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 multitaszk igénye szintén szükségessé teszi valamilyen operációs rendszer alkalmazását.

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

- Adjon áttekintést az elterjedt operációs rendszerek
  - felépítéséről,
  - működéséről,
  - előnyeiről,
  - 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 témakörében!
- 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 2.

Tartalomjegyzék

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 átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy 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 2.

Bálint Ádám
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

Bevezető

A bevezető tartalmazza a diplomaterv-kiírás elemzését, történelmi előzményeit, a feladat indokoltságát (a motiváció leírását), az eddigi megoldásokat, és ennek tükrében a hallgató megoldásának összefoglalását.

A bevezető szokás szerint a diplomaterv felépítésével záródik, azaz annak rövid leírásával, hogy melyik fejezet mivel foglalkozik.

1. fejezet

Elméleti áttekintés

1.1. Általánosan használt licencek

A mindennapi életben naponta szembesülünk a különböző gyártók vagy fejlesztők által elérhetővé tett programokkal, melyek mindegyike tartalmaz valamilyen megkötést a program 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 kizárólag a gyártó/fejlesztő kezében marad. A szoftver használatáért általában valamekkora 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.

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. 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 mind forrás, mind bináris formában, akár módosítással, akár anélkül, amennyiben a felhasználás feltételei teljesülnek. A szöveg azonban tartalmaz egy megkötést, amely a használat során gyakran kellemetlenségként jött elő a felhasználók körében: amennyiben a szoftver bármilyen részére vagy a szoftver használatára hivatkozás történik egy reklámanyagban, úgy a szoftver fejlesztőjét fel kellett tüntetni a szövegben. Amennyiben több, eredeti BSD licenc alá eső szoftvert tartalmazó rendszer került hivatkozásra, úgy az említett lista hamar nagy méretűvé válhatott. Ezen kellemetlenség miatt került először módosításra a BSD licenc.

A négy pont az alábbi feltételeket szabja a feljhasználó 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 nyilatkoztatot 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 mentesíti minden garanciális felelősség alól.

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. Az 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 továbbiakban.

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 hatályát, 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 tulajdont 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[1] 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.

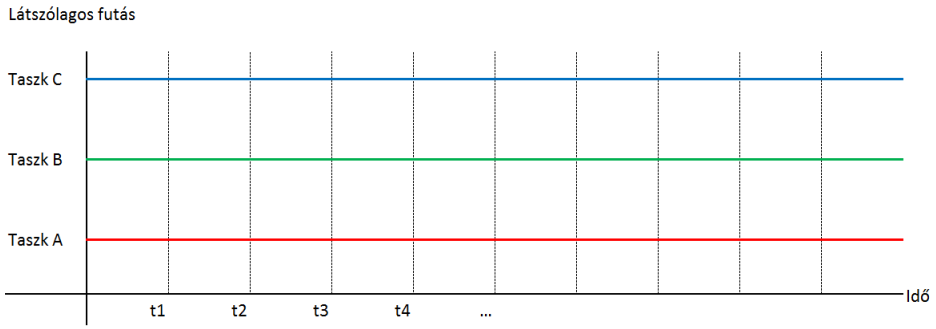
1.2.1. Operációs rendszer definíciója

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.

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

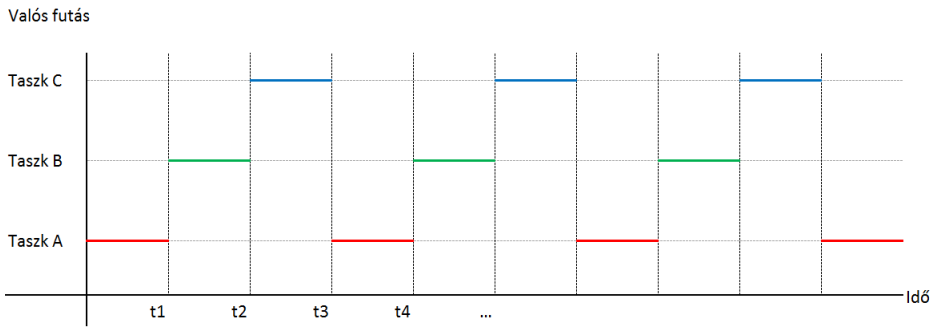
Valósidejű operációs rendszer

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



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

[1] A TiVo egy digitális videórögzítő eszközöket 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 meggátolta a módosított rendszerek alkalmazását.



1.2. ábra. *A valóságban minden folyamat egy kis időszeletet 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. 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 légzsákjának késedelmes nyitása akár halálos következményekkel is járhat.

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 alapelvét.

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.

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.

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).

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).

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 taszkhoz 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 ágyazni, 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. Legszigorúbb 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 védett 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 hatékonyságát.

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 eltéró 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[2]), amikor a szemafor értéke növelésre kerül.
- A szemafor elvétele (elterjedt elnevezések: take, wait, pend, P() művelet[3]), 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 előjel nélküli egész) használatával, viszont a beépített szemafor struktúra atomi műveletként kerül kezelésre, illetve a rendszer automatikusan tudja kezelni a várakozó folyamatok állapotok közti mozgatását.

Szemaforokat leggyakrabban szinkronizációs célból, vagy erőforrások védelmére használnak.

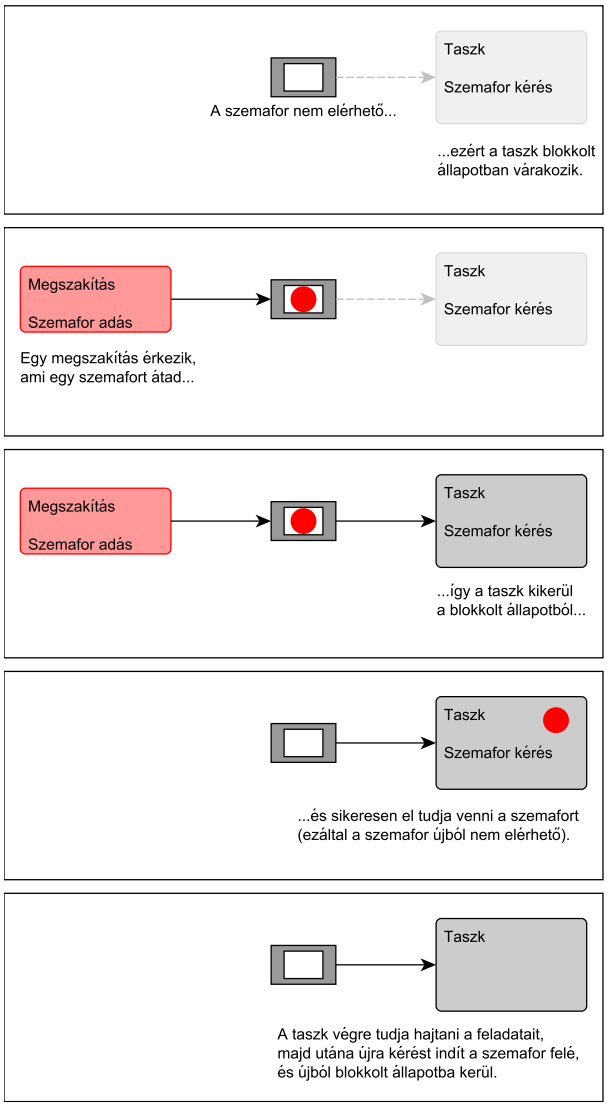
Bináris szemafor

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.

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 lehetőségük várakozó állapotba kerülni (1.5. ábra).

[2] A jelölés a holland *verhogen* (növelés) szóból származik.
[3] A jelölés a holland *proberen* (tesztelés) szóból származik.



1.4. ábra. *Szinkronizáció bináris szemafor segítségével.*

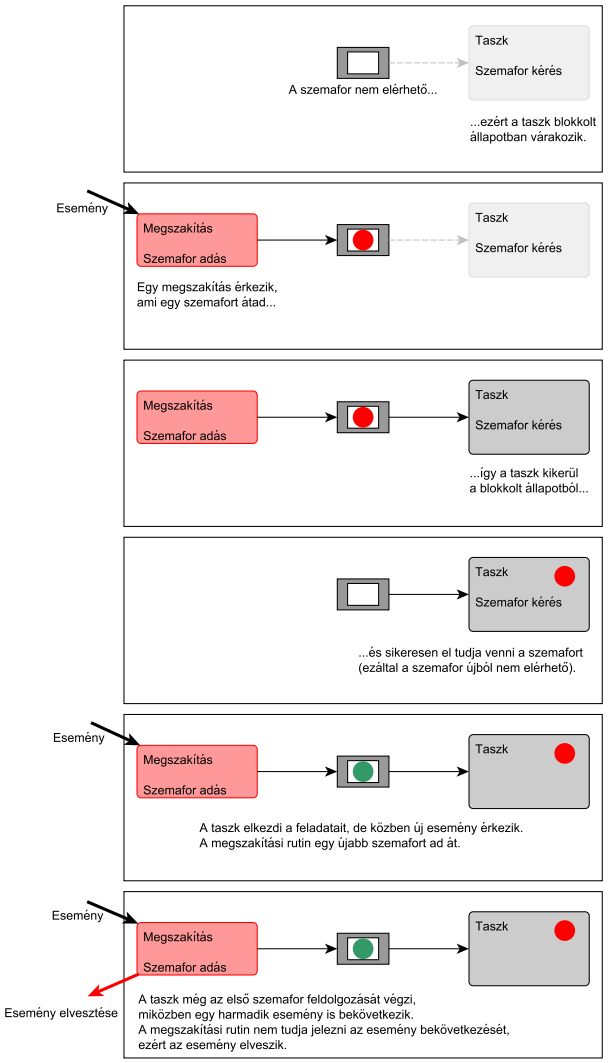
Számláló szemafor

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 (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.

- **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



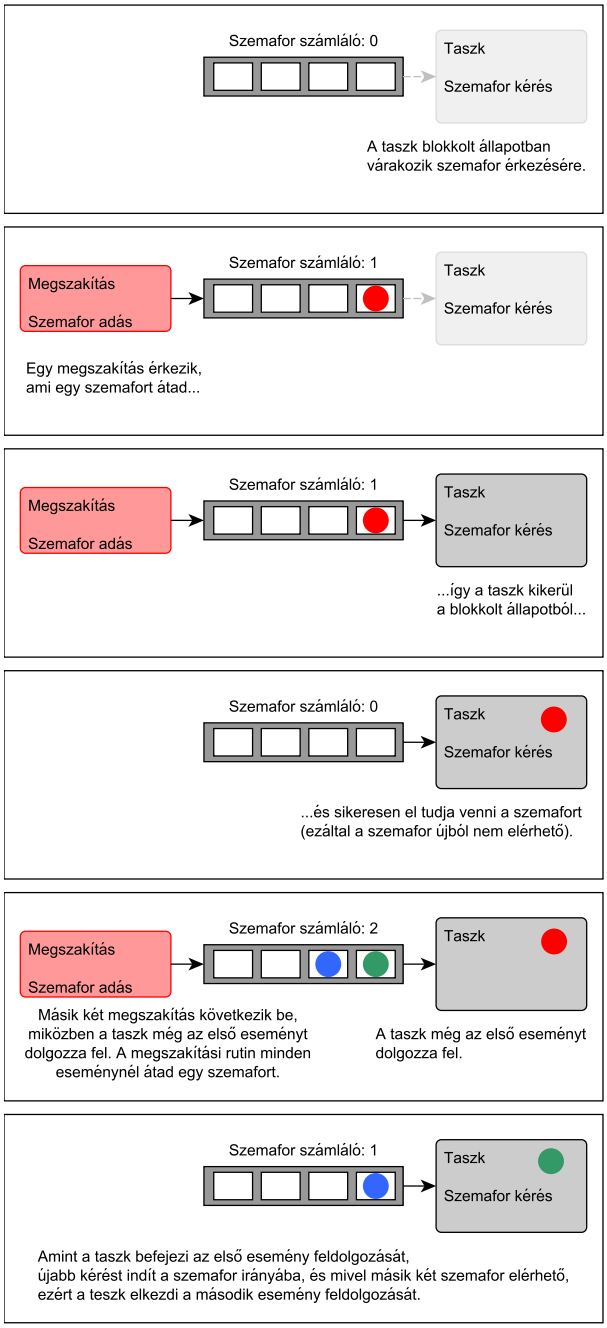
1.5. ábra. *Esemény bekövetkezésének elvesztése bináris szemafor használata során.*

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.

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 mindaddig 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 holtpont 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.*

nem kell felszabadítani: a jelző taszk vagy megszakítás jelzést ad a szemforon keresztül a feldolgozó taszknak. A feldolgozó taszk elveszi a szemafort, de a feldolgozás befejeztével a szemafort nem adja vissza.

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 táro-

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)

Vegyünk 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).

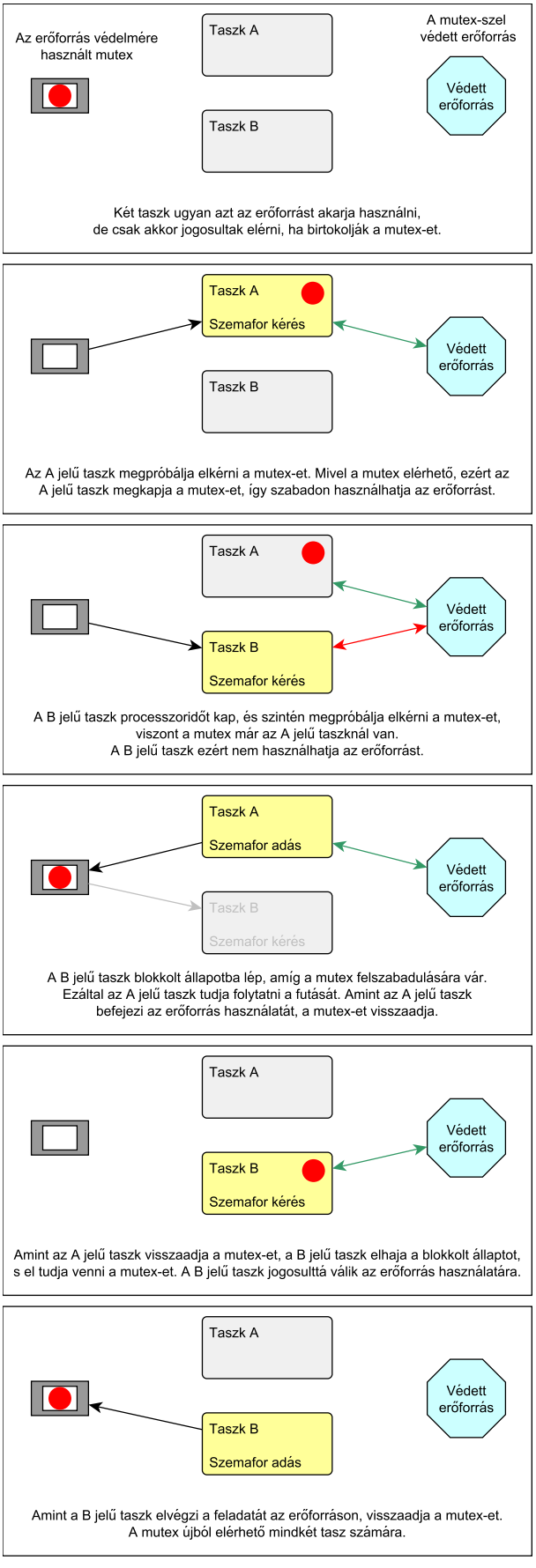
Induláskor a *TaskA* kapja meg a futás jogát, és lefoglalja a *ResA*-t. Közben lejár a *TaskA*-nak kiosztott időszelet, és *TaskB* kerül futó állapotba. A *TaskB* lefoglalja a *ResB* erőforrást, majd megpróbálja lefoglalni a *ResA* erőforrást is. Mivel a *ResA*-t már a *TaskA* használja, ezért a *TaskB* 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.

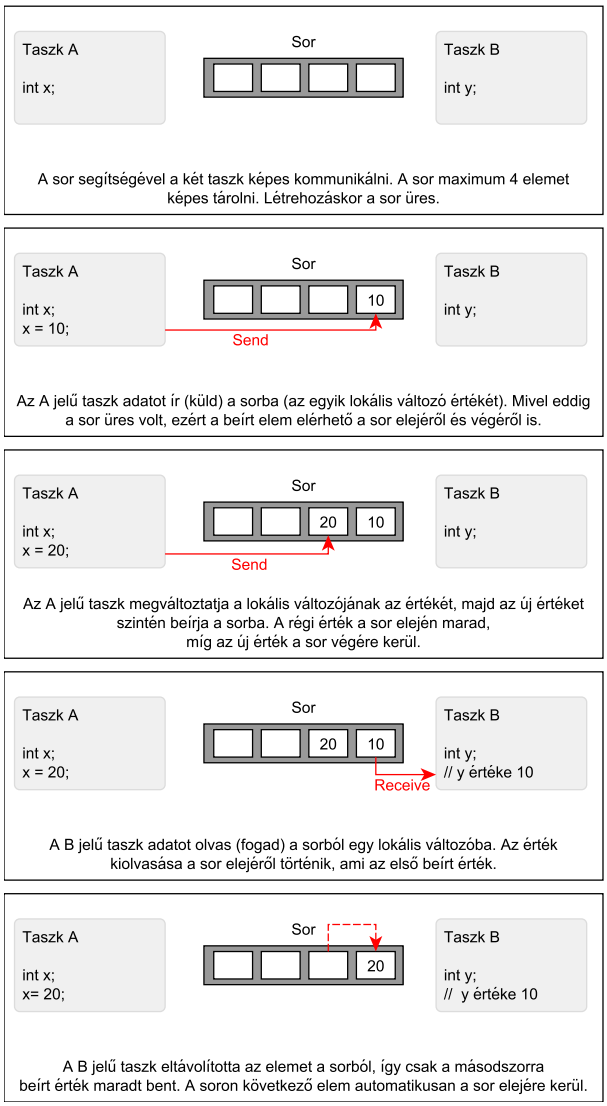
Újrahívható függvények (Reentrant functions)

Egy függvény reentráns (újrahívható), ha biztonságosan meghívható több különböző taszkból vagy megszakításból.

