Herczig Ádám

Szakdolgozat

Hőmérő hálózat

IoT felület

kialakítása

PÉCSI TUDOMÁNY EGYETEM

MŰSZAKI ÉS INFORMATIKAI KAR

VILLAMOSMÉRNÖKI SZAK

Pécs, 2017

Nyilatkozat

Alulírott Herczig Ádám diplomázó hallgató, kijelentem, hogy a szakdolgozatomat a Pécsi Tudományegyetem Műszaki és Informatikai karán készítettem a villamosmérnöki BSC diploma megszerzése végett.

Kijelentem, hogy a szakdolgozatom érdemi részét egyedül végeztem el. Az érdemi részen kívül csak meghivatkozott forrásokat használtam fel a szakdolgozatomba (szakirodalom, kód). Tudomásul veszem, hogy a szakdolgozatom leírt forráskódot a Pécsi Tudományegyetem, valamint a szakdolgozat témát kiíró konzulens saját céljaira szabadon felhasználhatja.

2017.10.24.

Herczig Ádám

Tartalomjegyzék

[1. Bevezetés 3](#_Toc500369644)

[2. Hardware 5](#_Toc500369645)

[2.1. Raspberry Pi 2 B 5](#_Toc500369646)

[2.2. UART 7](#_Toc500369647)

[2.3. Rendszer Inicializálás 8](#_Toc500369648)

[3. Segédprogramok 12](#_Toc500369649)

[3.1. Make 12](#_Toc500369650)

[3.2. Git 14](#_Toc500369651)

[3.3. Code::blocks 16](#_Toc500369652)

[3.4. Valgrind 16](#_Toc500369653)

[3.4.1. Valgrind Használata 17](#_Toc500369654)

[3.5. Dia 17](#_Toc500369655)

[3.6. Logrotate 18](#_Toc500369656)

[4. Program 20](#_Toc500369657)

[4.1. Tervezés 20](#_Toc500369658)

[4.2. Konfigurációs állomány beolvasása 22](#_Toc500369659)

[4.3. Soros port 24](#_Toc500369660)

[4.4. TAILQ és Mutex inicializálás 27](#_Toc500369661)

[4.5. Szálkezelés 28](#_Toc500369662)

[4.6. Rendszer naplózás 30](#_Toc500369663)

[4.7. Mozgóátlag 30](#_Toc500369664)

[4.8. Futó hiszterézis 31](#_Toc500369665)

[4.9. Olvasás 31](#_Toc500369666)

[4.10. Írás és küldés 36](#_Toc500369667)

[4.11. Feldolgozás 38](#_Toc500369668)

[4.12. Jelzéskezelés 40](#_Toc500369669)

[4.13. Program Befejezés 40](#_Toc500369670)

[5. Tesztelés 40](#_Toc500369671)

[6. Befejezés/összegzés 41](#_Toc500369672)

# Bevezetés

A mai hétköznapi életnek lassan már szerves része lesz, hogy mindenféle eszközt rá fogunk tudni csatolni az internet nagy világára. A legáltalánosabb szenzoroktól kezdve, az autóiparban használt robotokon, aktuátorokon át egészen a hétköznapi használatú eszközökig (Televízió, hűtő, riasztóközpont) és még sorolhatnánk tovább is, szinte mindent rá tudunk csatolni az internetre. Ezáltal minden olyan eszköz mely IP címmel rendelkezik, kapcsolatban lehet más egyéb hálózati eszközökkel, amennyiben ez szükséges.

Gondoljunk csak bele, megyünk haza munkából, okos telefonunkkal egy pillanat alatt leellenőrizhetjük a hűtőnk tartalmát vagy azt, hogy a lakásunk különböző pontjain milyen hőmérséklet van. Hideg téli estéken akár, ezáltal előre fel tudjuk fűteni számunkra megfelelő hőmérsékletűre a lakásunkat hőmérő szenzorok leolvasásával és egy termosztát vezérlésével, mindezt távolról, okos telefonunk segítségével mire hazaérnénk. Ezt nevezzük a magyar fordításban dolgok Internetének, szakkifejezéssel IoT-nek azaz „Internet of Things”-nek. „Gyakorlatilag az IoT a fizikai tárgyak, eszközök, járművek, épületek és egyéb beágyazott elektronikai elemek hálózati kapcsolatát és adatcseréjét jelenti.” Ezen IoT eszközök rendelkeznek valamilyenfajta operációs rendszerrel. A választék ezen rendszerekből lehet akár a Linux valamely disztribúciója, az ARM Mbed, a RIOT, az újabban egyre jobban felkapott Google Brillo vagy akár a Windows 10 for IoT, de ezeken kívül lehetne még sorolni jó pár, direkt erre a területre kifejlesztett rendszert.

Mivel hálózatba kapcsolható eszközökről beszélünk, számolnunk kell a nagyobb biztonsági kockázattal is. Ha óvatlanul van kiépítve egy ilyen hálózat vagy nem megfelelő az eszközök kommunikációjának titkosítása, akkor könnyen egy rossz szándékú hacker áldozatává válhatunk. Gond nélkül figyelheti az adat mozgást, annak gyakoriságát, esetleg befolyásolhatja is egy- egy eszköz működését. Ezért kellően jól kell megtervezni ezeket a hálózatok mind lakásunkon belül mind munkahelyeinken, ha ilyen munkakörben dolgozunk.

Szakdolgozatom témájának pont egy hasonló IoT feladatot választottam, melyet az előbb példának hoztam fel. Miután érdekeltnek tartom magam a Linux rendszerek programozásában, fejlesztésében, feladatként egy hasonló hálózat kialakítását kellett megterveznem és megvalósítanom. A feladat során egy mikrokontrollerekből álló hálózat központi vezérlőjének mérésadatgyűjtő szoftverét készítettem el. A mikrokontrollerekkel való kommunikáció soros felületen lett megvalósítva. Az eszközök állapotát és azok mért adatait paraméterezhető időközönként kellett lekérni, a szoftver segítségével. Ezen felül, a rendszernek tudnia kellett eszközönként előfeldolgozást beállítania. A mérési időn kívül, további fontos paramétereket a szoftvernek egy konfigurációs állományból kellett kiolvasni. A kiolvasott adatokat és állapotokat text fájlba kellett kiírnia a programnak. Az eszközök státuszairól statisztikai listák készítése volt még szempont, melyet szintén állományba kellett vezetni.

A rendszer központja, melyre készült maga a szoftver, Raspberryi PI 2B típusú eszköz volt. A program egész része C nyelven lett megírva. A mikrokontrollerek száma melyekről adatgyűjtést végez a szoftver véges, egy maximum értékben határoztam meg mennyit legyen képes kezelni a központi vezérlő. Persze ez a maximum érték módosítható szám. Maga a szoftver a sikeres konfigurációs fájl felolvasása utána a soros port és a TAILQ nevezetű FIFO[[1]](#footnote-1) inicializálásával foglalkozik. Ahhoz, hogy a rendszer olvasni tudjon adatokat a mikrokontrollerekből, parancsokat kellett kiküldeni feléjük majd a beolvasott adatokkal dolgoznia is tudnia kellett a központnak. Ezt összeszámolva 3 darab, teljesen különálló folyamatnak felelt meg. Miután ezeknek egymástól párhuzamosan kellett futniuk, elengedhetetlen volt a szálak belecsempészése a programba. A mért adatoknak 2 darab, előfeldogozó mérőalgoritmuson kellett végig menniük, hogy a felhasználó pontos, kisimított hőmérsékleti statisztikát lásson. A sikeresen feldolgozott adatokat és a hibás csomagokat vagy bármi hiba jelentkezését a rendszer naplójába, syslogba vezettem időbélyeggel ellátva.

A program mellé a könnyű portolhatóság végett készült egy Makefile is.

# Hardware

## Raspberry Pi 2 B

A hardware melyre a program készült egy Raspberry Pi 2 B típusú SoC[[2]](#footnote-2) volt. Ez az eszköz egy nagyjából bankkártya méretű elektronikai eszköz mely méretéhez képest rengeteg funkcióval és lehetőséggel rendelkezik.

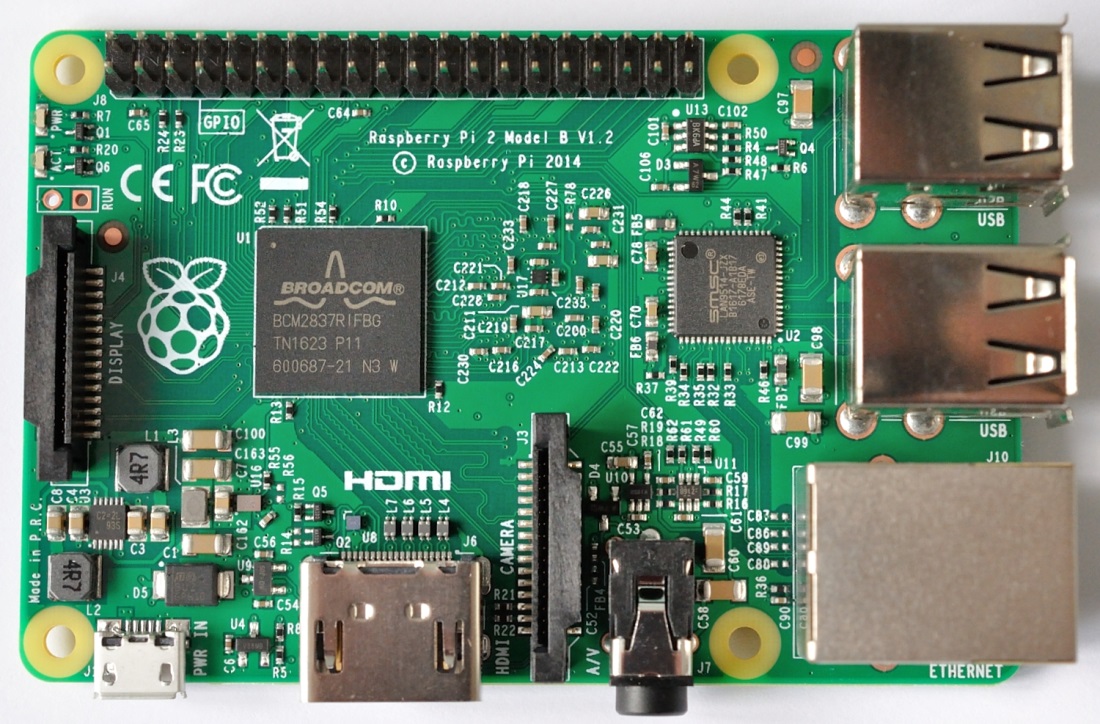
A hardware „szíve” egy Broadcom BCM2836 processzor mely mellé 1GB memóriát illesztettek a fejlesztők. A processzor 900MHz-es órajellel fut, ezzel is biztosítva a gyors és viszonylagos nagy teljesítményt a méretekhez képest. Magára az eszközre 40 db-os apa aljzatú általános célú be és kimeneti portot implementáltak a mérnökök, melyek különböző kommunikációs protokollt valamint kimeneti vezérlést biztosítanak további fejlesztőknek. Az eszközön található UART, SPI(2 db) és I2C (2 db) kommunikációs interface. Továbbá, az eszközön még található egy 100Mbit-es Ethernet csatoló és 4db USB-port. Sajnos a 2-es verzióba még nem található bluetooth és a wifi adapter, de 3-as verzióba ezek hiányát már pótolták a tervezőmérnökök. Viszont egy egyszerű USB-s wifi adapterrel könnyen áthidalható hálózati probléma a 2-es verzión. Az eszköz lehetővé teszi kamera, HDMI-s kijelző, touchpad és 3.5mm-es Jack dugó csatlakozását is.

Az eszköz alá microSD kártyával tudjuk biztosítani az operációs rendszert mely jellemzően valamilyen Linux-disztribúció. Hivatalosan Raspbian névre hallgató, a Linux Debian optimalizált verziója ajánlott, mint operációs rendszer, de ezen kívül a Microsoft is bejelentette, hogy a 2-es verzióra már lehetővé teszi a Windows 10 IoT Core változatát. Ezektől függetlenül az operációs rendszerek tárháza kellően nagy ahhoz, hogy megtalálhassuk a számunkra legoptimálisabb rendszert, amit rá akarunk illeszteni a hardware-re. Ilyen például a Fedora alapú Pidora vagy Minibian. Utóbbi a Raspbian minimális operációs rendszere GUI nélkül[[3]](#footnote-3). --WIKIPÉDIA

Tápegység gyanánt ajánlott egy 5V-os, 2A-es microUSB-s tápot használni. Megfelelő betáplálás esetén a 4 darab USB, portonként képes leadni 1.2 Ampert is anélkül, hogy segéd tápot kéne biztosítani hozzá. Az eszköz fizikai paraméterei:

* Méret:85,60mm x 56,50mm (kiálló csatlakozókat leszámítva)
* Súly: 45gramm

Habár eredetileg oktatási célokra tervezték az eszközt, rengeteg területen bevált már, mint céleszköz. Miután egy operációs gépről beszélünk, szinte bármilyen kisebb-nagyobb projektet lehet rá illeszteni, fejleszteni, már amit az ARM processzor biztosítani tud. Meggyőződésem a határ csak az adott felhasználó fantáziájától függ. Találkoztam már az eszközzel, mint torrent szerverrel, meteorológiai állomással, otthoni média központtal, de olvastam már terveket róla, mint okos ház szíveként is be lehetne vetni, akár. Utóbbi ötletnek semmi akadályát nem látom, bár személy szerint az okos ház projektet már a Raspberry Pi nagy testvérére a Raspberry PI 3-ra tudnám elképzelni. De nem kell messze menni, akár egy sima asztali gépként is lehet alkalmazni melyen, böngészhetünk az interneten. Mondjuk nagy elvárásokat nem szabad alátámasztani a kis számítógépnek. Számítási teljesítménye hozzávetőleg egy 300MHz-es Pentium II-es gépnek felel meg összességében.



