OULUN AMMATTIKORKEAKOULU

Olli Kyllönen

VAAKALAITTEISTO TUOTANNON LAADUNVALVONTAAN

VAAKALAITTEISTO TUOTANNON LAADUNVALVONTAAN

Olli Kyllönen
Opinnäytetyö
Kevät 2024
Sähkö- ja automaatiotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Sähkö- ja automaatiotekniikan tutkinto-ohjelma, Automaatiotekniikka

Tekijä: Olli Kyllönen
Opinnäytetyön nimi: Vaakalaitteisto tuotannon laadunvalvontaan
Työn ohjaaja: Timo Heikkinen
Työn valmistumislukukausi ja -vuosi: Kevät 2024 Sivumäärä: 43

Tämän opinnäytetyön tarkoituksena oli suunnitella ja toteuttaa laadunvalvontalaitteisto elektroniikka- ja sähkölaitteita valmistavalle yritykselle. Työn taustalla oli laatuongelma, jossa vääriä kaapelikeriä oli päätynyt lopputuotteisiin joko alihankkijoilta tai työntekijöiden inhimillisten virheiden takia. Oikea kaapelikerä tunnistettiin hyödyntämällä ERP-järjestelmästä löytyvää massa- ja tuotenumerodataa.

Laitteistoon kuului Raspberry Pi -minitietokone, 7-tuuman kosketusnäyttö, viivakoodinlukija, lämpötulostin, vaakapääte ja vaakasilta. Ohjelma toteutettiin käyttämällä Electron-runkorakennetta. Työn suunnitteluvaiheessa otettiin huomioon loppukäyttäjät, ja tavoitteena oli luoda mahdollisimman selkeä ja helppokäyttöinen kokonaisuus. Työn toteutus alkoi ongelman tunnistamisella ja ratkaisumallien ideoimisella. Laitteiston valinnassa huomioitiin työn tavoitteiden täyttämisen lisäksi työympäristön rajoitteet. Kokonaisuudesta pyrittiin tekemään kompakti, ergonominen ja käyttäjäystävällinen. Käyttöliittymä suunniteltiin intuitiiviseksi ja visuaaliseksi prosessin helpon seurattavuuden vuoksi.

Ohjelmaa tehtäessä koodi suunniteltiin modulaariseksi ja loogiseksi. Erityistä huomiota kiinnitettiin logiikkaan ja virheenhallintaan, jotta prosessista saatiin jouheva ja luotettava. Työn keskeisenä tuloksena laadunvalvontalaitteisto tulee ratkaisemaan kaapelikerien laatuongelmia ja parantamaan tuotannon laatua. Jatkossa laitteistoa voidaan kehittää edelleen esimerkiksi konenäkötekniikan avulla, mikä voisi tarjota entistä tarkempaa laadunvalvontaa yhdessä vaakalaitteiston kanssa.

Asiasanat: automaatio, automaatiotekniikka, laadunvalvonta, sovelluskehitys

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Electrical and Automation Engineering, Option of Automation Engineering

Author: Olli Kyllönen
Title of thesis: Weighing system for production quality control
Supervisor: Timo Heikkinen
Term and year when the thesis was submitted: Spring 2024
Number of pages: 43

The purpose of this thesis was to design and implement a quality control system for a company manufacturing electronic and electrical devices. The project was initiated due to a quality issue where incorrect cable reels were ending up in the final products either from subcontractors or due to human error by employees. The correct cable reel was identified by utilizing weight and product number data from the ERP system.

The equipment included a Raspberry Pi mini-computer, a 7-inch touchscreen, a barcode reader, a thermal printer, a scale terminal, and a scale platform. The program was implemented using the Electron framework.

In the design phase, the focus was on the end users, with the goal of creating a clear and user-friendly system. The implementation began with identifying the problem and brainstorming solutions. The selection of equipment considered not only the project goals but also the constraints of the work environment. The aim was to create a compact, ergonomic, and user-friendly setup. The user interface was designed to be intuitive and visual to ensure easy process monitoring.
The code was designed to be modular and logical. Special attention was paid to logic and error handling to ensure the process was smooth and reliable.

As a key result, the quality control system is expected to resolve the issues with cable reels and improve production quality. In the future, the system can be further developed, for example, by integrating machine vision technology, which could provide even more precise quality control in conjunction with the scale system.

Keywords: automation, quality control, application development

SISÄLLYS

1. JOHDANTO 6
2. ELECTRON JS 7
    2.1 Sovelluksen luominen Electron forge -työkalulla 8
    2.2 Pääprosessi 9
    2.3 Render-prosessi 11
    2.4 Preload-skripti 11
    2.5 IPC-moduulit 12
3 VAAKALAITTEEN SUUNNITTELU 14
    3.1 Laitteiston valinta 15
    3.2 Minebea Intec vaakalaitteisto 15
    3.3 Raspberry Pi 4 Model B keskusyksikkö 16
    3.4 Raspberry Pi kosketusnäyttö 16
    3.5 Datalogic Gryphon 4500 viivakoodinlukija 17
    3.6 TSC TX600 lämpötulostin 18
    3.7 Vaakapäätteen yhdistäminen keskusyksikköön 19
    3.8 Lämpötulostimen yhdistäminen keskusyksikköön 24
4. OHJELMISTORAKENNE 28
    4.1 Punnitusmoduulin rakenne 29
5 KÄYTTÖLIITTYMÄSUUNNITTELU 31
    5.1 Vaakalaitteen käyttöliittymä 31
    5.2 Käyttöliittymän mukautuminen 32
    5.3 Vikatilanne-esimerkki 34
6 OHJELMAN TESTAUS 36
    6.1 Testaus- ja ongelmanratkaisumenetelmät 36
    6.2 Loppudemo 37
7 LOPPUPOHDINTAA 38
    7.1 Vaihtoehtoiset menetelmät 39
    7.2 Työn parantamismahdollisuuksia 40
    7.3 Työskentelytapojeni analyysi 41
LÄHTEET 42

1. JOHDANTO

Teollisuuden yritykset kehittävät jatkuvasti ratkaisuja tuotannon optimoimiseksi ja laadun varmistamiseksi. Automaatio mahdollistaa tasalaatuisen suorittamisen ja vapauttaa henkilöstöä muihin tehtäviin. Tässä työssä suunnitellaan ja toteutetaan laadunvalvontalaitteisto, joka integroidaan osaksi tuotantoprosessia eliminoimaan virheellisten kaapelikerien päätyminen lopputuotteisiin.

Työn tavoitteena on tunnistaa vääränlaiset kaapelikerät ERP-järjestelmästä saatavan massa- ja tuotenumerodatan avulla. Laitteistoon kuuluvat vaakasilta, vaakapääte, Raspberry Pi -minitietokone, kosketusnäyttö, viivakoodinlukija ja lämpötulostin. Ohjelmisto toteutetaan Electron-runkorakenteella, joka mahdollistaa työpöytäsovellusten kehittämisen JavaScriptillä, HTML:llä ja CSS:llä.

Käyttöliittymä suunnitellaan selkeäksi ja helppokäyttöiseksi, jotta operaattorit voivat käyttää laitteistoa tehokkaasti vähäisellä perehdytyksellä. Prosessin eri vaiheiden ja mahdollisten vikatilanteiden kartoittaminen on oleellista, ja käyttöliittymä suunnitellaan reagoimaan näihin tilanteisiin.

Opinnäytetyössä käsitellään myös ohjelman logiikan suunnittelu ja toteutus, erityisesti punnituslogiikka, jossa kaapelikerän massa verrataan ERP-järjestelmästä saatavaan tavoitemassaan. Tämä työ osoittaa, kuinka teknologioita ja ohjelmointiratkaisuja yhdistämällä voidaan luoda tehokkaita ja luotettavia laadunvalvontajärjestelmiä teollisuuteen.

Raportissa esitellään laitteisto, käyttöliittymä, Electron JS ja ohjelmistorakenne. ERP-järjestelmää, backendin yleistä toimintaa ja ergonomiasuunnittelua ei käsitellä tässä raportissa.

2. ELECTRON JS

Electron JS on avoimen lähdekoodin kehitysalusta, jonka avulla voidaan rakentaa työpöytäsovelluksia käyttäen JavaScriptiä, HTML:ää ja CSS:ää. Electronin suuri etu on sen järjestelmäriippumattomuus, joka mahdollistaa Windowsilla kehitetyn sovelluksen toimimisen esimerkiksi Linuxilla ja macOS:llä. Electron sisältää myös Node.js-integraation, joka mahdollistaa Noden moduulien suorittamisen suoraan prosessissa. Tämän ansiosta voidaan käyttää Noden kirjastoja, jotka ovat hyödyllisiä sovellusta kehittäessä, sillä ne mahdollistavat esimerkiksi laitteiden integroinnit.

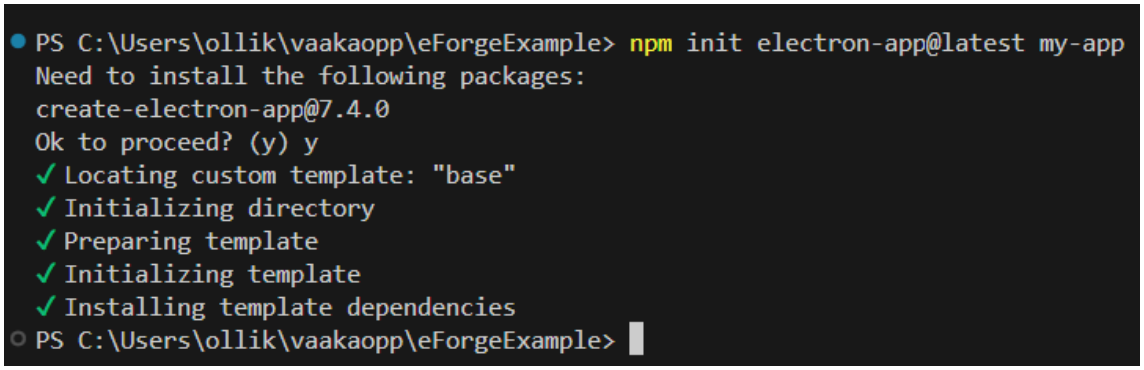
Electron-sovellus rakentuu kahteen prosessiin: pääprosessiin ja render-prosessiin. Pääprosessi on sovelluksen aloituspiste ja niitä voi olla vain yksi sovellusta kohden. Prosessi hallitsee koko sovelluksen elinkaarta, ikkunoiden hallintaa ja on yhteydessä natiiviin työpöytäympäristöön. Render-prosessi luodaan, kun pääprosessissa määritellään BrowserWindow-tapahtuma. Render-prosessi näyttää web-sisällön ja sen kehittämisessä tulee toimia web-kehittämisen standardien mukaan. Pääprosessin ja render-prosessin väliseen kommunikointiin on suositeltavaa käyttää preload-skriptiä, joka suojaa prosesseista lähtevää dataa.

Electronin monikäyttöisyyttä ja suorituskykyä havainnollistavat sillä tehdyt kuuluisat sovellukset, kuten Discord, Slack ja Spotify. Electronin haittapuolena on sen raskas rakenne. Jokainen render-ikkuna avataan omaan Chromium-ikkunaan, mikä kuluttaa paljon resursseja, erityisesti välimuistia. Laajempia sovelluksia kehitettäessä on hyvä ottaa huomioon tietokoneen riittävä suorituskyky. Electron-sovellukset ovat render-prosessin pakkauksen takia isompia tiedostokooltaan kuin natiivisti kehitetyt sovellukset. (1.)

