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 helye

Tartalomjegyzék

[1. Bevezetés 3](#_Toc499630620)

[2. Hardware 5](#_Toc499630621)

[2.1. Raspberry Pi 2 B 5](#_Toc499630622)

[2.2. UART 7](#_Toc499630623)

[2.3. Rendszer Inicializálás 8](#_Toc499630624)

[3. Segédprogramok 11](#_Toc499630625)

[3.1. Make 11](#_Toc499630626)

[3.2. Git 13](#_Toc499630627)

[3.2.1. Valgrind 13](#_Toc499630628)

[3.2.2. Használata 14](#_Toc499630629)

[3.3. Dia 14](#_Toc499630630)

[3.4. Logrotate 15](#_Toc499630631)

[4. Program 15](#_Toc499630632)

[4.1. Tervezés 15](#_Toc499630633)

[4.2. Konfigurációs állomány beolvasása 18](#_Toc499630634)

[4.3. Soros port 19](#_Toc499630635)

[5. Tesztelés 20](#_Toc499630636)

[6. Befejezés/összegzés 21](#_Toc499630637)

# 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 és egy termosztát vezérlésével, mindezt távolról, 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.” Az IoT eszközök nagy részének (szinte az összes beágyazott eszköz) az alapját valamilyen Linux alapú rendszer biztosítja.

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 olyan hálózat kialakítását kellett megterveznem, aminek a vezérlője egy Linux alapú vezérlő mely maximum 99 db hőmérő szenzorral tudja tartani a kapcsolatot. Az eszköz programját C nyelven írtam meg. A vezérlő feladata egy konfigurációs fájl felolvasása, majd a kommunikáció inicializálása. Inicializálás után több szálon futva, bizonyos időközönként hőmérséklet adatokat kell lekérnie, felolvasnia majd a felolvasott értékeket ki is kell értékelnie. Végül a kiértékelt adatokat egy webes felületen meg is kell tudnia jeleníteni a vezérlőnek. A szolga eszközökről beolvasott adatokat FIFO[[1]](#footnote-1)-ba kellett gyűjtenem majd a feldolgozó szállnak a FIFO-ból kivett értékékkel kell tovább dolgoznia kettő különböző mérési algoritmussal.

Hogy le tudjam fordítani a C nyelven megírt forráskódomat bármilyen Linux alapú céleszközön, külön egy Makefile-t kellett létre hoznom. A Makefile segítségével tudtam a Linux számára futtatható programmá fordítani a forrás fájlokat. Az adatok feldolgozására vagy a program futása során előforduló hibák logolására a Linux rendszer beépített syslog-ját használtam. Ezen kívül még egy másik beépített programot használtam fel a logok megfelelő tárolására, karbantartására. Ez a program a logrotate. Ennek a segítségével biztosítottam, hogy a sok loggolt adat és hiba miatt ne teljen meg az amúgy is kevés tárhellyel rendelkező Linux rendszerem. A fejlesztések során egy git repository[[2]](#footnote-2)-t hoztam létre ennek a projektnek. Ez a repository a [http://github.com](http://github.com/)/ oldalon nyilvánosan elérhető. A lefordított program memória használatának tesztelésére a Valgrind programot használtam. Maga a hardware, melyre a fejlesztés készült egy általam előszeretettel használt Raspberry Pi 2B típusú SoC [[3]](#footnote-3) volt.

