FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Diplomová práce

Vývoj servisní softwarové aplikace pro BMS

Studijní program: N0613A140028 – Informační technologie
Studijní obor: Aplikovaná informatika
Autor práce: **Bc. Jan Pluhař**
Vedoucí práce: Ing. Pavel Jandura, Ph.D.

Liberec 2024

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání diplomové práce

Vývoj servisní softwarové aplikace pro BMS

Jméno a příjmení:	**Bc. Jan Pluhař**
Osobní číslo:	M22000026
Studijní program:	N0613A140028 Informační technologie
Zadávající katedra:	Ústav mechatroniky a technické informatiky
Akademický rok:	2023/2024

Zásady pro vypracování:

1. Seznamte se s problematikou bateriového managementu (BMS) z pohledu požadavků na jeho interakci s uživatelem – konfigurace a diagnostika.
2. Proveďte rešerši komunikační standardů, vhodných pro danou aplikaci.
3. Navrhněte a zdokumentujte komunikační protokol mezi BMS a servisní aplikací. Při návrhu je vhodné vycházet z průmyslových standardů, kupř. J1939 a dalších.
4. Vytvořte servisní aplikaci pro PC, která umožní vizualizaci dat z BMS. Typicky datové i grafické zobrazení hodnot a čtení chybových kódů.

Rozsah grafických prací:	dle potreby dokumentace
Rozsah pracovní zprávy:	40 až 50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	čeština

Seznam odborné literatury:

[1] LAWRENZ, Wolfhard. *CAN system engineering: from theory to practical applications*. Second edition. London: Springer, 2013. ISBN 978-1-4471-5612-3.

[2] ANDREA, Davide, *Battery Management Systems for Large Lithium-Ion Battery Packs*, Artech, 2010. ISBN 978-1608071043

[3] SAE INTERNATIONAL, 2023. *J1939 Digital Annex*. J1939DA_202310. USA.

Vedoucí práce:	Ing. Pavel Jandura, Ph.D. Ústav mechatroniky a technické informatiky
Datum zadání práce:	12. října 2023
Předpokládaný termín odevzdání:	14. května 2024

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

doc. RNDr. Pavel Satrapa, Ph.D.
garant studijního programu

V Liberci dne 12. října 2023

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.
Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.
Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.
Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.
Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.
Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.
Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

13. 5. 2024 Bc. Jan Pluhař

Vývoj servisní softwarové aplikace pro BMS

Abstrakt

Tato diplomová práce se zabývá vývojem servisní softwarové aplikace pro systém správy baterií (BMS). Cílem práce bylo navrhnout a implementovat vlastní komunikační protokol pro komunikaci servisní aplikace s BMS a následně vyvinout vlastní servisní aplikaci. Byla provedena rešerše komunikačních standardů a poté byl navržen vlastní komunikační protokol na míru konkrétní aplikace. Dále byl v práci rozebírán návrh a implementace desktopové servisní aplikace s grafickým rozhraním, která umožňuje vizualizaci, konfiguraci a sběr měřených dat.

Klíčová slova: software, desktopová aplikace, komunikační protokol, stavový automat, BMS, UART, Electron, React, SQlite

Development of service software application for BMS

Abstract

This master's thesis focuses on the development of a service software application for a Battery Management System (BMS). The goal of the thesis was to design and implement a proprietary communication protocol for the interaction between the service application and the BMS, and then to develop a dedicated service application. A review of communication standards was conducted, followed by the design of a custom communication protocol tailored to the specific application. Additionally, the thesis discusses the design and implementation of a desktop service application with a graphical interface that enables visualization, configuration, and collection of measured data.

Keywords: software, desktop application, communication protocol, state machine, BMS, UART, Electron, React, SQlite

Poděkování

Děkuji panu Ing. Pavlu Jandurovi, Ph.D. za jeho ochotu a čas při tvorbě zadání a vedení diplomové práce.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zkratek

BMS	Battery Management System
SoC	System on a chip
SOC	State of charge
SOH	State of health
CAN	Controller Area Network
CRC	Cyclic redundancy check
PLC	Programmable logic controller
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
JSON	JavaScript Object Notation
UART	Universal asynchronous receiver-transmitter
API	Application programming interface
GUI	Graphical user interface
EIS	Elektrochemická impedanční spektroskopie
Li-S	Lithium sulfur batteries
LSB	Least significant bit
SSB	Solid-state batteries
UDS	Unified Diagnostic Services
SQL	Structured query language
USB	Universal Serial Bus
DLC	Data length code
ACK	Acknowledgement
TCP	Transmission Control Protocol
RTU	Remote Terminal Unit

1 Úvod

Tato diplomová práce se zabývá vývojem servisní softwarové aplikace pro systém správy baterií (Battery Management System, BMS), klíčovou technologií v mnoha současných i budoucích aplikacích spojených s ukládáním a správou energie. BMS hraje zásadní roli v řízení baterií, což je kritické pro optimalizaci výkonu a prodloužení životnosti bateriových článků, což má přímý dopad na celkovou efektivitu a bezpečnost zařízení, jako jsou elektrická vozidla, obnovitelné zdroje energie a mobilní zařízení.

Práce začíná seznámením s problematikou bateriového managementu z pohledu uživatele, který potřebuje s BMS interagovat za účelem diagnostiky a konfigurace. To zahrnuje základní přehled technologií, trendů, využití bateriového managementu a příklad konfigurace BMS. Dále následuje rešerše komunikačních standardů používaných v rámci bateriového managementu. Po teoretickém úvodu práce následuje praktická část vlastního řešení.