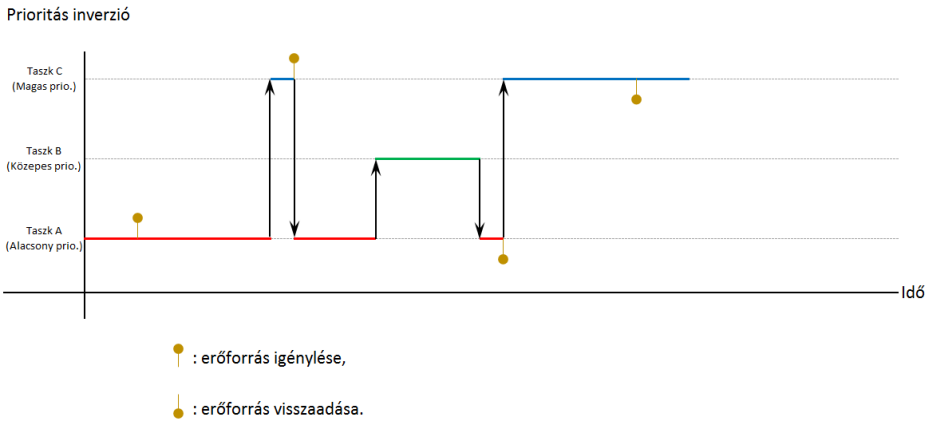
Minden taszk rendelkezik saját stack-kel és saját regiszterekkel. Ha egy függvény minden adatot a stack-jén vagy a regisztereiben tárol (vagyis nem használ globális és statikus változókat), akkor a függvény reentráns.



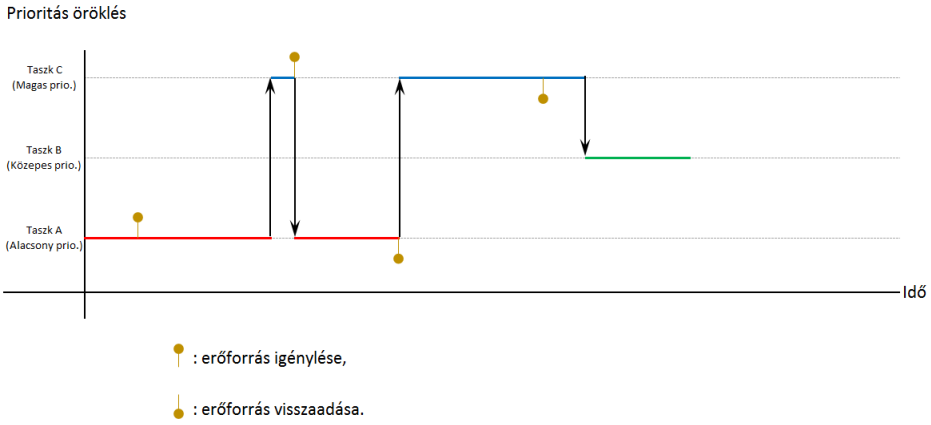
1.7. ábra. *Mutex működésének szemléltetése.*



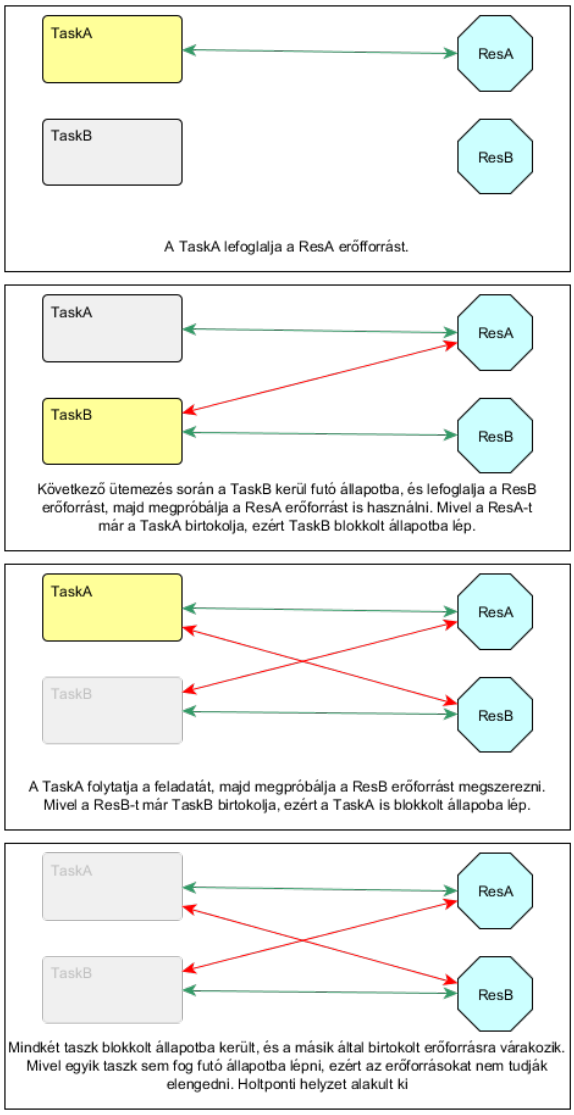
1.8. ábra. *Sor működésének szemléltetése.*



1.9. ábra. *Prioritás inverzió jelensége.*



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



1.11. ábra. *Holtponti 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 kártya. A rajta található mikrokontroller rendelkezik a legtöbb alkalmazásban előforduló perifériák mindegyikével. A teljesség igénye nélkül:

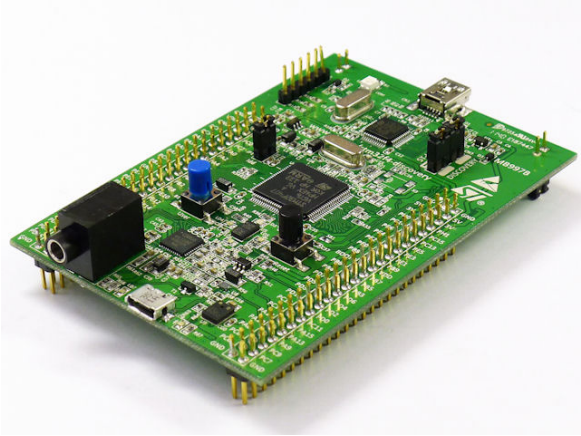
- GPIO-k,
- Soros kommunikációs portok (szinkron és aszinkron egyaránt),
- Ethernet port,
- Analog-Digital Converter,
- Digital-Analog Converter,
- Időzítők,
- High-Speed USB (OTG támogatással),
- SPI,
- I2C,
- I2S,

- 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 kezelni képes DMA-knak.

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

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



2.1. ábra. *STM32F4 Discovery fejlesztőkártya.*

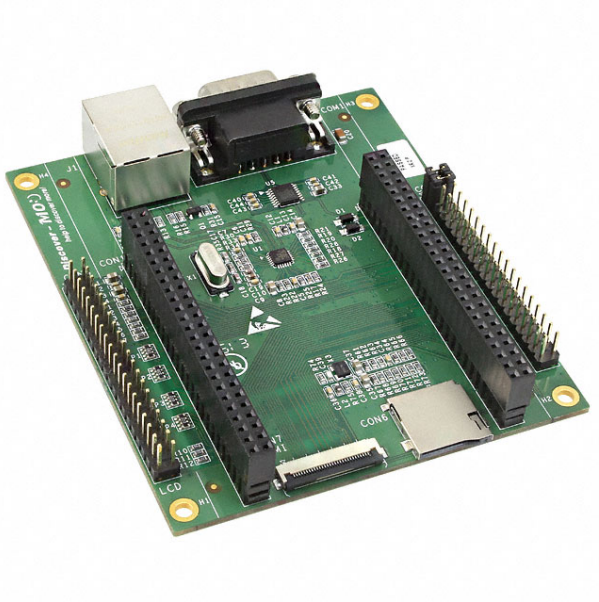
STM32F4 Discovery - Base Board

Az STM32F4 Discovery fejlesztőeszközhöz több kiegészítő kártya is kapható, melyek célja a kipróbálható 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.

Az első nagyteljesítményű, bankkártya méretű számítógépüket 2012 februárjában bocsátották piacra, melynek ára töredéke volt az asztali számítógépekének. Azóta több verziója is megjelent az eszköznek, melyet folyamatosan fejlesztettek mind teljesítményben, mind az



2.2. ábra. *STM32F4 Discovery BaseBoard kiegészítő kártya.*

integrált funkciók számában. 2015 novemberében a világ első 5 $-os számítógépével jelentek meg a piacon, melynek a Raspberry Pi Zero nevet adták.

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

2.3. ábra. *Raspberry Pi 3 bankkártya méretű PC.*

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.

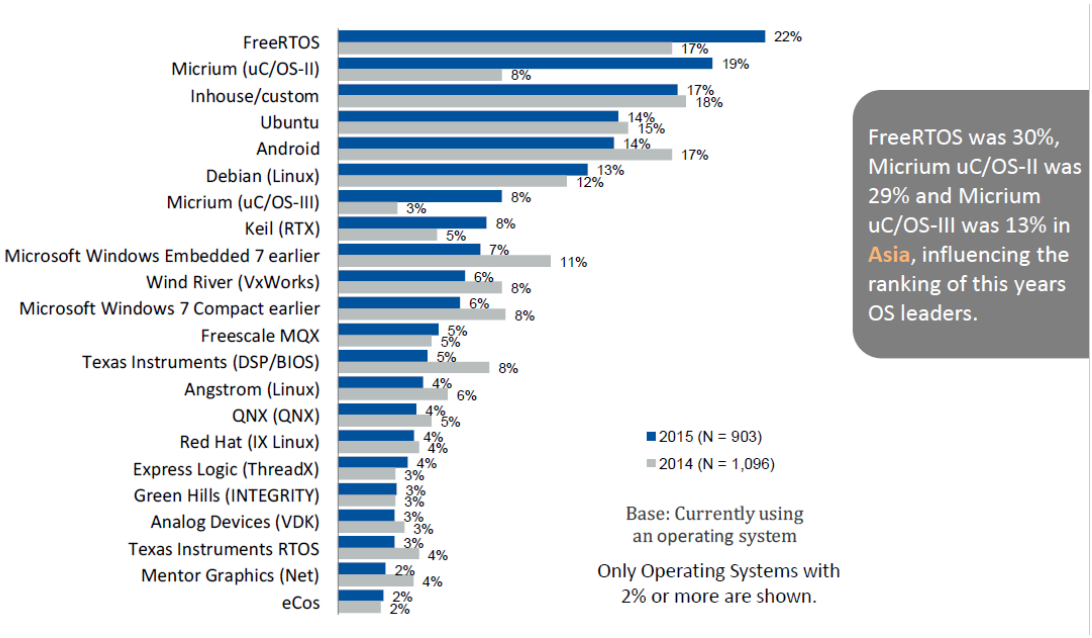
Már az első verzió megjelenésekor rendelkezésre álltak különböző linux disztribúciók portjai, melyekkel az eszköz asztali számítógépként használható volt. Népszerűségének köszönhetően a második verziótól kezdve már a Windows 10 IoT Core is támogatja a platformot.

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 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.



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

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

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 legkisebb beágyazott rendszereken is futtatható legyen. Ebből adódóan csak az alap funkciók vannak megvalósítva, mint ütemezés, taszkok közötti kommunikáció lehetősége, memóriamenedzsment, de nincs beépített támogatás hálózati kommunikációra vagy bármiféle külső hardver használatára (ezeket vagy nekünk kell megírnunk, vagy harmadik féltől 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 kizárólag a FreeRTOS API-n keresztül kommunikálnak.

2.3.2. Taszkok

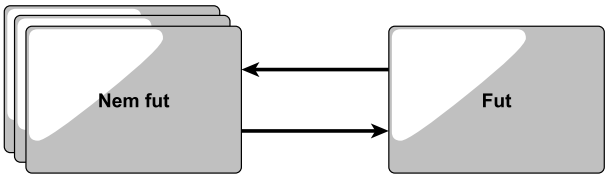
A FreeRTOS nem korlátozza a létrehozható taszkok és prioritások számát, amíg a rendelkezésre álló memória lehetővé teszi azok futtatását. A rendszer lehetőséget biztosít ciklikus és nem ciklikus taszkok futtatására egyaránt.

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. Kizárólag debug célokra
pxEndOfStack	A stack végére mutató pointer a stack túlcsordulásának detektálására
uxBasePriority	Az utojára taszkhoz rendelt prioritás. Mutex használata esetén a prioritás öröklés során megnövelt prioritás visszaállítására
ulRunTimeCounter	A taszk *Fut* állapotban töltött idejét tárolja futási statisztika készítéséhez

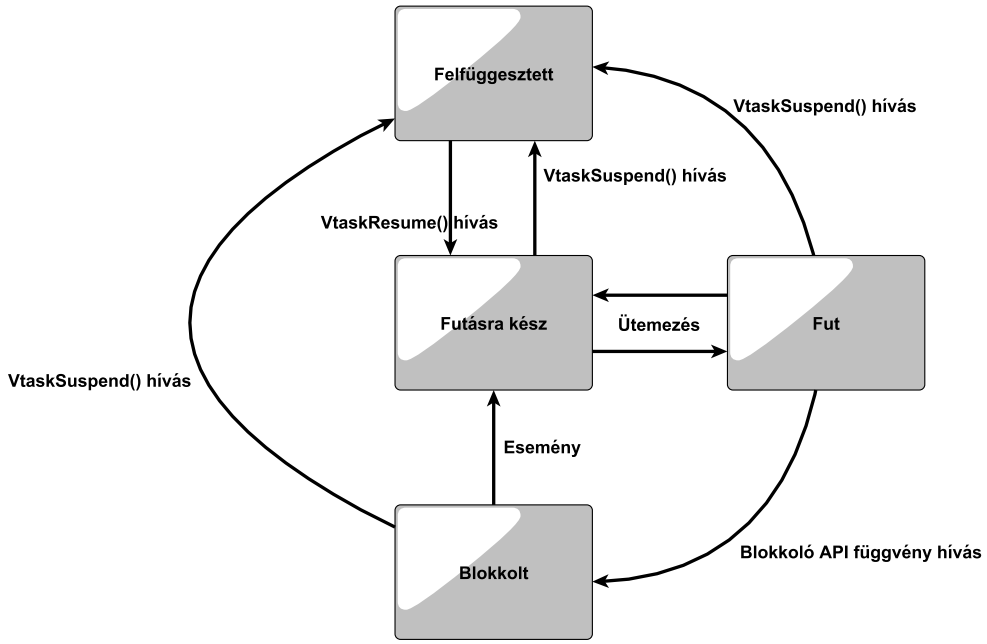
A beágyazott rendszerek döntő része egymagos processzorokat használ, amiből az következik, hogy egyszerre csak egy taszk futhat. A taszkok eszerint két nagy csoportba oszthatóak: éppen futó taszk (*Fut* állapot), illetve az összes többi (*Nem fut* állapot). Ez az egyszerűsített felosztás látható a 2.5.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



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

Annak, hogy egy taszk éppen miért nem fut, több oka lehet. Ez alapján a *Nem fut* állapot több állapotra feloszthatóó (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 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[1]. *Fix prioritásos*, mivel a rendszer magától nem változtatja a prioritásokat, *preemptív*, mert egy taszk *Futásra kész* állapotba lépésekor preemtálja az éppen futó taszkot, ha a futó taszk prioritása alacsonyabb.

[1] A FreeRTOS kooperatív ütemezést is támogat, viszont a valósidejű futás eléréséhez a preemptív ütemezés szükséges, ezért a továbbiakban csak a preemptív ütemezéssel foglalkozunk.

Idle taszk

A processzor folyamatosan utasításokat hajt végre működése közben (eltekintve a különböző energiatakarékos üzemmódoktól), ezért legalább egy taszknak mindig *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* 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 *kiéheztetésével* 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 függvénnyel kapcsolatban

1. Az Idle hook függvény nem hívhat blokkoló vagy felfüggesztést indító függvényeket[2],
2. Ha az alkalmazás valahol töröl egy taszkot, akkor az Idle hook függvénynek elfogadható időn belül vissza kell térnie[3].

2.3.3. Kommunikációs objektumok

A FreeRTOS három alap kommunikációs struktúrát kínál a felhasználónak:

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

[2] Az Idle hook függvény blokkolása esetén felléphet az az eset, hogy nincs Futásra kész állapotban lévő taszk.

[3] Az Idle taszk felelős a kernel erőforrások felszabadításáért, ha egy taszk törlésre kerül. Ha az Idle taszk bentragad az Idle hook-ban, akkor ez a tisztítás nem tud bekövetkezni.

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 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 csökkentve a RAM kihasználtságát (ekkor viszont különösen figyelni kell arra, hogy a kijelölt memóriaterület tulajdonosa egyértelmű legyen, vagyis ne történjen különböző taszkokból módosítás egyidőben, 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 soronkövetkező elemet, de azt nem távolítja el a sorból(*xQueuePeek()*).

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

A sorok egy taszkhoz sem tartoznak, így egy sorba akár több taszk is írhat, illetve 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 értékével 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.

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.

