyeiről,
  - o hátrányairól!
- Különböző fejlesztőkártyák segítségével hozzon létre beágyazott operációs rendszeres alkalmazást!
- A feladat megoldása során használjon különböző komplexitású és erőforrás-igényű operációs rendszereket!
- Hasonlítsa össze az operációs rendszereket különböző szempontok alapján!
- A hallgató végezzen irodalomkutatást a teljesítménymutatók mérésének témaében!
- Tegyen javaslatot az összehasonlítás alapját adó metrikákra és végezze el az összehasonlító teszteket!
- Adjon javaslatot az elemzett operációs rendszerek felhasználásának területeire!



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

# Operációs rendszerek összehasonlítása mikrovezérlős rendszerekben

DIPLOMATERV

*Készítette*

Bálint Ádám

*Egyetemi konzulens*

Szabó Zoltán

*Vállalati konzulens*

Mikó Gyula

2017. május 12.

# Tartalomjegyzék

<b>1. Elméleti áttekintés</b>	<b>8</b>
1.1. Általánosan használt licencek . . . . .	8
1.1.1. Zárt forráskódú szoftver . . . . .	8
1.1.2. Nyílt forráskódú szoftver . . . . .	8
1.2. Operációs rendszer feladatai . . . . .	11
1.2.1. Operációs rendszer definíciója . . . . .	11
1.2.2. Ütemezés . . . . .	12
1.2.3. Operációs rendszer által nyújtott szolgáltatások . . . . .	13
1.2.4. Operációs rendszer használata esetén felmerülő problémák . . . . .	19
<b>2. Operációs rendszerek bemutatása</b>	<b>24</b>
2.1. Használt fejlesztőkártyák . . . . .	24
2.1.1. STM32F4 Discovery . . . . .	24
2.1.2. Raspberry Pi 3 . . . . .	25
2.2. A választás szempontjai . . . . .	27
2.2.1. STM32F4 Discovery . . . . .	27
2.2.2. Raspberry Pi 3 . . . . .	28
2.3. FreeRTOS . . . . .	28
2.3.1. Ismertető . . . . .	28
2.3.2. Taszkok . . . . .	29
2.3.3. Kommunikációs objektumok . . . . .	31
2.3.4. Megszakítás-kezelés . . . . .	33
2.3.5. Erőforrás-kezelés . . . . .	35
2.3.6. Memória-kezelés . . . . .	36
2.4. µC/OS-III . . . . .	38
2.4.1. Ismertető . . . . .	38
2.4.2. Taszkok . . . . .	38
2.4.3. Kommunikációs objektumok . . . . .	40
2.4.4. Megszakítás-kezelés . . . . .	42
2.4.5. Erőforrás-kezelés . . . . .	42
2.4.6. Memória-kezelés . . . . .	42
2.5. Raspbian . . . . .	43
2.6. Windows 10 IoT Core . . . . .	43

<b>3. Teljesítménymérő metrikák</b>	<b>44</b>
3.1. Szakirodalmakban fellelhető metrikák . . . . .	45
3.1.1. Memóriaigény . . . . .	45
3.1.2. Késleltetés . . . . .	45
3.1.3. Jitter . . . . .	45
3.1.4. Rhealstone . . . . .	45
3.1.5. Legrosszabb válaszidő . . . . .	51
3.2. Vizsgált operációs rendszer jellemzők . . . . .	52
<b>4. Rendszterterv</b>	<b>53</b>
4.1. Megvalósítandó feladat . . . . .	53
4.2. Részletes terv . . . . .	54
4.3. Hardverterv . . . . .	56
4.4. Szoftverterv . . . . .	57
4.4.1. Késleltetés és Jitter . . . . .	57
4.4.2. Rhealstone értékek . . . . .	57
4.4.3. Legrosszabb válaszidő . . . . .	64
<b>5. Megvalósítás</b>	<b>66</b>
5.1. STM32F4 Discovery . . . . .	66
5.1.1. FreeRTOS . . . . .	66
5.1.2. µC/OS-III . . . . .	69
5.1.3. Vezérlő szoftver . . . . .	70
5.2. Raspberry Pi 3 . . . . .	71
5.2.1. Windows 10 IoT Core . . . . .	71
5.2.2. Raspbian . . . . .	72
<b>6. Mérés</b>	<b>74</b>
6.1. Mérési elrendezés . . . . .	74
6.2. Adatok feldolgozása . . . . .	75
<b>7. Eredmények kiértékelése</b>	<b>76</b>
7.1. Memóriaigény . . . . .	76
7.2. Mérések . . . . .	77
<b>8. Konklúzió</b>	<b>86</b>
8.1. További lehetőségek . . . . .	87
<b>Függelék A. A rövidebb licencek eredeti szövegei</b>	<b>95</b>
A.1. MIT License . . . . .	95
A.2. BSD . . . . .	95
A.2.1. 4-clause BSD (eredeti) . . . . .	95
A.2.2. 3-clause BSD (módosított) . . . . .	96
A.2.3. 2-clause BSD (egyszerűsített) . . . . .	97

<b>Függelék B. Kapcsolási rajz</b>	<b>98</b>
B.1. Top level . . . . .	98
B.2. Tápfeszültség . . . . .	98
B.3. Szenzorok . . . . .	99
B.4. ADS7924 . . . . .	99
B.5. BLE112 . . . . .	99
B.6. SD kártya . . . . .	100
B.7. GPIO . . . . .	100
B.8. Mérések kivezetései . . . . .	101
<b>Függelék C. NYÁK rajzolat</b>	<b>102</b>
C.1. Alkatrész oldal . . . . .	102
C.2. Forrasztási oldal . . . . .	102
<b>Függelék D. Folyamatábrák</b>	<b>103</b>
<b>Függelék E. Felhasználói füleletek</b>	<b>110</b>

## HALLGATÓI NYILATKOZAT

Alulírott *Bálint Ádám*, 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 átvettettem, 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 autentikált 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.

Budapest, 2017. május 12.

---

*Bálint Ádám*

hallgató

# Kivonat

Az ipari termékekben megtalálható beágyazott rendszerek által megvalósítandó feladatok komplex működési mechanizmusokat igényelhetnek.

Az elvégzendő feladat bonyolultságával arányosan nehezedik a feladatot megvalósító szoftver tervezése, mely a termék fejlesztési költségét is nagyban befolyásolja.

Beágyazott operációs rendszer használatával a fejlesztési idő csökkenthető a rendelkezésre álló tervezési minták alkalmazásával, a csoportban történő fejlesztéssel, illetve a meglévő szoftverek újrahasznosításával. A piacon elérhető operációs rendszerek között megtalálható real-time feladatok ellátására is alkalmas rendszerek. Ezek között léteznek ingyenesen elérhetők és olyanok is, amelyekhez az egyes perifériák használatát támogató kiegészítő modulok is kaphatóak.

A beágyazott rendszereket gyártó cégeknek nem áll rendelkezésükre egységesen elfogadott teljesítménymutató, amely alapján össze tudnák hasonlítani a különböző operációs rendszereket. Ennek következtében csak előzetes szimpátiára alapozva tudják meghozni a döntést a választásuk során.

A dolgozatban bemutatom a szakirodalmakban gyakran előforduló teljesítménymutatókat, majd kiegészítő hardver segítségével elvégzem a hozzájuk tartozó méréseket. A dolgozat végén értékelem az eredményeket, majd javaslatot teszek az egyes operációs rendszerek felhasználásának területére.

# Abstract

The tasks implemented by embedded systems found in industrial products can require complex operating mechanisms.

The complexity of the planning of the software is proportional to that of the tasks performed. This may also greatly influence the development cost of the product as they get more complex.

Development time may be reduced by using an embedded operating system which allows us to use available design patterns, team development, and recycling of existing software.

Operating systems available on the market include operating systems capable of performing real-time tasks. Amongst them, there are free-of-charge ones and some which are available with additional modules that support the use of each peripheral.

Manufacturers of embedded systems do not have a unified benchmark for operating systems as a basis of comparison, therefore they may make their decision based solely on prior sympathy.

In the dissertation, I present the commonly used performance indicators often found in the scientific literature and then perform the related measurements using additional hardware. At the end of the thesis I evaluate the results and furthermore make a suggestion for the use of each operating system.

# Bevezető

A piacon megtalálható termékek döntő többsége beágyazott mikrokontrollert használ a feladata elvégzéséhez. A bonyolult feladatokat végző eszközök esetén – mint például egy folyamatirányító egység, vagy egy UAV fedélzeti számítógépe – beágyazott operációs rendszer használata nélkül a feladat implementálása hosszadalmas szoftvertervezési folyamatot igényel, ami a fejlesztési költségeket is megnöveli.

Beágyazott operációs rendszer használatával a szoftverfejlesztési feladatok nagy mértékben egyszerűsödnek, és a keletkező forráskód is hordozhatóvá válik. Viszont az elérhető vagy megvásárolható operációs rendszerek közül a megfelelő kiválasztása nem egyszerű feladat, mert nincs egységesen alkalmazott teljesítménymutató.

A dolgozat első fejezetében részletesen bemutatom a nyílt forráskódú szoftverek által elterjedten használt licenceket, majd az operációs rendszer általános feladatait ismertetem.

A második fejezetben a kiválasztott fejlesztőkártyák és a kiválasztás szempontjainak részletezésén keresztül a diplomamunka során használt operációs rendszerek részletes elemzéséhez jutunk.

A harmadik fejezetben a szakirodalmakban fellelhető teljesítménymérő metrikákat ismertetem. A fejezetben szó esik arról, hogy egy mérésnek milyen követelményeknek kell megfelelni, majd a fejezet végén a mérendő paraméterek kerülnek felsorolásra.

A negyedik fejezetben a mérés során használt hardverrel szemben állított követelményeket ismertetem, melyet a rendszerterv követ. A rendszerterv részeként bemutatásra kerül az elkészült hardver, majd a mérést megvalósító szoftver működését részletezem.

A ötödik fejezet a megvalósítás során használt egyéb szoftvereket és eszközöket sora-koztatja fel, illetve a fejlesztés során felmerülő problémákat és az arra talált megoldásokat ismertetem.

A hatodik fejezet a méréshez használt eszközöket, a mérési elrendezést, illetve az adatok feldolgozásának módját tartalmazza.

A hetedik fejezetben a mérési eredmények találhatóak ábrákkal és szöveges magyarázattal.

A dolgozat utolsó fejezetében összefoglalom a mérési eredményekből levont következtéseket, illetve ajánlást teszek az egyes operációs rendszerek felhasználásának területeire és a fejlesztés során összegyűjtött tapasztalat alapján szubjektív szempontok alapján is értékelem a rendszereket.

# 1. fejezet

## Elméleti áttekintés

### 1.1. Általánosan használt licencek

A minden nap használt szoftverek általánosan használt licencek általánosan használatával kapcsolatban. Ezek között találhatóak egyedileg alkalmazott feltételek, de vannak elterjedt licencek, amiket például nyílt forráskódú szoftverek esetén gyakran alkalmaznak.

#### 1.1.1. Zárt forráskódú szoftver

Zárt forráskódú szoftver esetén a forráskód kizárolag a gyártó/fejlesztő kezében marad. A szoftver használatáért általában valamelyik összeget kell fizetni, amely gyakran magában foglalja a fellépő problémák megoldásában való segítségnyújtást is. Természetesen a zárt forráskód nem vonja kötelezően maga után, hogy a felhasználónak fizetnie kell a szoftver használatáért. Sok gyártó a termékét ingyen elérhetővé teszi (freeware). Ilyen esetben a kereskedelmi célú felhasználásból és/vagy a segítségnyújtásból származik a fejlesztést fedező bevétel. A gyártó a felhasználás feltételeit saját maga szabhatja meg, így ebben az esetben nem lehet általános elemzést végezni a licencekkel kapcsolatban.

#### 1.1.2. Nyílt forráskódú szoftver

A szoftverek másik nagy csoportját alkotják a nyílt forráskódú szoftverek. Ekkor a forráskód publikusan elérhető. A nyílt forráskódú szoftverek esetén is lehetőség van egyéni licenc használatára, azonban a szoftverek többségénél elterjedt, általánosan használt licencekkel találkozunk[1].

#### MIT License

Az MIT licenc a legegyszerűbb licencek egyike. A licenc alá eső forráskód szabadon másolható, módosítható, terjeszthető, akár más licenc alá helyezhető.

## **BSD licenc**

A BSD licenc eredetileg egy egyszerű és szabad felfogású licenc számítógép szoftverek számára, amit először 1980-ban használtak a BSD UNIX operációs rendszerhez[2]. Az idő előrehaladtával több módosítást kellett végrehajtani a licenc szövegezésén.

### **4-clause BSD (eredeti)**

Az eredeti, másnéven 4-clause BSD licenc négy pontban foglalta össze a felhasználás feltételeit. A licenc tartalmazza a szerzői jog keletkezésének évét, a fejlesztő szervezet nevét és a szerzői jog tulajdonosának nevét. A licenc megengedi a szoftver használatát minden forrás, minden bináris formában, akár módosítással, akár anélkül, amennyiben a felhasználás feltételei teljesülnek. A szöveg azonban tartalmaz egy megkötést, amely a használat során gyakran kellemetlenségekért köt elő a felhasználók körében: amennyiben a szoftver bármilyen részére vagy a szoftver használatára hivatkozás történik egy reklámanyagban, úgy a szoftver fejlesztőjét fel kell tüntetni a szövegben. Amennyiben több, eredeti BSD licenc alá eső szoftvert tartalmazó rendszer került hivatkozásra, úgy az említett lista hamar nagy méretűvé válhatott[2]. Ezen kellemetlenség miatt került először módosításra a BSD licenc.

A négy pont az alábbi feltételeket szabja a felhasználó számára:

1. A forráskódnak tartalmaznia kell a copyright szövegét, a felhasználás feltételeit felsorakoztató listát, illetve a nyilatkozatot.
2. Bináris formában történő terjesztés esetén a copyright szövegét, a felhasználás feltételeit felsorakoztató listát, illetve a nyilatkozatot a dokumentációban és/vagy valamely másik, a terjesztett szoftverhez adott anyagban fel kell tüntetni.
3. Bármiféle hirdetési anyagban, ami a szoftver jellemzőire vagy használatára hivatkozik, fel kell tüntetnie az alábbi elismerést: *Ez a termék a <szervezet> által fejlesztett szoftvert tartalmaz.*
4. Sem a <szervezet> neve, sem a közreműködő partneinek neve nem használható fel a szoftverből származó termék népszerűsítésére vagy ajánlására erre vonatkozó írásbeli engedély nélkül.

A nyilatkozatban tisztázásra kerül, hogy a szerzői jog tulajdonosa nem vonható felelősségre a szoftver használatából, felhasználásából eredő üzemkimaradás, adatvesztés, profitvesztés vagy egyéb veszteség bekövetkezése esetén.

Ez a licencelési forma ma már nem elterjedt, inkább a módosított (3-clause BSD) vagy az egyszerűsített (2-clause BSD) licenc használata ajánlott.

### **3-clause BSD (módosított)**

Az előző, eredeti verzióhoz képest a különbség, hogy eltávolították a hirdetésre vonatkozó elismerési feltételt, valamint a nyilatkozat szövegében a közreműködőket is mentesítette minden garanciális felelősségtől.

## **2-clause BSD (egyszerűsített)**

Az egyszerűsített (vagy gyakran FreeBSD licencnek is nevezett) BSD licenc két pontban foglalja össze a felhasználás feltételeit, amit a módosított licencből a népszerűsítésre vonatkozó pont eltávolításával kaptak.

## **GNU GPL**

A GNU General Public Licence azzal a céllal jött létre, hogy garantálja egy program szabad módosítását és terjesztését, illetve hogy az szabad szoftver maradjon a felhasználói számára. A licenc az angol *free* kifejezést nem a termék árára, hanem annak szabad felhasználására használja.

A licenc több módosításon keresztül ment. A két legutolsó változatot ismertetem a tövábbiakban.

## **GNU GPLv2**

A GNU GPLv2 licenc megengedi a program szabad terjesztését és módosítását, amennyiben a terjesztett/módosított szoftver örökli az eredeti szerzői jogi megjegyzéseket. A garanciális kötelezettségek elutasításra kerülnek, de nem tiltja meg (akár ellenszolgáltatás fejében) a vállalásukat.

Ha módosítás történt a programon, akkor fel kell tüntetni a módosító nevét és a módosítás dátumát.

A programtól elkülöníthető munkára nem kötelező kiterjeszteni a licenc hatállyát, amennyiben az külön kerül terjesztésre. Ha az eredeti (szerzői jogok alá eső) munkával együtt kerül terjesztésre, akkor automatikusan kiterjesztésre kerülnek a feltételek.

A program terjeszthető futtatható (bináris) formában is, amennyiben a forráskód publikusan elérhető marad.

A felhasználó jogait tovább korlátozni nem lehet.

Ha valamilyen okból kifolyólag (például bírói végzés folytán) a terjesztés csak bináris formában lehetséges, akkor a program nem terjeszthető.

Ha valamely országban a terjesztés nem lehetséges, úgy a szerzői jogok eredeti tulajdonosa földrajzi megkötést adhat a terjesztésre vonatkozóan, ami a licenc teljes értékű részét képezi.

A programot más, szerzői jogi szabályozásában különböző szoftverbe beépíteni csak a szerzőtől kapott engedély birtokában lehet.

A GPLv2 licenc nem engedi meg, hogy a program része legyen szellemi tulajdon képező szoftvernek. Ebben az esetben az LGPL licenc használata javasolt.

## **GNU GPLv3**

A GPL 2007-es módosítása során a tartalmi változtatásokon kívül formai változtatás is történt, ami az érthetőséget hivatott szolgálni.

A tartalmi változtatások megcélozzák a más licencekkel való kompatibilitást, a licenc-

cel védett szoftverhez járó felhasználói termék módosított programmal való használatának biztosítását (tivoizáció<sup>1</sup> elkerülése), jobban specifikálja a jogok elvesztését, illetve azok visszaszerzésének lehetőségeit, illetve megjelenik a *megkülönböztető* licenc definíciója. Megkülönböztető licencnek tekintendő minden olyan licenc, ami nem tartalmazza az eredeti jogokat, vagy megtiltja valamely, az eredeti licencben foglalt jog gyakorlását.

## 1.2. Operációs rendszer feladatai

Az operációs rendszer elsődleges feladata a használt processzor perifériáinak kezelése, azokhoz meghatározott interfész biztosítása. További feladata még a létrehozott taszkok ütemezése, a taszkok közötti kommunikáció és szinkronizáció megvalósítása.

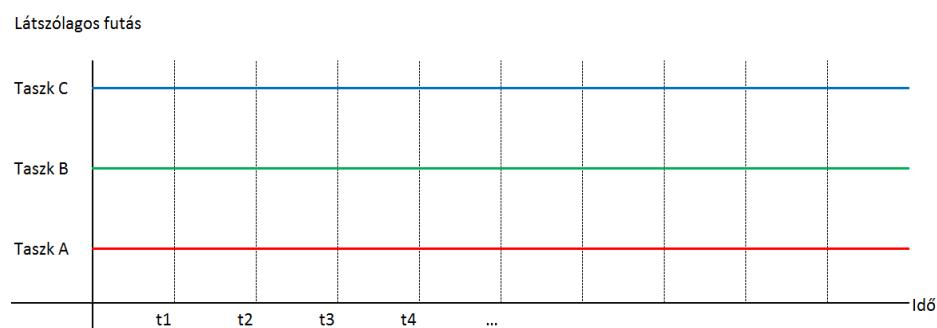
### 1.2.1. Operációs rendszer definíciója

Az operációs rendszer egy szoftver, ami kezeli a számítógép alap funkcióit és szolgáltatásokat biztosít más programok (vagy alkalmazások) számára. Az alkalmazások valósítják meg azt a funkcionalitást, amire a számítógép felhasználójának szüksége van, vagy a számítógép felhasználója akar. Az operációs rendszer által nyújtott szolgálatások az alkalmazások fejlesztését egyszerűsítik, ezáltal felgyorsítják a fejlesztés folyamatát és karbantarthatóbbá teszik a szoftvert[3].

Többféle operációs rendszert különböztetünk meg a futtható folyamatok, az egyidejűleg kezelt felhasználók számának és az ütemező működésének függvényében. A továbbiakban csak a multitaskot támogató operációs rendszerekkel foglalkozom.

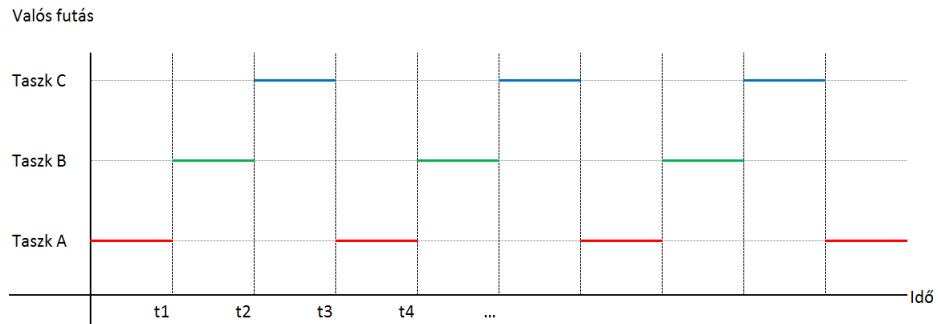
### Valósidejű operációs rendszer

A legtöbb operációs rendszer látszólag lehetővé teszi több program egyidejű futtatását (multitasking). A valóságban minden processzormag csak egy szálat tud egy időpillanatban futtatni. Az operációs rendszer ütemezője a felelős azért, hogy eldöntse, melyik program mikor fussion, és a programok közti gyors váltással éri el az egyidejű futás látszatát (1.2. ábra)[3].



**1.1. ábra.** Látszólag a folyamatok párhuzamosan futnak.

<sup>1</sup>A TiVo egy digitális videórögzítő eszközököt gyártó cég, ami az eladott termékeiben GPLv2 licenc alá eső szoftverek használt. A Series 1 termékét módosított szoftverrel is lehetett használni, viszont a 2000-es évek elején piacra dobott Series 2 DVR box-ban már digitális védelmet használtak, ami meggyalolta a módosított rendszerek alkalmazását.



**1.2. ábra.** A valóságban minden folyamat egy kis időszakot kap a processzortól.

Az operációs rendszer típusát az ütemező döntési mechanizmusa határozza meg. Egy real-time operációs rendszer ütemezője úgy van megtervezve, hogy a végrehajtási minta determinisztikus legyen. Ez beágyazott rendszerek esetén érdekes, mert a beágyazott rendszereknél gyakran követelmény a valósidejűség, vagyis hogy a rendszernek egy szigorúan meghatározott időn belül reagálnia kell egy adott eseményre[3]. A valósidejű követelményeknek való megfelelés csak úgy lehetséges, ha az operációs rendszer ütemezője előre megjósolható döntéseket hoz.

Valósidejű operációs rendszerek közül megkülönböztetjük a soft real-time és a hard real-time rendszereket.

Soft real-time rendszer esetén nem probléma, ha nem érkezik válasz a megadott határidőn belül, az csak a rendszer minősítését rontja.

Hard real-time rendszer esetén viszont a rendszer alkalmatlanná válik a feladatra, ha a határidőt nem tudja betartani. Például egy személygépjármű légsákjának késedelmes nyitása akár halálos következményekkel is járhat[4].

### 1.2.2. Ütemezés

Az operációs rendszer egyik meghatározó része a használt ütemező működésének elve, mely meghatározza az eszköz alkalmasságát egy adott feladatra. Az ütemező dönti el, hogy a futásra kész taszkok közül melyik futhat a következő ütemezési szakaszban. További feladata taszkváltáskor az éppen futó taszk állapotának elmentése és a futtatandó taszk állapotának betöltése. Az ütemező három okból futhat le:

- Egy taszk lemond a futás jogáról,
- Megszakítás érkezik (tick esemény, külső megszakítás),
- Egy taszk létrejött vagy befejeződött.

A taszkváltási mód szerint kétféle működést különböztetünk meg:

- **Preemptív ütemezés:** ekkor az ütemező megszakíthatja az éppen futó taszkot, amennyiben magasabb prioritású taszk futásra kész állapotban várakozik,
- **Nem-preemptív vagy kooperatív ütemezés:** ebben az esetben az ütemező csak akkor futtat új taszkot, ha az éppen futó taszk befejeződött vagy explicit lemond a

futásról (blokkolódik vagy átadja a futás jogát).

A továbbiakban ismertetem néhány elterjedt ütemezési mechanizmus alapelveit.

### **First-come first-served (Kiszolgálás beérkezési sorrendben)**

A taszkok futtatása érkezési sorrendben történik. A beérkező taszkok egy sorba kerülnek, ahonnan sorrendben kapják meg a CPU használat jogát.

Az átlagos várakozási idő nagy lehet, ha egy időigényes folyamatra több gyors lefutású folyamat várakozik[5].

Nem preemptív.

### **Shortest Job First (Legrövidebb feladat először)**

A várhatóan leggyorsabban lefutó taszk kerül futási állapotba ütemezés bekövetkezésekor. Helyes működés esetén az átlagos várakozási idő optimális lesz, viszont a gyakorlatban nehéz előre megjósolni egy taszk futási idejét[6].

Lehet preemptív és nem-preemptív is. Preemptív esetben Shortest Remaining Time First (Legrövidebb hátralevő idejű először) ütemezésről beszélünk.

### **Prioritásos ütemezés**

A következő taszk kiválasztása a prioritása alapján történik. Növelni lehet a hatékonyságát más metódusok egyidejű használatával (például a prioritást a várható futási idejéből határozzuk meg; a prioritást növeljük az idő elteltével; a prioritást csökkentjük az eddigi futó állapotban töltött idő arányában)[6].

Rossz tervezés esetén az alacsony prioritású feladatok nem jutnak processzoridőhöz (kiéheztetés).

Lehet preemptív és nem-preemptív is.

### **Round Robin**

Időosztáson alapuló mechanizmus. A várakozási sor egy cirkuláris buffer. minden taszk adott időszeletet kap, majd az időszelet lejártával a sorban következő taszk kapja meg a futás jogát.

### **Hibrid ütemezés**

A felsorolt ütemezési elveket akár keverve is lehet alkalmazni. Ilyen ütemezési mechanizmus például a *multilevel queue*, ahol minden sor saját ütemezési algoritmussal rendelkezik, és egy külön algoritmus felel az egyes sorok arbitrációjáért (1.3. ábra)[5].

#### **1.2.3. Operációs rendszer által nyújtott szolgáltatások**

Az operációs rendszer felelős az egyes taszkoknak szükséges memóriaterületek kezeléséért és a taszkok közötti kommunikáció megvalósításáért.



**1.3. ábra.** *Multilevel queue ütemezés.*

## Memória-kezelés

Egy taszk létrehozásakor az operációs rendszer osztja ki a taszk számára a használható memóriaterület helyét és méretét, és ezt az információt a rendszernek tárolnia kell. Ha az adott taszk befejezi a futását (megszűnik), akkor az operációs rendszer feladata a taszhöz tartozó memóriaterület felszabadítása is.

Amennyiben a taszk dinamikusan allokál memóriát futás közben, úgy ezen memória kezelése szintén az operációs rendszer feladatkörébe tartozik, viszont az egyszerűbb operációs rendszerek esetében a futás közben lefoglalt memória felszabadítása a taszk felelősége.

## Atomi művelet

Bizonyos műveletek során szükség van a műveletsor szigorúan egymás utáni futtatására. Ilyen eset például a megosztott adatterületre való írás, amikor ha taszkváltás következik be az adatterület írása közben, akkor a tartalmazott adat érvénytelen értéket vehet fel. Az ilyen, *oszthatatlan* műveleteket nevezzük atomi műveleteknek.

A konzisztencia biztosítása céljából az atomi műveleteket *kritikus szakaszokba* kell ágyazani, ezzel jelezve az operációs rendszernek, hogy a műveletek végrehajtása alatt nem következhet be taszkváltás, bizonyos esetekben megszakítás sem.

Az operációs rendszerek a kritikus szakaszokat több szinten megvalósíthatják. Legszebb esetben a megszakítások letiltásra kerülnek, és az ütemező a kritikus szakasz befejezéséig felfüggesztett állapotban van. A kritikus szakasz megvalósításának egy kevésbé drasztikus módja az ütemező letiltása. Ekkor a kódrészlet védtet a más taszkok általi preemptálástól, viszont a megszakítások nem kerülnek letiltásra.

A kritikus szakaszt a lehető leggyorsabban el kell hagyni, mert különben a beérkező megszakítások és magasabb prioritású taszkok késleltetést szenvednek, ami rontja az alkalmazás hatékonyiságát.

## Kommunikációs objektumok

Az alkalmazások egymástól független taszkok konstrukciójából állnak. Viszont ezeknek a taszkoknak ahhoz, hogy a feladatukat el tudják látni, gyakran kommunikálniuk kell egy-

mással.

A felsorolt kommunikációs objektumok listája nem teljes. Bizonyos operációs rendszerek ezektől eltérő struktúrákat is használhatnak, és az itt felsorolt struktúrák implementációja is eltérhet a leírtaktól (természetesen az sem biztos, hogy implementálva van az adott objektum).

## Szemafor

Két szemafor típust különböztetünk meg:

- Bináris szemafor, amikor a szemafor két értéket vehet fel,
- Számláló szemafor, mikor a szemafor több állapotot is felvehet.

A szemaforon két művelet értelmezett:

- A szemafor jelzése (elterjedt elnevezések: give, signal, post, V() művelet<sup>2</sup>), amikor a szemafor értéke növelésre kerül.
- A szemafor elvétele (elterjedt elnevezések: take, wait, pend, P() művelet<sup>3</sup>), amikor a szemafor először tesztelésre kerül, hogy tartalmaz-e elemet, ha igen, akkor az értékét csökkentjük, ha nem, akkor várakozunk addig, amíg valamelyik másik taszk elérhetővé nem teszi azt.

A szemafor látszólag egyszerűen helyettesíthető egy egyszerű változó (boolean vagy elő-jel nélküli egész) használatával, viszont a beépített szemafor struktúra atomi műveletként kerül kezelésre, illetve a rendszer automatikusan tudja kezelnı a várakozó folyamatok álla-potok közti mozgatását.

Szemaforokat leggyakrabban szinkronizációs célból, vagy erőforrások védelmére használ-nak.

## Bináris szemafor

Bináris szemafor esetén a szemafor két értéket vehet fel. Felfogható úgy is, mint egy egy adat tárolására alkalmas (egy elem hosszú) sor, melynek nem vizsgáljuk a tartalmazott értékét, csak azt, hogy éppen tartalmaz-e adatot vagy sem[4].

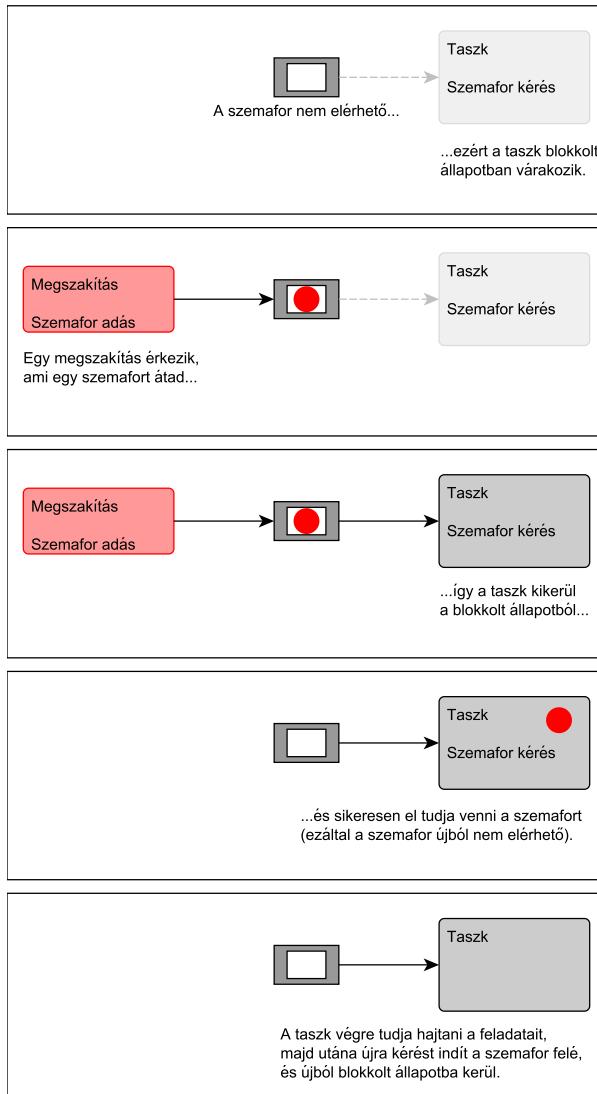
Leggyakoribb felhasználása a taszkok szinkronizálása. Ekkor az egyik taszk a futásának egy adott pontján várakozik egy másik taszk jelzésére. Ezzel a módszerrel megvalósítható a megszakítások taszkokban történő kezelése, ezzel is minimalizálva a megszakítási rutin hosszát. Erre láthatunk példát az 1.4. ábrán.

Bináris szemafor használatakor különös figyelmet kell fordítani arra, hogy ha a szemafor egy adott taszkban gyakrabban kerül jelzésre, mint ahogy feldolgozzuk, akkor jelzések veszhetnek el. Amíg az egyik jelzés várakozik, addig az utána következő eseményeknek nincs lehetőségeük várakozó állapotba kerülni (1.5. ábra).

---

<sup>2</sup>A jelölés a holland *verhogen* (növelés) szóból származik.

<sup>3</sup>A jelölés a holland *proberen* (tesztelés) szóból származik.



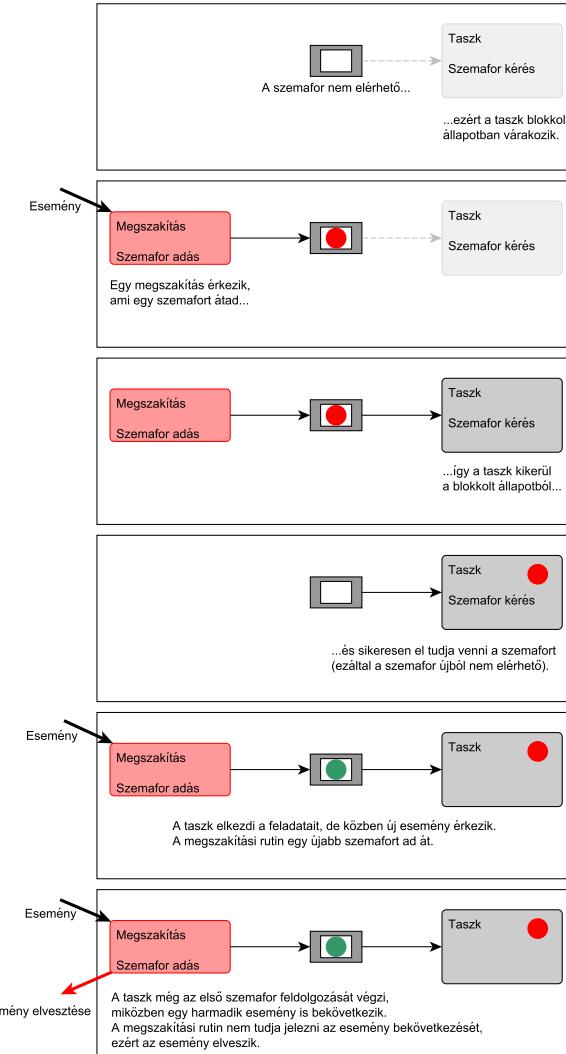
1.4. ábra. Szinkronizáció bináris szemafor segítségével[4].

### Számláló szemafor

A számláló típusú szemafor minden jelzéskor növeli az értékét. Ekkor (amíg el nem éri a maximális értékét) nem kerül *Blokkolt* állapotba a jelző taszk. A számláló szemafor felfogható úgy, mint egy egynél több adat tárolására képes sor[4] (1.6. ábra), melynek nem vizsgáljuk az értékét, csak azt, hogy éppen tartalmaz-e még adatot vagy sem.

Két felhasználása elterjedt a számláló szemaforoknak:

- **Események számlálása:** ekkor minden esemény hatására növeljük a szemafor értékét (új elemet helyezünk a sorba). A szemafor aktuális értéke a beérkezett és a feldolgozott események különbsége. A számlálásra használt szemafor inicializálási értéke nulla[4].
- **Erőforrás menedzsment:** ekkor a szemafor értéke a rendelkezésre álló erőforrások számát mutatja. Mikor az operációs rendszertől az erőforrást igényeljük, akkor a szemafor értékét csökkentjük, mikor felszabadítjuk a birtokolt erőforrást, akkor a szemafor



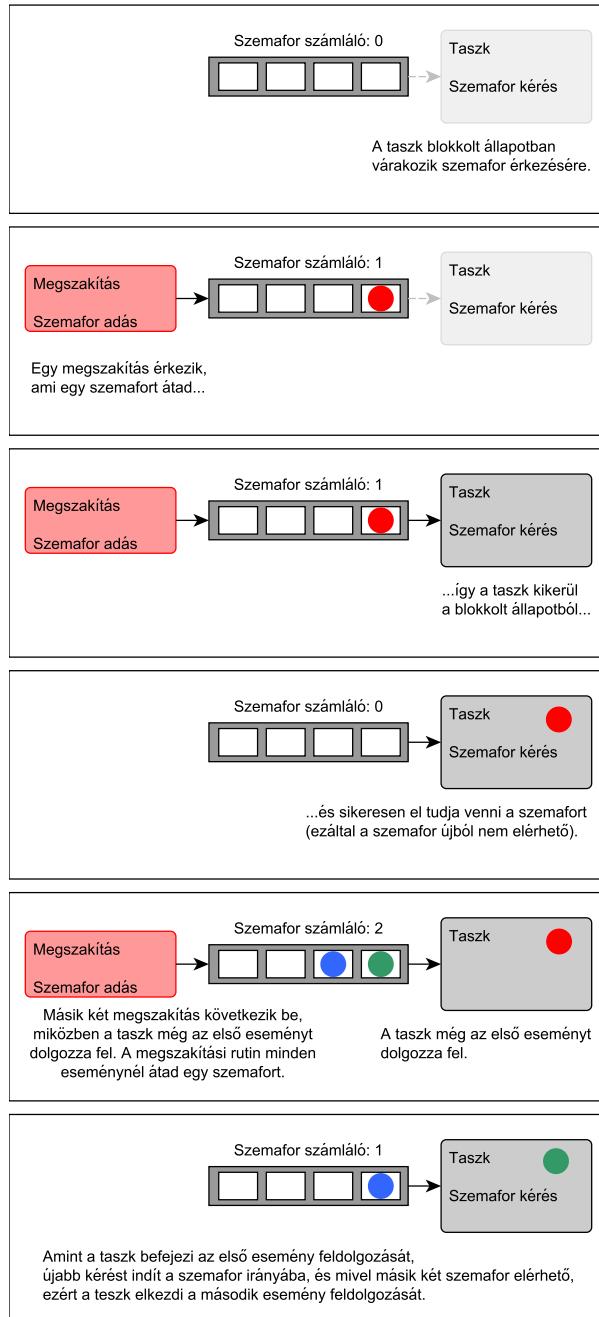
**1.5. ábra.** Esemény bekövetkezésének elvesztése bináris szemafor használata során.

