PÉCSI TUDOMÁNY EGYETEM
MŰSZAKI ÉS INFORMATIKAI KAR
VILLAMOSMÉRNÖKI SZAK

Herczig Ádám

Szakdolgozat

Hőmérő hálózat IoT felület kialakítása

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

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, amikor IP címmel rendelkező, különböző eszközök kapcsolatban állnak és adatokat cserélnek egymással. Szokták még ezt IoT-nek azaz „Internet of Things”-nek is nevezni. Más kifejezéssel élve „IoT minden olyan dolog és használati tárgy, amely egy hálózaton keresztül - beleértve az internetet is - más gépekhez csatlakozva működik emberi beavatkozás nélkül.” 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. Óvatlanul kiépített hálózat esetén vagy nem megfelelően titkosított kommunikációnál, könnyen egy rossz szándékú hacker áldozatává válhatunk. Gond nélkül figyelheti az adat mozgást, annak gyakoriságát, esetleg manipulálhatja 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 elő feldolgozást beállítania. A mérési időn kívül, további fontos paramétereket a szoftvernek egy konfigurációs állományból kellett kiolvasni. A kiolvasott adatokat és állapotokat az eszközökből egy text fájlba kellett kiírnia a programnak. Az eszközök státuszairól statisztikai listák készítése volt még része a feladatnak, melyet szintén állományba kellett vezetni.

A rendszer központja, melyre készült maga a szoftver, egy Raspberryi PI 2B típusú eszköz volt. A program egész része C nyelven készült el. A mikrokontrollerek számát, melyekről adatgyűjtést végez a szoftver, egy maximum értékben határoztam meg, mennyit legyen képes kezelni a központi vezérlő. Ez a maximum érték módosítható szám. Mivel a kommunikáció az eszközök között soros felületen valósult meg, elengedhetetlen volt a soros port megfelelő inicializálása. Az beolvasott adatok eltárolására és kezelésére egy speciális FIFO[1]-t használtam, a TAILQ-t. Működési szinten a rendszernek nem csak olvasnia kellett a soros portról, parancsokat is kellett tudnia küldeni az eszközök felé. A kiküldött parancsokra érkező válaszokat a központnak fel kellett dolgoznia és különbséget kellett tenni a beérkező parancsok között. A feldolgozásnak, olvasásnak és írásnak egymástól függetlenül kellett működnie, ezáltal, elengedhetetlen volt a szálkezelése megvalósítása a programba. A mért adatoknak legvégül 2 darab, feldolgozó mérőalgoritmuson kellett végig menniük, hogy a felhasználó pontos, kisimított hőmérsékleti eredményt lásson. A sikeresen feldolgozott adatokat, az egyéni rendszer által közölt információkat 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.

[1] FIFO:First in first out, ami elsőnek bejött, elsőnek megy ki

2. Hardware

2.1. Raspberry Pi 2 B

A hardware melyre a program készült egy Raspberry Pi 2 B típusú SoC[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 a további fejlesztőknek. Az eszközön található UART, SPI(2 db) és I^2C (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 integráltál bele a bluetooth és a wifi adaptert, de 3-as verzióba ezek hiányát már pótolták a tervezőmérnökök. Viszont egy egyszerű USB-s wifi adapterrel könnyen áthidalható a hálózati probléma a 2-es verzión. Az eszköz lehetővé teszi kamera, HDMI-s kijelző, touchpad és 3.5mm-es Jack dugó csatlakozását is.

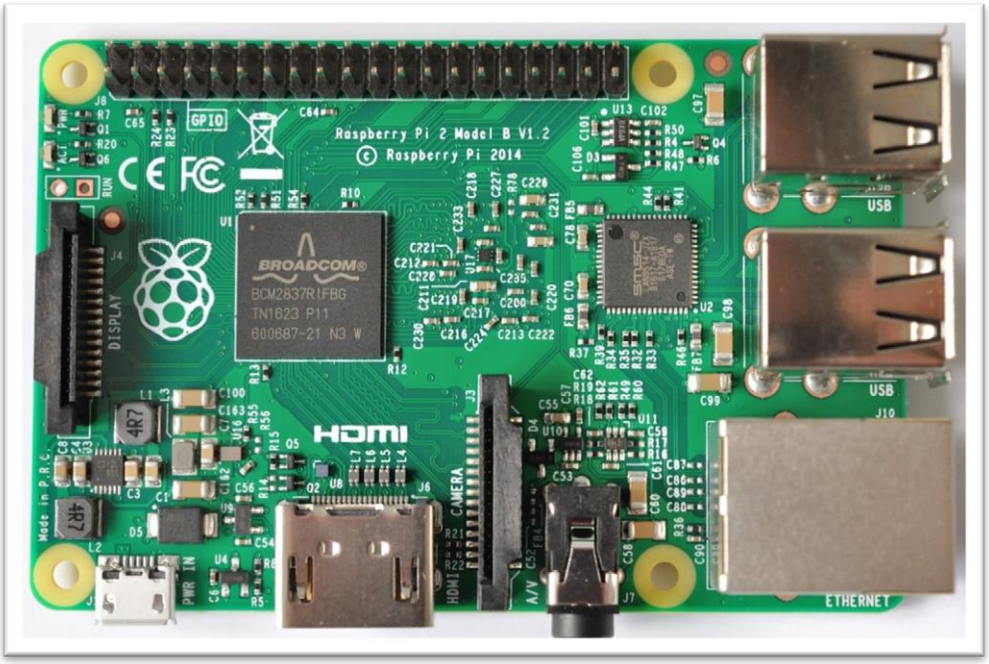
Az eszköz alá microSD kártyával tudjuk biztosítani az operációs rendszert mely jellemzően valamilyen Linux-disztribúció. Hivatalosan Raspbian névre hallgató, a Linux Debian optimalizált verziója ajánlott, mint operációs rendszer, de ezen kívül a Microsoft is bejelentette, hogy a 2-es verzióra már lehetővé 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 a Minibian. Utóbbi a Raspbian minimális operációs rendszere GUI[3] nélkül.

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

[2] SoC:System on chip, Egylapkás számítógép
[3] GUI:Graphical User Interface vagyis grafikus kezelői felület

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 arról, hogy okos ház központjaként 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

2.2. 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-knek 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 jelváltozás 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]-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.

[4] CRC: ciklikus redundancia ellenőrzés, egy hiba felismerő algoritmus

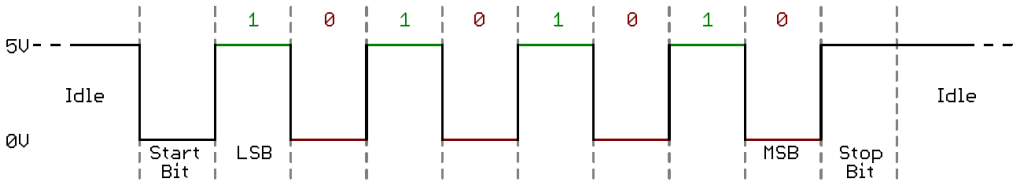
Páros paritás során a paritás bit 0, ha az adatbitek közül az „1”-sek száma páros, különben 1 a paritás. Páratlan paritás esetén a paritás bit akkor 0, ha az adatban lévő „1”-sek száma páratlan, viszont ha páros, akkor 1 a paritás bit.

- 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 tehetünk a hardware-ben. Az UART logikai jelszintje a következő képen látható.