1. ábra Raspberry PI 2 B

## UART

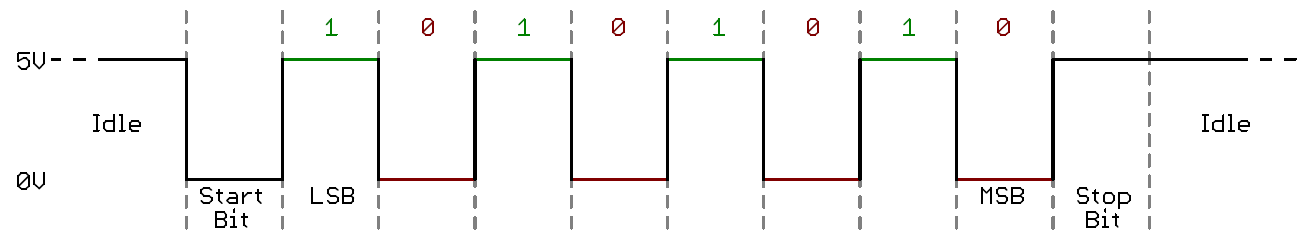
A program kommunikációja az OSI modell legalsó szintjén, a fizikai szinten UART-on valósult meg. Az UART jelentése „Universal Asynchronous Receiver/Transmitter” az az univerzális aszinkron adó –és vevő. Az UART a mikrokontrollereknek, SoC-nek olyan perifériája, amely lehetővé teszi az adatok fogadását és adását aszinkron módon. Az adatok küldése és fogadása soros porton történik. Így a bitek egyesével, egymás után shiftelve kerülnek elküldésre a vonalon. A bitek jelszintjei TTL (tranzisztor- tranzisztor-logika) szintnek felelnek meg. A logikai 0 az 0V vagy GND, míg a logikai 1, az a tápfeszültség szintje, mely jellemzően vagy 5 V vagy 3,3V. Összesen 3 vezetékre van csak szükség a kommunikáció biztosításához (Tx, Rx és GND). A fizikai kiépítés során ügyelni kell arra, hogy az adó Tx-e (transzmit-je) a vevő Rx-vel (Request-jével) legyen összekötve és fordítva, különben nem fog működni a kommunikáció. Ahogy a neve is mutatja, aszinkron módon történik a kommunikáció, tehát nincs közös órajel adó és vevő között. Ez okból kifolyólag előre kell definiálni mind az adóban, mind a vevőben a közös Baud rate-t, vagyis a kommunikáció sebességét. UART használat előtt a Baud rate-n kívül pár paramétert mindig előre be kell állítani, annak érdekében, hogy biztosítani tudjuk a megfelelő kommunikációt eszközeink között. Ezen paraméterek a következők:

* Baud rate: A baud rate egy mértékegység nélküli szám, amely megmondja, hogy 1 másodperc alatt hány jel változás történt. Ha a Baud rate 9600 akkor másodpercenként 9600 bitet továbbítunk a vonalon. Meghatározott értékek lehetnek csak a Baud ratek úgy, mint 9600, 19200, 38400, 56000 és 115200 attól függően az adott eszköz mennyit tud biztosítani.
* Adatbitek száma: Az adatbitek alatt azokat a biteket értjük melyek a start és a paritás bit között található, már ha használunk paritás bitet. Ha nem akkor értelemszerűen a start és stop között található bitek az adatbitek. Az adatbit 5,6,7 vagy 8 bit lehet. Általában 7 vagy 8 bitet szoktak használni a kommunikációra miután az általános ASCII tábla 7 bites. Így egy keret küldése során egy karaktert tudunk továbbítani.
* Paritás: A paritást az adatbitek visszaellenőrzésére használták még régebben, bár újabban erre a célra már inkább crc[[4]](#footnote-4)-t használnak. Ettől függetlenül még a mai napig is szokták használni a paritást. A paritás használata opcionális. Amennyiben szeretnék használni, választhatunk, hogy páros vagy páratlan paritást szeretnénk alkalmazni. Páros paritás során a paritás bit 0, ha az adatbitek közül az „1”-sek száma páros, különben 1 a paritás. Páratlan paritás esetén a paritás bit akkor 0, ha az adatban lévő „1”-sek száma páratlan, viszont ha páros, akkor 1 a paritás bit.
* Stopbit: Az adatküldés lezárását stopbittel jelezhetjük. Ilyenkor a vonal visszakerül magasba. A stopbit száma lehet 1,2 vagy 1.5. A stopbit száma meghatározza, hogy mekkora szünetre van szükség két adás között.

Ha nincs kommunikáció a vonalon, akkor állandóan magas állapotba van a vonal. Ha valamelyik eszköz kommunikációt szeretne kezdeményezni a vonalon, egy bit időre (START bit) lehúzza a vonalat a földre. START bit után jöhetnek a hasznos adatok melyet végül minimum egy STOP bit zár. Az UART kommunikációnál az adatbitek közül mindig a legkisebb értékű bitet (LSB) küldjük ki a vonalra elsőként majd legvégül a legnagyobb értékű bittel zárjuk a sort (MSB).

Általános jelölés módja egy keretnek soros port alkalmazásakor 8N1 vagy 7N2, ahol az első szám az adatbitek számát, a betű a paritást, míg az utolsó szám a stop bit számát adja meg.

Fontos tudni, hogy direktbe nem lehet akármilyen adatátviteli szabványt (RS-232, RS-485) rákötni az UART-ra. Gondoskodnunk kell a korrekt szintillesztésről. Megfelelő átalakító használata nélkül visszafordíthatatlan kárt tudunk tenni a hardware-ben. Az UART logikai jelszintje a következő képen látható.



2. ábra UART keretcsomag

## Rendszer Inicializálás

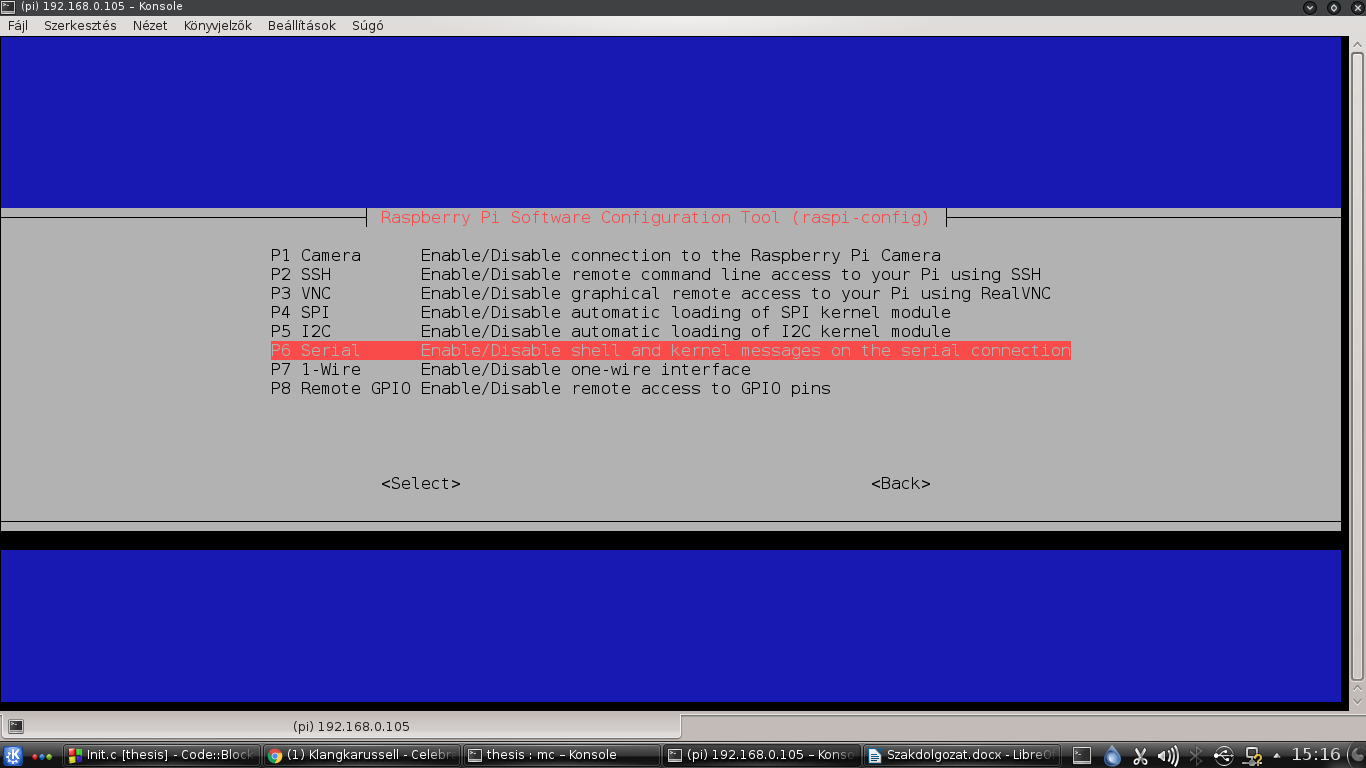
Operációs rendszer gyanánt egy Minibian-ra esett a választásom, egy grafikus felület nélküli, optimalizált Linux disztribúcióra. Ebből adódóan sikeres bootolás után, csak egy terminál ablak fogadott, mint kezelőfelület. Grafikus felület nélkül az egész rendszer elfért egy csupán 2 Gigabyte-os microSD kártyán is, ezzel is spórolva a költségeken. Csupán terminálból dolgozni megnehezítette volna a dolgomat így egy pár program feltelepítésre került.

Talán a legfontosabb a Midnight Commander (továbbiakban MC) mely egy egyszerű fájlkezelő program. Kinézetre egy az egyben régi Windows-os Norton Commander-re emlékeztet. Használata rendkívül egyszerű, már csak azért is, mert rendelkezik beépített szövegszerkesztővel (MCedit) és így nem kellett bajlódni a Linux alapértelmezett Vi nevezetű szövegszerkesztőjével. Előbbi segítségével tudtam elkészíteni a forráskódhoz tartozó fordító programot, a Makefile-t. MC-n kívül még Git verziókezelő és Valgrind debugger program került telepítésre.

Sikeres operációs rendszer telepítése során a rendszer inicializálása következett soron. Kettő nagyon fontos beállítást kellett alkalmazni a rendszeren. Először is engedélyezni kellett a rendszer soros port driver-ét. Alapértelmezett állapotban minden driver tiltva van, a felhasználónak kell beállítani azt, amire épp szüksége van. Ahhoz hogy a rendszer driverbeállító menüjébe be tudjunk könnyedén lépni a következő parancsot kell kiadni.

*$ sudo raspi-config*

A sudo parancsot csak akkor kell használni, ha nem rootként vagyunk bejelentkezve. Majd a felugró kék felületen, az „*Interfacing Options*”-be belépve a „*Serial*”-t kell engedélyezni.



. ábra Soros port hardware-s engedélyezése

Ezen kívül még az SSH[[5]](#footnote-5) is itt lett engedélyezve, hogy továbbiakban vezeték nélkül távolról is el tudjam érni az eszközt.

Sikeres beállítás és mentés után már csak egy helyen kellett változtatást végrehajtani a rendszeren. A \boot\ mappában található egy cmd.txt fájl. Ennek a tartalmából ki kell törölni a következő részt: *„console=ttyAMA0, 115200”.* Különben nem lehet tudni paraméterezni se a soros port elérési útvonalát, se a Baud ratet. Ahhoz hogy ezt érvényre juttassuk, a rendszernek mindenképp szüksége van egy teljes újraindításra.

Mivel az eszköz nem rendelkezik beépített wifi egységgel, így kezdetben csak Ethernet kábellel lehet csatlakozni a hálózatra. Szerencsére USB-s wifi sticket támogatja a Raspberry, így az adapter bedugása után automatikusan installálja a rendszer. Sikeres drivereztetés után a wifit még szoftveresen külön engedélyeztetni kell. Ahhoz hogy fel tudjunk csatlakozni az eszközzel a hálózatra, az /etc/network/interfaces táblát kell megfelelően, szerkeszteni. Fontos hogy ezt az állományt csak rendszergazda módban tudjuk módosítani és menteni mivel ez egy rendszerfájl. Ebben a fájlban tudjuk beállítani, hogy statikus vagy dinamikus módon kapjon címet a megnevezett routertől az eszköz, valamint az adott router titkosításhoz tartozó jelszót is. Beállítás után hogy érvényre jutassuk a módosításokat az eszközön mindenképp a következő parancsot meg kell adni a rendszernek:

*$sudo /etc/init.d/networking restart*

A parancs után a rendszer egy „OK” válasszal fogja nyugtázni a beállítás sikerességét. Más beállítással nem kell foglalkozni, már ami a hálózati csatlakozást biztosítja.