értékét növeljük. Ha a szemafor értéke nulla, akkor nincs rendelkezésre álló erőforrás. Erőforrások kezelésére használt számláló szemafor esetén az inicializálási érték az elérhető erőforrások száma[4].

## Mutex

Taszkok vagy taszkok és megszakítási rutinok között megosztott erőforrás kezelésekor a mutex (kölcsönös kizárás) használata indokolt. Mikor egy taszk vagy megszakítás hozzáférést indít egy erőforráshoz, akkor a hozzá tartozó mutex-et elkéri. Ha az erőforrás szabad, akkor az igénylő taszk megkapja a kezelés jogát, és mindenkor megtartja, amíg be nem fejezi az erőforrással való munkát (1.7. ábra). A mutex-et a lehető legkorábban (az erőforrással való munka befejeztével) fel kell szabadítani, ezzel is csökkentve az esetleges holt pont kialakulásának veszélyét.

Látható, hogy a mutex nagyon hasonlít a bináris szemaforhoz. A különbség abból adódik, hogy mivel a bináris szemafort leggyakrabban szinkronizációra használjuk, ezért azt



**1.6. ábra.** Számláló szemafor működésének szemléltetése[4].

nem kell felszabadítani: a jelző taszk vagy megszakítás jelzést ad a szemforon keresztül a feldolgozó tasznak. A feldolgozó taszk elveszi a szemforot, de a feldolgozás befejeztével a szemforot nem adja vissza[4].

## Sor (Queue)

A sorok fix méretű adatból tudnak véges számú üzenetet tárolni. Ezek a jellemzők a sor létrehozásakor kerülnek meghatározásra. Alapértelmezetten FIFO-ként működik, aminek bemutatását láthatjuk az 1.8. ábrán.

A sorba való írás során másolat készül az eredeti változóról, és ez a másolat kerül táró-

lásra a sorban.

### **Üzenet (Message)**

Az üzenet az adat másolása helyett csak az adatra mutató pointert továbbítja a fogadó fél számára. Mivel a kommunikáció során csak az adat referenciája kerül átvitelre, ezért különös figyelmet kell fordítani az adat konzisztenciájára.

### **Eseményjelző bitek (Event flag)**

Események bekövetkezésekor egy-egy bitet használva tartalmazza az eseményre vonatkozó információt. Előnye a szemaforral szemben, hogy több eseményjelző bitet felhasználva csoportosíthatjuk az eseményeket, így egyszerre több eseményről kapunk információt (és akár egyszerre több eseményre is várakozhatunk).

#### **1.2.4. Operációs rendszer használata esetén felmerülő problémák**

Az alkalmazás összetettségével együtt növekszik a hibák gyakorisága is. Operációs rendszer alkalmazásakor beleeshetünk abba a hibába, hogy nem gondoljuk alaposan át a szoftver működését, ami különböző problémák forrása lehet. A gyakran előforduló, tipikusnak tekinthető problémákra mutatok példákat a továbbiakban.

### **Kiéheztetés (Starving)**

Prioritásos ütemezés esetén, ha egy magas prioritású taszk folyamatosan futásra kész állapotban van, akkor az alacsony prioritású taszkok sosem kapnak processzoridőt. Ezt a jelenséget nevezzük kiéheztetésnek (starving), amit átgondolt tervezéssel könnyedén elkerülhetünk.

### **Prioritás inverzió (Priority inversion)**

Vegyük egy esetet, mikor legalább három, különböző prioritási szinten futó taszkot hozunk létre. A legalacsonyabb prioritásútól a magasabb felé haladva nevezzük őket *TaskA*-nak, *TaskB*-nek és *TaskC*-nek.

Kezdetben csak a *TaskA* képes futni, ami egy mutex segítségével megkapja egy erőforrás használati jogát. Közben a *TaskC* futásra kész állapotba kerül, ezért preemptálja a *TaskA*-t. A *TaskC* is használná az erőforrást, de mivel azt már a *TaskA* birtokolja, ezért várakozó állapotba kerül. Közben a *TaskB* is futásra kész állapotba került, és mivel magasabb a prioritása, mint a *TaskA*-nak, ezért megkapja a futás jogát. A *TaskA* csak a *TaskB* befejeződése (vagy blokkolódása) esetén kerül újra futó állapotba. Miután a *TaskA* befejezte az erőforrással a feladatait és felszabadítja azt, a *TaskC* újból futásra kész állapotba kerül, és preemptálja a *TaskA*-t.

A magyarázat illusztrációja az 1.9. ábrán látható.

A vizsgált példa során a *TaskB* késleltette a *TaskC* futását azzal, hogy nem engedte a *TaskA*-nak az erőforrás felszabadítását. Így látszólag a *TaskB* magasabb prioritással

rendelkezett, mint *TaskC*. Erre mondjuk, hogy prioritás inverzió lépett fel.

A prioritás inverzió problémájának egy megoldása a prioritás öröklés. Ekkor a magas prioritású taszk a saját prioritási szintjére emeli azt az alacsony prioritású taszkot, mely blokkolja a további futását (1.10. ábra). Amint a szükséges erőforrás felszabadul, az eredeti prioritási értékek kerülnek visszaállításra.

### Holtpont (Deadlock)

Szemaforok és mutexek használata során alakulhat ki holtponti helyzet. Nézzük azt az esetet, hogy van két, azonos prioritású taszk (a taszkok prioritása itt nem lényeges), melyek működésük során ugyan azt a két erőforrást használják. A két taszkot nevezzük *TaskA*-nak és *TaskB*-nek, a két erőforrást pedig *ResA*-nak és *ResB*-nek (1.11. ábra).

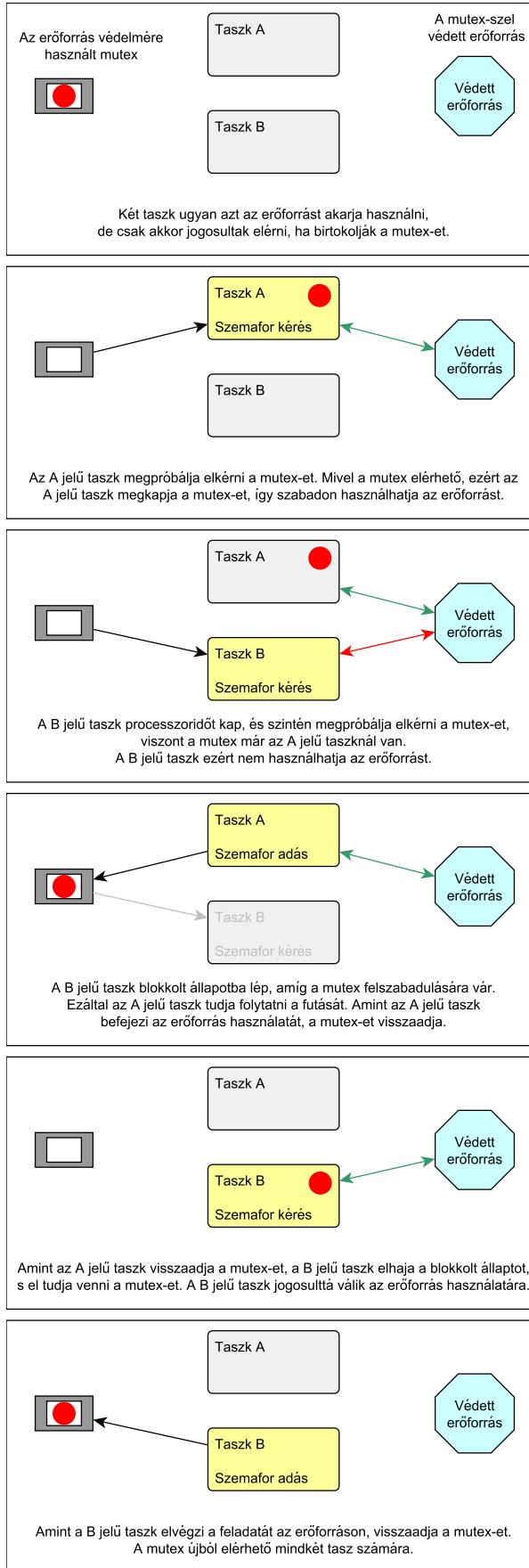
Induláskor a *TaskA* kapja meg a futás jogát, és lefoglalja a *ResA*-t. Közben lejár a *TaskA*-nak kiosztott időszelet, és *TaskB* kerül futó állapotba. A *TaskB* lefoglalja a *ResB* erőforrást, majd megpróbálja lefoglalni a *ResA* erőforrást is. Mivel a *ResA*-t már a *TaskA* használja, ezért a *TaskB* várakozó állapotba kerül. A *TaskA* újból megkapja a processzort, és hozzáférést kezdeményez a *ResB* erőforráshoz. Mivel a *ResB* erőforrást a *TaskB* folyamat birtokolja, ezért a *TaskA* is várakozó állapotba lép. Egyik folyamat sem tudja folytatni a feladatát, emiatt az erőforrásokat sem tudják felszabadítani.

A holtponti helyzetek elkerülésére és feloldására több szabály létezik, de beágyazott rendszereknél átgondolt tervezéssel, illetve időkorlát megadásával általában elkerülhető a kialakulásuk.

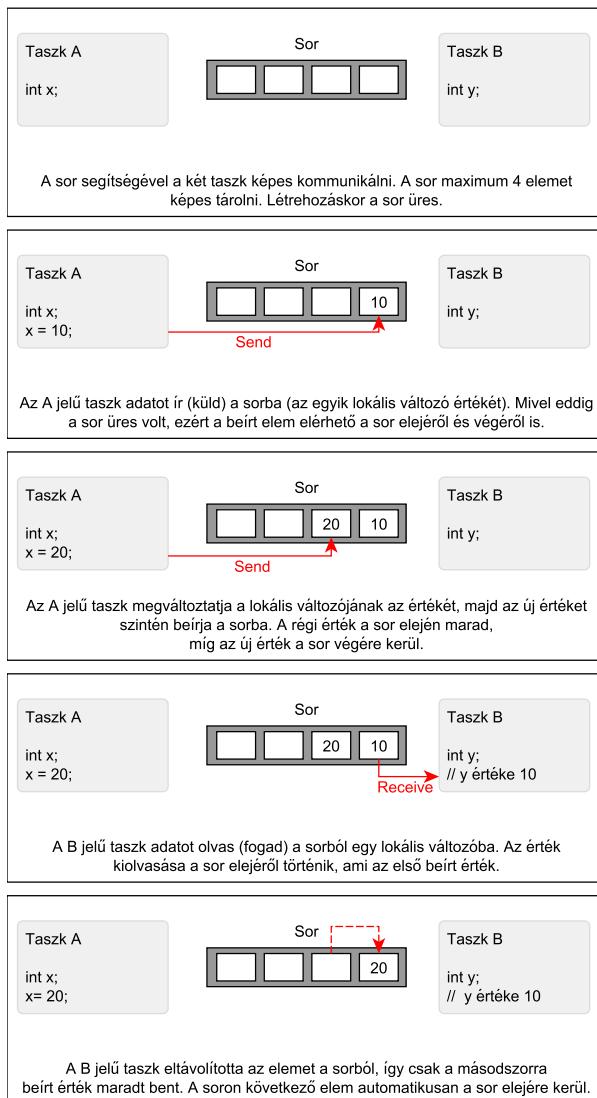
### Újrahívható függvények (Reentrant functions)

Egy függvény reentráns (újrahívható), ha biztonságosan meghívható több különböző taszk-ból vagy megszakításból.

Minden taszk rendelkezik saját stack-kel és saját regiszterekkel. Ha egy függvény minden adatot a stack-jén vagy a regisztereiben tárol (vagyis nem használ globális és statikus változókat), akkor a függvény reentráns[7].

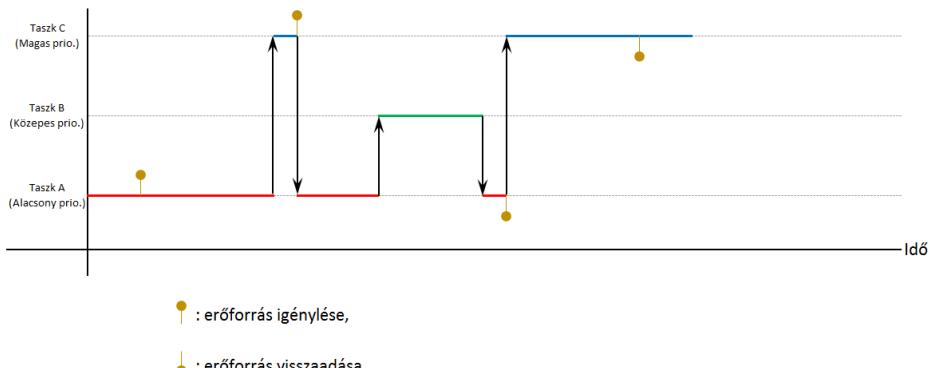


1.7. ábra. Mutex működésének szemléltetése[4].



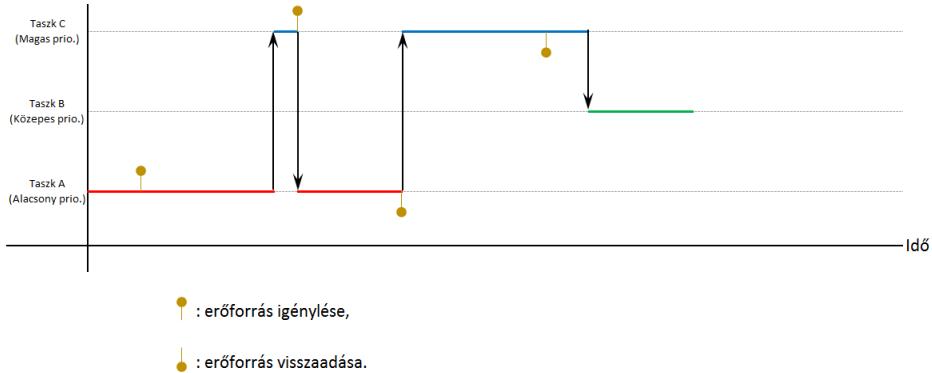
1.8. ábra. Sor működésének szemléltetése[4].

#### Prioritás inverzió

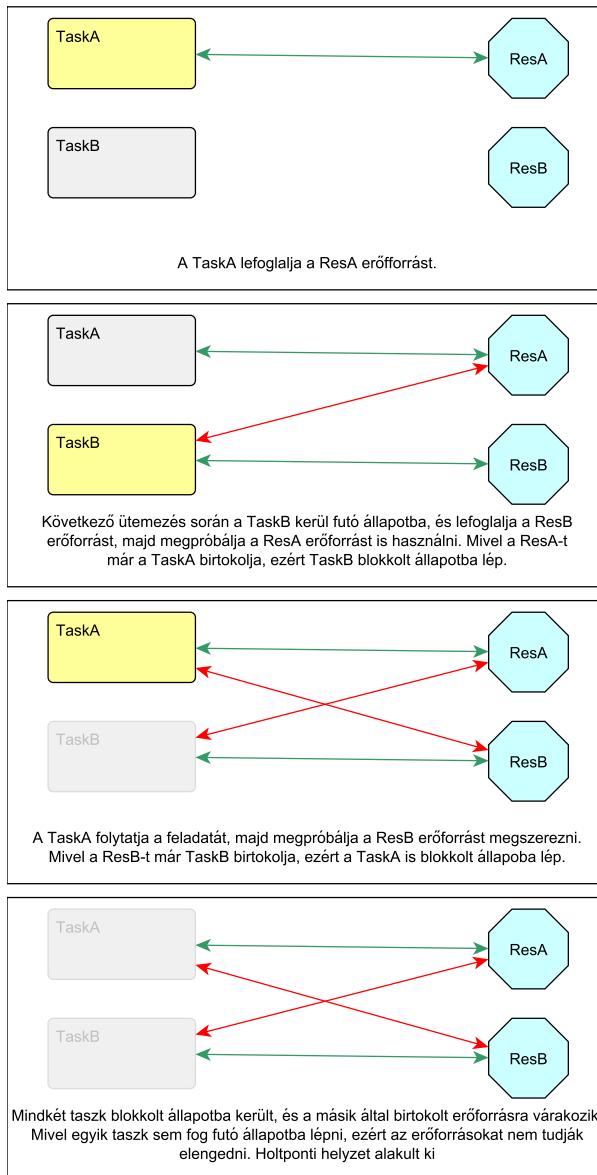


1.9. ábra. Prioritás inverzió jelensége.

Prioritás öröklés



**1.10. ábra.** Prioritás öröklés, mint a prioritás inverzió egyik megoldása.



**1.11. ábra.** Holtponyi helyzet kialakulásának egy egyszerű példája.

## 2. fejezet

# Operációs rendszerek bemutatása

### 2.1. Használt fejlesztőkártyák

#### 2.1.1. STM32F4 Discovery

Manapság egyre inkább teret nyernek maguknak az ARM alapú mikrokontrollerek, melyek nem csak nagy számítási kapacitásukkal, de egyre alacsonyabb árukkal szorítják ki versenytársaikat. Az egyik legelterjedtebb gyártó, az STMicroelectronics (továbbiakban STM) több fejlesztőkártyát is piacra bocsátott az elmúlt években, melyeken különböző mikrokontrollerek képességeit ismerheti meg a fejlesztő. A kapható fejlesztőkártyák amellett, hogy megkímélik a fejlesztőt a saját hardver tervezésétől – így a tervezési hibából adódó problémák keresésétől is –, a legtöbb esetben a gyártó széles körű támogatást is nyújt a termékekhez (mintaprogramok, fórumok, stb.).

Az STM által gyártott fejlesztőkártyák közül ár-érték arányának köszönhetően talán a leggyakrabban használt az STM32F407 Discovery (továbbiakban STM32F4 Discovery) kártya. A rajta található mikrokontroller rendelkezik a legtöbb alkalmazásban előforduló perifériák mindegyikével. A teljesség igénye nélkül:

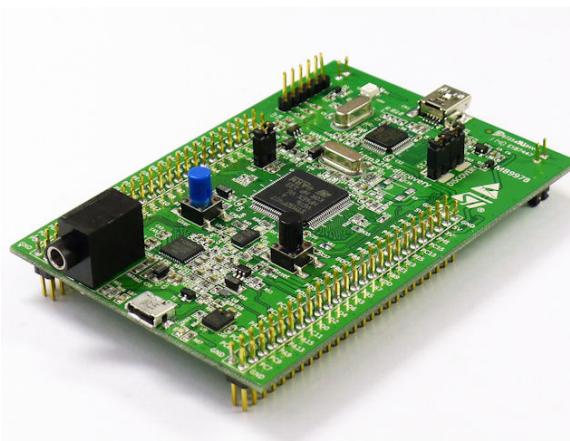
- GPIO-k,
- Soros kommunikációs portok (szinkron és aszinkron egyaránt),
- Ethernet port,
- Analog-Digital Converter,
- Digital-Analog Converter,
- Időzítők,
- High-Speed USB (OTG támogatással),
- SPI,
- I<sup>2</sup>C,
- I<sup>2</sup>S,

- SDIO (SD illetve MMC kártya kezeléséhez),
- CAN,
- Szoftveres és hardveres magszakítások.

A nagyszámú periféria mellett a mikrokontroller számítási kapacitása is kimagaslik a hasonló árkategóriájú eszközök köréből, köszönhetően az akár 168 MHz-es órajelének, a beépített CRC és lebegő pontos aritmetikai egységének, illetve a több perifériát is kezelní képes DMA-knak.

Az eszköz 1 MByte Flash memóriával és 192 kbyte RAM-mal rendelkezik[8].

A kártyán megtalálható több periféria, mely a különböző interfések kipróbalását teszi lehetővé (mint például gyorsulásszenzor, mikrofon).



**2.1. ábra.** *STM32F4 Discovery fejlesztőkártya[9].*

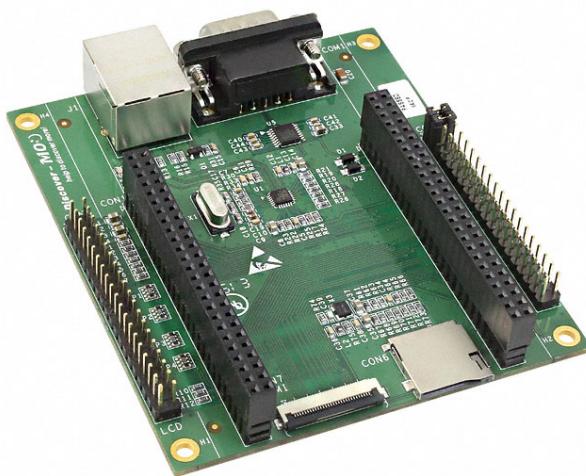
### STM32F4 Discovery - Base Board

Az STM32F4 Discovery fejlesztőeszközökhez több kiegészítő kártya is kapható, melyek célja a kipróbalható perifériák számának növelése. Egyik ilyen bővítőkártya az STM32F4DIS-BB, ami tartalmaz microSD-kártya foglalatot, az Ethernet interfész fizikai rétegét megvalósító IC-t, illetve a csatlakoztatáshoz szükséges RJ45-ös csatlakozót. Ezen kívül található rajta egy DB9-es csatlakozó – mely az egyik soros kommunikációs portot teszi elérhetővé –, egy FPC csatlakozó – mely kamera csatlakoztatását teszi lehetővé –, illetve az egyik oldali csatlakozósorra ráköthető 3,5 "-os TFT kijelző.

#### 2.1.2. Raspberry Pi 3

A Raspberry Pi Foundation-t 2008-ban alapították azzal a céllal, hogy a informatikai tudományok területén segítse az oktatást[11].

Az első nagyteljesítményű, bankkártya méretű számítógépüket 2012 februárjában bocsátották piacra, melynek ára töredéke volt az asztali számítógépekének. Azóta több verziója is megjelent az eszközöknek, melyet folyamatosan fejlesztettek mind teljesítményben, mind az integrált funkciók számában. 2015 novemberében a világ első 5 \$-os számítógépével jelentek



**2.2. ábra.** *STM32F4 Discovery Base Board kiegészítő kártya[10].*

meg a piacon, melynek a Raspberry Pi Zero nevet adták[11].

A legújabb verziójú kártya, a Raspberry Pi 3 model B az előző verzióhoz képest erősebb processzort kapott, illetve tartalmaz beépített Bluetooth és WiFi modult.



**2.3. ábra.** *Raspberry Pi 3 bankkártya méretű PC[12].*

A hardver főbb jellemzői:

- 1,2 GHz 64-bit quad-core ARMv8 CPU,
- 802.11n Wireless LAN,
- Bluetooth 4.1,
- Bluetooth Low Energy,

- 1 GB RAM,
- 4 USB port,
- 40 GPIO pin,
- HDMI port,
- Ethernet port,
- Kombinált 3,5 mm-es jack aljzat (audio és kompozit videó),
- MicroSD kártyafoglalat,
- VideoCore IV GPU.

Interfészek:

- SPI,
- UART,
- DPI (Display Parallel Interface),
- SDIO,
- PCM (Pulse-code Modulation),
- 1-WIRE,
- JTAG,
- GPCLK (General Purpose Clock),
- PWM.

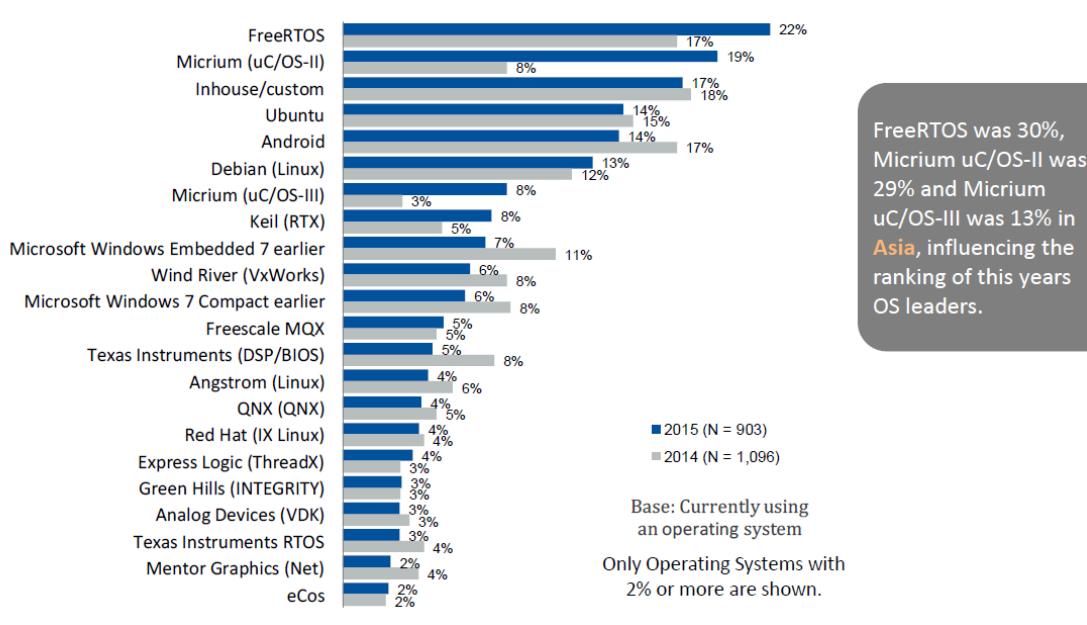
Már az első verzió megjelenésekor rendelkezésre álltak különböző linux disztribúciók portjai, melyekkel az eszköz asztali számítógépként használható volt. Népszerűségének köszönhetően a második verziótól kezdve már a Windows 10 IoT Core is támogatja a platformot.

## 2.2. A választás szempontjai

Az operációs rendszerek kiválasztásánál elsődleges szempont a hardverek támogatottsága, illetve az oktatási célra való elérhetőség volt.

### 2.2.1. STM32F4 Discovery

Az UBM Tech minden évben készít kutatást a beágyazott rendszereket piacán, melyben többek között a használt beágyazott operációs rendszerekkel kapcsolatban is publikál adatokat (a 2015-ös felmérés eredménye látható a 2.4. ábrán). A statisztika alapján a két



**2.4. ábra.** Az UBM Tech által 2015-ben publikált beágyazott operációs rendszer használati statisztika[13].

leggyakrabban használt operációs rendszer a FreeRTOS és a μC/OS-II. A μC/OS új verziója, a μC/OS-III a listán hátrébb kapott helyet, de még így is befért a tíz vezető rendszer közé. Mindhárom operációs rendszer népszerűsége nőtt a 2014-es évhez képest.

Az STM32F4 Discovery kártyához elérhető szoftvercsomag tartalmazza a FreeRTOS rendszert, ami jelzi a rendszer támogatottságának mértékét. A FreeRTOS hivatalos oldalán megvásárolhatóak a rendszer használatát bemutató könyvek, illetve online elérhetőek leírások, amik szintén segítik a rendszer megismerését. Ezáltal az első megvizsgált rendszernek a FreeRTOS-t választottam.

A Micrium μC/OS rendszerei oktatási célra ingyenesen elérhetőek, beleértve a rendszer dokumentációit is. A választott másik rendszer a μC/OS-III.

### 2.2.2. Raspberry Pi 3

A Raspberry Pi megjelenése óta támogatja különböző linux disztribúciók futtatását az eszközön, és a fejlesztőkártya második verziója óta a Windows 10 IoT Core is telepíthető rá. Az asztali alkalmazás fejlesztők körében mind a linux, mind a Windows elterjedten használt operációs rendszer, ezért a Raspberry Pi 3-on ezt a két rendszert vizsgálom meg.

## 2.3. FreeRTOS

A fejezet a [4] irodalom alapján készült.

### 2.3.1. Ismertető

A FreeRTOS a Real Time Engineers Ltd. által fejlesztett valósidejű operációs rendszer. A fejlesztők céljai között volt a rendszer erőforrásigényének minimalizálása, hogy a legki-

sebb beágyazott rendszereken is futtatható legyen. Ebből adódóan csak az alap funkciók vannak megvalósítva, mint ütemezés, taszkok közötti kommunikáció lehetősége, memória-menedzsment, de nincs beépített támogatás hálózati kommunikációra vagy bármiféle külső hardver használatára (ezeket vagy nekünk kell megírnunk, vagy harmadik félről származó könyvtárakat kell használnunk).

A rendszer módosított GPLv2 licencet használ. A licencmódosítás lehetővé teszi GPL-től eltérő licenccel ellátott modulok használatát, amennyiben azok a FreeRTOS-sal kizárolag a FreeRTOS API-n keresztül kommunikálnak[14].

### 2.3.2. Taszkok

A FreeRTOS nem korlátozza a létrehozható taszkok és prioritások számát, amíg a rendelkezésre álló memória lehetővé teszi azok futtatását. A rendszer lehetőséget biztosít ciklikus és nem ciklikus taszkok futtatására egyaránt.

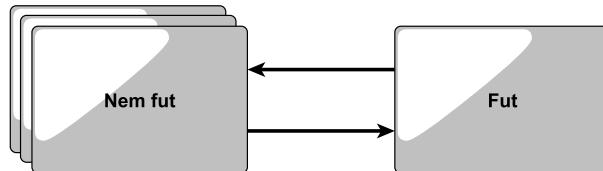
Minden taszkhoz tartozik egy TCB (*Task Control Block*) és stack. A TCB struktúra legfontosabb változói a 2.1. táblázatban láthatóak.

**2.1. táblázat.** A FreeRTOS TCB-jének főbb változói.

Változó	Jelentés
pxTopOfStack	Az stack utolsó elemére mutató pointer.
xGenericListItem	A FreeRTOS a TCB ezen elemét helyezi az adott állapothoz tartozó listába (nem magát a TCB-t).
xEventListItem	A FreeRTOS a TCB ezen elemét helyezi az adott eseményhez tartozó listába (nem magát a TCB-t).
uxPriority	A taszk prioritása.
pxStack	A stack kezdetére mutató pointer.
pcTaskName	A taszk neve. Kizárolag debug célokra.
pxEndOfStack	A stack végére mutató pointer a stack túlcordulásának detektálására.
uxBasePriority	Az utoljára taszkhoz rendelt prioritás. Mutex használata esetén a prioritás öröklés során megnövelt prioritás eredeti értékre történő visszaállítására.
ulRunTimeCounter	A taszk <i>Fut</i> állapotban töltött idejét tárolja futási statisztika készítéséhez.

A beágyazott rendszerek döntő része egymagos processzorokat használ, amiből az következik, hogy egyszerre csak egy taszk futhat. A taszkok eszerint két nagy csoportba oszthatóak: éppen futó taszk (*Fut* állapot), illetve az összes többi (*Nem fut* állapot). Ez az egyszerűsített felosztás látható a 2.5. ábrán.

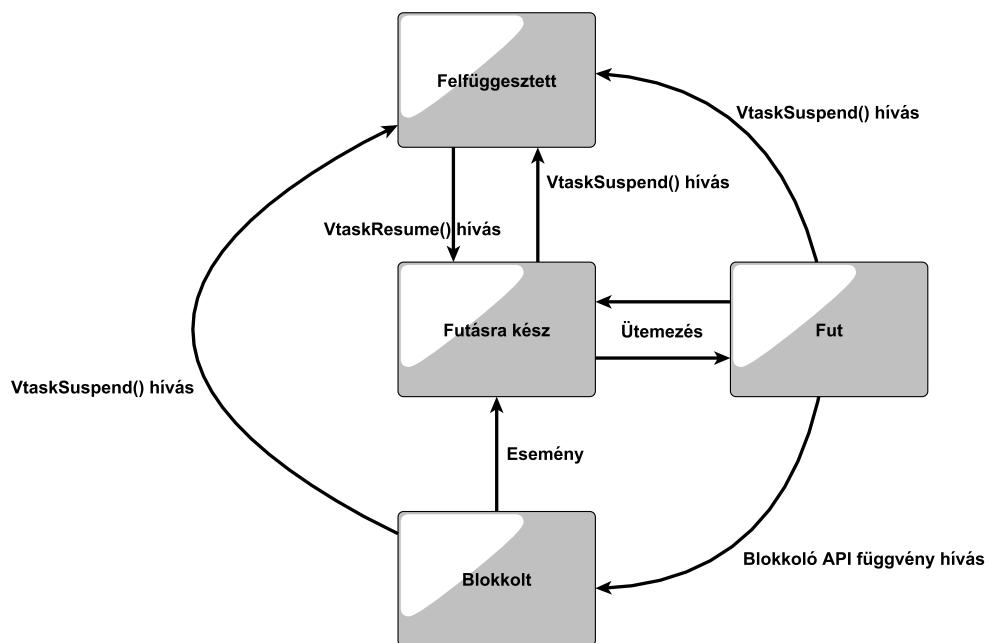
Annak, hogy egy taszk éppen miért nem fut, több oka lehet. Ez alapján a *Nem fut* állapot több állapotra felosztható (2.6. ábra). Ha egy taszk képes lenne futni, de például egy nagyobb prioritású taszk birtokolja a processzort, akkor a taszk állapota *Futásra* kész. Ha a taszk valamilyen eseményre vár (időzítés, másik taszk szinkronizáló jele), akkor a



**2.5. ábra.** Taszk lehetséges állapotai a FreeRTOS rendszerben (egyszerűsített).

taszk *Blokkolt* állapotban van. Az operációs rendszer lehetőséget ad arra, hogy a taszkokat függvényhívással *Felfüggesztett* állapotba kényszerítsük. Ekkor egy másik függvényhívással tudjuk visszahozni az ütemezendő feladatok sorába a taszkot.

A taszkok önként lemondhatnak a futásról (időzítés, szinkronizáció, *taskYIELD()* függvény hívása), viszont futó állapotba csak az ütemező helyezheti.



**2.6. ábra.** Taszk lehetséges állapotai a FreeRTOS rendszerben.

A FreeRTOS által használt ütemezési mechanizmust Fix Prioritásos Preemptív Ütemezésnek hívjuk<sup>1</sup>. *Fix prioritásos*, mivel a rendszer magától nem változtatja a prioritásokat, *preemptív*, mert egy taszk *Futásra kész* állapotba lépésekor preemptálja az éppen futó taszkot, ha a futó taszk prioritása alacsonyabb.

### Idle taszk

A processzor folyamatosan utasításokat hajt végre működése közben (eltekintve a különböző energiatakarékos üzemmódoktól), ezért legalább egy taszknak minden *Futásra kész* állapotban kell lennie. Hogy ez biztosítva legyen, az ütemező indulásakor automatikusan létrejön egy ciklikus *Idle taszk*.

Az *Idle taszk* a legkisebb prioritással rendelkezik, így biztosítva, hogy elhagyja a *Fut*

<sup>1</sup>A FreeRTOS kooperatív ütemezést is támogat, viszont a valósidejű futás eléréséhez a preemptív ütemezés szükséges, ezért a továbbiakban csak a preemptív ütemezéssel foglalkozunk.

állapotot, amint egy magasabb prioritású taszk *Futásra* kész állapotba kerül.

Taszk törlése esetén az Idle taszk végzi el a különböző erőforrások felszabadítását. Mivel a FreeRTOS nem biztosít védelmet egy taszk *kiélezettsével* szemben ezért fontos, hogy az alkalmazás tervezésekor biztosítsunk olyan időszeletet, amikor másik, nagyobb prioritású taszk nem fut.

### **Idle hook függvény**

Előfordulhat, hogy az alkalmazásunkban olyan funkciót szeretnénk megvalósítani, amelyet az Idle taszk minden egyes iterációjára le kell futtatni (például teljesítménymérés érdekében). Ezt a célt szolgálja az *Idle hook* függvény, ami az Idle taszk minden lefutásakor meghívódik.

Az Idle hook általános felhasználása:

- Alacsony prioritású háttérfolyamat, vagy folyamatos feldolgozás,
- A szabad processzoridő mérése (teljesítménymérés),
- Processzor alacsony fogyasztású üzemmódba váltása.

### **Korlátozások az Idle hook függvényrel kapcsolatban**

1. Az Idle hook függvény nem hívhat blokkoló vagyelfüggesztést indító függvényeket<sup>2</sup>,
2. Ha az alkalmazás valahol töröl egy taszkot, akkor az Idle hook függvénynek elfogadható időn belül vissza kell térnie<sup>3</sup>.

#### **2.3.3. Kommunikációs objektumok**

A FreeRTOS három alap kommunikációs struktúrát kínál a felhasználónak:

- sor,
- szemafor,
  - bináris,
  - számláló,
- mutex.

### **Sorok**

A FreeRTOS lehetővé teszi a sor végére és a sor elejére való írást is. A sorba való írás során másolat készül az eredeti változóról, és ez a másolat kerül tárolásra a sorban. Olvasáskor a paraméterként átadott memóriacímre kerül átmásolásra az adat, ezért figyelni kell

---

<sup>2</sup>Az Idle hook függvény blokkolása esetén felléphet az az eset, hogy nincs Futásra kész állapotban lévő taszk.

<sup>3</sup>Az Idle taszk felelős a kernel erőforrások felszabadításáért, ha egy taszk törlésre kerül. Ha az Idle taszk

arra, hogy az átadott változó által elfoglalt memória legalább akkora legyen, mint a sor által tartalmazott adattípus mérete. Amennyiben a továbbítandó adattípus már nagynak tekinthető, akkor érdemes a memóriára mutató pointereket elhelyezni a sorban, ezzel csökkenve a RAM kihasználtságát (ekkor viszont különösen figyelni kell arra, hogy a kijelölt memóriaterület tulajdonosa egyértelmű legyen, vagyis ne történjen különböző taszkokból módosítás egy időben, illetve biztosítani kell a memóriaterület érvényességét). A FreeRTOS API kétféle függvényt használ olvasásra:

- Az egyik automatikusan eltávolítja a kiolvasott elemet a sorból (*xQueueReceive()*),
- A másik kiolvassa a soron következő elemet, de azt nem távolítja el a sorból (*xQueuePeek()*).

A FreeRTOS-ban minden kommunikációs struktúra a sor valamelyen speciális megvalósítása.

A sorok egy taszhöz sem tartoznak, így egy sorba akár több taszk is írhat és olvashat egy alkalmazáson belül.

### Olvasás sorból

Sorból való olvasás során az olvasó függvény paramétereként megadhatunk egy várakozási időtartamot. A taszk maximálisan ennyi ideig várakozik *Blokkolt* állapotban új adat érkezésére, amennyiben a sor üres. Ha ezen időtartam alatt nem érkezik adat, akkor a függvény visszatér, és a visszatérési értékkel jelzi az olvasás sikertelenségét. A *Blokkolt* állapotból a taszk *Futásra kész* állapotba kerül, ha:

- Adat érkezik a sorba a megadott időtartamon belül,
- Nem érkezik adat a sorba, de a megadott várakozási idő lejárt.