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 holtponti 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 különbségek miatt a mutex védett a prioritás inverzió problémájával szemben[4], míg a bináris szemafor implementációjából hiányzik.

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 megszakítást 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.

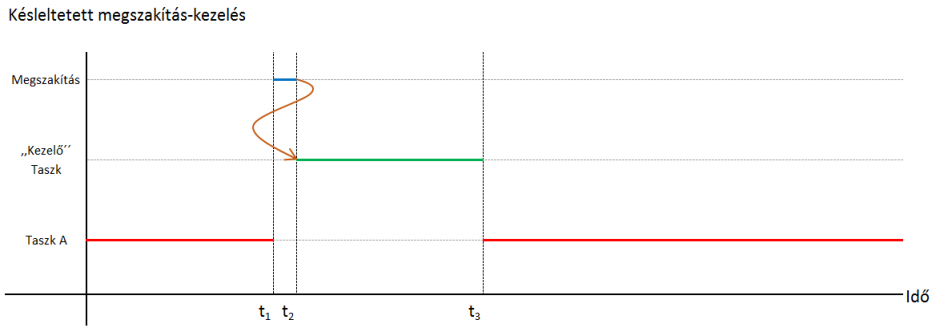
[4] A FreeRTOS prioritás öröklési mechanizmusa csak egyszerű implementációt tartalmaz, és feltételezi, hogy egy taszk csak egy mutex-et birtokol egy adott pillanatban.

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 szükséges lépéseket végezzük el (például eseményjelző bitek törlése), majd egy szemaforon keresztül jelezzük a feldolgozó taszknak az esemény bekövetkeztét. Ha a feldolgozás időkritikus, akkor a feldolgozó szálhoz rendelt magas prioritással biztosítható, hogy az éppen futó taszkot preemtálja. 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* állapoban a szemafor érkezésére. A jelzés hatására *Futásra kész* állapotba kerül, ahonnan az ütemező (az éppen futó taszkot preemtálva) *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. *Megszakítások késleletett feldolgozásának szemléltetése.*

Megszakítások 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 fájlában (*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,[5],
- 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, amig 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 nyílvántartja, 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 preemtálástól, viszont a megszakítások nem kerülnek letiltásra. Hátránya, hogy az ütemező elindítása hosszabb időt vehet igénybe.

Gatekeeper taszk

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

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

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

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

Beágyazott alkalmazások fejlesztése során is szükség van dinamikus memóriafoglalásra. 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-allokációs igényükben és az erőírt időzítési korlátokban, ezért nincs olyan memória-allokációs séma, amely minden alkalmazásban megállná a helyét. A FreeRTOS több allokációs algoritmust is a fejlesztő rendelekezésére bocsát, amiből az alkalmazásnak megfelelő kiválasztásával lehet elérni a megfelelő működést.

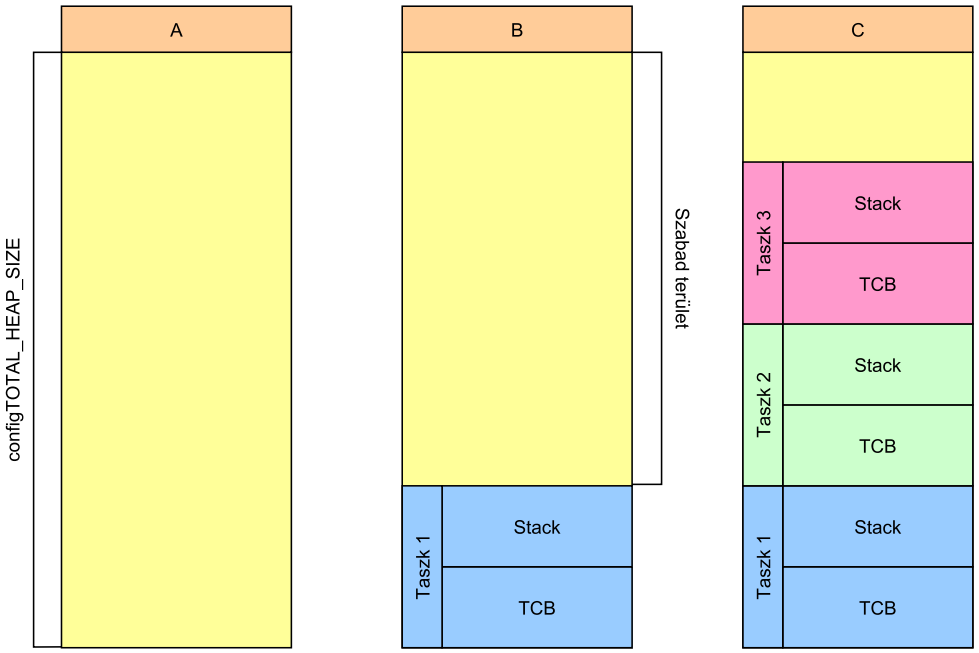
Heap_1.c

Kisebb beágyazott 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 kódméretet 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 memórifoglalás 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 töredezettség léphet fel. 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

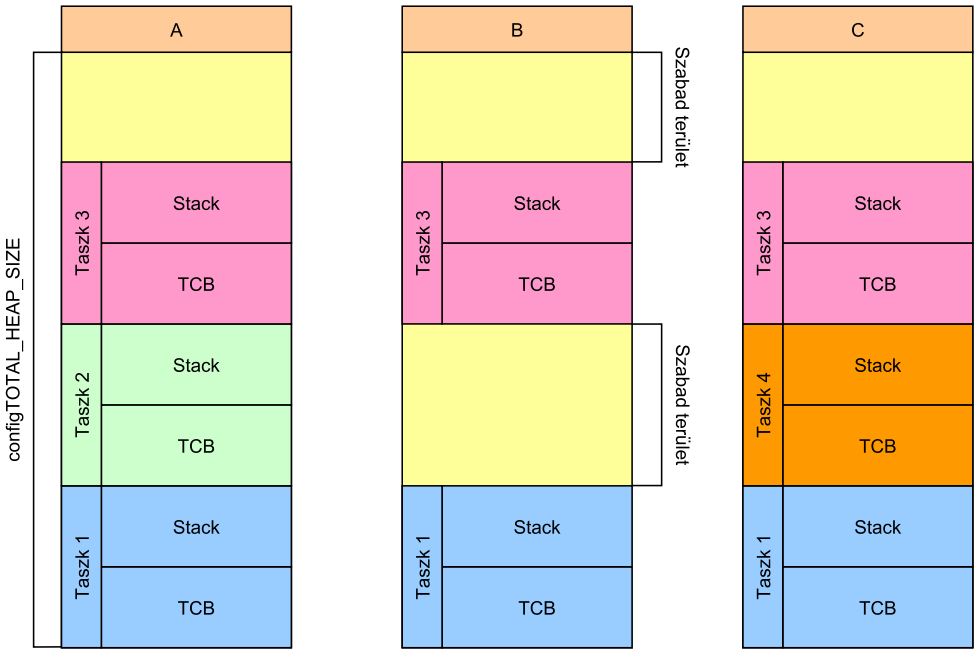
A *heap_3.c* a *standard library* függvényeit használja, de a függvények alatt felfüggeszti az ütemező működését, ezzel elérve, hogy a memória-kezelés thread-safe legyen.

A heap méretét nem befolyásolja a *configTOTAL_HEAP_SIZE* érték, ehelyett a linker beállításai határozza meg.

2.4. µC/OS–III

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



2.9. ábra. *A heap_2.c implementációjának működése.*

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 rendszert. 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 lépéseinek magyarázatával – a forráskódban található megjegyzések is segítik a fejlesztőt a programozás során.

Az operációs 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.

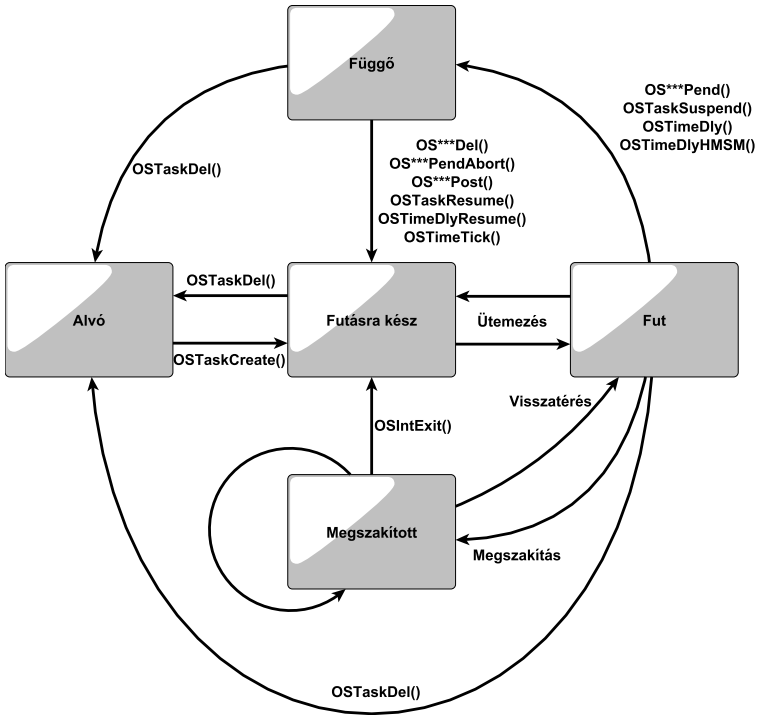
2.4.2. Taszkok

A µC/OS előző verziójában a prioritások száma korlátozott volt és minden prioritási szinthez maximum egy tasz 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 µC/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 µC/OS–III rendszerben.*

A taszkok nyílvántartása a µC/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 µC/OS–III által alkalmazott TCB főbb változói a 2.2. táblázatban láthatóak.

A FreeRTOS-hoz hasonlóan a µC/OS–III is támogatja a hook függvények használatát, melyekkel a rendszer funkcionalitását bővíthetjük.

Az Idle taszk is megtalálható, mely a legalacsonyabb prioritási szinten jön létre a rendszer inizializálásakor[6]. 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 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

[6] A µC/OS–III csak az Idle taszkot engedi a legalacsonyabb prioritási szinten futni, a fejlesztő által létrehozott taszkoknak mindeképp magasabb prioritással kell rendelkeznie!

2.2. táblázat. *A µC/OS TCB-jének főbb változói.*

Változó	Jelentés
StkPtr	A stack utolsó elemére mutató pointer
StkLimitPtr	A stack-en belül egy adott részre mutat. A stack túlcsordulásának ellenőrzésére használja a rendszer.
NextPtr és PrevPtr	Ezen pointerek használatosak a futásra kész taszkok TCB-jének kétszeresen láncolt listájának felépítésére.
NamePtr	A taszkhoz rendelt név.
StkBasePtr	A taszk stack-jéne alapcímére mutató pointer.
TaskEntryAddr	A taszk belépési pontját tartalmazza.
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 taszkhoz 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 időszeletek száma.
MsgPtr és MsgSize	Közvetlenül a taszknak küldött üzenetre mutató pointer és az üzenet mérete.

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 µC/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 befejeződöt, a memóriaterület felszabadításra kerül.

Olvasás sorból

A µC/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 taszkhoz 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 µC/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 támogadtja 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 konjuktív 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 újbóli engedélyezés után jut érvényre. Ez a megvalósítás volt jelen az operációs rendszer előző verziójában is.

Késleltett módszer (*Deferred Method*) esetén a kritikus szakaszokban a megszakítások nem kerülnek letiltásra, csak az ütemezőt függeszti 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 szemfor és mutex mellett az erőforrások védelmére használhatunk kritikus szakaszokat az megosztott erőforrások védelmére. Ahogy azt a *Megszakítás-kezelés* részben ismertettem, a µC/OS–III a kritikus szakaszokban beállítástól függően vagy letiltja a megszakításokat, vagy csak az ütemezőt függeszti fel. Fontos megjegyezni, hogy amennyiben egy megszakítási rutinnal szeretnénk közös memóriaterületet (változót; adatstruktúrát) használni, úgy csak a megszakítások letiltása nyújt védelmet az adat korruptálódásának 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 memóriafoglalás 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 későbbiekben), hiszen jól megtervezett szoftver esetén ez csak a program indulásakor fut le.

Az operációs rendszertől *OSMemGet()* függvényhívással kérhetünk egy szeletet a területből, 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 Rasp-

berry Pi használatához szükségesek. Több, mint 35000 csomag elérhető a rendszerhez, és folyamatos fejlesztés alatt áll.

Tekintve, hogy egy teljes értékű linux vált elérhetővé az eszközhöz, az asztali számítógépeknél megszokott módon használható a rendszer. Ebbe beletartozik 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 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[7].

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.

[7] Raspbian-hoz elérhetőek 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 keresztmetszetté 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. A rendelkezésre álló memória sem tekinthető korlátlannak, 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[1].

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:

- **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 mindig ugyan abban a sorrendben fussanak le).

[1] 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 szabvány-sorozat), a valósidejű operációs rendszerek értékelésére ez nem jelent megoldást.

- **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.

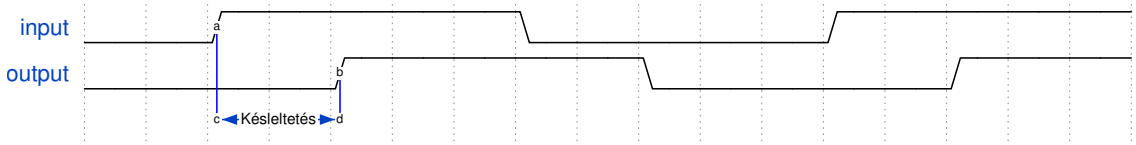
3.1. Szakirodalmakban fellelhető metrikák

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.

3.1.2. Késleltetés

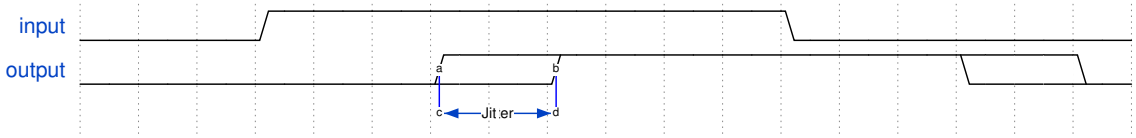
A rendszer késleltetése az az idő, ami egy esemény beérkezésétől a rendszer válaszáig eltelik. Ezt okozhatja a mikrovezérlő megszakítási mechanizmusához szükséges műveletek sora, az operációs rendszer ütemezőjének overhead-je, de a közben végrehajtandó feladat is nagy mértékben befolyásolja a nagyságát.



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

3.1.3. Jitter

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



3.2. ábra. *A késleltetés jitterének szemléltetése.*

3.1.4. Rhealstone

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 Rhealstone nevet adta[2].

[2] A név a Whetstone és Dhrystone 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 Rhealstone metrika célja, hogy a fejlesztőket segítse az alkalmazásukhoz leginkább megfelelő operációs rendszer kiválasztásában.

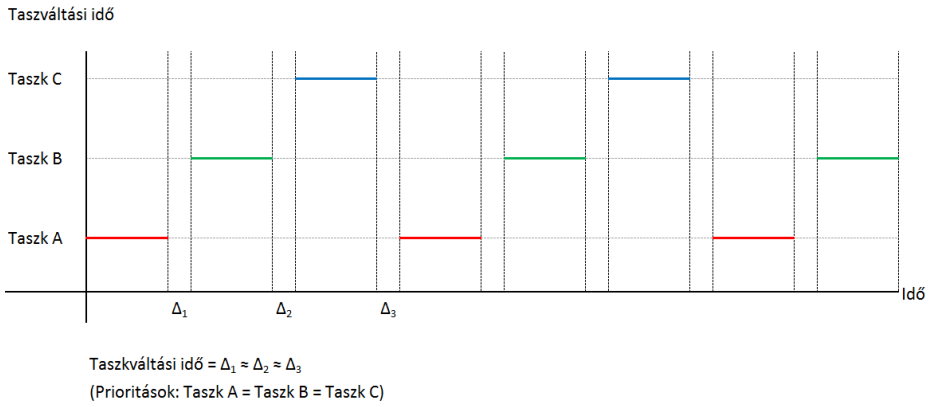
A Rhealstone hat kategóriában vizsgálja meg az operációs rendszer képességeit:

- Taszkváltási idő (Task switching time),
- Preemptálási idő (Preemption time),
- Megszakítás-késleltetési idő (Interrupt latency time),
- Szemafor-váltási idő (Semaphore shuffling time),
- Deadlock-feloldási idő[3] (Deadlock breaking time),
- Datagram-átviteli idő[4] (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. 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 taszváltási 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.

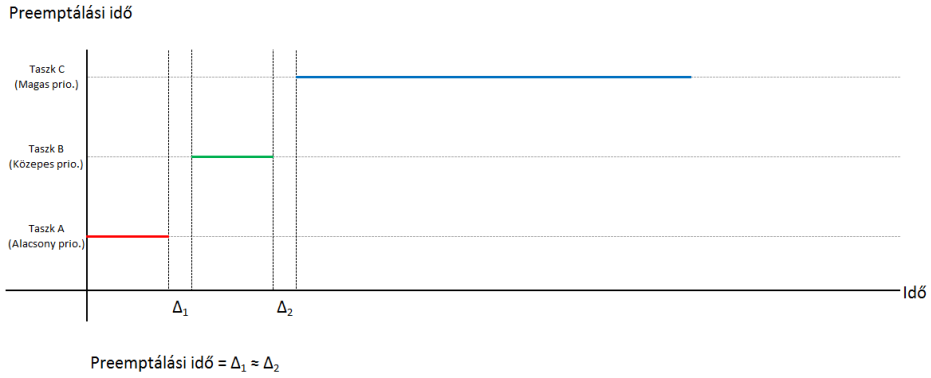
A rendszerek a futtatható taszkokat általában valamilyen listában tárolják, így különböző számú taszkkal elvégezve a mérést más eredményt kaphatunk.