2. ábra UART keretcsomag

2.3. 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

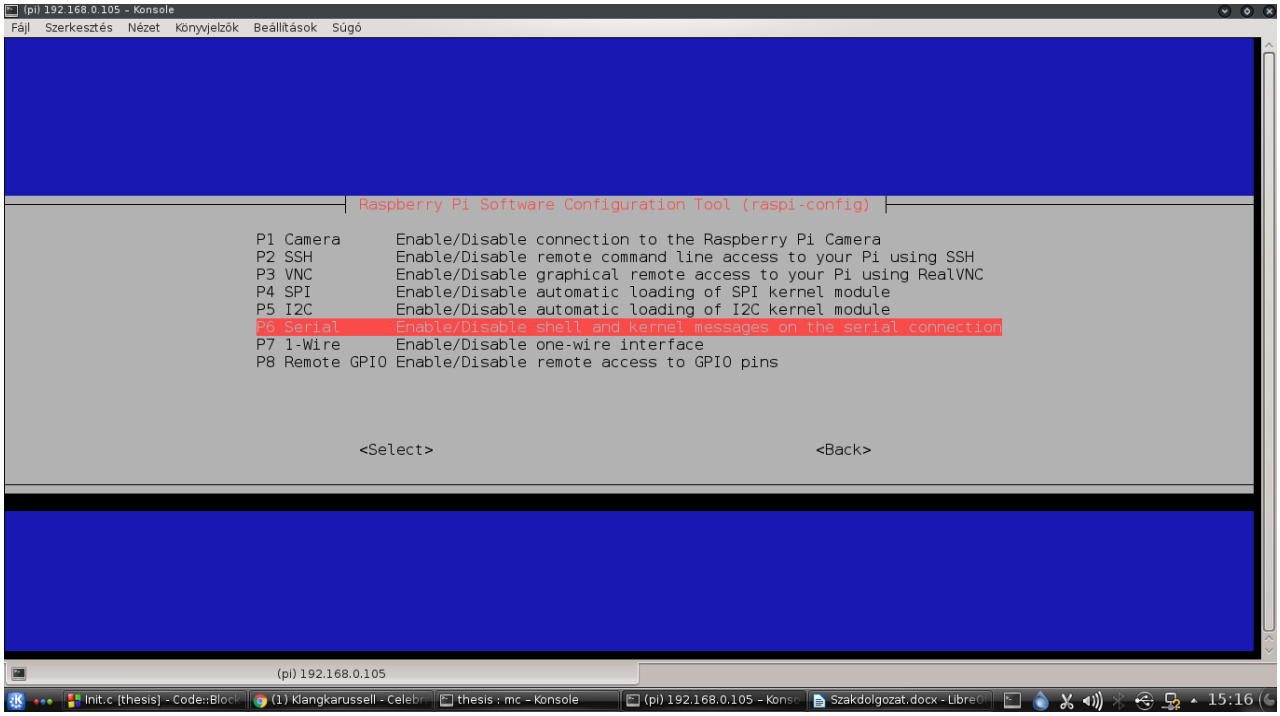
terminálból dolgozni megnehezítette volna a dolgomat így néhány program feltelepítésre került.

Talán a legfontosabb a Midnight Commander (továbbiakban MC), mely egy egyszerű fájlkezelő program. Kinézetre egy az egyben a 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 programjával. 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 értelemszerűen csak akkor kell használni, ha nem rootként vagyunk bejelentkezve. Majd a felugró kék felületen, az „*Interfacing Options*”-be belépve a „*Serial*”-t kell engedélyezni.



3. ábra Soros port hardware-s engedélyezése

Ezen kívül még az SSH[5] is itt lett engedélyezve, hogy továbbiakban hálózaton keresztül biztonságosan 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 volt egy teljes újraindításra.

Mivel az eszköz nem rendelkezik beépített wifi egységgel, így kezdetben csak Ethernet kábellel lehet csatlakoztatni a hálózatra. Szerencsére USB-s wifi adaptert támogatja a Raspberry, így az adapter bedugása után, azt 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 routerhez tartozó jelszót is. Beállítás után, hogy működjenek a módosítások az eszközön mindenképp a következő parancsot meg kellett 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 számomra. 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] kulcsokkal való azonosításra is. „Minden ssh-t használó gépnek

[5] SSH: Secure shell. Titkosított hálózati protokoll.
[6] RSA: nyílt kulcsú titkosító algoritmus*

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.

Ahhoz hogy az eszközt elérjem bárhonnan, külső hálózatból, egy azonosítót kellett adnom az eszköznek. Egy ismerősömnek köszönhetően, biztosítani tudott az eszköznek DynDNS[7] címet, ami azonosította az eszközt az interneten. A /home/pi/ gyökérkönyvtárban található egy shell script, mely egy Linux ütemező daemonnal, a cron-nal fél óránként meghívásra kerül és így biztosítva van a DNS név az eszközhöz. Ezáltal bármikor, kívülről is tudtam írni és tesztelni a programot. Továbbá, ennek segítségével hozzáfértem tesztelési eredményekhez is, mely a későbbiekben új, távlati célokat adott a projektnek.

[7] DDNS: Dinamikus ip címmel rendelkező eszközhöz tartozó rögzített domain név

3. Segédprogramok

3.1. 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, ami nagyon eltérhet, másoknál driver beállítások lehetnek különbözőek, de akár még a könyvtárak verziója is más lehet, ami majd problémát okozhatna. Ezért minden egyes céleszközön le kell tudni fordítani a programot a megadott forráskódokból.

Ettől függetlenül még, ha tesztelés során találtam valami hibát a Raspberry Pi-n és azt egyből tudtam javítani MCeditbe akkor sem 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. 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.

Magának a make-nek 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 le is fordítja 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. Egyúttal, ha új forrásfájl kerül be a projektbe a make észre fogja venni és fordítani fogja az összes többivel együtt és nem kell

külön-külön hozzáadogatni az új forrásfájlokat.