Egy sorból egyszerre több taszk is kezdeményezhet olvasást, ezért előfordulhat az az eset, hogy több taszk is *Blokkolt* állapotban várja az adat érkezését. Ebben az esetben csak egy taszk kerülhet *Futásra kész* állapotba az adat érkezésének hatására. A rendszer a legnagyobb prioritású taszkot választja, vagy ha a várakozó taszkok azonos prioritásúak, akkor a legrégebben várakozót helyezi a *Futásra kész* állapotba.

A sorok ún. halmaiba (set) foglalhatóak, így lehetőség nyílik egyszerre több adatra is várakozni. A taszk *Futásra kész* állapotba kerül, amint valamelyik sorba adat érkezik[4].

### Írás sorba

Az olvasáshoz hasonlóan a sorba való írás során is megadható egy várakozási idő. Ha a sorban nincs üres hely az adat írásához, akkor a taszk *Blokkolt* állapotba kerül, amelyből *Futásra kész* állapotba lép, ha:

- Megüresedik egy tároló a sorban,
- Letelik a maximális várakozási idő.

---

bentragad az Idle hook-ban, akkor ez a tisztítás nem tud bekövetkezni.

Egy sorba több taszk is kezdeményezhet írást egyidőben. Ekkor ha több taszk is *Blokkolt* állapotba kerül, akkor hely felszabadulásakor a legnagyobb prioritású taszkot választja a rendszer, azonos prioritású taszkok esetén a legrégebben várakozót helyezi *Futásra* kész állapotba.

## Szemaforok

A FreeRTOS rendszerben implementált szemaforok paraméterei között megadható a maximális tick szám, ameddig várakozhat a szemaforra az adott taszk. Ezen maximális időtartam megadása az esetek nagy részében megoldást jelent a holtponți helyzetekre. A FreeRTOS bináris és számláló szemaforok használatát is támogatja.

## Mutex-ek

FreeRTOS alkalmazása esetén a bináris szemafort általában szinkronizáció céljára használunk, míg a közös erőforrást mutex segítségével védjük. A felhasználásból adódó különbösségek miatt a mutex védett a prioritás inverzió problémájával szemben<sup>4</sup>, míg a bináris szemafon nem.

### 2.3.4. Megszakítás-kezelés

Beágyazott rendszereknél gyakran kell a környezettől származó eseményekre reagálni (például adat érkezése valamely kommunikációs interfészen). Az ilyen események kezelésekor a megszakítások alkalmazása gyakran elengedhetetlen.

Megszakítás használata esetén figyelni kell arra, hogy a megszakítási rutinokban csak *FromISR*-re végződő API függvényeket hívhatunk. Ellenkező esetben nem várt működés következhet be (blokkoljuk a megszakítási rutint, ami az alkalmazás fagyásához vezethet; kontextus-váltást okozunk, amiből nem térünk vissza, így a megszakítási rutinból sosem lépünk ki, stb.).

A FreeRTOS ütemezője a (STM32-re épülő rendszerekben) a SysTick eseményt használja az ütemező periodikus futtatásához. A megszakítási rutin futása közben emiatt nem történik ütemezés. Amennyiben valamely magasabb prioritású taszkunk a megszakítás hatására *Futásra* kész állapotba kerül, akkor vagy a következő ütemezéskor kapja meg a processzort, vagy explicit függvényhívással kell kérni az operációs rendszert az ütemező futtatására.

Az alacsonyabb prioritású megszakítások szintén nem tudnak érvényre jutni, így azok bekövetkezéséről nem kapunk értesítést (az első beérkező, alacsonyabb prioritású megszakítás jelző bitje bebillen az esemény hatására, de amennyiben több is érkezik a magasabb prioritású megszakítási rutin futása alatt, úgy azok elvesznek). Az említett problémák végett a megszakítási rutint a lehető legrövidebb idő alatt be kell fejezni.

---

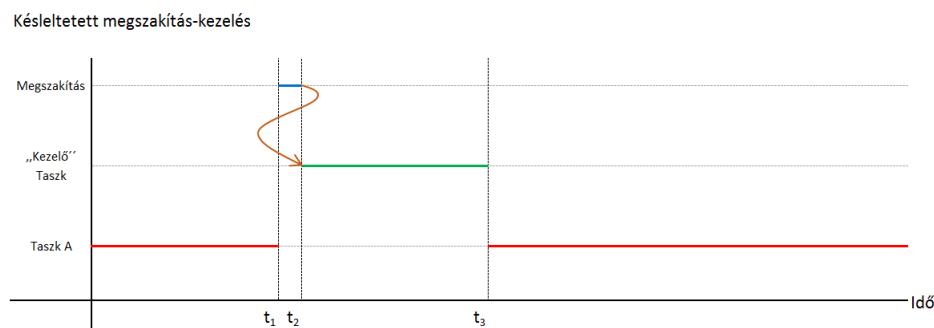
<sup>4</sup>A FreeRTOS prioritás öröklési mechanizmusa csak egyszerű implementációt tartalmaz, és feltételezi, hogy egy taszk csak egy mutex-öt birtokol egy adott pillanatban.

## Késleltetett megszakítás-kezelés

A megszakítási rutint a lehető legrövidebb idő alatt el kell hagyni, emiatt célszerű a kevésbé fontos műveleteket egy kezelő taszkban megvalósítani. A FreeRTOS a szemaforokon keresztül biztosít lehetőséget a megszakítás és taszk szinkronizációjára.

A megszakítás hatására a megszakítási rutinban csak a szükséges lépéseket végezzük el (például eseményjelző bitek törlése), majd egy szemaforon keresztül jelezük a feldolgozó tasznak az esemény bekövetkeztét. Ha a feldolgozás időkritikus, akkor a feldolgozó szálhoz rendelt magas prioritással biztosítható, hogy az éppen futó taszkot preemptálja. Ekkor a megszakítási rutin végén az ütemező meghívásával a visszatérés után azonnal feldolgozásra kerül az esemény.

Az eseményt kezelő taszk blokkoló *take* utasítással várakozik *Blokkolt* állapotban a szemafor érkezésére. A jelzés hatására *Futásra* kész állapotba kerül, ahonnan az ütemező (az éppen futó taszkot preemptálva) *Fut* állapotba mozgatja. Az esemény feldolgozását követően újra meghívja a blokkoló *take* utasítást, így újból *Blokkolt* állapotba kerül az újabb esemény bekövetkezéséig. A késleltetett megszakítás-kezelés elvét a 2.7. ábrán láthatjuk.



2.7. ábra. Megrakítások késleltetett feldolgozásának szemléltetése.

## Megrakítások egymásba ágyazása

Az STM32 mikrokontrollerek lehetővé teszik prioritások hozzárendelését a megszakításokhoz. Ezáltal létrejöhet olyan állapot, amikor egy magasabb prioritású megszakítás érkezik egy alacsonyabb szintű megszakítási rutin futása közben. Ekkor a magasabb prioritású megszakítás késleltetés nélkül érvényre jut, és a magasabb prioritású megszakítási rutin befejeztével az alacsony prioritású folytatódik.

A FreeRTOS rendszer konfigurációs fájljában (*FreeRTOSConfig.h*) beállítható azon legmagasabb prioritás, amihez a FreeRTOS hozzáférhet. Ezáltal a megszakításoknak két csoportja adódik:

- A FreeRTOS által kezelt megszakítások,
- A FreeRTOS-tól teljesen független megszakítások.

A FreeRTOS által kezelt prioritások kritikus szakaszba lépéskor letiltásra kerülnek, így azok nem szakíthatják meg az atomi utasításként futtatandó kódrészletet. Viszont a megszakítások ezen csoportja használhatja a *FromISR*-re végződő FreeRTOS API függvényeket.

A FreeRTOS-tól független megszakítások kezelése teljes mértékben a fejlesztő feladata. Az operációs rendszer nem tudja megakadályozni az érvényre jutásukat, így a kritikus szakaszban is képesek futni. A kiemelkedően időkritikus kódrészleteket célszerű ezekben a megszakítási rutinokban megvalósítani. A FreeRTOS-tól független megszakítási rutinokban tilos API függvényeket használni!

### 2.3.5. Erőforrás-kezelés

Multitaszk rendszerek esetén fennáll a lehetősége, hogy egy taszk kikerül a *Fut* állapotból még mielőtt befejezné egy erőforrással a műveleteket. Ha az erőforrást egy másik taszk is használni akarja, akkor inkonzisztencia léphet fel. Tipikus megjelenései a problémának:

- Periféria elérése,
- Egy közös adat olvasása, módosítása, majd visszaírása<sup>5</sup>,
- Változó nem atomi elérése (például több tagú struktúra értékeinek megváltoztatása),
- Nem reentráns függvények,

Az adat inkonzisztencia elkerüléséhez használhatunk mutex-et. Amikor egy taszk megkapja egy erőforrás kezelésének jogát, akkor más taszk nem férhet hozzá, egészen addig, amíg a birtokló taszk be nem fejezte az erőforrással a feladatát, és az erőforrás felszabadítását mutex-en keresztül nem jelezte.

A FreeRTOS támogatja a kritikus szakaszok használatát a *taskENTER\_CRITICAL()* és *taskEXIT\_CRITICAL()* makrók használatával.

A kritikus szakaszokat minden probléma nélkül egymásba lehet ágyazni, mivel a rendszerkernel nyilvántartja, hogy milyen mélyen van az alkalmazás a kritikus szakaszokban. A rendszer csak akkor hagyja el a kritikus szakaszt, ha a számláló nullára csökken, vagyis ha minden *taskENTER\_CRITICAL()* híváshoz tartozik egy *taskEXIT\_CRITICAL()* is.

A kritikus szakaszt a lehető leggyorsabban el kell hagyni, különben a beérkező megszakítások válaszideje nagy mértékben megnőhet.

A kritikus szakasz megvalósításának egy kevésbé drasztikus módja az ütemező letiltása. Ekkor a kódrészlet védett a más taszkok általi preemptálástól, viszont a megszakítások nem kerülnek letiltásra. Hátránya, hogy az ütemező elindítása hosszabb időt vehet igénybe.

### Gatekeeper taszk

A gatekeeper taszk alkalmazása a kölcsönös kizáras egy olyan megvalósítása, mely működésénél fogva védt a prioritás inverzió és a holtpont kialakulásával szemben.

A gatekeeper taszk egyedüli birtokosa egy erőforrásnak, így csak a taszk jogosult a közvetlen elérésre. A többi taszk közvetetten, a gatekeeper taszk szolgáltatásain keresztül tudja használni az erőforrást.

Amikor egy taszk használni akarja az erőforrást, akkor üzenetet küld a gatekeeper taszk-

---

<sup>5</sup>Magas szintű programozási nyelv használata esetén (pl. C) ez látszólag lehet egy utasítás, viszont a fordító által előállított gépi kód több utasításból állhat.

nak (általában sor használatával). Mivel egyedül a gatekeeper taszk jogosult elérni az erőforrást, ezért nincs szükség explicit mutex használatára.

A gatekeeper taszk Blokkolt állapotban vár, amíg nem érkezik üzenet a sorba. Az üzenet beérkezése után elvégzi a megfelelő műveleteket az erőforráson, majd ha kiürült a sor, akkor ismét Blokkolt állapotba kerül.

A megszakítások probléma nélkül tudják használni a gatekeeper taszkok szolgáltatásait, mivel a sorba való írás támogatott megszakítási rutinból is.

### 2.3.6. Memória-kezelés

Beágazott alkalmazások fejlesztése során is szükség van dinamikus memória foglalásra. Az asztali alkalmazásoknál megszokott *malloc()* és *calloc()* függvények több szempontból sem felelnek meg mikrokontrolleres alkalmazásokban:

- Kisebb rendszerekben nem biztos, hogy elérhető,
- Az implementációjuk sok helyet foglalhat,
- Nem determinisztikus a lefutásuk; a végrehajtási idő változhat különböző híváskor,
- Memóriatöredezettség léphet fel.

Az egyes alkalmazások különböznek memória-allocációs igényükben és az előírt időzítési korlátokban, ezért nincs olyan memória-allocációs séma, amely minden alkalmazásban megállna a helyét. A FreeRTOS több allocációs algoritmust is a fejlesztő rendelkezésére bocsát, amiből az alkalmazásnak megfelelő kiválasztásával lehet elérni a megfelelő működést.

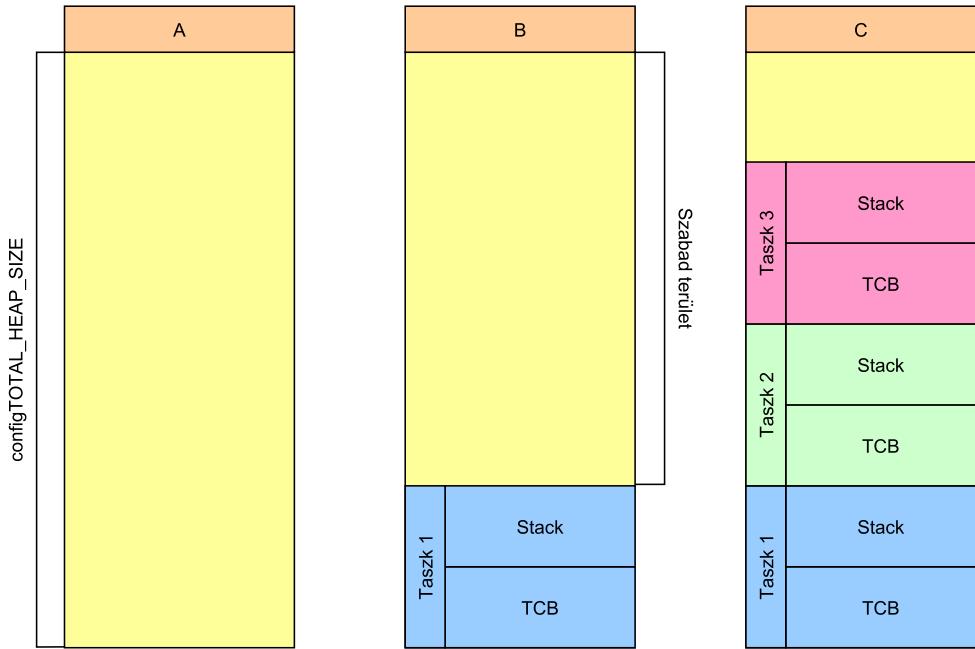
#### Heap\_1.c

Kisebb beágazott alkalmazásoknál gyakran még az ütemező indulása előtt létrehozunk minden taszkot és kommunikációs objektumot. Ilyenkor elég a memóriát lefoglalni az alkalmazás indulása előtt. Ez azt is jelenti, hogy nem kell komplex algoritmusokat megvalósítani a determinisztikusság biztosítására és a memóriatöredezettség elkerülésére, hanem elég a kódmeretet és az egyszerűséget szem előtt tartani.

Ezt az implementációt tartalmazza a *heap\_1.c*. A fájl a *pvPortMalloc()* egyszerű megvalósítását tartalmazza, azonban a *pvPortFree()* nincs implementálva. A *heap\_1.c* nem fenyegeti a rendszer determinisztikusságát.

A *pvPortMalloc()* függvény a FreeRTOS heap-jét osztja fel kisebb területekre, majd ezeket rendeli hozzá az egyes taszkokhoz. A heap teljes méretét a *configTOTAL\_HEAP\_SIZE* konfigurációs érték határozza meg a *FreeRTOSConfig.h* fájlban. Nagy méretű tömböt definiálva már a memóriafoglalás előtt látszólag sok memóriát fog felhasználni az alkalmazás, mivel a FreeRTOS ezt induláskor lefoglalja.

A 2.8. ábrán láthatjuk az induláskor rendelkezésre álló üres heap-et, illetve különböző számú taszk esetén a heap használatát.



**2.8. ábra.** A *heap\_1.c* implementációjának működése.

### Heap\_2.c

A *heap\_2.c* szintén a *configTOTAL\_HEAP\_SIZE* konfigurációs értéket használja, viszont a *pvPortMalloc()* mellett már implementálva van a *pvPortFree()* is. A legjobban illeszkedő (best fit) algoritmus biztosítja, hogy a memóriakérés a hozzá méretben legközelebb eső, elegendő nagyságú blokkból legyen kiszolgálva. Fontos megjegyezni, hogy a megvalósítás nem egyesíti a szomszédos szabad területeket egy nagyobb egységes blokkba, így töredzettség léphet fel<sup>6</sup>. Ez nem okoz gondot, ha a lefoglalt és felszabadított memória mérete nem változik.

A *heap\_2.c* fájl használata javasolt, ha az alkalmazás ismételve létrehoz és töröl taszkokat, és a taszkokhoz tartozó stack mérete nem változik (2.9. ábra).

A *heap\_2.c* működése nem determinisztikus, de hatékonyabb, mint a *standard library* implementációi.

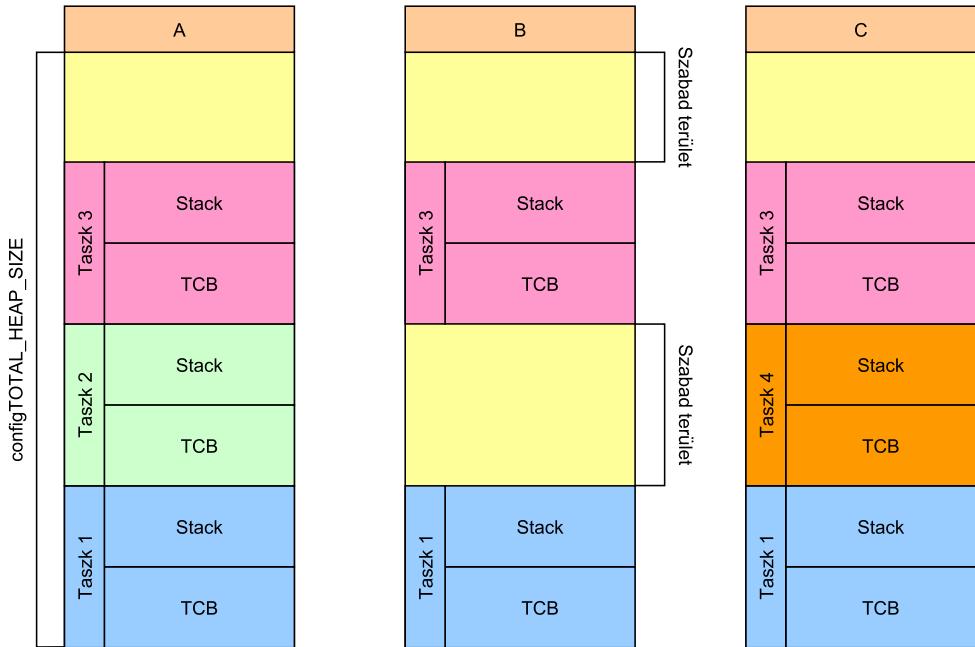
### Heap\_3.c

A *heap\_3.c* a *standard library* függvényeit használja, de a függvények alatt felfüggeszti az ütemező működését, ezzel elérve, hogy a memória-kezelés thread-safe legyen.

A heap méretét nem befolyásolja a *configTOTAL\_HEAP\_SIZE* érték, ehelyett a linker beállításai határozza meg.

---

<sup>6</sup>A *heap\_4*. megvalósítás hasonló a *heap\_2.c* fájlban található megvalósításokkal, viszont véde a memória-töredzettség ellen.



2.9. ábra. A *heap\_2.c* implementációjának működése.

## 2.4. μC/OS-III

A fejezet a [15] irodalom alapján készült.

### 2.4.1. Ismertető

A μC/OS-III valósidejű operációs rendszert a Jean Labrosse és Christian Légaré által alapított *Micrium* fejleszti. Az operációs rendszer oktatási célra ingyenesen elérhető, mely magában foglalja többek között a rendszer forráskódját és részletes dokumentációját is.

A vállalat kiemelt figyelmet fordít a forráskód minőségére, ezért szigorú szabályokat felállítva fejlesztették a rendszer. Ezen szabályok kiterjednek az egyes függvények és struktúrák nevére, és a függvényparaméterek sorrendjére is.

A részletes dokumentáció mellett – mely oldalakon keresztül ismerteti a rendszer könyvtárának struktúráját és tartalmát; külön fejezet foglalkozik a portolás lépéseiinek magyarázatával – a forráskódban található megjegyzések is segítik a fejlesztőt a programozás során.

Az operációs rendszer mellett a vállalat további modulokat is elérhetővé tesz, mellyel a rendszer funkcionalitását bővíti. Ilyen modul például a μC/TCP-IP, a μC/USB és a μC/FS.

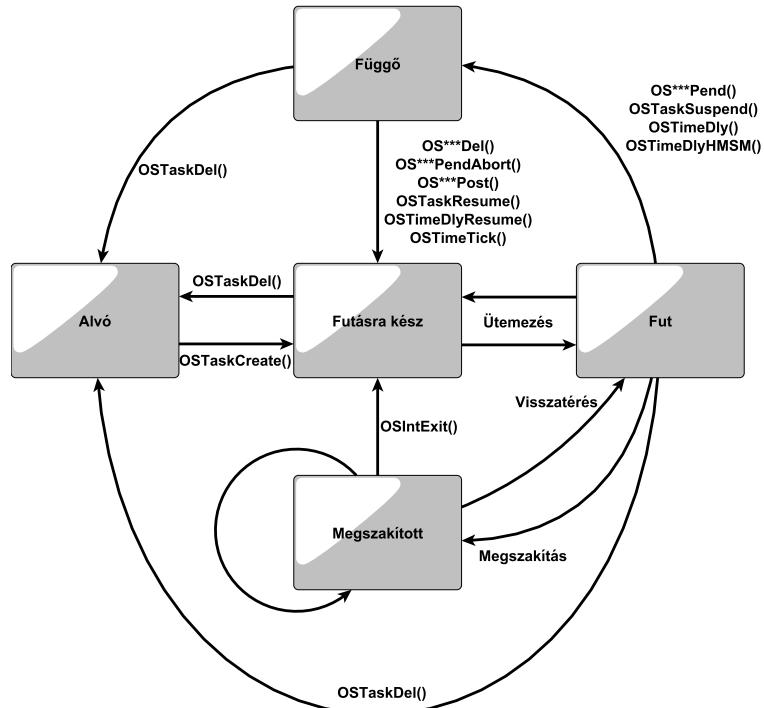
### 2.4.2. Taszkok

A μC/OS előző verziójában a prioritások száma korlátozott volt és minden prioritási szint-hez maximum egy taszk tartozhatott, ezáltal korlátozva a taszkok számát is. Az igényeket kielégítve ezt a korlátozást a rendszer harmadik verziójában eltörölték. Nincs megkötés sem a prioritásokra, sem a taszkok számára. Korlátot egyedül a használt mikrokontroller

erőforrásai jelentenek.

A preemptív ütemezés mellett implementálásra került a Round-Robin ütemezés, amely lehetővé teszi az azonos prioritású taszkok között a processzoridő periodikus kiosztását. A Round-Robin ütemezést fordítási időben engedélyezni kell, majd ezután futási időben szabadon engedélyezhetjük vagy letilthatjuk.

A  $\mu$ C/OS-III taszkjainak állapotmodellje kis mértékben különbözik a FreeRTOS esetében megismerttől. Nem különbözteti meg a *Felfüggesztett* és *Blokkolt* állapotokat, viszont a törölt (vagy még el nem indított) taszkokat *Alvó* állapotba sorolja. Ezen felül bevezeti a *Megszakított* állapotot, melybe megszakítás érkezésekor kerül a taszk. Az állapotmodell a 2.10. ábrán látható.



**2.10. ábra.** Taszk lehetséges állapotai a  $\mu$ C/OS-III rendszerben.

A taszkok nyílvántartása a  $\mu$ C/OS-III esetében is TCB használatával történik. A TCB struktúra által tartalmazott változók száma meglehetősen nagy, viszont ezért több információt is tárol az egyes taszkokról. A  $\mu$ C/OS-III által alkalmazott TCB főbb változói a 2.2. táblázatban láthatóak.

A FreeRTOS-hoz hasonlóan a  $\mu$ C/OS-III is támogatja a hook függvények használatát, melyekkel a rendszer funkcionálisát bővíthetjük.

Az Idle taszk is megtalálható, mely a legalacsonyabb prioritási szinten jön létre a rendszer inizializálásakor<sup>7</sup>. Ellentétben a FreeRTOS-sal, itt az Idle taszk nem végez hasznos tevékenységet, csupán számlálók értékét növeli, illetve az Idle hook függvényt hívja meg.

Az operációs rendszer beépített támogatással rendelkezik a különböző terhelési jellemzők mérésére, Amennyiben fordítási időben azt engedélyezzük, úgy információt kaphatunk a

<sup>7</sup>A  $\mu$ C/OS-III csak az Idle taszkot engedi a legalacsonyabb prioritási szinten futni, a fejlesztő által létrehozott taszkoknak mindenkorábban magasabb prioritással kell rendelkeznie!

**2.2. táblázat.** A  $\mu$ C/OS-III TCB-jének főbb változói.

Változó	Jelentés
StkPtr	A stack utolsó elemére mutató pointer.
StkLimitPtr	A stack-en belül egy adott részre mutat. A stack túlcordulásának ellenőrzésére használja a rendszer.
NextPtr és PrevPtr	Ezen pointerek használatosak a futásra kész taszkok TCB-jének kétszeresen láncolt listájának felépítésére.
NamePtr	A taszhöz rendelt név.
StkBasePtr	A taszk stack-jének alapcímére mutató pointer.
TaskEntryAddr	A taszk belépési pontja.
TaskEntryArg	A taszk indulásakor a taszk paramétereit tartalmazza.
PendDataTblPtr	Azon objektumok táblázatára mutató pointer, melyre a taszk éppen várakozik.
TaskState	A taszk aktuális állapota.
Prio	A taszk aktuális prioritása.
StkSize	A stack mérete.
SemCtr	A taszhöz rendelt beépített szemafor számlálója.
TickRemain	A várakozó taszk hátralevő várakozási ideje.
TimeQuanta és TimeQuantaCtr	Round-Robin ütemezés esetén a taszk számára kiosztott processzoridő és a hátralevő időszekletek száma.
MsgPtr és MsgSize	Közvetlenül a taszknak küldött üzenetre mutató pointer és az üzenet mérete.

processzor kihasználtságáról, a taszkonkénti processzor-terhelésről és a taszkonkénti stack használatról is.

#### 2.4.3. Kommunikációs objektumok

Az alap kommunikációs objektumok – mint szemafor, mutex, sor – mellett a rendszer lehetőséget teremt eseményjelző bitek használatára, taszkok közötti jelzés küldésére szemafor használata nélkül, illetve taszkok közötti üzenet küldésére sor létrehozása nélkül. Az objektumok használatakor függvény-paraméterként megadható, hogy az ütemező lefusson-e a függvényhívás hatására, vagy sem. Ez a lehetőség tipikusan akkor használható, ha egyszerre több objektumok használunk (például több különböző sorba helyezünk üzenetet, majd szemafort is felszabadítunk), és nem szeretnénk, hogy közben a taszk másik taszk által megszakításra kerüljön.

#### Sorok

Egy üzenet a küldött változóra, adatstruktúrára vagy akár függvényre mutató pointert, az adat méretét és az üzenet küldésének időpontját tartalmazó bélyegzőt foglalja magában.

Mivel az üzenet nem tartalmazza a tárolt adat típusát, ezért a fogadó taszknak ismernie kell a várt üzenet feldolgozásának módját. A sor több ilyen üzenet láncolt listáját jelenti.

A  $\mu$ C/OS-III-ban megvalósított sorok esetén az adat nem kerül másolásra, csupán az adat memóriacíme és merete jut el a fogadóhoz, ezért biztosítani kell, hogy az üzenet tartalma a feldolgozás befejeztéig változatlan maradjon. Ennek egyik lehetséges megvalósítása, hogy a küldő dinamikusan lefoglalható memóriaszeletet kér az operációs rendszertől, majd ezen memóriaterületre másolja a küldeni kívánt adatot. A fogadó oldalon, miután az adat feldolgozása befejeződött, a memóriaterület felszabadításra kerül.

### Olvasás sorból

A  $\mu$ C/OS-III esetében is megadható az a maximális időtartam, ameddig a taszk várakozik az üzenet beérkezésére. Ha a megadott időn belül nem érkezik adat, akkor a függvény paramétereként átadott változóban hibakód jelzi az olvasás sikertelenségét.

Amennyiben több taszk is várakozik ugyanazon sorba érkező adatra, úgy a legnagyobb prioritással rendelkező taszk jogosult az adat elvételére. Ha az adatot valamilyen okból minden várakozó taszhöz el kell juttatni, akkor a küldő küldhet broadcast üzenetet.

### Taszknak küldött üzenet

Általános alkalmazások esetén ritkán fordul elő, hogy egy sort több taszk is figyelne. Ezért a  $\mu$ C/OS-III-ban minden taszk rendelkezik saját üzenetsorral, melybe a többi taszk a megszokott módon helyezhet adatot. Ekkor nem szükséges külön objektumot létrehozni az üzenet továbbítására, csupán a fogadó taszk TCB-jének ismerete szükséges.

A célzott üzenetek használata amellett, hogy átláthatóbb kódot eredményez, az üzenetküldés folyamatát is hatékonyabban valósítja meg.

### Szemaforok

A rendszer támogatja a bináris és a számláló szemaforok használatát egyaránt. A szemafor létrehozásakor megadhatjuk annak inicializálási értékét.

Minden taszk rendelkezik saját beépített szemaforral, melynek segítségével a taszkok közötti szinkronizáció hatékonyabban megoldható, és külön objektum létrehozására sincs szükség.

### Mutex-ek

A szemafor mellett a mutex is elérhető. Egy taszk többször is lefoglalhatja az általa birtokolt mutex-et (maximum 250-szer), viszont ugyanannyiszor el is kell engednie azt. A mutex védett a prioritás inverzió jelenségével szemben.

### Eseményjelző bitek (Event flag)

Az eseményjelző bitek akkor bizonyulnak hasznosnak, ha egy taszk több esemény bekövetkezésére várakozik. Az eseményre való várakozás megvalósulhat diszjunktív módon, amikor

bármely esemény bekövetkezése kielégíti a feltételt (logikai VAGY), vagy megvalósulhat konjunktív módon, amikor minden esemény bekövetkezése szükséges (logikai ÉS). Az egyes jelző bitek jelentése teljes mértékben a fejlesztőre van bízva, azt az operációs rendszer nem korlátozza.

#### 2.4.4. Megszakítás-kezelés

A *os\_cfg.h* konfigurációs fájlban található *OS\_CFG\_ISR\_POST\_DEFERRED\_EN* makró segítségével kétféle megszakítás-kezelési módszer között válthatunk: közvetlen módszer és késleltetett módszer.

Közvetlen módszer (*Direct Method*) esetén a kritikus szakaszokban a megszakítások letiltásra kerülnek, és egy közben beérkező esemény csak az újból engedélyezés után jut érvényre. Ez a megvalósítás volt jelen az operációs rendszer előző verziójában is.

Késleltetett módszer (*Deferred Method*) esetén a kritikus szakaszokban a megszakítások nem kerülnek letiltásra, csak az ütemezőt függesszti fel az operációs rendszer. A módszer előnye, hogy a megszakítások csak nagyon kevés időre vannak letiltott állapotban, így sűrűn bekövetkező események sem vesznek el. A mechanizmus mögötti gondolat, hogy a megszakítások által generált *Post* utasítások egy úgynevezett *Interrupt Queue*-ba kerülnek, amit az *Interrupt Queue Handler* taszk dolgoz fel. Ezen taszk a legmagasabb elérhető prioritással rendelkezik, ezért a megszakítási rutin befejeztével rögtön *Futó* állapotba kerül. A *Post* metódus és paramétereinek bemásolása a sorba, illetve kimásolása a sorból processzoridőt igényel. A módszer komplexitása és a szükséges extra processzor-használat hátrányt jelenthet.

#### 2.4.5. Erőforrás-kezelés

A korábban bemutatott szemafor és mutex mellett az erőforrások védelmére használhatunk kritikus szakaszokat is. Ahogy azt a *Megszakítás-kezelés* részben ismertettem, a µC/OS a kritikus szakaszokban beállítástól függően vagy letiltja a megszakításokat, vagy csak az ütemezőt függesszti fel. Fontos megjegyezni, hogy amennyiben megszakítási rutinnal szeretnénk közös memóriaterületet (változót, adatstruktúrát) használni, úgy csak a megszakítások letiltása nyújt védelmet az adat hibás olvasásának lehetősége ellen.

#### 2.4.6. Memória-kezelés

A µC/OS-III a *malloc()* és *free()* utasítások használata helyett a fix méretű memóriarekeszek használatát javasolja dinamikus memória foglalás céljára. Erre a megoldásra beépített struktúrát nyújt a fejlesztő számára, ami folytonos memóriaterületen azonos méretű blokkokat foglal le, és ezen blokkokat lehet igényelni a rendszertől.

Egy alkalmazáson belül több, különböző méretű és számú blokkokat tartalmazó memóriaterület is létrehozható. A memóriaterület lefoglalásakor meg kell adni a blokkok méretét és számát. A foglaláshoz használhatjuk a tömböknél megszokott módon történő foglalást, de akár a *malloc()* függvényt is (amennyiben az nem kerül felszabadításra a későbbieken), hiszen jól megtervezett szoftver esetén ez csak a program indulásakor fut le.

Az operációs rendszertől *OSMemGet()* függvényhívással kérhetünk egy szeletet a területről, míg az *OSMemPut()* függvény segítségével adhatjuk azt vissza a rendszer számára.

A fix méretű memóriaterület használatához az *os\_cfg.h* fájlban az *OS\_CFG\_MEM\_EN* makrót kell 1 értékre állítani.

## 2.5. Raspbian

A Raspbian egy Debian-ra épülő, kifejezetten Raspberry Pi-re optimalizált linux disztribúció. Az operációs rendszer alap programokat és eszközöket tartalmaz, melyek a Raspberry Pi használatához szükségesek. Több, mint 35 000 csomag elérhető a rendszerhez, és folyamatos fejlesztés alatt áll.

Tekintve, hogy egy teljes értékű linux vált elérhetővé az eszközhöz, az asztali számítógépeknél megsokkott módon használható a rendszer. Ebbe bele tartozik a fejlesztésre elérhető szoftvercsomagok, fejlesztőkörnyezetek és egyéb alkalmazások is. Amennyiben a használni kívánt csomag nem elérhető a platformra, akkor a fejlesztő a forráskód birtokában a megfelelő fordítót használva lefordíthatja magának. Ezt elvégezheti magán a Raspberry Pi-n (natív fordítás), vagy akár egy asztali számítógépen a szükséges beállítások után ún. *cross-compile* alkalmazásával.

## 2.6. Windows 10 IoT Core

A Windows 10 IoT Core a Windows 10 kisebb erőforrással rendelkező eszközökre optimalizált változata. Az x86/x64 alapú rendszerek mellett az ARM mikroprocesszorokat tartalmazó eszközökön is futtatható. Szoftver fejlesztéséhez az *Universal Windows Platform* (UWP) API használható.

Az UWP API széleskörű támogatással rendelkezik a perifériák terén, a különböző Microsoft szolgáltatások (mint például az *Azure*) szintén használhatóak.

Sem a Raspbian-hoz, sem a Windows 10 IoT Core-hoz nem érhető el részletes dokumentáció. Ez a Raspbian esetén magyarázható azzal, hogy egy elterjedt linux disztibúciót használ alapjául, ezáltal a rendszer működése nem különbözik nagy mértékben attól. A Windows 10 IoT Core esetében a hivatalos oldalon elérhető *Dokumentáció* szekció, de ott pár darab promóciós videót találunk a rendszerről.

Mindkettő rendszerről elmondható, hogy nem hard real-time alkalmazásokhoz fejlesztik<sup>8</sup>.

Az elérhető kommunikációs objektumok listáját a használt programozási nyelv, és az ahhoz elérhető könyvtárakban implementált objektumok határozzák meg. A manapság használt nyelvek többségében (pl. C#, C++ a megfelelő könyvtárak alkalmazásával) az alap objektumok elérhetőek.

---

<sup>8</sup>Raspbian-hoz elérhetők kernel patch-ek, melyekkel a real-time működés megvalósítható.

### 3. fejezet

## Teljesítménymérő metrikák

Asztali alkalmazás fejlesztésekor a rendelkezésre álló teljesítmény és tárhely ma már nem számít akadálynak. Ha valamelyik erőforrás szűk keresztnetszetté válik, akkor alkatrészcserével általában megszüntethető a probléma.

Nem ez a helyzet beágyazott rendszerek esetén. A rendszer központi egységének számítási kapacitása általában nem haladja meg nagy mértékben az elégséges szintet, így különösen figyelni kell a fejlesztés során, hogy az implementált kód hatékony legyen[16]. A rendelkezésre álló memória sem tekinthető korlátlanak, és gyakran a bővítés is nehézkes, esetleg nem megoldható. Az eszköz fogyasztása is fontos szempont, amit a szoftver szintén nagy mértékben befolyásolhat.

Az operációs rendszer választása összetett folyamattá is bonyolódhat, mert mérlegelni kell az alkalmazásunk igényeit, az operációs rendszer támogatottságát (támogatott mikrokontrollerek, fórumok, gyártói támogatás), a becsülhető fejlesztési időt és az ezzel járó költségeket, illetve fizetős operációs rendszer esetén a rendszer árát.

A választást az sem segíti előre, hogy nincs egyértelmű módszer az operációs rendszerek értékelésére<sup>1</sup>.

Az alkalmazott mérési folyamatnak több szempontnak is eleget kell tennie, hogy az eredmény használható legyen. Egy mérés során több forrásból is eredhet hiba, melyek mértékét szeretnénk a lehető legkisebb szintre csökkenteni. A kulcsfontosságú szempontok az alábbiak[18, 19]:

- **Megismételhetőség:** egy mérésnek megismételhetőnek kell lennie. Ehhez szükséges a pontos mérési összeállítás, a mérés körülményei, a használt eszközök és szoftverek.
- **Véletlenszerűség:** a mérés során nem független események követhetik egymást, amik a mérés eredményét befolyásolják. Ezeket csak ritkán lehet teljes mértékben kiküszöbölni, ezért törekedni kell a mérési folyamatok véletlenszerű futtatására (például méréssorozat esetén az egyes folyamatok ne minden ugyan abban a sorrendben fussenek le).

<sup>1</sup>Bár a Német Szabványügyi Intézet (Deutsche Institut für Normung - DIN) az 1980-as évek végén hozott létre szabványt a folyamatirányító számítógépes rendszerek teljesítménymutatóinak mérésére (DIN 19242 szabványsorozat), a valósidejű operációs rendszerek értékelésére ez nem jelent megoldást[17].

- **Vezérlés:** a mérés során a vezérelhető paramétereket (melyek a mérést befolyásolhatják) lehetőségeinkhez mérten kézben kell tartani.
- **Szemléletesség:** a mérés eredményének reprezentatívnak kell lennie. Számértékek esetén két mérés eredményét össze kell tudnunk hasonlítani és tudnunk kell relációt vonni a két érték közé.

A továbbiakban különböző forrásokból vett szempontokat vizsgálok meg, majd azok alapján állítom fel a dolgozat során megfigyelt tulajdonságok listáját.

