Billag Q- kernel-userspace forbindelse og Log data struktur

Szymon Patryk Palka - 201409521

1. juni 2023

Indhold

1 Indledning

I dette dokument er alt udviklingsarbejdet for forbindelsen mellem user-space og kernel-space i form af komunikation mellem knapper og en program som kører på target. Den anden del er implementation af en abstrakt datastruktur som er en log der skal inholde sensor data.

Der blever henvist til en del kode som kan findes i sin helhed i filbilag mappen, i undermapper log og interupt.

2 kernel-userspace forbindelse

2.1 software design

Design af interrupt var MEGET tourbulent og tidskrevende. Årsagen vil være fokleret under del konklusion for Szymon Palka. Det started med brug af en library til rasberry pi som skulle tiladde os at tilgå GPIO'er og lave en interupt rutine . Tanken var meget simpelt. Brugeren sender et eksternt signal til en pin via en knap. Dette udløser en interupt i vores program som reagere afhængi af hvilken knap blev trykket på. Første problem kom da librery bcm2835, kunne ikke instaleres på target. Fejlfinding tog mange arbejdstimer og resulteret med en konsultation med en underviser. Persoen anbefaldte en anden library siden vores valg var ikke optimalt. Der skulle dog være en problem med det ny bibliotek (wireinPi) og det var at den skulle have stor problemer med at køre som statisk biblotek.

I software verden kan man linke bibloteker på to måder (os bekendt) statisk og dynamisk (DLL). Forskel ligger i optimering, hvor en statisk bibliotek skal kopires ind sammen med hver program som benytter den. Hvis man har 50 programer som bruger iostream, skal man koppiere iostream 50 gange. Det er kæmpe spild af plads (relativt) og derfor har man fundet på dynamisk eller delte(shared) biblioteker. Der læses biblioteket ind en gang, og alle programmer kan benytte sig af den ved at loade den ind efter behov ved runtime. Dette er virkelig smart dog virkede ikke på vores arkitektur instaleret på rasberry.

Vi kunne godt updatere arkitekturen, men det ville være meget risiko fyldt af følgene årsager. Vi er 8 medlemmer hvor halvdelen er software studerende som arbejdede på den gamle arkitektur, med komplicerede funktionaliteter som spi, og wifi. Mage af det var i parralet udvikling, og siden interupt tilgang fejlede i slutning af et sprint ville en skift af arkitekturen påtvinge tilpasning fra alle andre gruppe medlemer uden garanti at deres kode ville fungere. Det skal siges at problemet var specifikt for kobling mellem wireingpi version og vores arch version, ikke et generalt problem.

Løsning kom i form af hjælp fra en underviser som havde en udgave af wireing pi der kunne køre statisk. Vi beslutede os at implementere den og siden brug af en bibliotek ikke taler som eget arbejde var det underundnet hvor den kom fra. Vi skulle bare have noget som virkede.

Koden blev implementeret og virkede ved tests, men der kom en udfordring yderligere. På target kunne ISR rutine ikke gå igang. Endnu en hav af timer blev

brugt på fejlfinding som ende i en fiasko.

For tredje gang skulle der implementeres en løsning til bidning mellem brugerer , hardware og software. I ren Desperation blev der udført en forsøg på at 'Hacke' sig frem til en løsnng.

(Det følgene er virklig fejlægtig løsning, men den blev inkluderet siden det var en meget værdifuld læring i hvordan det skal ikke gøres)

Det bestående genbrug af en gamle GPIO driver fra udervisning, som skulle have en ISR rutine tilføjet. I Den rutine blev der oprettet en fil (alvorligt fejl!) som blev skrevet i af vores program. Programmet skrev sin prosses numer i dette fil som blev aflæst af driveren. Derefter kunne der sendes en system signal til programet.