2.1 Sovelluksen luominen Electron forge -työkalulla

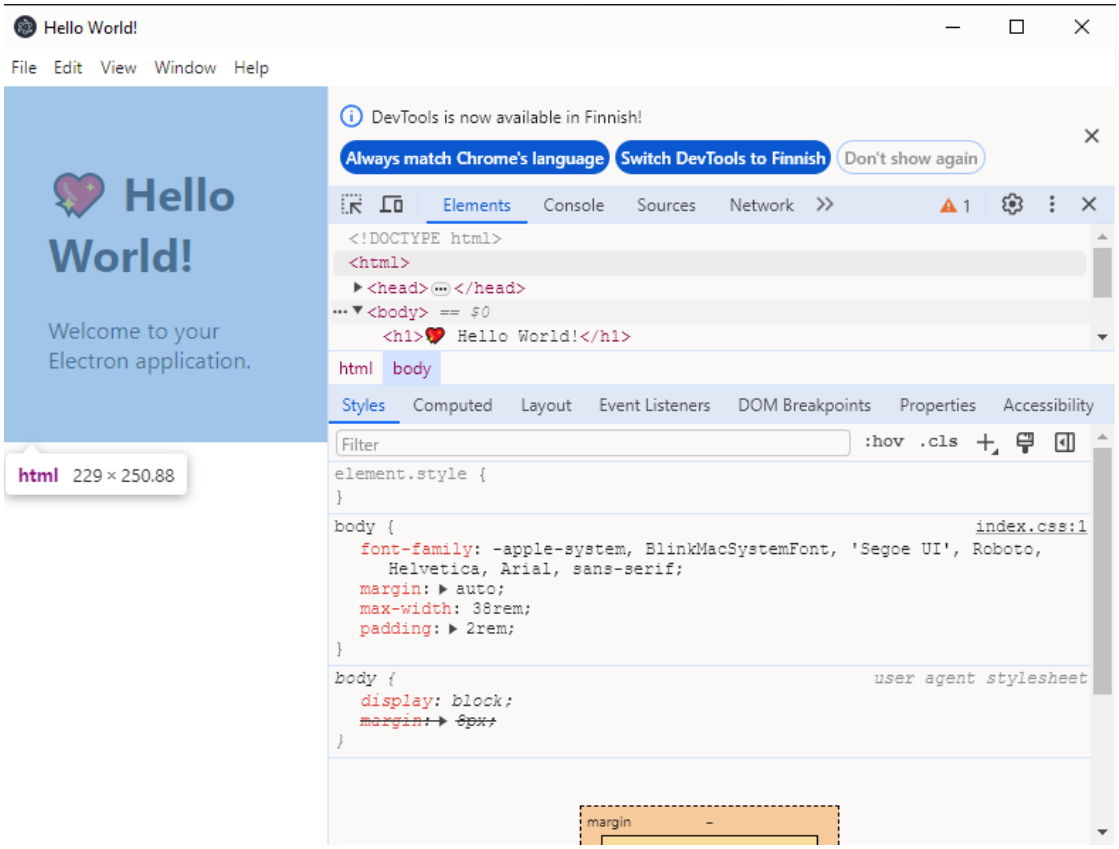
Electron-sovellus luodaan helpoiten käyttämällä Electron Forge -ohjelmistokehystä. Tämä kehys automatisoi sovelluspohjan rakentamisen, käsittelee resurssit ja riippuvuudet sekä paketoi ne yhteen tiedostoon. Se yhdistää useita erillisiä paketteja ja luo niille kehitysputken sovelluksen paketoinnista aina jakeluun asti.

Sovelluksen luominen Forge-työkalulla on suoraviivaista. Kuvassa 1 sovellus luodaan avaamalla terminaali tyhjässä kansiossa ja suorittamalla spesifin npm-skriptin, joka asentaa kaikki tarvittavat tiedostot automaattisesti sovelluskehitystä varten. Asennuksen yhteydessä voidaan päättää, halutaanko käyttää jotain templaattia. Esimerkiksi Webpack-templaatti automatisoi ja yksinkertaistaa JavaScript-koodin paketoimista, mikä voi olla monelle sovelluskehittäjälle ennestään tuttu ja täten hyödyllinen Electron-sovelluksessa.



KUVA 1. Electron-sovelluksen luominen Forge-työkalulla

Asennusesimerkissä ei ladattu mitään templaattia, vaan käytettiin peruskonfiguraatiota sovelluksen luomiseksi. Pakettien asennuksen jälkeen sovellus on valmis käynnistettäväksi komennolla npm run start. Kuvassa 2 tämä avaa tyhjän ikkunan, jonka jälkeen sovelluksen kehittäminen voidaan aloittaa. (2; 3; 4.)



KUVA 2. Electron Forgella luotu sovellus ensimmäistä kertaa käynnistettynä

2.2 Pääprosessi

Electronissa pääprosessin päätehtävä on luoda ja hallita sovellusikkunoita BrowserWindow-moduulin avulla. Jokainen yksittäinen BrowserWindow-instanssi luo sovellusikkunan, joka lataa verkkosivun erillisessä renderöintiprosessissa. Pääprosessi hallitsee myös render-prosesseja. BrowserWindow-tapahtuman tuhoamisen jälkeen myös render-prosessit lopetetaan.

Pääprosessi hallitsee sovelluksen elinkaarta, ja tämä tehdään yleensä käyttämällä app-moduulia. App-moduulilla voidaan määrittää, milloin render-prosessi avataan ja millä ehdoilla se voidaan sulkea. Moduuliin voidaan konfiguroida myös käyttöjärjestelmäkohtaisia ominaisuuksia. Esimerkiksi macOS:ssä ikkunoiden sulkemisen jälkeen prosessit voidaan tarvittaessa jättää taustalle päälle.

Ohjelmassa BrowserWindow-ikkunan konfiguraatio näkyy kuvassa 3. App-moduulin ready-tapahtuma ilmoittaa, kun Electron on latautunut ja valmis, minkä jälkeen ikkuna luodaan. Ikkunan avautuminen on määritetty sopivaksi käytettävälle näyttökoolle, ja se pakotetaan heti koko ruudul-

le. WebPreferences-osiossa poistetaan suora Node-moduulien integraatio, sillä niiden suora käyttö render-prosesseissa ei ole toivottua, koska se tekee koodista haavoittuvamman hyökkäyksille suoran Node.js rajapintaintegraation vuoksi. ContextIsolation on myös turvallisuustoimenpide, jolla eristetään preload.js-skripti erilliseksi render-prosesseista. Näiden turvallisuusasetusten jälkeen voidaan kommunikoida prosessissa lataamalla vielä preload.js-skripti.
Jos prosessin täytyy olla yhteydessä ulkoiseen laitteistoon, tämä toteutetaan myös pääprosessin kautta. (1.)

```
let mainWindow;

// Set up the main window when the app is ready
app.on('ready', () => {
    mainWindow = new BrowserWindow({
        width: 800,
        height: 480,
        fullscreen: true,
        webPreferences: {
            nodeIntegration: false,
            contextIsolation: true,
            preload: path.join(__dirname, 'preload.js'), // Use preload script for enhanced security
        },
    });

    // Load the HTML file into the window
    mainWindow.loadFile(path.join(__dirname, 'index.html'));
});
```

KUVA 3. Esimerkkikuva ohjelmasta ja browserWindow-ikkunan luomisesta pääprosessissa

Vaakalaitteessa tarvittiin lämpötulostinta tuotenumerotarran tulostamiseen, ja tässä kuvan 4 koodinäytteessä tulostin alustettiin kommunikoimaan prosessin kanssa. Tulostin yhdistettiin tietokoneeseen USB:n kautta, joten Noden USB-kirjasto tuotiin mukaan USB-moduulin tuontia varten.

```
// Initialize USB device
let device;
let interface;
try {
    // these vendor and device id's are also printer spesific.
    device = usb.findByIds(0x1203, 0x0232);
    device.open();
    interface = device.interfaces[0];
```

KUVA 4. Kuva lämpötulostimen alustamisesta prosessiin

2.3 Render-prosessi

Electron-sovelluksen render-prosessi vastaa käyttöliittymän näyttämisestä ja käyttäjän kanssa tapahtuvasta vuorovaikutuksesta. Render-ikkuna toimii Chromium-tekniikalla ja siinä voidaan ajaa HTML-, CSS- ja JavaScript-tiedostoja. Prosessi-ikkunan pohja luodaan HTML:ää käyttäen, joka on prosessin käynnistyspiste. CSS:llä tehdään ikkunan tyylimääritykset ja koko prosessi suorittaa JavaScriptiä.

Electronin Chromium-pohjaisuus näkyy muun muassa kehittäjätyökaluikkunassa, sillä se käyttää Chrome Developer Toolsia, joka on käytössä esimerkiksi Google Chrome -selaimessa. Tämä on sovelluskehittäjälle helpottava tekijä, sillä Chromiumin erittäin laajasti dokumentoidut kehittäjätyökalut voidaan avata suoraan Electron-prosessin render-ikkunassa.

Render-prosessista ei voida olla suoraan yhteydessä Electronin natiiveihin työpöytäominaisuuksiin tai Node.js:n moduuleihin. Tätä varten hyödynnetään Electronin rajapintaa ja IPC-kutsuja (Inter Process Communication). (1.)

2.4 Preload-skripti

Preload-skriptit ovat keskeinen osa Electronin arkkitehtuuria. Skripti suoritetaan ennen kuin render-prosessin web-sisältö alkaa latautua. Ne toimivat render-prosessina, mutta omaavat laajemmat oikeudet, kuten pääsyn Node.js:n rajapintaan, mikä ei olisi mahdollista normaalille render-prosessille. Tämän ansiosta sovelluskehittäjä voi altistaa turvallisesti halutut Node.js-toiminnot web-sovelluksen käyttöliittymään.

Preload-skripti ladataan prosessiin Electron-sovelluksessa pääprosessissa BrowserWindow-moduulin luomisen yhteydessä webPreferences-osiossa. Usein Node.js-integraatio on estetty ja käytetty kontekstieristystä. Nämä ovat oletusasetuksia Electron-prosesseissa. Kontekstieristys estää ei-toivottujen API:en pääsyn suoraan web-sisältöön. Sovelluksessa halutut API:t voidaan altistaa render-prosessille käyttämällä contextBridge-moduulia.

Kuvan 5 koodiesimerkissä tuodaan contextBridge- ja ipcRenderer-moduulit, joilla voidaan toteuttaa turvallinen ja asynkroninen viestintä render-prosessista pääprosessiin ja päinvastoin. Tämän jälkeen määritellään contextBridge-funktio, jolla määritetään objekti, joka altistetaan render-prosessille. Näiden määrittelyiden jälkeen voidaan lisätä useita haluttuja skriptejä altistettavaksi osaksi electronAPI:a. (1.)

```
const { contextBridge, ipcRenderer } = require('electron');

contextBridge.exposeInMainWorld('electronAPI', {
    onWeightUpdate: (callback) => ipcRenderer.on('weightUpdate', (event, data) => callback(data)),
```

KUVA 5. ContextBridge:llä määritetty JavaScript-objekti preload.js tiedostossa, joka altistetaan render-prosessille

2.5 IPC-moduulit

IPC (Inter Process Communication) on yksi laajimmin käytetyistä menetelmistä viestiä eri prosessien välillä Electronissa. Pää- ja render-prosessien eroavaisuuksien vuoksi IPC on ainut tapa tehdä useita perusohjelmointitehtäviä, kuten muuttaa web-ikkunan sisältöä. IPC-moduulilla voidaan hoitaa viestintä pää- ja render-prosessin välillä joko synkronisesti tai asynkronisesti.

Electronissa on kaksi komponenttia IPC-viestintään: ipcMain ja ipcRenderer. Kuten nimistä voi päätellä, pääprosessin komponentti on ipcMain ja render-prosessin ipcRenderer. IpcMain vastaanottaa viestejä render-prosesseista ja toteuttaa haluttuja toimintoja niiden pohjalta, esimerkiksi käskyn ulkoiselle laitteelle.

Tässä esitetään esimerkki prosessista, jossa käytetään IPC-viestintää, jotta pääprosessi saa tiedon render-prosessilta viivakoodien täsmäyksen onnistumisesta.

Kuvassa 6 render-prosessissa käytetään sendBarcodesMatched-funktiota BarcodesMatched-tapahtuman lähettämiseksi.

```
window.electronAPI.sendBarcodesMatched();
```

KUVA 6. Render-prosessista lähtevä BarcodesMatch-tapahtuma

Kuvassa 7 turvallisuussyistä rajatun tiedonsiirron takia tieto ei voi mennä suoraan pääprosessiin, vaan tieto kulkee preload-skriptin kautta.

```
onBarcodesMatched: (callback) => ipcRenderer.on('barcodesMatched', (event, ...args) => callback(...args)),
sendBarcodesMatched: () => ipcRenderer.send('barcodesMatched'),
```

KUVA 7. Preload.js:n määritelty BarcodesMatched -tapahtuma

Preload-skriptissä annetaan prosessille lupa kuunnella onBarcodesMatched-tapahtumaa. Alemmassa skriptissä lähetetään send-komennolla barcodesMatched -tapahtuma pääprosessiin. Nyrkkisääntönä voidaan pitää sitä, että kun jokin funktio on määritelty preload-skriptissä, se on käytössä koko prosessissa.