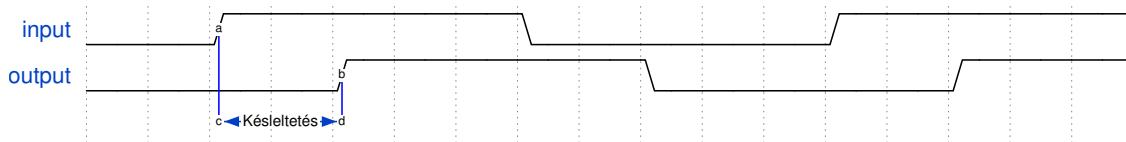
### 3.1. Szakirodalmakban fellelhető metrikák

#### 3.1.1. Memóriaigény

A mikrokontrollerek területén a memória mérete korlátozott (ROM és RAM egyaránt), ezért fontos, hogy a használt rendszer minél kisebb lenyomattal rendelkezzen[16].

#### 3.1.2. Késleltetés

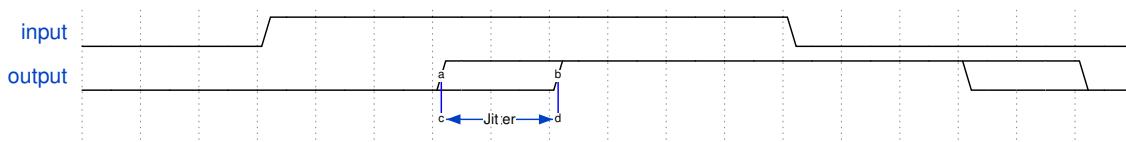
A rendszer késleltetése az az idő, ami egy esemény beérkezésétől a rendszer válaszáig eltelik[20]. Ezt okozhatja a mikrovezérlő megszakítási mechanizmusához szükséges műveletek sora, az operációs rendszer ütemezőjének overhead-je, de a közben végrehajtandó feladat is nagy mértékben befolyásolja a nagyságát.



**3.1. ábra.** Az operációs rendszer késleltetésének szemléltetése.

#### 3.1.3. Jitter

A jitter egy folyamat vizsgálata során a többszöri bekövetkezés után mért késleltetésekkel határozható meg.



**3.2. ábra.** A késleltetés jitterének szemléltetése.

#### 3.1.4. Rheatstone

1989-ben a Dr. Dobbs Journal cikkeként jelent meg egy javaslat, ami a valósidejű rendszerek objektív értékelését célozta meg. Rabindra P. Kar, az Intel Systems Group senior mérnöke ismertette a módszer előnyeit és szempontjait, melynek a Rheatstone nevet adta<sup>2</sup>.

<sup>2</sup>A név a Whetstone és Dhystone módszerek elnevezéseit követve, mint szójáték ered.

A cikk megjelenésekor már léteztek teljesítménymérő metódusok (például Whetstone, Dhrystone), de ezek a fordító által generált kódot, illetve a hardvert minősítették. A Rheatstone metrika célja, hogy a fejlesztőket segítse az alkalmazásukhoz leginkább megfelelő operációs rendszer kiválasztásában[21].

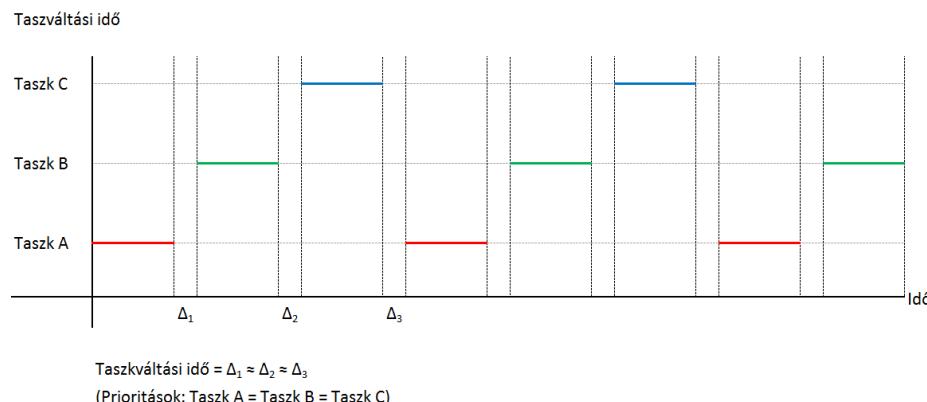
A Rheatstone hat kategóriában vizsgálja meg az operációs rendszer képességeit:

- Taszkváltási idő (Task switching time),
- Preemptálási idő (Preemption time),
- Megszakítás-késleltetési idő (Interrupt latency time),
- Szemafor-váltási idő (Semaphore shuffling time),
- Deadlock-feloldási idő<sup>3</sup> (Deadlock breaking time),
- Datagram-átviteli idő<sup>4</sup> (Datagram throughput time).

1990-ben megjelent egy második cikk is, amelyben amellett, hogy az egyes kategóriák példaprogramjait közölték (iRMX operációs rendszerhez), pár kategória meghatározását megváltoztatták[22]. Ezen változtatásokat az adott kategória részletezésekor ismertetem.

### Taszkváltási idő

A taszkváltási idő a két független, futásra kész, azonos prioritású taszkok váltásához szükséges átlagos időtartam.



**3.3. ábra.** A taszkváltási idő szemléltetése.

A taszkváltási idő alapvető jellemzője egy multitaszk rendszernek. A mérés a taszkokat nyilvántartó struktúrák hatékonyságáról ad képet. A taszkváltási időt a használt processzor architektúrája, utasításkészlete is befolyásolja[21].

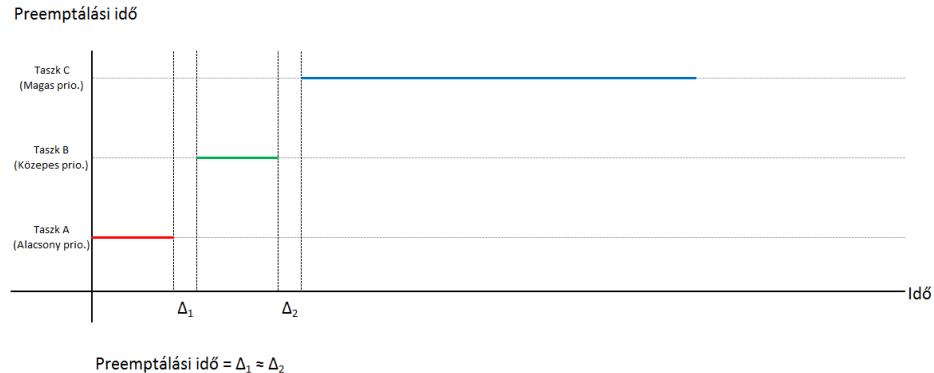
A rendszerek a futtatható taszkokat általában valamilyen listában tárolják, így különböző számú taszkkal elvégezve a mérést más eredményt kaphatunk[21].

<sup>3</sup>A vizsgálat során nem alakul ki tényleges holtpont. A mérés a prioritás-inverzió jelenségét szimulálja, de az eredeti forrás tiszteletére megtartottam a terminológiát.

<sup>4</sup>Bár a kategória neve időnek nevezi a mért mennyiséget, az eredeti dokumentum alapján a mérés

## Preemptálási idő

A preemptálási idő egy magasabb prioritású taszk érvényre jutásához szükséges átlagos időtartam[21].

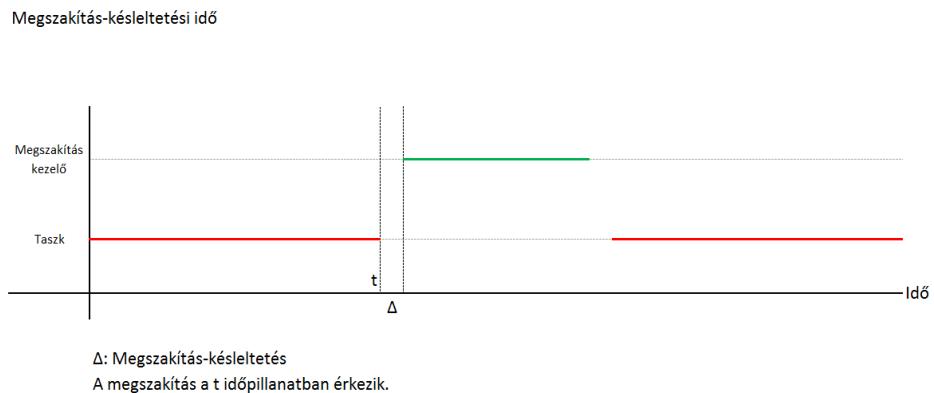


**3.4. ábra.** A preemptálási idő szemléltetése.

A preemptálási idő nagyban hasonlít a taszkváltási időhöz, azonban a járulékos utasítások miatt általában hosszabb időt jelent[21].

## Megszakítás-késleltetési idő

A megszakítás-késleltetési idő egy esemény beérkezése és a megszakítás-kezelő rutin első utasítása között eltelt átlagos időtartam[21].



**3.5. ábra.** A megszakítás-késleltetési idő szemléltetése.

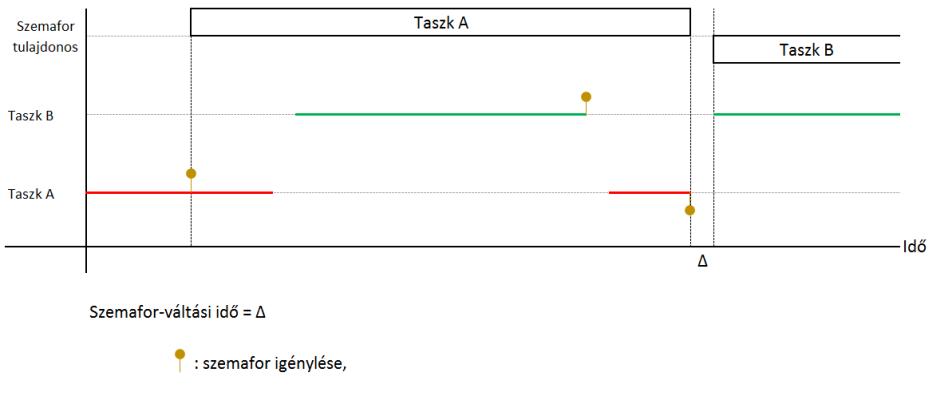
## Szemafor-váltási idő

Az 1989-es cikk szerint szemafor-váltási idő az az átlagos időtartam, ami egy szemafor elengedése és egy, a szemantika szerinti taszk elindulása között eltelt időtartam[21].

Ezt a meghatározást 1990-ben annyival módosították, hogy a szemafor-váltási idő egy már birtokolt szemafor kérése és a kérés teljesítése között eltelt időtartam, a birtokló taszk futási idejétől eltekintve (3.7. ábra)[22].

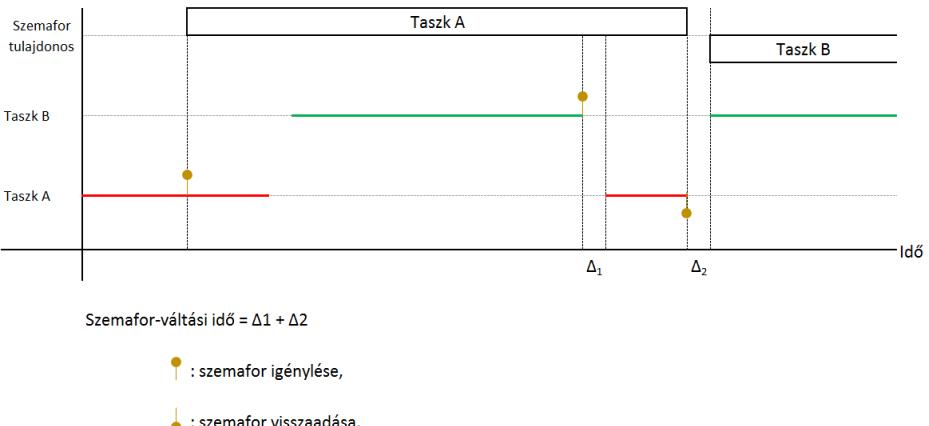
*kB/sec*-ben értelmezett adatot eredményez. Ha az 1 kB átvitelhez szükséges időt mérjük, akkor viszont

Szemafor-váltási idő 1.



**3.6. ábra.** A szemafor-váltási idő szemléltetése (1989-es meghatározás alapján).

Szemafor-váltási idő 2.



**3.7. ábra.** A szemafor-váltási idő szemléltetése (1990-es meghatározás alapján).

A mérés célja az overhead meghatározása, mikor egy szemafor kölcsönös kizárást (mutex) valósít meg[22].

### Deadlock-feloldási idő

A deadlock-feloldási idő az az átlagos időtartam, ami egy olyan erőforrás eléréséhez szükséges, amit egy alacsonyabb prioritású taszk már birtokol (3.8. ábra)[21].

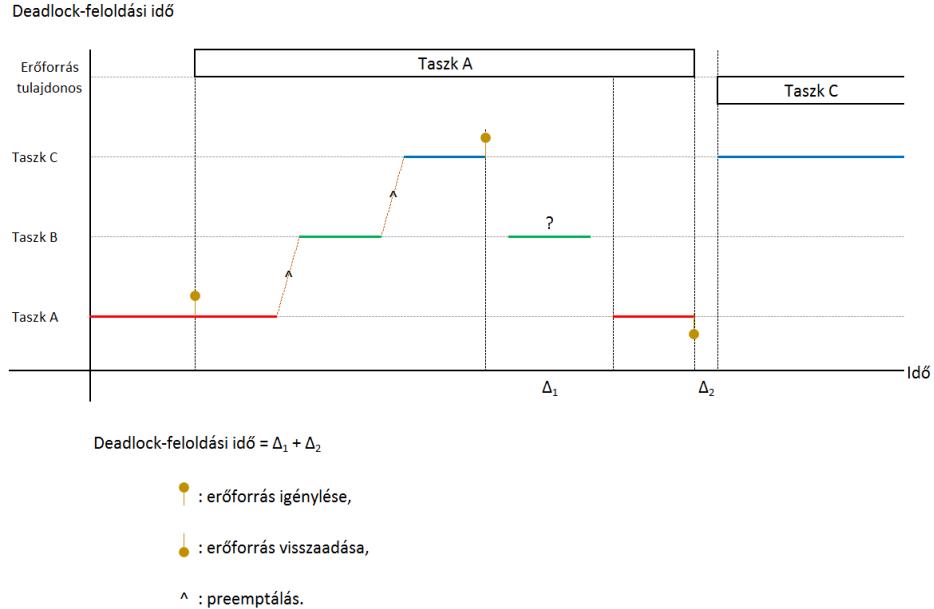
Vagyis a deadlock-feloldási idő a birtoklási probléma feloldásához szükséges összesített idő egy alacsony és egy magas prioritású taszk között.

### Datagram-átviteli idő

A datagram-átviteli idő a taszkok között elérhető adatsebesség az operációs rendszer objektumait kihasználva (vagyis nem megosztott memórián vagy pointeren keresztül). Az

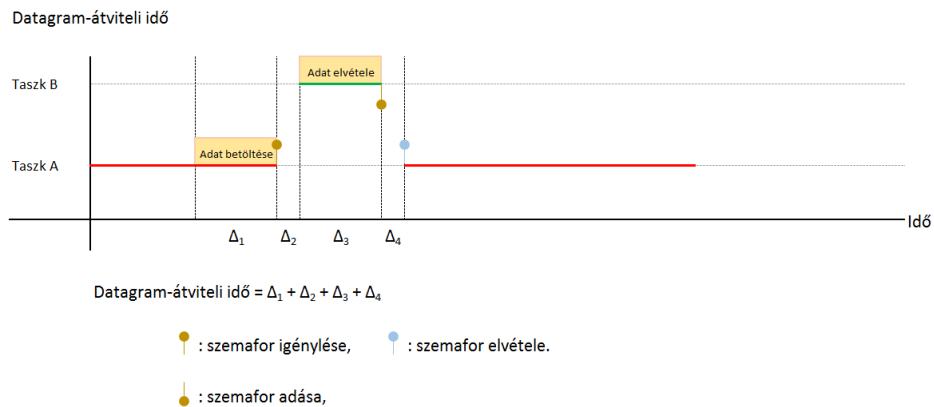
---

megoldódik ez a probléma.



**3.8. ábra.** A deadlock-feloldási idő szemléltetése.

adatküldő taszknak kapnia kell értesítést az adat átvételeiről (3.9. ábra)[21].

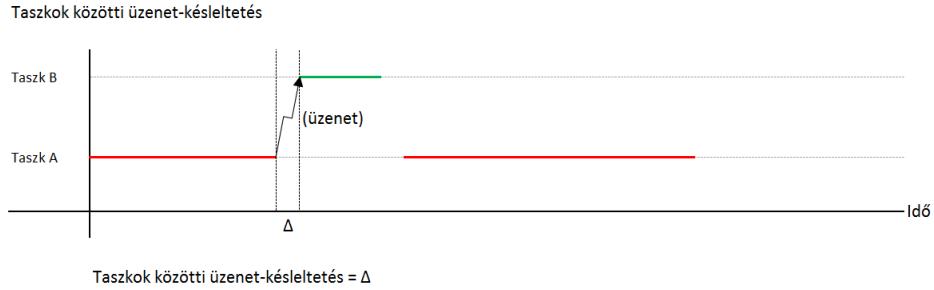


**3.9. ábra.** A datagram-átviteli idő szemléltetése.

Az egy évvel később megjelent cikkben ezt a kategóriát is módosították kis mértékben. Egyrészt a megnevezést taszk közötti üzenet-késleltetésre változtatták, másrészt nem a maximális adatsebesség meghatározása a mérés célja, hanem az adattovábbítást végző objektum kezelésének és az operációs rendszer járulékos műveleteinek hatékonyságának megmérése (3.10. ábra)[22].

### Rhealstone jellemzők összegzése

Az elvégzett mérések várhatóan mikroszekundum és milliszekundum nagyságrendű értékeket adnak vissza[21]. minden értéket másodpercre váltva, majd a reciprokát véve az értékek összegezhetők egy reprezentatív értékké. Az átváltásnak köszönhetően a nagyobb érték jobb teljesítményt jelent, ami a teljesítménymutatók világában elvárt.



**3.10. ábra.** A taszk közötti üzenet-késleltetési idő szemléltetése.

### Objektív Rheatstone érték

Objektív értékelés esetén minden jellemző azonos súllyal szerepel a számolás során ((3.1) képlet).

$$r_1 + r_2 + r_3 + r_4 + r_5 + r_6 = \text{objektív Rheatstone/sec}, \quad (3.1)$$

ahol

$r_1$  a taszkváltási időből származó érték,

$r_2$  a preemptálási időből származó érték,

és így tovább.

### Súlyozott Rheatstone érték

Az esetek döntő többségében a vizsgált feladatok nem azonos gyakorisággal szerepelnek egy alkalmazásban, sőt, előfordulhat, hogy valamely funkciót nem használ az alkalmazás. Ekkor informatívabb eredményt kapunk, ha az egyes jellemzőkre kapott számértékeket különböző súllyal vesszük bele a végeredmény meghatározásába ((3.2) képlet)[21].

$$n_1 \cdot r_1 + n_2 \cdot r_2 + n_3 \cdot r_3 + n_4 \cdot r_4 + n_5 \cdot r_5 + n_6 \cdot r_6 = \text{alkalmazás specifikus Rheatstone/sec}, \quad (3.2)$$

ahol

$n_1$  a taszkváltási időhöz tartozó súlytényező,

$r_1$  a taszkváltási időből származó érték,

$n_2$  a preemptálási időhöz tartozó súlytényező,

$r_2$  a preemptálási időből származó érték,

és így tovább.

Az súlytényezők értéke nulla is lehet.

### 3.1.5. Legrosszabb válaszidő

2001-ben a Nemzetközi Automatizálási Társaság (International Society of Automation – ISA) egy jelentésben fejtette ki azt az álláspontját, miszerint a késleltetés nem jellemzi a valósidejű rendszert, mert lehet, hogy a legtöbb esetben az előírt időn belül válaszol, de ritkán előfordulhatnak késleltetések vagy kihagyott események, amiket a mérés során nem lehet detektálni[23].

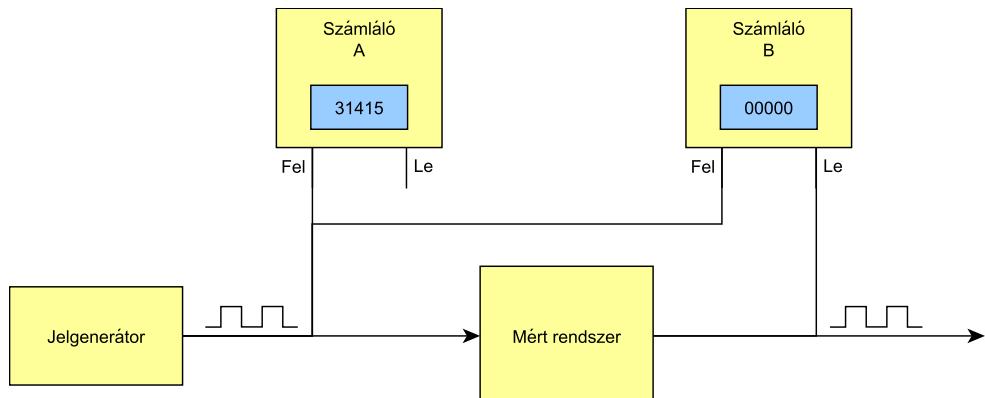
A Társaság egy olyan mérési összeállítást javasol a fejlesztőknek, ami egyszerűsége ellenére képes meghatározni a rendszer legrosszabb válaszidejét (3.11. ábra).

A méréshez szükséges eszközök:

- Mérő rendszer (legalább egy be- és kimenettel),
- Jelgenerátor,
- Két darab digitális számláló.

#### Mérési elrendezés

A jelgenerátor kimenetét a mérő rendszer bemenetére, illetve minden számláló *count up* bemenetére kötjük, míg a mérő rendszer kimenetét a kimeneti számláló *count down* bemenetére kötjük[23].



3.11. ábra. A legrosszabb válaszidő mérési összeállítása.

#### Mérési elv

A mérés során azt a legkisebb frekvenciát keressük, amit a rendszer már nem tud követni, vagyis impulzusokat veszít. Ezáltal a kimenetén megjelenő impulzusok száma különbözní fog a bemenetére adott impulzusok számától, amit a számlálók segítségével detektálunk. A kapott frekvencia a legrosszabb válaszidő reciproka[23].

#### Mérés menete[23]

1. A rendszeren futó program a bemenetére érkező jelet a kimenetére másolja. A mérés során digitális és analóg I/O láb is használható.

2. Mérési eszközök csatlakoztatása.
3. Alacsony frekvenciáról indulva növeljük a bemeneti jel frekvenciáját. Az  $A$  számláló folyamatosan számol felfele. Amíg a rendszer képes követni a bemenetet, addig a  $B$  számláló 1 és 0 között váltakozik. Amikor a rendszer már nem képes követni a bemenetet, akkor a  $B$  számláló elkezdt felfele számolni.
4. Csökkentjük a bemeneti frekvencia értékét egészen addig, amíg a rendszer újból képessé nem válik a bemenet követésére. A kapott frekvencia a legrosszabb válaszidő reciproka.

A mérést célszerű elvégezni különböző terhelés mellett. Ha valamelyik funkció használata közben (adattároló vezérlése, hálózati kommunikáció, stb.) a  $B$  számláló elindul, akkor az adott frekvencián a rendszer nem determinisztikus.

### **3.2. Vizsgált operációs rendszer jellemzők**

A feladat megoldása során elsődlegesen az operációs rendszerek jellemzőit vizsgálom, ezért nem kerülnek külön tesztelésre az egyes hardverek előnyei. Az egyes rendszer-jellemzőket terheletlenül és terhelés alatt is megmériem.

Egy ipari alkalmazás szimulációját is megvalósítom, mely egy másik összehasonlítási alapot nyújt a dolgozathoz. Az alkalmazást felhasználom a terhelés alatti mérés megvalósításához is.

A dolgozat további részeiben a 3.1. fejezetben felsorolt összes jellemző meghatározására képes rendszert állítok össze, mellyel végrehajtom a méréseket. A meghatározandó jellemzők:

- Memóriaigény,
- Késleltetés,
- Jitter,
- Rheatstone értékek (első publikáció szerinti értékek),
- Legrosszabb válaszidő.

## 4. fejezet

# Rendszerterv

### 4.1. Megvalósítandó feladat

A mérések elvégzéséhez tervezni kellett egy olyan eszközt, amely könnyen kezelhető interfészet biztosít a mérések elvégzéséhez és az ipari alkalmazáshoz szükséges alkatrészeket is tartalmazza.

Az ipari alkalmazás tervezésekor törekedtem arra, hogy használjak olyan perifériákat, melyek egy termék működése során előfordulhatnak. Ez egy egyszerű GPIO-tól kezdve a vezeték nélküli kommunikációt is magában foglalja.

A szimulált alkalmazás mögötti gondolat egy helyi és egy távoli állomás monitorozása. A helyi állomáson egy hőmérő, egy potméter, két LED, illetve két kapcsoló kapott helyet. A távoli állomást egy *TI Sensortag* valósítja meg, mellyel a kapcsolatot Bluetooth low energy használatával oldottam meg. A mért adatokat grafikusan megjelenítem és ezzel egyidőben SD kártyára is mentésre kerülnek az értékek.

A TI Sensortag egy megvásárolható, több szenzort tartalmazó kis méretű modul, melyet a *Texas Instruments* gyárt kifejezetten IoT alkalmazások prototípusának tervezéséhez. Különböző vezetéknélküli technológiát használó verziója elérhető. Az általam választott Sensortag *CC2650* mikrokontrollert használ központi egységgé, melyet kifejezetten 2,4 GHz-en kommunikáló alkalmazásokhoz fejlesztettek. A mikrokontroller alacsony fogyasztása lehetővé teszi az elemről történő táplálást, és a beépített 32-bites ARM Cortex-M3 processzornak és a tartalmazott széleskörű periféria-készletnek köszönhetően különálló alkalmazások fejlesztésére is alkalmas.

A merőkártyán a Bluetooth kapcsolat létrehozását a Bluegiga *BLE112* moduljával valósítottam meg. A modul *CC2540* mikrokontroller köré épül, mely 8051 architektúrájú magot tartalmaz. Az eszköz vezérelhető UART-on vagy USB-n keresztül is, de akár különálló alkalmazás is megvalósítható a használatával. A gyártó ún. *BGScript* nyelvet kínál az eszközhöz, mellyel a modul összetettebb feladatokra is felprogramozható.

Érdemes pár mondatban áttekinteni a Bluetooth low energy kommunikáció történetét és működését.

A Bluetooth low energy (gyakran BLE-ként vagy Bluetooth Smart-ként találkozhatunk vele) a Bluetooth 4.0-ás verzióval került bevezetésre. Célja, hogy olyan vezeték nélküli

kommunikációt valósíthassunk meg, ami alacsony fogyasztású, gyors csatlakozási lehetősséggel bír és biztonságos. Kétféle Bluetooth 4.0 eszköz különböztethető meg a támogatott protokollok alapján:

**Single-mode eszköz:** kizárálag a Bluetooth low energy technológiát támogatja,

**Dual-mode eszköz:** a Bluetooth low energy technológia mellett a klasszikus Bluetooth technológiákat is támogatja, beleértve a Bluetooth korábbi verzióit is.

BLE alkalmazásokban általában egy *kliens* és legalább egy *szerver* eszköz vesz részt a kommunikációban. A szerverek tipikusan szenzorral rendelkező eszközök, mint hőmérő vagy szívritmus-figyelő eszközök, így ők szolgáltatják az információt. A kliens gyűjti be az adatokat a szerverektől, és jeleníti meg a felhasználó számára. Kliensként működhet akár egy telefon vagy számítógép is.

Minden szerver rendelkezik profillal (Profile), melyben az eszkösről, illetve a lekérhető adatokról található információ. A profil több szolgáltatást (Service) is tartalmazhat, és bizonyos alkalmazások esetén lehetnek kötelezően megvalósítandó szolgáltatások (például szívritmus-figyelő esetén). A szolgáltatások határozzák meg, hogy milyen adatok olvashatók ki az eszközből, ezen adatokat miként lehet elérni és milyen biztonsági követelmények vannak az adatokra nézve. A szolgáltatásokon belül találhatóak a karakterisztikák (Characteristic), amik a tényleges adatot teszik elérhetővé. A karakterisztika ismert típussal rendelkező érték. Mind a profil, a szolgáltatások és a karakterisztikák rendelkeznek UUID-val, ami egyértelműen azonosítja őket. A karakterisztikák elérhetőek ún. handle-n keresztül is, mely egy kezelhetőbb módot kínál az érték olvasására és írására.

## 4.2. Részletes terv

Első körben az elérhető perifériákat és a felhasználható eszközöket sorakoztattam fel.

A 4.1. fejezetben leírtak szerint a naplázás SD kártyára történik. Ezen kívül hőmérőklet mérését is meg kell valósítanom, amihez analóg kimenetű hőmérőt választottam (*MCP9700A*). A helyi állomáson található két szabadon állítható LED, illetve két kapcsoló is, melyek GPIO lábakat foglalnak el. További GPIO lábakat igényel az *IN*, a négy darab *ID*<sup>1</sup> és az *OUT* mérési pontok. A kapcsolók és az *IN* lábak esetén megszakítást is támogatnia kell a lábaknak. A potméter kimeneti lába szintén analóg jelet közvetít a fejlesztőkártyák felé, így a szükséges analóg lábak száma összesen kettő. Mivel a Raspberry Pi nem rendelkezik analog-digital konverziót támogató lábakkal, ezért választanom kellett egy olyan digitális interfésszel rendelkező ADC IC-t, mely a mért adatokat a Raspberry Pi számára elérhetővé teszi. Erre a célra az *ADS7924* négy csatornás IC-t választottam, mely I<sup>2</sup>C-n keresztül vezérelhető, és ugyanezen interfészen keresztül olvasható ki belőle a mért adat.

A vezeték nélküli kommunikációhoz használt BLE112 modul vezérelhető soros interfészen (*flow-control* használatával és nélküle egyaránt) vagy USB-n keresztül. A fejlesztés során

<sup>1</sup>Az ID lábakok az éppen futó taszk 4 bites azonosítója mérhető.

felmerülő hibák megtalálása szempontjából a soros kommunikáció kedvezőbb, ezért azt választottam.

Az STM32F4 Discovery kártya nem rendelkezik kijelzővel, és a kiegészítő kártyához kapható TFT kijelző sem állt rendelkezésemre, ezért az adatok megjelenítését és a LED-ek vezérlését asztali alkalmazással oldottam meg, amely soros kommunikációt keresztül küldi és fogadja az adatokat.

Külső megtáplálásra is lehetőséget akartam teremteni, ezért *DC Jack* csatlakozó is került az áramköri lapra. Mivel az STM32F4 Discovery rendelkezik 5 V-os kivezetésekkel, ezért a külső feszültségforrás feszültségét is 5 V-ra választottam, és az áramköri elemeknek szükséges 3 V-ot *LM317* feszültség-stabilizátor IC-vel állítottam elő. A bemeneti feszültség forrásának (külső feszültségforrás vagy a fejlesztőkártya megfelelő lába) kiválasztása jumper megfelelő helyzetbe állításával történik.

Ezen kívül még SD kártya foglalat is helyet kapott a lemezen. Bár a STM32F4 Base Board és a Raspberry Pi is rendelkezik foglalattal, ezzel a kiegészítéssel a Base Board használata opcionálissá válik.

A BLE112 modul programozható, és erre a fejlesztés közben is sor került, ezért a programozást lehetővé tevő csatlakozó is rákerült a mérőkártyára.

A fejlesztőkártyák felé a kapcsolatot tüskesorok alkalmazásával oldottam meg. Ez a megoldás egyszerűsíti a tervezést, mert nem kellett minden kártyához illeszkedő hüvelyisorokat elhelyeznem, figyelve a helyes lábkiosztásra, miközben a lábak nagy részét nem használja az alkalmazás. A főbb kivezetéseket  $2 \times 10$ -es csatlakozóval terveztem, ezáltal húsz polusú szalagkábelrel és a hozzá tartozó csatlakozóval a kártya oldali csatlakoztatás egyszerűsödik. A csatlakozóra az alábbi eszközök kerültek kivezetésre:

- Tápfeszültség (5 V),
- Kapcsolók (2 db),
- LED-ek (2 db),
- UART (BLE112 vezérléséhez),
- Mérés bemenete,
- Mérés során használt ID lábak (4 db),
- Mérés kimenete,
- Hőmérő analóg kivezetése,
- Potméter kivezetése,
- I<sup>2</sup>C.

Emellett tüskesorokat alkalmaztam a mérési pontok kivezetésére (IN, OUT és ID lábak), illetve az SD kártya foglalat csatlakoztatása is tüskesoron keresztül történhet. A BLE112 modul lábainak többsége szintén kivezetésre került, ezáltal a modul képességeinek tesztelésekhez is használható a kártya.

### 4.3. Hardverterv

A tápfeszültséget szolgáltathatja az STM32F4 Discovery fejlesztőkártya, vagy külső feszülféreg forrás is. A külső forrásból való táplálás esetén a túláram-védelmet *multifuse* alkalmazásával oldottam meg. A multifuse védett oldala és az STM kártyáról érkező 5 V-os feszültség  $1 \times 3$ -as tüskesor két szélső lábára csatlakozik, ahol jumper kapcsolja a kiválasztott forrást a középső lábra. A középső laban látja el a mérőkártya többi részét 5 V-os feszültséggel, és ebből állítja elő az LM317 a 3 V-os feszültséget. A bemenet védett a fordított polaritással rákapcsolt feszültséggel szemben. A tápfeszültség előállításáért felelős áramkör kapcsolási rajza a B. Függelék B.2. ábráján látható.

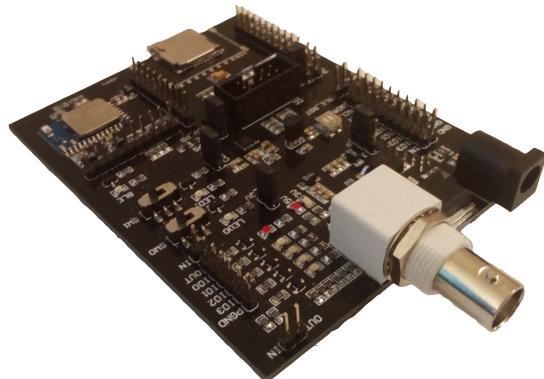
A szenzorok (hőmérő és potméter) analóg kimenete szűrés után a kimeneti tüskesorra és az ADC IC bemeneteire csatlakoznak. A használt ADS7924 IC kihasználatlan bemeneteit földre kötöttem. Az IC a bemeneteit multiplexálva az egyik kimeneti lábára vezeti ki, ahova tetszés szerint jelkondicionáló áramkört helyezhetünk, így nem kell minden bemenethez külön áramkört használni. A tervezés során nem használtam jelkondicionáló áramkört, viszont későbbi alkalmazásokhoz ezt lehetővé tettem  $1 \times 2$ -es tüskesor elhelyezésével. A mérések során jumper zárja rövidre a tüskesor két lábat. Az ADC IC tápfeszültsége lekapcsolható. A szenzorok bekötése a B. Függelék B.3. ábráján látható.

A Bluetooth modul vezérléséhez szükséges lábak a fejlesztőkártyákhoz csatlakozó tüskesorra vannak kivezetve. Ezen kívül a programozáshoz szükséges lábak a programozó csatlakozóra lettek kötve. Emellett a modul tápfeszültségét szolgáltathatja a programozó vagy a mérőkártya, amit szintén jumper határoz meg. A BLE112 modul bekötése látható a B. Függelék B.5. ábráján.

Az SD kártya foglalatának lábai (védőellenállások használatával) tüskesorra vannak kivezetve. Az SD kártya tápellátása független a mérőkártya többi részétől, abból a célból, hogy később más tesztalkalmazásokhoz is felhasználható legyen. Két jumper segítségével közösíthető a tápfeszültség és föld is.

A méréshez használt kivezetések visszajelző LED-eket is kaptak, egyrészt a fejlesztés során felmerülő problémák felderítésére, másrészt egyéb alkalmazásoknál legyen lehetőség a használatukra. A LED-ek (és a hozzájuk tartozó kiegészítő áramkör) beültetése opcionális. A méréshez a bemeneti lábhoz BNC csatlakozó is elhelyezésre került, ezzel megkönnyítve a jelgenerátor csatlakoztatását.

Az elkészült nyomtatott áramkori terv a C. Függelék C.1. és C.2. ábráján látható.



**4.1. ábra.** Az elkészült mérőkártya.

#### 4.4. Szoftverterv

A méréshez használt szoftver tervezésekor figyelnem kellett arra, hogy a mérések könnyen elvégezhetőek legyenek, miközben a mért értékeket ne befolyásoljam. Emellett szempont volt a mérési adatok egyszerű feldolgozhatósága is.

Minden olyan mérésnél, amelynél a bemenetet használni kellett, ott megszakítás következik be a bemenetre érkező szintváltásoknál. A megszakítás-kezelő rutinban az adott mérésnek megfelelően végeztem el a további műveleteket.

Azon méréseknél, amelyeknél nem a bemenetre érkező jelre indul a mérés, azoknál létrehoztam egy mérést indító taszkot, ami másodpecenként tízszer indítja el az adott mérési folyamatot.

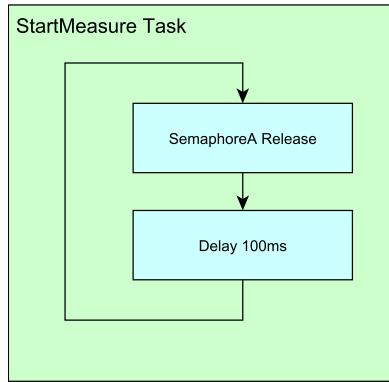
##### 4.4.1. Késleltetés és Jitter

A késleltetés mérésénél a megszakítás hatására a bemeneten levő jelet másoltam a kimenetre a megszakítási rutinban. Mivel minden operációs rendszer kezeli a megszakításokat, így ezzel a módszerrel információt kapunk a rendszer sebességéről.

##### 4.4.2. Rheatstone értékek

###### Mérések indítása

A Rheatstone értékek mérésénél (a megszakítás-késleltetési idő kivételével) a mérési folyamat független a bemenetre érkező jeltől. Hogy az egyes mérések ne terheljék folyamatosan a processzort, illetve hogy a mérés során az adatok könnyen feldolgozhatóak legyenek, létrehoztam egy taszkot, mely periodikusan elindítja a mérést, majd várakozik a következő indítási eseményig. A mérés kezdetének jelzését szemaforral oldottam meg. A taszk működése látható a 4.2. ábrán.