Napjainkban elengedhetetlen a megfelelő biztonsági szint beállítása azokon az eszközökön melyek kilátnak az Internetre. Így volt ez a Raspberry PI-nél is. A biztonságos kapcsolat létrehozása érdekében az SSH beállítása elengedhetetlen volt. Az SSH tulajdonképpen egy titkosított protokoll melynek segítségével akár távolról is hozzá lehet férni az eszközhöz, egy titkosított hálózati kapcsolaton keresztül. Nem csak hozzáférést, de akár adat másolást is biztosít, titkosított csatornán.

Attól hogy titkosított csatornán kommunikálunk SSH esetén még nem jelenti azt, hogy nincs szükség további beállításokra. Mivel a Raspberry szolgált ebben az esetben szerver gyanánt így az /etc/ssh/sshd\_config táblában kellett további beállításokat elvégezni. Alapértelmezett állapotban az SSH mindig a 22-es porton hallgatózik. A portnak így más számot állítottam be ezzel is védekezve, ha külső támadás érne. Továbbá az SSH használata során van lehetőség RSA[[6]](#footnote-6) kulcsokkal való azonosításra is. „Minden ssh-t használó gépnek van egy host-azonosító RSA kulcsa (default 1024 bit). A szerver gépen az sshd daemon ezen kívül generál egy szerver RSA kulcsot is (default 768 bit), amelyet óránként frissít és amit soha nem tárol a merevlemezen.” RSA kulcs használata esetén, a felhasználónak nem kell bonyolult jelszó beírásával bajlódnia, viszont a 768 bites titkosítás már feltörésre kerül. Bár az 1024 bites RSA kulcsot még nem törték fel, mégse alkalmaztam ezt a lehetőséget, helyette egy általam ismert erős jelszó került beállításra, amit csak én ismerek.

Egy ismerősömnek köszönhetően tudtam adni az eszköznek DynDNS[[7]](#footnote-7) címet is. A gyökérkönyvtárban található egy shell script mely egy Linux ütemező daemonnal fél óránként meghívásra kerül és így biztosítva van a DNS név az eszközhöz. Magát a DNS címet az ismerősöm biztosította számomra. Ezáltal bármikor, otthoni hálózaton kívülről is tudtam írni és tesztelni a programot. Továbbá, ennek segítségével hozzáfértem tesztelési eredményekhez is, mely a későbbiekben új, távlati célokat adott a projektnek.

# Segédprogramok

## Make

Szakdolgozatom megírása során fontos szempontnak tartottam, hogy ne csak egy fajta Linux alapú rendszeren lehessen használni az általam tervezett programot, hanem az összes olyan eszközön, melyen valamilyen Linux disztribúció van. Magát a lefordított bináris programot értelemszerűen nem lehet csak úgy másolgatni egyik eszközről a másikra, mert minden egyes eszköz más-más módon lett megtervezve. Egyes eszközöknél a hardware implementáció az mely nagyon eltérhet, másoknál driver beállítások lehetnek különbözőek, de akár még a könyvtárak verziója is más lehet, ami majd problémát okozhatna.

Ettől függetlenül még, ha tesztelés során találtam valami hibát a Raspberry Pi-n és azt egyből tudtam javítani Mceditbe akkor nem szerettem volna újra fordítani a teljes programot és másolgatni a kész binárist vissza a cél eszközre csak a módosított fájl vagy fájlokat. E célból a projektben létrehoztam egy Makefile-t. Magát a Makefile-t a make-kel, egy Linux alapú parancssori programmal tudtam fordítani. Fő célja leegyszerűsíteni és automatizálni a fordítást. Tételezzük fel, találtunk egy hibát, egyetlen egy forrásfájlban vagy csak szimplán módosítottuk, akkor alapesetben ilyenkor újra fordítaná az egész programot a main-től kezdve az utolsó forrásfájlig bármelyik fejlesztő környezet, mint például ahogy a Code::blocks is. A make ezt a procedúrát hidalja át azzal a technikával, hogy figyeli melyik fájl vagy fájlok módosult(ak) és csak az(oka)t fordítja le ismét. Így megspóroljuk azt a munkát és időt, amit egy teljes fordításkor használunk fel. Nagyobb programoknál jön ki igazán az előnye, amikor a projektben 10-11, esetleg még több forrásfájl is lehet.

Annyi feltétele van, hogy ahol található a projekt, abban a mappában létre kell hozni egy fájlt, amibe leírjuk a make fordítási metódusait. Célszerű Makefile nevet adni neki, mert akkor nem kell feleslegesen make -f <file> kapcsolót és fájl nevet használnunk. Makefile név esetén elég csak egy make parancsot kiadni és máris ellenőrzi a forráskódokat, hogy történt-e módosítás valamelyik fájlon és ha változást talál akkor azt fordítja is egyből.

Fordítási elvén kívül még az is előnyére írható hogy roppant egyszerű megírni egy Makefile-t. Egy Makefile sémája a következő:

cél : függőségek

parancs(ok)

Fontos hogy a parancs(okat) ne szóköz, hanem egy tabulátor előzze meg! Egy Makefile-ban lehetőség van változók létrehozására, ezzel is egyszerűbbé téve a megírási folyamatot.

CC=gcc

A változókra való hivatkozás a következő módon történik.

$(CC)

Továbbá a make rendelkezik automatikus változókkal is mely jelentősen leegyszerűsíti a Makefile megírását és használatát. Néhány példa az automatikus változókra.

* $\* Teljes forrásfájl neve kiterjesztés nélkül
* $< out-of-date forrásfájl neve kiterjesztéssel
* $. forrásfájl teljes neve elérési útvonal nélkül
* $&. forrásfájl neve elérési útvonal és kiterjesztés nélkül
* $: csak az elérési útvonal
* $@ Teljes aktuális cél neve

Ezen felül a make rendelkezik saját függvényekkel is.

$(subst from,to, text)

$(subst oo,OO,book on the roof) → bOOk on the rOOf

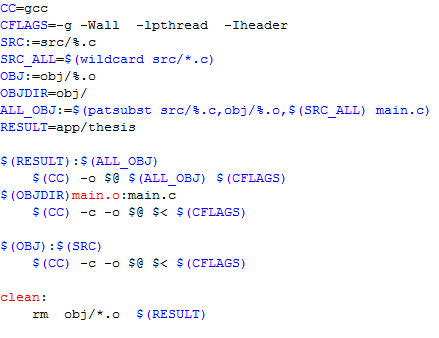
A függvény a megadott string mintában kicseréli azokat a karaktereket ahol két darab o van egymás mellett két darab nagy O-ra.

$(patsubst pattern,replacement,text)

$(patsubst %.c,%.o,counting.c reading.c) → counting.o, reading.o

Minden olyan fájl, mely .c-re végződik, kicseréli .o végűre. Utóbbi függvénnyel nem kell felsorolni a projektben megtalálható összes forrásfájlt, hanem így automatizálva mindig az összesre megcsinálja a hozzá tartozó objekt fájlt. Ezáltal ha új forrásfájl kerül be a projektbe a make észre fogja venni és fordítani fogja az összes többivel együtt.

Az általam készített Makefile tesztelve lett mind Raspberry-n mind otthoni munkalaptopon, de még virtuális környezetben is, ahol mind tökéletesen le tudott fordulni és eltudta készíteni a futtatható binárist. A programhoz írt Makefile a következő:



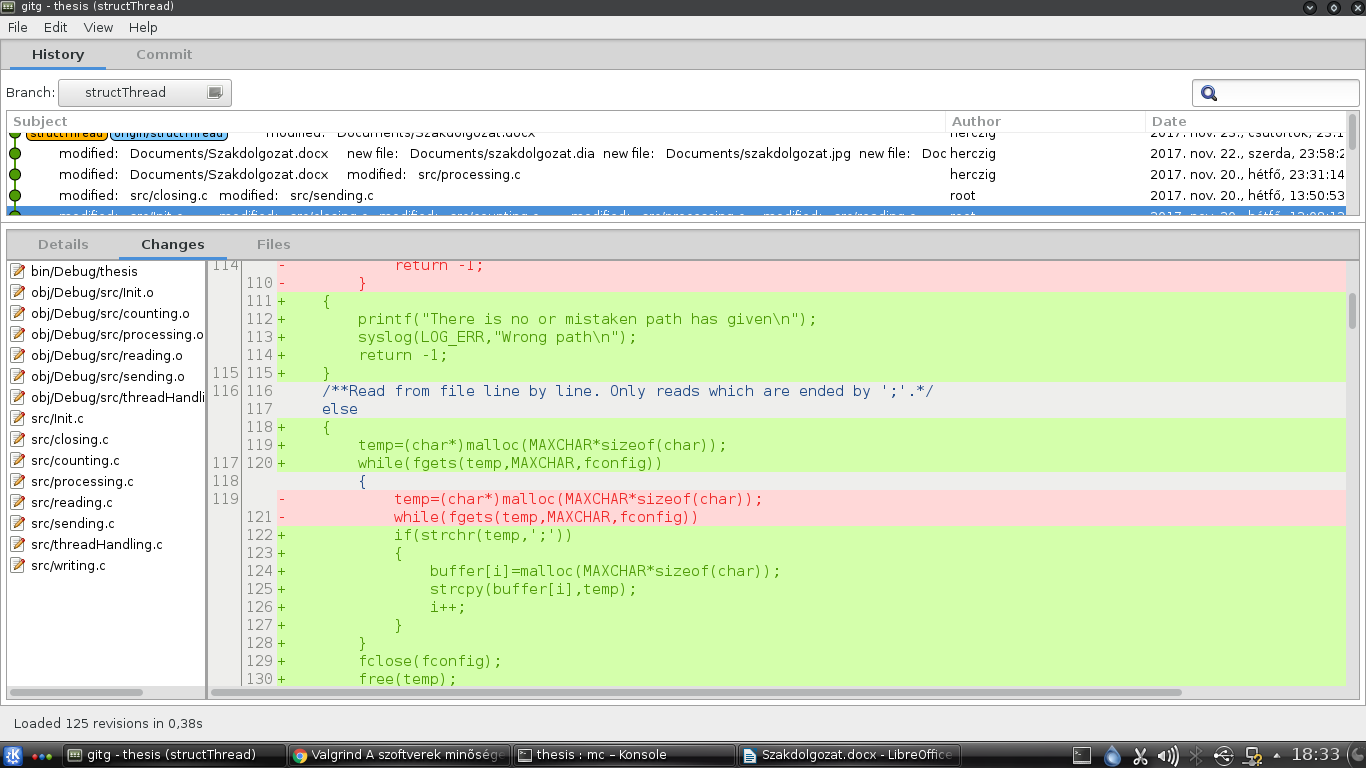
<https://www.gnu.org/software/make/manual/html_node/Text-Functions.html>

## Git

A Git egy verziókezelő szoftver mely arra szolgál, hogy kisebb-nagyobb projektek esetén, nyomon lehessen követni a projektek állapotát, a forráskódok tartalmát és verzióját. Ez egy nyílt forráskódú ingyenes szoftver melyet anno Linus Torvalds fejlesztett ki. Git kezelésről rengeteg magyar és angol forrás található az Interneten, ezért részletesen nem tervezek belemenni a program kifejtésébe, működési elvébe.

Nagyvonalakban arról szól a Git, ha új állomány került be a projektbe, akkor a „git add <file>” paranccsal megmondjuk a Git-nek, melyek az új fájlok, amelyeket szeretnénk hozzáadni a projekthez. Majd a „git commit” parancsot kiadva a helyi könyvtárról csinál egy helyi adatbázist a .git könyvtárba. Lokális adatbázist aztán a „git push origin <branch>” paranccsal tudjuk feltölteni a szerverre, amit majd később bárki elérhet, aki arra jogosult. Nagy előnye a sebességében valamint a fa szerkezetében rejlik. Nem szükséges a fő szálon dolgoznunk végig. Ha eszünkbe jut bármiféle újítás, amit nem akarunk a fő ágon vinni, mert csak „kísérletezgetünk” az új ötlettel, nyugodtan létrehozhatunk új ágat (branch) is. Ha az új ágon történt fejlesztés jónak tűnik, akkor az új ágat össze lehet „merge”-lni[[8]](#footnote-8) a fő ággal és onnantól kezdve más is láthatja az újítást.

Projekt kezdéskor mindig inicializálni kell egy .git könyvtárat a „git init” paranccsal, ami a munka könyvtárunk lesz. További előnye még, hogy ezeket a műveleteket mind tudja biztosítani titkosított csatornán is (SSH) ezzel is növelve az adatbiztonságot. Használati szinten ugyanolyan parancssoros program, mint a make, viszont sok grafikus alkalmazást fejlesztettek mellyel vissza lehet nézni a history-ban ki, mikor, mit csinált az adott projekten.

Linuxon eddig, amiket használtam a gitk és gitg program. Komolyabb fejlesztésekhez elengedhetetlen program. Többször is nagy hasznát vettem, hogy régi kódot kerestem vissza, mert az új kód nem volt se hatékonyabb se jobb a réginél.

4. ábra Gitg

## Code::blocks

A munka érdemi része nem a Raspberry-n készült, hanem laptopon egy programmal, a Code:blocks fejlesztő környezettel. Ez egy ingyenes program melyben rengeteg hasznos plugin, és eszköz található. A programnak köszönhetően, egyszerűbb és átláthatóbb a programozás, melynek köszönhetően könnyebb volt modulárissá írni az egész projektet. Jelentősen megkönnyítette a munkámat az automatikus szövegillesztő bonyolultabb változó neveknél, függvényeknél vagy egyes headerek beillesztésénél. Így rengeteg időt és energiát tudtam megtakarítani.