Prvním cílem práce je vlastní návrh komunikačního protokolu. V práci je detailně popsán návrh komunikace mezi servisní aplikací a BMS. Je zde popsáno připojení BMS k počítači se servisní aplikace a zahájení komunikace. Pro spolehlivou komunikaci je navržena a popsána struktura zprávy s detekcí chyb. Kvůli různorodosti dat a požadavků na konfiguraci jsou v práci popsány požadavky umožňující aplikaci bohatou interakci s BMS. Životní cyklus komunikace je řízen vlastní implementací stavového automatu, který je v práci detailně popsán.

Druhou částí práce je vytvoření servisní aplikace pro BMS s využitím navrženého protokolu. V práci jsou popsány technologie zvolené pro implementaci servisní aplikace a následně je detailně popsána struktura a její fungování. Také jsou ukázána jednotlivá okna aplikace umožňující datovou i grafickou vizualizaci získávaných dat z BMS. Je popsán návrh a implementace ukládání měřených dat a také možnost uživatelské konfigurace vnitřních proměnných. Uživatel má také možnost uloženou konfiguraci načítat nebo exportovat. V závěru práce je popsáno jak testování aplikace s využitím fyzického hardwaru, tak i automatizovanými testy.

2 Problematika BMS a jeho význam

Bateriový management je klíčovým prvkem v elektrických zařízeních, které využívají akumulátory. Jeho hlavním cílem je optimalizovat výkon, životnost a bezpečnost baterií. S rozvojem mobilních zařízení, elektrických vozidel a obnovitelných zdrojů energie nabývá bateriový management stále většího významu. Tato rešerše se zaměřuje na klíčové aspekty bateriového managementu, jeho význam v různých oblastech použití a aktuální trendy v tomto odvětví.

2.1 Technologie bateriového managementu

Monitoring stavu baterie

(Battery State Monitoring) BMS sleduje klíčové parametry baterie jako je napětí, proud, teplota a stav jejího nabití. Tato data jsou důležitá pro správné řízení nabíjení a vybíjení baterie a zabraňují přetížení či podbití bateriových článků, což může vést k poškození nebo elektrickému zkratu.

Řízení nabíjení a vybíjení baterií

Optimalizace procesů nabíjení a vybíjení je klíčová pro zlepšení životnosti baterie a maximalizaci výkonu zařízení. To zahrnuje správné nastavení nabíjecího a vybíjecího proudu, kontrolu teploty baterie a minimalizaci ztrát energie.

Například při přibližování se k hornímu limitu napětí článků může BMS požádat o postupné snížení nabíjecího proudu nebo může při hrozícím přepětí nabíjení úplně odpojit. Na druhé straně, když se napětí článků přibližuje ke spodní hranici, BMS požádá klíčová zařízení o snížení zátěže. V případě elektrického vozidla může být toto provedeno snížením povoleného točivého momentu trakčního motoru. [30]

Balancování článků

Bateriové balancování zajišťuje rovnoměrné rozložení náboje mezi jednotlivými články baterie. Tím se zabraňuje přetížení některých článků a prodlužuje se celková životnost baterie.

Diagnostika a predikce poruch

Moderní BMS obsahuje pokročilé diagnostické funkce, které detekují anomálie v chování baterie a umožňují predikci možných poruch. To umožňuje preventivní údržbu a minimalizaci rizika selhání baterie.

2.2 Význam bateriového managementu

Elektrická vozidla

V automobilovém průmyslu je BMS klíčovým faktorem pro dosažení vyšší dojezdové vzdálenosti, rychlejšího nabíjení a delší životnosti baterií. Optimalizace bateriového managementu přispívá k širší adopci elektrických vozidel a snižuje jejich provozní náklady.

Spotřební elektronika

V mobilních telefonech, noteboocích a další spotřební elektronice je bateriový management nezbytný pro zajištění dlouhé výdrže baterie a bezpečného používání zařízení.

Obnovitelné zdroje energie

V bateriových systémech pro ukládání energie z obnovitelných zdrojů, jako jsou solární nebo větrné elektrárny, hraje bateriový management klíčovou roli při optimalizaci využití získané energie a udržení stability energetické sítě.

2.3 Trendy v bateriovém managementu

Inteligentní algoritmy a umělá inteligence

S rostoucí dostupností výkonných počítačů se stále více využívají inteligentní algoritmy a techniky umělé inteligence pro optimalizaci bateriového managementu. Tyto technologie umožňují adaptivní řízení baterie podle specifických podmínek provozu a zvyšují její výkon a životnost.

Pokročilé senzory

Rozvoj senzorů a senzorických technologií umožňuje přesnější monitorování stavu baterie a diagnostiku poruch. Pokročilé senzory mohou nepřímými metodami, jako je např. elektrochemická impedanční spektroskopie (EIS) sledovat mikroskopické změny ve struktuře baterie nebo detekovat potenciálně nebezpečné situace jako přehřátí či zkrat.

Vývoj nových typů baterií

Vedoucí výzkum v oblasti nových typů baterií, jako jsou lithium-sirné(Li–S - Lithium sulfur batteries)[20] nebo baterie s pevným elektrolytem(SSB - Solid-state batteries)[31], klade nové nároky na bateriový management. Nové typy baterií vyžadují specifické řídicí strategie a technologie pro dosažení optimálních výsledků. [3][9]

2.4 Interakce uživatele s BMS

V kontextu BMS je interakce s uživatelem a její optimalizace nezbytná pro zajištění efektivního využití, správy a udržování bateriových packů. Klíčovými aspekty těchto systémů jsou konfigurace, diagnostika, a komunikace, které umožňují uživatelům monitorovat a případně upravovat provozní parametry baterie tak, aby bylo dosaženo optimálního výkonu a prodloužení její životnosti.

Konfigurace BMS

Konfigurace BMS je zásadním prvkem, který umožňuje adaptaci systému na specifické požadavky bateriového packu a aplikace. Tato konfigurace zahrnuje nastavení parametrů, jako jsou napěťové a proudové limity, parametry pro řízení nabíjení a vybíjení, a nastavení pro monitorování stavu baterie (SOC, SOH). Možnost individuálního nastavení těchto parametrů podporuje optimalizaci výkonu baterie a její ochranu před potenciálním poškozením[7].