**4.2. ábra.** *StartMeasure taszk folyamatábrája.*

### Taszkváltási idő

A taszkváltási idő méréséhez három, azonos prioritású taszkot hoztam létre. Az első –  $\mathcal{A}$  jelű – taszk szemaforon keresztül várakozik az indító jelzésre. Ennek beérkezése után szemafor segítségével elindítja a  $\mathcal{B}$  jelű taszkot, amely ugyanígy indítja el a  $\mathcal{C}$  jelű taszkot. Ezután egy hosszabb lefutású ciklusba lépnek, így folyamatosan processzoridőt kérnek az operációs rendszertől. A ciklus lejártával várakoznak a következő indításra. A folyamat a 4.3. ábrán látható.

### Preemptálási idő

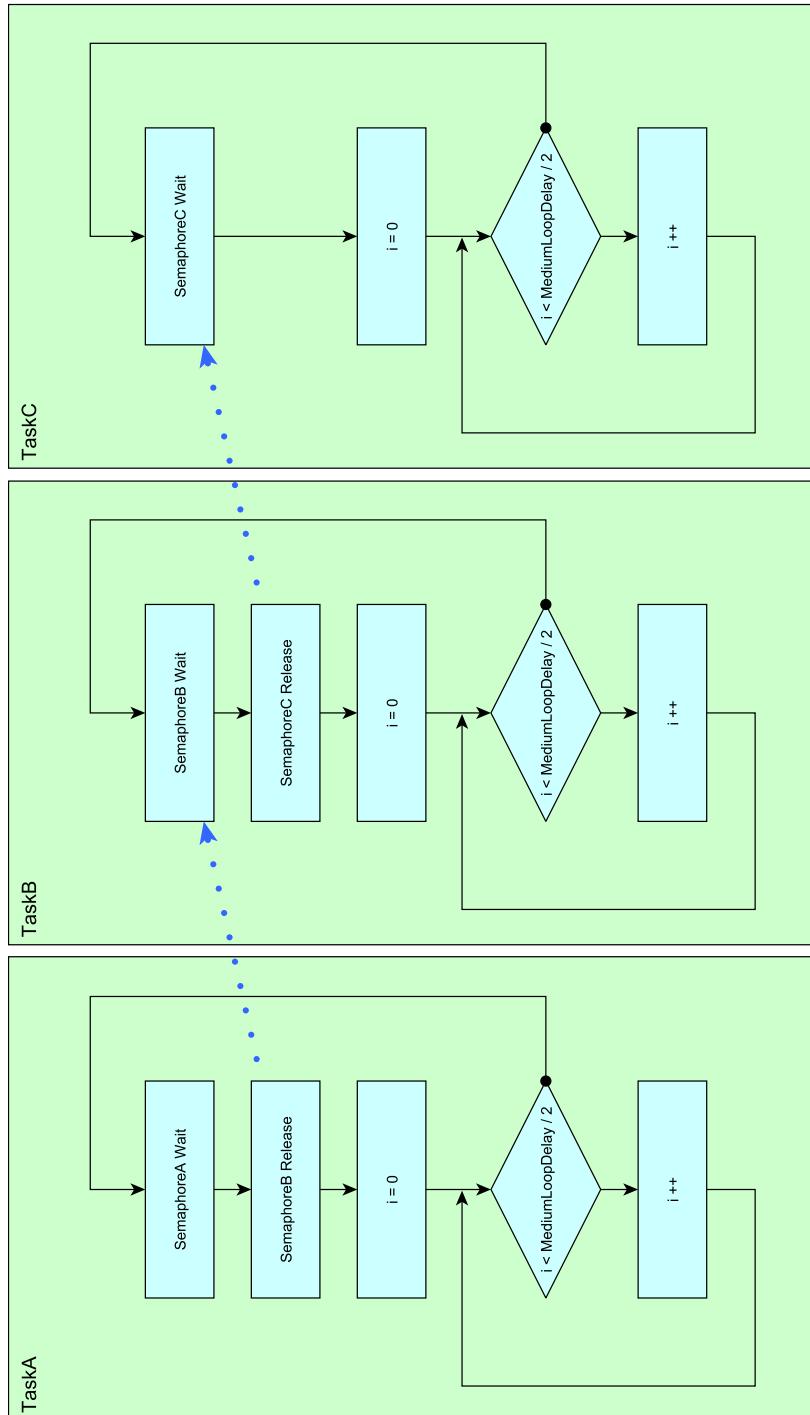
A preemptálási idő méréséhez három, különböző prioritású taszkot használtam. A taszkok indítása a már megismert módon történik, viszont a közepes és magas prioritású taszkok ( $\mathcal{B}$  és  $\mathcal{C}$  jelű) kis ideig várakoznak, így adva az  $\mathcal{A}$  jelű taszknak futási jogot. Különbség még a *taszkváltási idő* méréséhez képest, hogy a három taszk ciklusa különböző hosszú. A várakozási időt és a ciklusok hosszát úgy választottam meg, hogy a közepes prioritású  $\mathcal{B}$  taszk preemptálja  $\mathcal{A}$ -t, és a magas prioritású  $\mathcal{C}$  taszk preemptálja  $\mathcal{B}$ -t. A folyamat a 4.4. ábrán látható.

### Megszakítás-késleletetési idő

A megszakítás-késleltetési idő mérése során a bemenetre érkező jel fel- és lefutó éle által generált megszakítás szemaforon keresztül kerül jelzésre a kezelő taszk számára. Mivel a meghatározás szerint a megszakítás bekövetkezésétől az első hasznos utasításig eltelt időt kell meghatározni, ezért a gépi kód generálását nem bízom a fordítóra, hanem *assembly* nyelven végzem el a bemeneti láb beolvasását és a kimeneti láb beállítását. Így az egyes utasítások végrehajtásának ideje egyértelműen meghatározható. A mérési folyamat a 4.5. ábrán látható.

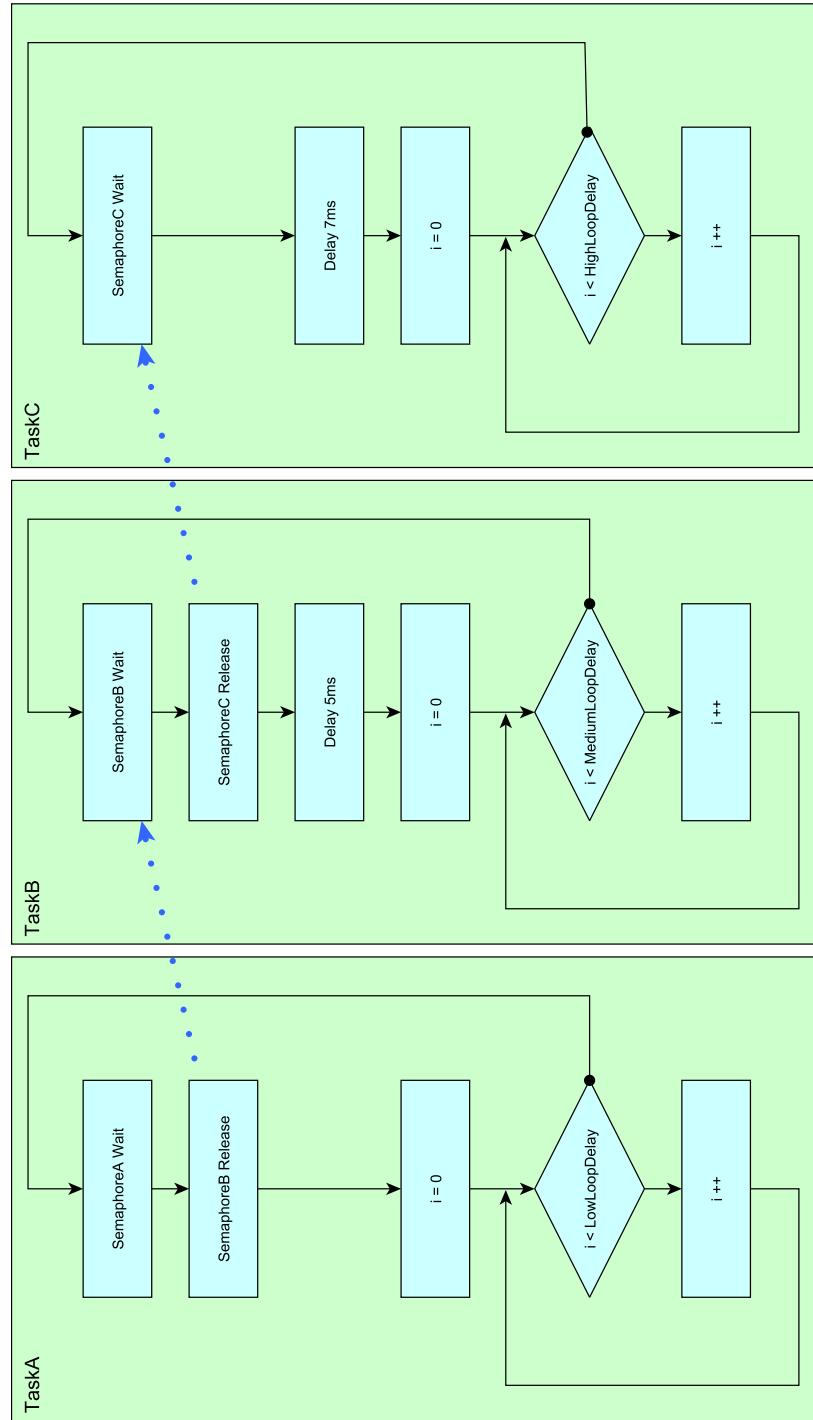
### Szemafor-váltási idő

A mérést két taszk valósítja meg, amik különböző prioritással rendelkeznek. Miután minden két taszk elindult, a magasabb prioritású  $\mathcal{B}$  taszk várakozik. Ezen idő alatt az alacsony



4.3. ábra. Taszkváltási idő méréséhez használt taszkok folyamatábrája.

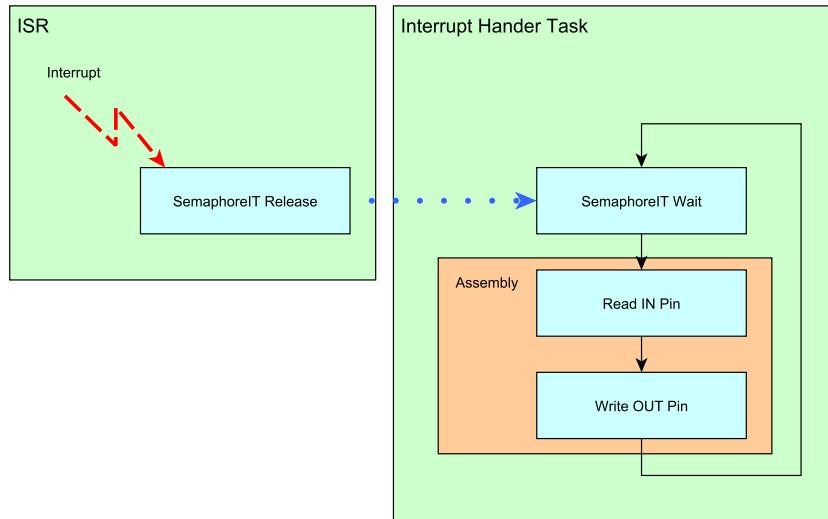
prioritású  $\mathcal{A}$  taszk lefoglalja a szemafort, majd hosszabb lefutási ciklusba lép. Ekkor újból az  $\mathcal{B}$  taszk kapja meg a futás jogát, és a ciklus lejártával elengedi a szemafort. Ekkor a  $\mathcal{B}$  taszk lefoglalja, majd egyből el is engedi a szemafort, és a két taszk várakozik a következő indításra. A mérést megvalósító taszkok a 4.6. ábrán láthatóak.



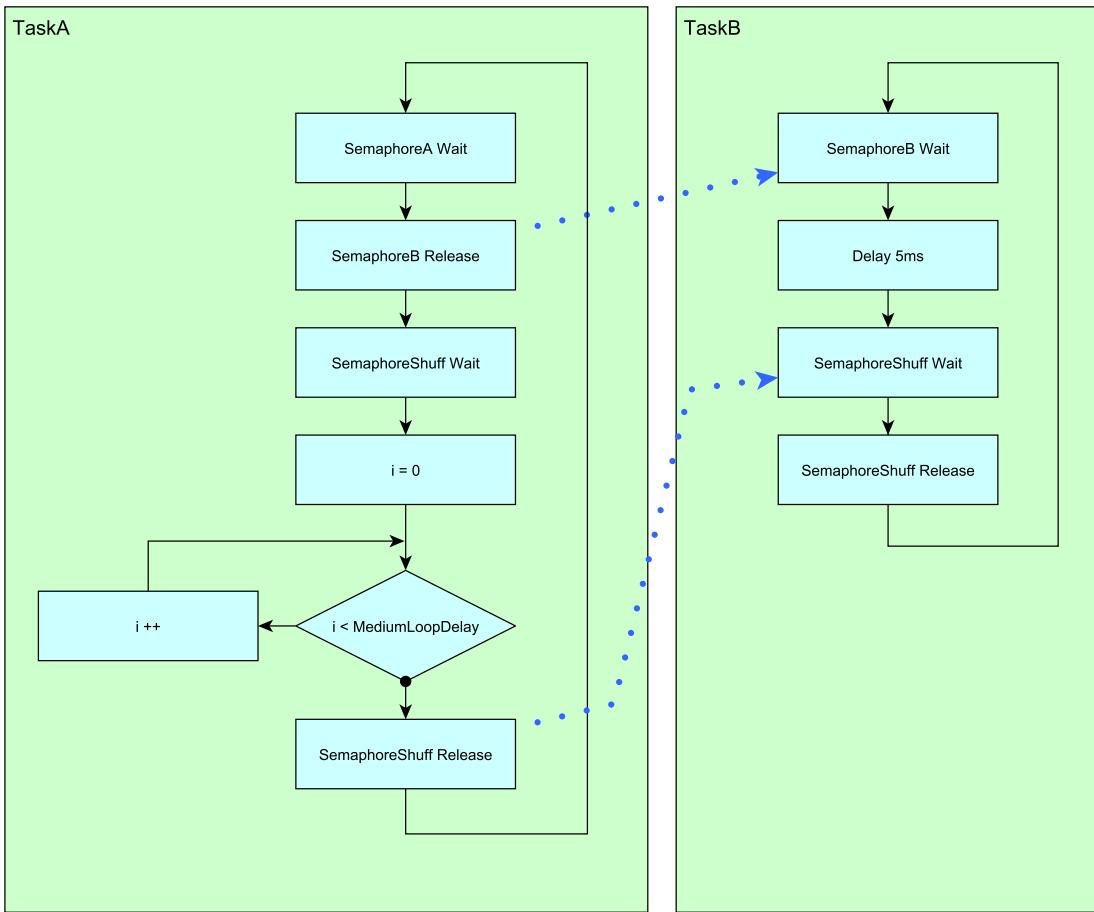
4.4. ábra. Preemptálási idő méréséhez használt taszkok folyamatábrája.

### Deadlock-feloldási idő

Három különböző prioritású taszk vesz részt a mérésben. Miután minden taszk elindult – és a magasabb prioritásúak várakoznak, hogy az alacsony prioritású taszk futási jogot kapjon – az alacsony prioritású  $\mathcal{A}$  taszk lefoglalja a mutex-et, majd hosszú lefutású ciklusba lép. Közben a közepes prioritású  $\mathcal{B}$  taszk késleltetése letelik, és preemptálja  $\mathcal{A}$ -t. A  $\mathcal{B}$  taszk is üres ciklus futtatásába kezd. Mikor a  $\mathcal{C}$  taszk várakozási ideje is lejár, akkor megpróbálja lefoglalni a mutex-et, amit az alacsony prioritású taszk birtokol. Az ezt követő viselkedés

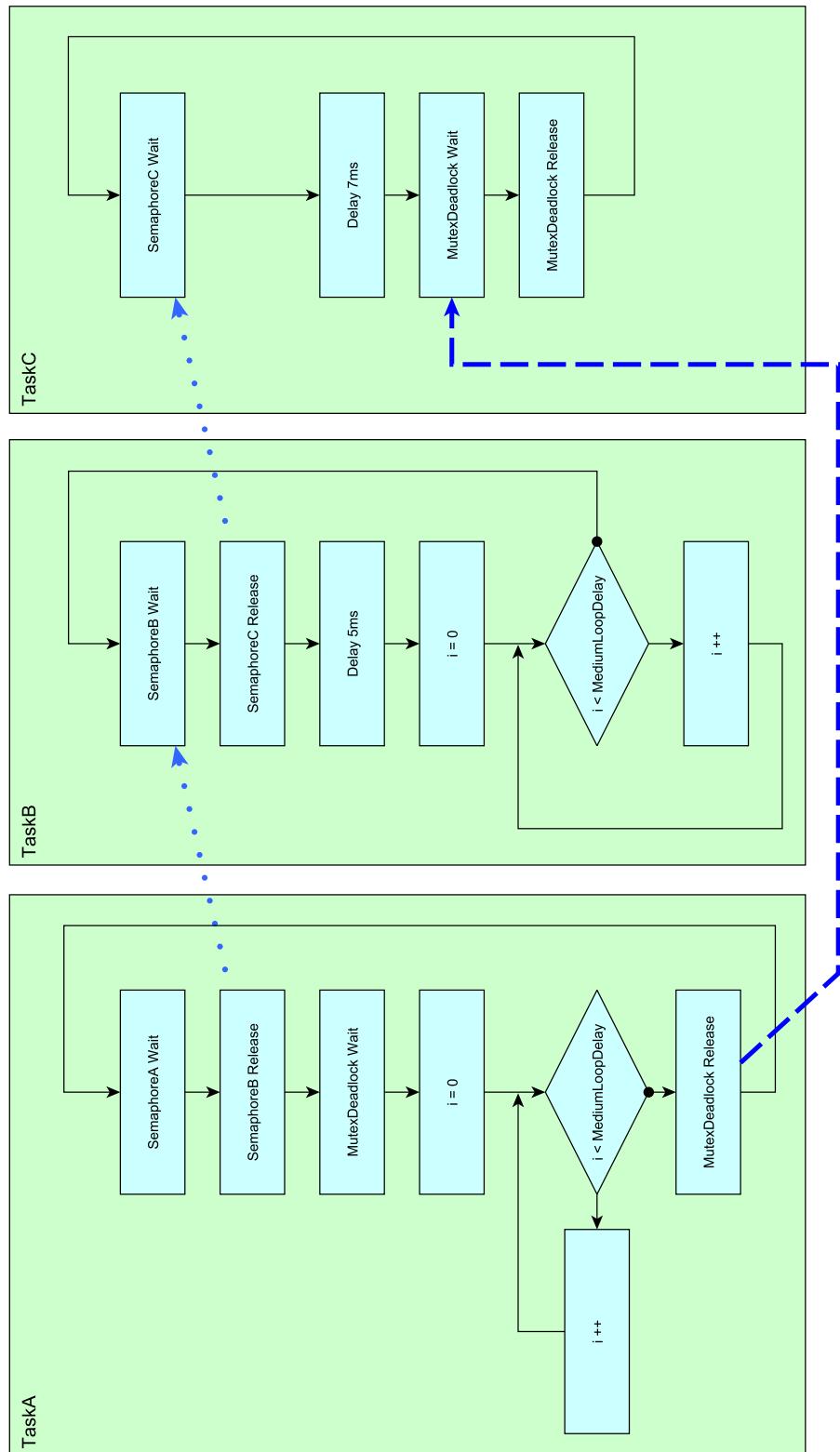


**4.5. ábra.** Megszakítás-késleltetési idő méréséhez használt taszk folyamatábrája.



**4.6. ábra.** Szemafor-váltási idő méréséhez használt taszkok folyamatábrája.

függ attól, hogy az  $\mathcal{A}$  taszk örökli-e a  $\mathcal{C}$  taszk prioritását vagy sem, de valamikor az alacsony prioritású taszk befejezi a ciklus futtatását, majd felszabadítja a mutex-ot, amit a magas prioritású  $\mathcal{C}$  taszk lefoglal, majd azonnal el is engedi. A folyamat a 4.7. ábrán látható.

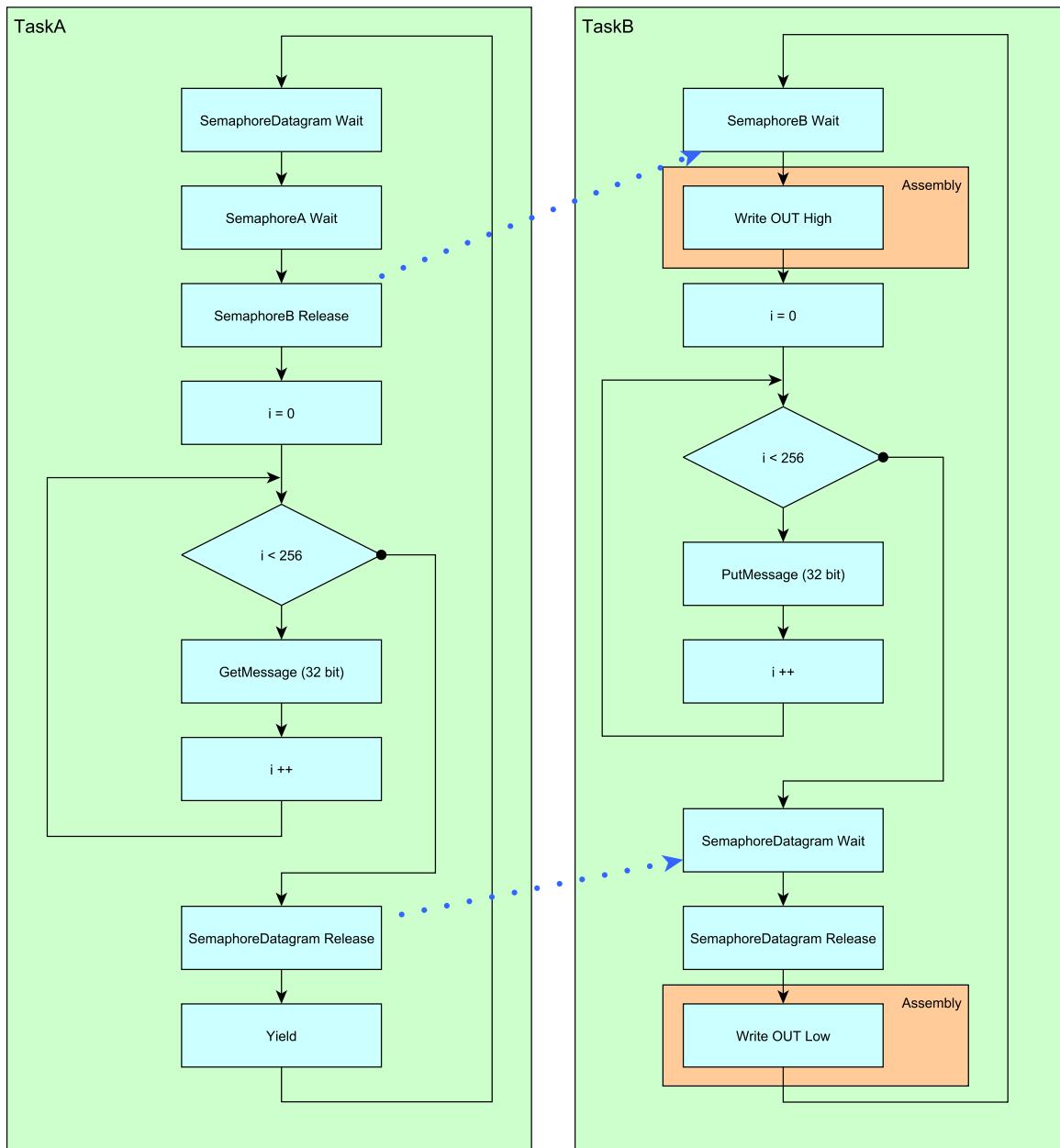


4.7. ábra. Deadlock-feloldási idő méréséhez használt taszkok folyamatábrája.

### Datagram-átviteli idő

A datagram-átviteli idő mérését kettő, különböző prioritású taszk valósítja meg. Az adat vételét szemaforral jelzi a vételi oldalt megvalósító taszk, amit már a mérés kezdete előtt

birtokol. A taszkok elindulása után a magas prioritású  $\mathcal{B}$  jelű taszk a kimeneti lábat magas értékbe állítja, majd a 32-bites  $i$  változó értékét 256 alkalommal betölti a sorba ( $1 kB$  adat). Ezután megpróbálja elvenni a méréshez használt szemafort, aminek következtében az alacsony prioritású  $\mathcal{A}$  jelű taszk folytatja futását. A sorból elveszi a betöltött adatokat, majd a szemafor elengedésével jelez a  $\mathcal{B}$  jelű taszknak, amely egyből felszabadítja a szemafort, majd a kimeneti láb alacsony szintre állításával jelzi az adatátvitel befejeződését. A kimeneti láb írása ebben az esetben is assembly nyelven történik. A mérés folyamata a 4.8. ábrán látható.



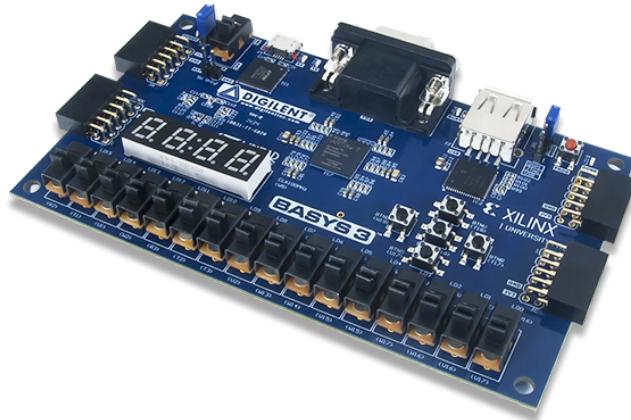
**4.8. ábra.** Datagram-átviteli idő méréséhez használt taszkok folyamatábrája.

#### 4.4.3. Legrosszabb válaszidő

A legrosszabb válaszidő mérését a Rhealstone értékek meghatározásánál használt *megszakítás-késleltetési idő* szoftverével hajtottam végre. Mivel a mérés során szükség van olyan számlálóra, mely rendelkezik felfele és lefele számláló bemenettel, így megoldást kellett találnom a mérés kivitelezésére.

Új hardver tervezése időigényes folyamat lett volna, és a megfelelő alkatrészek megtalálása esetén is meg kellett volna oldali az aktuális érték kijelzését. Végül amellett döntöttem, hogy FPGA-t használok a feladat elvégzésére.

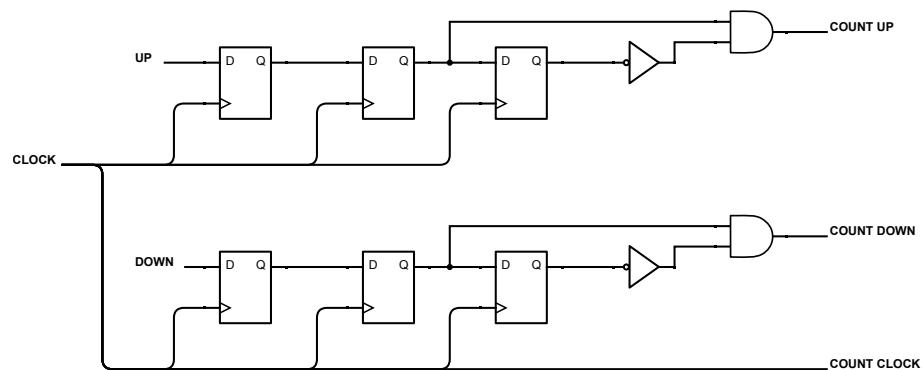
Rendelkezésemre állt egy *Basys 3* FPGA fejlesztőkártya, amin megfelelő kivezetések és hétszegmenses kijelző is található. A kártyához kapott példaprogramok között volt olyan, amiben megtalálható volt BCD-hétszegmens dekóder, így az én feladatom a BCD számláló megvalósítására és a bemeneti jelek helyes kezelésére terjedt ki. Az éldetektálást végző logika a 4.10. ábrán látható módon lett megvalósítva<sup>2</sup>. A *COUNT UP* és *COUNT DOWN* jelek felhasználásával növelem, illetve csökkentem a számláló értékét, figyelve arra, hogy az értéket ne módosítsam, ha a két jel egyszerre érkezik.



**4.9. ábra.** *Basys 3* fejlesztőkártya[24].

---

<sup>2</sup>A közvetlenül a bemenetre helyezett tárolók feleslegesnek tűnhetnek, viszont ha az időzítési követelmények nem teljesülnek, akkor az első flip-flop kimenetén metastabil állapot alakulhat ki. Ennek elkerülésére ökölszabályként alkalmazandó, hogy az első tároló kimenete mindenkorábban csak egy áramkörök elem bemenetére csatlakozik.



**4.10. ábra.** A bemeneti jelek éldetektálását végző áramkör.

## 5. fejezet

# Megvalósítás

### 5.1. STM32F4 Discovery

A STM32F4 Discovery kártyára való fejlesztés során az *Atollic TrueSTUDIO* fejlesztőkörnyezet ingyenes verzióját, illetve az *STM32CubeMX* konfiguráló alkalmazást használtam. Mindkét operációs rendszernél kikapcsoltam az optimalizációt.

#### 5.1.1. FreeRTOS

Az STM honlapjáról letölthető STM32CubeMX alkalmazás támogatja a FreeRTOS és FatFS<sup>1</sup> használatát, így a megfelelő perifériák beállítása és a szükséges modulok kiválasztása után működő példaprogram generálható.

A generált kód áttekintése közben észrevettem, hogy az operációs rendszer függvényeinek nevei nem egyeznek meg a 2.3. fejezetben megismert függvények neveivel. Ennek oka, hogy a CubeMX a FreeRTOS függvényeit a CMSIS-RTOS API-val elfedi, és egy egyszerűsített felületet biztosít a fejlesztő számára. Ez az egyszerűsített felület például a szemáforok használatánál figyelhető meg, ahol nincs külön metódus a megszakításból történő használatra, hanem az API vizsgálja meg, hogy megszakítás-kezelő rutinból történt-e a hívás, és ennek megfelelően végzi el a további műveleteket. Mivel az összehasonlítás során a fejlesztési folyamat nehézségei is szempontként szerepelnek, ezért éitem az API használatával.

Mielőtt a 4.4. fejezetben ismertetett taszkokat implementáltam volna, a rendszer megismerésével foglalkoztam. Valahogy el kellett érniem, hogy az éppen futó taszhöz azonosítót rendeljek, amit valahogy az ID lábakon jelezni is tudok. Ehhez kihasználtam azt, hogy minden taszk rendelkezik névvel, így megfontolt elnevezéssel minden taszhöz rendelhető 4 bites azonosító, amit a TCB-ből is egyszerűen kiolvashatok.

Az ASCII karakterek táblázatát megvizsgálva látható, hogy az „A” karaktertől kezdődően a betűk alsó bitjei használhatóak az ID lábak értékének meghatározására. Az azonosítók kiosztásánál úgy döntöttem, hogy a 0x0 érték tartozzon az ütemezőhöz, a 0x1 az Idle taszhöz, és a nagyobb értékek tartozzanak a mérésben részt vevő taszkokhoz. Így az egyes taszkok nevei „B”-től „O”-ig terjedhetnek.

A következő lépés volt a rendszer ütemezési mechanizmusának tanulmányozása. Meg

---

<sup>1</sup>Fat fájlrendszer használatához elérhető könyvtár, mely az SD kártya használatához szükséges.

kellett keresnem azokat a részeket, amelyek minden ütemezésnél lefutnak, és az ütemezés indulásakor a 0x0 azonosítót kellett kitennem az ID lábakra, míg az ütemezés végén a kiválasztott taszk azonosítóját. A FreeRTOS két esetben futtatja le az ütemezőt:

- SysTick esemény bekövetkezésekor,
- portYIELD() függvény hívásakor.

Ezért a *portYIELD()* függvény elején, illetve a *SysTick* megszakítás-kezelő rutin elején az 5.1. listában látható assembly utasításokkal helyeztem ki az ütemező azonosítóját. A *portYIELD()* függvény minden esetben kontextus-váltást kér a *PendSV* megszakításon keresztül, ellenben a *SysTick* kezelő csak szükséges esetben. Emiatt a *SysTick* rutint módosítanom kellett, és amennyiben nem szükséges kontextus-váltás, úgy az aktuális taszk azonosítóját helyezem az ID lábakra.

Kontextus-váltás esetén a *PendSV* megszakítás-kezelő végzi el az éppen futó taszk állapotának elmentését és a soron következő taszk legutolsó állapotának betöltését. A taszk-váltás elvégzése után, a megszakításból való visszatérés előtt a betöltött taszk azonosítóját helyezem az ID lábakra az 5.2. listában látható módon.

A taszk nevének eléréséhez a FreeRTOS-ban használt TCB struktúra változót kellett leszámolnom. A taszk neve előtti változók összesen 52 byte hosszúak, ebből adódik az 5.2. lista hatodik sorában található konstans ofszet.

### 5.1. lista. Ütemező azonosítójának kihelyezése a lábakra.

```

1 " push {r0, r1, r2}          \n"
2 " movw r0, #0x0C14          \n" /* GPIOD cimenek betoltese */
3 " movt r0, #0x4002          \n"
4 " ldr r2,[r0, #0]           \n"
5 " and r2, r2, #0xFFFFFFFF0F \n"
6 " str r2, [r0, #0]           \n" /* Null ertekek kiirasa a labakra */
7 " pop {r0, r1, r2}          \n"

```

### 5.2. lista. Aktuális taszk azonosítójának kihelyezése a lábakra (FreeRTOS).

```

1 " push {r0, r1, r2}          \n"
2 " ldr r0, pxCurrentTCBConst \n" /* Aktualis TCB betoltese. */
3 " ldr r2, [r0]                \n"
4 " movw r0, #0x0C14          \n" /* GPIOD cimenek betoltese */
5 " movt r0, #0x4002          \n"
6 " ldr r1, [r2, #52]           \n" /* Taszk ID kiolasas */
7 " lsl r1, r1, #4              \n" /* ID helyre mozgatasa */
8 " and r1, r1, #0x0000000F00 \n" /* Maszkolasok */
9 " ldr r2, [r0, #0]           \n"
10 " and r2, r2, #0xFFFFFFFF0F \n"
11 " orr r2, r2, r1             \n"
12 " str r2, [r0, #0]           \n" /* Ertek kihelyezese a labakra */
13 " pop {r0, r1, r2}          \n"

```

A módosítások tesztelése után a 4.4. fejezetben bemutatott taszkok implementálása következett.

Minden mérésnek új projektet létrehozni nem lett volna célszerű, ezért makrók segítségével lehet kiválasztani az aktuális mérést. A konfigurációs fájl részlete látható az 5.3. listán<sup>2</sup>.

<sup>2</sup>A BLINKING\_LED a fejlesztés során segítette a hibás működés észrevételét.

A következő lépés az ipari alkalmazás implementációja volt. minden taszk működésének ismertetése hosszadalmas – és egyben felesleges is – volna, ezért csak a fejlesztés szempontjából érdekes részleteket emelem ki.

A távoli felügyeletet soros kommunikáció segítségével valósítottam meg, amihez célszerű volt valamilyen egyszerű protokollt meghatároznom. Mivel több szenzor adatát is továbbítanom kellett, és a küldött információk sorrendje változhatott, ezért a protokollnak tartalmaznia kellett az adatok forrását. A legnagyobb továbbítandó érték 4 byte hosszú volt, ezért az üzenet formátumát az alábbi módon határoztam meg:

- Az adat forrása (szenzor neve) nyolc karakteren,
- Elválasztó karakter (:),
- Adat (4 byte),
- Lezáró karakter (sortörés).

Például: "led0\_ffff:UUUU\n".<sup>3</sup>

**5.3. lista.** Mérés kiválasztását megvalósító makrók.

```

1 /**
2  * Mérési folyamat:
3  * MEAS_LATENCY:           késleltetés mérés
4  * MEAS_TASK_SWITCHING_TIME: taszkváltási idő mérés
5  * MEAS_PREEMPTION_TIME:   preemtálási idő mérés
6  * MEAS_INTERRUPT_LATENCY_TIME: megszakítás-késleltetési idő mérés
7  * MEAS_SEMAPHORE_SHUFFLING_TIME: szemafor-váltási idő mérés
8  * MEAS_DEADLOCK_BREAKING_TIME: deadlock-feloldási idő mérés
9  * MEAS_DATAGRAM_THROUGHPUT_TIME: datagram-átviteli idő mérés
10 */
11 /**
12  * Terhelés engedélyezése:
13  * MEAS_W_LOAD
14 */
15 #define MEAS_LATENCY
16 // #define BLINKING_LED
17 // #define MEAS_W_LOAD

```

A kommunikáció során előfordul, hogy a mikrokontroller számára küldött karakter nem kerül időben fogadásra, és a periféria tárolója túlcsordul. Ekkor a periféria kezelése után az alkalmazás jelez a távoli vezérlőnek, hogy küldje újra az utolsó parancsot.

Az SD kártyára való naplázás során két probléma merült fel, melynek megoldása hosszabb időt vett igénybe. Egyrészt amikor a naplófájl mérete elérte a kártya formázása során megadott szektor méretet, akkor az írás nem folytatódott. Erre hosszas kutatás után sem találtam elfogadható megoldást, viszont a fájl módosítását végző függvények kritikusz szakaszba ágyazásával a probléma megoldódott<sup>4</sup>. A másik probléma akkor jelentkezett, ha működés közben eltávolítottam a memória kártyát, majd újból visszahelyeztem a foglalatba. A visszahelyezés után a kártyára való írás nem történt meg, a kezelő függvények

<sup>3</sup>Az UUUU karakterlánc a 0x55555555 értéknek felel meg.

<sup>4</sup>Mivel a fájl kezelése viszonylag hosszú ideig tart, ezért ez a megoldás egy valós alkalmazásban nem felelne meg!

hibával tértek vissza. A *FatFS* függvényeinek tanulmányozásával azt vettem észre, hogy tagváltozóban tárolja, hogy az adott eszköz inicializálása megtörtént-e. Az inicializálást tároló változót csak az eszközt kezelő *driver* beállításakor törl, így a kártya inicializálása a többszöri behelyezés esetén nem valósul meg. Az alkalmazásban a kártya lecsatolásakor törlöm a változó értékét.

A BLE112 Bluetooth modul kezeléséhez szükséges függvények könyvtárát a gyártó a fejlesztők számára regisztráció után elérhetővé teszi. A fejlesztőnek csak az adatok küldését és fogadását megvalósító függvényeket kell implementálnia. A fogadó függvény a bejövő adatot feldolgozza, majd meghívja az üzenethez tartozó függvényt, ahol a szükséges feladatokat elvégezhetjük.

A soros kommunikáció és a naplázás során *gatekeeper* taszkokkal valósítottam meg az erőforrások kezelését.

Az egyes taszkok működése a D. Függelékben található.

### 5.1.2. μC/OS-III

A μC/OS-III ütemezője szintén a *SysTick* eseményt használja a periodikus ütemezés megvalósítására, viszont a mechanizmus különbözik a FreeRTOS esetében megismerttől.