# Hardware

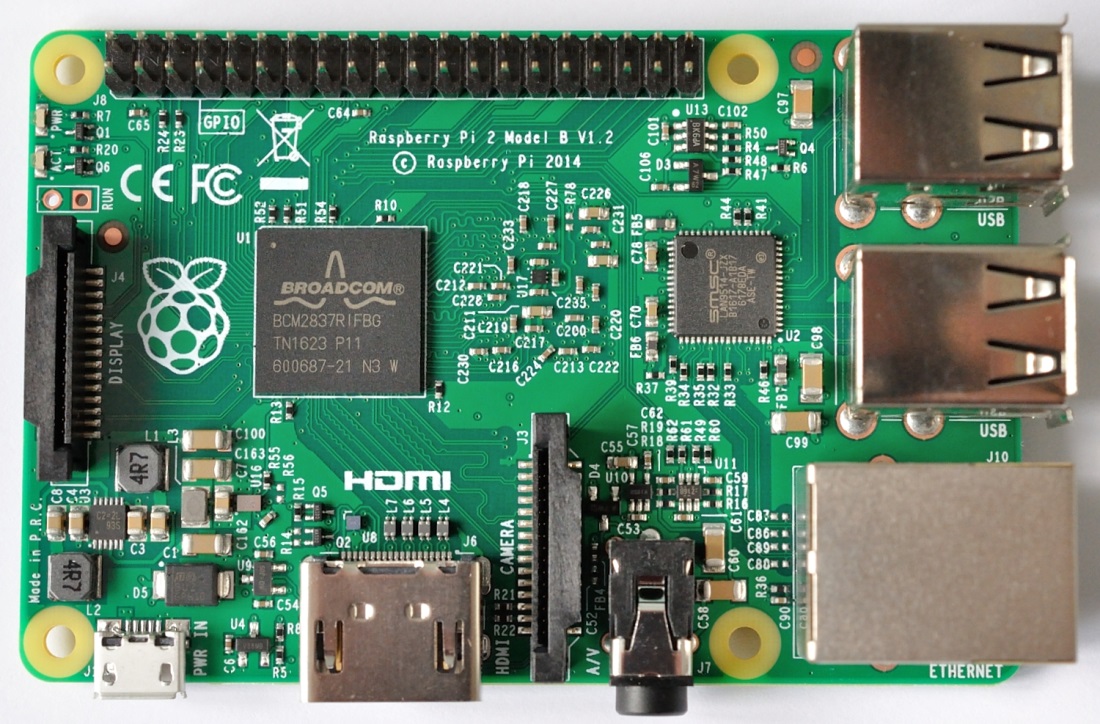
## Raspberry Pi 2 B

Szakdolgozatomat egy Raspberry Pi 2 B típusú egylapkás SoC-re készítettem. 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 vezión. 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[[4]](#footnote-4).--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 szíve is lehetne, 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

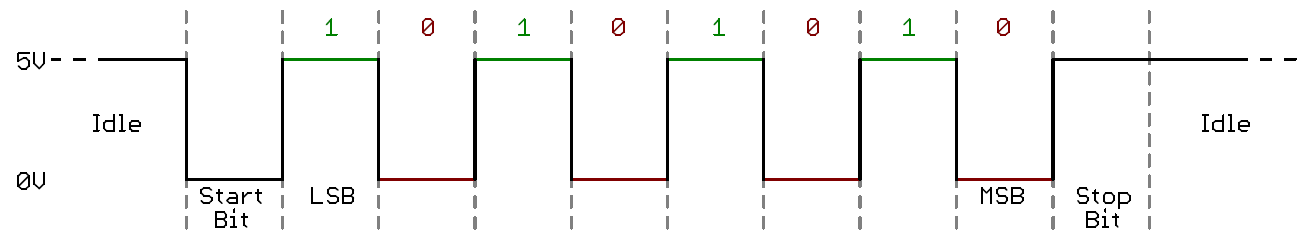
A program kommunikációja fizikai szinten UART-on valósult meg. Az UART jelentése „Universal Asynchronous Receiver/Transmitter” az az univerzális aszinkron adó –és vevő. Az UART a mikrokontrollereknek, SoC-nek olyan perifériája, amely lehetővé teszi az adatok fogadását és adását aszinkron módon. Az adatok 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-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 megfelelő szintillesztésről, megfelelő átalakító használatával ellenkező esetben visszafordíthatatlan kárt tudunk tenni a hardware-ben. UART logikai jelszinte a következő képen látható.