Linux har normal nogle standart signaler som kan sendes til programer (processer) som kill, der lukker for processen. Der er også en Usersig1 som kan have en bruger bestemt betydning. Når programmet modtog dette signal skulle den aflæse hvilken knap aktiveret interuptet, men det kom aldrig så langt.

Dette var virklig dårlig løsning som var en desperat forsøg, på at skære ned på udvikligstiden ved at 'klistre' noget sammen. Der blev arbejdet meget opdelt på dette punkt i projektet og løsningen til dette funktionalitet var faktisk meget lige til, men af en eller andet årsag var den usynlig under selve process.

Dette ilustreret faren via meget selfstendig sprints, og selvom vi holdt opføldningsmøder 2 gange om ugen, blev der aldrig kaster et andet sæt øjne på det. Alle havde det virklig travlt. Men man må aldrig give op selvom situation er håbløst, og med dette gåpåmod startede udvikligen for 4rde gang. Nu helt forefra, og uden brug af externe middler.

Målet var at udvikle en character driver som vil kunne lave en interrupt på brug af GPIO pins. Med det ny klarhed blev der fundet en lige til løsning.

Vi laver en almen driver ti GPIO med standard file operations (read , write osv). Oprette en ISR rutine for hver knap GPIO der bliver aktiveret når spænding skiftes. Og det helt geniale og simple måde at forbinde user space med kernel space, var en blokerende read.

Når Programmet kører, ville dan have en tråd for hver knap som prøver at læse tilstanden af pinen. Dette bliver blokeret indtil en interrupt forekommer. Tråden står stille indtil dette sker, men når interupt er gået af vil den kunne kære videre i sin kode som har en opgave at informere main at der blev tryket på den her knap.

2.2 software implementation

Dette implementation af GPIO character driver er basaret på en gruppe aflevering i HAL. Forfateren har været med til at udvikle og skrive følgende kode, dog af respekt for andres arbejde ville der nævnes det fælles inddsats .Interupt funktionaliteten er ikke resultat af dette gruppe aflevering.

Der vil også blive fokuseret på udvalgte dele af koden, men heleheden kan findes som en fil samt en makefile i bilag.

Makefile og overlay filen blev taget direktra fra opgaven (meg meget små ændringer)

2.2.1 GPIO

Første ting som implementeres er en struct til de GPIO'er som vi ønsker aktiveret. Dette var rigtig smart ide siden vi var lidt uklar på antal knapper nødvendigt til produktet. Ved at automatisere opsætnings processen vile en eventualt tiføjelse kun kræve en opdatering til listen. Ergo 3 inputs, som er nr af GPIO, dens navn, og om det er en input (0) eller output (1).

```
static struct gpio_dev gpio_devs[4] = {{18, "in_b18", 0}
                                      , {23, "in_b23", 0}
                                      , {17, "in_b17", 0}
                                      , {24, "in_b24", 0} };
```

Det næste skridt var at implementere init som inholder alokering af maksimale antal af devices via allocchardevregion(). Oprettes en klasse for vores driver via classCreate(). Initialisere vores fileoperations via cdevinit(), dog her skal der poienteres at selvom write() blev implementeres blev den ikke benyttet i projektet. Der blev også registreret en platform driver via platformdriverregister(), som gøre det mulig at styre devices conectet via en bestemt databus. Det er en krav til hot plug funktionalitet. Det vil ikke være streks nødvendig at implementere for vores produkt, men det tilader plads til udvidelser. Det sidste er registrering af ISR rutiner som kan ses i afsnit 2.2.2

Der skulle selfølge implementeres write() og read() funktion som også kan ses i afsnit 2.2.2.

Det som er mere interesant er probe funktionaliteten som kan ses ned under. Den sørger for at dynamisk allokere GPIO'er, sæte deres retning og oprete nodes i \dev udfra vores struct.