Kuvassa 8 pääprosessi kuuntelee preload-skriptissä määriteltyä barcodesMatched-tapahtumaa ipcMain-funktiolla, jolloin data saadaan onnistuneesti ja turvallisesti tuotua render-prosessista pääprosessiin. (1.)

```
// IPC event for barcode matches
ipcMain.on('barcodesMatched', () => {
    console.log('Barcodes matched event received');
    mainWindow.webContents.send('barcodesMatched');
});
```

KUVA 8. Main.js kuuntelija barcodesMatched -tapahtumalle

3 VAAKALAITTEEN SUUNNITTELU

Tuotantoprosessin viimeisessä vaiheessa asennettava kaapelikerä on osa yrityksen laajempaa laitekokonaisuutta. Laadunvalvontamenetelmät otettiin tarkasteluun, kun tuotantoprosessissa havaittiin toistuvasti vääränlaisten kaapelikerien päätyminen lopputuotteeseen. Väärien kaapelikerien syitä olivat muun muassa huolimattomuus, prosessin epäjohdonmukaisuus sekä samankaltaiset, mutta eri ominaisuuksia omaavat kaapelit. Lisäksi alihankkijat toimittivat ajoittain virheellisiä kaapelikeriä. Asiakkaalle asti päätyneiden virheellisten kaapelikerien vaihtaminen aiheutti merkittäviä kustannuksia. Tehokkaampi laadunvalvonta vähentäisi näitä ongelmia, parantaisi asiakastyytyväisyyttä ja toisi taloudellisia säästöjä.

Ratkaisumalleja arvioitaessa havaittiin, että kaapelin massan hyödyntäminen olisi tehokas menetelmä kaapelityyppien erotteluun. Tässä lähestymistavassa jokaiselle kaapelityypille määriteltäisiin standardimassa, ja ohjelma asettaisi toleranssirajat, joiden sisään oikeanlaisen kaapelikerän tulisi mahtua. Mikäli kaapelikerän massa ylittäisi tai alittaisi nämä toleranssit, ohjelma hylkäisi kaapelin automaattisesti. Tämä menetelmä tarjoaa luotettavan ja automatisoidun tavan varmistaa kaapelityyppien oikeellisuus tuotantoprosessissa.

Pelkän massan perusteella tapahtuva tunnistaminen osoittautui riittämättömäksi, koska eri valmistajien kaapelikerät saattavat olla samankaltaisia massaltaan. Tunnistamisprosessia varten integroitiin käyttöön yrityksen tietokannassa oleva data, joka sisältää kaapeleiden osanumerot. Osanumero on uniikki jokaiselle kaapelityypille, ja se mahdollistaa yksiselitteisen tunnistamisen yhdessä massavertailun kanssa.

Tuotteiden tunnistamisessa yrityksen logistiikassa käytetään arvokilpiä, jotka sisältävät tuotteiden osanumerot viivakoodimuodossa. Samaa menetelmää sovelletaan kaapelikerälavoissa, joissa kullekin kaapelille on luettavissa viivakoodi, joka ilmaisee yksilöllisen osanumeron.

Ennen massavertailua toteutettiin viivakoodivertailu, jossa lopputuotteen kyljestä luetaan viivakoodi, jonka avulla haetaan ERP-järjestelmästä dataa, kuten tuotteeseen kuuluvan kaapelikerän osanumero. Saatua osanumeroa verrataan kaapelikerälavan kyljessä olevaan osanumeroon. Näiden täsmätessä voitiin siirtyä vertailemaan massaa.

Onnistuneen osanumerojen täsmäyksen ja massavertailun jälkeen laitteiston lämpötulostin tulostaa tuodun osanumeron tarrana, joka liimataan kiinni kaapelikerään.

Prosessin tekniset vaatimukset määriteltiin selkeäksi tässä vaiheessa. Tunnistusprosessiin tarvittiin vaakakokonaisuus massan mittaamiseen, keskusyksikkö ohjelman ajamiseen, viivakoodinlukija kaapelikerien tunnistamiseen ja lämpötulostin osanumerotarran tulostamiseen.

3.1 Laitteiston valinta

Laitteiston valinnassa otettiin huomioon useita tekijöitä, kuten helppokäyttöisyys, ergonomisuus, tilankäyttö ja budjetti. Tavoitteena oli integroida vaakalaitteisto saumattomasti nykyisiin tuotantoolosuhteisiin. Tästä syystä laitteiston tuli olla kiinteä osa tuotantoprosessia, ei irrallinen yksikkö, joka hankaloittaisi tuotantoa.

3.2 Minebea Intec vaakalaitteisto

Kaapeleiden massan mittaukseen valittiin Minebea Intecin teollisuusvaakasilta ja Midrics 1 -vaakapääte. Vaaka valittiin sen tarkan, kestävän ja toistettavan suorituskyvyn perusteella, jotka ovat kriittisiä laadunvalvonnassa, missä marginaalit ovat pieniä. Vaakapäätteen kotelo on IP65-luokiteltu ja valmistettu ruostumattomasta teräksestä, mikä tekee siitä ihanteellisen teollisuuskäyttöön. Vaaka kommunikoi rs-232-väylän kautta. Vaakasilta on kooltaan 1 m x 1 m, IP67-luokiteltu ja varustettu säädettävillä kumitassuilla, jotka pitävät sen paikallaan. Vaakasillan ja vaakapäätteen välillä on 6 metrin kaapeli, joka tuo asetteluvapautta. Vaakapäätteen sisäiset ohjelmistot ja säädettävät parametrit mahdollistavat laitteen hienosäädön tuotantoprosessin vaatimuksiin. Kuvassa 9 on vaakapääte ja vaakasilta. (5.)

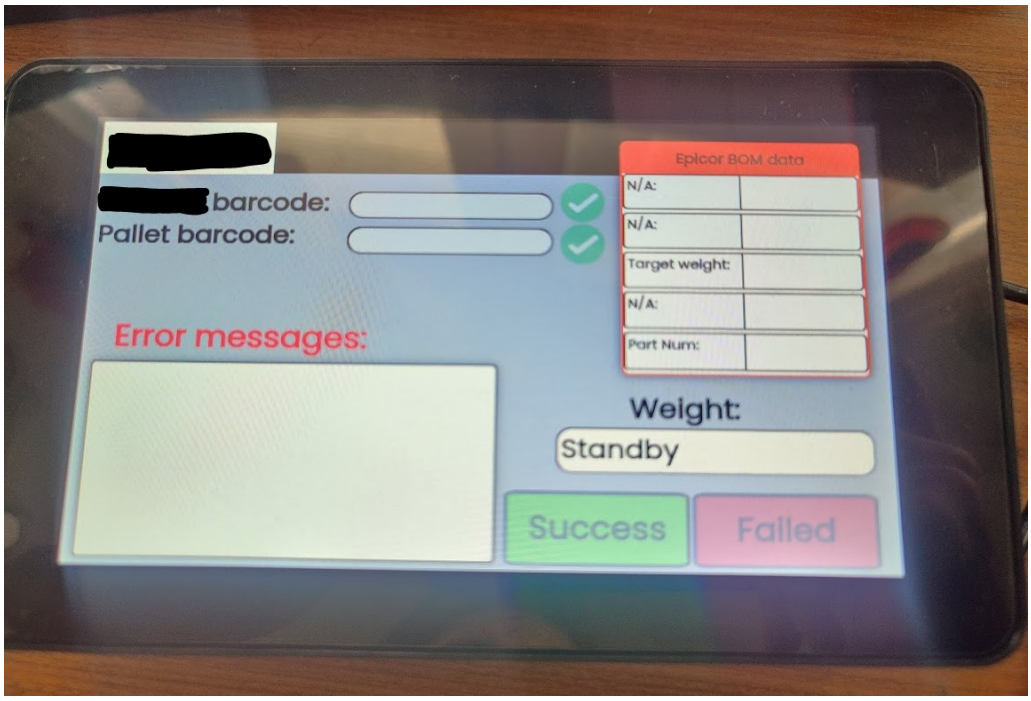
KUVA 9 Minebea intec lattiavaaka ja vaakapääte (5)

3.3 Raspberry Pi 4 Model B keskusyksikkö

Laitteiston ohjaukseen käytetään Raspberry Pi 4 Model B:tä, joka on Linux-pohjainen minitietokone. Alun perin opetuskäyttöön suunniteltu Raspberry Pi on nykyisin suosittu elektroniikkaharrastajien ja projektikehittäjien keskuudessa sen joustavuuden ja kustannustehokkuuden ansiosta. Uusimpien versioiden suorituskyky riittää vaativampiinkin projekteihin, ja sen laaja käyttäjäyhteisö tarjoaa runsaasti tukimateriaalia ja dokumentaatiota mahdollisten ongelmien ratkaisemiseksi.
Raspberry Pi:n erikoisuus GPIO-pinnit (General Purpose Input/Output) mahdollistavat laajat mukautusmahdollisuudet, joiden avulla voidaan ohjata eri komponentteja ja yhdistää lisälaitteita. (6; 7.)

3.4 Raspberry Pi kosketusnäyttö

Käyttöliittymän näyttämiseen laitteistossa käytetään Raspberry Pi:n omaa kosketusnäyttölisäosaa. Valittu näyttö on 7 tuuman kokoinen, ja sen resoluutio on 800x480 pikseliä.
Näyttö ei vaadi erillistä virtalähdettä, sillä se saa virtansa suoraan Raspberry Pi:ltä. Näytön liittäminen Raspberry Pi:hin tapahtuu käyttämällä sekä lattakaapelia että GPIO-pinnejä. Lattakaapeli huolehtii kuvasignaalin siirrosta, kun taas GPIO-pinnit vastaavat näytön virransaannista.
Kuvassa 10 näkyy projektiin hankittu kotelo, johon sai kätevästi näytön ja keskusyksikön samaan pakettiin ilman roikkuvia johtoja. (6.)



KUVA 10 Raspipaketti kotelossa käyttöliittymänäkymässä

3.5 Datalogic Gryphon 4500 viivakoodinlukija

Laadunvalvonnan ensimmäinen vaihe toteutettiin Datalogic Gryphon 4500 -viivakoodinlukijalla, joka on langaton ja pystyy lukemaan myös QR-koodeja. Langattomuus on pakollista prosessille, sillä koodeja tulee lukea jopa kymmenen metrin etäisyydeltä. Lukijan integrointi laitteistoon oli suoraviivaista, sillä se toimii näppäimistön tavoin, emuloiden näppäinpainalluksia. Kuvassa 11 tuotekuva lukijasta. (8.)

KUVA 11 Datalogic gryphon 4500 viivakoodinlukija (8)

3.6 TSC TX600 lämpötulostin

Lämpötulostinta käytettiin osanumerotarrojen tulostamiseen laadunvalvontaprosessin päätteeksi. Tämä tulostustekniikka on ihanteellinen, koska se ei vaadi värijauhetta, mustetta tai värinauhoja, mikä minimoi huoltotarpeen. Kuvassa 12 oleva TSC TX600 lämpötulostin tuottaa tarkkoja tulosteita, jopa 600 dpi:n tarkkuudella, mikä tekee siitä sopivan vaativiin tuotemerkintöihin ja toimitusetikettien luomiseen. Tulostin tukee useita kommunikaatioväyliä, kuten Ethernetiä ja Bluetoothia mahdollistaen monet eri käyttötarkoitukset.