Program fordítás során bármilyen szintaktikai hibát automatikusan jelzett. Ha például egy pontos vessző lemaradt egy parancs végéről az adott sorban egy piros körrel jelzi a felhasználónak a hiba helyét és annak okát is, ezzel is megkönnyítve a hibakeresést is. Maga a program mind az objekt, mind a futtatható binárist elkészíti nekünk a projekt könyvtárunkba. Lehetővé teszi különböző fordítók használatát valamint a fordítókhoz különféle kapcsolók illesztését is.

## Valgrind

Programom tesztelése során sokszor sikerült olyan hibákat csinálnom melyeket néha még a Code::blocks beépített debuggerével se sikerült egyértelműen megtalálnom. Ezek jellemzően a dinamikus memória kezelésből fakadtak. Vagy olyan helyre akartam írni amit nem foglaltam le, vagy olyan memória részt szerettem volna felszabadítani amit le se foglaltam. Esetleg olyan területet szerettem volna felszabadítani, amit már felszabadítottam korábban. Sok fórumot olvasva, az egyik hozzászólásnál találkoztam a témában említett szóval. Maga a komment nagyon sok olvasó által maximális értékelést kapott, így utánanéztem miért is értékelték olyan sokan maximumra a programot. De mi is ez a Valgrind?

„A Valgrind egy nyílt forrású segédeszköz a memóriakezelési hibák felderítésére. Az x86-os processzora fordított programokba figyeli a memóriaszivárgást és a helytelen elérést. ...A Valgrind valójában egy dinamikusan összeépített könyvtár, ez a parancsfájl végzi az összeépítést.”

A Valgrinddal a következő hibákat tudjuk felderíteni:

* olyan memóriaterületet olvasása, amely nem kapott kezdeti értéket;
* felszabadított területet olvasása, illetve írása;
* túlcímzett terület olvasása, illetve írása;
* a verem nem megfelelő területeinek olvasása, illetve írása;
* memóriaszivárgás;
* rosszul használt malloc/new/new[] és free/delete/delete[] páros;

-a POSIX pthread API helytelen használata.

Az utolsó előtti pont rávilágít, hogy nem csak C, de C++ nyelven írt programok ellenőrzésére is kiváló program a Valgrind.

//http://epa.oszk.hu/02800/02894/00047/pdf/EPA02894\_linuxvilag\_41\_29\_31.pdf

## Valgrind Használata

Használata roppant egyszerű. Miután terminál parancsos program, terminálba először a valgrindot írjuk majd utána azt a lefordított programot, melyet vizsgálni szeretnénk. Először üdvözöl minket, majd végig futtatja a programunkat cím és memória ellenőrzésekkel. Mondhatni olyan, mint egy emulátor. Rengeteg hasznos funkciója van, de munkám során csak a memcheck részével foglalkoztam az ellenőrzés végett. Ahol hibát talál, memória címre és sorszámra kiírja az adott fájlban milyen jellegű hibát talált. Ezek lehetnek írási vagy olvasási hibák egyaránt. Ha a program végére ért, kiértékeli mind a stack-et mind a heap-et. Statisztikaszerűen kiírja mennyi byte lett lefoglalva, mennyi lett felszabadítva a stack-be és a heap-be egyaránt. Ezen felül összesíti az összes hibát.

## Dia

Szokták mondani, ha az ember tud csinálni egy jó folyamatábrát, akkor már a program megírása gyerekjáték. A programtervezéshez mindenképpen szükségem volt egy megfelelő folyamatábrára.

Erre a célra egy nagyon egyszerű programot használtam, a Dia diagramkészítőt. A program maga egy ingyenes szoftver, ami az összes operációs rendszeren elérhető. A programról első ránézésre az embernek a Windows alatt ismert Paint program ugrik be, szerkesztő felülete végett. Ez egy olyan rajzoló program, melyben az összes geometria alakzat eszköztárból elérhető számunkra. Továbbá lehetőség van mindenféle alakzat saját kezű megrajzolására is. A geometriai formákon kívül vektorok kezelésére is van lehetőség a programban. A megrajzolt, megszerkesztett formákba lehetőségünk van szavakat írni, de akár írhatunk fölé, mellé, alá, effektíve mindenhova. Szinte korlátlan az eszköztára. Ebből kifolyólag talán a legideálisabb program hogy mindig a legmegfelelőbb folyamatábráját tudjuk elkészíteni az adott munkánkhoz.

Az elkészített munkalapot dia kiterjesztéssel tudjuk elmenteni, de parancssorból másodpercek alatt tudjuk konvertálni akár jpg-be akár pdf-be a dokumentumunkat.

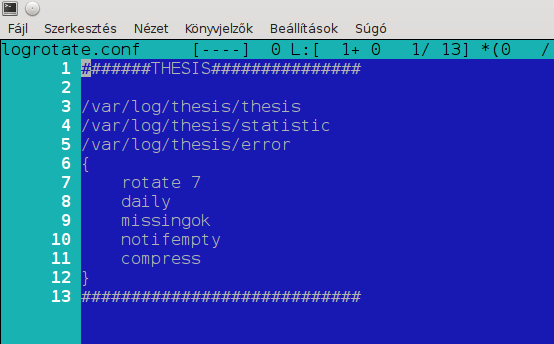
## Logrotate

A program működése során információkat közöl a felhasználóval. Ezen üzenetek két részre bonthatóak. Egyik része az üzeneteknek a lemért hőmérsékleti adatok, míg másik része az eszköz működési állapota. Utóbbi alatt olyan rendszerüzeneteket kell érteni, mely jelzi a felhasználónak a konfigurációs állomány megnyitásának és annak felolvasásának sikerességét, a soros port beállításának visszajelzését, vagy csak az adott szolga eszköz állapotát. Hiba esetén szintén jelezni kell a szoftvernek a felhasználó felé, hogy tudja, mi lehet a baj, mit kell javítani. Mind a kettő típusú üzenet kiíratásra kerül a standard kimenetre valamint a syslog[[9]](#footnote-9)-ba is. A program külön helyre naplóz rendszer szinten, amit nem tart karban a rendszer. Ezáltal állandóan írja az adatot addig, amíg el nem fogy a microSD kártyán a hely.

Erre a célra találták ki a logrotate-t mely egy olyan Linuxos program melynek segítségével rendszerüzeneteket hatékonyan lehet tömöríteni, rotálni, törölni vagy akár email-be küldeni beállított címzettnek. Magának a programnak van egy konfigurációs fájlja mely az /etc/ mappában található. A fájl neve logrotate.conf és ezen belül tudjuk beállítani, hogy az adott syslog szolgáltatás hogyan legyen kezelve. Elsőként a fájlban az adott fájlok elérési útját kell megadni amelyeket karban szeretnénk tartani. A felsorolást követően, kapcsos zárójelek között a következő főbb paramétereket tudjuk megadni.

* Művelet típusa: törlés, email küldés, tömörítés;
* Működés gyakorisága: óra, nap, hét, hónap;
* Mérethatár: ha beállított méretet eléri, elvégzi a beállított műveletet.

Természetesen van lehetőség többféle logfájl, különböző módon való kezelésére. Ilyenkor csak egymás alá kell felsorolni az adott egységeket és a hozzájuk tartozó paramétereket kapcsos zárójellel határolva. A projekt gyökérkönyvtárában található util mappában található egy etanol logrotate.conf fájl, amit a megfelelő helyre kell csak másolni és a szolgáltatás újraindításával máris kezeli a rendszer a beállításban található állományokat.



# Program

## Tervezés

Legelső lépésben mielőtt bármihez is neki kezdtem volna, végig kellett gondolnom hogyan is fog működni a programom. Mit kell tudnia és amit tudnia kell, azt hogyan kell tudnia. Célom az volt, hogy minél átláthatóbbá programot írjak, annyi modult írva amennyire szükség volt a könnyű áttekinthetőség véget. Az egyik legfontosabb szempont a tervezés során az összes hiba lehetőség felismerése és azoknak megfelelő lekezelése. Hiba lekezelés mellett, szempont volt továbbá, a felhasználó informálása az adott hibáról és a hibaelhárítás módjának megfelelő javaslása. Miután a feladat egy hőmérésgyűjtő rendszer kialakítása volt, szépen sorban fel kellett építeni a szoftvert.

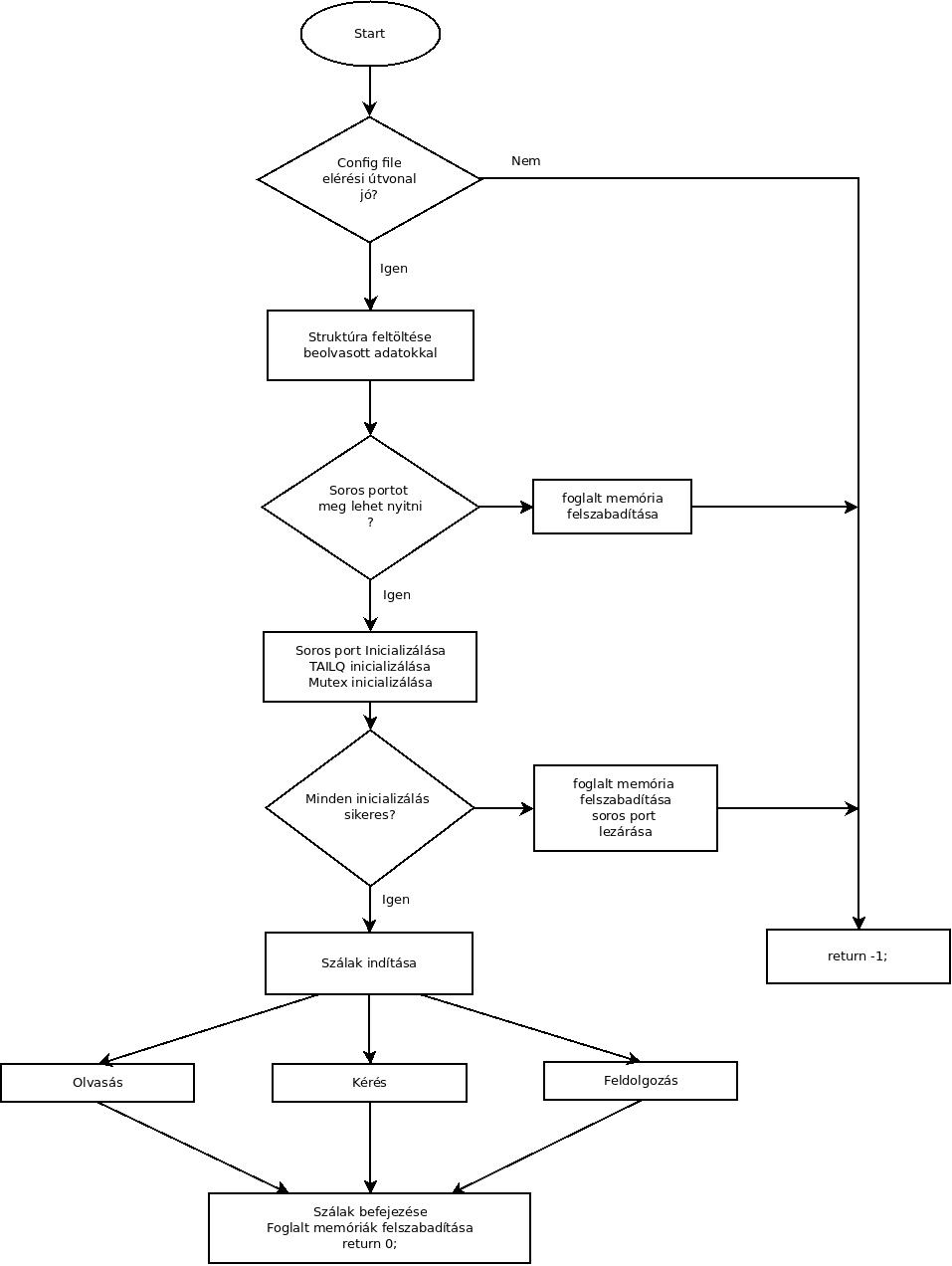
A programnak induláskor, legelső lépésben egy konfigurációs fájlt kell tudni megnyitni, amit a projekt gyökér mappáján belül lévő Util mappába helyeztem el. Sikeres megnyitás és felolvasás után, a kapott értékkel a soros port beállítása következett. A util mappában elhelyeztem még két darab etalon konfigurációs fájlt. Használatuk elengedhetetlen a megfelelő működés érdekében.

Konfigurációs állomány betöltése és a soros port megnyitása és megfelelő beállítása után, a FIFO és a mutex inicializálása következett. Ennek a két elemnek az inicializálását, mivel elég rövid és egyszerű, egy rövid függvénybe sikerült megvalósítani.

Sikeres előkészületek után a következő feladat, a szálak elindítása volt. Az olvasó, a kérést küldő és a feldolgozó szál indításánál figyelni kellett mindegyik szál becsatolására a főszálba. E nélkül hamar véget ért volna a program működése, hiszen akkor a főszál nem várta volna meg az alszálak befejezését és idő előtt kilépett volna. A szálakon belül nemcsak az olvasás és feldolgozás történik, hanem statisztikavezetés is,

Számolni kellett azzal a lehetőséggel, ha szignált kap a program a kerneltől vagy a felhasználótól, akkor hogyan történik a kilépés, mi lesz az ideiglenesen foglalt memóriával. Így a szálak indítása és becsatolása közé került a jel kezelés megvalósítása.