Az általam készített Makefile tesztelve lett mind Raspberry-n, mind otthoni munkalaptopon, de még virtuális környezetben is, ahol minden esetben tökéletesen le tudott fordulni és eltudta készíteni a futtatható binárist. 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)
```

4. ábra Programhoz tartozó Makefile

3.2. Git

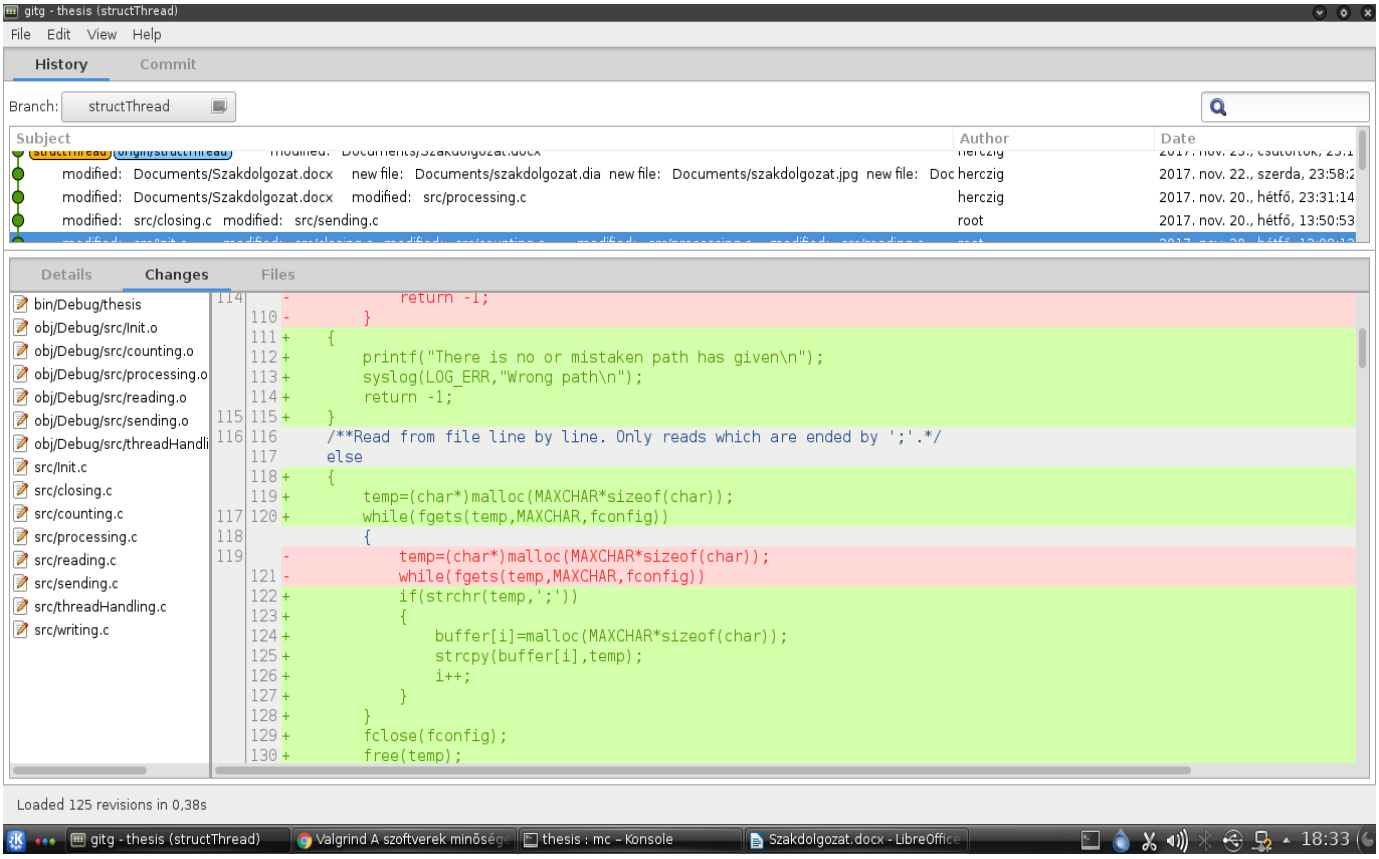
A Git egy verziókezelő szoftver, arra szolgál, hogy kisebb-nagyobb projektek esetén, nyomon lehessen követni a projektek állapotát, a forráskódok tartalmát és verzióját. Ez egy nyílt forráskódú ingyenes szoftver melyet anno Linus Torvalds, a Linux atyja fejlesztett ki. Git kezelésről rengeteg magyar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

Nagyvonalakban arról szól, projekt kezdéskor mindig inicializálni kell egy .git könyvtárat a „git init” paranccsal, ami a munka könyvtárunk lesz. Új állományok létrehozása és a projekthez való hozzáadása esetén a „git add <file1><file2>” paranccsal megmondjuk a Git-nek, melyek az új fájlok, amelyeket szeretnénk hozzáadni a projekthez. Összes fájl esetén a „git add –all” paranccsal egyszerűsíthetjük a műveletet. 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] a fő ággal és onnantól kezdve más is láthatja az újítást.

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.



5. ábra Gitg

[8] Merge: Itt beintegrálás, beépítés, összefésülés

3.3. 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 program segítségével, 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 négyzettel 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 és könyvtárak illesztését is. Utóbbi esetben a pthread könyvtárat kellett külön illeszteni a szálkezelés miatt.

3.4. 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, ahová nem foglaltam le memóriát, 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.

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.

3.5. 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 elnevezés gyanánt, 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. Konverzió használata terminál parancssorból:

$ dia –e output.jpg minta.dia

Az –e kapcsolóval a betöltött minta.dia fájlból output.jpg kiexportálását végez a program. A cél fájl kimenete tetszőleges, lehet .pdf vagy akár .png is.

3.6. Logrotate

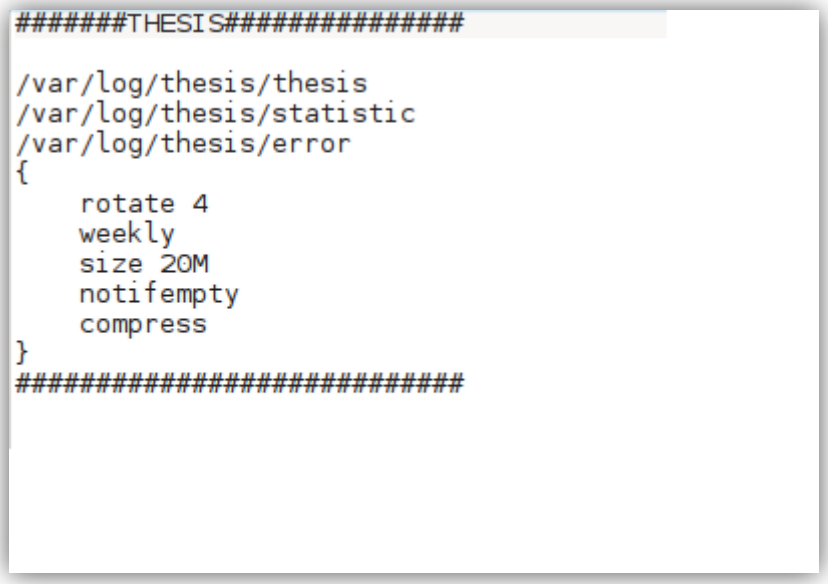
A program működése során információkat közöl a felhasználóval. Ezen üzenetek két részre bonthatóak. Egyik része az üzeneteknek a lemért hőmérsékleti adatok, míg másik része az eszköz működési állapota. Utóbbi alatt olyan rendszerüzeneteket kell érteni, mely jelzi a felhasználónak a konfigurációs állomány 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]-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, egy olyan Linuxos programot 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 által készített rendszernapló hogyan legyen kezelve. Elsőként a fájlban az adott fájl elérési útját kell megadni, amelyeket karbantartani szeretnénk. A felsorolást követően, kapcsos zárójelek között a következő főbb paramétereket tudjuk megadni.

[9] syslog: rendszer által naplózott üzenetek

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

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 az */etc/logrotate.d/* mappába kell csak másolni és a szolgáltatás újraindításával máris kezeli a rendszer a beállításban található állományokat.



6. ábra Logrotate.conf tábla

A fenti képen az általam ideálisnak vélt paraméterek láthatóaks. Hetente egyszer biztosan lefut a program. A „rotate 4” biztosítja, hogy 4 hétig rotálja az adatokat, és az annál régebbieket törölje a logrotate. Ha előbb érné el valamelyik fájl mérete a 20MB-ot, akkor nem várja meg a heti rotálást és majd a törlést, amint eléri a határt, már rotálja is a fájlt. Amennyiben valamelyik fájl üres, akkor azzal nem foglalkozik, feleslegesen nem fogja rotálni, ezt a „notifempty” paraméter biztosítja. Heti rotálás során még egy tömörítést is végre hajt a megjelölt fájlokon a „ compress” paraméternek köszönhetően.

4. Program

4.1. Tervezés

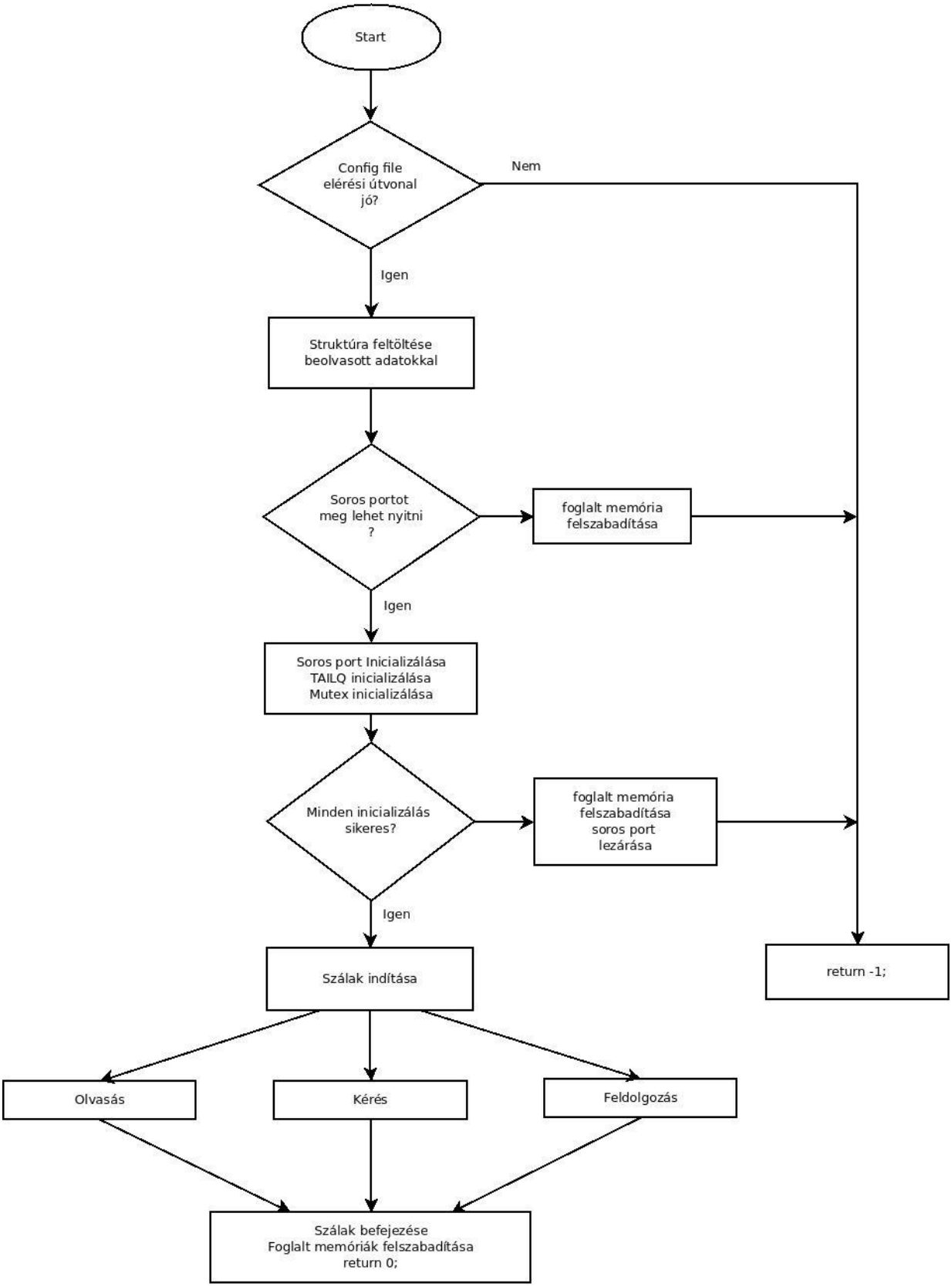
Legelső lépésben mielőtt bármihez is neki kezdtem volna, végig kellett gondolnom hogyan is fog működni a programom. Mit kell tudnia, amit tudnia kell, azt hogyan kell tudnia. Célom az volt, hogy minél átláthatóbbá programot írjak, annyi modult írva amennyire szükség volt a könnyű áttekinthetőség véget. 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, átgondolt rendszertervezésre volt szükség. Szükségesnek éreztem egy használati útmutatót, egy Read_me fájlt csatolni a projekthez.

A programnak induláskor, legelső lépésben egy konfigurációs fájlt kellett 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 helyeztem el még a syslog és a logrotate konfiguráló fájlokat, továbbá a gyökérmappába található a Read_me fájl is. Utóbbi ismerteti a program működését, a helyes konfigurációs fájl kitöltési módját és util mappa tartalmának megfelelő használatát. Az inicializálás utolsó lépésében még a FIFO és a mutexek maradtak hátra.

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 és hibakezelés is.

A szálkezeléseken kívül még jelzéskezelést és a szálakból való megfelelő kilépést kellett biztosítani a programban, a megfelelő működés érdekében.

Végül, de nem utolsó sorban, ha a szálak befejezik 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, a soros port is eredeti állapotát kapta vissza és mind a fájlleíró mind a syslog is bezárásra került.



7. ábra Komplett folyamatábra

4.2. Konfigurációs állomány beolvasása

Ahogy a folyamatábra is mutatja program indítása során először egy megadott konfigurációs állományt próbál megnyitni a program, amennyiben létezik és olvasható az állomány. A konfigurációs fájlba kell előre megadni a soros port paraméterét, a Baud rate-et. Ezen felül még az eszközök számát, a futó hiszterézis delta értékét, a mérési idők gyakoriságát is. A fájlnak tartalmaznia kell még az eszközök címét, nevét, és a mozgóátlag értékeket. 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 byte memóriafoglalást kap egy ideiglenes mutató ahol MAXCHAR 128 byte-nak felel meg. Ezek után egy while ciklusban soronként felolvasásra kerül az állomány tartalma. Az olvasás egészen EOF[10]-ig tart. Olvasás során egy-egy feltételt illesztettem a ciklusba, ha az aktuális sor ’\n’ –újsor- vagy ’#’ –komment- karakterrel kezdődik, akkor azt a sort átugorja a program, ellenkező esetben az adott sort egy pointerekből álló tömb i-edik elemébe másolja át, melynek előtte dinamikusan foglaltam memóriát. Így lehetőségem volt a hasznos sorokat később, egyenként feldolgozni. A felolvasási sorrend kötött, abból a szempontból, hogy előrébb várja a program az eszközök számát, mint paramétert és később az eszközök nevét, állapotát, a mérési idejüket és mozgóátlag tagszámukat, hisz csak így tud megfelelően foglalni memóriát az utóbbi adatoknak a program. Memória foglalási hiba esetén az eddig lefoglalt memória felszabadításra kerül és hibaüzenettel kilép a program. Végezetül az ideiglenes mutató által mutatott terület felszabadításra, az állomány pedig bezárásra került.

Felolvasás után a program, a pointer tömb nulladik elemétől sztring összehasonlításokat végez. Minden egyes összehasonlítás előtt, két segéd pointer kerül felhasználásra. Az első mutató megkapja a sztringben lévő ’=’ jel címe utáni következő címet. A második mutató, a sztring kezdőcímét kapja meg azzal a különbséggel, hogy a sztringben az ’=’ jel helyére lezáró karakter kerül. Sikeres címátadások után a második mutató által mutatott karakterlánc összehasonlításra kerül 5 különböző minta sztringgel. Ha valamelyik mintára az összehasonlítás nullát ad vissza, az első mutató által mutatott érték egy típus konverzió után átadódik az eszközstruktúra megfelelő eleméhez. Amennyiben az első mutató NULL értéket kapna, az adott sor eldobásra kerül és a ciklus előveszi a következő sort a

[10] EOF:End of File az az fájl végéig

mutató tömbből. Ugyan ez az eljárás igaz akkor is, ha az aktuális sztring egyik mintával se passzolt.