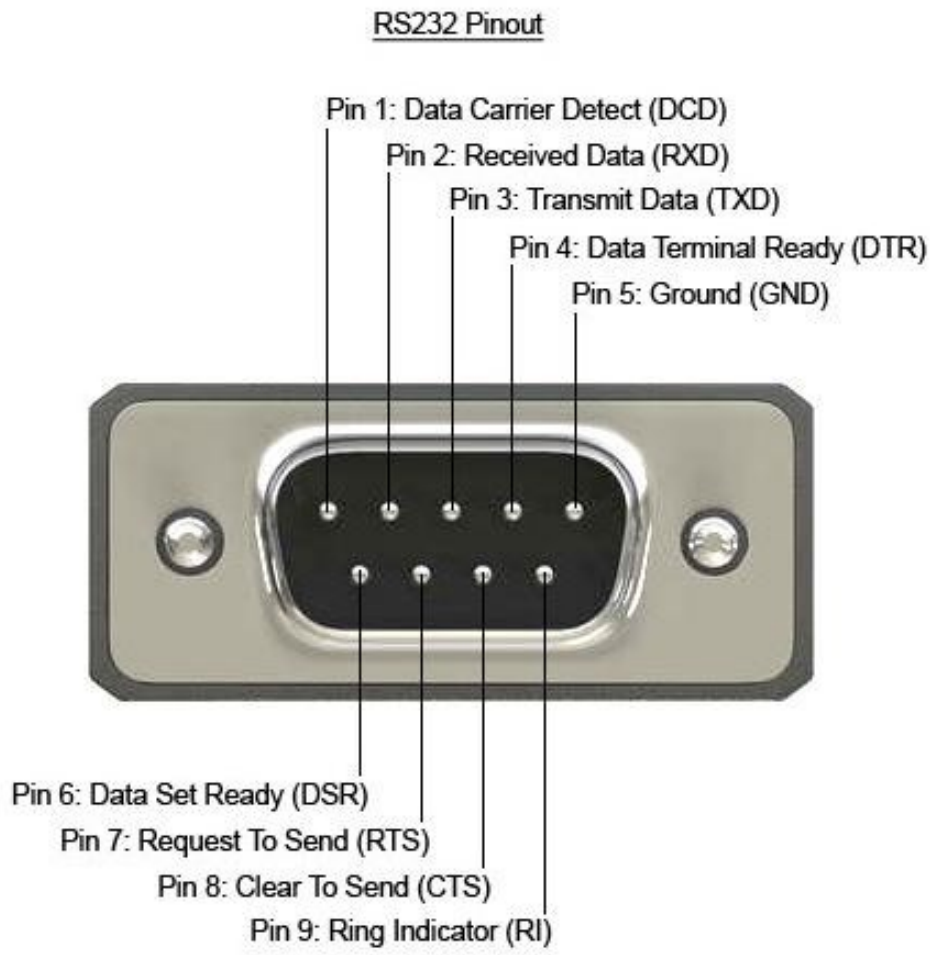
Erityispiirteenä tulostimessa on TSC:n kehittämä TSPL (TSC Printer Language), ohjelmointikieli, joka mahdollistaa laajat mukautukset tulostusasetuksiin. TSPL:n avulla voidaan säätää muun muassa tarran kokoa, fonttityyppiä ja tekstin sijaintia, mikä tekee tulostimesta joustavan työkalun erilaisiin merkintätarpeisiin. (9; 10.)

KUVA 12 TSC TX600 lämpötulostin

3.7 Vaakapäätteen yhdistäminen keskusyksikköön

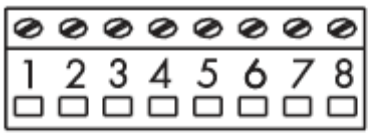
Vaakapäätteen yhdistämisessä keskusyksikköön kohdattiin useita haasteita. Vaakapäätteen mukana ei toimitettu valmista RS-232-kaapelia, joten sellainen täytyi valmistaa itse. Yrityksen laboratoriosta löytynyt ylijäämäinen sarjaliikennekaapeli muokattiin laitteistoon sopivaksi. Vaa'an dokumentaatiossa ei ollut selkeitä ohjeita kaapelin kytkemiseen, joten kokeiltiin eri vaihtoehtoja ja pääteltiin parhain menetelmä.
Kuvassa 13 näkyy RS-232-sarjaliikennekaapelin pinnidiagrammi. Aluksi oletettiin, että lähes kaikkia pinnejä tulisi käyttää onnistuneen yhteyden saavuttamiseksi, mutta lopulta havaittiin, että vain kolme pinniä riitti kommunikaation varmistamiseksi vaa'an ja tietokoneen välillä.

Prosessissa käytettiin pinnejä 2 (Received Data, RXD), 3 (Transmit Data, TXD) ja 5 (Ground, GND). RXD-pinnin kautta vastaanotetaan dataa vaa'alta. TXD-pinnin kautta lähetetään dataa vaa'alle. GND-pinni toimii maadoituspinninä.



KUVA 13. RS232 sarjaliikenneväyläkaapelin pinnidiagrammi (11)

Kuvassa 14 näkyvät vaa'an piirikortin päälle lisätyn RS-232-väylän reppukortin liitännät. Liitännöistä käytettiin pinnejä 5, 7 ja 8. Kaapeleita yhdistettäessä tulee huomioida lähtö- ja tulologiikka. Kun data lähtee vaa'alta pinnistä 8, se tulee yhdistää väyläkaapelin RXD-pinniin. Jos datansiirtojohtimet yhdistetään epähuomiossa samannimisiin pinneihin, tieto ei kulje oikein.



Pin 1: +12 V: Supply voltage for Minebea Intec printers
Pin 2: Reset_Out (peripheral device restart)
Pin 3: +5 V Out
Pin 4: Ground (GND)
Pin 5: Clear to send (CTS)
Pin 6: Data terminal ready (DTR)
Pin 7: Data input (R×D)
Pin 8: Data output (T×D)

KUVA 14. Vaakapäätteen reppukortin pinnidiagrammi (12)

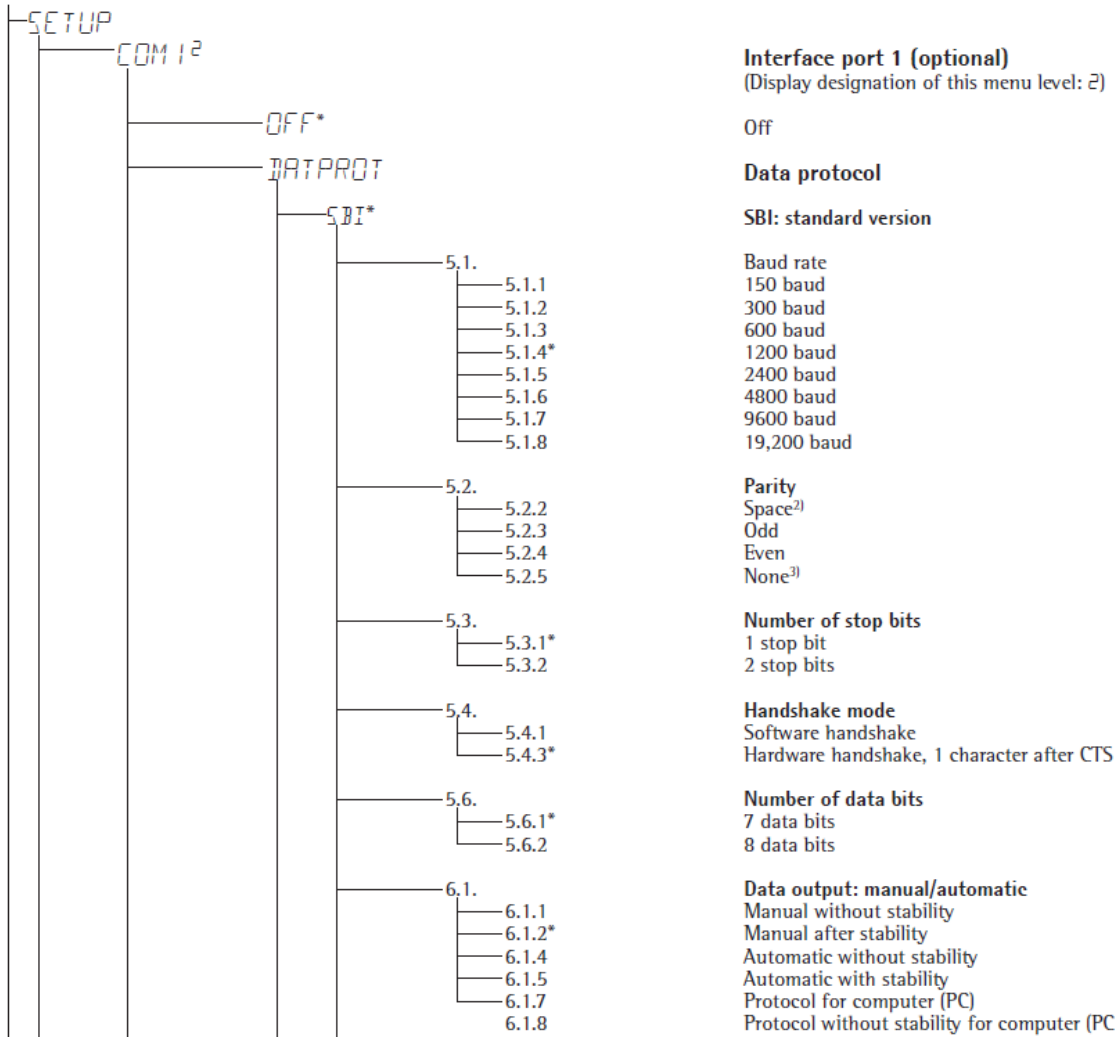
Kun vaakapäätteen fyysiset kytkennät oli suoritettu (kuva 15), siirryttiin luomaan yhteyttä tietokoneeseen. Haluttiin varmistaa, että vaaka ja tietokone todella kommunikoivat keskenään ennen jatkotoimenpiteitä, joten avattiin yksinkertainen PuTTY-ikkuna vaa'an tulosteen lukemiseksi. Ennen ikkunan avaamista konfiguroitiin vaa'an ja tietokoneen asetukset onnistuneen kättelyn varmistamiseksi.

KUVA 15. Kuva vaakapäätteen kytkennöistä, ylhäällä keskellä RS-232-väylän kytkennät

Kuvassa 16 näkyvistä vaa'an sisäisistä asetuksista konfiguroitiin siirtonopeus 9600 baudiin, käytettiin yhtä pysäytysbittiä, ohjelmistokättelyä, 8:aa databittiä ja asetettiin datan tuloste automaattiseksi ilman vakautta, jolloin vaaka syöttää tulostetta jatkuvasti. Nämä asetukset täytyi asettaa samoiksi tietokoneen laitehallinnan porttiasetuksissa. Asetusten täsmätessä kättely onnistui ja laitteet pystyivät kommunikoimaan keskenään.

Vaa'an tuloste saatiin näkymään PuTTY-ikkunassa, joten vaakaa voitiin alkaa integroimaan ohjelmaan. Prosessiin integroitaessa havaittiin heti, että vaa'an tuloste tulisi siistiä ohjelmallisesti,

jotta dataa voitaisiin hyödyntää. Tulosteessa oli ylimääräisiä välilyöntejä, plus- ja "kg"-merkki, jotka olisivat aiheuttaneet sellaisenaan ongelmia.



KUVA 16. Ote vaa'an manuaalista, sarjaportin asetuksista (12)

Kuvassa 17 käsitellään vaa'an tulostetta. Vaa'an käsittelyyn Pythonilla tarvitaan tapauksen mukaan erilaisia kirjastoja; tässä tapauksessa käytettiin json-, sys- ja serial-kirjastoja. Portti, jota kuunnellaan, määritellään, ja tämä voidaan varmistaa tietokoneen laitehallinnasta, josta nähdään mihin porttiin vaaka on yhdistetty. Lisäksi tiedonsiirtonopeus asetetaan 9600 baudiin.

Koodissa vaa'an tuloste siistitään siten, että jokainen rivi dekoodataan UTF-8-merkistöksi ja poistetaan ylimääräiset välilyönnit tai rivinvaihdot. Sen jälkeen etsitään kg- ja plusmerkit, erotetaan massadata, muunnetaan se liukuluvuksi ja JSON-muotoon. Tämän käsittelyn jälkeen tuloste on siistitty ja se voidaan lähettää pääprosessiin stdout-virran kautta. Data muutetaan JSON-muotoon

tiedonkäsittelyn helpottamiseksi. JSON on kevyt dataformaatti, joka on tuettu natiivisti myös JavaScriptissä, johon tämän tiedon käsittely siirtyy.