Végül, de nem utolsó sorban, ha a program befejezi a mérést és a feldolgozást, akkor a rendszertől lefoglalt memóriát vissza kellett adni a rendszernek. Ezen felül még a soros port visszaállítása se maradhatott el. Így a felhasznált memóriák mind felszabadításra kerültek és a soros port is eredeti állapotát kapta vissza.



. ábra Komplett folyamatábra

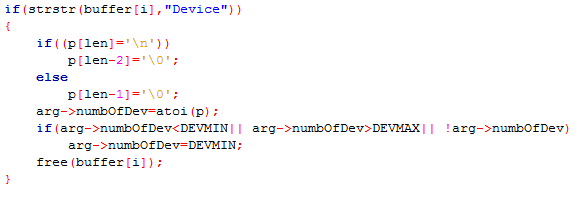
## Konfigurációs állomány beolvasása

Ahogy a folyamatábra is mutatja program indítása során először egy megadott konfigurációs állományt próbál megnyitni a program, amennyiben létezik vagy olvasható az állomány. A konfigurációs fájlba kell előre megadni a soros port paramétereit. Ezen felül még az eszközök számát, a futó hiszterézis delta értékét, a mozgóátlag tagszámát és a mérési idő gyakoriságát is, valamint a megadott eszközök címét, nevét és állapotát (aktív vagy passzív épp az eszköz). Amennyiben nincs jó helyen a konfigurációs fájl vagy nem olvasható, valahogy tudatni kell az információt a felhasználóval miért lépett ki a program. Ebből kifolyólag, mind a standard kimenetre, mind syslogba ki lett írva a hiba oka. A felhasználónak a standard kimenetre, nekem, fejlesztőnek pedig syslogba.

Sikeres állomány megnyitás után, először MAXCHAR[[10]](#footnote-10) byte memóriafoglalást kap egy ideiglenes mutató ahol MAXCHAR 128 byte-nak felel meg. Sikeres memória allokáció esetén a lefoglalt terület kezdőcímét kapja meg az ideiglenes mutató. Ezek után egy while ciklusban felolvasásra kerül az állomány tartalma. Az olvasás egészen EOF[[11]](#footnote-11)-ig tart. Olvasás során egy feltételt illesztettem a ciklusba, ha az aktuális sorban van pontosvessző karakter, akkor az adott sort egy pointerekből álló tömb i-edik elemébe másolja a program, melynek előtte dinamikusan foglaltam memóriát. Ebből adódóan fontos a fájl pontos kitöltése. Ahol hasznos adat van, azt a sort pontosvesszővel kell zárni, melyre figyelmeztet is a program. Így lehetőségem van a hasznos sorokból később egyenként dolgozni. A felolvasási sorrend kötött, abból a szempontból, hogy előrébb várja a program az eszközök számát, mint paramétert és később az eszközök nevét, státuszát, állapotát és a mérési idejüket hisz csak így tud megfelelően foglalni memóriát az utóbbi adatoknak a program. Végezetül az ideiglenes mutató által mutatott terület felszabadításra kerül.

Felolvasás után a pointer tömb nulladik elemétől elkezd a program vizsgálni szórészletekre. Első feltételként megvizsgálja, hogy az adott sorban létezik-e egyenlőség jel. Amennyiben igen, egy switch-case szerkezettel szórészletekre vizsgál a program az adott tömb elemben. Ha például megtalálható az „=” és egy adott minta string a pointer által mutatott sorban, akkor az egyenlőségjel utáni értéket hozzárendeli a struktúra adott eleméhez. Igyekeztem az összes paraméterre megfelelő szórészletet használni összehasonlítási alapként.

Jelen esetben az eszközök számához „Device” mintát adtam meg:



6. ábra Mintakeresés és adatátadás

Míg a konfigurációs állomány tartalma:

NumberOfDevice=6;

Jól láthatóan csak akkor kapja meg a beolvasott értéket, ha bizonyos feltételeknek megfelel, ezzel biztosítva, hogy ne lehessen akármilyen értéket beírni. Eszközök számánál maradva a beolvasott értéket csak akkor lehet átadni a struktúra elemének, ha a beolvasott érték egy minimum és egy maximum érték közé esik. Nulla nem lehet, hiszen jelen esetben annak nincs értelme, nulla darab eszközzel nem indul el a mérés. A minimum érték 1, a maximum érték jelen esetben 31. Mindkettő határérték makróként lett definiálva, nem pedig konstansként. Minden egyes kiértékelt elem után, a kiértékelés után a pointer tömb adott eleme felszabadításra kerül.

Mindenhol igaz ez a minimum-maximum feltétel kivétel a Baud rate-nél, hiszen ott megadott értékek közül kell valamelyiknek megfelelnie. Amennyiben hibás vagy egyáltalán nincs beolvasott érték, akkor sincs baj. Ebben az esetben mindegyik elem kap egy default értéket. Ez alól egyedül az eszközök száma kivétel. Ez esetben, ha nincs leírt szám, akkor nullát kap értékként és mínusz egyes hiba kóddal és hibaüzenettel kilép a program.

Sikeres adat feltöltés után az eszközök nevére, címére, állapotára és mérési idejére kerül a sor. Ennél a résznél már nem egyenlőség jelre vizsgál a program, hanem tabulátorra, így különböztetve meg a két részt. Ha van tabulátor, akkor a sor végétől indul a feldolgozás a lezáró nulla karakter megfelelő illesztése miatt. A tabulátorok száma irreleváns, de minimum egynek kell lennie két adat között. Az eszközök tulajdonságai egy másik struktúrába kerülnek eltárolásra. A sorrend itt elég kötött, a sornak elsőként címet, nevet, majd állapotot és végül hozzátartozó időt kell tartalmazni. Ebből mutat is példát a program amennyiben sikertelen a felolvasás, hátha eltévesztette a sorrendet a felhasználó.

Az eszközök feldolgozása visszamenőlegesen történik meg, ennek során az első érték az adott tömbből az eszköz mérési ideje lesz. Ezt az értéket az utolsó tabulátor előfordulásának helyével keresi meg. Megtalálás után ezt a címet egy segéd pointer kapja meg, ami egy címmel arrébb lesz léptetve. Léptetés után az eszköz struktúra idő eleme megkapja a segéd pointer által mutatott értéket. Amennyiben az atoi() függvény 0-t ad vissza akkor kap egy alapértelmezett 10 másodperces értéket. Idő érték adás után az eszköz állapota következik. Lehetőség van passzív eszközöket is megadni a rendszerben, mely későbbi indításkor átállítható aktív állapotúvá, így a rendszer majd megpróbál kommunikálni vele, de passzív állapotban lévő eszközt figyelmen kívül hagyja. Ez vagy 1 vagy 0 lehet. Az 1 az aktív a 0 a passzív állapotnak felel meg. Állapot után az eszköz neve következik. Az eszközök neve miatt még egy segéd pointerre volt szükségem, hogy tudjam, hol kezdődik, hol végződik az eszköznév. Mivel az eszközök nevét előre nem lehet tudni így a neveknek dinamikusan lett memória foglalva melyeknek a nagysága a két segédpointer távolsága plusz egy byte ami a lezáró nullának kellett. Befejezésként a címátadás történt meg.

Utolsó műveletként még a program megvizsgálja, hogy van-e két azonos című eszköz. Ha igen akkor azokat az eszközöket passzív állapotba helyezi és szól a rendszer a felhasználónak a címütközésről. Minden egyes feldolgozásnál ügyelni kellett, hogy a beolvasott terület mindig fel legyen szabadítva a kiértékelés után. Sikeres művelet esetén nulla, hiba esetén mínusz egy a visszatérési értéke a függvénynek és syslogban valamint standard kimenetet jelez a rendszer a hiba okáról.

## Soros port

Sikeres konfiguráció kiolvasása esetén a soros port beállítása következett. Erre a célra C-ben a termios.h header tökéletes segítséget nyújt. Soros port beállítása esetén, legelső lépésben meg kell tudnia nyitni a soros portot a programnak. Ha meg tudja nyitni, akkor egy nem negatív számot ad vissza a rendszer, mint soros port fájl leíróját. Ezzel a számmal tudjuk meghivatkozni a soros portot íráshoz és –vagy olvasásához. Jó tudni, hogy a Linux 0-tól 2-ig lefoglalja a fájlleírókat boot-olás után. A 0 az a standard kimenetnek, az 1 a standard bemenetnek, míg a 2 a standard hibakimenetnek felel meg. Ezek nem fixek, a fejlesztőnek lehetősége van a soros portot beállítani pl 1-re. csak ha végzett a programja, akkor illik visszaállítani a fájlleírót. Soros port elérési útjának 4 féle opciót adtam meg,

* /dev/USB0;
* /dev/AMA0;
* /dev/ttyS0;
* /dev/ttyS1.

Az USB0-ra a tesztelés miatt volt szükség, míg az AMA0 az a Raspberry gyári soros port kezelője. A ttyS0-ra és ttyS1-re azért volt szükség, hogy ha más Linux eszközre kerülne fel a program mindenképp meg tudja nyitni a portot. Hiba esetén syslog-ba és standard kimenetre, a képernyőre is kiírja a hiba okát.

Sikeres soros port megnyitás után a soros port paraméterezése következett. A rendszernek mindig van valamilyen soros port beállítása, amelyről nem sokat lehet tudni de használat után vissza kell állítani, ha a program befejezte a működését. Ezért két termios struktúra deklarálására volt szükség. Egy, amelyben el lett mentve az aktuális állapot (old\_term), egy pedig az új paraméterek beállításainak (term). Soros portot két féle üzemmódban lehet használni, kanonikus illetve nem kanonikus módban. Kanonikus feldolgozás esetén a beolvasás soronként történik meg, tehát mindig új sor karakterig vagy EOF-ig olvas. Ellentétben a kanonikussal, a nem kanonikus üzemmód során általunk megadott karakter számig olvas a soros port. Nekem utóbbira volt szükségem, hiszen adatok beolvasásánál egyáltalán nem várhattam új sor karakterig vagy sor vége karakterig, ha érkezett legalább egy karakter azt egyből be kellett olvasni, hogy a rendszer gyorsan tudjon dolgozni. Eredeti állapot lementése után, a soros port különböző módjainak beállítása következett. Itt lehetőség van bemeneti, kimeneti, helyi és vezérlő módok valamint vezérlőkarakterek beállítására. Fontos a megfelelő beállítás, hiszen ezen múlik a kommunikáció a mikrokontrollerek és központi vezérlő között. Vezérlő flag-eknek a következő értékek lettek bit műveletekkel beállítva:

* CS8;
* CLOCAL;
* CREAD.

A CS8 a 8 adatbitet és egy stop bitet jelenti, a CLOCAL a modem nélküli hálózati kapcsolat beállítását biztosítja és a CREAD paraméter biztosítja, hogy nem csak írásra, hanem olvasásra is használni akarjuk a soros portot.

Mivel az olvasás ellenőrzését crc módszerrel oldottam meg, bemeneti flag-eknek a paritás vizsgálat ignorálása lett csak beállítva.

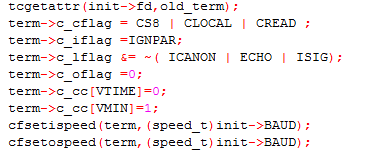
Lokális módnál ki lettek maszkolva a következő paramaméterek:

* ICANON;
* ECHO;
* ISIG.

Az „ICANON” maszkolásával érjük el a nem kanonikus üzemmódot. Az „ECHO” a beviteli karakterek visszajelzésének tiltása az író oldalnak, az „ISIG” pedig a SIGINTR, SIGUSP, SIGDSUSP és SIGQUIT jelek tiltását teszi lehetővé.

Vezérlő karaktereknek a c\_cc[TIME] és a c\_cc[MIN] kapott értéket. Az előbbi 0 értéket kapott, utóbb pedig egyet. Ennek során csak a MIN értékét használjuk, mellyel a beolvasott karakterek számát definiáljuk. Ahány karakter érték van megadva, annyi ideig olvas a soros port majd újra kezdi az olvasást.

Végére a sebesség megadásra maradt. A ki és bemeneti sebességnek a konfigurációs fájlból felolvasott érték adódik át vagy 9600-at kap, mint alapértelmezett érték, ha nem volt megadva semmi se a konfigurációs fájlban.



7. ábra Soros port beállítás

A beállítások végeztével a ki - és bemeneti puffer ürítésre kerül mielőtt, érvényre jutna a paraméterek beállításának mentése. Ellenkező esetben eddig foglalt memória területek felszabadításra kerülnek és hiba üzenettel kilép a program.

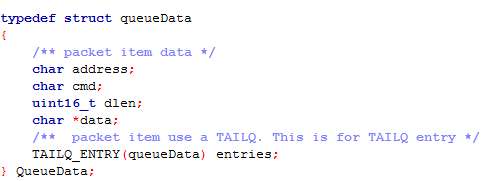
## TAILQ és Mutex inicializálás

A rendszer teljes inicializálásához már csak a TAILQ-t és a mutexet előkészítése maradt hátra. Előbbi segítségével a láncolt listát tudtam egyszerű megvalósítani, míg utóbbi segítségével tudtam biztosítani a megfelelő adatátadást szálak között.

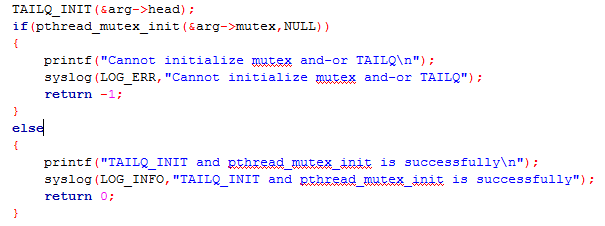
A mutex a „mutual exclusive” rövidítése mely a kölcsönös kizárást jelenti. Mutex használatára azért van szükség, hogy a FIFO kezelését egyszerre csak egy szál végezhesse. Ha az olvasó szál lefoglalja a mutexet, addig a feldolgozó szál várni kényszerül. Így nincs meg a veszélye, hogy egy csomagot két szál használna egyszerre, egy időbe.