```
p=strchr(configbuffer[i],'=');
if(!p)
    continue;
p++;
s=strtok(configbuffer[i],"=");
if(strcmp(s,"Serial.PollingTime")==0)
    {
        arg->pollTime=atoi(p);
        if(arg->pollTime<POLLTIME || arg->pollTime>MAXTIME )
            arg->pollTime=POLLTIME;
        i++;
        continue;
    }
```

8. ábra Összehasonlítás és érték átadás

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 32. Mindkettő határérték makróként lett definiálva, nem pedig konstansként.

Mindenhol igaz ez a minimum-maximum feltétel kivétel a Baud rate-nél, hiszen ott megadott értékek közül kell valamelyiknek megfelelnie. A megadott értékeknek a következőket adtam meg: 9600, 38400 57600 és 115200. 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, mínusz egyes hibakóddal és hibaüzenettel kilép a program. Megfelelő adatfeltöltés után a visszatérési érték a következő sor száma lesz a mutató tömbből.

Adatfeltöltés után az eszközök neve, címe, mérési ideje és még a hozzá tartozó mozgóátlag tagszám beolvasása következik. Minden eszköz ezektől függetlenül rendelkezik két további paraméterrel, státusszal és watchdog számlálóval. Az eszközök tulajdonságai egy másik struktúrába kerülnek eltárolásra. Ez a struktúrának az eszközök számának függvényében kap memóriafoglalást. A feldolgozás során a ciklus változója megkapja a beolvasott sorok számának és az eddig feldolgozott sorok számának különbségét. A sorrend itt kevésbé kötött, elemenként annyi a feltétel hogy elsőként cím legyen megadva, utána a sorrend kötetlen. Ebből mutat is példát a program amennyiben sikertelen a felolvasás, hátha eltévesztette a sorrendet vagy rosszul töltötte ki a felhasználó a paramétereket az

állományban. Minden egyes sor feldolgozása után, ha sikeres volt, ha nem a ciklusváltozó dekrementálódik, a sorszám pedig inkrementálódik. A feldolgozás itt már kicsit máshogy működik, habár az elv az hasonló az előzőekhez. Címre, időre, névre és mozgóátlag tagszámra szórészlet alapján végez a program összehasonlítást. Címnél „address”, mérési időnél „MeasuringTime”, eszköznévre „name” míg mozgóátlagnál „movingAverage” szórészletet keres a soron lévő mutató által mutatott sztringben. Amennyiben az eszköznéven kívül valamelyikre talál példát, az előző megoldás analógiája alapján az „=” jel utáni értéket, konverzióval hozzá rendeli az adott eszközstruktúra eleméhez, majd növeli a mutató tömb számát és csökkenti a ciklusváltozót. Az eszközök watchdog és a státusz elemei a címbeolvasásnál kapnak kezdő értékeket. Minden eszköznél a státusz kezdetben 1-es állapotú az az létezik, míg a watchdog értéke 0. A név feltöltésnél nem konverzió történik, hanem sztring másolás. Viszont a nevek hosszát nem tudhatjuk előre így egy extra műveletként memóriafoglalás is történik a sztring hosszának függvényében.

Utolsó előtti műveletként még a program megvizsgálja, hogy van-e két azonos című eszköz. Ha igen akkor azokat az eszközöket jelenti a rendszer a felhasználónak a