2. ábra UART keretcsomag

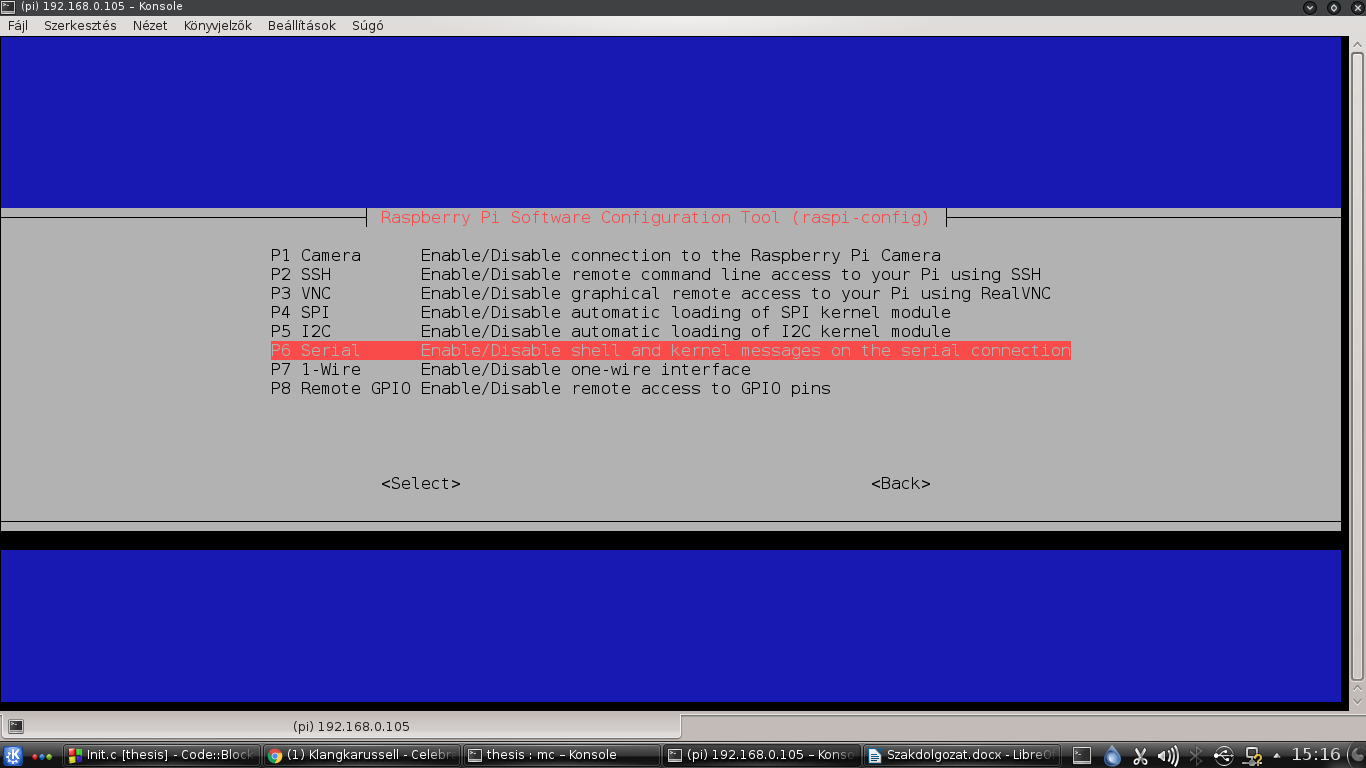
## Rendszer Inicializálás

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 telepítésre került. Talán a legfontosabb a Midnight Commander (továbbiakban MC) mely egy egyszerű fájl kezelő 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 verzió kezelő, Dia diagramkészítő 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.

*$ sudo raspi-config*

Majd a felugró kék felületen, az „*Interfacing Options*”-be belépve a „*Serial*”-t kell engedélyezni.



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

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 megcsinálni. 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 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.” Bár RSA kulcsot nem alkalmaztam egy általam ismert erős jelszó beállításra került amit csak én ismerek.