Dette sørger for at resurcerne er registreret når devicet detekteres i sted for at blive alokeret ved initialisering. Det kan tænkes som meningsløst for vores projekt siden vores knapper er altid forbundet, men rent desing messigt tilader det mere fleksebilitet i fremtiden. Det kunne tænkes at vi skulle udstyre vores brugergranse flade med en dongle, som temperatur måler og der vil funktionaliteten med hotpluging være på plads. Måske ikke sammen med knapperne, men kunne genbruges som seperat driver, uden et mere omstendingt arbejde. Koden kører en lykke over vores gpioer og opretter de overnævnte ting. Ved fejl sørges der for frigivning af ressurcer. Dette tankegang er prevelent gennem hele chracater driver.

```
static int gpioDriver_probe(struct platform_device* pdev) {

    int err = 0;
    for (int i = 0; i < gpios_len; i++)
    {
        err = gpio_request(gpio_devs[i].no, gpio_devs[i].name);
        if (err)
        {
```

```
            pr_err("Failed gpio request for %s\n", gpio_devs[i].name);
            goto Error;
        }
        if (gpio_devs[i].dir == 0)
        {
            gpio_direction_input(gpio_devs[i].no);
        }
        if (gpio_devs[i].dir == 1)
        {
            gpio_direction_output(gpio_devs[i].no, 0);
        }
        gpioDriver_dev = device_create(gpioDriver_class,
        NULL, MKDEV(MAJOR(devno), i), NULL, gpio_devs[i].name);
        if (IS_ERR(gpioDriver_dev))
        {
            pr_err("Failed gpio request for %s\n", gpio_devs[i].name);
            goto Error;
        }
    }
    return 0;
Error: for (int i = 0; i < gpios_len; i++)
{
    gpio_free(gpio_devs[i].no);
    device_destroy(gpioDriver_class, MKDEV(MAJOR(devno), i));
}
return err;

}
```

Modsvaret til probe er remove() som dealokerer resurcer når devices ikke kan ses (bliver unpluget) som også implementeres.

de sidste værd at nævne er open() og release() der håndtere åbning og lukning af device filerne via user applikationen.

```
int gpioDriver_open(struct inode* inode, struct file* filep)
{
    int major, minor;
    major = MAJOR(inode->i_rdev);
    minor = MINOR(inode->i_rdev);
    printk("Opening gpio Device [major],
    [minor]: %i, %i\n", major, minor);
    return 0;
}
int gpioDriver_release(struct inode* inode, struct file* filep)
{
```

```
    int minor, major;
    major = MAJOR(inode->i_rdev);
    minor = MINOR(inode->i_rdev);
    printk("Closing/Releasing gpio Device [major],
    [minor]: %i, %i\n", major, minor);
    return 0;
}
```

2.2.2 Interupt

For at kunne oprette en interupt rutine skal den requestes til en specifikt GPIO med en type af interupt. Her vælgte vi falling edge som passer godt ind med design af knapper der er (logisk) høje og lige pludslig går lav når brugeren trykker på dem. Dette fald i spæning registreres og interupt rutine går igang. som man kan se i koden bliver structen taget i brug til gpio nr og navn dog skulle da stadig oprettes en isr rutine per knap. Opridenlig skulle driveren udvikles med mulighed for nem scalering, hvor der løbbes en lykke over GPIO'er som vil tilføjes.Det overnavnete tilgang med seperat ISR rutiner modarbejder denne tankegang. Årsagen var at ny interupt rutine kunne ikke oprettes ved runtime, og en løsning til dette problem kunne være at benyte sig af en ISR som kan sæte foreskelig flag afhængi af hvilken gpio triggede interuptet. Dette blev ikke implementeret af tidsmæssig årsager.

```
 err = request_irq(gpio_to_irq(gpio_devs[0].no), gpio_isr18
                 , IRQF_TRIGGER_FALLING, gpio_devs[0].name, NULL);
    if (err)
    {
        pr_err("Failed to make interupts");
        goto fail_interupt;
    }
```