Raa'an vaakadatan kaikki käsittely ohjelmallisesti tapahtuu tässä samassa moduulissa. Weight-avaimen sisältämää dataa voidaan sitten helposti käsitellä muissa prosesseissa. Jos tietokone tai vaakalaite muuttuu, on helppo palata samaan moduuliin muuttamaan vaa'alle spesifejä asetuksia muun konfiguraation pysyessä samana.

```python
PORT = '/dev/ttyUSB0'  # Serial port to listen on
BAUDRATE = 9600  # Communication speed via serial port

# Attempt to open the serial port
try:
    ser = serial.Serial(PORT, BAUDRATE, timeout=1)  # Open serial port with a timeout of 1 second
    print("Listening on serial port:", PORT)  # Inform the user that the script is listening on the specified port

    while True:  # Infinite loop to continuously read from the serial port
        try:
            # Read a line from the serial port
            line = ser.readline().decode('utf-8').strip()  # Decode bytes to string and strip trailing newlines/spaces

            # Check if the received line contains valid data
            if line:
                # Find the start and end indices of the weight value in the line
                weight_start = line.find('+')  # Locate the start of the weight
                weight_end = line.find('kg', weight_start)  # Locate the end of the weight

                # Ensure both '+' and 'kg' were found
                if weight_start != -1 and weight_end != -1:
                    # Extract the weight string and convert it to a float
                    weight_str = line[weight_start + 1:weight_end].strip()  # Extract the weight value string
                    weight = float(weight_str)  # Convert the weight string to a floating point number

                    # Create and send JSON formatted weight data
                    json_data = json.dumps({'weight': weight})  # Convert the dictionary to a JSON string
                    sys.stdout.write(json_data + '\n')  # Output the JSON string followed by a newline
                    sys.stdout.flush()  # Ensure the data is sent immediately

        except serial.SerialException as e:
            print("Serial port error:", e)  # Handle errors related to the serial port
            break  # Exit the loop if there is a serial port error
```

KUVA 17. Ote vaa'an koodista

3.8 Lämpötulostimen yhdistäminen keskusyksikköön

Tulostinta alettiin yhdistämään prosessiin vasta työn loppuvaiheilla, kun logiikka oli jo melkein kokonaan valmis. Tulostimen integraatiossa oli aluksi selvitettävä tulostimen ominaisuudet, kuten liitettävyys ja tarran konfiguraatiomahdollisuudet. Onneksi tulostimesta oli laajasti dokumentaatiota valmistajan sivuilla.

Eri liitäntätavoista päädyttiin käyttämään USB:tä, koska se on laajasti tuettu ja soveltui prosessin käyttötarkoitukseen.

Tulostimen ollessa fyysinen laite sen kättely käsiteltiin Electronin pääprosessissa. Koska käytössä oli USB-väylä, Noden USB-kirjasto tuotiin mukaan, jotta kommunikointi voisi toimia. Tulostimen määrittämiseen ei riittänyt pelkkä portin kutsunta, vaan laitespesifit toimittaja- ja laite-ID:t piti etsiä. Tässä tapauksessa niitä ei ollut määritelty laitteen arvokilvessä, joten ne haettiin laitehallinnan "ominaisuudet"-valikon kautta.

Tulostimen kanssa kommunikaatiossa ilmeni käyttöjärjestelmäkohtaisia eroja. Lähes kaikki kehitystyö ja testaus tehtiin Windows-pohjaisella tietokoneella, mutta koska laitteisto oli Linux-pohjainen, kohdattiin ongelmia, jotka paljastivat eroja näiden kahden käyttöjärjestelmän välillä. Tulostinta testattaessa Raspberry Pi:llä huomattiin, että Windowsille tehty koodi ei toiminut suoraan Linuxilla. Laitteistoa ei pystytty alustamaan oikein. Vikakoodien ja dokumentaation perusteella päädyttiin ratkaisuun, jossa Linuxin kernelin ajurit irrotettiin (Kuva 18.). Tällä toimenpiteellä saavutettiin tilanne, jossa laitteen hallintaan pakotettiin käyttämään Noden usb-kirjaston ajureita eikä Linuxin omia. Tämä toimenpide ratkaisi ongelman.

Tulostimen tulostama tarra konfiguroitiin käyttämällä valmistajan TSC:n omaa TSPL-kieltä. Kieli sisältää lukuisia komentoja, joiden avulla voidaan tulostaa teollisuuskäyttöön lähes minkälaisia tarroja tahansa. Tässä työssä tarvittiin vain kaapelikerän osanumeron tulostus, joten syvempi käyttö jää demonstroimatta.

```
// Initialize USB device
let device;
let interface;
try {
    // these vendor and device id's are also printer spesific. Will
    device = usb.findByIds(0x1203, 0x0232);
    device.open();
    interface = device.interfaces[0];
    // kernel detaching is Linux spesific. No need for it if using W
    if (interface.isKernelDriverActive()) {
        try {
            interface.detachKernelDriver();
        } catch (error) {
            console.error('Failed to detach kernel driver:', error);
        }
    }
    interface.claim();
} catch (error) {
    console.error('USB device initialization failed:', error);
}
```

KUVA 18. Vaa'an alustus pääprosessissa

Tässä koodissa on komentosarja, jolla tulostetaan kaapelikerän osatarra prosessin päätteeksi (Kuva 19.). Komennot toimivat seuraavasti: SIZE-komennolla määritetään etiketin koko. Kaikki numerokomennot ovat tuumia, ellei toisin määritellä, eli tässä tapauksessa tarran kooksi asetetaan 2 tuumaa x 2 tuumaa. CLS-komento tyhjentää tulostimen puskurin ennen seuraavaa etikettiä, jotta aiempaa dataa ei jää jäljelle. DIRECTION-komento asettaa tulostussuunnan; tässä tapauksessa ”0” tarkoittaa, että tulostus tapahtuu normaalissa suunnassa, ei käännettynä. REFERENCE-komennolla määritetään referenssipiste, jonka mukaan tekstin positio voidaan määrittää suhteessa X- ja Y-akseliin. Arvo ”0” tarkoittaa, että referenssipiste sijaitsee vasemmassa yläkulmassa. OFFSET-komennolla määritetään, että etiketti on oikeassa kohdassa tarran leikkausta varten. Jokin arvo on pakko asettaa leikkaustilassa, sillä muuten aiheutuu ”paper jam” -virhekoodi. TEXT-komennolla luodaan tulostettava teksti. Tässä tapauksessa teksti on asetettu referenssipisteen suhteen koordinaatteihin x=100 ja y=200. ”7” on käytetty fontti ja fontin suunta on ”0” (normaali). ”6” ja ”6” ovat koon kertoimia leveys- ja korkeussuunnassa. Lopuksi tulostetaan teksti. PRINT-komennon arvolla 1 tulostetaan yksi tarra kerrallaan. CUT-komennolla leikataan tarra komentosarjan jälkeen.

```
let tsplCommand = '';
tsplCommand += 'SIZE 2,2\n';
tsplCommand += 'CLS\n';
tsplCommand += 'DIRECTION 0\n';
tsplCommand += 'REFERENCE 0,0\n';
tsplCommand += 'OFFSET 0.00 mm\n';
tsplCommand += `TEXT 100,200,"7",0,6,6,"P/N:${PartPartnumber}"\n`;
tsplCommand += 'PRINT 1,1\n';
tsplCommand += 'CUT\n';
let buffer = Buffer.from(tsplCommand, 'utf-8');
```

KUVA 19. Tarran luonti TSPL-kielellä

Näin luodaan yksinkertainen tarra TSPL-kielellä lämpötulostimelle. Ongelmia tuottivat eri komentojen väliset riippuvuudet. Esimerkiksi ilman CLS-komentoa tulostin ei suostunut tulostamaan ollenkaan, ja oli hankala selvittää, mistä ongelma johtui. Tätä esimerkkiä on helppo jatkojalostaa uusilla TSPL-komennoilla, jos syntyy tarve esimerkiksi QR-koodin luomiseen. (13.)

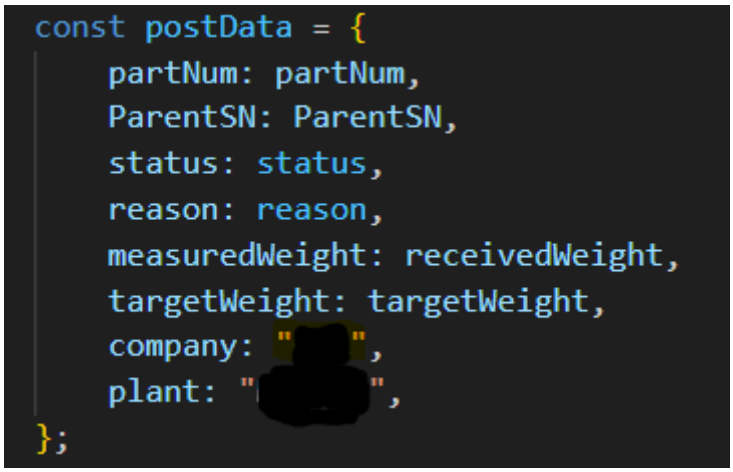
4. OHJELMISTORAKENNE

Ohjelmistorakennetta suunniteltaessa otettiin huomioon lukuisia tekijöitä. Prosessin tuli edetä selkeässä järjestyksessä, jotta operaattori voi vähäiselläkin perehdytyksellä ymmärtää sen toiminnan. Vikatilanteet tuotiin esiin käyttöliittymään selkeästi ja prosessi pysähtyi virhetilanteessa. Mahdollisia tilanteita, joissa prosessi ei etene toivotulla tavalla, oli useampia: viivakoodille ei löydy tietoa, backend on pois käytössä, viivakoodit eivät täsmää, vain muutaman esimerkin mainitakseni.

Prosessi alkaa työvaiheesta, jossa lopputuotteen kyljestä viivakoodin skannauksen jälkeen vastaava kaapelikerä etsitään ja kaapelilavan kyljessä oleva viivakoodi skannataan. Kaapelilavan viivakoodin ja ERP-järjestelmästä haetun osanumeron koodin tulisi täsmätä. Jos koodit täsmäävät, laadunvalvonnan ensimmäinen osa on suoritettu.

Seuraavassa vaiheessa kaapelikerä asetetaan vaakasillalle, ja ohjelma vertaa kaapelikerän massaa ERP-järjestelmästä tuotuun tavoitemassaan. Massalle on asetettu ohjelmallisesti toleranssi, jonka sisään mitattavan kaapelin tulee asettua suhteessa tavoitemassaan. Kaapelikerän ollessa hyväksytyissä rajoissa punnitus hyväksytään, ja lämpötulostin tulostaa tarran, jossa on kaapelin osanumero. Laadunvalvontaprosessi on tällöin suoritettu. Osanumerotarra toimii myös eräänlaisena laadunvalvontalappuna, sillä jokaisesta kaapelikerästä, josta se puuttuu, voidaan päätellä, että tämä työvaihe on laiminlyöty.

Jokaisesta punnituksesta kerätään dataa, joka lähetetään POST-metodilla yrityksen tietokantaan (kuva 20). Dataa voidaan tarkastella myöhemmin mahdollisissa ongelmatilanteissa, ja laadunvalvonnan kannalta oleellinen data on saatavilla. Lähetettävä data kattaa muun muassa punnitusstatuksen, osanumeron ja tavoitemassan.