A program fejlesztésének nagy része viszont már laptopon készült a Code::blocks IDE fejlesztőkörnyezettel. Ez egy ingyenes program melyben rengeteg hasznos plugin, és eszköz található.

Jelentősen megkönnyítette a munkámat az automatikus szöveg illesztő bonyolultabb változó neveknél, függvényeknél vagy egyes headerek beillesztésénél. 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, ezzel is megkönnyítve a hiba keresést is.

# 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.

A programhoz írt Makefile a következő:

CC=gcc

CFLAGS=-g -Wall -lpthread -Iheader

SRC:=src/%.c

SRC\_ALL=$(wildcard src/\*.c)

OBJ:=obj/%.o

OBJDIR=obj/

ALL\_OBJ:=$(patsubst src/%.c,obj/%.o,$(SRC\_ALL) main.c)

RESULT=app/thesis

$(RESULT):$(ALL\_OBJ)

$(CC) -o $@ $(ALL\_OBJ) $(CFLAGS)

$(OBJDIR)main.o:main.c

$(CC) -c -o $@ $< $(CFLAGS)

$(OBJ):$(SRC)

$(CC) -c -o $@ $< $(CFLAGS)

clean:

rm obj/\*.o $(RESULT)

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

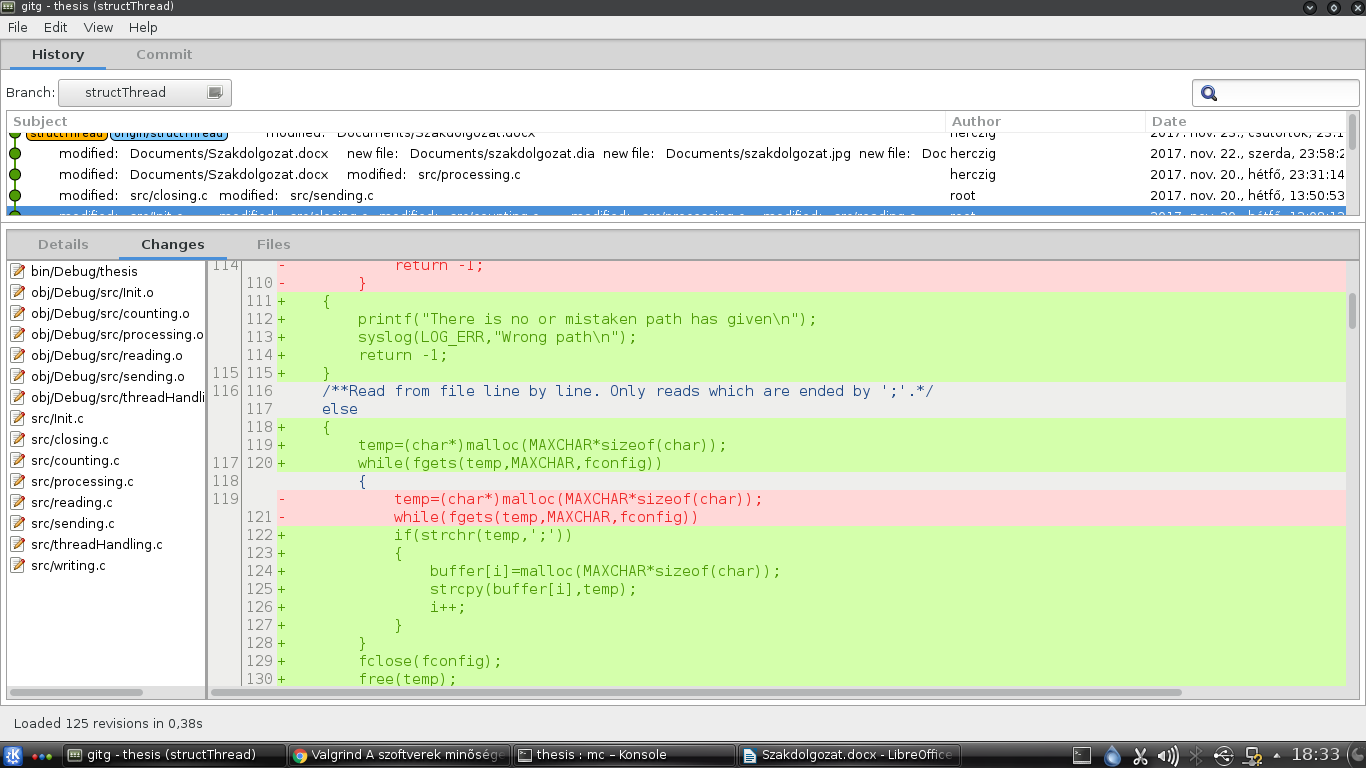