A C programnyelvben a BSD[[12]](#footnote-12) jóvoltából léteznek előre megírt FIFO megvalósítások. Így nem kellett külön megírni a láncolt lista működését. Ezen megoldások, a <sys/queue.h> headerben találhatóak meg. Itt található meg maga a TAILQ FIFO is. A TAILQ, mint ahogy a többi eszköz is az előbb említett headerben mind makrókkal lettek megvalósítva. Mivel makrókként vannak megvalósítva, ezért az előfordító a behelyettesítésüket még a fordítási folyamat előtt elvégzi, így azok kiszámítása nem rontja a kész program teljesítményét.

A TAILQ egy olyan speciális FIFO melyben az elemek struktúrákként vannak számon tartva. A TAILQ listakezelő változója a TAILQ\_HEAD makró. Ezen keresztül érhetjük el lista bármelyik elemét. A struktúra elemek tartalmaznak egy pár mutatót, egyik az első elemre mutat, míg a másik az utolsó elemre a TAILQ-ban. Mivel többszörös láncolt listáról beszélünk, így tetszőleges elemet kivehetünk a sorból, anélkül hogy végig mennénk az egész soron. Új elem hozzáadására lehetőség van egy meglévő elem előtt vagy után, illetve a TAILQ első és utolsó részeként is hozzálehet adni. Ha egy új elemet első elemként adjuk hozzá a meglévő sorhoz, értelemszerűen a meglévő sor eggyel tovább csúszik. A TAILQ fontos része még a TAILQ\_ENTRY mely az adattároló listakezelő által használt mező típusa, leírója.



8. ábra TAILQ\_ENTRY definiálása

Utolsó lépésben az inicializálás során a TAILQ\_INIT makrót és a pthread\_mutex\_init() függvényt használtam fel egy függvényben.

9. ábra TAILQ és pthread\_mutex inicializálás

## Szálkezelés

A szálak egy folyamaton belül egymástól külön ütemezhető, párhuzamosan futó utasítássorozatok. Míg a folyamatok között csak a végrehajtandó kód a közös, a szálak ugyanabban a címtartományban futnak. Ellentétben a folyamatokkal, a szálak osztozkodnak az erőforrásokon. Ilyenek az időzítők, a kód- és adatterületek, a fájlleírók valamint a jelzéselrendezések is a folyamaton belül. Egy szál maximum addig fut, ameddig a főszál, vagyis ameddig a folyamat be nem fejezi a futását, de akár előbb is vége lehet egy szál futásának. Viszont a vermen nem osztozkodnak a szálak, mindegyik szál rendelkezik a saját maga vermével.

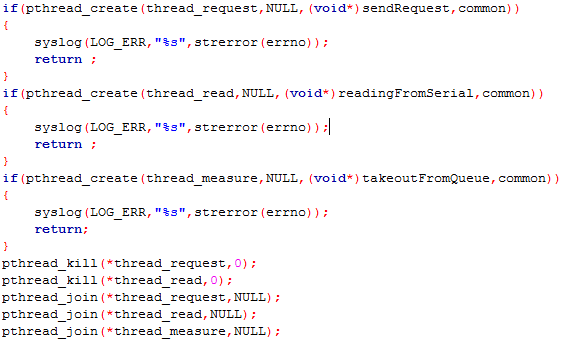
A szálkezeléshez a legelterjedtebb szálkönyvtárat használtam a POSIX féle pthread.h headert. A három szál melyeket létrehoztam a programban egy író, egy olvasó és egy feldolgozó szál. A szálaknak a létrehozását, becsatolását és jelzés kezelését egy külön modulba írtam meg.

Szál létrehozására a következő függvényt alkalmaztam.

**pthread\_create(pthread\_t \*thread\_id***,***const pthread\_attr\_t \*attr, void\*(\*start\_routine) (void \*), void \*arg);**

Első paramétere a létrehozandó szál azonosítója, második paramétere a létrehozandó szál attribútumát hivatott beállítani. Attribútum beállításnál tudjuk elérni, hogy a szál becsatolható legyen vagy ne a főszálba és még jó pár, más egyéb száljellemző adható meg a létrehozandó szálnak. Ha NULL-t kap paraméternek, akkor alapértelmezett attribútumot állít a szálnak. Harmadik paraméternek egy függvény nevet vár, amit majd futtatni szeretnénk mellékszálként. Nem feltétlenül kell void típusú függvényt írni. Ha így döntünk, mindenképp kell cast[[13]](#footnote-13)-olni a függvényt paraméter megadáskor. Utolsó, negyedik paraméterként pedig az átadott függvény paraméterét várja a pthread\_create, márha rendelkezik paraméterrel. Amennyiben nincs paramétere akkor NULL-t kell beírni. Ha mégis van paramétere a függvénynek fontos, hogy csak egy paramétert tudunk a pthread\_create függvénybe átadni. Ezért így is kell megírni a futtatandó függvényt. Jellemzően ilyenkor érdemes struktúrát átadni, abba több változó is szerepelhet.

A feldolgozó szállal ellentétben a másik két szálon alkalmaztam a jelzéskezelést. Számoltam azzal az eshetőséggel, ha a felhasználó szeretné megszakítani a processzt, vagy ha a kernel valamiért megszakítást küld a programnak, akkor a futás során allokált memóriák ne



10. ábra Szálak indítása, jelzések küldése és szálak becsatolása

vesszenek el, minden dinamikusan foglalt memóriát fel kell szabadítani. Ezért a főszálba érkező szignált a

**pthread\_kill(pthread\_t thread, int sig);**

függvénnyel tovább tudtam irányítani abba a szálba, amelyiknek az azonosítóját adtam meg. Így szálon belül történik meg a jelzés lekezelés. A feldolgozó szálnál nem akartam külön szignál kezelést csinálni, így oda egy számlálót raktam be. Ahányszor üres a FIFO annyiszor növekszik a számláló. Ha egy bizonyos idő után is üres a FIFO, melyre a TAILQ\_EMPTY makróval vizsgálok rá, akkor megszakítom a végtelen ciklust feltételezve olyan rég óta nincs beolvasott adat, hogy az olvasó szál már nem is működik. Viszont, ha mégis befut egy új adat, amivel tud dolgozni tovább a feldolgozó szál akkor ez a számláló nullázva lesz. Minden létrehozott új szálat be kellett csatolni a főszálba a

**pthread\_join(pthread\_t thread, void \*\*retval);**

függvénnyel különben a főszál nem várta volna meg az alszálak befejezését és semmilyen mérést vagy feldolgozást nem tudott volna elvégezni a program.

## Rendszernaplózás

A rendszer a működése során nagyon sok információt közöl az aktuális állapotról. Felhasználó szinten a legfontosabbak a hibaértesítések és a kiértékelt hőmérsékletek az adott eszközöktől. Ezen információkat a program a standard kimenetre is kiírja. Emellett külön készül logolás, rendszernaplózás időbélyeggel ellátva mind a felhasználónak, mind a fejlesztőnek. Utóbbi célra a Linux beépített syslogját alkalmaztam.

A syslog-gal lehetőségünk van az üzenetek prioritását szabályozni, ezáltal külön állományba készíteni az adott típusú üzeneteket. A syslog különbséget tesz nyolc féle típusú üzenet között. Ezek a fontossági szintek a legalacsonyabbtól a legmagasabb felé:

* LOG\_DEBUG
* LOG\_INFO
* LOG\_NOTICE
* LOG\_WARNING
* LOG\_ERR
* LOG\_CRIT
* LOG\_ALERT
* LOG\_EMERG

A különbség a prioritási szintekben rejlik. Így el lehet határolni a hibákat az információ, a megjegyzés vagy a kritikus szintű eseményektől és nem kell egy állományba végig kutatni a hiba okát a mérési adatok között.

Az üzenetek típusa mellett, lehetőség van különböző jellegű szolgáltatások megkülönböztetésére. Ezáltal a saját magunk írt programunk képes külön készíteni logolást, és nem kell rendszerüzenetek között keresni a programunk üzeneteit. Ezen szolgáltatások lehetnek kernel, rendszerdémon, időzítés, levelező, felhasználó vagy egyéb témakörbe besorolhatóak. Munkám során az egyéb témakörbe tartozó szolgáltatást alkalmaztam mely a LOG\_LOCALn állandó, ahol n egy tesztőleges szám 0-7-ig. Ebből adódóan nyolc féle egyéb szolgáltatás külön logolására is lehetőség van.

A jól elhatárolhatóság végett, a projekt gyökérmappájában található egy rsyslog.conf fájl. Ezzel az állománnyal tudjuk jelezni a rendszernek, hogy a programunk által készített logokat milyen szolgáltatási móddal és milyen prioritási szintekkel szeretnénk elkészítetni. Ezt a fájlt, adott eszközön az /etc/rsyslog.d/ mappába kell bemásolni. A fájl csak minta, lehetőség van a módosítására bármikor. Módosítás esetén viszont a syslog szolgáltatást újra kell indítani, hogy érvényre jusson a módosítás. Ahhoz, hogy a rendszernaplózáshoz tartozó függvényeket és állandókat tudjuk használni a syslog.h könyvtárat kell betölteni az adott forrásállományba.

Használata roppant egyszerű, először meg kell nyitni azt az állományt, amit rsyslog.conf-ba megadtunk. Ezt az openlog függvénnyel tudjuk megvalósítani.

**void openlog(const char \*név, int kapcsolók, int szolgáltatás);**

A név nem kötelező kitöltésű paraméter, csak azt mondja meg a syslogban milyen néven keressük a bejegyzéseket. NULL esetén a program nevét helyettesíti be a szolgáltatás. Három kapcsoló közül lehet választani, de akár mindegyik felhasználható akár bitenként vagy művelettel. Ezek a kapcsolók a következők:

* LOG\_PID
* LOG\_PERROR
* LOG\_CONS

Az első kapcsolóval a folyamat azonosítóját tudjuk logba írni. A másodikkal a szabványos hibacsatornára is megtudjuk jeleníteni a naplóüzeneteket. Az utolsóval azokat az üzeneteket tudjuk kiíratni a szabványos kimenetre, melyek valamilyen hiba miatt nem kerülnek be a rendszernaplóba.

Üzenetek bejegyzése a syslog függvénnyel valósítható meg.

**void syslog(int fontosság, char \* üzenet…);**

A syslog működési elve nagyon hasonló a printf() függvényhez, annyi különbséggel hogy az üzenet elé írjük a fontossági szintet és új sor karaktert automatikusan illeszti a syslog nem nekünk kell beírni.

Program befejezését követően, mint minden állományt, ezt is le kell zárni. Lezárásra a

**void closelog();**

függvényt kell alkalmazni.

## Mozgóátlag

A beolvasott értékek nem biztos, hogy megfelelően mért adatok, lehetnek benne kisebb-nagyobb hibák, melyek finomításra szorulnak. Így egy-egy algoritmuson is végig mennek mire a felhasználó a végleges adatot látná.

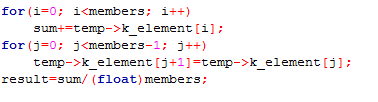
Előbb a mozgóátlag kerül használatra a FIFO-ból kivett adatokkal. A mozgó átlag egy olyan statisztikai módszer mellyel egy sorozat meghatározott részét átlagoljuk az idő függvényében. Más megközelítésben, „A mozgóátlag azt jelenti, hogy az idősor t-edik eleméhez hozzárendelünk egy számtani átlagot, melyet az t-edik elem környezetében lévő bizonyos számú elemből számítunk”. Magát az algoritmust gyakran használják a pénzügyi életben, de mérnöki feladatokban is kiemelkedő szerepe van. Mérnöki feladatok során a véges impulzus válaszú (FIR) szűrőknél bevett módszer. Használata során megkülönböztetünk sima, exponenciális és súlyozott mozgóátlagot. Szakdolgozatom során a sima mozgóátlagot használtam fel. A sima mozgóátlag kitűnően és egyszerűen simítja ki a különböző kilengéseket egy elemzés során. Ezáltal adott időre véve átláthatóbb képet kapunk az adott környezetről és az alapirányzatról, vagyis a trendről.

A mozgóátlag akár lehet 2 tagú vagy többtagú is. Minél többtagú annál inkább elfedi a kilengéseket és kisimítja az idősort.