KUVA 20. Prosessin jälkeinen lähetettävä data

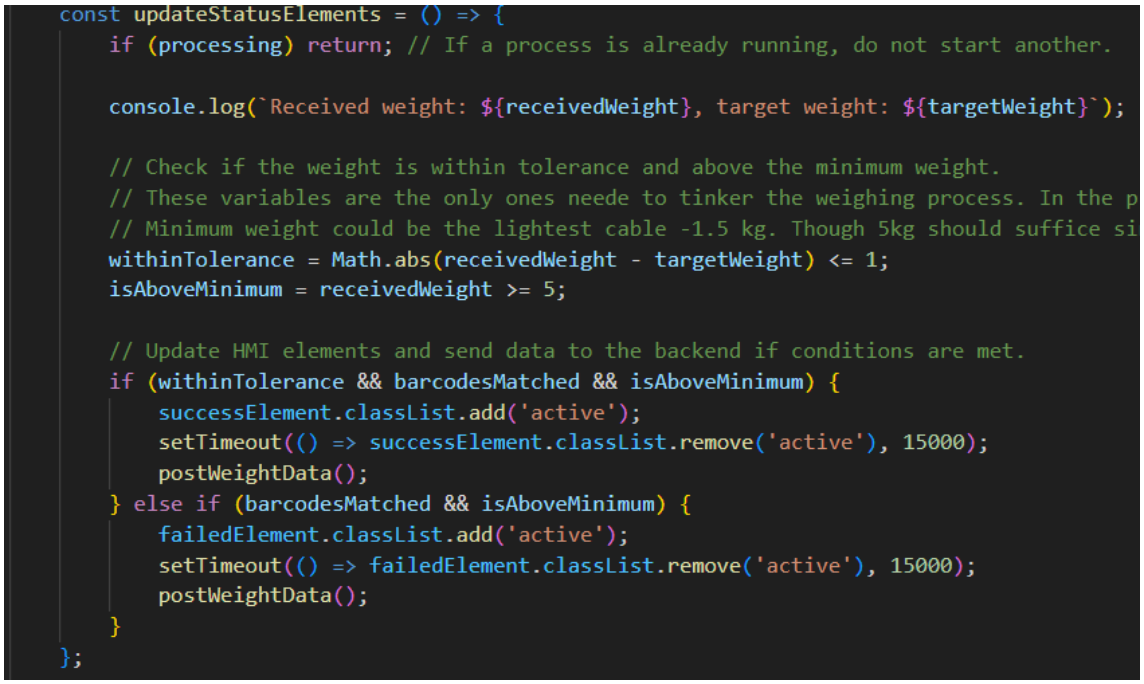
4.1 Punnitusmoduulin rakenne

Ohjelman jokaista moduulia ei ole mielekästä käydä läpi tämän opinnäytetyön puitteissa, mutta esimerkkinä esitellään, miten kaapelikerän massan punnituslogiikka on toteutettu.

Kuvan 21 näyte on osa punnituslogiikan moduulia, jossa toteutetaan kaapelin massan vertailu. Näytteen "updateStatusElements" -funktio sisältää myös kaapelin massan vertailulogiikan. Toleranssi määritellään muuttujalla "withinTolerance" siten, että tavoitemassa ja mitattu massa eroavat enintään yhden kilon. Tämä toleranssi on massavertailun laadunvalvonnan ydin. Arvoa voidaan muuttaa nopeasti soveltumaan esimerkiksi tulevaisuuden muutoksiin.

Prosessin tarpeiden ja eri muuttujien kartoittaminen on otettu huomioon esimerkiksi tässä moduulissa siten, että minimimassa on asetettu "isAboveMinimum" -muuttujassa. Tämän arvon tulee ylittyä, jotta prosessi etenee. Arvoksi on asetettu viisi kiloa, koska jokaisen kaapelikerän massa ylittää tämän raja-arvon, mutta arvo on kuitenkin tarpeeksi korkea, jotta esimerkiksi jonkun objektin nojatessa vaakasillan reunaan, virhemittausta ei synny.

Muuttujien alustamisen jälkeen niitä käsitellään if-lausekkeessa, jossa yllä mainitun kahden muuttujan lisäksi myös toisesta moduulista tulevan tiedon "barcodesMatched" pitää olla tosi ennen kuin logiikka voidaan suorittaa loppuun. Toleranssi määrittää, onko mittaus onnistunut vai epäonnistunut. Käyttöliittymän indikaattorielementit aktivoituvat tuloksen mukaisesti, ja tämän ollessa prosessin viimeinen vaihe, data lähetetään backendiin.



```
const updateStatusElements = () => {
    if (processing) return; // If a process is already running, do not start another.

    console.log(`Received weight: ${receivedWeight}, target weight: ${targetWeight}`);

    // Check if the weight is within tolerance and above the minimum weight.
    // These variables are the only ones neede to tinker the weighing process. In the p
    // Minimum weight could be the lightest cable -1.5 kg. Though 5kg should suffice si
    withinTolerance = Math.abs(receivedWeight - targetWeight) <= 1;
    isAboveMinimum = receivedWeight >= 5;

    // Update HMI elements and send data to the backend if conditions are met.
    if (withinTolerance && barcodesMatched && isAboveMinimum) {
        successElement.classList.add('active');
        setTimeout(() => successElement.classList.remove('active'), 15000);
        postWeightData();
    } else if (barcodesMatched && isAboveMinimum) {
        failedElement.classList.add('active');
        setTimeout(() => failedElement.classList.remove('active'), 15000);
        postWeightData();
    }
};
```

KUVA 21. Näyte punnituslogiikan moduulista

Mittauslogiikka suunniteltiin aukottomaksi, jotta sitä ei voi suorittaa vahingossa. Se odottaa viivakoodien vertailusta tietoa viivakoodien täsmäyksestä, ja minimimassa estää viime kädessä massavertailun tapahtumisen virheellisesti.

5 KÄYTTÖLIITTYMÄSUUNNITTELU

Käyttöliittymää suunniteltaessa tulisi ottaa huomioon ennen kaikkea sen lopullinen käyttöympäristö ja siihen liittyvät muuttujat. Niin prosessia kuin käyttöliittymää suunniteltaessa on oleellista kartoittaa kattavasti prosessin kulku, vikatilanteet ja käyttöympäristö. Usein käyttöliittymän ongelmakohdat tulevat esiin vasta ohjelmiston valmistumisen jälkeen, kun käyttöliittymää yritetään ottaa käyttöön todellisessa käyttöympäristössä. Vaikka käyttöliittymä olisi täysin toimiva, osa ratkaisuista voi olla aivan liian vaikeaselkoisia ja tehottomia, mikä voi synnyttää turhaa kouluttamistarvetta. (14.)

Koska operaattori ei ole välttämättä tietoinen prosessin kulusta, käyttöliittymä tulee tehdä mahdollisimman yksinkertaiseksi – kuitenkin niin, että sillä voidaan indikoida ja hallita prosessin kaikkia ominaisuuksia.

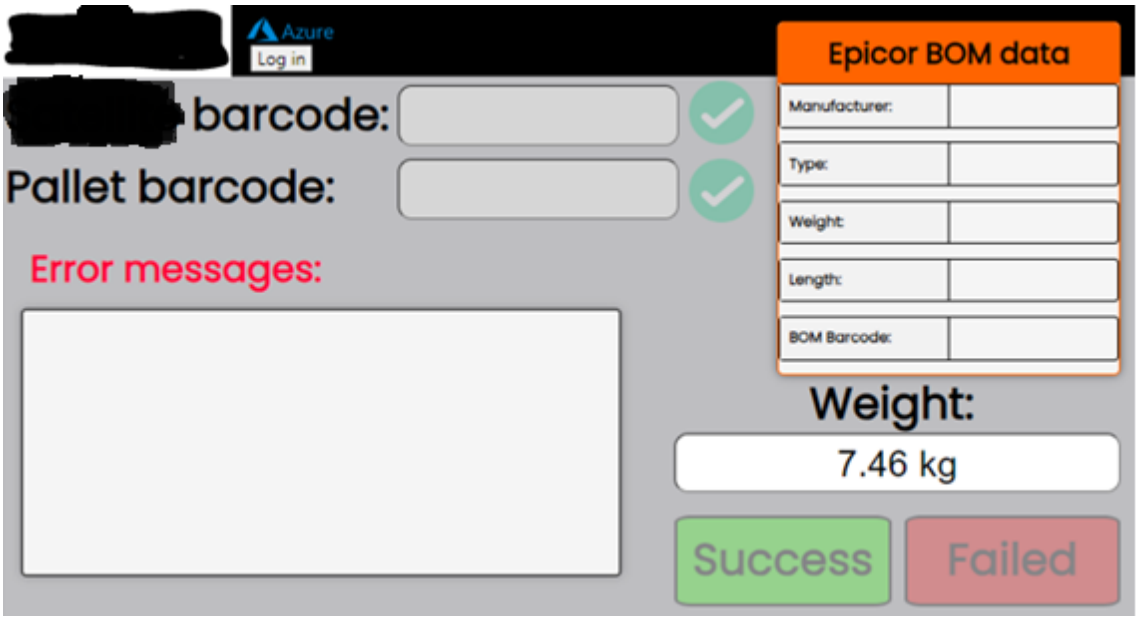
5.1 Vaakalaitteen käyttöliittymä

Käyttöliittymä toteutettiin käyttämällä HTML- ja CSS-teknologioita ilman ulkoisia ohjelmistokehyksiä, kuten Tailwindiä. Vaakalaitteen käyttöliittymää suunniteltaessa oli alusta asti selvää, että näytön rajallinen 7 tuuman koko aiheuttaisi haasteita suunnitteluun. Käytännössä tämä tarkoitti, että kriittisistä elementeistä tulee tehdä tarpeeksi selkeälukuisia. Näyttö tarjosi kuitenkin myös mahdollisuuksia siinä mielessä, että se oli myös kosketusnäyttö, joten ideoin käyttöliittymän siten, ettei hiirtä tarvitse käyttää ollenkaan.

Käyttöliittymän visuaalista ilmettä rakentaessa halusin kiinnittää huomiota yhdennäköisyyteen yrityksen yleisen tyyli-ilmeen kanssa. Etsin fontit ja värikoodit yrityksen mediapankista, lisäksi olin itse käyttänyt muita yrityksen eri ohjelmistoratkaisuja, joista otin inspiraatiota.

Käyttöliittymää tehdessä käytin usein VS Coden lisäosaa, joka avasi liveserverin käyttöliittymänäkymästäni. Tämä oli erittäin hyödyllistä erityisesti positiointeja tehdessä, sillä näin suoraan tekemäni muutokset ja hienosäätö oli helpompaa.

Käyttöliittymässä viivakoodien vertailu ja massan vertailu on haluttu jaotella selkeästi erikseen siten, että prosessin kulku etenee länsimaalaiselle luku- ja kirjoitustavalle loogisesti vasemmalta oikealle (Kuva 22.). Massanäyttö on asetettu suoraan tuodun datan alapuolelle, jotta operaattori voi helposti nähdä kaapelikerän massan suhteessa tavoitemassaan. Vikakoodilaatikko näyttää vikakoodeja jokaisesta mahdollisesta prosessin osa-alueesta. Virheet indikoidaan yleisesti punaisella värillä. Prosessin vaiheisuus on konfiguroitu siten, että kaikki data tyhjenee yhden suoritetun prosessisyklin jälkeen.