## Git

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 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[[6]](#footnote-6) a fő ággal és onnantól kezdve más is láthatja az újítást.

Projekt kezdéskor mindig inicializálni kell egy .git könyvtárat, 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-t 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.



3. ábra Gitg

## 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

## 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[[7]](#footnote-7)-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, amit tudnia kell, azt hogyan kell tudnia.

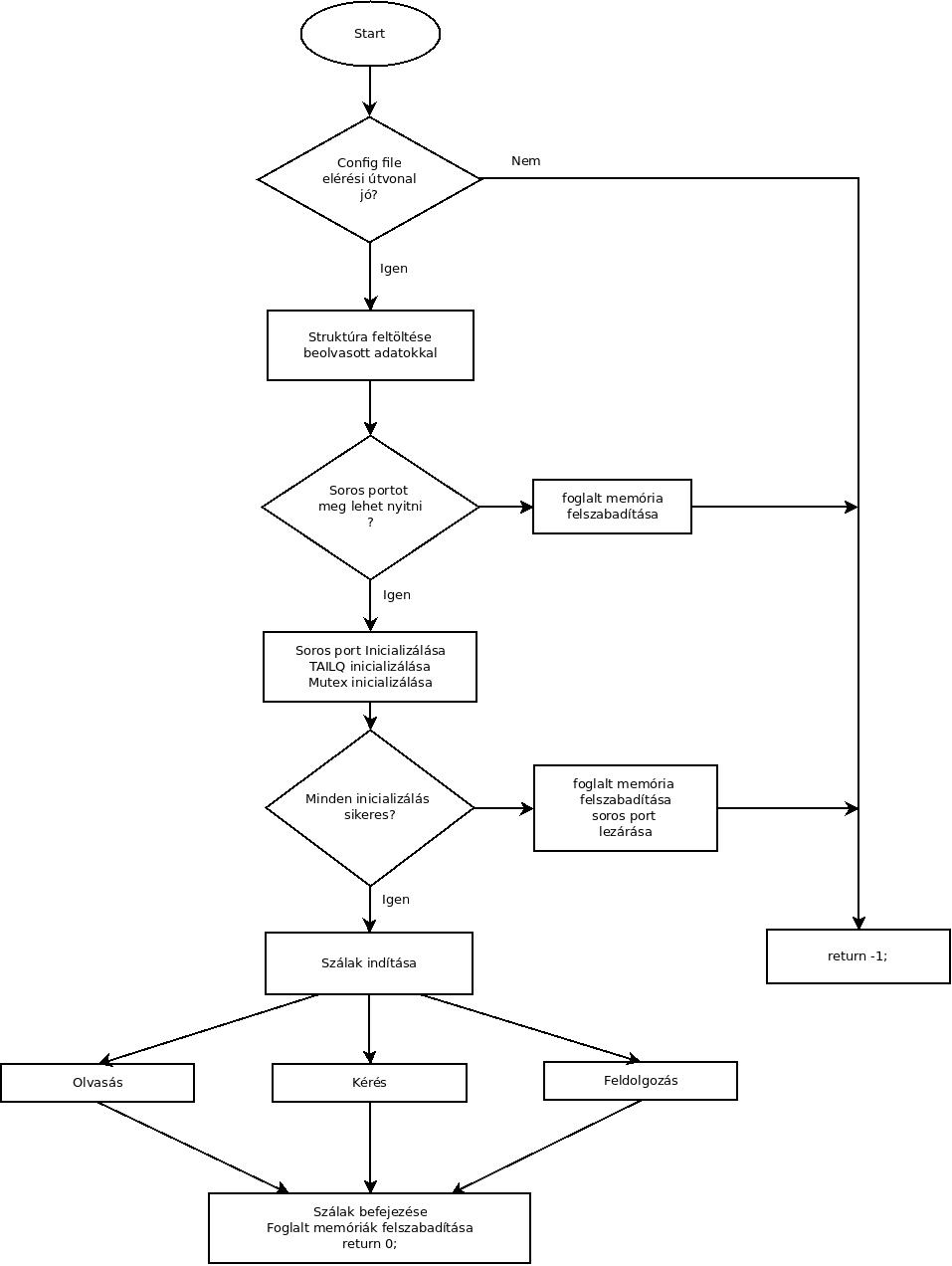
Miután a feladat egy hőmérő rendszer kialakítása volt, fontos volt számon tartani, hogy nem csak olvasni kell az adatokat, hanem feldolgozni és kérést küldeni is tudnia kell. Ennek a 3 folyamatnak egyszerre kellett megvalósulnia. Ebből kifolyólag párhuzamos műveleteket követelt meg a rendszer. A megoldás így a szálkezelés implementálása volt a programban.