Som nævnt tideligere skulle der oprettes en rutine for hver af knapper. Nedunder kan man se en eksempel på en af dem. Der blev oprettet en kø som skulle holde øje med de kommene interupts, og en funtion som vil køres hver gang der forkommer en interupt. I den funktion vækker vi køen og sætter en flag. Her blev vi udfordret under tests hvor interuptet blev opfanget flere gange ved en knappe tryk (præl). for at modarbejde dette blev vi enig om at tjekke hvis flaget blev sat ned igen. hvis dette var ikke tilfaldet, betyder det at kernel var stadig igang med at afvikle det som kom efter vores blokering, og interuptet skulle ignoreres.

praktisk svarer dette til at selvom bruger trykker 3 gange umenskligt hurtigt, vil de blive registreret som en tryk, så længe programet arbejder. Dette er selfølgi per knap, så intet forhindrer i at trykke op og ned hvis man vil dette.

```
static DECLARE_WAIT_QUEUE_HEAD(wq17);
```

```
static int flag17 = 0;

static irqreturn_t gpio_isr17(int irq, void* dev_id) {


    if (flag17 == 1)
        return IRQ_HANDLED;
    flag17 = 1;
    wake_up_interruptible(&wq17);


    return IRQ_HANDLED;
}
```

Det sidste som påkreves er at blokere read(). Dette bliver gjort ved en if sætning der tjekker hvilke minor nr (hvilken pin) prøver at benytte read. Dette minor bliver blokeret intil den machende interupt rutine vækker den rigtig kø og sæter et flag. Når dette er opføldt resetes flag (så vi er klar til det næste interupt) og driveren kan læse en værdi og kopier den til buffer i userspace. Der kan poienteres 2 ting omkring det nedstående kode, hvor den første er sleep() funktionen. Det er generalt ikke en god ide at bruge sleep siden det er rent spild af processor kræft. Årsagen var præl.Ved at vente lidt med at sætte flaget ned igen, sikrer vi os med at spænings niveuet på pin'en er stabiliseret. Ikke en ideale løsning, men en som viste sig at fungere i test.

Den anden ting er at i virkligheden status på pins er ligegyldig. Dette bliver brugt som en blokering i en tråd i vores program som har opgaven at informere main().

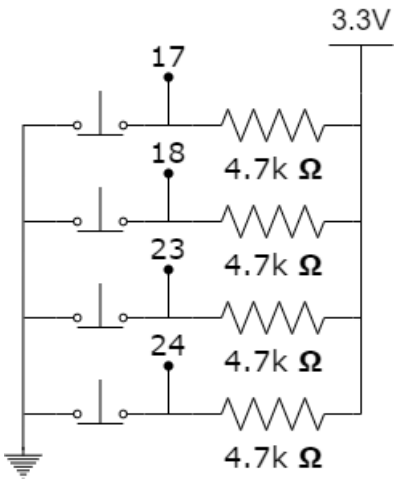
```
ssize_t gpioDriver_read(struct file* filep, char __user* buf,
    size_t count, loff_t* f_pos)
...
...
...
 if (iminor(filep->f_inode) == 3)
    {
        wait_event_interruptible(wq24, flag24 == 1);
        msleep(200);
        flag24 = 0;
    }

    char kbuf[12];
    int len, value, err, minorNum;
    minorNum = iminor(filep->f_inode);
    value = gpio_get_value(gpio_devs[minorNum].no);
```

```
len = count < 12 ? count : 12; /* Truncate to smallest */
len = snprintf(kbuf, len, "%i", value); /* Create string */
err = copy_to_user(buf, kbuf, ++len); /* Copy to user */
```

2.3 test af software

Testen blev udført løbende ved at instalere modulet på target og printe til kernellog strategiske steder (er vi kommet ind is ISR handler? blev der alokeret major/ minor nr? osv) . Det final test foregik via hardware, og beskrivelse ses i afsnit 2.5