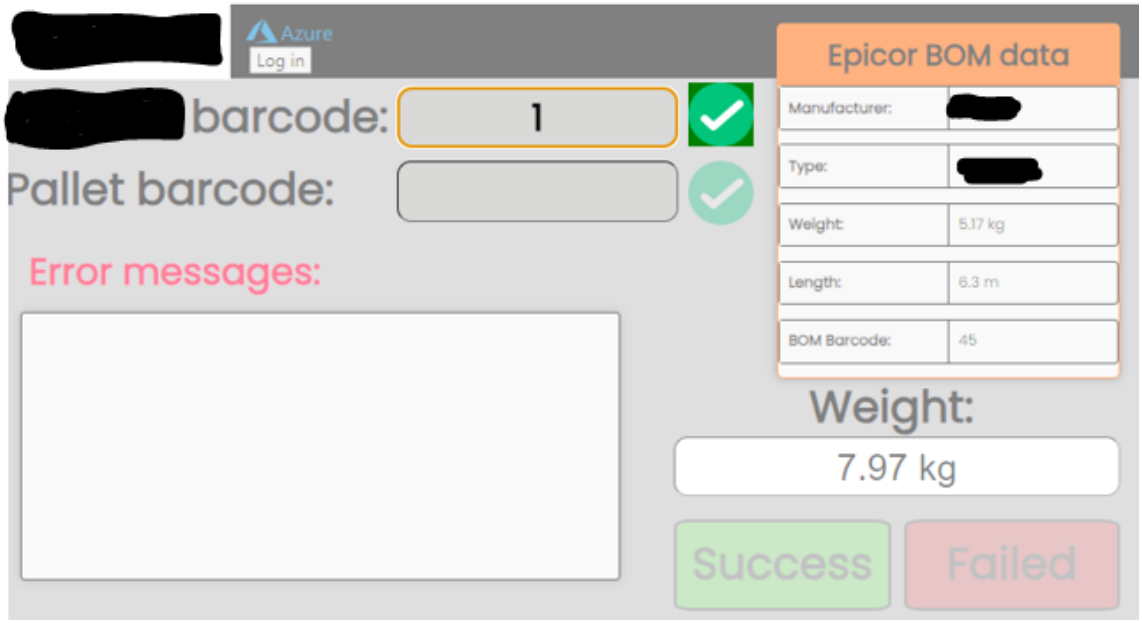
*KUVA 22. Käyttöliittymä prosessin lähtötilanteessa. *Käytetty valedataa ja sensitiivinen informaatio peitetty.*

5.2 Käyttöliittymän mukautuminen

Hyvässä käyttöliittymäsuunnittelussa käyttöliittymän tulee mukautua prosessin eri vaiheisiin intuitiivisella tavalla. Tässä prosessissa haluttiin erottaa kaksi eri laadunvalvonnan vaihetta toisistaan siten, ettei pelkkää massaa voida vertailla ilman viivakoodien vertailun läpäisyä. Tämä on tuotu esiin myös käyttöliittymässä.

Kuvassa 23 nähdään, miten käyttöliittymä mukautuu, kun tuotteen kyljestä luetaan onnistuneesti viivakoodi, jolla löydetään dataa backendistä. Ennen tätä näkymää ruudulle tulee seitsemän sekuntia kestävä koko ruudun ponnahdusikkuna, jossa kerrotaan käyttäjälle, että viivakoodi lähetet-

tiin backendiin onnistuneesti. Käyttöliittymän databoksiin tuodaan tietoa kaapelikerästä, kuten tavoitemassa, joka kertoo operaattorille, minkä painoista kaapelikerää odotetaan. Luetun ensimmäisen kentän oikeinmerkki korostuu, mikä kertoo, että koodi on luettu onnistuneesti. Lisäksi käyttöliittymä sumenee lukuun ottamatta kahta input-kenttää, mikä kertoo operaattorille, että tämä vaihe täytyy läpäistä ennen seuraavaa vaihetta. Tämä myös estää muihin kenttiin koskemisen. Sumennus on tehty manipuloimalla CSS:n opacity- ja z-index-parametrejä.



*KUVA 23. Käyttöliittymä onnistuneen tuotteen viivakoodin luvun jälkeen. *Käytetty valedataa ja sensitiivinen informaatio peitetty.*

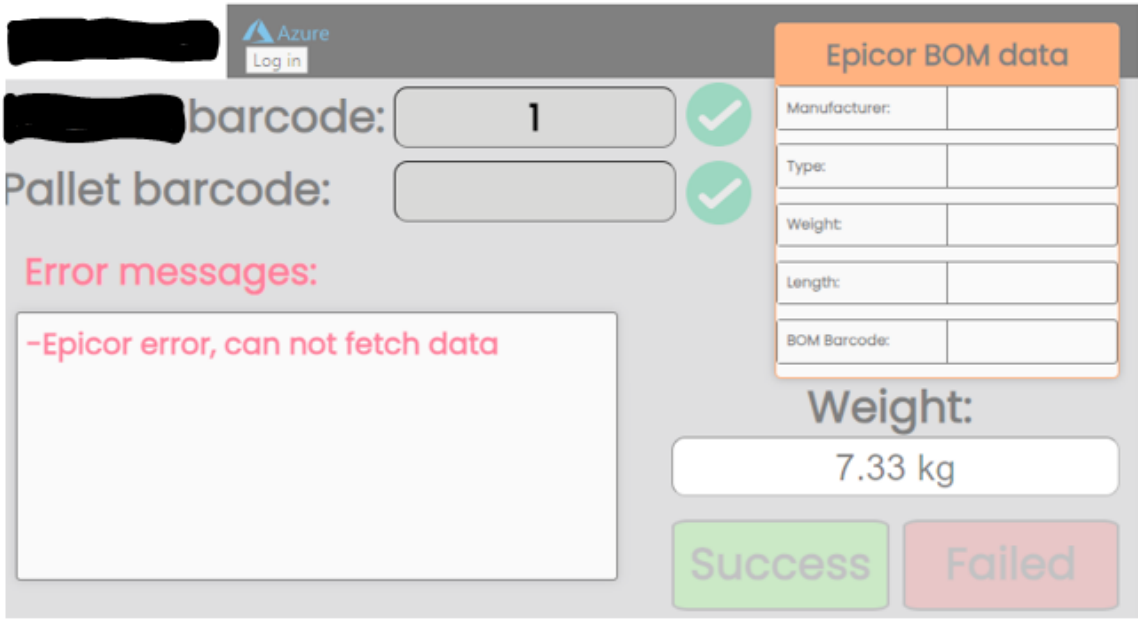
Kun kaapelilavan viivakoodi on luettu ja se on oikea, näytölle tulee seitsemän sekunnin kestävä ponnahdusikkuna, joka kertoo selkeästi, että viivakoodit täsmäävät (Kuva 24.). Tämän lisäksi toisenkin kentän oikeinmerkki korostuu ja käyttöliittymän sumennus poistuu, indikoiden, että voidaan siirtyä seuraavaan vaiheeseen.

Tämän jälkeen suoritetaan massanvertailuprosessi, jossa oikean alakulman "Success" tai "Failed" -elementti syttyy riippuen mittauksen tuloksesta. Tulos voi olla kumpi tahansa, mutta tämän jälkeen käynnistyy 15 sekunnin ajastin, jonka jälkeen käyttöliittymä palautuu lähtötilanteeseen uutta mittausta varten.

*KUVA 24. Käyttöliittymä, kun viivakoodien vertailu on suoritettu onnistuneesti. Kuvassa poistuva ponnahdusikkuna *Käytetty valedataa ja sensitiivinen informaatio peitetty.*

5.3 Vikatilanne-esimerkki

Koodiin on asetettu useita konsolilogeja vikatilanteita ja kehitystä varten (Kuva 25.). Niistä oleellisimmat on tuotu käyttöliittymän vasemmassa alalaidassa sijaitsevaan virhekoodiboksiin. Käyttöliittymään on tuotu vain sellaista informaatiota, josta operaattorille on oleellista hyötyä. Virhekoodeja voi tulla myös useampia samaan aikaan, ja ne asettuvat laatikkoon allekkain. Käyttöliittymään tuotuja virheinstansseja ovat esimerkiksi: viivakoodille ei löydy dataa, viivakoodit eivät täsmää (tästä on myös rastilla poisklikattava ponnahdusikkuna), lavan viivakoodi on luettu vahingossa ensin ja backend on pois käytöstä.



*KUVA 25. Esimerkkivirhenäkymä käyttöliittymässä. *Käytetty valedataa ja sensitiivinen informaatio peitetty.*

Tässä esimerkkitilanteessa viivakoodille ”1” ei löytynyt dataa, ja tämä voidaan havaita myös siitä, ettei databoksiin saatu haettua mitään tietoja. Virhekoodiboksiin tulee näkyviin teksti, joka kertoo käyttäjälle, että vika on backendissä. Virhekoodi poistuu, kun korjaava toimenpide tehdään, tässä tapauksessa kun luetaan viivakoodi, jolle löytyy dataa.

6 OHJELMAN TESTAUS

Kotitestauksia varten luotiin oma valevaakamoduuli, joka tulosti samanlaista tulostetta kuin oikea vaaka. Parametrit voitiin säätää suoraan mieluisiksi, ja koko massavertailulogiikka voitiin toteuttaa ja testata toimivaksi kotona. Backendinä käytettiin itse luotuja vale-JSON-tiedostoja, joihin data muokattiin vastaamaan oikean backendin arvoja, jolloin ohjelma toimi samalla tavalla kuin oikeassa käyttöympäristössä. Koodia siirrettiin eri laitteiden välillä käyttämällä GIT:iä, mikä mahdollisti sen, ettei juuri mitään kehitystyötä tarvinnut tehdä hitaammalla Raspberry Pi:llä.

Tulostimen yhdistäminen ja testaaminen tehtiin kokonaan tehtaalla, koska TSPL-kieli oli niin herkkä muutoksille. Testausta tehtiin useassa vaiheessa, jotta varmistettiin tarran tulostuminen ennen suuria muutoksia.

6.1 Testaus- ja ongelmanratkaisumenetelmät

Koodia rakennettaessa ja testattaessa käytettiin useita eri menetelmiä vianetsintään. Useimmiten ohjelma käynnistettiin, yritettiin tehdä haluttu toiminto ja mahdollinen vikakoodi tarkistettiin konsolista. Vikakoodi luettiin ja arvioitiin, ymmärrettiinkö suoraan, mistä vika johtuu. Korjausta yritettiin tehdä itse, ja ohjelmaa kokeiltiin suorittaa uudelleen. Jos vikakoodia tai virheen mahdollista syytä ei ymmärretty, siirryttiin internettiin lukemaan useita eri foorumeja ja artikkeleja. Tekoälyä hyödynnettiin tässä prosessissa. Näillä menetelmillä onnistuttiin lopulta ratkaisemaan kaikki ohjelman aikana syntyneet ongelmat itse.

Prosessia testattaessa käyttöliittymään luotiin paljon erilaisia nappeja, joilla voitiin testata eri moduulien ominaisuuksia ilman, että koko prosessia täytyi viedä läpi. Esimerkiksi tulostinta testattaessa luotiin nappi, joka antoi success-tiedon tulostinfunktiolle, jolloin tulostimen toimintaa voitiin kokeilla helposti ja nopeasti.

6.2 Loppudemo

Laitteiston valmistuttua ja sen toimivuuden varmistuttua pidettiin yhteyshenkilön kanssa esittelytilaisuus, jossa kerrottiin, mitä oli tehty ja miksi. Koko prosessin vaiheet esiteltiin samalla näyttäen laitteistosta, miten kyseinen osa laadunvalvontaa toteutetaan.
Demo ei ollut täysin eksakti esimerkki, sillä tuotannon kaapelikerälavat eivät vielä sisältäneet viivakoodeja, joista olisi voitu lukea dataa. Lisäksi testin aikana laitteisto ei ollut kytkettynä oikeaan backendiin, joten viivakoodien osalta käytettiin valedataa. Demo oli kuitenkin suunniteltu niin, että kaikille tuli selväksi, mitä tehdään.

Demon jälkeen koodista tehtiin dokumentaatio, ja opinnäytetyö päättyi siihen. Ennen tuotantoon laittamista koodi pitää alustaa firman järjestelmiin, mutta sen hoitaa joku muu.

7 LOPPUPOHDINTAA

Näin jälkikäteen ajateltuna olen erittäin onnekas, että sain tehdä näin sisältörikkaan opinnäytetyön. Jos ajatellaan, että opinnäytetyö on isoin näyttö opintovuosien aikana kerrytetystä osaamisesta, kohdallani tämä todellakin oli sitä. Kun minulle tarjottiin työtä, mikään käytetyistä teknologioista tai laitteista ei ollut tuttua. Tiesin oikeastaan vain, miten kaapelikerän massan vertauslogiikka tulisi toteuttaa aikaisemmissa opinnoissani käytyjen logiikkaopintojen takia.

Insinöörin työnkuva voi olla pitkälti asioiden selvittämistä ja ratkaisujen keksimistä – sitä työssä riitti. Lähdin liikkeelle siitä, että selvitin, mikä on Electron ja miten sen avulla voi rakentaa ohjelman laitteiston ohjaamiseen. Vaikka koin, että olin sisäistänyt miten Electron toimii, valkeni se oikeasti vasta myöhemmin ohjelmaa tehdessä. Työmallini ohjelman teossa oli: tee ensin, korjaa jälkeen. Koin, että tällä tavalla opin myös erittäin paljon, ja hiljalleen Electronin lainalaisuudet alkoivat avautua minulle.