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

[4] Bár a kategória neve időnek nevezi a mért mennyiséget, az eredeti dokumentum alapján a mérés $^{kB}/_{sec}$-ben értelmezett adatot eredményez. Ha az 1 kB átviteléhez szükséges időt mérjük, akkor viszont megoldódik ez a probléma.

Preemptálási idő

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

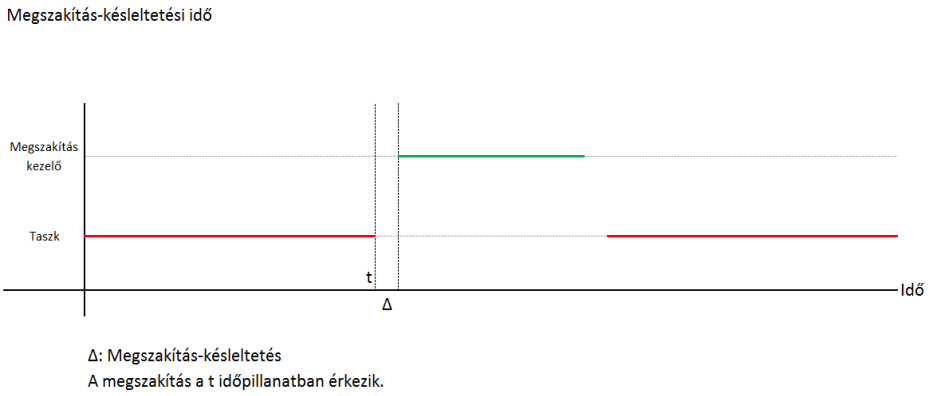


3.4. ábra. *A preemptálási idő szemléltetése.*

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

Megszakítás-késleltetési idő

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



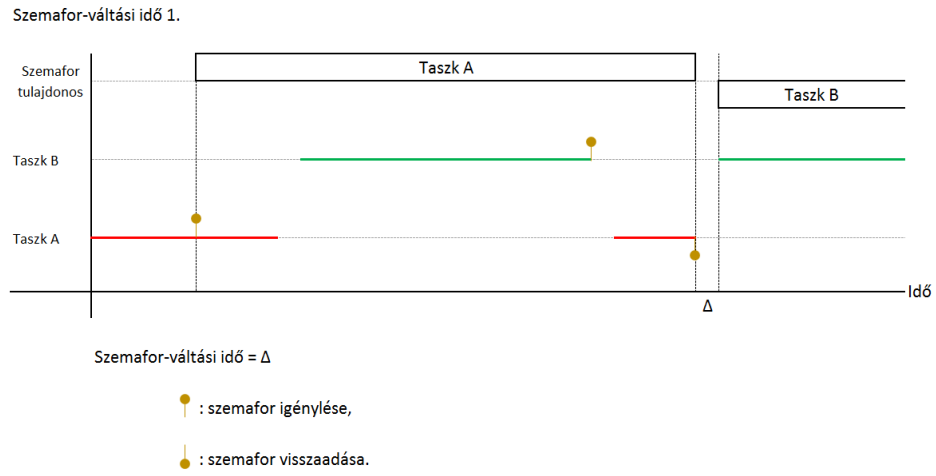
3.5. ábra. *A megszakítás-késleltetési idő szemléltetése.*

Szemafor-váltási idő

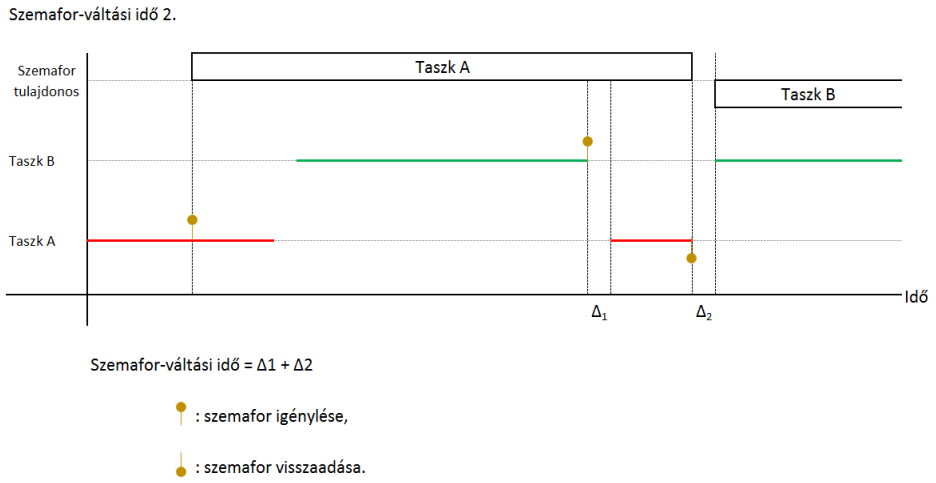
Az 1989-es cikk szerint szemafor-váltási idő az az átlagos időtartam, ami egy szemafor elengedése és egy, a szemaforra várakozó taszk elindulása között eltelik (3.6. ábra).

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).

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



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



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

Deadlock-feloldási idő

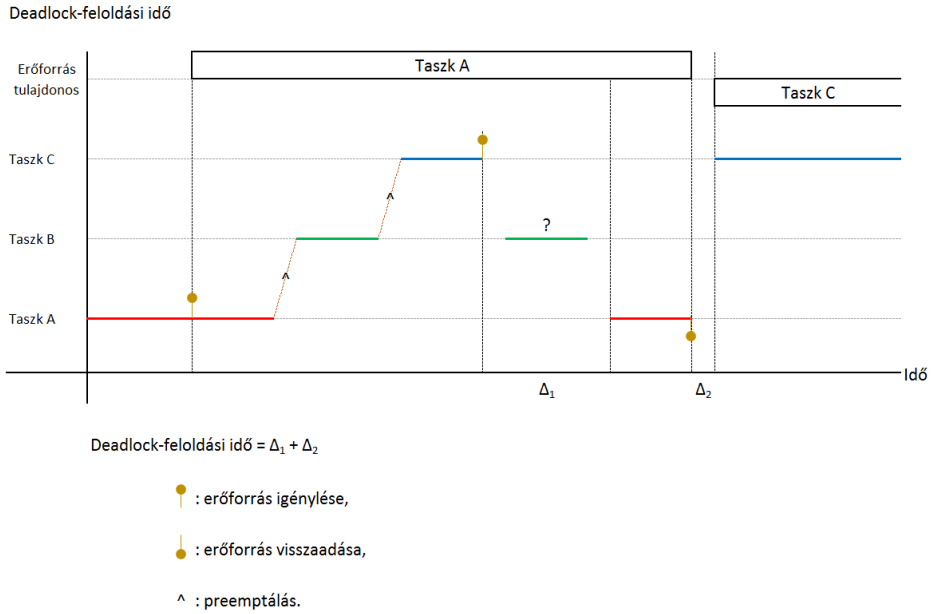
A deadlock-feloldási idő az az átlagos időtartam, ami egy olyan erőforrás eléréséhez szükséges, amit egy alacsonyabb prioritású taszk már birtokol (3.8. ábra).

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

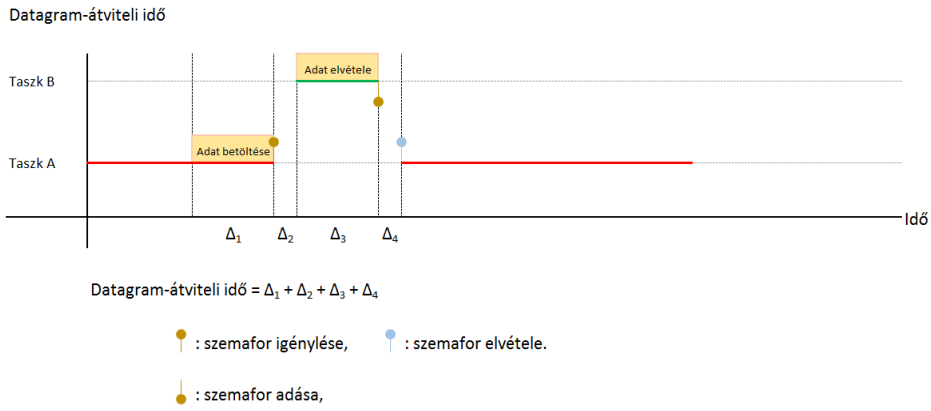
Datagram-átviteli idő

A datagram-átviteli idő a taszkok között elérhető adatsebesség az operációs rendszer objektumait kihasználva (vagyis nem megosztott memórián vagy pointeren keresztül). Az adatküldő taszknak kapnia kell értesítést az adat átvételéről (3.9. ábra).

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

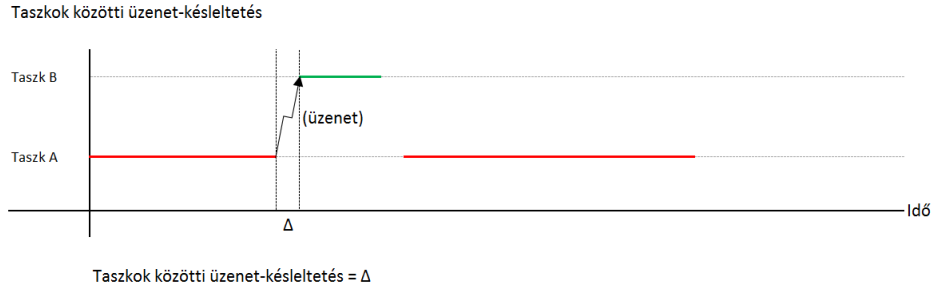


3.8. ábra. *A deadlock-feloldási idő szemléltetése.*



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

objektum kezelésének és az operációs rendszer járulékos műveleteinek hatékonyságának megmérése (3.10. ábra).



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

Rhealstone jellemzők összegzése

Az elvégzett mérések várhatóan mikroszekundum és milliszekumdum nagyságrendű értékeket adnak vissza. Minden értéket másodpercre váltva, majd a reciprokát véve az értékek összegezhetőek egy reprezentatív értékké. Az átváltásnak köszönhetően a nagyobb érték jobb teljesítményt jelent, ami a teljesítménymutatók világában elvárt.

Objektív Rhealstone érték

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

$$r_1 + r_2 + r_3 + r_4 + r_5 + r_6 = {}^{\text{objektív Rhealstone}}/_{\text{sec}}, \tag{3.1}$$

ahol

r_1 a taszkváltási időből származó érték,

r_2 a preemptálási időből származó érték,

és így tovább.

Súlyozott Rhealstone érték

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

$$n_1 \cdot r_1 + n_2 \cdot r_2 + n_3 \cdot r_3 + n_4 \cdot r_4 + n_5 \cdot r_5 + n_6 \cdot r_6 = {}^{\text{alkalmazás specifikus Rhealstone}}/_{\text{sec}}, \tag{3.2}$$

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.

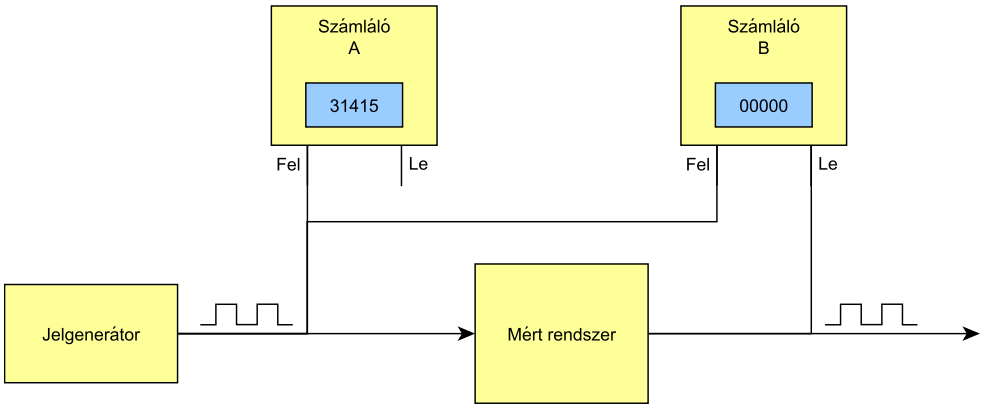
A Társaság egy olyan mérési összeállítást javasol a fejlesztőknek, ami egyszerűsége ellenére képes meghatározni a rendszer legrosszabb válaszidejét (3.11. ábra).

A méréshez szükséges eszközök:

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

Mérési elrendezés

A jelgenerátor kimenetét a mérendő rendszer bemenetére, illetve mindkét számláló *count up* bemenetére kötjük, míg a mérendő rendszer kimenetét a kimeneti számláló *count down* bemenetére kötjük.



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 különbözni fog a bemenetére adott impulzusok számától, amit a számlálók segítségével detektálunk. A kapott frekvencia a legrosszabb válaszidő reciproka.

Mérés menete

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 megmérem.

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,
- Rhealstone értékek (első publikáció szerinti értékek),
- Legrosszabb válaszidő.