Figur 1: kredsløb for knapper

2.4 hardware design

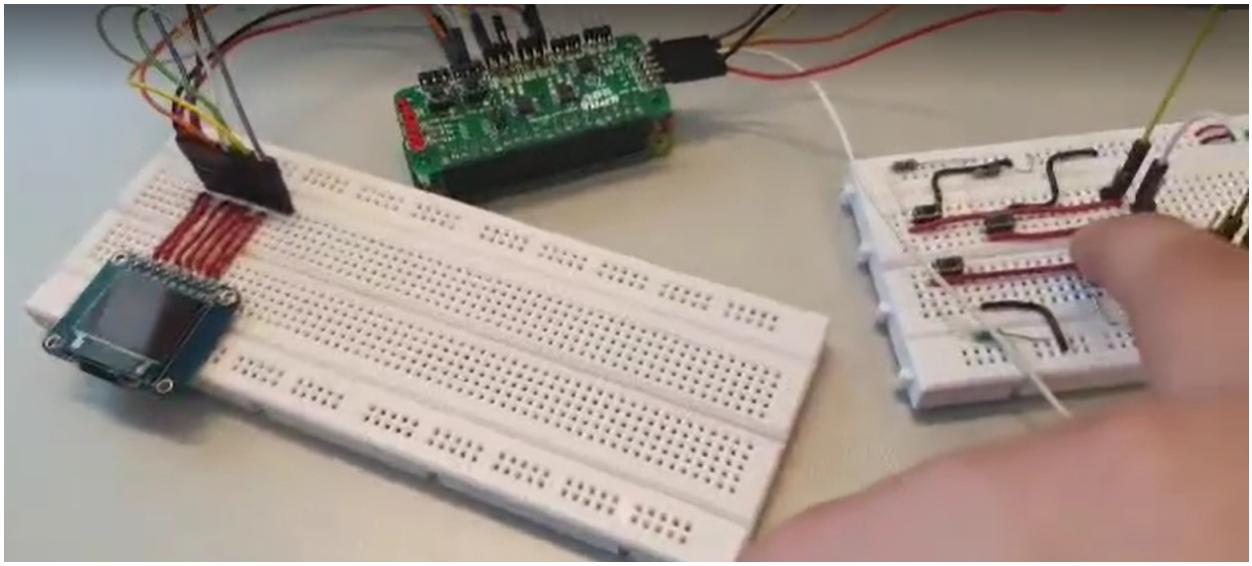
Design af knappe system skulle være forholdsvis simpelt, dog gennemgik en del iterationer. Vi var meget usikker om der skulle være touch , eller hvor mange knapper skulle man bruge for at tillade brugeren at navigere i menuerne af vores produkt. Brugeren skal have en mulighed for at fortage et vælg, og skifte mellem mulighederne. Dette kræver teknisk set 2 inputs (fortag vælget og næste mulighed). Dette blev udvidet på bagrund af fælles beslutninger til en op, nen, enter og tænd/sluk knapp til at styre brugergrænsefladen. Vores tanke var at man vill kune tænde for skærmen og vælge data men ville se udfra en liste. Hvis man vil komme tilbage skulle der findes en "tilbage" mulighed i bunden af listen, dog i retroperspektiv ville det være mere bruger venligt at implementere en "cancel/tilbage" knap.

Vi designet knapper til at være aktiv high med en pull down modstand, hvor både jord og forsøgning skulle tages udfra rasberry pi.

2.4.1 hardware implementation

Vores beslutning skabte urelmessig opførsel såsom flere interupts på en tryk af knappen. Vi prøvede at fejlfinde mellem kode og hardware men kunne ikke se at noget skulle være andet end præl skulle være skydigt. Vi prøvede at forbedre dette ved at slukke for interupt rutinent lige efter aflæsning af en logisk HIGH på pinden , og at instalere en RC kredsløb (efter anbefalding fra underviseren) dog uden lykke.