Työssä hauskaa oli se, kun pääsin tekemään asioita myös käsilläni. Muun muassa vaakapäätteen mukana ei tullutkaan valmista kaapelia niin kuin olin ajatellut, joten pääsin taas tutustumaan ja ottamaan uusia asioita haltuun. Toisaalta, kun työ ei ollut pelkkää ohjelmointia, teki se siitä myös mielekkäämmän.

Projektin loppuvaiheessa olin jo unohtanut prosessiin kuuluvan tulostimen olemassaolon ja siinä vaiheessa sen kanssa tappeleminen turhautti. Luotin kuitenkin prosessin aikana kehittyneisiin ongelmanratkaisutaitoihin ja sain sen toimimaan noin puolessatoista viikossa.

Työn ehdottomasti hienoin vaihe oli, kun pääsin demoamaan laitteistoa yrityksessä noin 20-päiselle yleisölle. Pidin yhteishenkilöni kanssa esityksen, missä kerroin mitä tehtiin ja miksi. Lisäksi näytin prosessin toiminnan käytännössä. Tuntui hyvältä, kun ihmiset olivat kiinnostuneita työstä ja pääsin vastaamaan heidän kysymyksiinsä laitteistoon liittyen. Se oli konkretiaa siitä, että kokonaisuus toimi ja työstä oli oikeasti hyötyä.

Kokonaisuus oli hyvin työelämälähtöinen. Olin lukuisissa palavereissa, joissa käsiteltiin laitteiston osaston tulevaisuudennäkymiä ja pääsin antamaan aihepiireihin omia näkemyksiä. Kommunikoin

lukuisten eri tahojen kanssa, kun asioita piti sopia tai selvittää. Hauskana anekdoottina sanottakoon, että pääsin hoitamaan myös reklamaatioprosessia, kun vaakapäätteen reppukortti hajosi.

Tässä tekstiosuudessa ei käsitellä erikseen ergonomian ja mittaustelineen suunnittelua. Tein alustavat piirrokset eksakteilla mitoilla vaakasillan päälle asetettavasta telineestä, jonka tarkoituksena on vähentää nostokorkeutta ja täten vähentää operaattoriin kohdistuvaa stressiä päivittäisessä työskentelyssä. Kiinnitimme erityistä huomiota pöytään, jolla näyttö ja tulostin sijaitsevat. Prosessi ei vaadi pitkäkestoista katselua, joten pöytä on seisomakorkeudella, jolloin tarvittavan informaation voi lukea ja katsoa nopeasti. Pöydästä haluttiin tarpeeksi iso, jotta vaakasillan sai puoliksi pöydän alle, jolloin kokonaisuus vei vähemmän tilaa tuotannosta. Loppukokonaisuudesta ei valitettavasti ole kuvaa tuotannon osastomuuton takia.

7.1 Vaihtoehtoiset menetelmät

Vaakalaitelaitteistoa tehdessä keskustelimme usein yhteyshenkilöni kanssa menetelmistä, joilla olisi voinut saavuttaa vastaavan laadunvalvonnallisen lopputuloksen. Yksi menetelmistä, joka nousi toistuvasti esiin, oli konenäkö. Konenäöllä olisi voitu korvata massanvertailuprosessi siten, että kaapelikerä olisi tunnistettu pelkästään ulkoisten ominaisuuksien perusteella.

Konenäköä käytetään jatkuvasti yhä laajemmin teollisuuden laadunvalvonnassa. Konenäkö on parhaimmillaan erittäin tarkka, sillä se kykenee jopa millin tuhannesosan tarkkuuteen. Prosessiin konenäkö olisi tuonut lisäulottuvuuden siten, että sillä olisi voinut havaita kaapelikerien ulkoiset poikkeamat, joita massavertailussa ei oteta lainkaan huomioon. Jos budjettirajoituksia ei olisi, niin lähes aukottoman laadunvalvontajärjestelmän olisi saanut yhdistämällä massan mittaamisen ja konenäön. Tällöin konenäön rooliksi olisi jäänyt yksinomaan ulkoisten poikkeamien havainnointi.

Todelliseksi vaihtoehdoksi konenäkö ei koskaan noussut, sillä se olisi ollut paljon hintavampaa toteuttaa, ja nykyinenkin toteuttamistapa täyttää prosessin vaatimukset erittäin hyvin. Kuitenkin tekniikan kehittyessä ja hinnan tullessa väistämättä alas voisi konenäkö olla tulevaisuudessa erittäin varteenotettava vaihtoehto osana prosessia. En pitäisi mahdottomuutena, että konenäkö retrofitattaisiin osaksi nykyistä systeemiä vuosien päästä. (15.)

7.2 Työn parantamismahdollisuuksia

Laitteistoa suunniteltaessa ja tehdessä tyhjästä on luonnollista, että lopputuloksesta ei tule ensimmäisellä kerralla täydellistä. Koen kehittyneeni työn aikana niin paljon, että on vain muutama asia, mitä tekisin samalla tavalla, jos lähtisin toteuttamaan työtä alusta uudestaan.

Laitteistovalinta oli mielestäni lähes täysin onnistunut. Budjetin salliessa olisin halunnut pienen Windows-PC:n keskusyksiköksi, joka olisi ollut tehokkaampi ja täten mieluisampi käyttää, toisaalta kerran, kun Raspberry Pi:llä saa ohjelman käynnistymään ei siinäkään ole mitään ongelmia.

Käyttöliittymän suunnittelussa jäi paljon parannettavaa. Käyttöliittymä toimii nyt hyvin sille määrätyssä näyttökoossa, mutta en tajunnut tehdä siitä alusta asti responsiivista, joten se ei toimi millään muulla näyttökoolla. Vastaisuudessa käyttäisin myös jotain CSS-frameworkia, kuten Bootstrapia, tai Tailwindiä, joiden avulla esimerkiksi elementtien positiointi olisi helpompaa. Tulevaisuudessa ajattelen, että käyttöliittymä voisi olla jollain isommalla monitorilla paremman luettavuuden vuoksi. Toteutunut pöytäpinta-ala on sen verran laaja, ettei kompaktille ratkaisulle ole nykyiselläänkään perusteita.

Virheidenkäsittelyä voi jatkokehittää. Todellisia jumitilanteita varten käyttöliittymään olisi voinut asettaa painikkeet, joiden avulla voisi resetoida prosessin suoraan ilman, että koko applikaatiota tarvitsisi käynnistää uudestaan. Toisaalta näihin tilanteisiin ei tullut erillistä ohjeistusta, tai pyyntöä. Painikkeiden läsnäolo voisi myös aiheuttaa prosessin väärinkäyttöä.

Itse logiikkaa tehdessä tekisin alkuun paremman suunnitelman siitä mitä prosessi vaatii, ja täten saisin jaettua koodin vielä selkeämpiin osiin, joka parantaisi sen luettavuutta ja jatkokäyttöä. Electronia käytettäessä tekisin monikäyttöisemmät Preload-altistukset, jolloin jokaiselle pyynnölle ei tarvitsisi erillistä skriptiä. Tarkastelisin, onko Electron todella paras menetelmä toteuttaa sovellus. Electroniin olisi helppo tukeutua sen ollessa tuttu myös tulevissa projekteissa, mutta optimaalisin menetelmä se ei välttämättä ole.

7.3 Työskentelytapojeni analyysi

Eniten haasteita projektin läpiviennissä tuotti aikataulujen arviointi. Oli hyvin hankala arvioida kuinka paljon aikaa jonkin ominaisuuden kehittämiseen ja toteuttamiseen kuluisi, kun monessa asiassa aiheeseen ei ollut aikaisempaa tarttumispintaa. Kuitenkin koen, että olisin voinut silti onnistua paremmin aika-arvioissa. Ihan jo työelämää silmällä pitäen olisi suotavaa, että pystyisi kertomaan viikko- ja päivätarkan aikataulun ja pysymään siinä.

Olisin voinut myös pitää parempaa kirjaa tekemistäni asioista, vaikka viikkoraportin muodossa, tällöin olisi muodostunut parempi kokonaiskuva työn etenemisestä. Lisäksi tämä olisi pitänyt yhteyshenkilöni paremmin ajan tasalla ja he olisivat voineet tarjota apua ja näkökulmia helpommin.

Minulle tarjottiin mahdollisuutta ulkoisten resurssien käyttöön, mutta en oikein osannut hyödyntää niitä tässä työssä. Tuntui hieman kömpelöltä kysyä viikon päähän apua johonkin tiettyyn asiaan, kun olin saattanut ratkaista asian jo sillä aikaa. Jälkikäteen ajateltuna tämä oli ainoastaan hyvä asia, sillä itse opin enemmän ja yritys säästi rahaa.

Kaiken kaikkiaan koin, että onnistuin toimimaan työyhteisön jäsenenä ja viemään annettua projektia eteenpäin toiveiden mukaisesti. Sain runsaasti oppia edessä siintävään työelämään ja aion käyttää tätä suoritettua projektia apuna muun muassa työnhaussa.

LÄHTEET

1. Electron JS 2023. Hakupäivä 15.4.2024. https://www.electronjs.org/docs/latest.
2. Electron Forge 2024. Hakupäivä 13.5.2024. https://www.electronforge.io/.
3. Webpack.js. 2024. Hakupäivä 13.5.2024. https://webpack.js.org/.
4. Ojala, Olli-Pekka 2022. Ohjelmiston jatkuvaa käyttöönottoa pilotoitiin veikkauskisasovelluk-sella. Hakupäivä 13.5.2024. https://vanha.oamk.fi/oamkjournal/2022/ohjelmiston-jatkuvaa-kayttoonottoa-pilotoitiin_veikkauskisasovelluksella/.
5. Minebea intec 2024. Industrial bench scale Midrics. Hakupäivä 9.1.2024. https://www.minebea-intec.com/en/industrial-scales/bench-scales/bench-scale-midrics.
6. Raspberry Pi 2024. Hakupäivä 2.5.2024. https://www.raspberrypi.com/.
7. Cellan-Jones, Rory 2011. A 15 pound computer to inspire young programmers. BBC news. Hakupäivä 2.5.2024. https://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2011/05/a_15_computer_to_inspire_young.html.
8. Datalogic 2024.Gryphon-4500 series. Hakupäivä 3.5.2024. https://www.datalogic.com/eng/retail-manufacturing-transportation-logistics-healthcare-other-applications/handheld-scanners/gryphon-4500-series-pd-836.html.
9. TSC printers 2024. Tx-series 4 inch performance printers. Hakupäivä 2.5.2024. https://usca.tscprinters.com/en/products/tx-series-4-inch-performance-desktop-printers.
10. Brother 2018. Mustesuihku-, laser- tai suora lämpötulostus: Mikä on paras yrityksellesi? Hakupäivä 2.5.2024. https://www.brother.fi/about-brother/news/2018/news-ink-laser-or-thermal.
11. Stratus engineering 2016. RS232 9-pin pinout explained. Hakupäivä 9.1.2024. https://stratusengineering.com/wp-content/uploads/2016/07/RS232-9-pin-pinout-explained.jpg.
12. Minebea Intec 2024. Weight indicators Midrics. Hakupäivä 10.1.2024. https://www.minebea-intec.com/en/weighing-electronics/desktop-weight-indicators-controllers/weight-indicator-midrics.
13. TSPL programming 2009. TSC barcode programming manual. Hakupäivä 4.3.2024. http://www.bar-tech.com.tw/eng/download/12790046025_TSPL_TSPL2_Programming_2009_06_24.pdf.

14. Laakso, Sari & Laakso, Karri-Pekka 2004. Hyvän käyttöliittymän varmistaminen GUIDe-prosessimallilla. Hakupäivä 17.4.2024. https://www.cs.helsinki.fi/u/salaakso/papers/GUIDe-suomeksi.pdf-.
15. Algol technics 2018. Konenäkö tarkistaa ja ohjaa. Hakupäivä 15.5.2024. https://www.algoltechnics.fi/artikkelit-ja-asiakastarinat/konen%C3%A4k%C3%B6-tarkistaa-ja-ohjaa.