A Idle taszk mellet az operációs rendszer inicializálásakor létrejön *Tick* taszk is, ami periodikusan várakozik a beépített szemaforára. *SysTick* megszakítás érkezésekor a megszakítás-kezelő rutin jelez a Tick taszknak a szemaforon keresztül, majd a *Pend* függvényhívás következtében az ütemező lefut. Mivel a *Tick* taszk magas prioritással rendelkezik, ezért a késleltetés kicsi. A Tick taszk elvégzi a számlálók kezelését, majd várakozik a következő jelzésre, ezzel újabb ütemezést elindítva.

A taszkok ütemezését az *OSSched()* függvény végzi, amely szükség esetén a *PendSV* megszakítások keresztül kér kontextus váltást.

A μC/OS-III esetében az ütemező azonosítóját az *OSSched()* függvény elején, illetve a *SysTick* esemény bekövetkezésekor kellett kitennem az ID lábakra (a FreeRTOS esetében megismert utasítások segítségével – 5.1. lista).

Az *OSSched()* lefutása során két lehetőség áll fenn:

- Nem szükséges kontextusváltást kezdeményezni. Ebben az esetben az *OSSched()* függvény visszatérése előtt történik meg az aktuális taszk azonosítójának kitétele az ID lábakra.
- Kontextusváltás következik be. Ekkor a *PendSV* megszakítás befejeződése előtt kerül az azonosító kihelyezésre.

A μC/OS-III TCB struktúrája különbözik a FreeRTOS-nál használt TCB struktúrájától. Egyszerűbb a TCB nem tartalmazza a nevet, csak arra mutató pointert, másrészt a TCB kezdetétől számított ofszet is különbözik. Az ofszet értéke 32 byte, mely az 5.4. listában az ötödik sorban látható. A kinyert memóriacímről még be kell olvasni az azonosítót, ami a kilencedik sorban történik.

A μC/OS-III a szemaforok és sorok mellett támogatja a közvetlenül a taszkoknak küldött jelzések és üzenetek használatát, mely hatékonyabb futást eredményez. Mivel a két rendszer

összehasonlítása volt a cél, ezért úgy döntöttem, hogy azonos implementációt használok, és nem használom a beépített objektumokat.

Az datagram-átviteli idő taszkjainak implementációja során felmerült a probléma, hogy a μC/OS-III az üzenetre mutató pointereket használja a továbbítás során. A mérés leírásában hangsúlyozva szerepelt, hogy az átvitel ne pointer használatával történjen, ezért a mérés során a sorba az adat címe helyett magát az adatot helyeztem, és a fogadó taszk esetében is figyeltem, hogy helyesen olvassam ki az üzenet tartalmát<sup>5</sup>.

A további taszkok implementációja úgy történik, mint ahogy a FreeRTOS esetében láthattuk.

#### 5.4. lista. Aktuális taszk azonosítójának kihelyezése a lábakra ( $\mu C/OS-III$ ).

```

1 " push {r0, r1, r2}          \n"
2 " movw r1, #:lower16:OSTCBCurPtr \n"
3 " movt r1, #:upper16:OSTCBCurPtr \n"
4 " ldr r2, [r1]           \n"
5 " ldr r0, [r2, #32]      /* NamePtr betoltese */
6 " mov r2, r0             /* Regiszter masolasa */
7 " movw r0, #0x0C14      /* GPIOID cimenek betoltese */
8 " movt r0, #0x4002      \n"
9 " ldr r1, [r2]           /* Taszk ID kiolvasas */
10 " lsl r1, r1, #4        /* ID helyre mozgatasa */
11 " and r1, r1, #0x000000F0 /* Maszkolasok */
12 " ldr r2, [r0, #0]
13 " and r2, r2, #0xFFFFFFF0F
14 " orr r2, r2, r1
15 " str r2, [r0, #0]      /* Ertek kihelyezese a labakra */
16 " pop {r0, r1, r2}      \n"

```

### 5.1.3. Vezérlő szoftver

A távoli vezérlést megvalósító szoftvert Qt fejlesztőkörnyezet használatával valósítottam meg.

A vezérlő és az eszköz között a kommunikációt UART<sup>6</sup> valósítja meg a *Base Board*-on elérhető kivezetésen keresztül. A csatlakozáskor a vezérlő jelzi az eszköz számára, amire az eszköz a LED-ek és kapcsolók aktuális állapotával válaszol. A kapott értékeknek megfelelően az szoftver frissíti a felhasználói felületet.

A felületen a helyi és távoli hőmérséklet mellett a potméter állása grafikusan is megjelenítésre kerül, míg a környezeti hőmérséklet és a fényerősség értéke számokkal kerül kijelzésre. Jobb oldalon a kapcsolók és LED-ek jelenlegi állapotára utaló indikátorok láthatóak. A LED-ek értéke változtatható.

Az eszköztől érkező adatok a szenzorokból kinyert nyers adatok, melyeket a vezérlő szoftver dolgoz fel. A SensorTag által mért értékeket az adott szenzor adatlapjában meghatározott számolás alapján számoltam, míg a helyi hőmérséklet esetében a hőmérő adatlapjában található hőmérséklet-feszültség karakterisztika alapján végeztem közelítő számítást.

---

<sup>5</sup>Mivel az adattovábbítás során 32 bites adatot használtam, és az architektúra által használt pointerek is 32 bitesek, ezért a megvalósítás nem okozott problémát. Összetett struktúra átvitеле esetén bonyolultabb lett volna a helyzet.

<sup>6</sup>A soros kommunikáció beállításához a példaprogramok között megtalálható *Terminal* alkalmazást vettettem alapul.

Minden érték esetén átlagolt adat kerül megjelenítésre.

A grafikus felület az E. Függelék E.1. ábráján látható.

## 5.2. Raspberry Pi 3

A Raspberry Pi 3-on használt rendszereknél a Rhealstone értékek mérése nem megvalósítható<sup>7</sup>, és a háttérben futó több tíz – esetenként több száz – egyéb folyamat mellett nem is lenne célszerű. Ezért a két rendszer esetén a késleltetés és a legrosszabb válaszidő kerül meghatározásra, illetve szubjektív szempontok alapján értékelem majd őket.

Mintkét rendszernél két alkalmazás került implementációra:

- A szimulált ipari alkalmazás,
- Grafikus felület nélküli alkalmazás, mely a bemenetre érkező jelet a kimenetre másolja.

Az ipari alkalmazás esetén az adatok az eszközön kerülnek megjelenítésre.

### 5.2.1. Windows 10 IoT Core

A Windows 10 IoT Core rendszer telepítése a *Windows 10 IoT Core Dashboard* szoftver használatával történik. A szoftver felületén ki kell választanunk a használt fejlesztőeszközt (esetünkben Raspberry Pi 2 & 3), a telepíteni kívánt rendszert és a használt SD kártyát. Ezen felül megadhatjuk még az eszköz nevét és az adminisztrátori jelszót, illetve beállítható, hogy mely ismert WiFi hálózat beállításait szeretnénk használni az eszközön. A felhasználói feltételek elfogadását követően a szoftver letölti az operációs rendszert és telepíti az SD kártyára.

A rendszer indulása után a Raspberry Pi – amennyiben kapcsolódik a hálózatra – megjelenik a megtalált eszközök listájában. Az eszköz címét böngészőbe beírva az eszköz adminisztrációs felületére jutunk, ahol többek között kezelhetjük a futó folyamatokat és moniterezhetjük a rendszer terhelését is. Miután a böngészőn keresztül bekapsoljuk a *Remote Desktop* szolgáltatást, a *Windows IoT Remote Client* alkalmazással távoli asztalként is használhatjuk a rendszert.

A Windows 10 IoT Core lehetővé teszi *headed* és *headless* alkalmazások fejlesztését.

**Headed:** Rendelkezik grafikus felülettel. Egyszerre csak egy headed alkalmazás futhat.

**Headless:** Nem rendelkezik grafikus felülettel. A háttérben egyszerre akár több headless alkalmazás is futhat.

A fejlesztés során *Visual Studio 2017*-et használtam. Az új projekt létrehozásánál az *Windows Universal* csoporton belül az üres sablonból indultam ki. A fejlesztőkörnyezet kezelőfelülete gyors fejlesztést tesz lehetővé, így az ipari alkalmazás felhasználói felületének összeállítása nem okozott gondot<sup>8</sup>.

<sup>7</sup>A Windows 10 IoT Code forráskódja nem elérhető, és a Raspbian rendszer esetében is kernelmódosításokat kéne eszközölni.

<sup>8</sup>Raspberry Pi-n nem jelenítettem meg az értékeket külön grafikus elem használatával.

A háttérben zajló folyamatok implementációja során sem ütköztem komolyabb problémába, a legtöbb periféria beépített osztályok segítségével könnyen kezelhető volt. Viszont a Bluetooth használata nehézséget okozott. Alapos kutatás után arra jutottam, hogy a Windows 10 UWP Bluetooth API még aktív fejlesztés alatt áll, és amennyiben a kommunikáció létrehozása sikerülne, valószínűleg a továbbiakban újabb problémák merülnének fel. Ezért altenatív megoldást kerestem, és mivel a Raspberry Pi rendelkezik UART interfésszel is, ezért a BLE112 modul használata mellett döntöttem.

A Bluegiga egyik mérnökének GitHub oldalán elérhető a Bluetooth modulhoz könyvtár, mely az MIT licenc feltételei mellett használható. Az 5.1.1. fejezetben ismeretett metódusok megvalósítása után a kommunikáció az elvárt módon működött.

Az alkalmazás felülete az E. Függelék E.2. ábráján látható.

A headless alkalmazás pár sorban megvalósítható volt. Feliratkoztam a kijelölt GPIO láb változását jelző eseményre, és az eseményhez rendelt függvényben a kimeneti lábra másoltam a bemenet értékét.

### 5.2.2. Raspbian

A linux disztribúció telepítéséhez a rendszer képfájlját le kellett tölteni a Raspberry Pi hivatalos weboldaláról, majd külön szoftver segítségével kellett a fájlokat az SD kártyára másolni.

A fejlesztés megoldható lett volna asztali számítógépen, viszont a Raspbian lehetőségei ezt nem tették szükségessé. A rendszer csomagkezelőjének segítségével feltelepítettem a Qt fejlesztőkörnyezetet, és magán a Raspberry Pi-n végeztem a fejlesztést.

Mivel a Qt nem rendelkezik kiemelt támogatással a Raspberry Pi-vel kapcsolatban, ezért először az egyes perifériákat bírtam működésre.

Linux rendszerek esetén a legtöbb periféria állományként jelenik meg a fájlrendszeren belül[25]. Ez Raspbian esetén is igaz maradt, és a GPIO lábakat a `/sys/class/gpio/` elérési útvonalon érhetjük el. Az itt található `export` fájlba a használni kívánt láb számát beírva megjelenik a láb kezelését lehetővé tevő fájlokat tartalmazó mappa. Például az

```
echo 4 > /sys/class/gpio/export
```

utasítás hatására létrejön a `/sys/class/gpio/gpio4` mappa, amelyen belül az alábbi fájlok segítségével lehet a GPIO lábat kezelní:

**direction:** Meghatározza, hogy a lábat kimenetként vagy bemenetként használjuk. Lehetséges értékei: `in`, `out`.

**value:** A lab jelenlegi értékét olvashatjuk ki a fájlon keresztül. Kimenet esetén a fájlba való írással változtathatjuk meg a lab állapotát. Lehetséges értékei: `0`, `1`.

**edge:** A bemeneti lab változásakor a `value` fájlból kiolvasott érték megfelel a labon megfigyelhető jelszintnek, viszont a fájl metadatai nem változnak (mint például az utolsó módosítás dátuma). A `edge` fájl tartalmának megváltoztatásával elérhető, hogy az adott változás esetén fájlleírón keresztül detektálható legyen a változás. Lehetséges értékei: `none`, `rising`, `falling`, `both`.

A GPIO lábak kezeléséhez létrehoztam egy osztályt, ami négy tagfüggvényt tartalmaz.

**Init(GPIOPin,GPIODirection):** A paraméterként átadott lábat konfigurálja fel a szintén paraméterként megadott irányba.

**Read():** Visszatér a lab aktuális értékével.

**Write(GPIOState):** A lab értékét a paraméterként kapott értékre módosítja.

**WatchEdge(Enable):** Engedélyezi a lapon bekövetkező változás detektálását.

Az implementáció során az ismertetett állományok segítségével inicializálom a lábat. Olvasás és írás esetén a lábhoz tartozó `value` fájl tartalmát olvasom ki, illetve módosítom. Az eldetektálás engedélyezésekor az `edge` fájl írásával minden a felfutó, minden a lefutó él változásának jelzését engedélyezem, és *QFileSystemWatcher* objektum használatával figyelem a `value` fájl változását. Amennyiben a fájlon változás történik, úgy összehasonlítom a lab aktuális értékét a legutóbb kiolvasott értékkel (valóban történt-e változás), és az osztály *EdgeDetected(value)* signal-ján keresztül jelzést küldök.

Tekintettel arra, hogy Windows 10 IoT Core rendszer esetén nem a Raspberry Pi beépített Bluetooth eszközét használtam, úgy döntöttem, hogy Raspbian esetén is a BLE112 Bluetooth modul segítségével valósítom meg a vezeték nélküli kommunikációt. Ehhez szükség volt az utasítások magas szintű kezelését lehetővé tevő objektumra, amely UART-on keresztül kezeli a modult.

A soros kommunikáció használatát a Qt támogatja, és mivel az asztali alkalmazásnál is alkalmaztam, így a beállítása nem okozott gondot. Viszont a Bluetooth modulhoz nem találtam C++ fejlesztéshez használható könyvtárakat, ezért végül a C# programozás során használt könyvtár alapján implementáltam a szükséges függvényeket.

Az I<sup>2</sup>C kommunikáció megvalósításához a *wiringPi* függvénykönyvtárat használtam.

Az perifériákat egyesével tesztelve megbizonyosodtam a működésük ról, majd a tesztelés során készített programok felhasználásával összeállítottam a teljes alkalmazást. Az alkalmazás felületéről az E. Függelék E.3. ábráján láthatunk képet.

A grafikus felület nélküli alkalmazás a Raspbian esetében is néhány sorral megvalósítható volt.

## 6. fejezet

# Mérés

A méréseket *Tektronix TLA5201B* logikai analizátorral, illetve a 4.4.3. fejezetben bemutatott FPGA használatával végeztem el<sup>1</sup>.

A Tektronix TLA5201B logikai analizátor 32 csatornával, 2 Mbit memóriával rendelkezik és a *MagniVu™* technológia alkalmazásával akár 125  $ps$ -os mintavételezés is megvalósítható. A mérések során a memória mérete miatt egységesen 20  $ns$ -os mintavételi idővel dolgoztam<sup>2</sup>, de egyes méréseknél már ez a beállítás is nagy méretű kimeneti fájlt generált.

### 6.1. Mérési elrendezés

A logikai analizátor *A1* csatornájának 0–3 számozású vezetékeit a mérőkártya *ID* lábaira csatlakoztattam, míg a 6-os és 7-es vezetékeit rendre az *IN* és *OUT* lábakra kötöttem. Az analizátor beállításaiban elneveztem a vezetékeket és az *ID* lábakat csoportba rendeztem.

A késleltetés és *megszakítás-késleltetési idő* mérésénél az *IN* lábhoz tartozó jelszint változására, míg a többi mérés esetén az indító taszk azonosítójára<sup>3</sup> állítottam be a trigger-feltételt. A mérések során repetitív mérési módot alkalmaztam, melyek eredményeit minden mérés után külön fájlba mentettem.

Azon méréseknél, ahol bemeneti jel használata szükséges, ott *Hameg HMF2550* jelen-generátor szolgáltatta a gerjesztést a mérőkártyán található BNC csatlakozón keresztül. A STM32F4 fejlesztőkártyán futó rendszerek esetén 5  $kHz$ -es, a Raspberry Pi-n futó rendszerek esetén 2  $kHz$ -es négyzetjellel dolgoztam.

A mérések feldolgozását *Python* szkripttel végeztem el. Két szkript megírása vált szükségessé. Az egyik szkript a mérés során keletkezett száz fájlból készített egy darab fájlt, míg a másik szkript végezte a meghatározandó adatok kinyerését az egyesített fájlból.

<sup>1</sup>A tervezés során felmerült, hogy oszcilloszkóppal végezzem el a méréseket, viszont minden mérés többszöri elvégzése időigényes folyamat lett volna.

<sup>2</sup>A 20  $ns$  mintavételezési idő a mikrokontroller órajeléhez viszonyítva (168  $MHz$ ) pontatlan mérést tesz lehetővé, de a mérési eredmények így is jó összehasonlítási alapot nyújtanak.

<sup>3</sup>A mérést indító taszk minden esetben 2-es azonosítóval rendelkezett.

## 6.2. Adatok feldolgozása

Az analizátor kimeneti fájlja az értékeket tabulátorral választja el egymástól, ami a feldolgozás során nehezítette volna a feladatot<sup>4</sup>. Ezért az egyesítés végző szkript reguláris kifejezés segítségével pontosvesszőre cseréli az elválasztó karaktert. Az eredeti és az egyesített fájl részlete<sup>5</sup> látható a 6.1 és a 6.2. listában.

**6.1. lista.** A Tektronix TLA5201B logikai analizátor kimeneti fájljának részlete.

Sample	ID	IN	OUT
0	1	0	1
1	1	0	1
2	1	0	1

**6.2. lista.** Az egyesített fájl részlete

```
Sample;ID;IN;OUT;
0;1;0;1;
1;1;0;1;
2;1;0;1;
```

Az egyesített fájlokat feldolgozó Python szkript bemeneti paraméterként a fájl nevét és a mérés azonosítóját várja. Az argumentumok függvényében feldolgozza az adatokat, és kimeneti fájlba másolja az egyes időtartamokat  $\mu s$  mértékegységen kifejezve, illetve a fájl végén az értékek várható értékét és az emprikus szórást is feltünteti.

Az adatok várható értéke a mért értékek átlaga ((6.1) képlet), míg a tapasztalati szórás értékét a (6.2) kifejezés szerint határoztam meg.

$$\hat{\mu} = \frac{\sum_{i=1}^N t_i}{N}. \quad (6.1)$$

$$\sigma = \sqrt{\frac{\sum_{i=1}^N (t_i - \hat{\mu})^2}{N-1}}. \quad (6.2)$$

---

<sup>4</sup>A Python nyelv *csv* fájlokat feldolgozó modulja nem működik megfelelően whitespace tagoló karakterek alkalmazása esetén.

<sup>5</sup>A sor végi pontosvessző oka, hogy az első méréseknel az utolsó oszlopban a két minta között eltelt idő (fix 20 ns) is mentésre került. Később tárhely-takarékkosság miatt ezt az oszlopot töröltem.

## 7. fejezet

# Eredmények kiértékelése

### 7.1. Memóriaigény

Az Atollic TrueSTUDIO *GCC toolchain*-t használt a bináris állomány előállítására, ezáltal minden fordítás után információt kapunk a felhasznált memóriaterületekről az alábbi formában:

```
Print size information
  text      data      bss      dec      hex filename
 61752       16    55032   116800   1c840 <outputFile>
```

Minden érték byte-ban értendő. A **text** a flash memóriába kerülő adat méretét jelöli, a **data** a RAM-ban tárolt, inicializált adat mérete, míg a **bss** a RAM-ban tárolt inicializálatlan adat. Végül az összesített értéket láthatjuk decimális és hexadecimális alakban.

Az eredményeken látszik, hogy a  $\mu$ C/OS nagyobb területet használ a ROM-ból, de szignifikánsan kevesebb a RAM igénye (7.1. és 7.2. táblázat). A taszkok létrehozása minden rendszer esetén jelentős területet igényelt a flash memóriából, viszont az egyes mérések között nem változott jelentősen a memóriaigény.

Az ipari alkalmazást szimuláló taszkok hozzáadásával a két rendszer közötti összesített különbség csökkent, de különböző arányban használják a csak olvasható és az írható memóriaterületet.

**7.1. táblázat.** A FreeRTOS által elfoglalt memória az ipari alkalmazás taszkjai nélkül.

	text	data	bss	Összesített
Késleltetés	11872	16	58792	70680
Taszkváltási idő	16504	16	58816	75336
Preemptálási idő	16520	16	58816	75352
Megszakítás-késleltetési idő	16264	16	58800	75080
Szemafor-váltási idő	16440	16	58816	75272
Deadlock-feloldási idő	17048	16	58820	75884
Datagram-átviteli idő	17344	16	58820	76180

**7.2. táblázat.** A  $\mu$ C/OS-III által elfoglalt memória az ipari alkalmazás taszkjai nélkül.

	text	data	bss	Összesített
Késleltetés	17400	12	38692	56104
Taszkváltási idő	18700	12	40744	59456
Preemptálási idő	18732	12	40744	59488
Megszakítás-késleltetési idő	18460	12	39376	57848
Szemafor-váltási idő	18680	12	40140	58832
Deadlock-feloldási idő	20016	12	40792	60820
Datagram-átviteli idő	20436	12	40188	60636

**7.3. táblázat.** A FreeRTOS által elfoglalt memória az ipari alkalmazás taszkjaival.

	text	data	bss	Összesített
Késleltetés	58536	20	59452	118008
Taszkváltási idő	59008	20	59468	118496
Preemptálási idő	59024	20	59468	118512
Megszakítás-késleltetési idő	58692	20	59452	118164
Szemafor-váltási idő	58944	20	59472	118436
Deadlock-feloldási idő	59556	20	59476	119052
Datagram-átviteli idő	59132	20	59476	118628

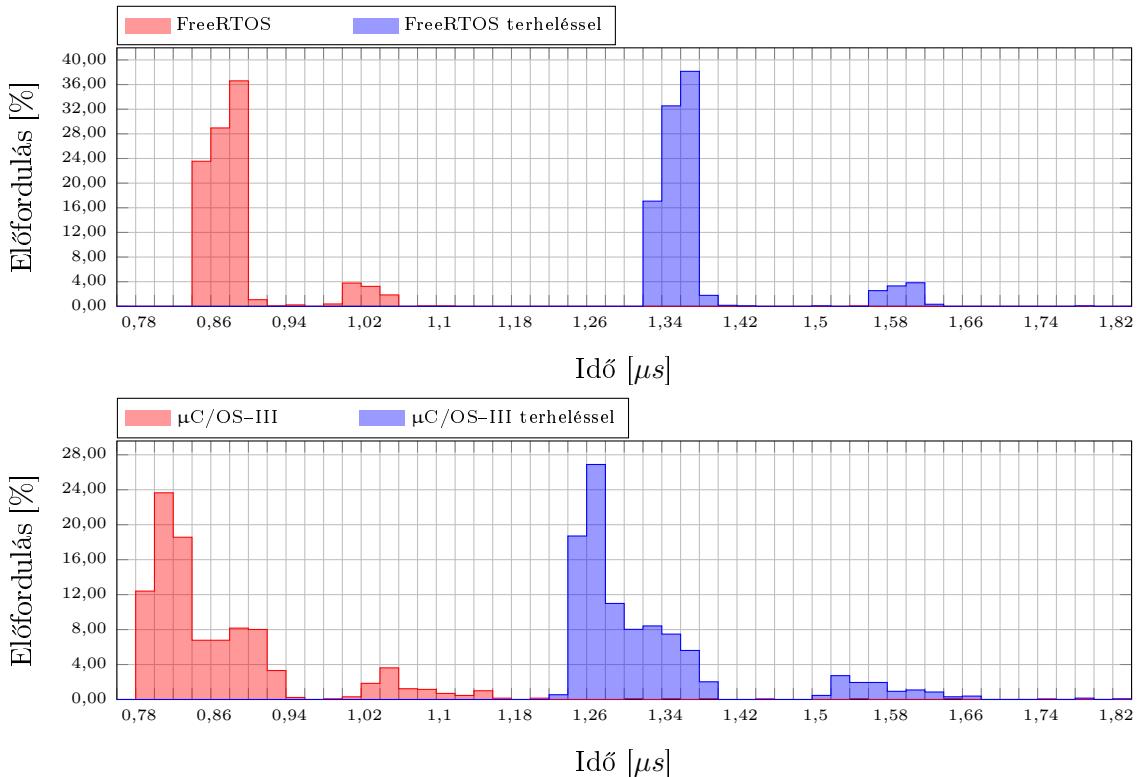
**7.4. táblázat.** A  $\mu$ C/OS-III által elfoglalt memória az ipari alkalmazás taszkjaival.

	text	data	bss	Összesített
Késleltetés	60928	16	53536	114480
Taszkváltási idő	61436	16	55584	117036
Preemptálási idő	61468	16	55584	117068
Megszakítás-késleltetési idő	61120	16	54216	115352
Szemafor-váltási idő	61412	16	54984	116412
Deadlock-feloldási idő	62752	16	55632	118400
Datagram-átviteli idő	61752	16	55032	116800

## 7.2. Mérések

A mért adatokból számolt várható értékeket és a tapasztalati szórás értékeit a 7.5. táblázat foglalja össze. A megjelenített hiszogrammok nem minden esetben tartalmazzák a teljes mintahalmazt, mert bizonyos értékek nagy szórással rendelkeztek, és a hasznos tartomány áttekinthetősége romlott volna. A táblázatban feltüntetett értékek meghatározásában azonban részt vettek. A minták száma a mérések során változó volt, és a terheléses méréseknél minden esetben kevesebb minta állt rendelkezésre, mert amennyiben a mérési folyamatot egy másik taszk futása megszakította, úgy az adat nem került felhasználásra.

A késleltetés mérésének hiszogramján (7.1. ábra) látható, hogy a  $\mu$ C/OS-III által produkált eredmények módusza kisebb értéket vesz fel, viszont szélesebb tartományban szóródnak. Mindkét operációs rendszer esetén kialakult egy második csomósodási pont ( $1,02 \mu s$  és  $1,58 \mu s$  körül), aminek oka az lehet, hogy a gerjesztésként használt jel kritikus szakasz



**7.1. ábra.** Késleltetés terheléssel és terhelés nélkül.

közben érkezett a bemenetre. A jitter értékét a mérés szorása adja.

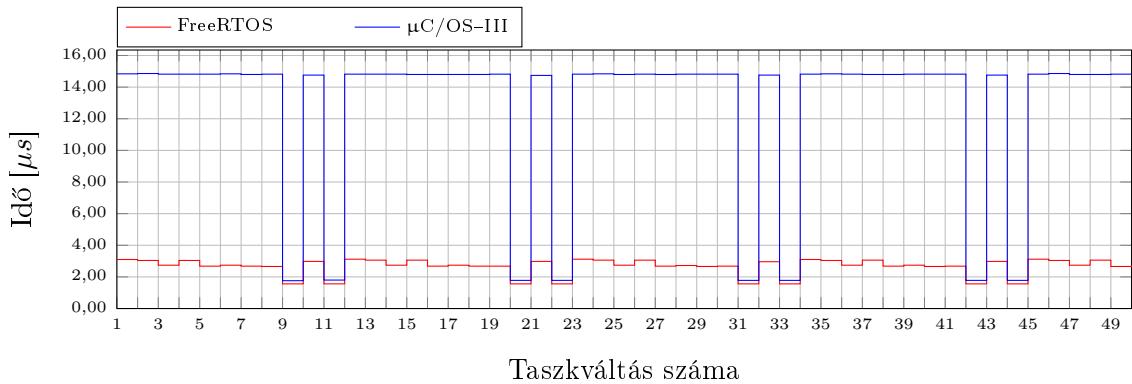
A terheléses mérés elemzésekor ritkán előfordultak kiugróan magas értékek. Ezek oka az SD kártyára való írás kritikus szakaszban történő megvalósítása volt, ezért a számolás során nem vettet figyelembe őket.

A taszkváltási idő eredményeinek elemzésekor feltűnt, hogy minkét rendszer esetében periodikusan megjelenik  $\sim 1,5 \mu s$  értékű minta (7.2. ábra). Megkeresve, hogy mely váltásokhoz tartoznak az értékek kiderült, hogy amikor az egyik taszk futása véget ér – és ezáltal önként mond le a processzor használatáról –, akkor következnek be az említett gyors váltások. Mivel a mérés leírásában az ütemezést aktív taszkok között kell elvégeznie az operációs rendszernek, ezért ezeket a mintákat kiszűrtem. Az így kapott eredmény látható a 7.3. ábrán.

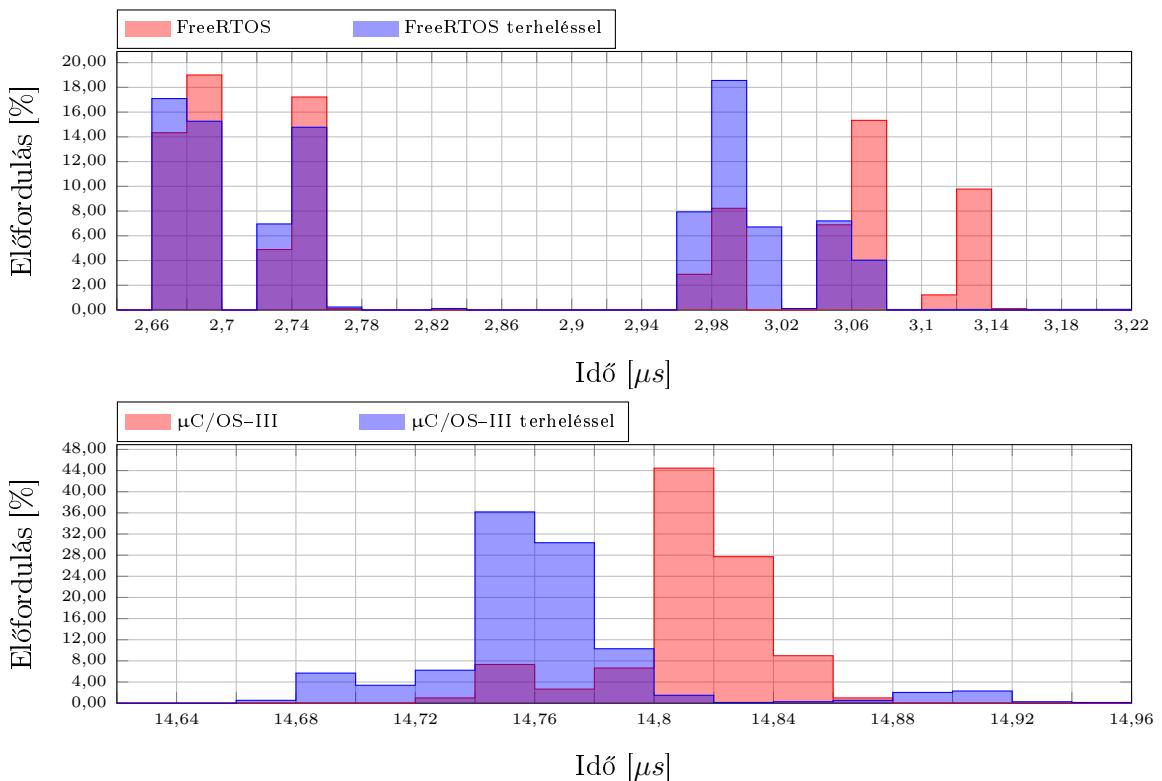
A FreeRTOS taszkváltásai a  $\mu C/OS-III$  esetében tapasztalt értékek töredéke alatt bekövetkezik. A két eredmény közti különbség oka lehet, hogy a  $\mu C/OS-III$  *SysTick* esemény hatására az ütemezőt többször is lefuttatja.

Mindkét rendszer esetén érdekes tapasztalat, hogy mikor a terhelést végző alkalmazás is futott, akkor nem hogy lassult az ütemezés, hanem kis mértékben még gyorsabban is futott le.

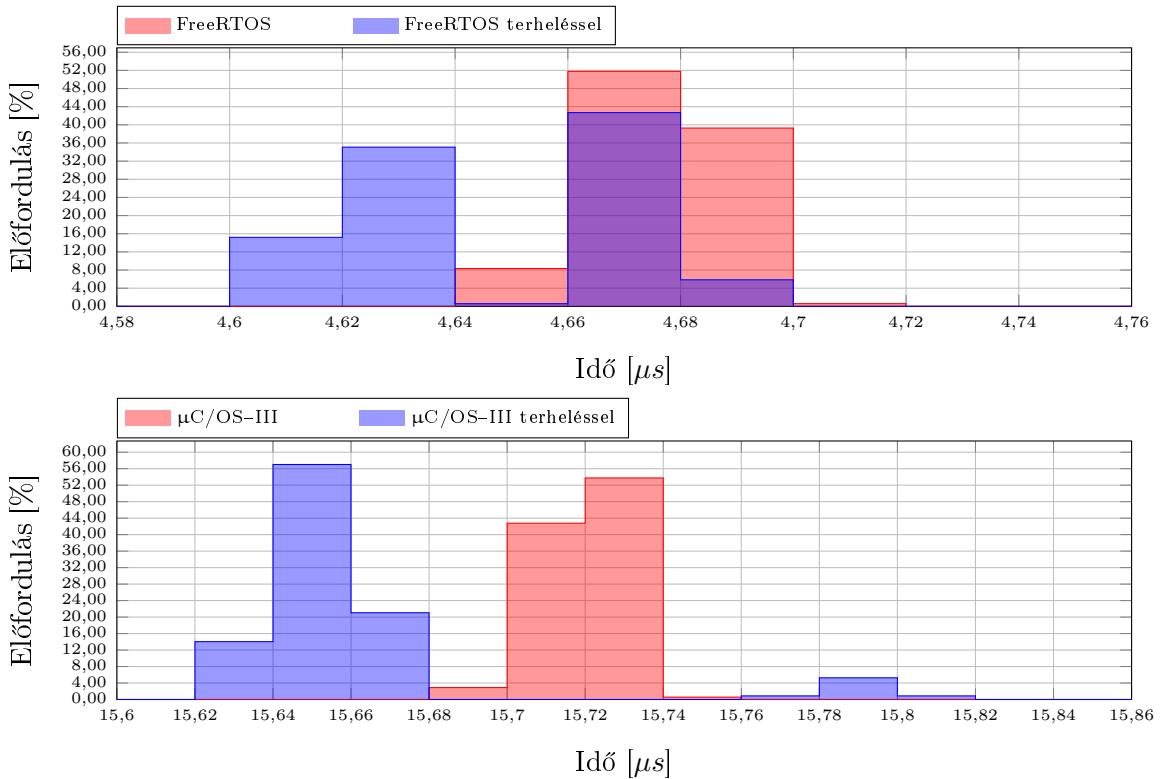
A preemptálási idő eredményei hasonlóak a taszkváltási időnél tapasztaltakkal. Ez abból a szempontból nem meglepő eredmény, hogy a preemptálási idő várhatóan nagyobb lesz a taszkváltási időnél. Viszont, ha megnézzük, hogy a két rendszer preemptálási ideje mennyivel lassabb a taszkváltási idejénél, akkor a  $\mu C/OS-III \sim 0,9 \mu s$ -ot, míg a FreeRTOS



**7.2. ábra.** Taszkváltási idő az egyes taszkváltások esetén.



**7.3. ábra.** Taszkváltási idő terheléssel és terhelés nélkül.



**7.4. ábra.** Preemptálási idő terheléssel és terhelés nélkül.

$\sim 1,8 \mu s$ -ot lassult (a várható értékekből számolva).

A megszakítás-késleltetési idő hisztogramjainak alakja hasonló a késleltetés mérésénél kapott görbék alakjával, viszont – ahogy az várható volt – a móduszok nagyobb (közel tízszeres) értéket vesznek fel.

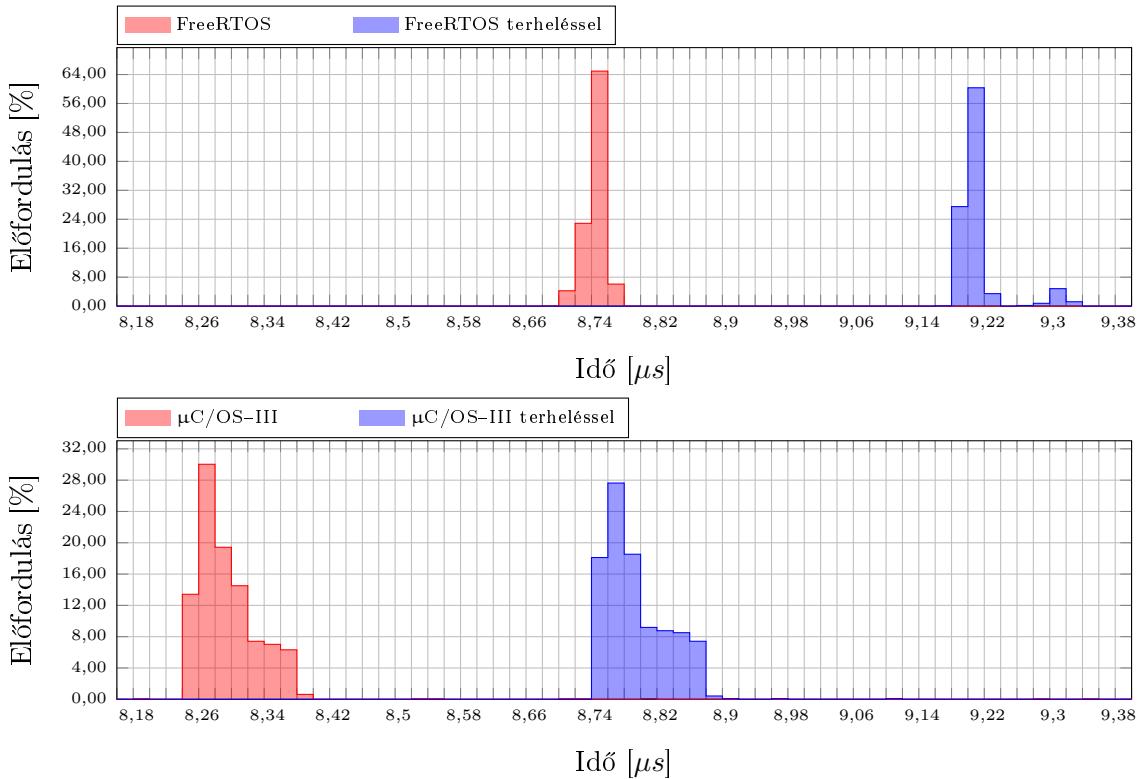
A szemafor-váltási időt minden rendszernél nagyon kicsi szórás jellemzi. Ennél a mérsénél is megfigyelhető, hogy terhelés hatására kis mértékben gyorsabban végrehajtják az operációs rendszerek az objektum kezelését.

A deadlock-feloldási idő esetében sem szignifikáns a két operációs rendszer között a különbség, viszont amíg a μC/OS-III futását látszólag nem befolyásolja az egyéb taszkok futása, addig a FreeRTOS esetében kis mértékű lassulás tapasztalható.