På Dette tidspunkt vendte vi strategien om og beslutede at det ville være nemere at holde pinsene høj, og forbinde dem til jord når brugeren trykkede. Det endelig resultat kan ses på fig XX. Der har vi en fælles forbindelse til 3.3V fra sasberry pi som går gennem en modstand til hver af de GPIO pins som skal modtage vores input fra brugeren. Ved hver pin er der forbundet en knap (switch) som åbner og lukker forbindelsen til jord. Når en bruger trykker på en knap fuldfører kan kredsløbet og trækker spædning ved pin, til 0, dlv logisk LOW.



Figur 2: test opsætning

2.5 hardware test

Modul test af hardware bestående af målinger af spæding ved analog discovery, dog dette var ikke meget sigende siden alle fremgangsmåder påviste det samme. Det fungeret perfekt. Derfor skulle der gøres en mere omfætne prøvelse hvor elementer kan spille med hianden. Derfor var den endelig funtionalitet testet på følgende vis.

Test blev udført ret simpelt via at opbygge kredsløber fra figur 1 på fumlebrat , og forbinde den til rasberry. Skærmen var testet i forvejen så fremgangsmåden var en big bang tilgang hvor der blev testet for den ønskede funktionalitet, efter systemer blev integreret. testen kan ses i figur 2

Efter par iterationer nævnt i design afsnitet, kunne testen konkluderes som succes.

3 Log

3.1 Software design

Oprindig krav til datastruktur var en måling hver femte minut, non stop i en måned.

dette giver 288 målinger om dagen, 2016 om ugen og 8064 om månen.

Hver måling inhold 6 data punkter på derverne tidspunkt, og hvis der skulle antags at de skulle gemmes som int vil hver af dem koste 4 byte. det vil resultere i ca 194 kb i hukkomelsen. RasberryPi modeler starter med omkring 500mb ram dog at dynamisk alokering på heap kræver mere hukomelse (kræver bla. adresse til hvor det er opbevaret) og er generalt langsomere at tilgå.

En standart værdi for en stack på en rasberry er ofte 8192kB. Der er ikke behov for en specifikt undersøgelse siden formålet er at ilustrerer forholdet til at vejlede designet.

Udfra disse antangelser vil væres data fylde 2.37 procent af vores stack som umildbart lyder ikke som et problem. Dog hvis systemet skulle udvides til en mere profesionalt bruger kunne situation ændres hurtigt. Man kunne forestille sig have flere måle enheder. 12 enheder vil fordoble data. Hvis man vil have logen over et år i sted for (eksempelvis vægten af planten kunne være relevant at se over længere tid) ville forbruget blive omkring 12 gange større. Disse 2 antagelser vil skubbe vores forbrug op på 57 procent, kun til opbevaring af data.

Med dette i tankerne kastede vi os ud i at designe en klasse efter behov.

En af de første idea var seperation af implementering og brug. Dette vil tilade ændringer i implementering, og nemere brug af klassen, eftersom brugeren behøver ikke sætte sig ind i hvordan data bliver håndteret, men kun at kalde de rigtige funtioner. Dette blev opnået via arv. Brugeren har adgang til virtual basis klasse, hvor implementation ligger i en arvet klasse. Dette prencip kaldes for information hiding.

Det næste skridt var at reducere data punkter. Årsagen til dette skyldes formålet med vores produkt er at presentere overordet data for brugeren, mens der forkommer ikke en dybere analyse af hvad dette data betyder. Derfor blev vi ening om ændre opløsning på data.

en time består af 12 målinger, en hver femte minut. Efter disse målinger tages der en gennemsnit og logges i en uge som har 7 dage med 24 timer og en måned som har 30 dage. Dette skære ned til 7*24+12+30 =210 målinger

siden dag har samme opløsning som uge (per time) i sted for at holde doppelt data vil dag skulle uddrages fra ugen når der er behov for det.