Pro příklad proměnných a parametrů v konfiguraci BMS lze vycházet z dokumentace k platformě pro správu baterií (Battery Management System, BMS) RDDRONE--BMS772 od společnosti NX[24]

V BMS jsou definovány 3 druhy proměnných: **proměnné BMS**, jako jsou napětí, proudy a teploty; **konfigurační parametry BMS**, které umožňují nastavení systému, například parametry baterie; a **hardwarové parametry**, jako je maximální proud omezený disipací výkonu MOSFETů. Příklady proměných lze vidět na tabulkách 2.1,2.2 a 2.3

Diagnostika

Diagnostické funkce BMS poskytují hlubší vhled do stavu bateriového systému, včetně detekce a identifikace potenciálních problémů nebo poruch. Vzdálená diagnostika přes protokoly, jako je UDS (Unified Diagnostic Services) s využitím sběrnice CAN (Controller Area Network), umožňuje uživatelům získávat data o stavu a výkonu baterie v reálném čase. To umožňuje včasnou reakci na identifikované problémy a minimalizaci rizika výpadku nebo poškození baterie [7][11].

Parameter	Unit	type	Description
c-batt	C	float	temperature of the external battery temperature sensor
v-out	V	float	The voltage of the BMS output
v-batt	V	float	The voltage of the battery pack
i-batt	A	float	The last recorded current of the battery
i-batt-avg	A	float	The average current since the last measurement
i-batt-10s-avg	A	float	The 10s rolling average current, updated each 1s
s-out	-	bool	true if the output power is enabled
s-in-flight	-	bool	true if the system is in flight
p-avg	W	float	Average power consumption over the last 10 seconds
e-used	Wh	float	Power consumption since device boot

Tabulka 2.1: Příklad proměnných BMS[24]

Monitorování a ochrana

BMS nepřetržitě monitoruje klíčové parametry baterie jako jsou napětí, proud, teplota a stav nabití (SOC). Toto monitorování zajišťuje, že baterie je provozována v rámci jejího bezpečného operačního rozmezí, čímž se minimalizuje riziko poškození způsobeného přetížením, přehřátím, nebo hlubokým vybíjením. Ochranné funkce BMS, včetně elektrické a tepelné ochrany, jsou klíčové pro udržení integrity a bezpečnosti bateriového systému[30].

Komunikace a rozhraní

Rozhraní BMS a jeho schopnost komunikovat s externími systémy jsou nezbytné pro integraci bateriového managementu do širších aplikací a systémů. Standardní komunikační protokoly, jako je RS232 nebo CAN, umožňují výměnu dat mezi BMS a řídicími systémy vozidla, energetickými systémy nebo uživatelskými rozhraními. Flexibilita v komunikaci a možnost přizpůsobení rozhraní podle potřeb aplikace umožňuje efektivní správu a kontrolu bateriového systému[7].

Parameter	Unit	type	Description
n-cells	-	uint8_t	Number of cells used in the BMS board
t-meas	ms	uint16_t	Cycle for complete battery measurement and SOC estimation
t-ftti	ms	uint16_t	Cycle for diagnostics (Fault Tolerant Time Interval)
t-cyclic	s	uint8_t	Wake up cyclic timing of the AFE during sleep mode
i-sleep-oc	mA	uint8_t	Overcurrent threshold detection in sleep mode
v-cell-ov	V	float	Max allowed voltage for one cell
v-cell-uv	V	float	Min allowed voltage for one cell
c-pcb-ot	C	float	PCB Ambient temperature over temperature threshold
c-cell-ot	C	float	Over temperature threshold for cells
i-out-max	A	float	Maximum current threshold to open the switch during normal operation, if not overruled

Tabulka 2.2: Příklad konfiguračních parametrů BMS[24]

Parameter	Unit	type	Description
v-min	V	uint8_t	Minimum stack voltage for the BMS board to be fully functional
v-max	V	uint8_t	Maximum stack voltage allowed by the BMS board
i-range-max	A	uint16_t	Maximum current that can be measured by the BMS board
i-max	A	uint8_t	Maximum DC current allowed in the BMS board (limited by power dissipation in the MOSFETs)
i-short	A	uint16_t	Short circuit current threshold
t-short	us	uint8_t	Blanking time for the short circuit detection
i-bal	mA	uint8_t	Cell balancing current under 4.2V with cell balancing resistors of 82 ohms
m-mass	kg	float	The total mass of the (smart) battery

Tabulka 2.3: Hardwarové parametry BMS[24]

3 Komunikační standardy pro BMS

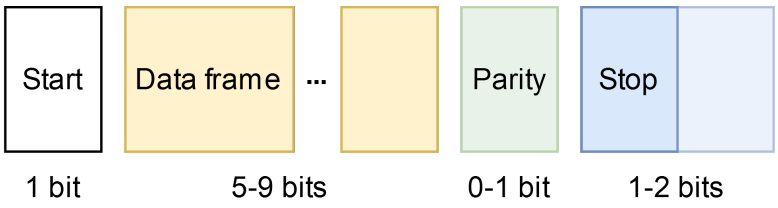
3.1 Universal Asynchronous Receiver/Transmitter (UART)

UART je komunikační protokol, který se často používá pro sériovou komunikaci mezi mikrokontroléry, senzory, periferními zařízeními a počítači. Poskytuje jednoduchý způsob přenosu a příjmu dat asynchronně, bez potřeby sdíleného hodinového signálu mezi zařízeními.

UART komunikace zahrnuje dva hlavní prvky: vysílač (Tx) a přijímač (Rx). Tyto prvky vyměňují data sériově, přes dvě komunikační linky. Komunikace může probíhat jednosměrně i obousměrně v jednu chvíli. Linka Tx se používá k odesílání dat od odesílatele k přijímači, zatímco linka Rx slouží k přijímání dat od odesílatele.