Páratlan k tagszám esetén az yt (t=1,2…n) idősorból számított k tagú mozgó átlagok sorozata a t=j+1-edik időszaktól a t=n-j-edik időszakig tart, ahol j=(k-1)/2. A t-edik időszakhoz rendelt mozgóátlag:

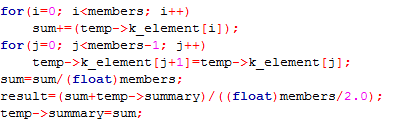


Forráskód szinten a következő módon lett megvalósítva páratlan tagszám esetén:



. ábra Páratlan tagszámú mozgóátlag

Páros tagszám esetén az adott időszak mindig két, eredetileg megadott időszak közé esik. Ez a problémát lehet centírozással, magyarul középre igazítással javítani. A középre igazítás lényege hogy a kiszámított mozgóátlagokat páronként külön átlagoljuk, így a legvégén kéttagú mozgó átlagok sorozatát számítjuk ki. Páros tagszámú megoldás:



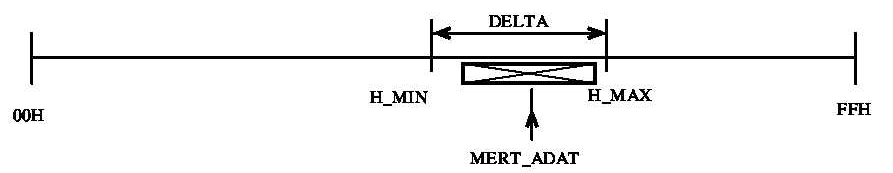
. ábra Páros tagszámú mozgóátlag

Hátránya hogy a mérés elején és a végén nem ad pontos értéket, mert a t[0] időben csak egy mért érték áll rendelkezésre, nem pedig annyi amennyivel korrekt átlagot lehetne számolni. Ugyan ez érvényes az utolsó mérésre is, a végén az utolsó mérés környezete nem adja vissza a környezet hiányossága miatt a pontos mozgóátlagot. Ezt hibát hivatott javítani még a futó hiszterézis.

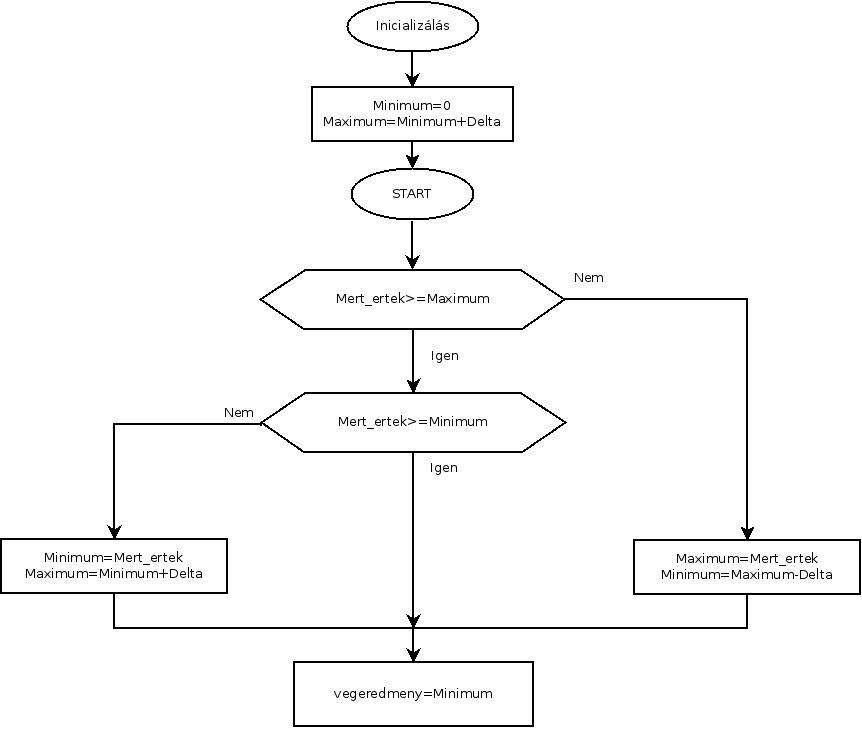
<http://tudasbazis.sulinet.hu/hu/szakkepzes/kereskedelem-es-marketing/kereskedelmi-es-marketing-modulok/atlagos-novekedesi-mutatok-mozgoatlag-szamitasa-megadott-adatokbol/mozgoatlag-szamitasa-utmutato-a-feladat-megoldasahoz>

## Futó hiszterézis

A mozgóátlag által visszaadott érték a még pontosabb eredmény érdekében egy utolsó algoritmuson is végig megy. Ez az algoritmus a futó hiszterézis.

Az algoritmus két paramétert vár, egy delta értéket és egy mért értéket. Utóbbi esetben a mozgóátlag visszaadott értékét. Továbbá rendelkezik egy minimum és maximum értékkel. A minimum és a maximum érték a feldolgozó szál elején megkapja a kezdeti értékeket. A minimumot nullára állítja a rendszer, míg a maximumot a minimum és a delta összege adja.

A mozgóátlagból kapott értéknek a minimum és maximum érték közé kell esni. Ha ez így van, akkor a végleges eredmény a minimum érték lesz. Amennyiben kisebb a minimum értéknél, úgy a minimum érték megkapja a mért értéket, a maximum érték pedig az új minimum érték és delta összege lesz. Másik esetben, mikor a mért érték meghaladja a maximumot, a maximum értéke változik a mért értékre, míg a minimum új értéke az új maximum érték és a delta különbsége lesz. Minden esetben a végleges eredmény az új minimumérték lesz.

A minimum és maximum értékek eltárolásra kerülnek, hogy a következő mérésnél abba a környezetben ellenőrizze a következő mért értéket, így biztosítva a delta hiszerézis mozgását. A delta így folyamatosan tud „csuszkálni” és így még pontosabb mérést tudunk kapni hosszútávon.

. ábra Futó hiszterézis folyamatábra

## Olvasás

Az olvasás során a Motorola protokoll rádiós keretformátumát használtam. Az egész olvasás egy nagyobb switch-case szerkezeten és egy while cikluson alapszik, ami addig igaz, míg az olvasás visszatérési értéke nem mínusz egy és a loop változó nem módosul 0-ra. A keretformátum a következő módon épül fel:

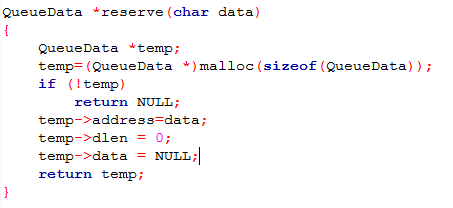
* n\* 0x55, ahol n >=1
* 0xFF
* 0x01
* cím
* parancs
* adathossz
* adat
* crc

Mivel egy hosszabb felsorolásból áll egy teljes keret olvasása, így célszerűnek láttam egy enumerátort létrehozni. Ennek megfelelően, definiáltam egy „PacketState”-t, vagyis egy csomagállapot enumerátort, melyben a felsorolt nevek találhatóak meg annyi különbséggel, hogy az adathosszt és a crc-t két részre bontottam. Erre azért volt szükség, mert a protokoll szerint mind az adathossz, mind a crc két byte hosszúságú.

Olvasás során először egy darab 0x55 (U) karaktert vár az olvasó szál. Könnyen kivehető bináris formába is, hiszen felváltva vannak az egyesek és a nullák benne, 0b01010101. Ha bejött egy ilyen karakter, akkor választási opció áll fenn. Vagy még egy ilyen karaktert várt a rendszer vagy 0xFF-et. Más adat beolvasása esetén újra indul az olvasás elölről. Olvasási szinten „n” értékét 5-ben maximalizáltam, ne olvasson a rendszer feleslegesen a végtelenségig 0x55-t. Ha ötnél több 0x55 jött a vonalon szól a felhasználónak a program, hogy túl sok 0x55 jött be a vonalon és nem jött egy 0xFF se.

Ellenkező esetben, ha 0xFF jött, akkor a következő olvasás során 0x01-et vár a program. 0x01 esetén a switch-ben lévő „State” kifejezés a következő értéket fogja kapni, ami az „address” lesz.

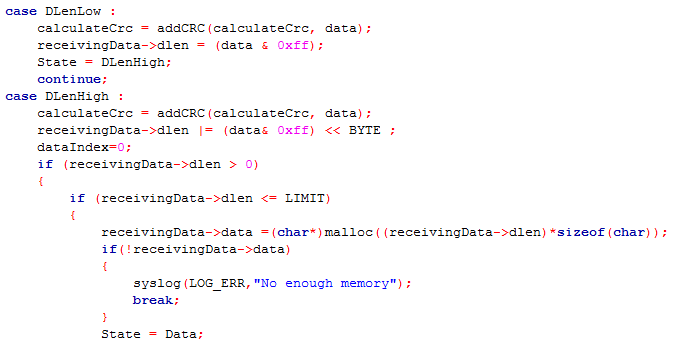
Cím azaz „address” állapot esetén, a következő beolvasott karakter során egy függvény hívódik meg, melyben egy komplett keret allokációjára kerül sor. Memóriafoglalást megelőzően, a vett adat már hasznos adatnak minősül, így erre már kell crc-t számolni, melynek értéke egy két byte-os változóban, a „calculateCRC”-ben tárolódik el. A keret típusa egy előre definiált „Queuedata” struktúra melyben található két karakter típusú változó, egy uint16\_t vagyis két byte-os változó, egy karakter típusú mutató és a TAILQ\_ENTRY makró. A karakterváltozók a címnek és a parancsnak, a két byte-os változó az adathossznak, míg a mutató az adatnak fog megfelelni. A függvény egy karakter típusú mutató paraméterrel rendelkezik. Sikeres allokáció során a paraméterként kapott adatot a struktúra „address” eleméhez rendeli a függvény. A többi elem nulla és NULL értéket kap. Utóbbit a mutató kapja. Végül a függvény visszatérési értéke az allokált keret címe lesz. Memóriafoglalás megelőzően egy feltétel vizsgálat történik meg. A vizsgálat során a program ellenőrzi, hogy az ideiglenes keretnek van-e már címe vagy még nincs. Amennyiben nincs, akkor történik meg a memóriafoglalás, ellenkező esetben túlfutás történik, mely statisztikában van vezetve. Ez csak akkor fordulhat elő, ha előbb érkezik be egy új cím, mint a teljes keret többi része. Ilyenkor egy bizonyos késleltetést célszerű beleilleszteni az adatkérés részben mivel az olvasás folyamatos.



14. ábra Memóriafoglalás egy egész keretnek és címátadás

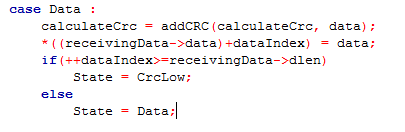
Címfeldolgozás után parancsot várunk a soros portról. A beérkező adat ugyanúgy, ahogy a cím is, egy crc számításon esik át. A beérkezett adatot ezután a keretcsomag „cmd” vagyis parancs eleme kapja meg. Az összes hasznos adat a továbbiakban, az adathossz két byte-ja, a tetszőleges hosszúságú adat és a crc két byte-ja szintén crc számításokon fog keresztül menni. Parancs érték átadás során a következő állapot a „DLenLow” ahová majd adatot olvasunk.

Az adathossz 2 byte hosszúságú, mely Little Endian formátumba érkezik az adatvonalról. A Little Endian a beérkező byte-ok sorrendjéről ad információt. Ilyenkor a legkisebb értékű byte érkezik elsőnek, majd őt követi a legnagyobb értékű byte. Létezik ennek egy másik verziója, a Big Endian, ahol ez pont fordítva valósul meg. A beérkező adat először 0xFF-fel lesz maszkolva, majd utána kapja meg ezt az értéket a keret adathossz eleme. Ezáltal tudjuk rögzíteni az alsó byte-ot az adathosszban. Ezt követően érkezik az adathossz második része a „DLenHigh”. Ez, akár csak az előző módon, ez is maszkolva lesz 0xFF-fel, de mielőtt még hozzárendelnénk az adathosszhoz, a maszkolt értéket egy byte-tal shifteljük (elcsúsztatjuk) balra az adathossznál, majd így a felső byte-ra bitenkénti vagy művelettel rendeljük hozzá az adathosszhoz. Így kapja meg a megfelelő értéket az adathossz. Az adathossz beolvasása után rávizsgálunk az értékre. Amennyiben nagyobb, mint nulla, adatot fogunk várni a következő olvasás során, ellenben, ha nem nagyobb nála, akkor nem fog hőmérsékleti adat érkezni csak crc. Nem nulla értéknél az adathossz értéke egy felső korlát vizsgálaton is átesik. A felső korlát makróként lett definiálva és értéke 1024 byte, vagyis 1Kbyte. Ennél hosszabb adat semmiképp se érkezhet a szolgaeszközök felől, nem csak hogy lelassítanák a hálózatot és túlfutásokat eredményezne, de adatmennyiségre is túl nagy. Ha nagyobb a korlátnál, megszakad a mérés, újra kezdődik a teljes keretolvasás és növekszik a hibás csomagok száma a statisztikában. Megfelelő adathossznál, tehát 0 és 1024 közötti értéknél, adathossznyi memória kerül foglalásra az adat mutatónak és a következő olvasási állapot már hőmérsékleti adatnak felel meg. 0 értékű adathossz nem számít hibának. Ilyenkor csak egyszerű pollingról, életjelérkezésről beszélünk az adott című eszköz felől. Ez esetben a következő olvasás már crc alsó byte-nak számít.



15. ábra Adathosszon végzett bit műveletek

Mért adat érkezése során, miután a beérkező adaton megtörtént a crc számítás, egy segéd változóval indexeljük, hogy a beérkező adat melyik címre érkezzen a lefoglalt memóriaterületen. Az index minden esetben 0-ról indul, minden egyes alkalommal, új adathossz esetén. Az index értéke mindig eggyel inkrementálódik és a következő beérkező adat a lefoglalt memóriaterület következő címére fog így kerülni. A megnövelt index minden egyes értékátadásnál ellenőrzés alá kerül, hogy meghaladta-e az adathosszt vagy egyenlő-e vele. Amikor a vizsgálat igazzá értékelődik ki, az olvasási állapot adatról, crc alsó byte-ra vált át.