Vælg af datastruktur til at representerede dette kom udfra behov at tilføre nye målinger i fronten. En effektiv sturktur til dette er en kø. En anden krav er at der skal slettes den sidste måling. Med det i takerne var deque (doppelt kø) oplagt. Den er lyn hurtig til at manipulere data ved enderne (hovdet og hale) og dens størelse kan justeres meget nemt (modsat en vector).

I selve køen vil der blive indsat en vektor som inholder en måling fra et bestemt sensor på en fikset plads.

Dette vuderes til en bedre lysning end an anden kø, siden størelsen er konstant og meget lille (6 pladser)

I praktis var der nogle ting som ikke fungerede efter designet. Første var retur vardier, der endelig bestemes af designet på skærmen. Dette resulteret i tilføjelse af ny get funktioner, som get lates som retunerer den seneste værdi, og isEmpty som tjekker om loggen er tom. Returværdi blev også lavet op til en mere brubart form der passede ind i skærmens funktionalitet.

i sidste ende på grund af manglende komunikation mellem hardware og software blev antal sensorer skoret ned til 4, hvor den ene behøvedes endelig ikke logges (vandtanks niveu).

Disse ændringer skete omkring integrations fasen, som resulteret i ændring af designet til noget som ikke menede om det oprindelig design. Der blev fortage en valg om at beholde de gamle struktur, siden det virkede som oprindelig ønsket, og igen giver gode muligheder for udvidelse

3.2 Software implementation

Selve implementering kan opsimeres i setLog() funktionen. Årsagen er at den datatype deque er. Den bestor formenligt af en datapunkt og en pointer til det næste datapunkt. en recursiv data struktur. Dette kan analyseres udelukkende på bagrund hvordan nyt data bliver opretet. Der påkreves ikke størelse før strukturen er initialiseres, dette betyder at det ikke er en array. Elementer kan frit tilføjes og sletes fra enderne via push, og pop, som stemmer overens med dette implementation.

```
void setLog(const std::array<int, 4>& values) override {

        // push hour
        _hour.push_front(values);
        m++;

        if (_hour.size() > 12)
            _hour.pop_back();

        //After full hour take avarage and push to week

        if (m == HOUR)
        {
            m = 0;

            waterLevel = 0; moisture = 0; light = 0; weight = 0;

            for (size_t i = 0; i < 12; i++)
            {

                waterLevel += _hour[i][0];

                moisture += _hour[i][1];

                light += _hour[i][2];

                weight += _hour[i][3];
            }

            _week.push_front({ waterLevel /
            HOUR,moisture / HOUR ,light /
            HOUR ,weight / HOUR });

            h++;

        }

        //After full Day take avarage and push it to month
```

```
        if (_day.size() > 24)
            _day.pop_back();

        if (h == DAY)
        {
            h = 0;

            waterLevel = 0; moisture = 0; light = 0; weight = 0;

            for (size_t i = 0; i < DAY; i++)
            {

                waterLevel += _week[i][0];

                moisture += _week[i][1];

                light += _week[i][2];

                weight += _week[i][3];
            }

            _month.push_front({ waterLevel
            / DAY,moisture / DAY ,light
            / DAY ,weight / DAY });

        }

        if (_month.size() > 30)
            _month.pop_back();
    }
```

som der kan ses i koden overfor bliver der heletiden sørger for at maximalt størrelse af køen overholdes. Når den opnåes tages der gennemsnit som så pushes i næste kø og så videre. selve kø eksisterer ikke i memory med mindre er bliver alokeret en data punkt. Da strukturen består an en kø med indlejrete vektor kan de enkelte data tilgåes ved at benytte doppelt pointer [][].

overordnet fungerer dette som en roterende buffer, hvor længden af hver kø kan være valgfrit(der skal ændres i implementation, men vi det er ændring af enkelt definition, som gør det nemt at updatere).