4. fejezet

Rendszerterv

5. fejezet

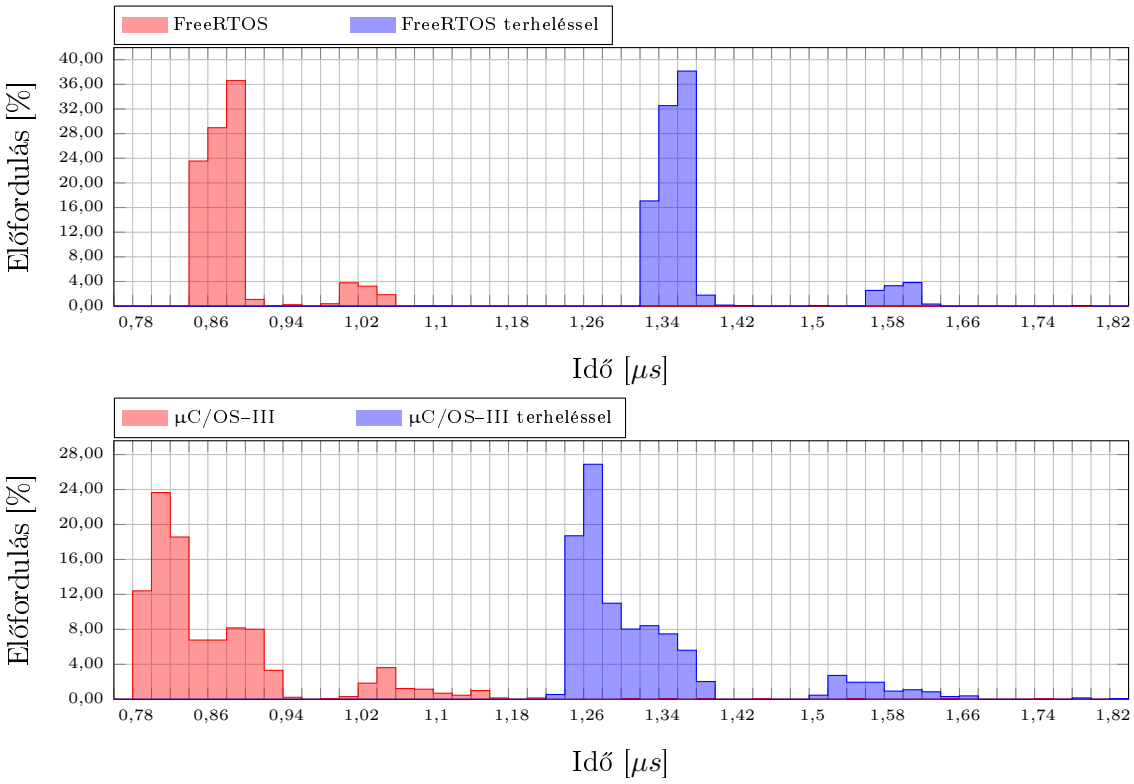
Eredmények kiértékelése

6. fejezet

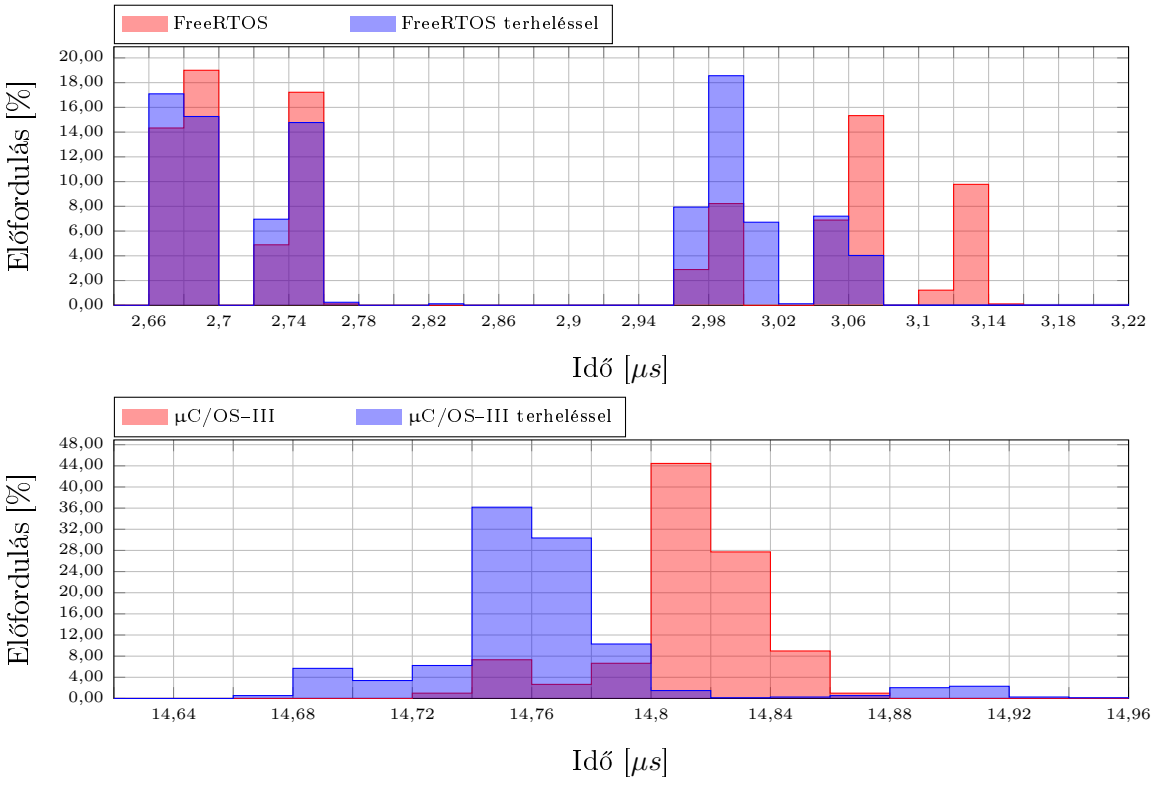
Konklúzió

Köszönetnyilvánítás

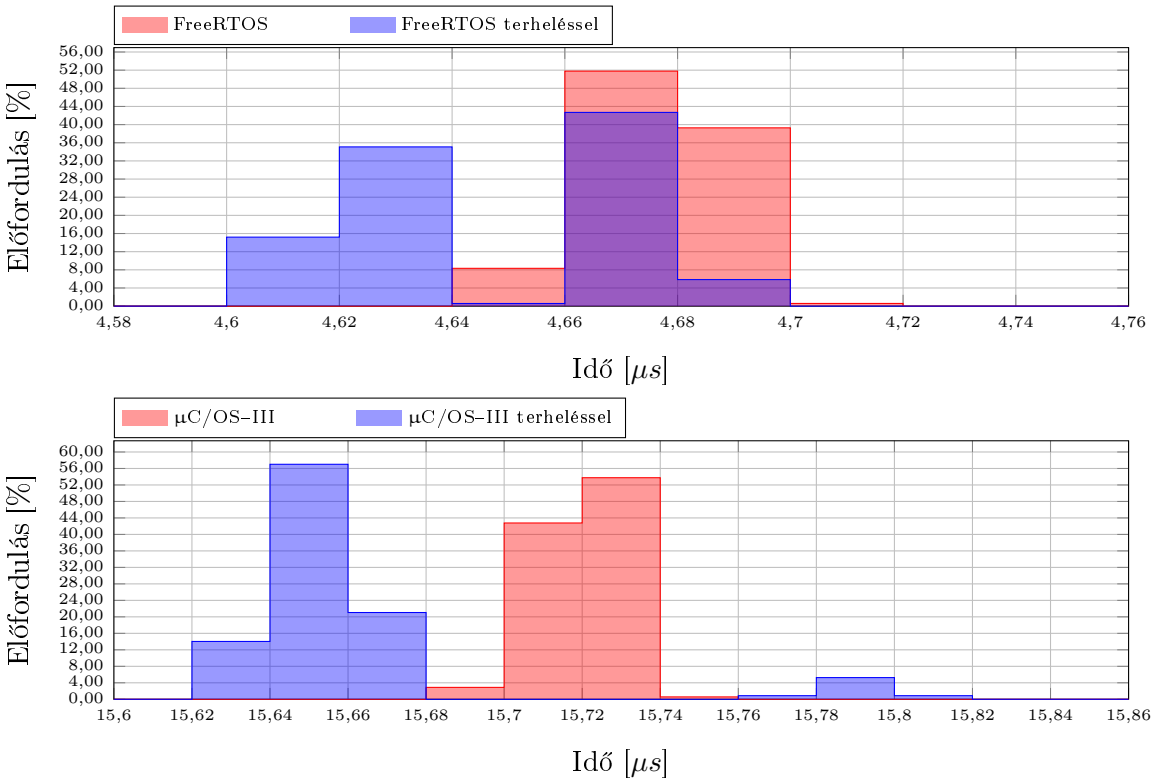
Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.



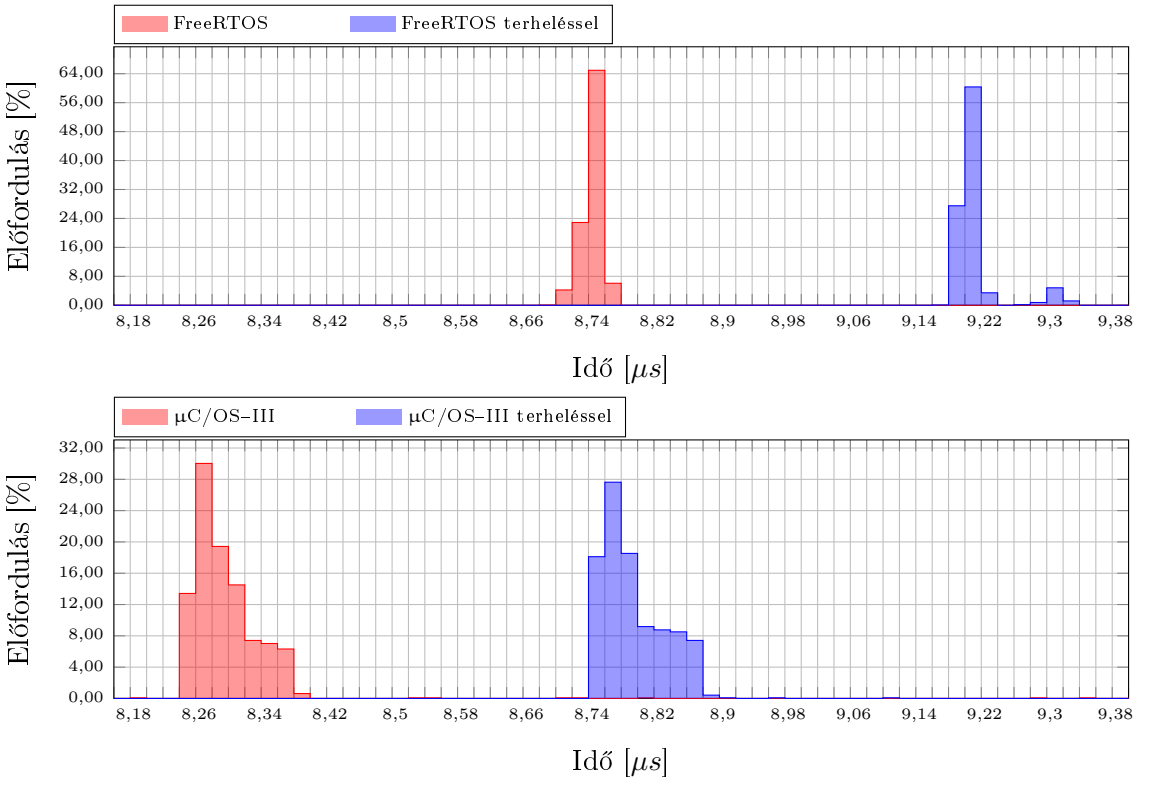
6.1. ábra. *Késleltetés és Jitter terheléssel és terhelés nélkül.*



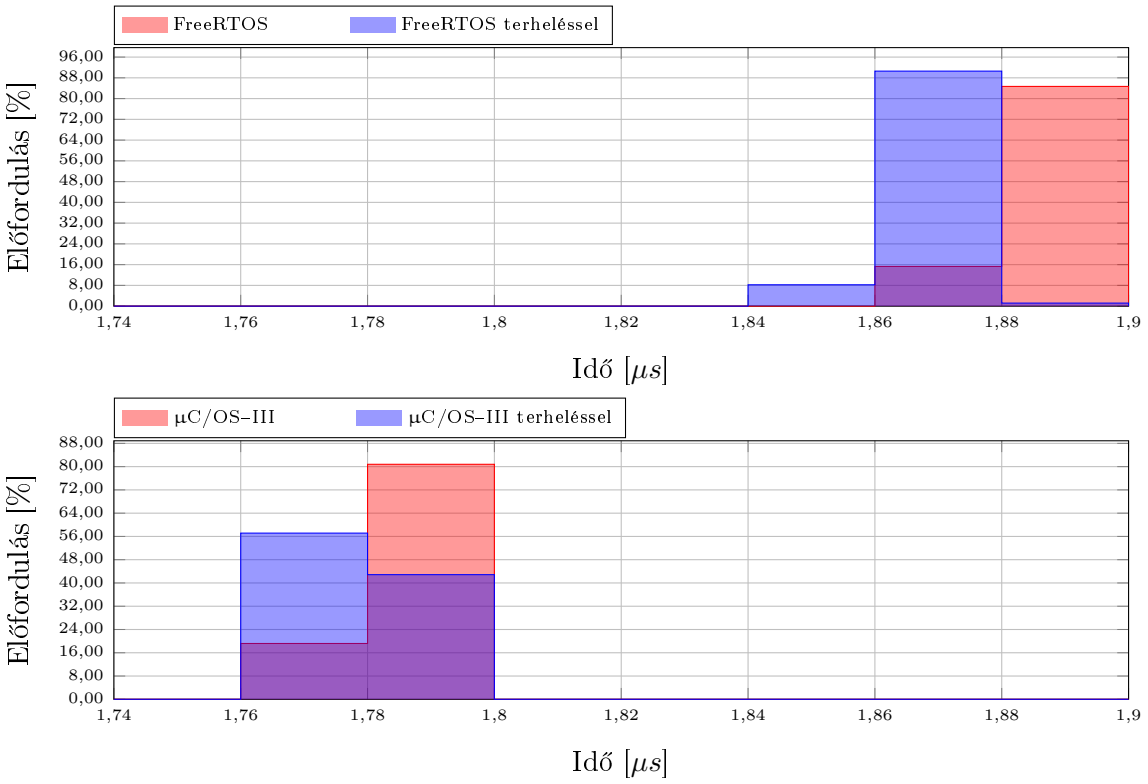
6.2. ábra. *Taszkváltási idő terheléssel és terhelés nélkül.*



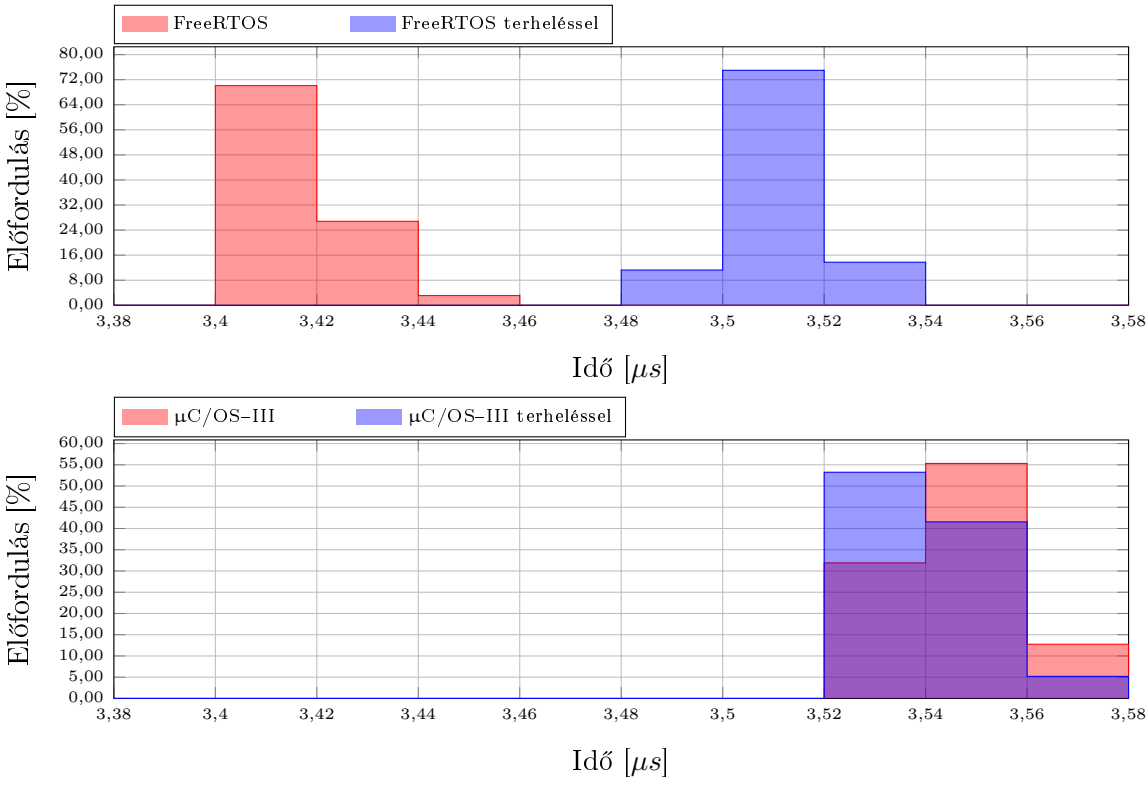
6.3. ábra. *Premptálási idő terheléssel és terhelés nélkül.*



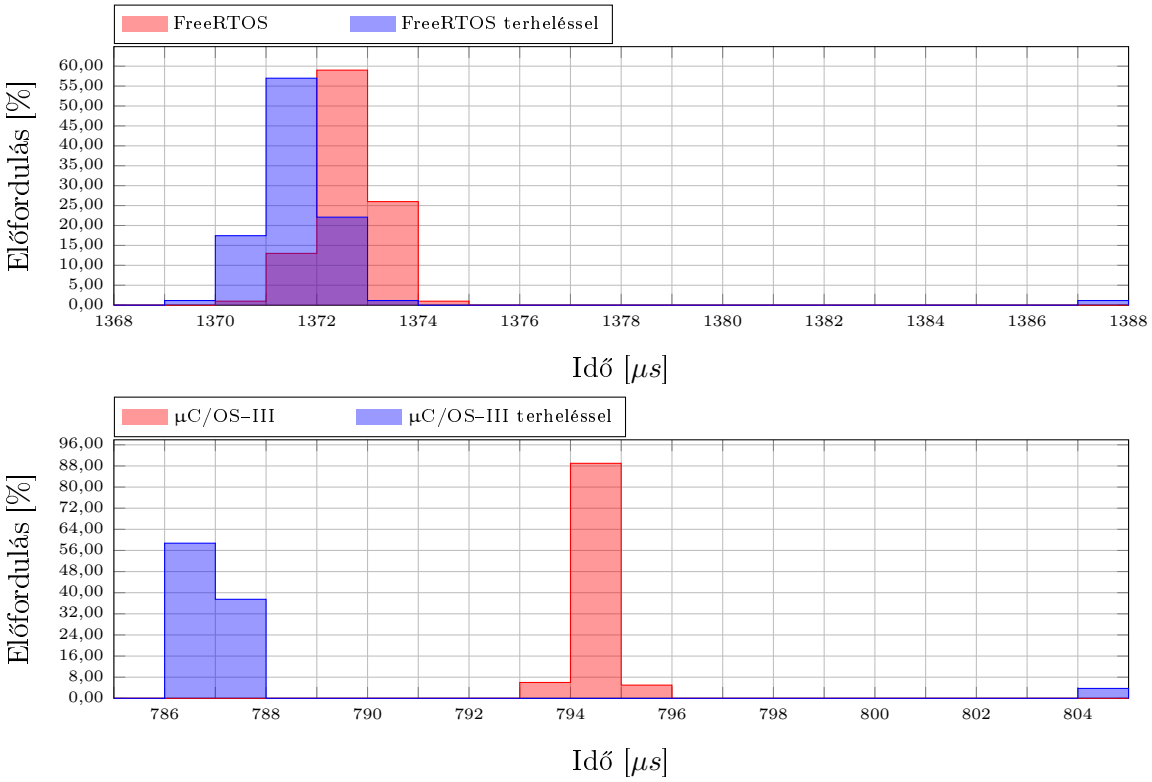
6.4. ábra. *Megszakítás-késleltetési idő terheléssel és terhelés nélkül.*