3.1.1 Struktura paketů

Přenášená data UARTem jsou uspořádána do paketů. Každý paket obsahuje 1 startovací bit, 5 až 9 datových bitů (v závislosti na UARTu), volitelný paritní bit a 1, nebo 2 stop bity:



Obrázek 3.1: UART paket

Start bit

Datová linka UARTu je v nečinnosti udržována na vysoké napěťové úrovni. Při začátku přenosu vysílající UART stáhne napětí linky z „High“ na „Low“ po dobu jednoho hodinové cyklu. Když přijímací UART detekuje přechod linky z „High“ na „Low“, začne číst bity s frekvencí přenosové rychlosti.

Datový rámec

Datový rámec obsahuje samotná aplikační data. Délka dat může být 5 až 9 bitů. Pokud se využívá bit parity, dat může být maximálně 8 bitů. Ve většině případů se data přenášení v LSB formě.

Parita

Parita je jedna z nejednodušších metod k odhalení poškození dat při přenosu. Parita se rozlišuje na sudou a lichou. Paritní bit se nastavuje podle počtu 1 v datovém rámci. Dle zvolené parity se paritní bit nastaví buď na 1, nebo 0 tak, aby počet 1 pro sudou paritu byl sudý a pro lichou paritu lichý. Použití paritního bitu je nepovinné. Pro lepší detekci chyb je vhodné použít např. cyklický redundantní součet.

Stop bit

Stop bit, označuje konec datového paketu. Obvykle je dlouhý dva bity, ale často se používá pouze jeden bit. Pro ukončení přenosu udržuje UART datovou linku na „High“. [5][6]

3.1.2 Synchronizace a vzorkování

Standardní digitální data nemají význam bez nějaké formy synchronizace. Zatímco v tradičním přenosu digitálních dat je synchronizace dosaženo pomocí hodinového signálu, UART používá asynchronní přístup.

Při asynchronní komunikaci UART neexistuje samostatný hodinový signál pro synchronizaci mezi vysílačem (Tx) a přijímačem (Rx). Vysílač generuje bitový proud na základě svého vlastního hodinového signálu a přijímač musí správně vzorkovat příchozí data uprostřed každé bitové periody. Ačkoli interní hodiny vysílače a přijímače jsou nezávislé, přijímač musí vzorkovat data ve středu každého bitové periody, aby byla zajištěna robustnost přenosu proti mírné odlišnosti vysílací a vzorkovací hodinové frekvence.

K dosažení správného vzorkování se přijímač synchronizuje se startovacím bitem, který indikuje začátek přenosu. Tímto se zahájí proces synchronizace. Frekvence hodinového signálu (baud rate) posílaného na přijímač je mnohem vyšší než skutečný baud rate, což umožňuje správné vzorkování ve středu každého bitové periody.

Konkrétní postup synchronizace a vzorkování se liší podle počtu hodinových cyklů přijímače. Například, pokud jedna bitová perioda odpovídá 16 hodinovým cyklům přijímače, pak proces synchronizace a vzorkování probíhá následovně:

- Přijímač je spuštěn klesající hranou startovacího bitu.
- Přijímač čeká 8 hodinových cyklů, aby určil vzorkovací bod blízko středu bitové periody.

- Poté přijímač čeká dalších 16 hodinových cyklů, aby se dostal do středu periody prvního datového bitu.
- První datový bit je vzorkován a uložen do registru přijímače a poté přijímač čeká dalších 16 hodinových cyklů před vzorkováním druhého datového bitu.
- Tento proces se opakuje, dokud nejsou vzorkovány a uloženy všechny datové bity, a poté během vzestupné hrany stopového bitu se vrací UART rozhraní do nečinného stavu.

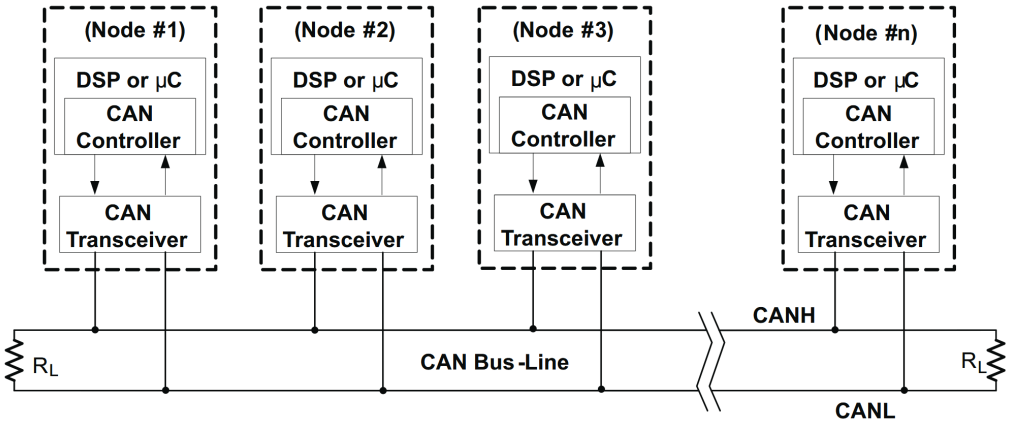
Tímto způsobem je dosaženo úspěšného přenosu dat přes UART bez potřeby samostatného hodinového signálu pro synchronizaci. [4]