Oprindeligt kode retuneret hele data typen i form af en kø, som indhold vektorer der holdt data fra alle sensorer fra en målig. Dette viste sig at være for kompliceret at benytte sig af andre bruger, og blev ændret langt efter implementation til at retunere en array af datapunkter for en enkelt sensor. Pga pladsmangel var der ikke muligt at fremvise mange værdier, og getweek() blev ændret til at retunere en gennemsnits værdi per dag, seneste 7 dage tilbage. Det kan ses i koden ned uder.

```
std::array<int, 7> getWeek(int type) const override {

        std::array<int, 7> data = { -1,-1,-1,-1,-1,-1,-1 };

        int max =_month.size()>7    ?   7    : _month.size();

        if (max!=0)
        {

        for (size_t i = 0; i < max ; i++)
        {
            data[i] = _month[i][type];
        }

        }

        return data;
    }
```

Der kan også ses en eksplosit override, som oversktiver klassens interface (virtualt klasse), for at gemme implementering.

Måden man vælger hvilken sensor målinger man vil få fat i er ved at give en int til funktionen. den fortaler hvilken plads i vektoren vil mankigge på. Dette kan lade sig gøre fordi målinger er altid på en foubestemt plads. eksempelvis er vandtanks niveu altid på plads 0. for at gøredet mere bruger venligt bestemte vi os for at bruge keywords som er mere siegne.

```
#define WATERLVL    0
#define MOIST       1
#define LIGHT       2
#define WEIGHT      3
```

getLatest() var trivialt at implementere siden den seneste værdi er første pladsen i hour atributten.

samme med isEmpty() som benyter sig af .empty() funktion for deque på alle dele i vores log. hvis alt er tomt retuneres der true.

3.3 Software test

testen blev primært udført via en test program hvor der blev testet følgende.

gennemsnit tages korrekt : alle vardier for timer sætes til noget som resulterede med en kendt gennemsnit. derefter tjekkes den første måling på ugen, som er gennemsnitet af en hel dag.

roterene buffer: der oprettes 600 dages antal målinger, og hele logen printes ud. der brudte ikke være mere end 30 datapunkter (1 per dag)

is empty() testes før noget data bliver oprettet, og brudte vise 1. dette gentages efter tifaldig data pushes in, og brudte visse 0.

logclear() testes sammen med isempty() hvor efter en sletning af loggen via logclear() brudte den videregive at loggener blevet tom.

Get latest er trivielt da den skulle retunere den seneste værdi som blev pushet.

```
int main() {

    int a, b, c, d;

        Log log;

        // Test setLog method

    bool lvl = log.isEmpty();

    std::cout << lvl << endl;



        for (size_t i = 0; i < 12 * 24 * 600; i++)
        {
            //int a = rand() % 10;
            int b = rand() % 10;
            int c = rand() % 10;
            int d = rand() % 10;

            a = 10; // Weight 0

            b = 20; //WaterLvl 1

            c = 3; //Light 2

            d = 4;  //Moist 3

            log.setLog({ a,b,c,d });
        }

      auto  latest = log.getLatest(WEIGHT);

        std::cout << "get latest: " << latest << endl;


        auto weekLog = log.getWeek(WEIGHT);
```

```
        std::cout << "Week Log:" << std::endl;


        for (size_t i = 0; i < 7; i++)
        {
            int val = weekLog[i];

            cout << val << endl;
        }

        std::cout << "my month :" << endl;

        auto  mymonth = log.getMonth();

        for (size_t i = 0; i < mymonth.size(); i++)
        {
            int val = mymonth[i][0];

            std::cout << val << endl;
        }

       log.logClear();

       latest = log.getLatest(WEIGHT);


       std::cout << "get latest: " << latest << endl;

         mymonth = log.getMonth();

       for (size_t i = 0; i < mymonth.size(); i++)
       {
           int val = mymonth[i][0];

           std::cout << val << endl;
       }
    return 0;
}
```