```
else if(strstr(configbuffer[i],"name"))
    {
        len=strlen(p);
        p[len-1]='\0';
        arg->sensors[sensorsNumber].names=malloc((len-1)*sizeof(char));
        if(!arg->sensors[sensorsNumber].names)
            {
                perror("arg->sensors[sensorsNumber].name:\n");
                syslog(LOG_ERR,"no enough memory to allocate for sensors names");
                return -1;
            }
        strcpy(arg->sensors[sensorsNumber].names,p);
        i++;
        k--;
        continue;
    }
```

9. ábra Eszköznév átadás

címütközésről és mínusz egyes hibakóddal kilép a program. 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 hibáról és annak okáról.

Végezetül a program felszabadítja a pointer tömb összes elemét.

4.3. 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 hibacsatornának 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 SIGINT, 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, addig 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.

```
tcgetattr(init->fd,old_term);
term->c_cflag = CS8 | CLOCAL | CREAD ;
term->c_iflag =IGNPAR;
term->c_lflag &= ~( ICANON | ECHO | ISIG);
term->c_oflag =0;
term->c_cc[VTIME]=0;
term->c_cc[VMIN]=1;
cfsetispeed(term,(speed_t)init->BAUD);
cfsetospeed(term,(speed_t)init->BAUD);
```

10. ábra Soros port beállítása

A beállítások végeztével a ki - és bemeneti puffer kiürítésre kerül mielőtt, érvényre jutna a paraméterek beállításának mentése. Sikertelen beállítás esetén az eddig foglalt memória területek felszabadításra kerülnek és hiba üzenettel kilép a program.

4.4. TAILQ és Mutex inicializálás

A rendszer teljes inicializálásához már csak a TAILQ és a mutexek előkészítése maradt hátra. Előbbi segítségével a FIFO-t tudtam egyszerűen 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 és a watchdog számlálót ne írja egyszerre két szál. Ha az olvasó szál lefoglalja a mutexet, addig a feldolgozó szál várni kényszerül. Ugyan ez igaz a kérést küldő és az olvasó szál kapcsolatára is watchdog oldalon. Így nincs meg a veszélye, hogy egy csomagot két szál használna egyszerre, egy időbe.

A C programnyelvben a BSD[11] 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 több más 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 a 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

[11] BSD:Berkeley Software Distribution, Berkeley-i egyetemen kifejlesztett disztribúciók gyűjtő neve

```
typedef struct queueData
{
    /** packet item data */
    char address;
    char cmd;
    uint16_t dlen;
    char *data;
    /**  packet item use a TAILQ. This is for TAILQ entry */
    TAILQ_ENTRY(queueData) entries;
} QueueData;
```

11. ábra TAILQ_ENTRY definiálása

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 szeretnénk hozzáadni 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.

Utolsó lépésben az inicializálás során a TAILQ_INIT makrót és a pthread_mutex_init() függvényt használtam fel egy függvényben.

```
TAILQ_INIT(&arg->head);
if(!(pthread_mutex_init(&arg->temperature_mutex,NULL) || pthread_mutex_init(&arg->watchdog_mutex,NULL)))
    {
        printf("TAILQ_INIT and pthread_mutex_init is successfully\n");
        syslog(LOG_INFO,"TAILQ_INIT and pthread_mutex_init is successfully");
        return 0;
    }
else
    {
        printf("Cannot initialize temperature_mutex and-or watchdog_mutex\n");
        syslog(LOG_ERR,"Cannot initialize mutexes");
        return -1;
    }
```

12. ábra TAILQ és pthread_mutexek inicializálás

4.5. 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 kérést küldő, 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 az őt hívó szálba. Ezen felül még megadható attribútumként ütemezési paraméter, ütemezési szint és ütemezés örökítésnek módja az alszálakra a létrehozandó szálon. Ha NULL-t kap paraméternek, akkor alapértelmezett attribútumot állít a szálnak, tehát becsatolhatónak veszi a szálat. Harmadik paraméternek egy függvény nevet vár, amit majd futtatni szeretnénk a létrehozott alszálon. Nem feltétlenül kell void típusú függvényt írni. Ha így döntünk, mindenképp kell cast[12]-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.

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.

4.6. 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

[12] Cast:Típus rákényszerítés

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 tetszőleges szám 0-7-ig. Ebből adódóan nyolc féle egyéb szolgáltatás külön logolására is lehetőség van.

A jól elhatárolhatóság végett, a projekt gyökérmappájában található util mappában van 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 kezelni. 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 a logban megjeleníteni. A másodikkal a szabványos hibacsatornára is ki tudjuk írni 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 sztring elé írjük a fontossági szintet és az 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.

4.7. Watchdog

A tervezés során kalkulálni kellett az elveszett csomagok figyelésével és annak jelzésével. Amennyiben megfelelő a hálózat állapota, minden egyes elküldött csomagra megjön a válasz. De ez csak az elmélet, amit nem árt ellenőrizni is. Történhet bármilyen apró zaj, vagy nem várt komolyabb tranziens melytől egy-egy csomag elveszhet. Rosszabb esetben akár huzamosabb idő után sincs mért adat, ami aggodalomra adhat okot. Utóbbi esetben ezt valahogy jelezni kellett a felhasználónak, mely az adott eszköz állapotának átállításával történt meg. Erre a feladatra megoldásként egy egyszerű számlálót alkalmaztam.

Minden egyes életjel parancs kiküldés során, az adott című eszköz watchdog számlálóját a program eggyel növeli, melyeknek kezdeti értéke nullára lett beállítva még konfiguráció beolvasáskor. Olvasás során a beérkezett csomagot a program cím alapján

megvizsgálja, majd ez alapján csökkenti a hozzátartozó számlálót.

A watchdog kezelést a kérést küldő és az olvasó szálak között valósítottam meg. Mivel két külön szál kezel egy közös számlálót így elengedhetetlen volt itt is a mutex alkalmazása. Erre a watchdog mutex lett inicializálva még a mért adat mutexével együtt, hiszen a mért adat és a watchdog számlálót nem lehetséges egy mutex-szel kezelni. Elsősorban mert lassítaná és bizonytalanná tenné a rendszert, másrészt így sokkal átláthatóbbá teszi a programot. Ez a fajta lassítás akár okozhatná a számlálók nem megfelelő kezelését, hisz egy elküldött csomag válasza később érkezne be, mert a program egy mutex felszabadítására várakozik és így nem csökkentené a számlálót időben.

A watchdog kezelésénél fontos volt számon tartani, hogy egy-két elvesztett csomag miatt, nem kell már egyből átállítani az adott eszköz státuszát. Elvégre, egy-két zaj nem okoz teljes hibát a hálózatban, viszont célszerű már ezt is jelezni a felhasználónak, ami a statisztikában mutatkozik meg. Egy adott nagyságú számnál már beszélhetünk arról, hogy van egy két hibás szenzor a hálózatban. Makróként definiáltam egy maximális értéket, amit ha az adott eszköz számlálója elér, akkor állítódik csak át a státusza nullára és jelez a rendszer mind syslogban, mind pedig a standard kimeneten.

Amennyiben a hiba kijavításra került vagy csak megszűnt a zavarás a hálózatban, célszerű a státuszt nem nullán hagyni. Ha már egy csomag beérkezik az adott eszköz felől és a program úgy látja, hogy a számláló elérte vagy meghaladta már az adott mennyiségű hiba korlátot akkor kinullázza a számlálót, az állapotát az eszköznek pedig visszaállítja egyre és tiszta lappal figyeli újra majd az eszközt.

4.8. Mozgóátlag

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

Előbb a mozgóátlag, mint algoritmus kerül használatra a FIFO-ból kivett adatokkal. A mozgóátlag 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 életben 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 y_t (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:

A szoftverben a mozgóátlag függvény három paraméterrel dolgozik. Az első egy struktúra, ez tartalmazza a már mért mozgóátlag értékeket adott eszközöknél. Második paramétere az a struktúra, amelyben a program elején definiálva lettek az eszközök. A harmadik paraméter pedig a mérendő eszköz címe. Minden egyes meghívás során egy segédváltozó megkapja a feldolgozandó című eszköz mozgóátlag értékét. Páratlan tagszám esetén egyszerűbb a megoldás, mint páros tagszám esetén.

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:

Minden páros és páratlan tagszámú mérés végén kezdeti érték átmásolódik az őt követőben, az pedig az azt követőbe. Ez addig tart ahány tagszám lett megadva a program indításkor.

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 a hibát hivatott javítani még a futó hiszterézis.

$$\hat{y}_t^{(k)} = \frac{1}{k} \sum_{l=t-j}^{t+j} y_i$$

13. ábra Egyszerű mozgóátlag általános képlete