3.2 Controller Area Network (CAN)

3.2.1 Úvod

Protokol Controller Area Network (CAN), vyvinutý společností Bosch, je zaměřen na maximalizaci spolehlivosti, efektivity a flexibility komunikace v náročných prostředích. Jeho klíčovými charakteristikami jsou schopnost multi-master komunikace, prioritní bitová arbitráž a robustní chybové detekční mechanismy. CAN umožňuje efektivní výměnu krátkých zpráv mezi uzly v síti bez potřeby centrálního řízení, což je ideální pro aplikace vyžadující konzistentní a spolehlivou výměnu dat, jako jsou automobilové sítě, průmyslové kontrolní systémy a další.[17]

3.2.2 Fyzická vrstva

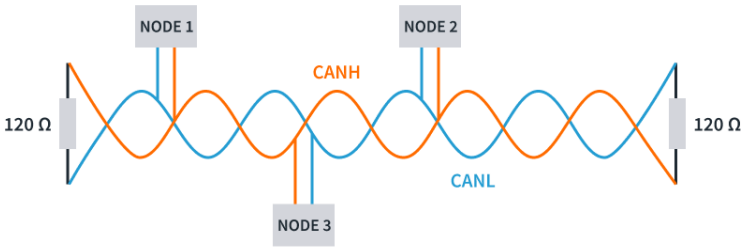


Obrázek 3.2: CAN sběrnice[17]

Fyzická vrstva protokolu CAN se zabývá přenosem dat přes fyzické médium, přičemž existují různé specifikace jako High-Speed CAN (ISO 11898-2), Low-Speed CAN (ISO 11898-3) a Single Wire CAN (SAE J2411). Například High-Speed CAN

využívá dva vodiče CANH (CAN High) a CANL (CAN Low) ve zkrouceném páru s terminačními rezistory 120 Ω na obou koncích, což zabraňuje odrazu signálů. V automobilovém průmyslu jsou uzly na komunikační sběrnici obvykle napájeny ze stejného zdroje, čímž je zajištěn rovný potenciál země.Viz obrázek 3.3

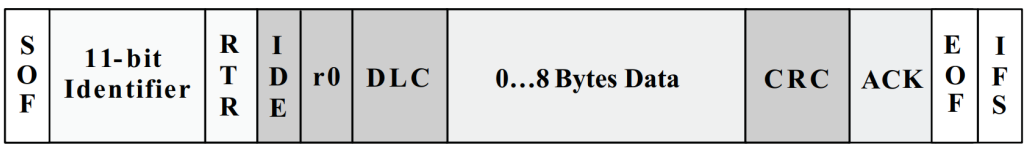
Velkou výhodou CAN sběrnice je přenos data pomocí diferenciálního signálu mezi dvěma vodiči (CANH a CANL), který je odolnější vůči elektromagnetickému rušení. Tato technika umožňuje spolehlivý přenos dat i v prostředí náchylném k rušení tím, že využívá rozdílu potenciálů mezi dvěma signály. Když řadič posílá logickou '0', sběrnice je ve dominantním stavu s potenciálním rozdílem přibližně 2V, zatímco při posílání logické '1' je v recesivním stavu s rozdílem přibližně 0V. Pokud tedy dojde k zarušení signálu na vodičích šum na obou vodičích bude téměř stejný a při rozdílu napětí na vodičích se šum vyruší.[29]



Obrázek 3.3: Hardwarové rozhraní sběrnice CAN[29]

3.2.3 Standardní CAN rámec

Komunikační protokol CAN je protokol s detekcí kolizí, vícenásobným přístupem a arbitráží na základě priority zpráv (CSMA/CD+AMP). CSMA znamená, že každý uzel na sběrnici musí čekat na po stanovenou dobu v nečinosti, než se pokusí odeslat zprávu. CD+AMP znamená, že kolize se řeší pomocí bitové arbitráže na základě předem naprogramované priority každé zprávy v identifikátoru. Přístup na sběrnici vždy získá identifikátor s vyšší prioritou. To znamená, že poslední logický vysoký bit v identifikátoru pokračuje ve vysílání, protože má nejvyšší prioritu. Protože každý uzel na sběrnici se podílí na zápisu každého bitu,(v okamžiku, kdy je zapisován), rozhodující uzel ví, zda umístil na sběrnici logickou jedničku. [17]



Obrázek 3.4: Rámec standardního CAN[17]

Na obrázku 3.4 je vidět struktura zprávy standardního CAN protokolu. V následujících odrážkách jsou vysvětleny jednotlivé části rámce. Převzato z [17]

- SOF – The single dominant start of frame (SOF) bit označuje začátek zprávy a používá se k synchronizaci uzlů na sběrnici po jejich neaktivitě
- Identifier – The Standard CAN 11-bit identifier určuje prioritu zprávy. Čím menší binární hodnota tím větší je priorita zprávy.
- RTR – The single remote transmission request (RTR) bit je dominantní, když je vyžadována informace od jiného uzlu. Všechny uzly přijímají tuto žádost, ale identifikátor určuje specifikovaný uzel. Odpovídající data jsou také přijímána všemi uzly a využívána jakýmkoli uzlem, který má o data zájem. Tímto způsobem jsou všechna data používaná v systému uniformní.
- IDE – A dominant single identifier extension (IDE) bit znamená, že se přenáší standardní CAN identifikátor bez žádných rozšíření.
- r0 – Reserved bit (rezervovaný bit pro budoucí užití).
- DLC – The 4-bit data length code (DLC) obsahuje číslo reprezentující počet bytů přenášených dat.
- Data – může být přeneseno až 64 bitů aplikačních dat.
- CRC – The 16-bit (15 bitů plus oddělovač) cyclic redundancy check (CRC) obsahuje kontrolní součet(počet přenesených bitů) předchozích aplikačních dat pro detekci chyb.
- ACK – Každý uzel, který přijme přesnou zprávu, přepíše tento recesivní bit v původní zprávě na dominantní bit, což indikuje, že byla odeslána zpráva bez chyb. Pokud přijímající uzel detekuje chybu a ponechá tento bit recesivní, zahodí zprávu a odesílající uzel zprávu po re-arbitraci opakuje. Tímto způsobem každý uzel potvrzuje (ACK) integritu svých dat. ACK je tvořen 2 bity: jeden je bit potvrzení a druhý je oddělovač.
- EOF – Toto sedmibitové pole označující konec rámce (EOF, end-of-frame) signalizuje konec CAN rámce (zprávy) a deaktivuje bit stuffing, čímž indikuje chybu stuffing, pokud je dominantní. Když se během normálního provozu objeví pět po sobě jdoucích bitů stejné logické úrovně, do dat je vložen bit opačné logické úrovně.
- IFS – Tento 7bitový mezirámcový prostor (IFS, interframe space) obsahuje čas potřebný pro kontrolér, aby přesunul správně přijatý rámec na jeho správné místo v oblasti vyrovnávací paměti zpráv.