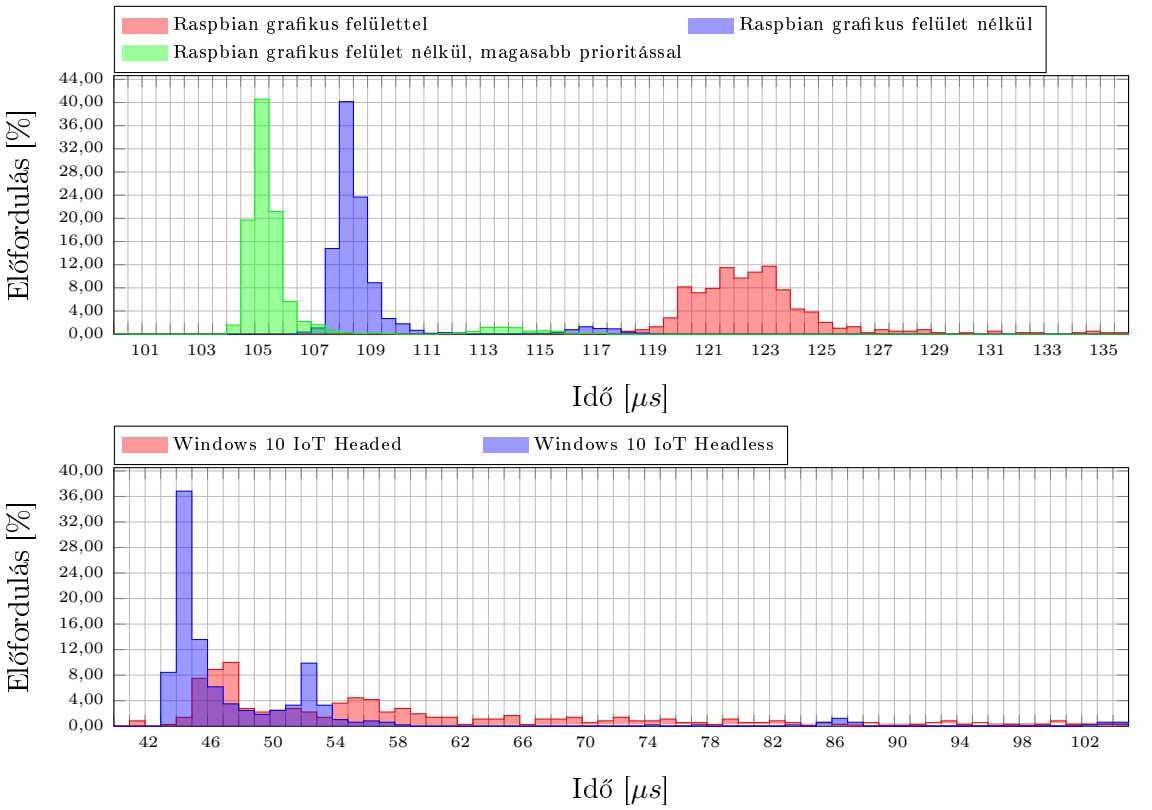
6.5. ábra. *Szemafor-váltási idő terheléssel és terhelés nélkül.*



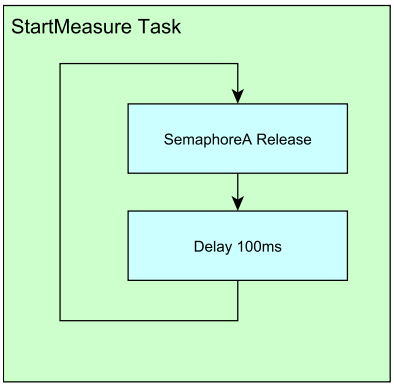
6.6. ábra. *Deadlock-feloldási idő terheléssel és terhelés nélkül.*



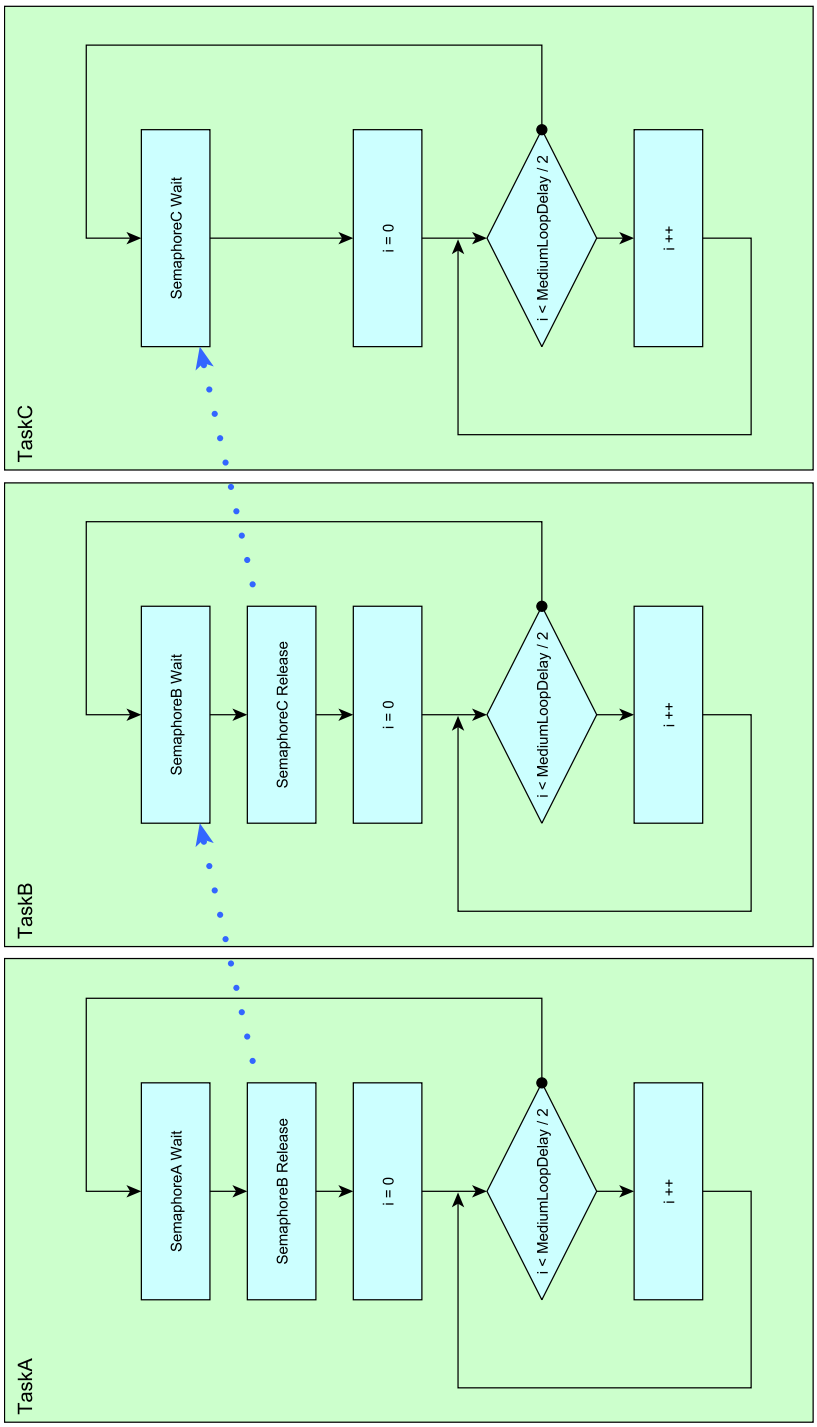
6.7. ábra. *Datagram-átviteli idő terheléssel és terhelés nélkül.*



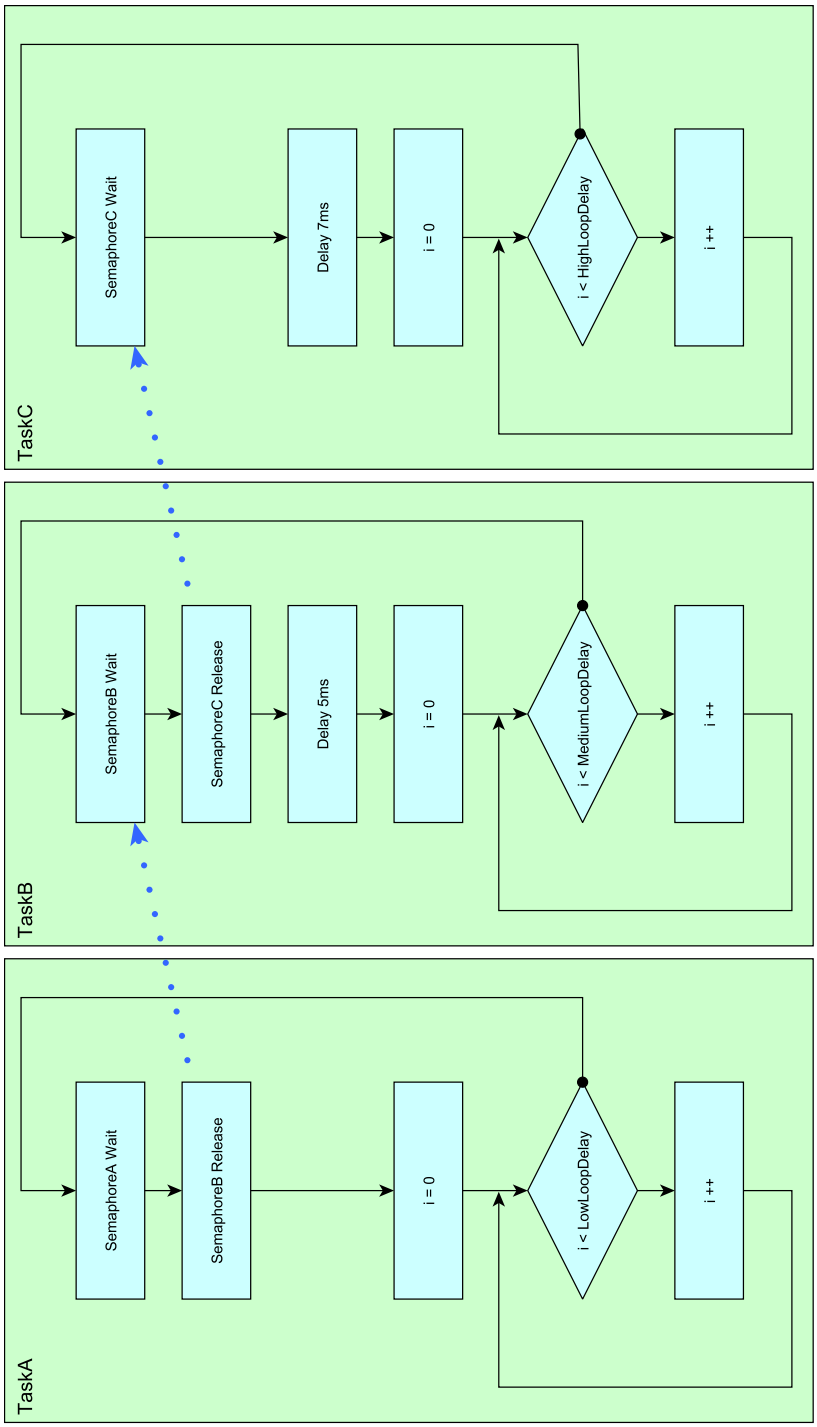
6.8. ábra. *Késleltetés és Jitter Raspberry Pi-n futó rendszereknél terheléssel és terhelés nélkül.*



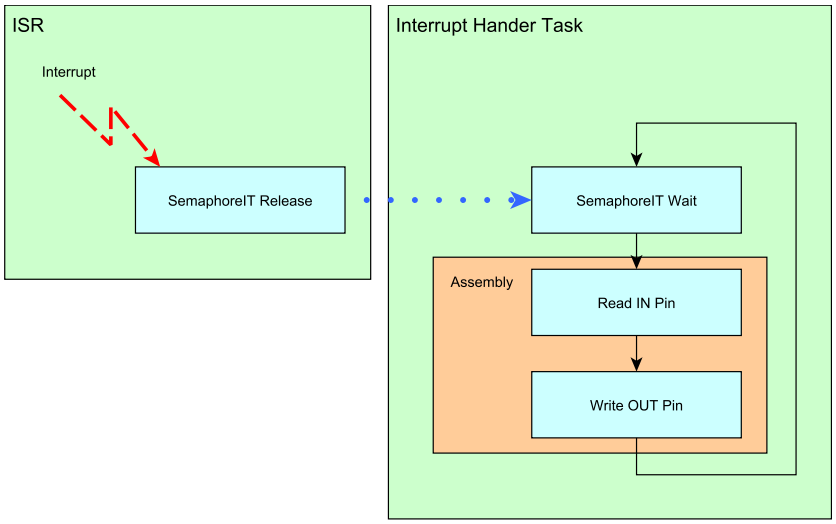
6.9. ábra. *StartMeasure taszk folyamatábrája.*



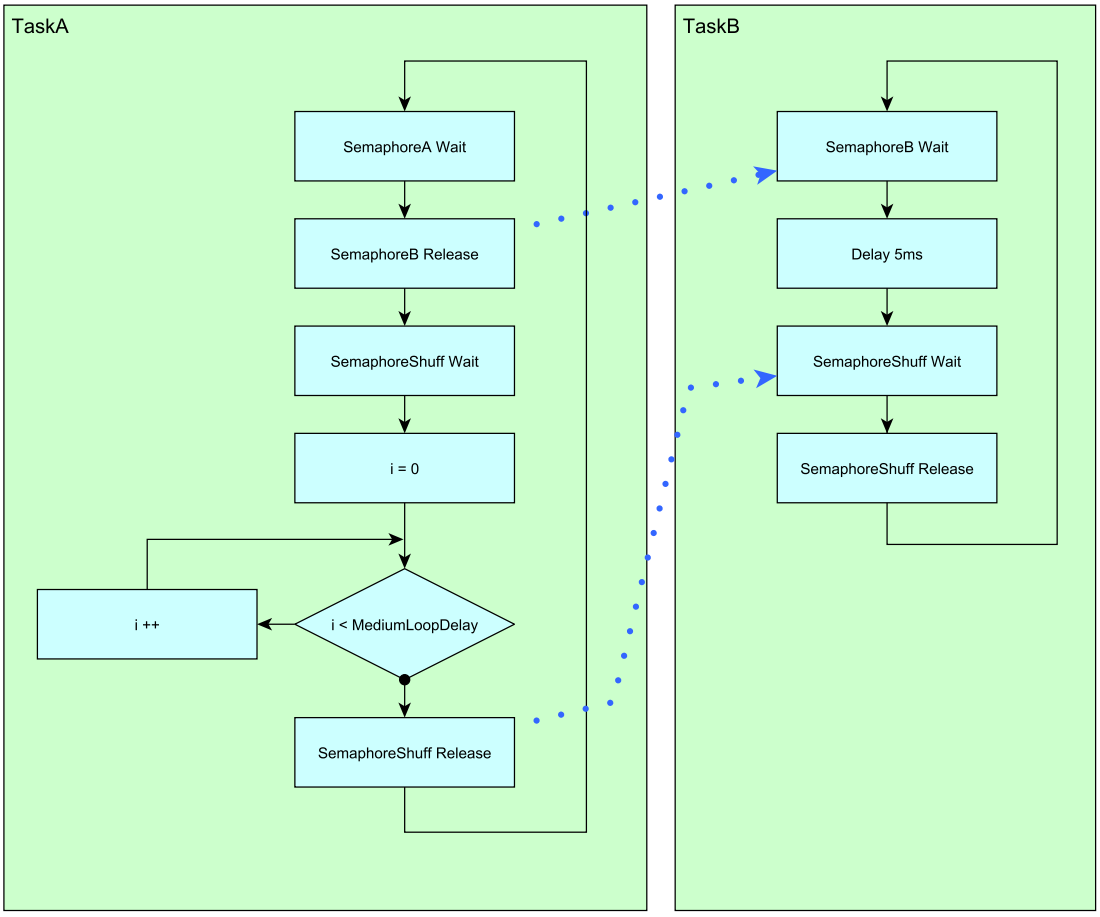
6.10. ábra. *Taszkváltási idő méréséhez használt taszkok folyamatábrája.*



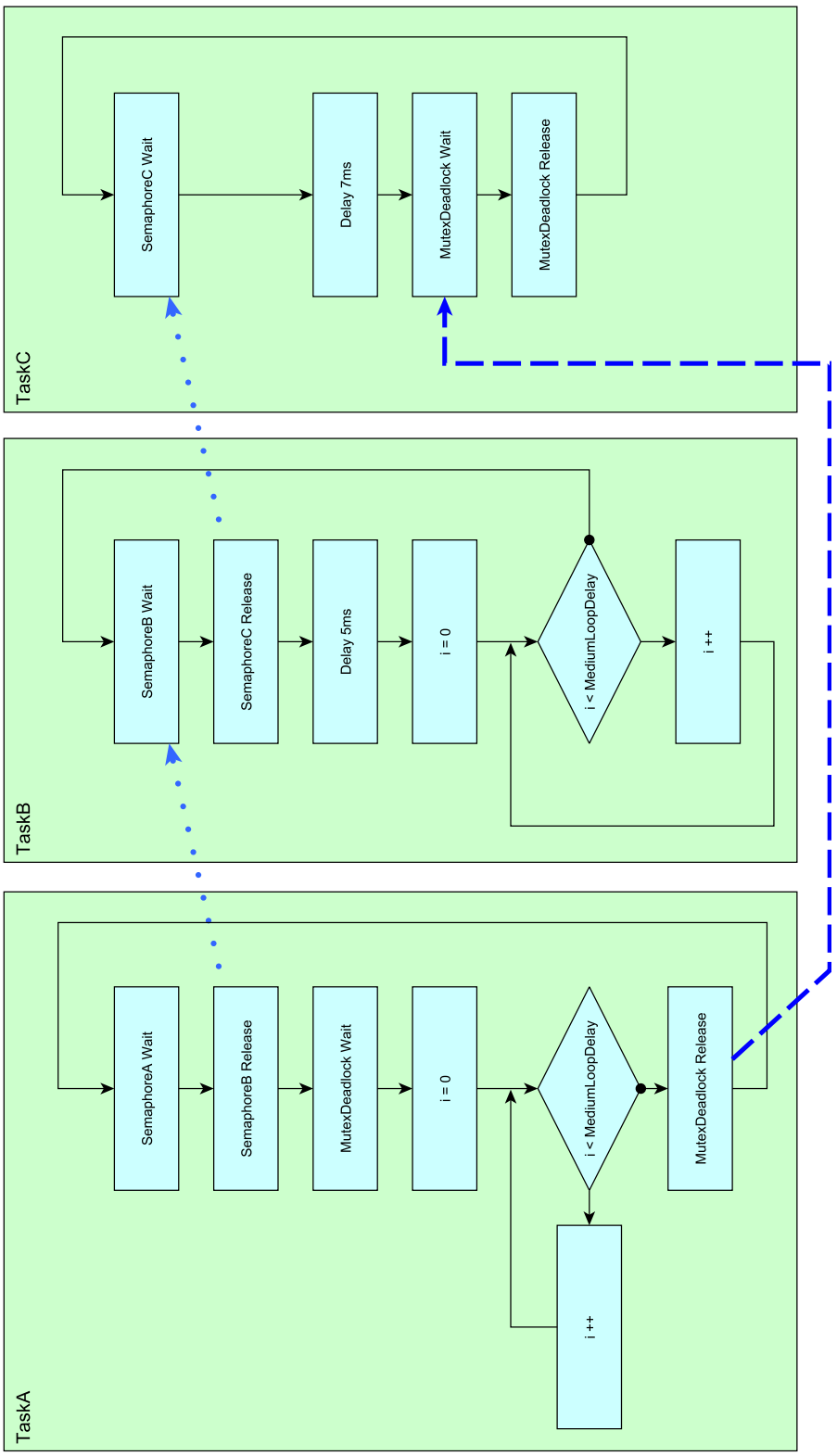
6.11. ábra. *Preemtálási idő méréséhez használt taszkok folyamatábrája.*