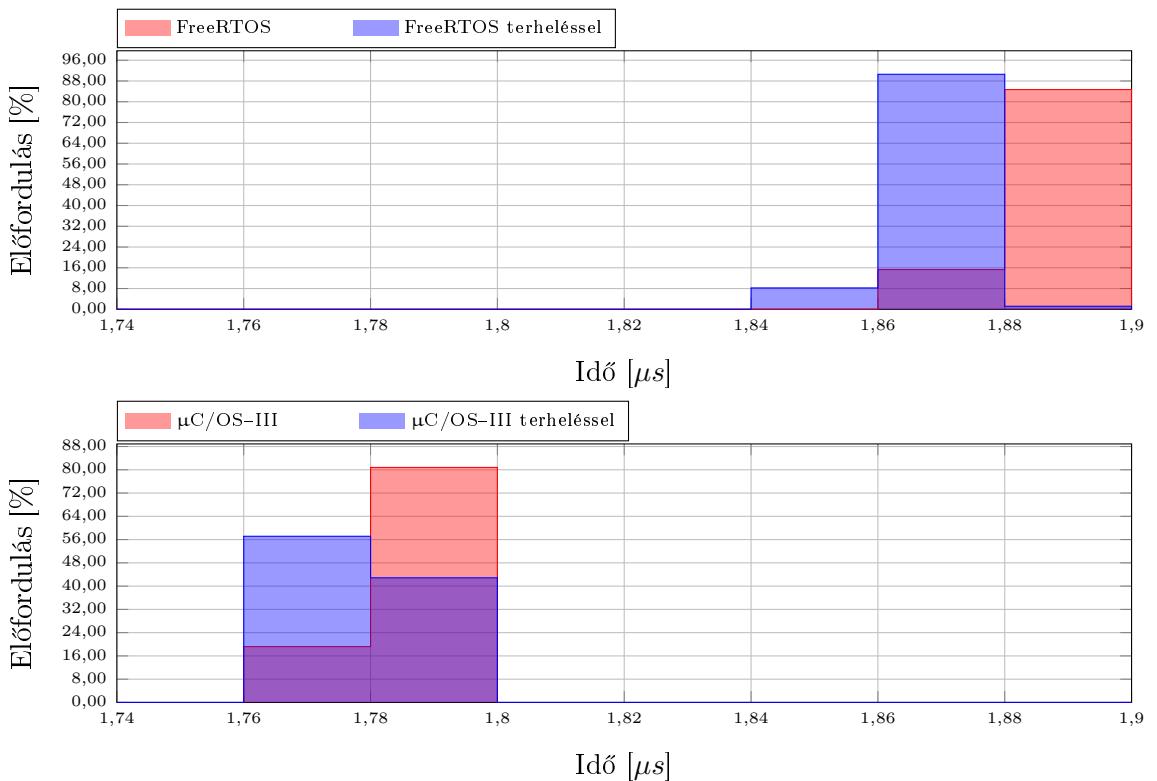
Nagyobb mennyiségű adat átvitelében a μC/OS-III jelentősen jobb eredményt mutatott. Mindkét rendszer rövidebb idő alatt végrehajtotta az adatátvitelt, mikor az ipari alkalmazás szimulációját végző taszkok is futottak.

A Raspberry Pi 3 rendszerei esetén a megszakítás-késleltetési idő mérésénél kapott értékek magasabbak, és nagy szórással rendelkeznek (7.9. ábra). Grafikus felület használata esetén minden rendszernél látható, ahogyan a minták széles tartományon terülnek szét. Raspbian esetében a grafikus felület kikapcsolásával az adathalmaz módusza egyértelművé válik, és az adatok szórása is lecsökken. A mérést megnövelte prioritással is elvégezve a módusz értéke csökken – ezzel együtt a várható érték is –, de az adathalmaz szórására nincs pozitív hatással.

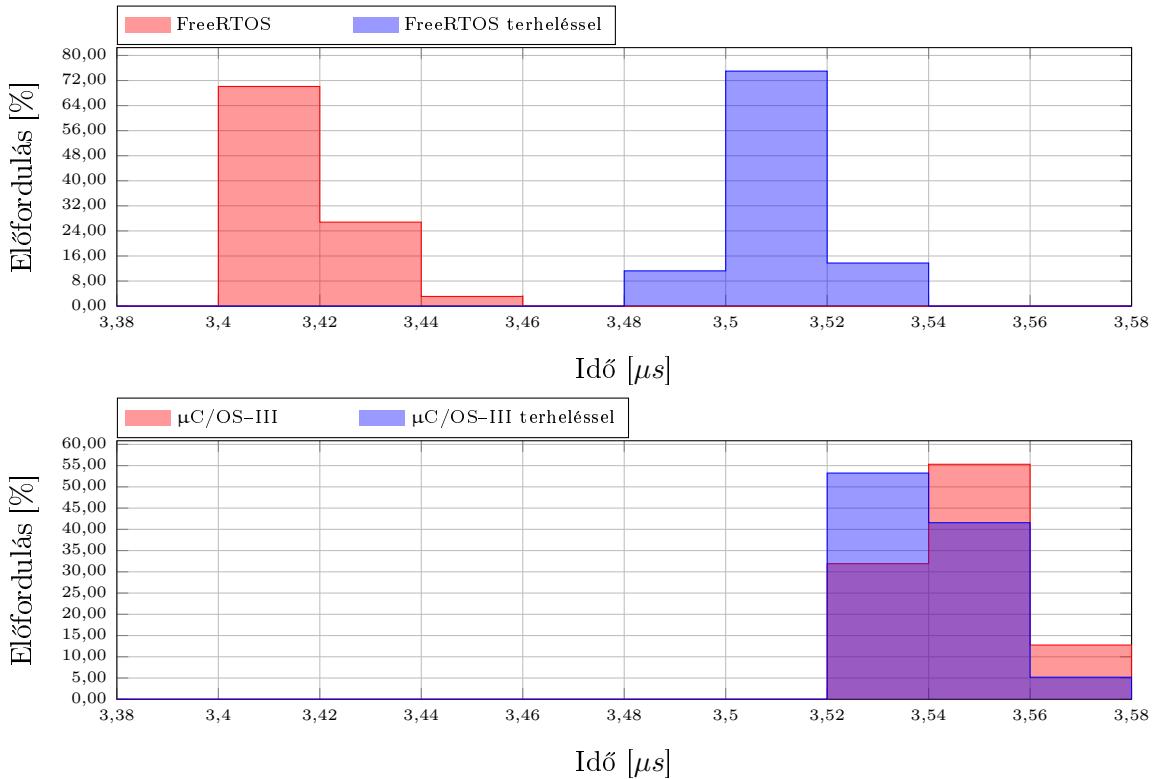
Windows 10 IoT Core esetén a grafikus felület mellőzése nem okozott jelentősebb vál-



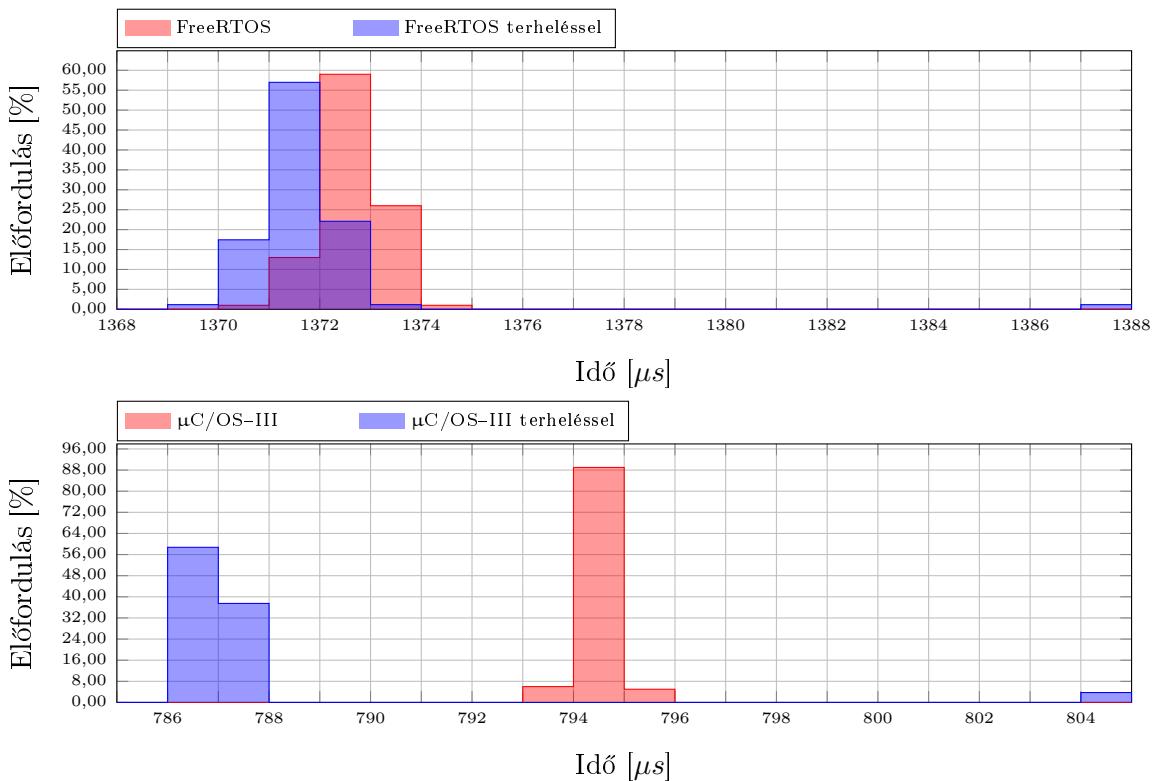
**7.5. ábra.** Megszakítás-késleltetési idő terheléssel és terhelés nélkül.



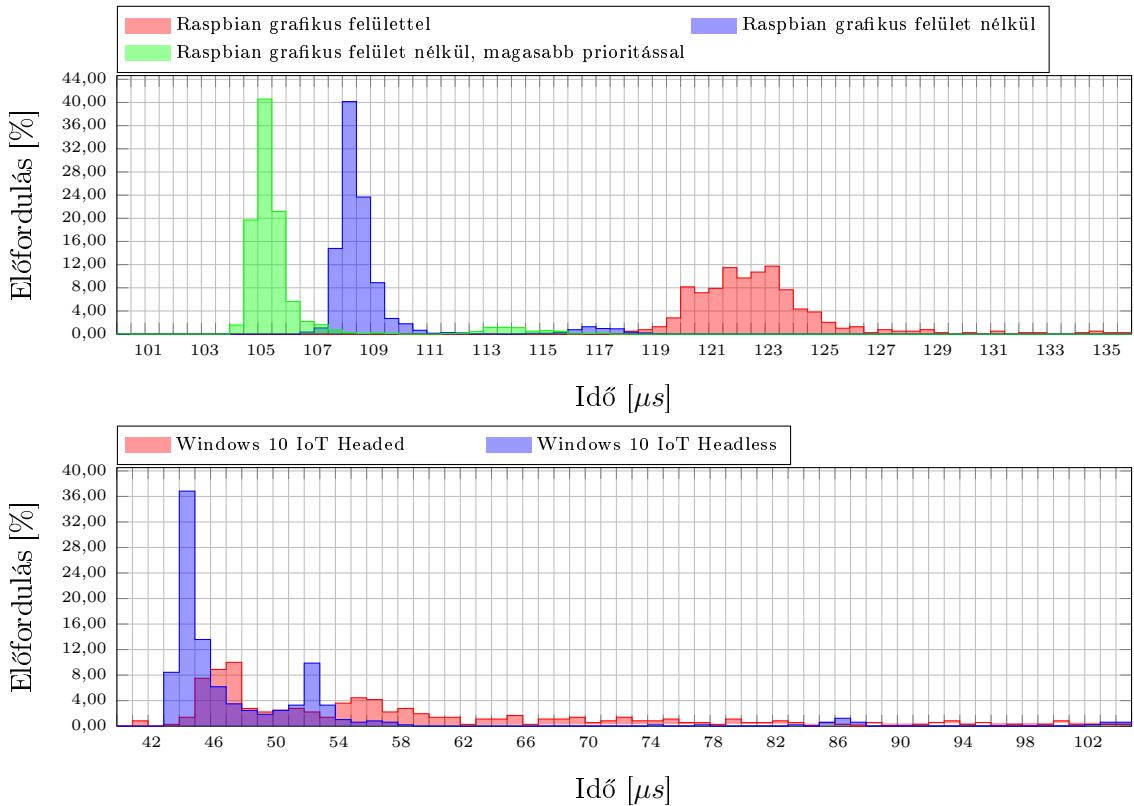
**7.6. ábra.** Szemafor-váltási idő terheléssel és terhelés nélkül.



7.7. ábra. Deadlock-feloldási idő terheléssel és terhelés nélkül.



7.8. ábra. Datagram-átviteli idő terheléssel és terhelés nélkül.



**7.9. ábra.** Késleltetés a Raspberry Pi-n futó rendszerekkel terheléssel és terhelés nélkül.

tozást a módusz tekintetében, viszont a 7.9. ábrán látható, hogy az értékek sokkal inkább csoportosulnak a módusz közelében, viszont ekkor is két kiugró érték van jelen a hisztogrammon.

**7.5. táblázat.** Az STM32F4 Discovery fejlesztőkártyán futó rendszerek mérések eredményei táblázatba rendezve. A várható érték és szórás  $\mu\text{s}$  mértékegységben értendő.

Mérés	FreeRTOS			$\mu$ C/OS-III		
	$\hat{\mu}$	$\sigma$	Minta	$\hat{\mu}$	$\sigma$	Minta
Késleltetés	0,88	0,12	1300	0,88	0,22	1300
Késleltetés terheléssel	1,39	0,38	1183	1,32	0,15	1285
Taszkváltási idő	2,85	0,18	900	14,80	0,04	902
Taszkváltási idő terheléssel	2,85	0,23	819	14,75	0,05	738
Preemptálási idő	4,67	0,01	168	15,71	0,01	173
Preemptálási idő terheléssel	4,65	0,12	171	15,66	0,13	114
Megszakítás-késleltetési idő	8,80	0,49	1298	8,47	1,59	1298
Megszakítás-késleltetési idő terheléssel	9,25	0,50	1162	9,01	1,75	1208
Szemafor-váltási idő	1,88	0,01	98	1,78	0,01	99
Szemafor-váltási idő terheléssel	1,86	0,01	85	1,77	0,01	63
Deadlock-feloldási idő	3,41	0,01	97	3,54	0,01	94
Deadlock-feloldási idő terheléssel	3,50	0,01	80	3,53	0,01	77
Datagram-átviteli idő	1371,64	0,65	100	793,53	0,32	100
Datagram-átviteli idő terheléssel	1370,72	1,82	86	786,54	3,32	80

**7.6. táblázat.** A Raspberry Pi 3 eszközön futó rendszerek mérések eredményei táblázatba rendezve. A várható érték és szórás  $\mu s$  mértékegységben értendő.

Mérés	Windows10 IoT Core			Raspbian		
	$\hat{\mu}$	$\sigma$	Minta	$\hat{\mu}$	$\sigma$	Minta
Késleltetés	90,40	119,05	393	125,01	28,31	396
Késleltetés grafikus felület nélkül	51,60	54,71	488	109,32	9,70	1997
Késleltetés megnövelt prioritás esetén				106,26	11,03	1996

A legrosszabb válaszidő meghatározásánál követtem a leírt módszer lépéseit. FreeRTOS és  $\mu$ C/OS-III esetében a frekvenciát  $kHz$ -ben, két tizedesjegy pontossággal kerestem meg, míg a Raspberry Pi rendszereinél csak tíz hertz pontosságot használtam<sup>1</sup>.

A 7.7. táblázatban foglalt eredmények leginkább a Raspberry Pi rendszerei esetén érdekesek. A Windows 10 IoT annak ellenére, hogy a 7.9. ábra alapján jobban teljesít a Raspbian rendszernél, már nagyon alacsony frekvenián is téveszt. Ezzel szemben grafikus felület nélkül a Raspbian nagyságrendekkel stabilabb működést mutat, mint amit a Raspberry Pi 3-on elvégzett többi mérés esetén tapasztaltam.

**7.7. táblázat.** A legrosszabb válaszidő mérésének eredményei.

	Legmagasabb frekvencia
FreeRTOS	26,27 kHz
$\mu$ C/OS-III	18,23 kHz
Windows 10 IoT Core Headed	40 Hz
Windows 10 IoT Core Headless	50 Hz
Raspbian grafikus felülettel	70 Hz
Raspbian grafikus felület nélkül	1,2 kHz

A FreeRTOS és  $\mu$ C/OS-III értékeit egyszerű számolás segítségével ellenőrizhetjük. A megszakítás-késleltetési idő mérésénél a mérési adatok alapján a legrosszabb esetben  $13,20 \mu s$  és  $24,98 \mu s$  késleltetést okoztak a rendszerek. Ezeket az értékeket felhasználva a (7.1) és a (7.2) kifejezések eredménye felső korlátot ad a legmagasabb frekvenciának<sup>2</sup>. A  $\mu$ C/OS-III esetében számolt  $20016 kHz$  nagyon közel áll a mért értékhez. A FreeRTOS esetében tapasztalt eltérés csak annyit jelent, hogy a megmért adatok nem tartalmaznak a legrosszabb válaszidőhöz közeli értékeket.

$$\frac{1}{2 \cdot 13,20 \mu s} \approx 37878,79 kHz. \quad (7.1)$$

<sup>1</sup>Az eredményekből látszik, hogy kifejezetten alacsony értékek adódtak. Ekkora frekvenciánál csak percek múltával derült ki, hogy az eszköz téveszt vagy sem. A pontosabb mérés aránytalanul sok időbe telt volna.

<sup>2</sup>A késleltetés nem haladhatja meg a félperiódus hosszát, különben egy egész periódus elveszik. Ennek következménye a nevezőben levő szorzó.

$$\frac{1}{2 \cdot 24,98 \mu s} \approx 20016,01 \text{ } kHz. \quad (7.2)$$

Az objektív Rhealstone értékek meghatározásának az ipari alkalmazás taszkjai nélkül elvégzett méréseket vettet alapul, és az eredmények alsó egész részét vettet.

**7.8. táblázat.** *Objektív Rhealstone értékek.*

Mérés	FreeRTOS	$\mu$ C/OS-III
Taszkváltási idő	350877	67567
Preemptálási idő	214132	63653
Megszakítás-késleltetési idő	113636	118063
Szemafor-váltási idő	531914	561797
Deadlock-feloldási idő	293255	282485
Datagram-átviteli idő	729	1260
Objektív Rhealstone érték	1504543	1094825

## 8. fejezet

# Konklúzió

Az objektív eredményeket tekintve a FreeRTOS és a  $\mu$ C/OS-III között a taszkváltási időben, a preemptálási időben és datagram-átviteli időben tapasztalható szignifikáns eltérés. A dolgozat végén számolt objektív Rhealstone érték a FreeRTOS számára kedvez, de nem szabad figyelmen kívül hagyni, hogy a  $\mu$ C/OS-III esetében nem használtam az operációs rendszer által nyújtott szolgáltatások tetemes részét, amik hatékonyabb használatot tettek volna lehetővé.

Sem a FreeRTOS, sem a  $\mu$ C/OS-III használata során nem ütköztem megoldhatatlan problémába, viszont a  $\mu$ C/OS első indítása több időt vett igénybe az elvégzendő portolási lépések következtében.

A FreeRTOS operációs rendszer tökéletes belépési lehetőség azoknak, akik előtte még nem használtak beágyazott eszközökön operációs rendszer, tekintve, hogy az STM32CubeMX szoftverrel percek alatt sikeresen lehet részük, és a rendszer által nyújtott szolgáltatások segítségével a legtöbb alkalmazás megvalósítható. Bár a rendszer nem tartalmaz különböző kiegészítő szolgáltatásokat – mint például TCP/IP stack –, de ezek a legtöbb esetben külső forrásokból elérhetők.

A  $\mu$ C/OS-III érezhetően összetettebb rendszer. Elérhető hozzá USB és TCP/IP stack, melyet a gyártó az operációs rendszerhez optimalizál. Mivel a  $\mu$ C/OS kereskedelmi céllal történő felhasználása jogdíj köteles, ezért inkább bonyolult termékek fejlesztése esetén érdemes elgondolkodni, amikor az egyes funkciók más rendszeren való implementálása már aránytalanul sok többletmunkát igényelne.

A Windows 10 IoT Core használata során pozitív tapasztalat volt a C# nyelv által elérhető hatékonyság. Az *async* metódusok használatával nem kellett az egyes szálak létérehozásával törődni, és az *event*-ek segítségével könnyen kezelhettem a beérkező adatokat. A fejlesztőrendszer több helyen is elvégzi a programozó helyett a munka nehéz részét. Viszont az alkalmazás implementálása végén szembesültem a tényel, hogy a felsorolt előnyök mellett hátránya is van a UWP alkalmazás fejlesztésének. Például a felhasználó nem tudja bezárni az alkalmazást, ezáltal a fejlesztőnek nincs lehetősége értesítést kapni arról, hogy az alkalmazást leállították. Természetesen megkerülő megoldásokkal megoldható a probléma, de ezt mindekképp a rendszer hátrányaként említeném.

A Windows 10 IoT Core rendszert a C# programozásban jártas fejlesztőknek ajánlom,

főleg kijelzővel rendelkező eszközökhöz. Viszont amennyiben az alkalmazásban időkritikus feladat is szerepel, akkor érdemes más platformot is számításba venni.

A Raspbian rendszerre való fejlesztést magán a Raspberry Pi eszközön végeztem, és nem találkoztam különösebb problémával. A Raspberry Pi-re való fejlesztés elsődleges nyelve a Python, ezért a legtöbb funkcióhoz elérhetőek Python modulok. Mivel én Qt-t használtam a fejlesztés során, így volt, amit magamnak kellett megoldanom – például a GPIO lábak kezelése –, amit Pythonban könnyedén megvalósíthattam volna.

Mivel a Raspbian-hoz elérhetőek kernel patch-ek, melyekkel a rendszer real-time működése megvalósítható, ezért úgy gondolom, hogy akár folyamatirányítási feladat ellátására is alkalmas eszköz lehet.

## 8.1. További lehetőségek

A mérőkártya használata során apróbb tervezési hibák derültek ki. Ilyen hiba volt az ADS7924 bemeneti lábára helyezett nagy értékű kondenzátor, ami a hőmérő értékének olvasásakor a potméter állásától függő értékeket eredményezett.

A FreeRTOS és a μC/OS-III esetében a taszkok azonosítónak kihelyezése a lábra assembly nyelven történt. Ezáltal az utasítások végrehajtási ideje pontosan meghatározható, és a mérési eredmények korrigálhatóak. Ez a végrehajtott feldolgozás során nem valósult meg, de szükség esetén kivitelezhető.

A μC/OS-III mérését megvalósító taszkok nem használják ki az operációs rendszer előnyeit, és ez torzítja a rendszer eredményeit. A méréseket célszerű lenne elvégezni az említett szolgáltatások használatával is.

A diploma során elkészített projektek és a felhasznált dokumentumok a <https://github.com/Lyque/diplomaterv> oldalon elérhetőek.

# Köszönetnyilvánítás

Szeretnék köszönetet mondani családomnak azért a mérhetetlen türelemért és megértésért, amit a dolgozat készítése alatt tanúsított, és szeretném megköszönni munkatársaimnak a tervezés során adott tanácsokat.

# Ábrák jegyzéke

1.1.	Látszólag a folyamatok párhuzamosan futnak. . . . .	11
1.2.	A valóságban minden folyamat egy kis időszeletet kap a processzortól. . . . .	12
1.3.	Multilevel queue ütemezés. . . . .	14
1.4.	Szinkronizáció bináris szemafor segítségével[4]. . . . .	16
1.5.	Esemény bekövetkezésének elvezetése bináris szemafor használata során. . . . .	17
1.6.	Számláló szemafor működésének szemléltetése[4]. . . . .	18
1.7.	Mutex működésének szemléltetése[4]. . . . .	21
1.8.	Sor működésének szemléltetése[4]. . . . .	22
1.9.	Prioritás inverzió jelensége. . . . .	22
1.10.	Prioritás öröklés, mint a prioritás inverzió egyik megoldása. . . . .	23
1.11.	Holtponti helyzet kialakulásának egy egyszerű példája. . . . .	23
2.1.	STM32F4 Discovery fejlesztőkártya[9]. . . . .	25
2.2.	STM32F4 Discovery Base Board kiegészítő kártya[10]. . . . .	26
2.3.	Raspberry Pi 3 bankkártya méretű PC[12]. . . . .	26
2.4.	Az UBM Tech által 2015-ben publikált beágyazott operációs rendszer használati statisztika[13]. . . . .	28
2.5.	Taszk lehetséges állapotai a FreeRTOS rendszerben (egyszerűsített). . . . .	30
2.6.	Taszk lehetséges állapotai a FreeRTOS rendszerben. . . . .	30
2.7.	Megszakítások késleletetett feldolgozásának szemléltetése. . . . .	34
2.8.	A heap_1.c implementációjának működése. . . . .	37
2.9.	A heap_2.c implementációjának működése. . . . .	38
2.10.	Taszk lehetséges állapotai a μC/OS-III rendszerben. . . . .	39
3.1.	Az operációs rendszer késleltetésének szemléltetése. . . . .	45
3.2.	A késleltetés jitterének szemléltetése. . . . .	45
3.3.	A tasztáltási idő szemléltetése. . . . .	46
3.4.	A preemptálási idő szemléltetése. . . . .	47
3.5.	A megszakítás-késleltetési idő szemléltetése. . . . .	47
3.6.	A szemafor-váltási idő szemléltetése (1989-es meghatározás alapján). . . . .	48
3.7.	A szemafor-váltási idő szemléltetése (1990-es meghatározás alapján). . . . .	48
3.8.	A deadlock-feloldási idő szemléltetése. . . . .	49
3.9.	A datagram-átviteli idő szemléltetése. . . . .	49
3.10.	A taszk közötti üzenet-késleltetési idő szemléltetése. . . . .	50

3.11. A legrosszabb válaszidő mérési összeállítása. . . . .	51
4.1. Az elkészült mérőkártya. . . . .	57
4.2. StartMeasure taszk folyamatábrája. . . . .	58
4.3. Taszkváltási idő méréséhez használt taszkok folyamatábrája. . . . .	59
4.4. Preemptálási idő méréséhez használt taszkok folyamatábrája. . . . .	60
4.5. Megszakítás-késleltetési idő méréséhez használt taszk folyamatábrája. . . . .	61
4.6. Szemafor-váltási idő méréséhez használt taszkok folyamatábrája. . . . .	61
4.7. Deadlock-feloldási idő méréséhez használt taszkok folyamatábrája. . . . .	62
4.8. Datagram-átviteli idő méréséhez használt taszkok folyamatábrája. . . . .	63
4.9. Basys 3 fejlesztőkártya[24]. . . . .	64
4.10. A bemeneti jelek éldetektálását végző áramkör. . . . .	65
 B.1. A mért rendszer csatlakoztatását megvalósító tüskesort és az egyes modulokat tartalmazó rajz. . . . .	98
B.2. Tápfeszültség előállítása és jelzése LED-ek használatával. . . . .	98
B.3. A helyi állomáson elhelyezett hőmérő és potméter bekötése. . . . .	99
B.4. ADC IC bekötése. . . . .	99
B.5. BLE112 Bluetooth modul és a hozzá csatlakozó programozó tüskesor bekötése. . . . .	99
B.6. SD kártya foglalat kivezetése a tüskesorra. . . . .	100
B.7. Vezérelhető LED-ek és kapcsolók bekötése. . . . .	100
B.8. A mérési folyamatot egyszerűsítő kivezetések bekötése. . . . .	101
 C.1. Alkatrész oldali rajzolat. . . . .	102
C.2. Forrasztási oldali rajzolat. . . . .	102
 D.1. BLE112 modultól érkező válaszok feldolgozását taszk folyamatábrája. . . . .	103
D.2. Analog-Digital konverziót végző taszkok folyamatábrája. . . . .	104
D.3. A LED-ek állapotát vezérlő taszkok folyamatábrája. . . . .	104
D.4. Kapcsolók változását kezelő taszkok folyamatábrája. . . . .	105
D.5. BLE112 modult vezérlő taszk folyamatábrája. . . . .	106
D.6. BLE112 modulnak küldött üzenetek továbbítását végző taszk folyamatábrája. . . . .	107
D.7. A vezérlőnek küldött üzenetek továbbítását végző taszk folyamatábrája. . . . .	107
D.8. A vezérlő felöl érkező üzenetek feldolgozását végző taszk folyamatábrája. . . . .	108
D.9. A vezérlő program csatlakozását kezelő taszk folyamatábrája. . . . .	109
 E.1. Távoli vezérlést végző felhasználói felület. . . . .	110
E.2. Windows 10 IoT Core alkalmazás felhasználói felülete. . . . .	110
E.3. Raspbian alkalmazás felhasználói felülete. . . . .	111

# Táblázatok jegyzéke

2.1.	A FreeRTOS TCB-jének főbb változói. . . . .	29
2.2.	A µC/OS-III TCB-jének főbb változói. . . . .	40
7.1.	A FreeRTOS által elfoglalt memória az ipari alkalmazás taszkjai nélkül. . .	76
7.2.	A µC/OS-III által elfoglalt memória az ipari alkalmazás taszkjai nélkül. . .	77
7.3.	A FreeRTOS által elfoglalt memória az ipari alkalmazás taszkjaival. . . . .	77
7.4.	A µC/OS-III által elfoglalt memória az ipari alkalmazás taszkjaival. . . . .	77
7.5.	Az STM32F4 Discovery fejlesztőkártyán futó rendszerek mérések eredményei táblázatba rendezve. A várható érték és szórás $\mu s$ mértékegységben értendő. . . . .	83
7.6.	A Raspberry Pi 3 eszközön futó rendszerek mérések eredményei táblázatba rendezve. A várható érték és szórás $\mu s$ mértékegységben értendő. . . . .	84
7.7.	A legrosszabb válaszidő mérésének eredményei. . . . .	84
7.8.	Objektív Rheatlstone értékek. . . . .	85

# Irodalomjegyzék

- [1] Open Source Initiative. Licenses by Name. <https://opensource.org/licenses/alphabetical>. Megtekintve: 2017. május 7. 6:48.
- [2] Bellevue Linux Users Group. The Linux Information Project. <http://www.lininfo.org/bsdlicense.html>. Megtekintve: 2017. május 6. 22:33.
- [3] Real Time Engineers ltd. What is an RTOS? <http://www.freertos.org/about-RTOS.html>. Megtekintve: 2017. május 7. 6:49.
- [4] Richard Barry. *Mastering the FreeRTOS<sup>TM</sup> Real Time Kernel*. 2016.
- [5] Jerry Breecher. Operating System Scheduling. Worcester Polytechnic Institute.
- [6] Geoffrey M. Voelker. Principles of Operating Systems. UCSD Department of Computer Science and Engineering, 2002.
- [7] Szabó Zoltán. Beágyazott operációs rendszerek jegyzet. Budapesti Műszaki és Gazdaságtudományi Egyetem, 2015.
- [8] STMicroelectronics. *Reference manual*, 9. 2016. Rev 13.
- [9] ELEKTRICKS. STM32F4-Discovery + ChibiOS = Data Acquisition System. <http://www.elektricks.net/stm32f4-discovery-chibios-data-acquisition-system/>. Megtekintve: 2017. május 11. 20:01.
- [10] SEMICONSPACE. STM32F4DIS-BB :: ST MICRO :: BOARD BASE STM32F4 DISCOVERY. <https://semiconspace.com/product/stm32f4dis-bb-st-micro-board-base-stm32f4-discovery/>. Megtekintve: 2017. május 11. 20:05.
- [11] Raspberry Pi Foundation. Putting the power of digital making into the hands of people all over the world. Strategy 2016–2018.
- [12] PIMORONI. Raspberry Pi 3. <https://shop.pimoroni.com/products/raspberry-pi-3>. Megtekintve: 2017. május 11. 20:59.
- [13] Rich Quinnell. 2015 Embedded Markets Study. 2015.
- [14] Real Time Engineers ltd. License Details. <http://www.freertos.org/a00114.html>. Megtekintve: 2017. május 7. 18:27.

- [15] Micriµm. *µC/OS-III™ The Real-Time Kernel*. Micriµm Press, 2012.
- [16] Colin Walls. Why and how to measure your RTOS performance. <http://www.embedded.com/design/operating-systems/4437792/Why-and-how-to-measure-your-RTOS-performance>. Megtekintve: 2017. május 7. 18:59.
- [17] Wolfgang A. Halang Roman Gumzej. *Real-time Systems' Quality of Service*. Springer-Verlag London, 2010.
- [18] Larisa Rizvanovic. Comparison between Real time Operative systems in hardware and software, 2001.
- [19] James Bieman Norman Fenton. *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 2012. Third edition.
- [20] Glauco Caurin Rafael V. Aroca. A Real Time Operating System (RTOS) Comparison. 2009.
- [21] Kent Porter Rabindra P. Kar. Rhealstone: A Real-Time Benchmarking Proposal. *Dr. Dobb's Journal*, 1989. <http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1989/8902/8902a/8902a.htm>. Megtekintve: 2017. május 7. 21:15.
- [22] Rabindra P. Kar. Implementing the Rhealstone Real-Time Benchmark. *Dr. Dobb's Journal*, 1990. <http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/DDJ/1990/9004/9004d/9004d.htm>. Megtekintve: 2017. május 7. 21:15.
- [23] Paul Baracos Joseph K. Dupré. Benchmarking Real-time Determinism. 2001. <https://web.archive.org/web/20140328155945/http://www.isa.org/~pmcd/acs/brtd.html>. Megtekintve: 2017. május 7. 21:15.
- [24] DIGILENT. Basys 3. <https://reference.digilentinc.com/reference/programmable-logic/basys-3/start>. Megtekintve: 2017. május 11. 20:05.
- [25] Bányász Gábor. Beágyazott operációs rendszerek jegyzet. Budapesti Műszaki és Gazdaságtudományi Egyetem, 2015.

# Függelékek

## A. Függelék

# A rövidebb licencek eredeti szövegei

### A.1. MIT License

The MIT License (MIT)

Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

### A.2. BSD

#### A.2.1. 4-clause BSD (eredeti)

Copyright (c) <year> <copyright holder>. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright

notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement:

This product includes software developed by the <organization>.

4. Neither the name of <copyright holder> nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY COPYRIGHT HOLDER "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### A.2.2. 3-clause BSD (módosított)

Copyright (c) <YEAR>, <OWNER>

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS

SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

### A.2.3. 2-clause BSD (egyszerűsített)

Copyright (c) <YEAR>, <OWNER>  
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

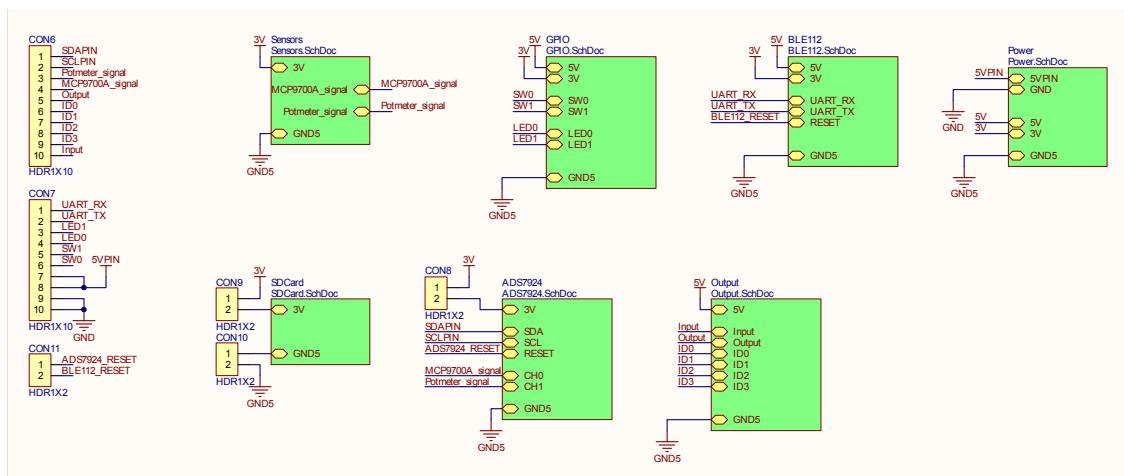
1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

## B. Függelék

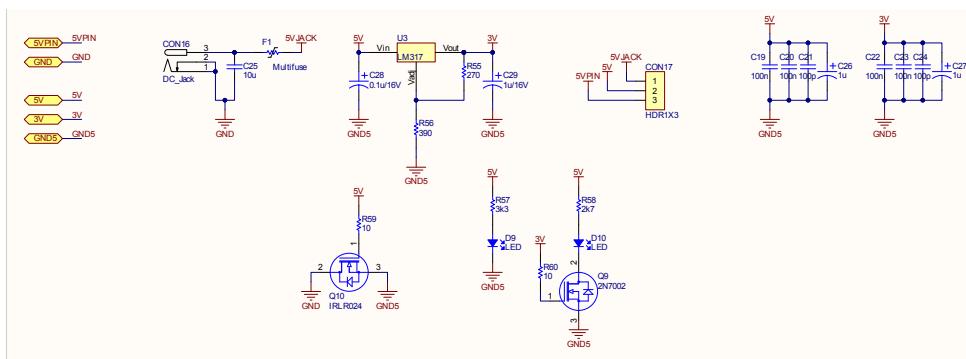
# Kapcsolási rajz

### B.1. Top level



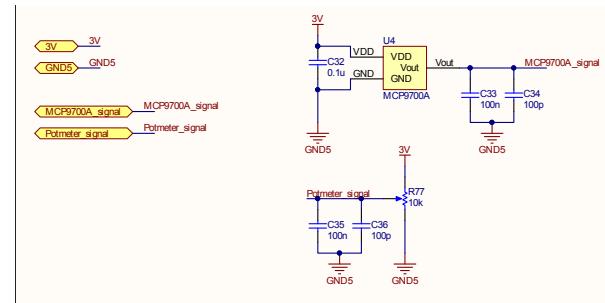
**B.1. ábra.** A mért rendszer csatlakoztatását megvalósító tűkésort és az egyes modulokat tartalmazó rajz.

### B.2. Tápfeszültség



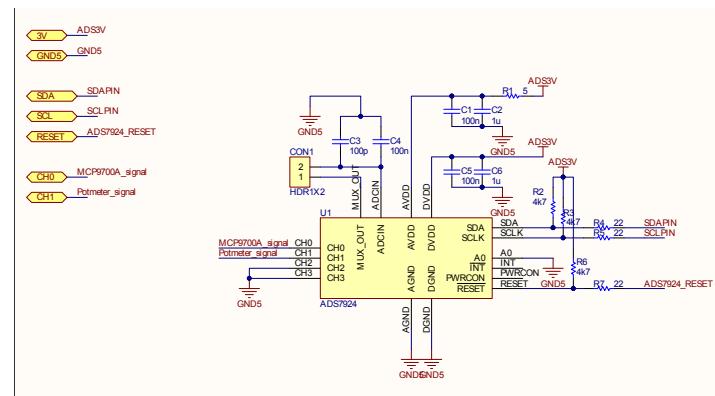
**B.2. ábra.** Tápfeszültség előállítása és jelzése LED-ek használatával.