További szempont volt a szélkezelésnél a kölcsönös kizárásra (mutex) való figyelés. Ugyanis e nélkül a program gond nélkül feldolgozta volna azt a csomagot, aminek az olvasása még nem fejeződött be teljesen, ez pedig hatalmas hibát okozna a rendszerben.

A beolvasott adatok elhelyezése is egy fontos szempont volt. A cél a következő volt, a rendszer aktuális adatokat mutasson, tehát a szokásos verem kezelés teljesen kizárt volt. Ahhoz, hogy amit legelőször olvastunk be, azt tudjuk kiértékelni legelőször, azt csak is speciális módon, FIFO használatával tudtam csak megoldani. FIFO kialakítására láncolt listára volt szükség melyet egy speciális megoldással, a TAILQ-val valósítottam meg. A TAILQ a queue.h headerben található meg.

Szerettem volna modulárisra írni a programot, minden kisebb-nagyobb feladatot külön-külön modulokba megírni a könnyebb átláthatóságért. Szál létrehozáskor egy adott szál mindig csak egy paramétert kaphat, viszont a szálaknak nem csak egy argumentummal kellett dolgozniuk, hanem többel. Ilyen argumentumok a fájlleíró, a mutex és a TAILQ head része is. Ez okból kifolyólag mindenképp egy struktúra definiálásra is szükségem volt mely ezeket az argumentumokat tartalmazza.

Végül, ami még nehezítette a rendszert a jelzés kezelés megoldása volt. Ha hibába fut a rendszer vagy külső jelet kap user interface (Ctrl+C) felől akkor azt le tudja kezelni és rendesen ki tudjon lépni a program. Ennek tükrében álltam neki először folyamatábrát tervezni mely a következő képen nézett ki.



## Konfigurációs állomány beolvasása

Ahogy a folyamatábra is mutatja program indítása során először egy megadott konfigurációs állományt próbál megnyitni, amennyiben létezik vagy olvasható az állomány. A konfigurációs fájlba kell előre megadni a soros port paramétereit. Ezen felül még az eszközök számát, futó hiszterézis 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 logba ki lett írva a hiba oka. A felhasználónak a standard kimenetre, nekem a fejlesztőnek pedig syslogba.