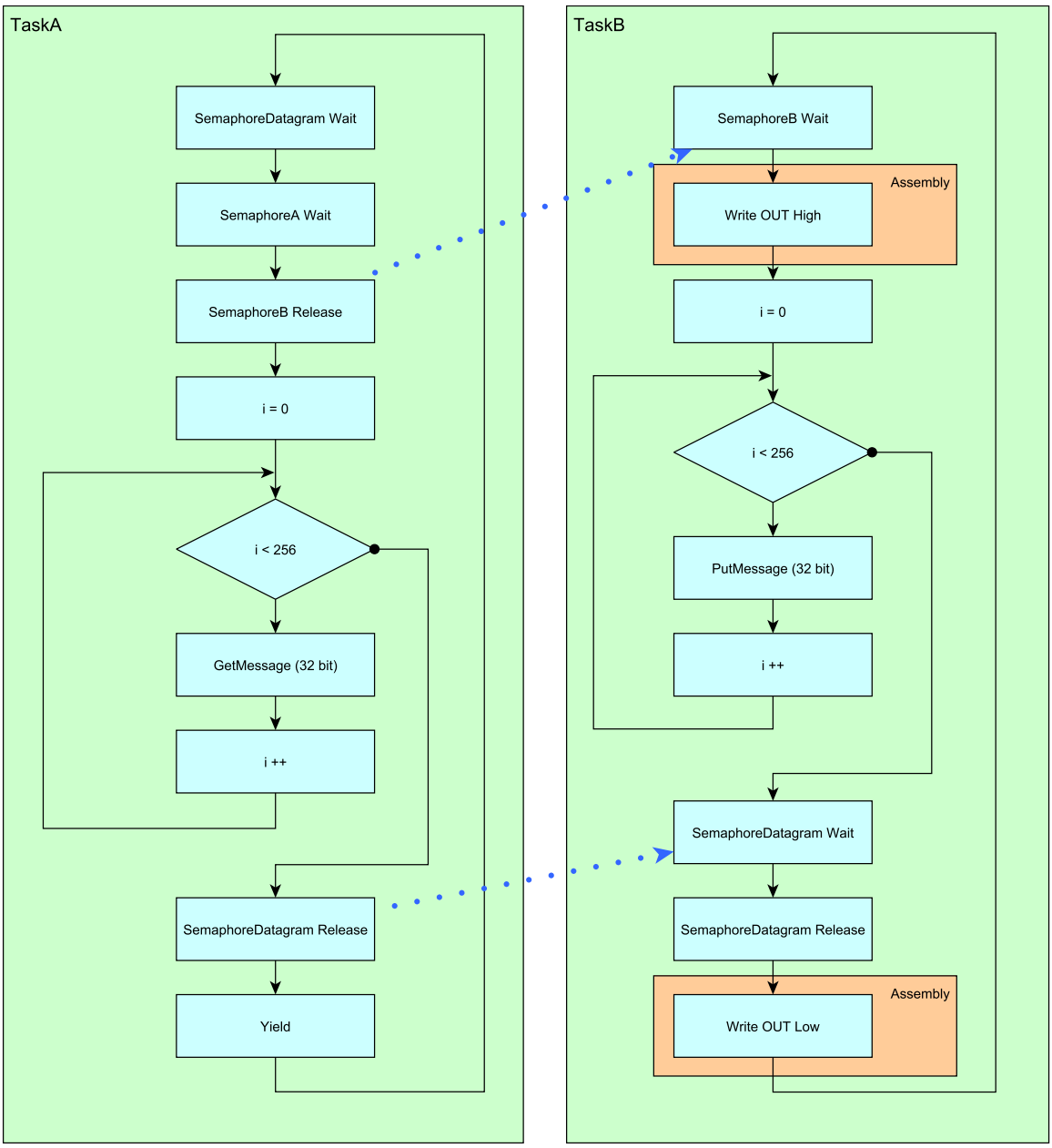
6.12. ábra. *Megszakítás-késleltetési idő méréséhez használt taszk folyamatábrája.*



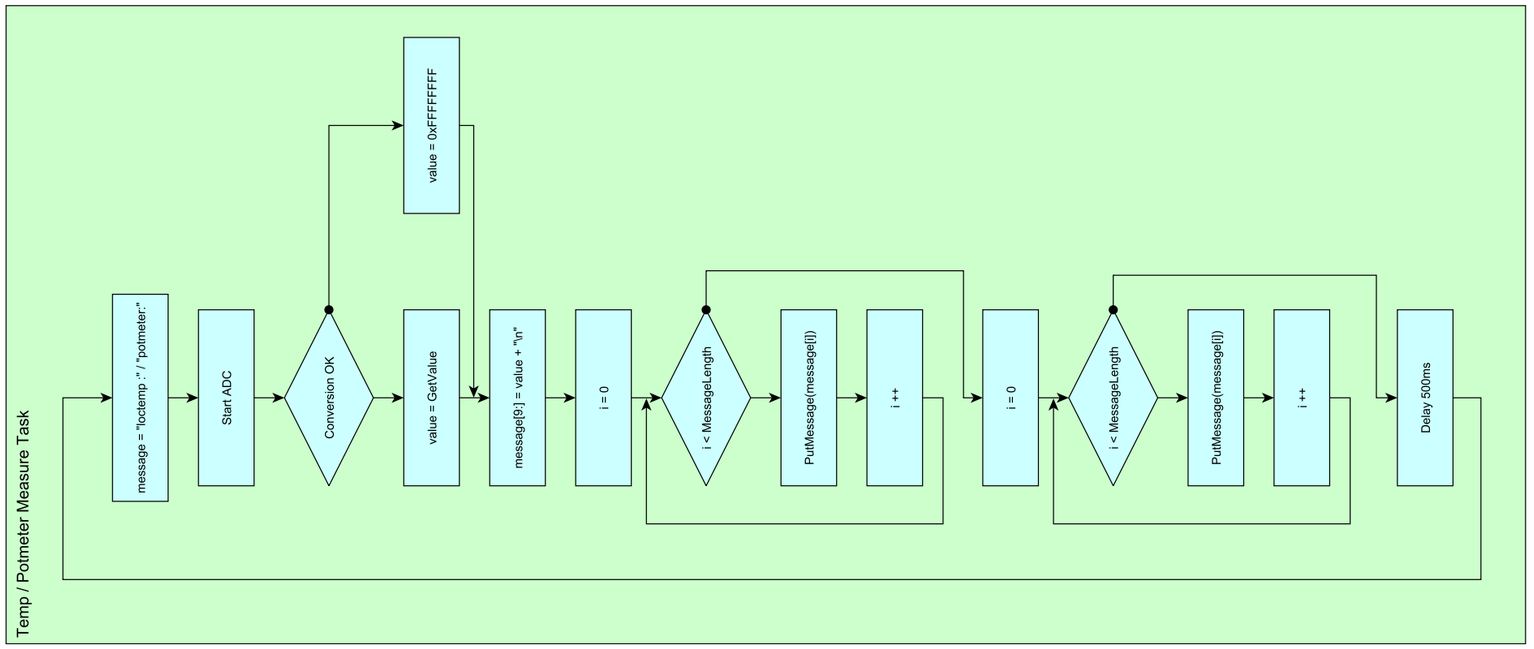
6.13. ábra. *Szemafor-váltási idő méréséhez használt taszkok folyamatábrája.*



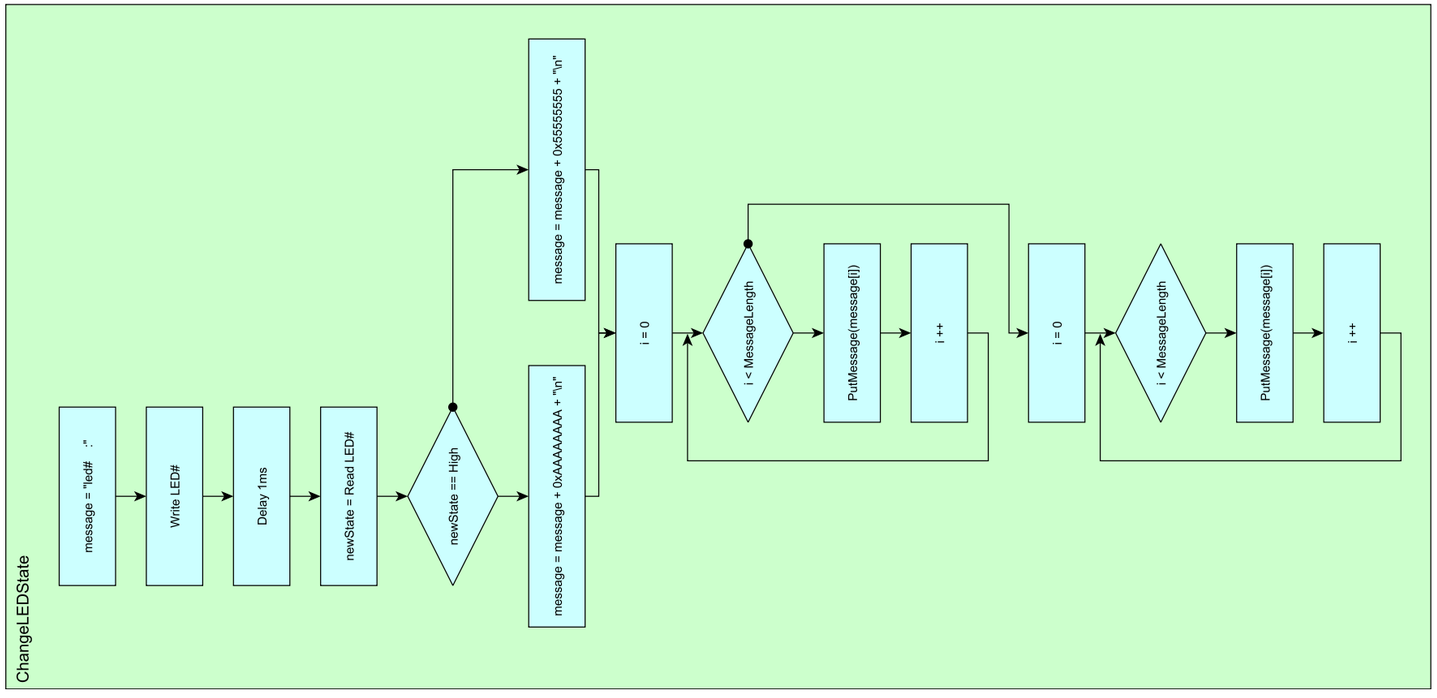
6.14. ábra. *Deadlock-feloldási idő méréséhez használt taszkok folyamatábrája.*



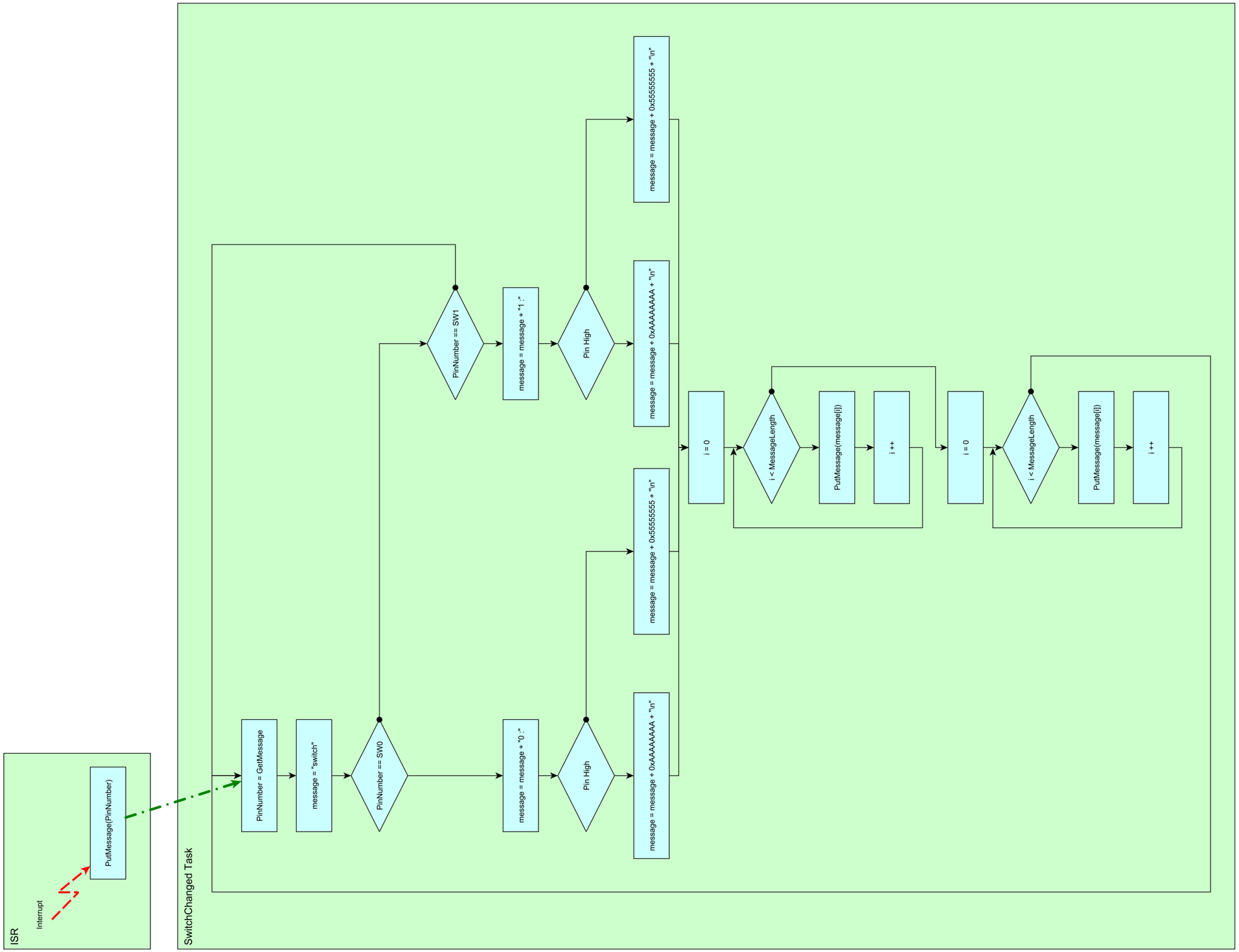
6.15. ábra. *Datagram-átviteli idő méréséhez használt taszkok folyamatábrája.*



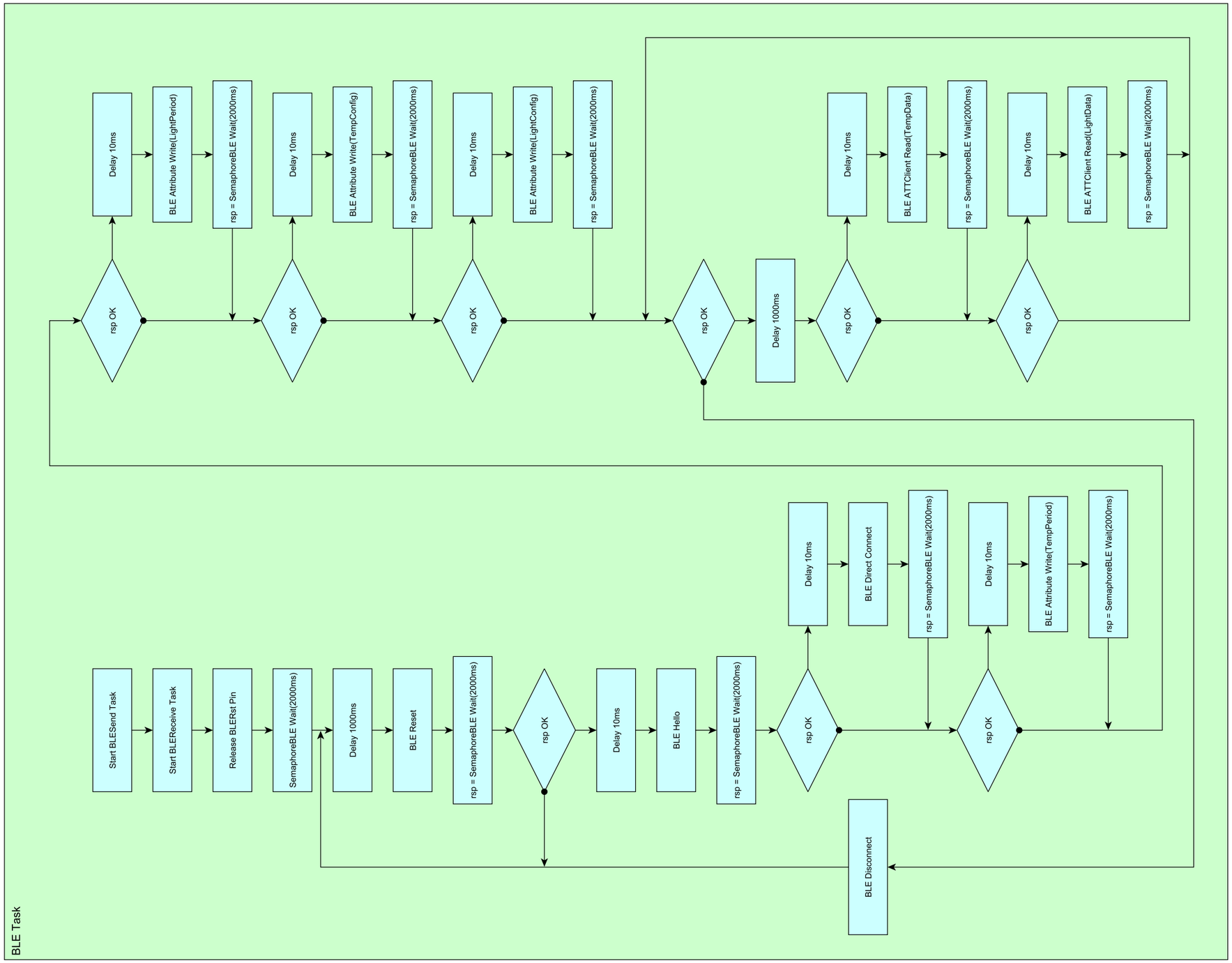
6.16. ábra. *Analog-Digital konverziót végző taszkok folyamatábrája.*