```
if(tagNumber%2)
{
    for(i=0; i<tagNumber; i++)
        sum+=temp->k_element[i];
    for(j=0; j<tagNumber-1; j++)
    {
        temp->k_element[k]=temp->k_element[k-1];
        k--;
    }
    result=sum/tagNumber;
}
```

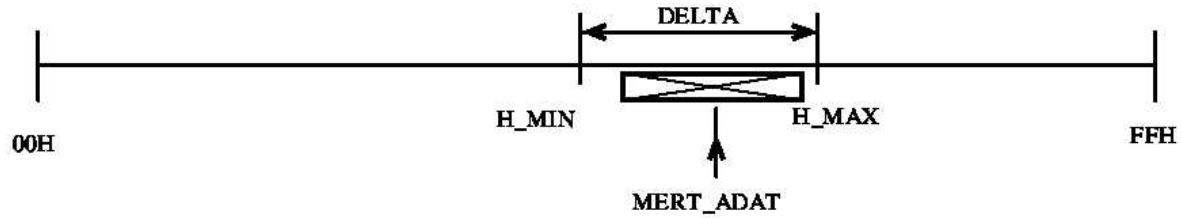
14. ábra Páratlan tagszámú mozgóátlag mérés

```
else
{
    for(i=0; i<tagNumber; i++)
        sum+=(temp->k_element[i]);
    for(j=0; j<tagNumber-1; j++)
    {
        temp->k_element[k]=temp->k_element[k-1];
        k--;
    }
    sum=sum/tagNumber;
    result=(sum+temp->summary)/(tagNumber/2.0);
    temp->summary=sum;
}
```

15. ábra Páratlan tagszámú mozgóátlag mérés

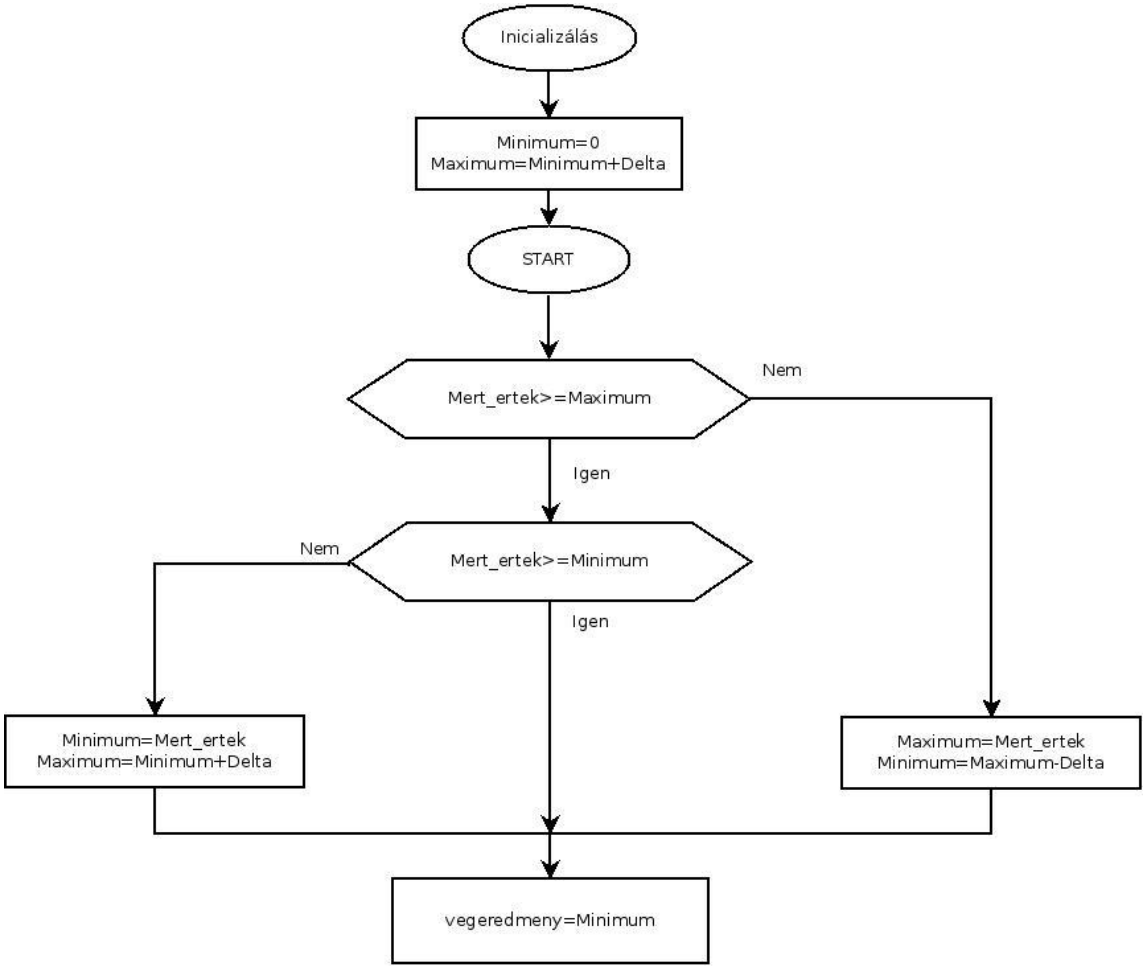
4.9. Futó hiszterézis

A mozgóátlag által visszaadott érték a még pontosabb eredmény érdekében, egy utolsó algoritmus is használatra kerül. 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.



116. ábra Futó hiszterézis működési elve

A minimum és a maximum érték a feldolgozó szál elején megkapja a kezdeti értékeket majd ezt letárolja az eszközstruktúrában. A minimumot nullára állítja a rendszer, míg a maximumot a minimum és a delta összege adja inicializálás során. A mozgóátlagból kapott értéknek a minimum és maximum érték közé kell esni. Ha ez így van, akkor a végleges eredmény a kezdeti 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 érték hiszterézis mozgását. A delta így folyamatosan tud „csúszkálni” és így még pontosabb mérést tudunk kapni hosszútávon.



17. ábra Futó hiszterézis folyamatábra

4.10. Olvasás

Az olvasás során a Motorola protokoll rádiós keretformátumát használtam. Az egész olvasás egy nagyobb switch-case szerkezeten és egy while cikluson alapszik. Utóbbi addig értékelődik ki igazzá, míg az olvasás visszatérési értéke nem mínusz egy. 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 a szolga eszköz felől, akkor a következő olvasás során 0x01-et várt 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”