. ábra Mért adat beolvasás indexeléssel

A hőmérséklet adat küldésének befejezése során, vagy 0 adathossz esetén, az olvasási folyamat elérkezik az aktuális csomag végéhez, ahol már csak a crc értéket kell beolvasni és összehasonlítani. A crc más néven ciklikus reduncia ellenőrzés egy bonyolult hibafelismerő algoritmus. A crc feldolgozás az adathossz analógiáján alapul.

Először a „CrcLow” adat kerül maszkolás után átadásra a keret crc eleméhez, majd a következő beolvasásnál a „CrcHigh” értéke lesz átadva úgy, mint ahogy az történt „DLenHigh”-nál. Végül a kapott crc és a hasznos adatokon számított crc értéknek meg kell egyeznie különben nem sikerült hibamentesen az olvasás. Ha a kettő crc érték, a számított és a kapott crc érték megegyezik, a függvény megvizsgálja a parancsot. Három lehetőség van. Első esetben a parancs értéke 1 és van adat a keretben. Ebben az esetben a használt keret címét megkapja egy másik keret, melyet beleszúrunk a TAILQ végébe, ha az olvasó szál le tudta foglalni a mutexet. Mutex feloldás után a használt keret címe NULL lesz, az olvasási állapot meg „EmptyPacket” tehát amivel indult az egész olvasás. A statisztikában a hasznos adat értékét növeli a program. Második esetben a parancs értéke megegyezik a PING makróval mely 0x69. Ennél a helyzetnél kiírjuk syslogba annak az eszköznek a nevét mely életjelet küldött és növeljük a statisztikában a polling csomagok és a hasznos csomagok számát. Harmadik esetben valamilyen hiba lépett fel, mely mind standard kimenetre, mind a syslogba kiíratásra kerül.

Az olvasás végén, ha bármi hiba adódott, ami miatt kiléptünk a switch-case valamelyik ágából és volt lefoglalt memória az felszabadításra kerül. Az olvasási állapot „EmptyPacket”-re lesz állítva és a statisztikai értékek, a hibás csomagoktól kezdve a hasznos csomagokig mind belekerülnek syslogba. Így épült fel egy teljes olvasási folyamat.

Szignál érkezése során, az olvasószál ciklusában lévő loop változó 0 értéket kap, melynek következtében a ciklusból kilép a függvény és felszabadításra kerül minden olyan memória mely dinamikusan lett lefoglalva. A syslogba bekerül a szignál fogadása és az olvasó szál befejezése, mint információ.

## Írás és küldés

Az író szál két részből épül fel. A szál főrészét, a kérés küldés alkotja. Ezen belül hívódik meg az író modul. A küldések során két ágra van bontva az egész folyamat. Az egyik ág a Polling, vagyis az életjel kérés, a másik ág a Term vagyis hőmérséklet kérés. Ahogy az olvasás, az írás is a motorola rádiós keretformátumot használja.

Az író modul öt fontos paraméterrel dolgozik melyek a következőek:

* fájlleíró;
* cím;
* parancs;
* adathossz;
* cím.

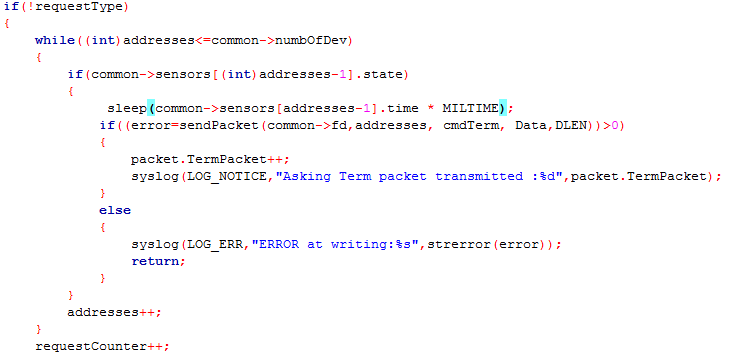
Első lépésben először karakter típusú memóriafoglalás történik. A lefoglalt memóriaterület nagyága adathossz plusz tizenhárom byte. A lefoglalt terület címét egy buffer nevezetű mutató kapja meg. A tizenhárom byte a következő adatoknak kell:

* 5 \* 0x55
* 0xFF
* 0x01
* cím
* parancs
* adathossz alsó byte
* adathossz felső byte
* crc alsó byte
* crc felső byte

Jól látszódik legfontosabb rész, az adat hiánya. Az adat az adathosszból fog adódni, már ha nem 0 lesz az adathossz. A hasznos értékeknél, ahogy az olvasó modulnál is, crc számolásra van szükség. A crc érték egy két byte-os változóba kerül rögzítésre. Memóriafoglalás után minden hasznos adatra elvégzi a program a crc számítást és a számított értéket a két byte-os változóba írja bele. A legvégén a crc változó tartalmát bitművelettel hozzá rendeli két egy-egy byte-os változóhoz. Először lemaszkolja a 0xFF-fel az alsó byte-ot majd jobbra shiftelve egy byte-tal a crc változó tartalmát újra lemaszkolja a másik egy byte-os változóba.

Crc számítás után az ideiglenes buffer feltöltésre kerül a fenn említett sorrendben. Ha van adathossz, akkor az adat az adathossz és a crc közé kerül beírásra. Feltöltés után egy szerre kerül kiíratásra a buffer tömb tartalma. Amennyiben a kiíratás visszatérési értéke az a szám amennyi byte-ot ki akartunk írni a művelet sikeres volt. Ellenkező esetben a naplófájlba jelzi a hiba okát a rendszer. Művelet után a lefoglalt memória felszabadításra kerül.

A kérés küldések során a parancsban egy konstans változó található, aminek értéke vagy 0x69 vagy 0x01. Előbbi az életjel parancsot, utóbbi a hőmérés parancsot reprezentálja. A két kérés 2:1 arányban történik. Tehát minden harmadik parancs hőmérséklet lekérés. Szál indításkor egy számláló kerül deklarálásra melynek kezdeti értéke 0. A while ciklus belsejében a számláló minden egyes küldés során inkrementálódik. A számlálót a program mindig elosztja hárommal és annak maradékából számítja ki, hogy most milyen parancsot kell küldenie. Nulla maradék esetén hőmérés parancs kerül kiküldésre. A számlálót a ciklus elején a program ellenőrzi, hogy elért-e egy maximális értéket. Minden alkalommal mikor eléri a felső határt, kinullázza a számlálót így biztosítva a megfelelő kéréseket adott időnként.

Számláló maradékának kiértékelése után, címenként mennek ki a kérések a vonalra. A címek az eszközstruktúrában lévő tömbből kerülnek felolvasásra. Nem mindegyik eszköz kap kérést, előtte a program megvizsgálja az adott című eszköznek az állapotát. Ha 0 az állapot, akkor ezt az eszközt átugorja és növeli az eszközstruktúra tömb elemét, hogy a következő eszköz címére küldjön ki parancsot.

Egy kérés alatt mindig az összes eszköz felsorolásra kerül, amennyiben a státusza nem nulla.

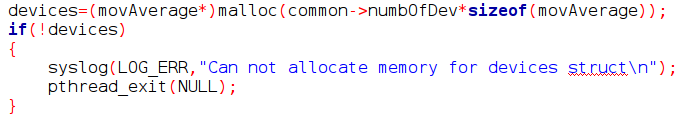
Mielőtt az összes eszköznek menne ki kérés a sleep() függvénnyel késleltetjük az aktuális kérés kiküldését. A sleepben található MILTIME egy tízes szorzó, ezzel biztosítva a megfelelő késleltetést másodpercben. Késleltetés nélkül túl korán olvasnánk be adatot, állandóan nagy lenne a forgalom a vonalon. Két különböző típusú kéréshez két különböző késleltetés tartózik. Elvégre az életjel gyakorisága nem biztos, hogy megegyezik az általunk óhajtott mérési idővel. Ebből kifolyólag az adott címzett a hozzá tartozó idővel van megkésleltetve, ha hőmérés parancs megy ki. Életjelkérésnél az eszközök mérési ideje azonos.

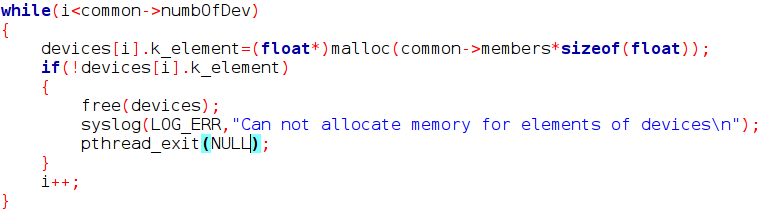
17. ábra Hőmérés parancsküldés eszközönként

## Feldolgozás

Harmadik és egyben utolsó szál a feldolgozó szál. Ebben a szálban történik azon adatok kiértékelése, mely olvasás során belekerültek a FIFO-ba. A kiértékelés a pontosabb eredmények érdekében kettő darab algoritmust használ, a mozgó átlagot és a futó hiszterézist. A mért eredményeket század pontosan kell megjeleníteni a felhasználói felületen.

A szál létrehozásakor két mutató kerül allokálásra. Az első mutató egy olyan struktúra, mely egy lebegőpontos változót és egy lebegőpontos mutatót tartalmaz. Ebből a struktúrából annyi darabra van szükség, ahány darab eszközzel dolgozik a rendszer. Így az eszközök számának függvényben kerül sor memóriafoglalásra melynek a kezdőcímét kapja meg az első mutató.



A lebegőpontos változót a páros tagszámú mozgóátlaghoz kell felhasználni, mint részeredmény. Minden eszközhöz tartozik egy mozgóátlag tagszám. A tagszám mondja meg, hány mért értékből kell mozgóátlagot számolnia majd az algoritmusnak. Ahány tagszámú mérést szeretnénk használni az adott eszközön annyi memóriát kell még külön foglalni az eszközstruktúrában lévő mutatónak.

Sikeres memóriafoglalás után, minden eszköznél a mutató elemei kinullázásra kerülnek, hiszen az eszközökön történő számítások a folyamat elején teljesen kiszámíthatatlan adatokat mutatnának az inicializálatlan elemektől.

Inicializálás követően elindul a feldolgozó ciklus. A ciklus elején a TAILQ\_EMPTY makróval kivizsgálásra kerül a FIFO állapota. Üres FIFO esetén, a while ciklus feltétel vizsgálatában lévő „loop” változó inkrementálódik. Ha elér egy maximum értéket a „loop” akkor nulla értéket kap, majd utána kilép a ciklusból. Minden egyes futás során van egy vizsgálati idő késleltetés, ezzel biztosítva, hogy ne lépjen ki pillanatok alatt a ciklusból és ne fejeződjön be a szál futása idő előtt.

Amennyiben a TAILQ\_EMPTY hamisan értékelődik ki, azaz nem üres a FIFO, először megpróbálja lefoglalni a mutexet a feldolgozó szál. Sikeres foglalás esetén, a legelső csomagot kiveszi a FIFO-ból és visszaadja a mutexet. A kivett csomag címét egy ideiglenes mutatónak adja át. Ezt követően egy for ciklussal megvizsgálja, hogy melyik című eszközt vette ki a FIFO-ból a szál, elvégre nincs garancia, hogy sorba jönnek a csomagok. Ha megvan, hogy melyik című eszközről jött a csomag, akkor a csomag adat eleme egy konverzión esik át. A nyers adat, karakter típusú, de nekünk lebegőpontos adatra van szükségünk. A konverzió az atof függvénnyel történik meg. Az átalakított adatot az eszközstruktúra nulladik eleme kapja. Az átadott értéken először mozgóátlag számítás történik. Ezt követően a mozgóátlag által visszaadott értéket megkapja a futóhiszterézis, mint paramétert.

Végül a futóhiszterézis már a végleges adatot adja visszatérési értékként, mely kiíratásra kerül a standard kimenetet eszköz névvel társítva, valamint bekerül a rendszernaplóba is. Kiíratás után a FIFO-ból kivett csomag adat eleme, majd a csomag is felszabadításra kerül.

## Jelzéskezelés

## Program Befejezés

# Tesztelés

# Befejezés/összegzés

1. FIFO:First in first out, ami elsőnek bejött, elsőnek megy ki [↑](#footnote-ref-1)
2. SoC:System on chip, Egylapkás számítógép [↑](#footnote-ref-2)
3. GUI:Graphical User Interface vagyis grafikus kezelői felület [↑](#footnote-ref-3)
4. CRC: ciklikus redundancia ellenőrzés, egy hiba felismerő algoritmus [↑](#footnote-ref-4)
5. SSH: Secure shell. Titkosított hálózati protokoll. [↑](#footnote-ref-5)
6. RSA: nyílt kulcsú titkosító algoritmus\* [↑](#footnote-ref-6)
7. DDNS: Dinamikus ip címmel rendelkező eszközhöz tartozó rögzített domain név [↑](#footnote-ref-7)
8. Merge: Itt beintegrálás, beépítés, összefésülés [↑](#footnote-ref-8)
9. syslog: rendszer által naplózott üzenetek [↑](#footnote-ref-9)
10. MAXCHAR: itt most 128 [↑](#footnote-ref-10)
11. EOF:End of File az az fájl végéig [↑](#footnote-ref-11)
12. BSD:Berkeley Software Distribution, Berkeley-i egyetemen kifejlesztett disztribúciók gyűjtő neve [↑](#footnote-ref-12)
13. Cast:Típus rákényszerítés [↑](#footnote-ref-13)