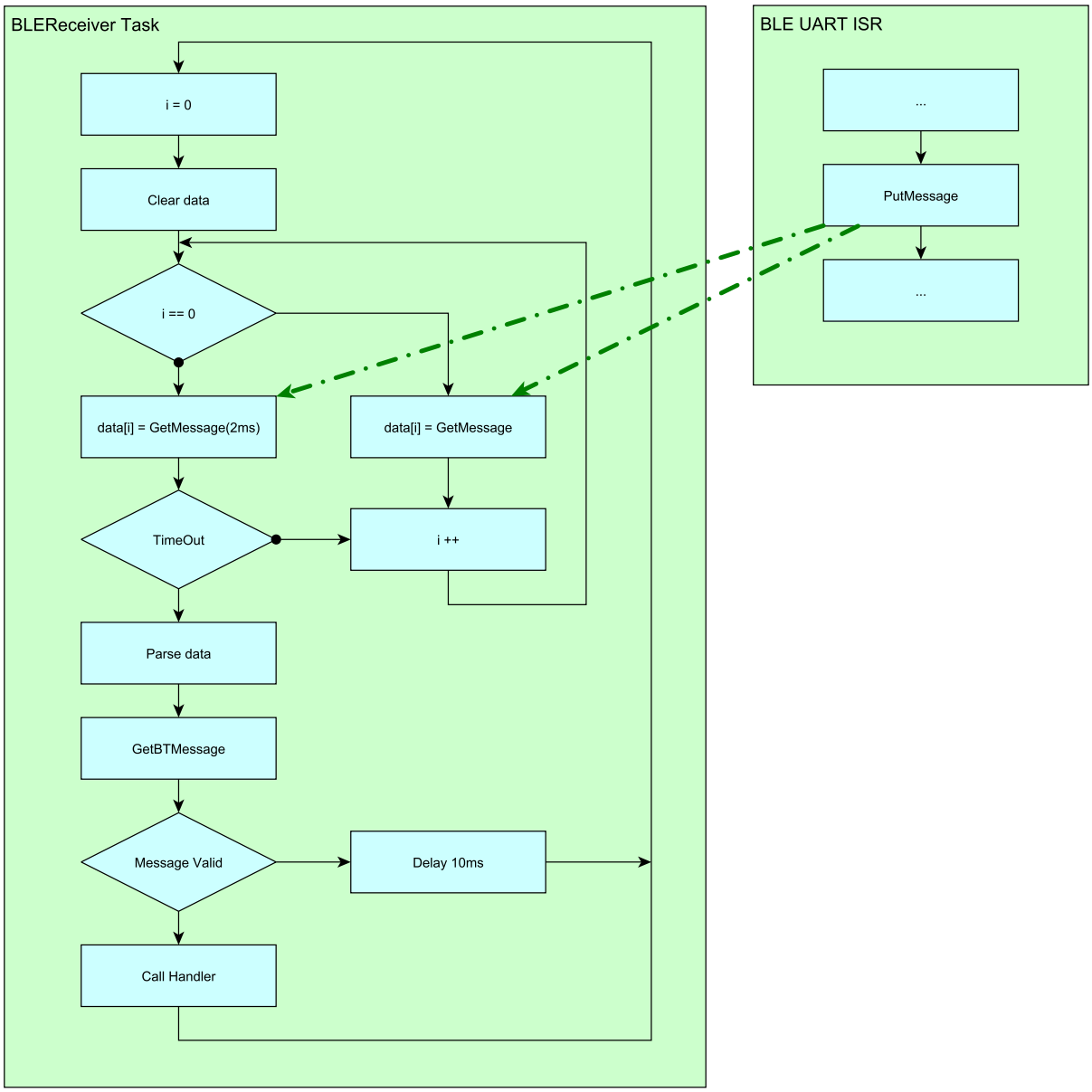
6.17. ábra. *A LED-ek állapotát vezérlő taszkok folyamatábrája.*



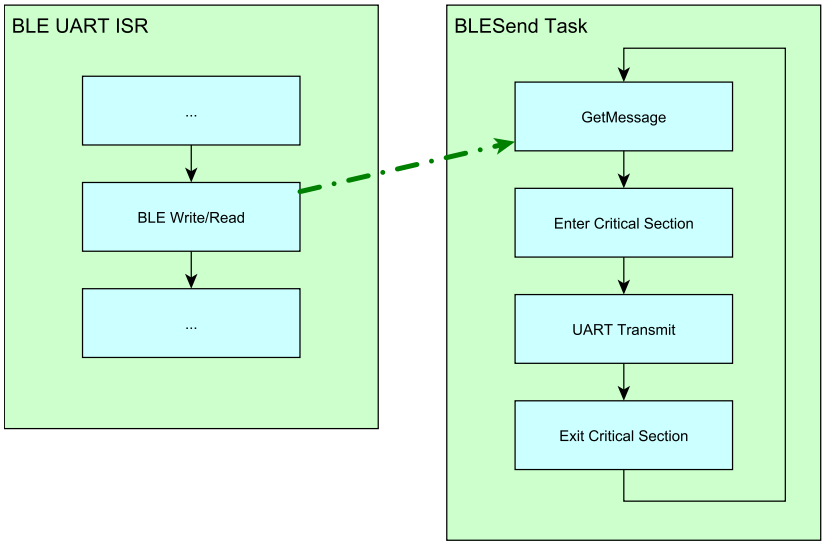
6.18. ábra. *Kapcsolók változását kezelő taszkok folyamatábrája.*



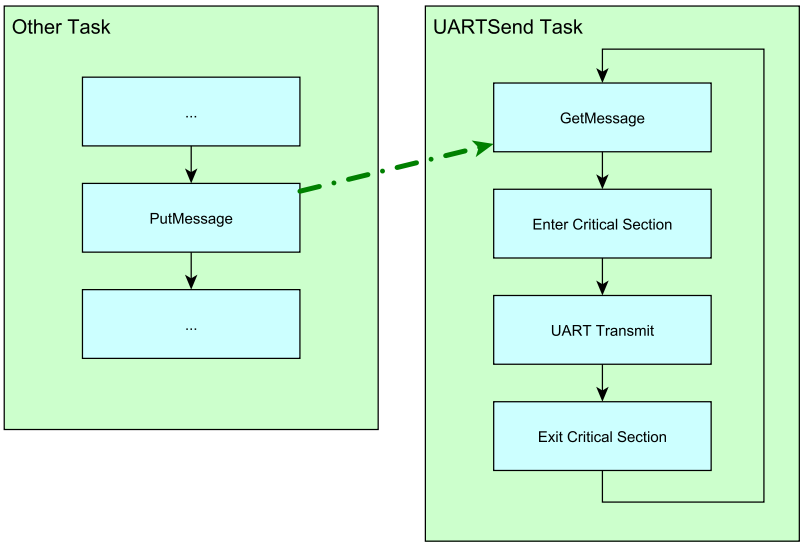
6.19. ábra. *BLE112 modult vezérlő taszk folyamatábrája.*



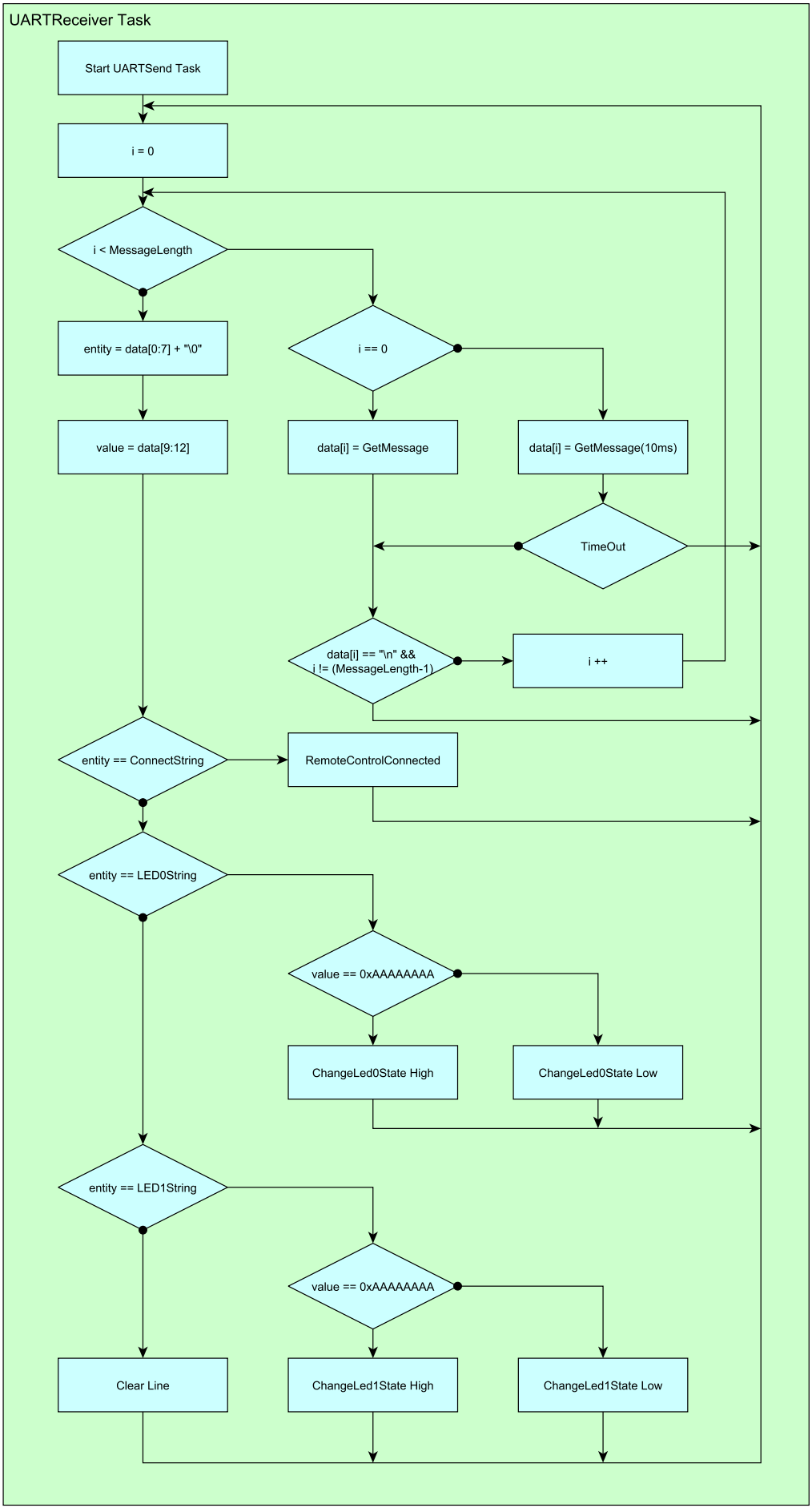
6.20. ábra. *BLE112 modultól érkező válaszok feldolgozását taszk folyamatábrája.*



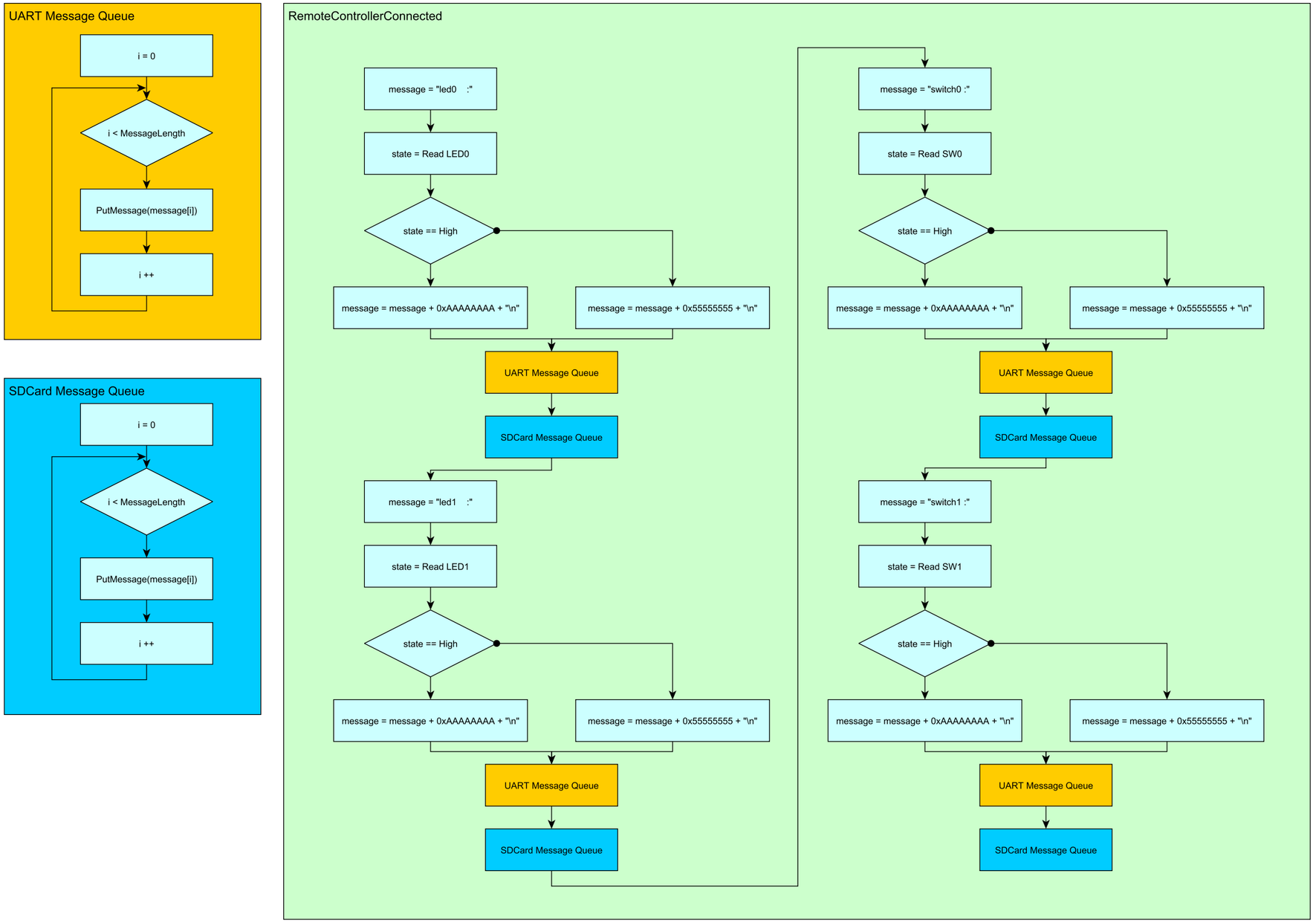
6.21. ábra. *BLE112 modulnak küldött üzenetek továbbítását végző taszk folyamatábrája.*



6.22. ábra. *A vezérlőnek küldött üzenetek továbbítását végző taszk folyamatábrája.*



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



6.24. ábra. *A vezérlő program csatlakozását kezelő taszk folyamatábrája.*

Ábrák jegyzéke

Táblázatok jegyzéke

Irodalomjegyzék

[1] James C. Candy. Decimation for sigma delta modulation. *IEEE Trans. on Communications*, 34(1):72–76, January 1986.

[2] Peter Kiss. *Adaptive Digital Compensation of Analog Circuit Imperfections for Cascaded Delta-Sigma Analog-to-Digital Converters.* PhD thesis, Technical University of Timişoara, Romania, April 2000.

[3] Wai L. Lee and Charles G. Sodini. A topology for higher order interpolative coders. In *Proc. of the IEEE International Symposium on Circuits and Systems*, pages 459–462, Philadelphia, PA, USA, May 4–7 1987.

[4] Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar. Diplomaterv portál (2011 február 26.). `http://diplomaterv.vik.bme.hu/`.

[5] Richard Schreier. *The Delta-Sigma Toolbox v5.2.* Oregon State University, January 2000. URL: http://www.mathworks.com/matlabcentral/fileexchange/.

[6] Ferenc Wettl, Gyula Mayer, and Péter Szabó. *LaTeX kézikönyv.* Panem Könyvkiadó, 2004.

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

Második dolog

B.1. Még egy kis melléklet