```
QueueData *reserve(char data)
{
    QueueData *temp;
    temp=(QueueData *)malloc(sizeof(QueueData));
    if (!temp)
        return NULL;
    temp->address=data;
    temp->cmd=0;
    temp->dlen = 0;
    temp->data = NULL;
    return temp;
}
```

18. ábra Ideiglenes keret memória foglalása

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 olvasásnak folyamatosnak kell lennie.

Címfeldolgozás után parancsot várunk, mint adatot 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 byte-onként. 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

```
case DLenLow :
    calculateCrc = addCRC(calculateCrc, data);
    receivingData->dlen = (data & 0xff);
    State = DLenHigh;
    continue;
case DLenHigh :
    calculateCrc = addCRC(calculateCrc, data);
    receivingData->dlen |= (data& 0xff) << BYTE ;
    dataIndex=0;
    if (receivingData->dlen > 0)
    {
        if (receivingData->dlen <= LIMIT)
        {
            receivingData->data =(char*)malloc((receivingData->dlen)*sizeof(char));
            if(!receivingData->data)
            {
                syslog(LOG_ERR,"No enough memory");
                break;
            }
            State = Data;
```

19. ábra Adathossz maszkolás

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,

```
case Data :
    calculateCrc = addCRC(calculateCrc, data);
    *((receivingData->data)+dataIndex) = data;
    if(++dataIndex>=receivingData->dlen)
        State = CrcLow;
    else
        State = Data;
```

20. ábra Adatok indexelése

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.

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.

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 hozzácsatolunk a TAILQ végéhez, 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 lehetőségnél kiírjuk syslogba annak az eszköznek a nevét mely életjelet küldött és növeljük a statisztikában a polling csomagok és a hasznos csomagok számát.

```
toQueueuPacket=receivingData;
pthread_mutex_lock(&common->mutex);
TAILQ_INSERT_TAIL(&common->head,toQueueuPacket,entries);
pthread_mutex_unlock(&common->mutex);
```

21. ábra Mutex lefoglalás és keret beszúrás TAILQ-ba

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.

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 és a standard kimenetre is kiíratásra kerül a szignál fogadása és az olvasó szál befejezése, mint információ.

4.11. Kérés küldés

Az kérést küldő szál két részből épül fel. A szál főrészét, maga a 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 program. 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. Sikeres és sikertelen kérésküldés is vezetve van statisztikában. A sikeres kérések szeparálva vannak a statisztikába, mint PollPacket és TermPacket, hiba esetén csak wError inkrementálódik.
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, mivel a sleep a paraméterben kapott értéke milliszekundum nagyságrendű. 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, értéke pedig a konfigurációs állományban megadott RequestTime-mal egyenlő érték.

4.12. 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.

```
devices=(movAverage*)malloc(common->numbOfDev*sizeof(movAverage));
if(!devices)
{
    syslog(LOG_ERR,"Can not allocate memory for devices struct\n");
    pthread_exit(NULL);
}
```

22. ábra Eszközstruktúra memória foglalás

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

```
while(i<common->numbOfDev)
    {
        devices[i].k_element=(float*)malloc(common->sensors[i].movAve_tag_number*sizeof(float));
        if(!devices[i].k_element)
            {
                free(devices);
                syslog(LOG_ERR,"Can not allocate memory for elements of devices\n");
                pthread_exit(NULL);
            }
        i++;
    }
```

23. ábra Mozgó átlag részelemek foglalása

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 a mozgóátlagnak lebegőpontos adatra van szüksége. A konverzió az atof() függvénnyel történik meg. Az átalakított adatot mindig az eszközstruktúra nulladik eleme kapja. Az átadott értéken először mozgóátlag számítás történik. Ezt követően a mozgóátlag által visszaadott értéket átadódik a futó hiszterézisnek, mint paraméter.

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

Amennyiben a loop változó értéke a ciklusban 0-t kapna, úgy a ciklusból kilépve felszabadításra kerül az eszköz struktúra ás a hozzá tartozó mutatók is.

4.13. Szignál kezelés

A program ideális esetben sosem áll le. Erről a három szálban lévő végtelen ciklus gondoskodik. Csak és kizárólag felhasználói megszakításra vagy kernel által szakítható meg. Mindkét esetben jelküldéssel idézhető elő a megszakításkérés. Ahhoz hogy a processz biztonságosan lépjen ki, mindent bezárjon és felszabadítson abból, amit használt, a kapott jelet le kell tudnia kezelni.

Az olvasó és a feldolgozó szálban az eszközstruktúra „loop” eleme biztosítja a végtelen feltételt, míg az kérést küldő szálban egy globális, statikus „volatile”[13] bool típusú változó biztosítja a feltételt. Az eszköz struktúrában lévő loop változó egy szintén „volatile” változó, melyre a többszálas megoldás végett volt szükség.

Ebből kifolyólag a szignál kezelés a kérést küldő szálban valósítottam meg. Szignál

[13] Volatile: Memóriából felolvasott változó

kezelésre az

int sigaction(int *signum*, const struct sigaction **act*, struct sigaction **oldact*);

függvényt használtam. A „signum”-nak azt a jel értéket adom meg, minek hatására a jelzés kezelő függvényt meg akarom hívni. Jelen esetben a SIGINT[14]-et használtam. Második paramétere az általunk definiált jelkezelő struktúra. Ennek a struktúrának az egyik eleme a lekezelő függvény. Harmadik argumentuma jelen esetben egy NULL értékű argumentum mely régi jelzés kezelők visszaállítására szolgálna. Létezik egy másik szignál kezelő függvény is, viszont a „sigaction” függvény sokkal több rendszerre portolható a másik függvénnyel ellentétben.

A jelkezelő függvényben tudatom a felhasználóval a standard kimeneten, hogy jelzés érkezett és a kérést küldő szálban definiált globális „volatile loop” változót nullára állítom mely a szál fő ciklusának a feltételét adja. A változó módosítás után a szálban lévő művelet befejezi az életjel vagy hőmérés parancsküldő feladatát, majd kilép a fő ciklusból. Ciklusból való kilépést követően az eszközstruktúrában definiált loop változó is nulla értéket kap. Ennek köszönhetően a másik két szál fő ciklusa is befejezi futását, majd végül a két szál is kilép.

4.14. Program befejezés

A program csak akkor fejezi be futását, ha valamilyen okból megszakítják a működését. A megszakítás lehet belső eredetű vagy felhasználói interface felőli szignál küldés.

Akárhonnan érkezik szignál, annak lekezelése után a szálakon belül megtörténnek a memória felszabadítások. Már csak az inicializálás során allokált memória marad hátra, valamint a soros port visszaállítása. Soros port visszaállítása után nullától kezdve az eszközök számáig az összes eszköznév majd az eszköz struktúra felszabadításra kerül. Ha ezzel végzett a program a mutexek felszámolása marad hátra. Legvégül a soros port fájlleírója és a logfájl bezárása marad. Sikeres bezárás után a program értesíti a felhasználót a standard kimeneten és rendszernaplóban is a sikeres program befejezésről.

[14] SIGINT : Ctrl+C által generált jel, értéke 2