Sikeres állomány megnyitás után először 128 byte memória foglalást kap egy ideiglenes mutató. Erre a memória részre lesz beolvasva soronként a konfigurációs fájl tartalma ott ahol a sor végén „ ; ” lezáró karakter található. Ez okból kifolyólag fontos a konfigurációs fájl helyes kitöltése melyet a szoftver is jelez, ha ilyen jellegű hiba lépne fel. Az olvasás egészen EOF[[8]](#footnote-8)-ig tart. Sikeres beolvasás során a beolvasott sor egy pointer tömb i-edik pointeréhez lesz másolva mely előtte dinamikusan kapott memóriát a beolvasott sor hosszának függvényében. Amennyiben véget ért a soronkénti beolvasás a mutató tömb elemei soronként átesnek egy „=” karakter vizsgálaton. Abban a sorban ahol van találat, további vizsgálat következik. Ha megtalálható benne egy szó részlet például „Baud” akkor az egyenlőség jel utáni értéket megkapja a struktúra adott eleme. Igyekeztem az összes paraméterre megfelelő szórészletet használni összehasonlítási alapként. Jelen esetben:

arg->BAUD=atoi(p);

Csak akkor kapja meg a beolvasott értéket, ha bizonyos feltételeknek megfelel. Baud rate-nél maradva a beolvasott értéket csak akkor lehet átadni a struktúra elemének, ha a beolvasott érték 9600 vagy 38400, vagy 57600 vagy 115200 . A többi értéknél a beolvasott értéknek egy minimum és maximum érték közé kellett esnie. Amennyiben a struktúra adott eleme nem kapott értéket az állományból vagy nem megfelelő értéket kapott akkor az adott elem egy alapértelmezett adattal lesz feltöltve. Amennyiben az eszközök száma nincs kitöltve vagy nem megfelelő az érték akkor 0-val lesz feltöltve. Ekkor egy értesítést generál a felhasználónak a program és befejezi a futását a procesz.

Sikeres adat feltöltés után az eszközök nevére, címére és állapotára kerül a sor. Ennél a résznél már nem egyenlőség jellel vannak elválasztva az adatok, hanem tabulátorral. Így az adott sorban tabulátor karakterre történik vizsgálat. Ha van tabulátor, akkor a sor végétől indul a feldolgozás a lezáró nulla karakter megfelelő illesztése miatt. Az eszközök tulajdonságai egy másik struktúrába kerülnek eltárolásra. Így először az eszköz állapota kerül feldolgozásra. Ez vagy 1 vagy 0 lehet. Az 1-es az aktív a 0 a passzív állapotnak felel meg. Állapot után az eszköz neve következik. Az eszközök neve miatt kettő 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 melynek a nagysága a két segédpointer távolsága plusz egy byte ami a lezáró nullának kellett.

Befejezésként a címátadás történt meg. Minden egyes feldolgozásnál ügyelni kellett, hogy a beolvasott terület fel legyen szabadítva a kiértékelés után. Sikeres művelet esetén nulla, hiba esetén mínusz egy a visszatérési értéke a függvénynek.

## 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 tudni 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 sikeres 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 deklarálása került sor. Egy, amelyben el lett mentve az aktuális állapot (old\_term), egy pedig az új paraméterek beállításainak (term).

# Tesztelés

# Befejezés/összegzés

1. :FIFO: First In First Out, ami elsőnek megy be, elsőnek is jön ki [↑](#footnote-ref-1)
2. Repository: Tároló [↑](#footnote-ref-2)
3. SoC: System on Chip, az az egy komplett rendszer egy chip-en [↑](#footnote-ref-3)
4. GUI:Graphical User Interface vagyis grafikus kezelői felület [↑](#footnote-ref-4)
5. SSH: Secure shell. Titkosított hálózati protokoll. [↑](#footnote-ref-5)
6. Merge: Itt beintegrálás, beépítés, összefésülés [↑](#footnote-ref-6)
7. syslog: rendszer által naplózott üzenetek [↑](#footnote-ref-7)
8. EOF: End of File, fájl vége [↑](#footnote-ref-8)