3.2.4 Detekce chyb

Komunikační protokol CAN zahrnuje množství postupů pro kontrolu chyb a omezení poruch, které zvyšují jeho robustnost a spolehlivost. Rámec dat viz obrázek 3.4, nejběžnější typ zprávy, se skládá z pole arbitráže, datového pole, pole CRC

a pole ACK. Pole arbitráže obsahuje 11 nebo 29bitový identifikátor(rozšířený CAN) a RTR bit, který je pro datové rámce dominantní. Datové pole může obsahovat až osm bajtů dat, zatímco pole CRC zahrnuje 16bitový kontrolní součet pro detekci chyb. Pole ACK pak umožňuje ostatním uzlům potvrdit přijetí zprávy bez chyb.[17]

Dálkový rámec slouží k vyžádání přenosu dat od jiného uzlu, ale na rozdíl od datového rámce neobsahuje žádná data a je označen recesivním RTR bitem. Chybový rámec, vyvolaný detekcí chyby v zprávě, vede k vysílání chybového rámce všemi uzly v síti a automatickému opětovnému přenosu zprávy vysílacím uzlem. Proti opakovanému vysílání chybových rámců zabezpečuje systém pomocí počítadel chyb v kontroléru CAN.

Protokol CAN také zahrnuje procedury pro kontrolu chyb na úrovni zpráv pomocí pole CRC a potvrzení, jakož i na úrovni jednotlivých bitů, kde se monitoruje shoda vysílaných a přijímaných bitů. Dále je zde kontrola formátu zprávy a pravidlo bit-stuffing, které po pěti po sobě jdoucích bitech stejné logické úrovně vyžaduje vložení doplňkového bitu, čímž se zajišťuje synchronizace v síti a prevence zaměnění datového proudu za chybový rámec. Zpráva je považována za bezchybnou, pokud je poslední bit pole EOF přijat ve stavu bez chyby. Pokud je v tomto poli dominantní bit, způsobí to opakování přenosu vysílačem. Díky těmto kontrolám chyb CAN protokol zajišťuje vysokou úroveň spolehlivosti přenosu dat.[17]

3.2.5 Výhody integrace CAN s BMS

Integrace systému řízení baterií (BMS) s komunikační sítí CAN v elektrických vozidlech nabízí řadu výhod, které zvyšují bezpečnost, efektivitu a spolehlivost vozidla. CAN protokol umožňuje efektivní a spolehlivou komunikaci mezi různými kontrolními jednotkami v vozidle, včetně BMS, což je klíčové pro správu a monitorování stavu baterie.

BMS hraje zásadní roli v ochraně baterie a zajištění spolehlivého provozu elektrického vozidla tím, že monitoruje a reguluje klíčové parametry baterie, jako je napětí, proud a teplota jednotlivých článků. To zahrnuje optimalizaci stádia nabíjení a vybíjení baterie, což prodlužuje její životnost a zvyšuje celkovou efektivitu systému.

Integrace s CAN síťovým protokolem dále rozšiřuje možnosti BMS tím, že umožňuje real-time komunikaci mezi BMS a ostatními systémy vozidla. Toto propojení zajišťuje, že všechny důležité informace, jako je stav nabití (SoC), stav zdraví baterie (SoH) a teplota, jsou neustále sledovány a optimálně regulovány pro každou situaci. CAN síť navíc umožňuje BMS rychle reagovat na jakékoli anomálie nebo potenciální bezpečnostní rizika, jako jsou přetížení baterie, zkrat nebo přehřátí, čímž značně zvyšuje bezpečnost vozidla.

Díky jednoduché kabeláži a využívání společných vodičů CAN sběrnice zvyšuje flexibilitu systému a umožňuje jednodušší přidávání nových komponent do sítě nebo aktualizací systému bez nutnosti významných úprav celého elektrického systému vozidla. Toto usnadňuje inovace a umožňuje výrobcům vozidel efektivněji imple-

mentovat nové technologie a bezpečnostní opatření.

Ve výsledku integrace BMS a CANu vede k výraznému zlepšení bezpečnosti, spolehlivosti a efektivity elektrických vozidel, což přináší výhody jak pro výrobce, tak pro uživatele vozidel.[8]

3.3 Modbus

Modbus je komunikační protokol, který byl původně vyvinut společností Modicon (nyní Schneider Electric) v roce 1979 pro použití s jejich programovatelnými logickými automaty (PLC). Modbus pracuje na principu master-slave, kde jedno zařízení (master) iniciuje komunikační session a ostatní zařízení (slaves) na požadavky odpovídají. Master může být typicky PLC nebo počítač, zatímco slaves jsou často senzory, aktuátory nebo další PLC. Tento protokol se stal de facto standardem v průmyslové automatizaci a je široce používán k propojení elektronických zařízení na sériových linkách. Díky své jednoduchosti, otevřenosti a snadné implementaci je Modbus populární volbou v mnoha různých průmyslových aplikacích, včetně systémů pro správu baterií (BMS). [21]