```
if(!(old&&fileconf))
{
    syslog(LOG_ERR,"there is a NULL in argumentum list.");
    return;
}
int i=0;
tcflush(fileconf->fd,TCIOFLUSH);
tcsetattr(fileconf->fd,TCSANOW,old);
while(i<fileconf->numbOfDev)
{
    free(fileconf->sensors[i].names);
    i++;
}
free(fileconf->sensors);
pthread_mutex_destroy(&fileconf->watchdog_mutex);
pthread_mutex_destroy(&fileconf->temperature_mutex);
printf("All allocated memory is free and mutex destroyed succesfully."
       "Setting back is succesfully done.\n "
       "Thank you for use this device, hope you enjoy it!\n");
syslog(LOG_NOTICE,"Setting back is succesfully done");
close(fileconf->fd);
closelog();
```

24. ábra Program befejezés

5. Tesztelés

Programírás elkészülése után már csak tesztelni és javítani kellett a teszt során felmerülő hibákat. Kezdeti eszközhiány véget először szoftveres teszteléseket végeztem el. Erre a célra az assert.h könyvtár állt rendelkezésemre. Ennek segítségével az összes általam írt és használt függvényt le tudtam tesztelni melyek rendelkeznek visszatérési értékkel. A void visszatérés nélküli függvények más szimulációs módon kerültek tesztelésre. A szoftveres tesztelésre egy külön mappát hoztam létre, mely megtalálható a projekt mappában.

A szoftveres tesztelésre külön applikációt írtam, melyekben a különböző moduloknak külön-külön írtam meg a tesztelő függvényeit. Az assert makró működési elve hogy paraméterként átadunk neki egy függvényt, általunk megszabott paraméterekkel és egyenlővé tesszük az általunk várt eredménnyel. Amennyiben a lefutás sikeres, tehát azt adja vissza a vizsgálandó függvény, amit terveztünk akkor nulla a visszatérési értéke az assertnek. Ellenkező esetben megszakítást küld a programnak és hiba üzenettel kilép a programból. Ezáltal jó pár hibát észre tudtam venni melynek, lekezelése a program íráskor nem volt egyértelmű, de szerencsére az assert rávilágított ezeknek a hiányosságára. Így került ellenőrzésre a crc ellenőrző modul, a mozgó átlag, a futó hiszterézis, a konfigurációs állomány felolvasó és a soros port beállító modul is.

A létrehozott szálakat viszont nem tudtam az assert makróval tesztelni, így valahogy szimulálni kellett egy szolga eszközzel. Ebből kifolyólag a Raspberry-t összekötöttem a laptopommal. A hardveres megvalósítása a tesztkörnyezetnek egy USB-RS232 átalakító kábellel és egy RS232-UART szintillesztővel történt. A megfelelő hardveres összekötés után a Raspberry elindítását követően figyelmesen végig követtem a működési fázisokat. Kezdeti fázisban sok késleltetést és kiíratást használtam a különböző szálakon, hogy végig tudjam követni, mikor-mit csinál a szoftver.

Elsőként az kérést küldő szálat teszteltem. Elvégre ez volt a legegyszerűbb, itt csak adatokat kellett küldeni a soros porton. A laptopon a Terminal.exe-vel fogadtam a vezérlőről küldött adatokat.

Második fázisban az olvasó szál került tesztelésre. Ezzel kicsit többet kellett bajlódni, már csak a csomagok helyes sorrendje és a crc számolás miatt. Az elején csak egyszerű karaktereket, a protokoll elemeit majd karaktersorozatot küldtem az eszköz felé. Végül, a motorola protokollnak megfelelően teljes csomagokat küldtem. A csomagokat sikeresen be tudta olvasni és elhelyezni a TAILQ-ba az olvasószál melyet a valós csomag statisztika növekedése is bizonyított.

Harmadik fázisban a feldolgozó szál tesztelése következett. Itt tudtam leginkább hasznát venni a Valgrindnek, a megfelelő eszközstruktúrának való memóriafoglalás végett. Sokszor akadt memóriatúlcímzés már csak azért is, mert az eszközcímek 1-től kezdődnek, de az eszközstruktúra tömbelemei nulladik helyről kezdődtek és így sokszor sikerült eggyel túlcímezni a tömböt. Szerencsére a tesztelés során észrevettem ezt a hibát és tudtam javítani. Végül már egyszerre a 3 szálat is tudtam egy időben tesztelni biztató eredményekkel.

Sok próbálkozás után számos sikeresnek mondható tesztet tudtam elvégezni. Sikeres szállétrehozások után a vezérlő a laptop felől kapott teszt hőmérséklet adatok, illetve hamis adatokat is. A tesztek mind belekerültek a rendszernaplóba és minden alkalommal növelték a statisztika megfelelő elemét, ezzel is alátámasztva a tesztek sikerességét.

6. Összegzés és véleményezés

A feladatkiírásban felsorakoztatott pontokat sikeresen abszolváltam a szakdolgozatomban. Ahhoz, hogy egy ilyen rendszer működőképes termék legyen akár a piacon is minden olyan elemet felhasználtam mely a munkám megkönnyítette és segítette. Ezen elemek bemutatásra kerültek a szakdolgozat elején. A céleszközhöz való távoli hozzáférés dinamikus DNS segítségével és SSH protokollon keresztül biztosítva van. Így bármikor nyomon követhető az eszköz.

A konfigurációs fájlból történő felolvasás megvalósításra került melynek eredménye a standard kimenet megjelenésre került. A kommunikáció megfelelő beállításai megtörténtek, megfelelő hardveres szintillesztéssel mind RS232-es, mind RS485-ös protokollon képes kommunikálni az eszköz. s

Megfelelő működéshez a POSIX által biztosított pthread.h könyvtárral valósítottam meg a szálkezelést a programban. Emellett sikeresen fel tudtam használni a sys.queue header által biztosított TAILQ-t mint FIFO-t programban. A headerben definiált makrók segítségével könnyedén tudtam kezelni a FIFO elemeit. Az adatok kiíratása, beolvasása és feldolgozása rendszernaplóba vannak vezetve, ezáltal bármikor visszakereshetőek az eredmények. Továbbá biztosítva van a valós idejű adatközlés a felhasználó felé a standard kimeneten.

A mért adatok megfelelő pontosításához a mozgóátlag és a futó hiszterézis, mint két feldolgozó algoritmus implementálva lett a rendszerbe. A két algoritmus által feldolgozott nyersadatok címmel és hozzátartozó névvel megtalálhatóak a rendszernaplóban.

Távlati tervben még pár szempont felmerült, mint fejlesztési igény számomra. Az eszköz mögé lehetőség szerint valamilyen adatbázis illesztés van tervbe. Ez lehet akár SQL, akár ORACLE adatbázis is. Ezáltal lehetne rendszerezni különböző lekérésekkel az eszközök adatait. Továbbá egy megfelelő webes felület illesztése is tervben van amellyel, megadott időközönként, de valós időben frissítve lennének a rendszer eszközei által küldött hőmérsékleti adatok. A webes felülethez való kapcsolódás megvalósítását még egy megfelelően titkosított csatornán tervezem elkészíteni.

7. Irodalomjegyzék

A szakdolgozatom elkészítéséhez felhasznált forrásokat és dokumentációkat nem csak webes felületről, hanem szakirodalmi könyvekből is szereztem.

Asztalos Márk, Bányász Gábor, Levendovszky Tihamér - LINUX programozás, második átdolgozott kiadás

Syslog : Pere László - Programozás C nyelven-1081 Budapest Népszínház u. 29, Kiskapu Kft, 2003

IoT : https://hu.rs-online.com/web/generalDisplay.html?id=i%2Fiot-internet-of-things

Paritás : https://hu.wikipedia.org/wiki/Parit%C3%A1sbit

Raspberry PI : https://hu.wikipedia.org/wiki/Raspberry_Pi

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

SSH: https://wiki.hup.hu/index.php/SSH

Valgrind: http://epa.oszk.hu/02800/02894/00047/pdf/EPA02894_linuxvilag_41_29_31.pdf

Mozgóátlag: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

Mozgóátlag képlet és használat : 193.6.12.228/uigtk/uise/gtklev/uzlterv_lev_1ea.ppt

írás-olvasás: http://vili.pmmf.hu/~zamek/c/serialframe/packet_8c-source.html

crc: http://vili.pmmf.hu/~zamek/c/serialframe/crc_8c-source.html