### B.3. Szenzorok



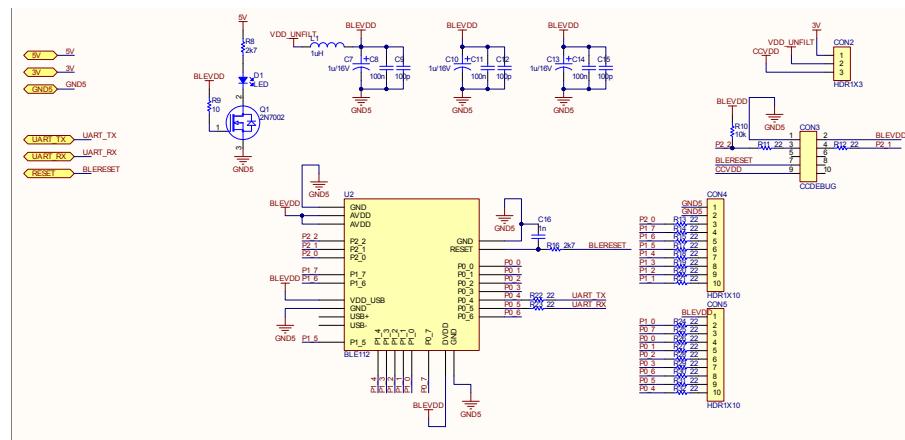
**B.3. ábra.** A helyi állomáson elhelyezett hőmérő és potmérő bekötése.

### B.4. ADS7924



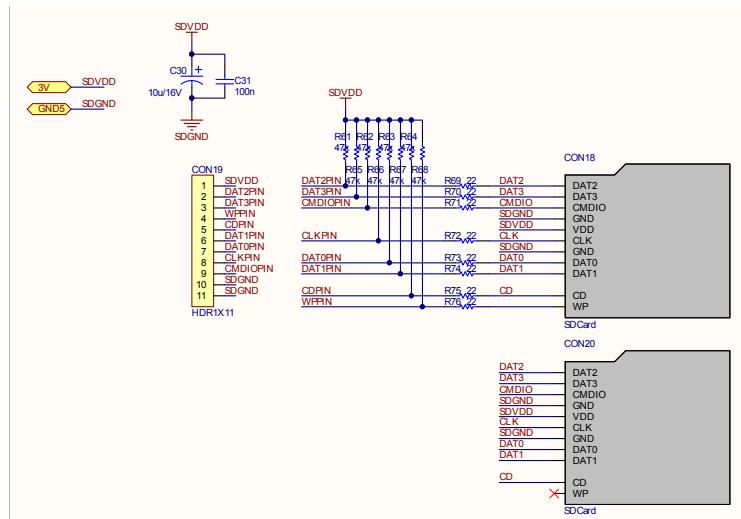
**B.4. ábra.** ADC IC bekötése.

### B.5. BLE112



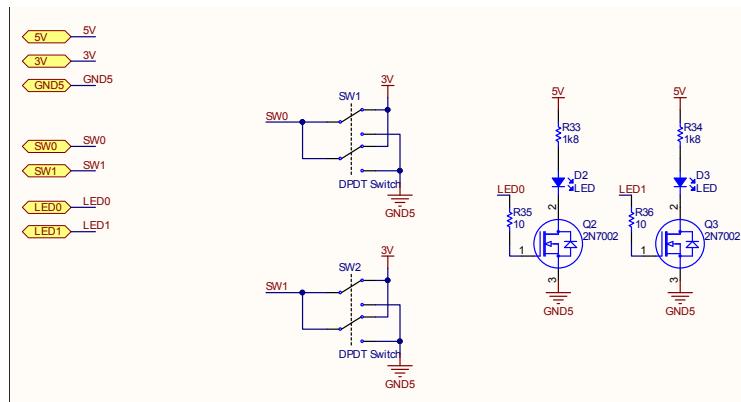
**B.5. ábra.** BLE112 Bluetooth modul és a hozzá csatlakozó programozó tüskesor bekötése.

## B.6. SD kártya



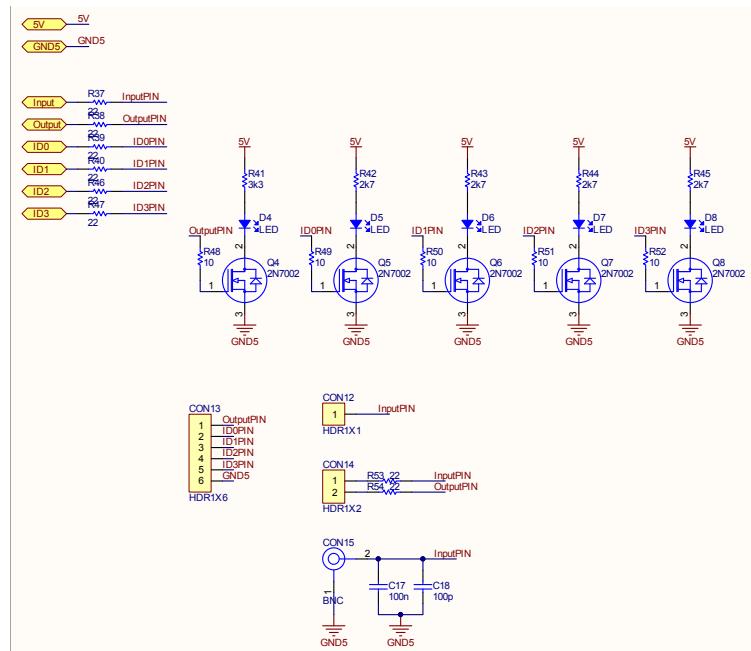
**B.6. ábra.** *SD kártya foglalat kivezetése a tüskesorra.*

## B.7. GPIO



**B.7. ábra.** Vezérelhető LED-ek és kapcsolók bekötése.

## B.8. Mérések kivezetései

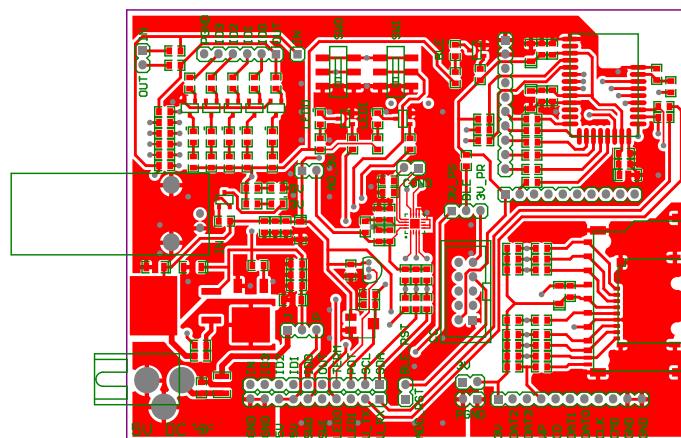


**B.8. ábra.** A mérési folyamatot egyszerűsítő kivezetések bekötése.

## C. Függelék

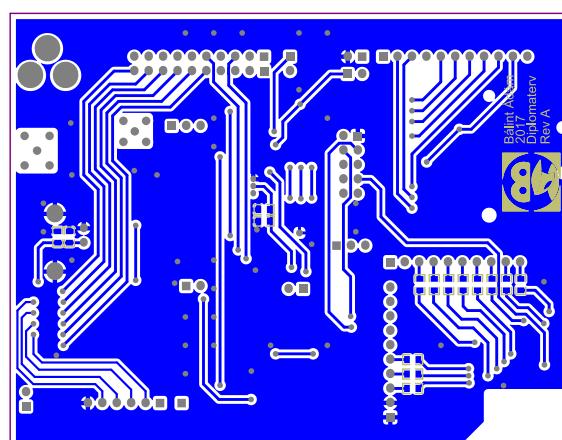
### NYÁK rajzolat

#### C.1. Alkatrész oldal



C.1. ábra. Alkatrész oldali rajzolat.

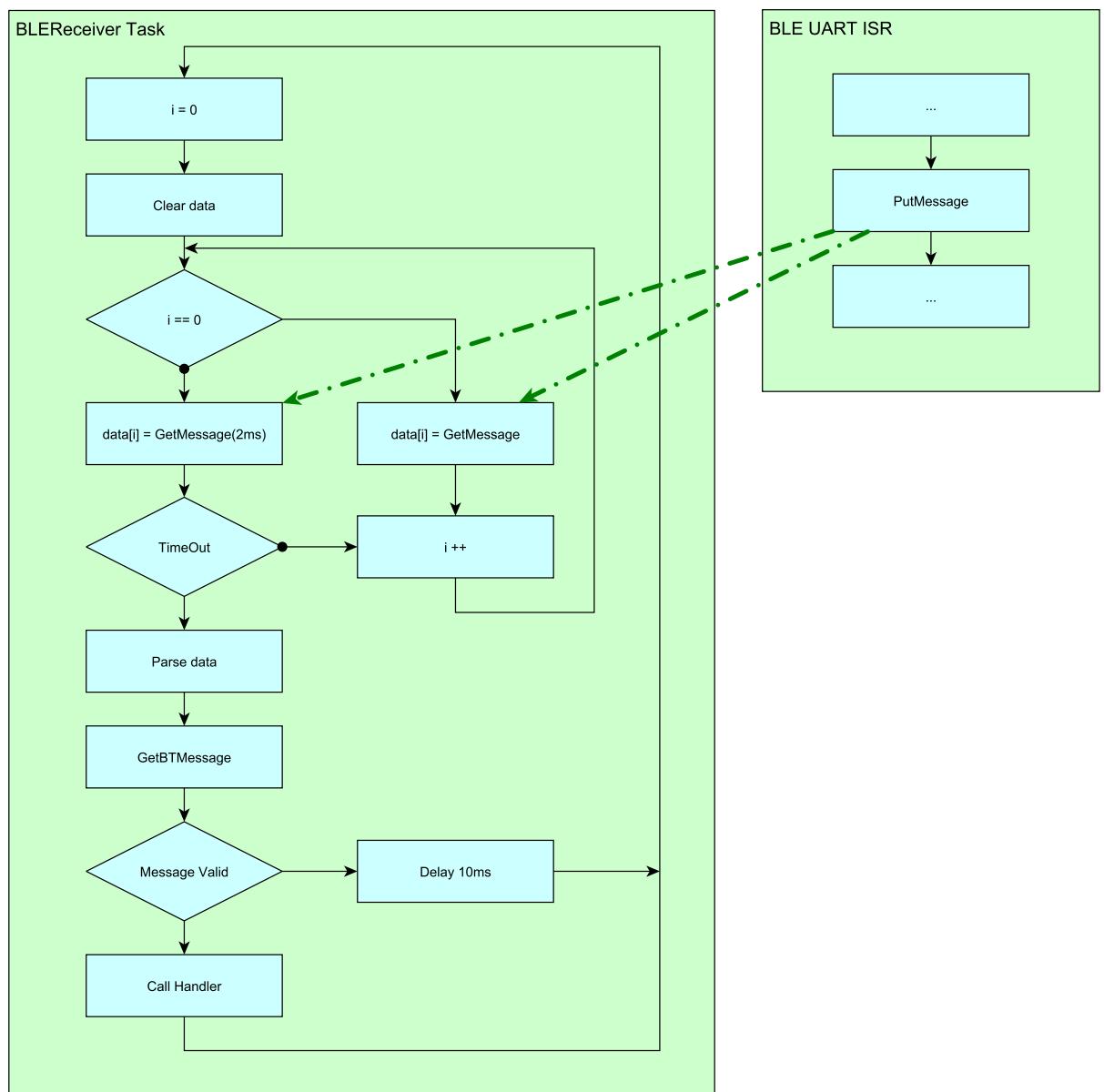
#### C.2. Forrasztási oldal



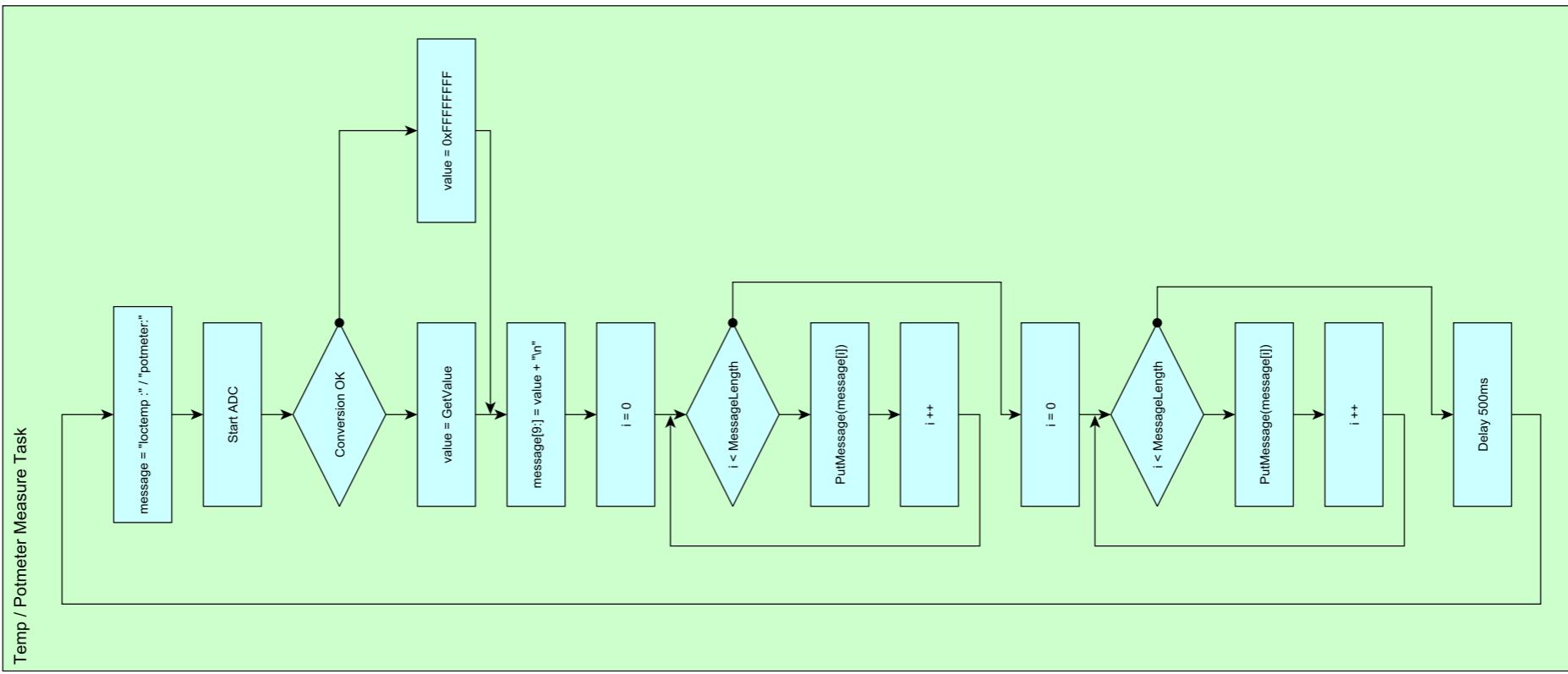
C.2. ábra. Forrasztási oldali rajzolat.

## D. Függelék

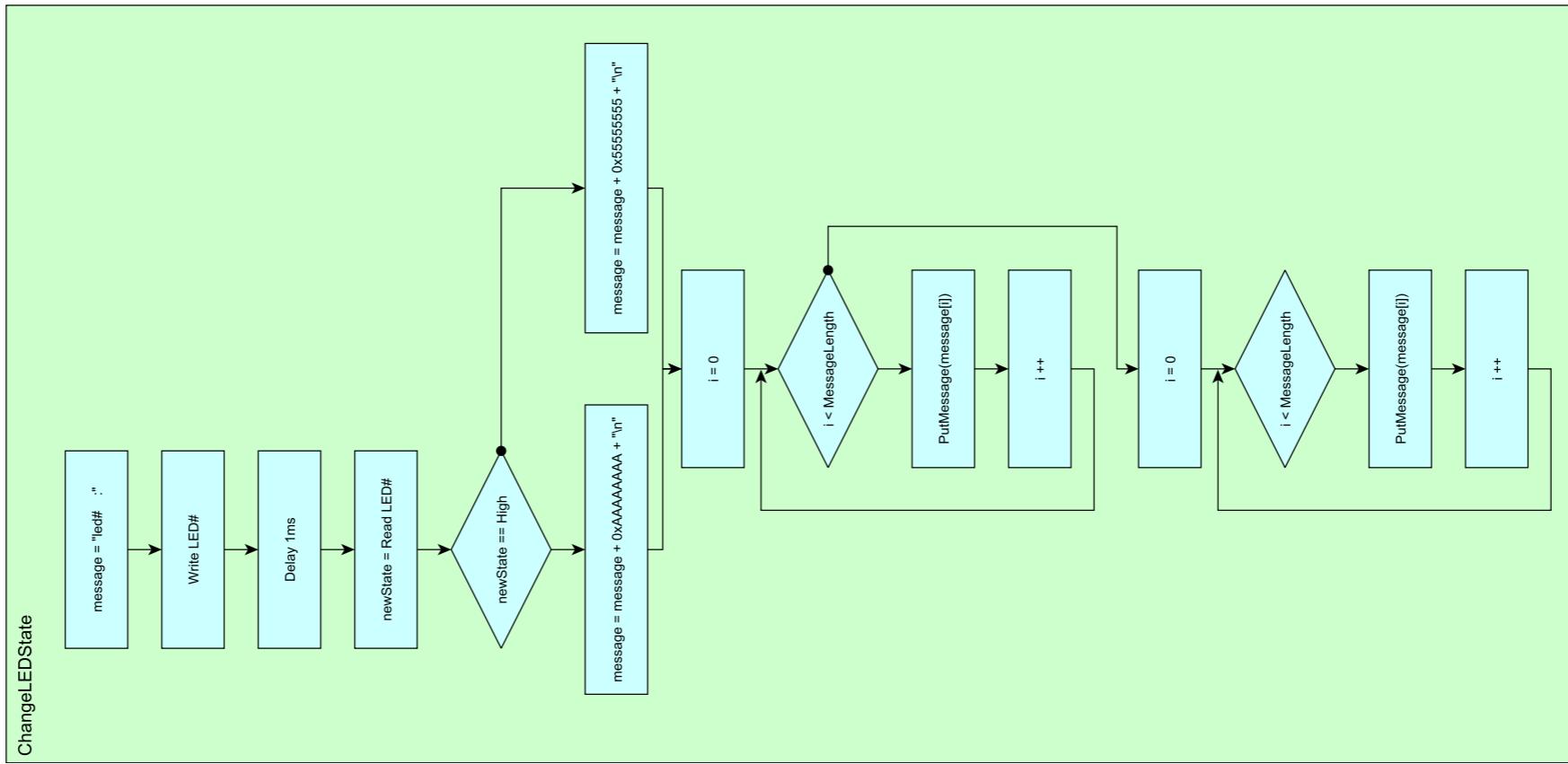
### Folyamatábrák



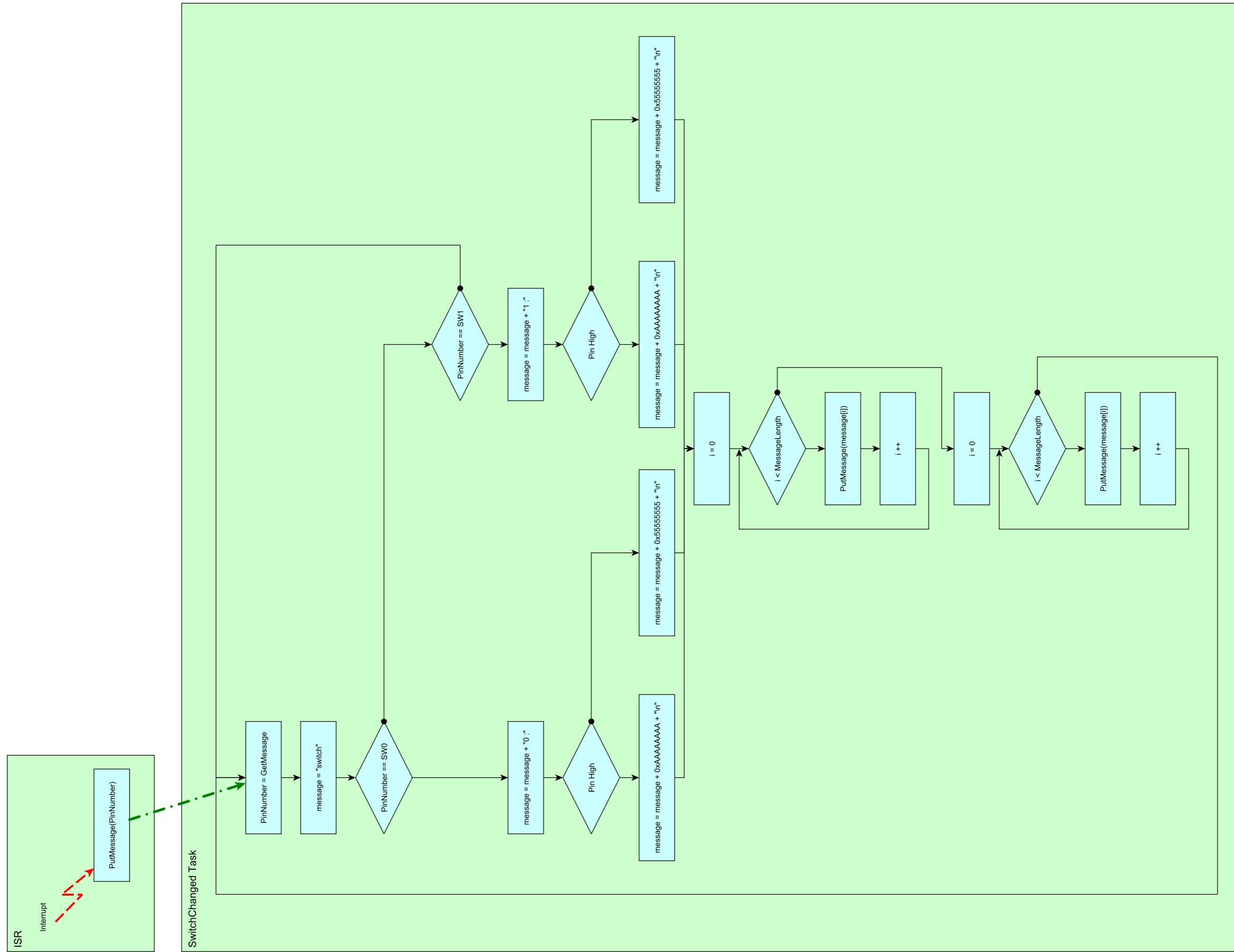
**D.1. ábra.** BLE112 modultól érkező válaszok feldolgozását taszk folyamatábrája.



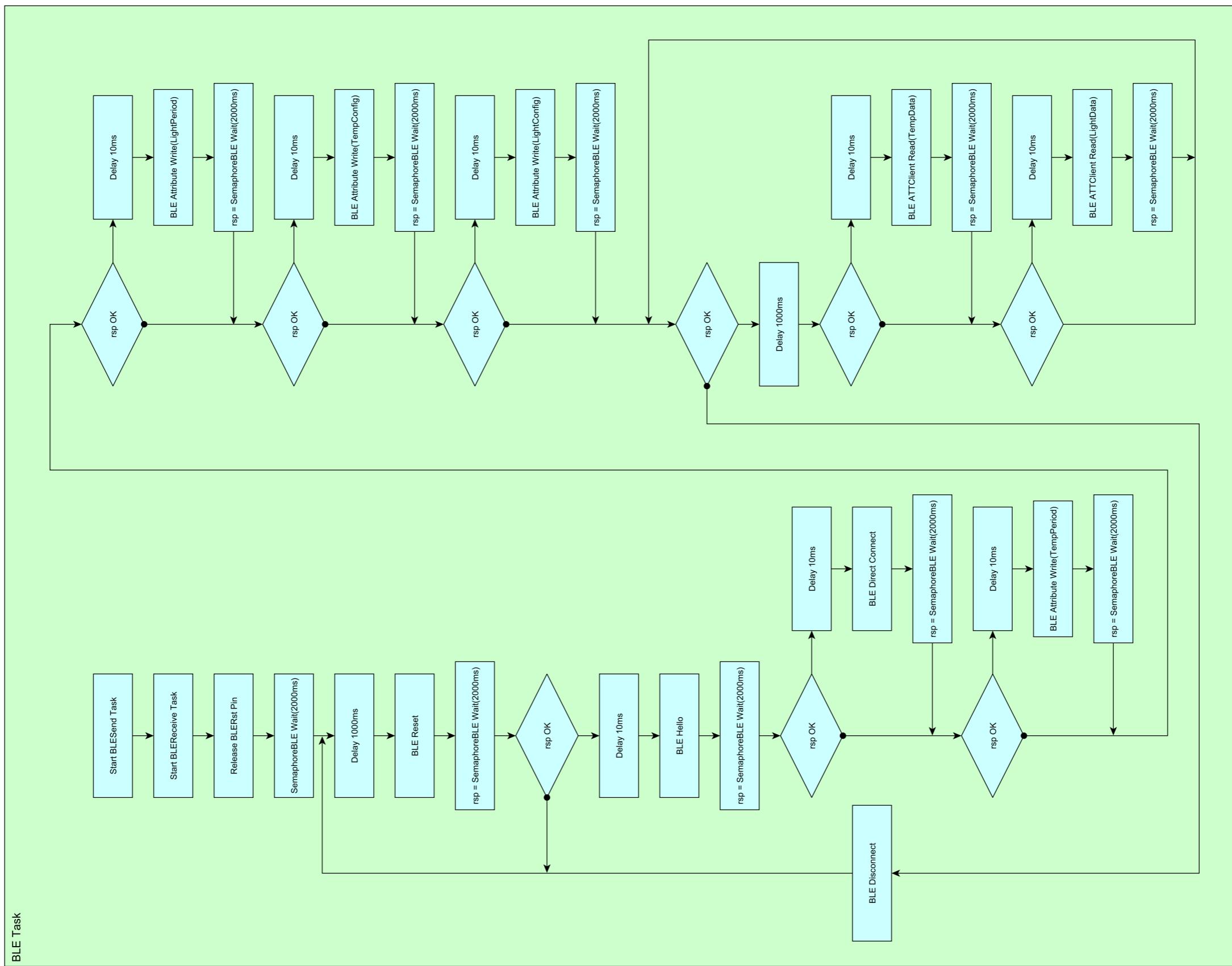
D.2. ábra. Analog-Digital konverziót végző taszkok folyamatábrája.



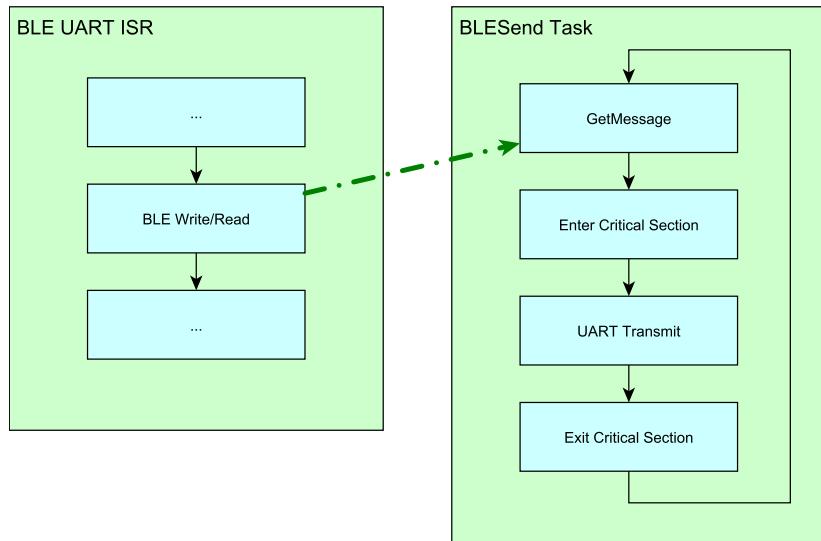
D.3. ábra. A LED-ek állapotát vezérlő taszkok folyamatábrája.



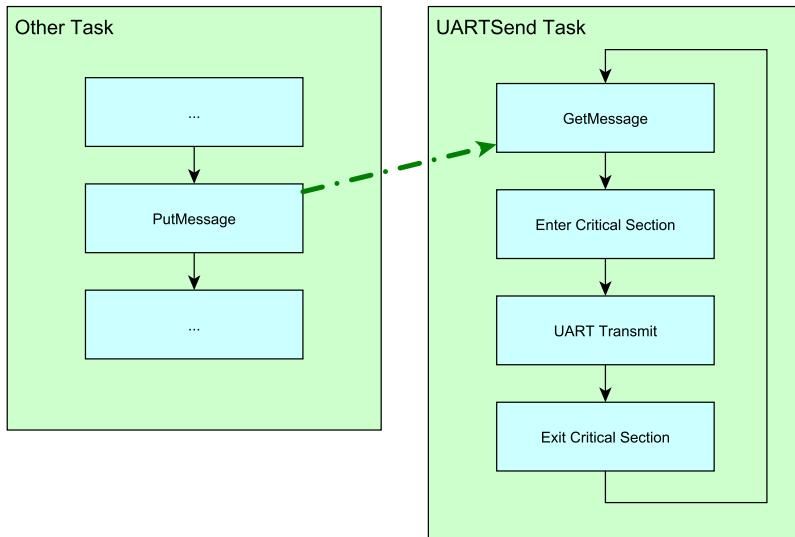
D.4. ábra. Kapcsolók változását kezelő taszkok folyamatábrája.



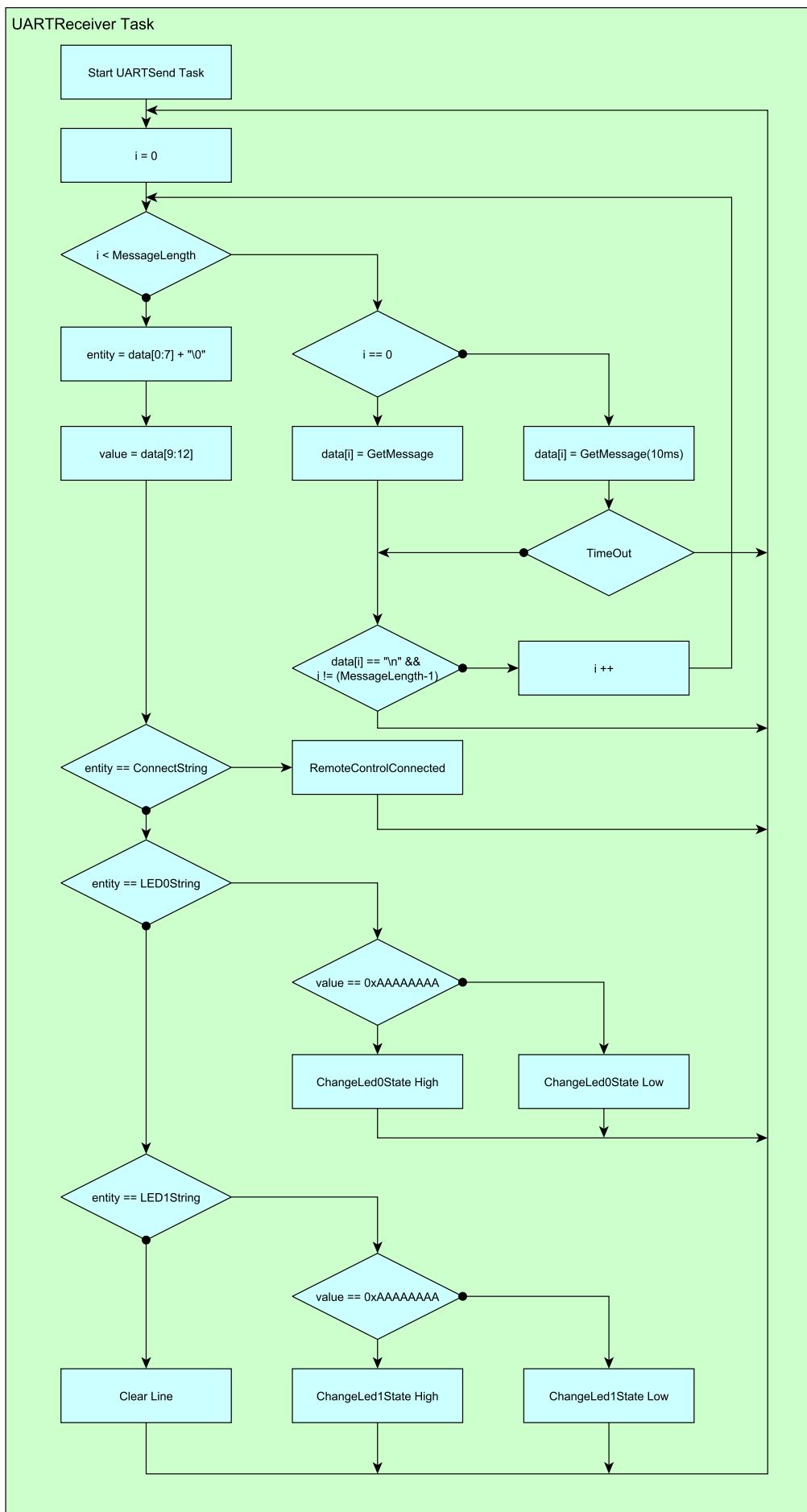
D.5. ábra. BLE112 modult vezérlő taszk folyamatábrája.



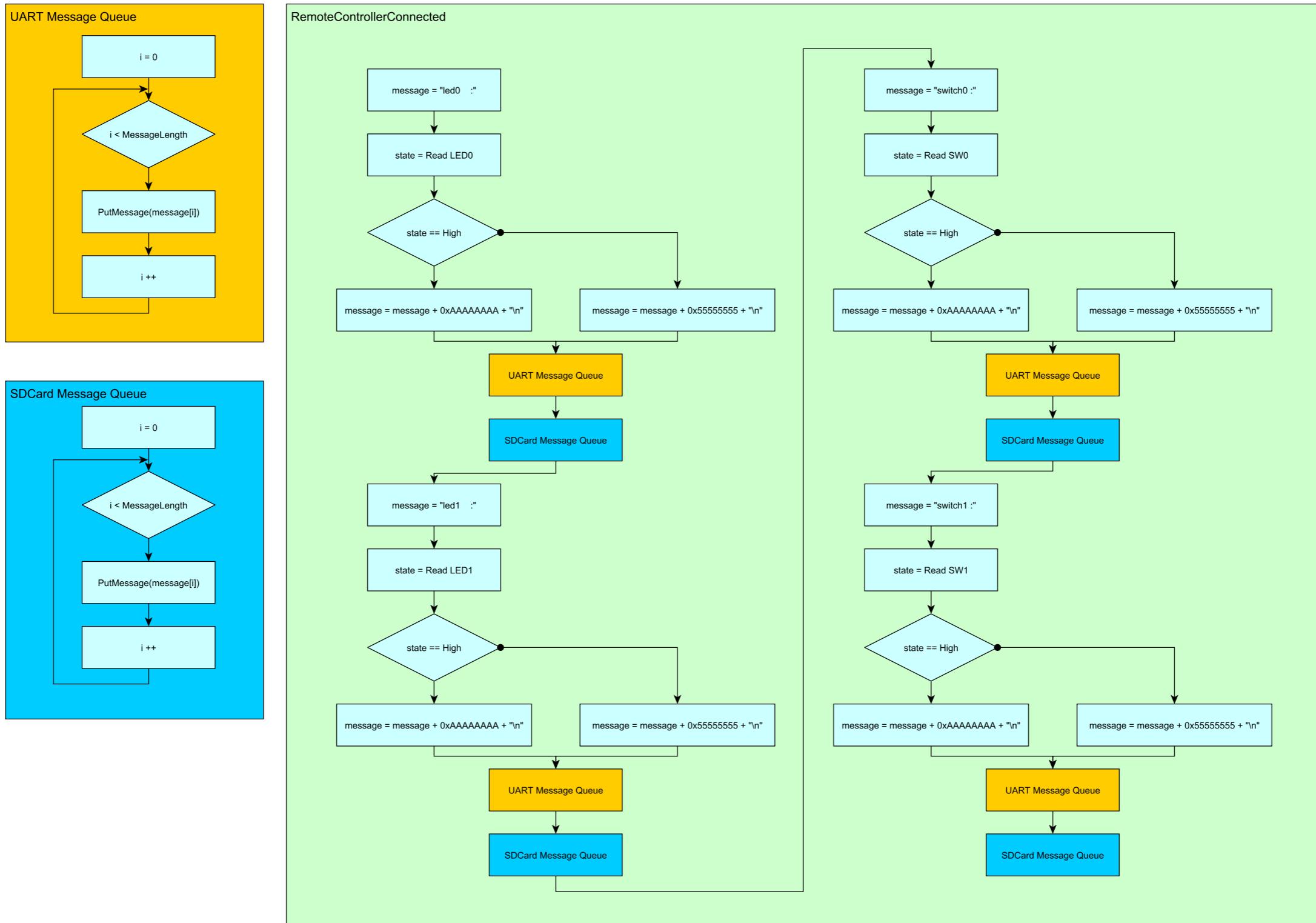
**D.6. ábra.** *BLE112 modulnak küldött üzenetek továbbítását végző taszk folyamatábrája.*



**D.7. ábra.** *A vezérlőnek küldött üzenetek továbbítását végző taszk folyamatábrája.*



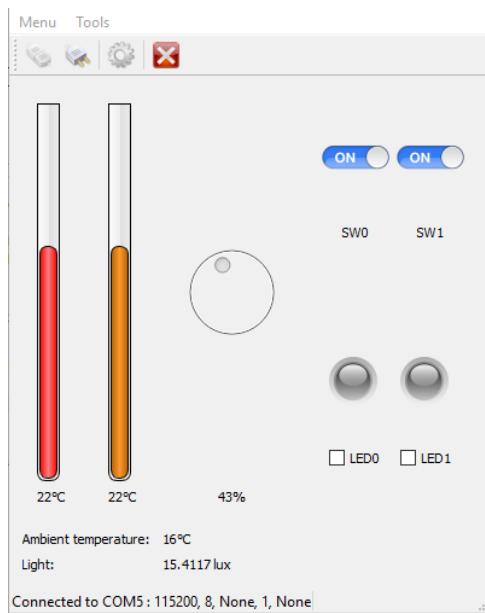
**D.8. ábra.** A vezérlő felől érkező üzenetek feldolgozását végző taszk folyamatábrája.



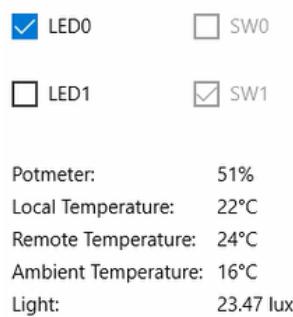
D.9. ábra. A vezérlő program csatlakozását kezelő taszk folyamatábrája.

## E. Függelék

### Felhasználói füleletek



**E.1. ábra.** Távoli vezérlést végző felhasználói felület.



**E.2. ábra.** Windows 10 IoT Core alkalmazás felhasználói felülete.



**E.3. ábra.** *Raspbian alkalmazás felhasználói felülete.*