3.3.1 Datové rámce Modbus

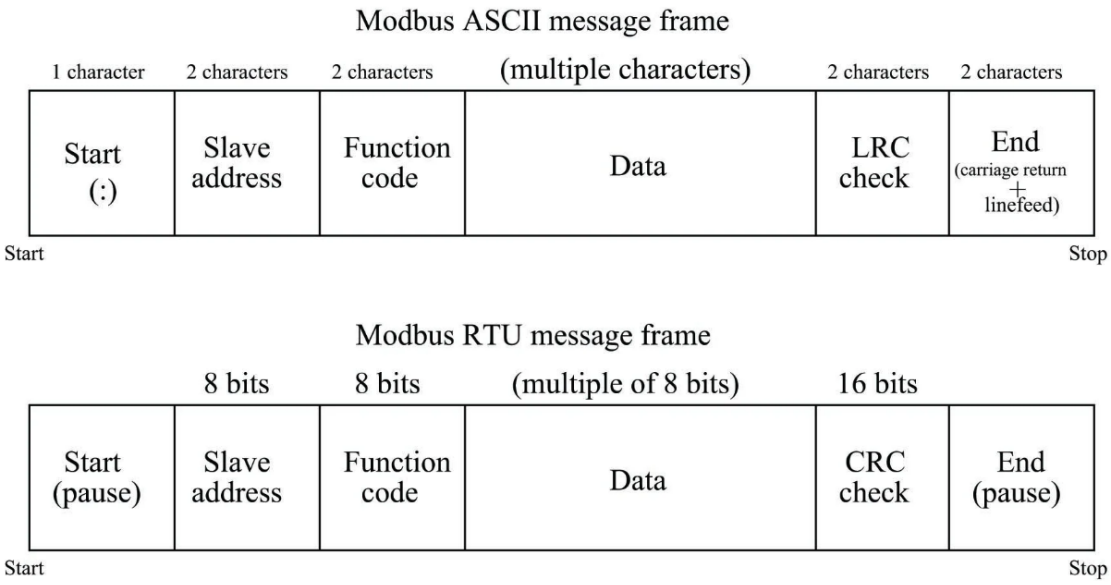
Komunikační standard Modbus definuje sadu příkazů pro čtení (příjem) a zápis (odesílání) dat mezi hlavním zařízením (master) a jedním nebo více podřízenými zařízeními (slaves) připojenými k síti. Každý z těchto příkazů je identifikován číselným kódem a v rámci Modbus rámců jsou specifikovány adresy vnitřních registrů hlavního a podřízených zařízení (zdroje a cíle dat) společně s funkčním kódem.

V rámci standardu Modbus jsou definovány dva různé formáty: ASCII a RTU. Rozdíl mezi těmito dvěma režimy spočívá v tom, jak jsou reprezentovány adresy, funkční kódy, data a bity pro kontrolu chyb. V ASCII režimu Modbus jsou adresy podřízených zařízení, funkční kódy a data reprezentovány ve formě ASCII znaků (7 bitů každý), což umožňuje jejich přímé čtení jakýmkoliv terminálovým programem (např. Minicom, Hyperterminal, Kermit atd.), který zachytává sériový datový proud. To zjednodušuje ladění: data v rámcích Modbus jsou přímo čitelná v lidsky srozumitelné formě. V režimu RTU jsou adresy zařízení, funkční kódy a data vyjádřena v čistě binární formě. Pro oba režimy se používají různé techniky kontroly chyb.

Jak je vidět z porovnání obou rámců (viz obrázek 3.5), ASCII rámce vyžadují téměř dvojnásobek bitů ve srovnání s RTU rámci, což činí Modbus ASCII pomalejším než Modbus RTU při jakékoliv dané datové rychlosti.

Obsah pole „Data“ se výrazně liší v závislosti na tom, která funkce je vyvolána, a na tom, zda je rámec vydán hlavním zařízením nebo podřízeným zařízením.

Protože Modbus je striktně protokol „vrstvy 7“, tyto zprávy jsou obvykle vloženy do jiných datových rámců definovaných protokoly nižší úrovně. Například standard



Obrázek 3.5: Datové rámce Modbus[16]

Modbus TCP zapouzdří jednotlivé Modbus datové rámce jako pakety TCP/IP, které jsou poté (obvykle) znovu zapouzdřeny jako Ethernetové pakety, aby dorazily k cílovému zařízení. Tento „vícevrstvý“ přístup, dělající z Modbusu high-level protokol, se může zdát neohrabaný, ale nabízí velkou flexibilitu, protože rámce Modbus mohou být zasílány téměř přes jakýkoli typ virtuální nebo fyzické sítě.[16]

3.3.2 Funkční kódy a adresy dat

Funkční kódy Modbus jsou číselně kódované a určují typ operace, jako je čtení nebo zápis do paměti zařízení. Základní funkční kódy zahrnují:

Čtení výstupních cívek (01) a vstupních kontaktů (02), které jsou obvykle jednobitové. Čtení a zápis registrů určených pro uchování dat (tzv. „holding“ registery, kód 03 a 06) a analogových vstupních registrů (04), které jsou obvykle 16bitové. Zápis jednotlivých (05) nebo více výstupních cívek (15) a více „holding“ registrů (16).

Adresace v Modbus je definována standardem Modbus 984, který stanovuje pevné numerické adresy pro různé typy dat v zařízení. Tyto adresy jsou rozděleny do několika rozsahů, přičemž každý rozsah má přiřazen určitý typ dat (např. diskrétní výstupy, diskrétní vstupy, analogové vstupy, a „holding“ registry). Adresy začínají od čísla 1, což je odlišné od mnoha jiných digitálních systémů, které začínají od nuly.

Mapování Modbus adres je zásadní pro integraci s moderními zařízeními. Výrobci zařízení obvykle poskytují dokumentaci s mapovacími referencemi, jež umožňují identifikovat, které Modbus adresy odpovídají konkrétním registrům nebo bitům v zařízení. Toto mapování může být kritické, protože reálná šířka bitů proměnné nemusí přesně odpovídat 16bitové šířce Modbus registrů, což vyžaduje specifické

nastavení (např. užití dvou po sobě jdoucích registrů pro 32bitové proměnné).[16]

3.3.3 Relativní adresování

Relativní adresování v Modbus je založeno na principu, že každá čtecí nebo zapisovací funkce operuje pouze v konkrétním omezeném adresním prostoru, a proto není nutné v datovém rámci uvádět celou adresu.

Například: Funkční kód 02 čte diskrétní vstupní registry s absolutním rozsahem adres 10001 až 19999. U příkazu pro čtení tedy není potřeba specifikovat první číslici adresy registru. Pro lepší pochopení relativního adresování lze použít analogii s hotelem s více patry, kde první číslice každého čísla pokoje odpovídá patru. Toto uspořádání umožňuje specifikaci polohy pokoje dvěma způsoby - pomocí absolutního čísla pokoje nebo relativního čísla pokoje na daném patře.

Relativní adresy v Modbus začínají od nuly, zatímco absolutní adresy začínají od jedničky, což vytváří dodatečný posun o 1 mezi relativní a odpovídající absolutní adresou.

Příklady použití relativního adresování v praxi: Čtení obsahu kontaktního registru 12440: funkční kód 02 Modbus; relativní adresa 2439. Čtení obsahu analogového vstupního registru 30050: funkční kód 04 Modbus; relativní adresa 49. Čtení obsahu holding registru 41000: funkční kód 03 Modbus; relativní adresa 999. Zápis více výstupních cívek v registru 00008: funkční kód 15 Modbus; relativní adresa 7.

V každém případě se relativní adresa přičte k první adrese daného rozsahu, aby se dosáhlo absolutní adresy v zařízení Modbus. [16][22]

Nevýhody Modbusu

Modbus má i přes své výrazné rozšíření řadu nevýhod, které je potřeba brát v úvahu před jeho použitím. Například ho není vhodné používat tam, kde jsou potřeba malé odezvy odpovědí, i při vyšších rychlostech přenosu je hlavní zpoždění v odezvách způsobeno zpracováním dat na straně slavů, nikoli časem přenosu.

Modbus není vhodné použít ani pro Event-driven architekturu. Sítě Modbus (jak RTU, tak TCP) nejsou navrženy pro efektivní zvládání událostmi řízených operací, jsou určeny pro dotaz->odpověď komunikaci.

Také není bezpečný pro přenos citlivých dat, protože Modbus nemá vestavěné bezpečnostní funkce, což ho činí nevhodným pro přenos citlivých nebo důvěrných dat v sítích, kde by mohl být problém s neoprávněným přístupem.

Modbus také omezuje kapacita přenosu dat kvůli omezení velikosti paketů na přibližně 120 bajtů, což vyžaduje několik zpráv pro větší přenosy dat.

Protokol není ideální pro reprezentaci složitých datových modelů, protože nepodporuje složité strukturování dat, což ho činí nevhodným pro reprezentaci propojených dat, ve kterých je vytvořena nějaká objektová struktura.

Modbus není nejlepší volbou také pro řídicí aplikace, které vyžadují přesné načaso-

vání a rychlé reakce, zejména v systémech s mnoha uzly, kvůli jeho half-duplexní povaze.

Modbus také není vhodné použít pro síťě s velkým počtem slavů na 1 mastera, protože jsou adresovány pomocí 1 bytu a jejich maximální počet je 247 i méně v závislosti na použitém médiu.[14]

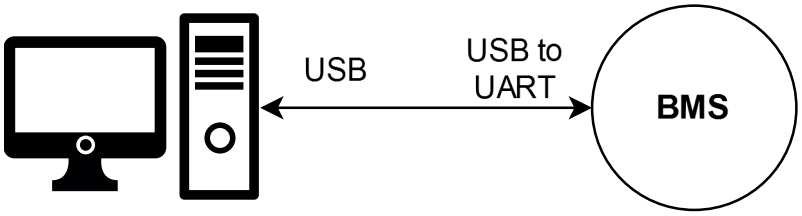
3.3.4 Shrnutí

Modbus je populární a rozšířený komunikační protokol kvůli jeho univerzálnosti a dlouhodobé podpoře Modbus organizace, která jej udržuje. Díky tomu, že se jedná o otevřený software, je kompatibilní s mnoha průmyslovými zařízeními a je snadno integrovatelný do existujících systémů. Protokol je jednoduchý na implementaci a provoz, což zjednodušuje vývoj a pozdější údržbu kódu. Jednou z dalších výhod Modbusu je jeho větší volnost na transportním protokolu, kdy lze Modbus provozovat po ethernetu(TCP/IP) nebo po různých sériových přenosech jako je RS232, RS-422, RS-485 atd.

4 Návrh komunikačního protokolu

4.1 Požadavky

Komunikační protokol pro komunikaci servisní aplikace a BMS byl navrhován na základě následujících faktů a požadavků. Servisní aplikace bude provozována na x86-64 počítačích/noteboocích s operačním systémem Windows. Jediná možnost jak připojit BMS k PC je za pomocí USB, kdy BMS bude komunikovat přes USB pomocí USB-UART(viz obrázek 4.1) převodníku a komunikuje pře sériovou linku. Na komunikační protokol není požadována přenositelnost mezi různými zařízeními, ale bude implementován s konkrétní proprietární BMS a vyvíjenou servisní aplikací na míru. Proto je možné implementovat nestandardní protokol i na straně BMS. Díky tomu vzniká jistá flexibilita návrhu a je možné protokol zjednodušit nebo navrhnout přesně na míru dané aplikace.



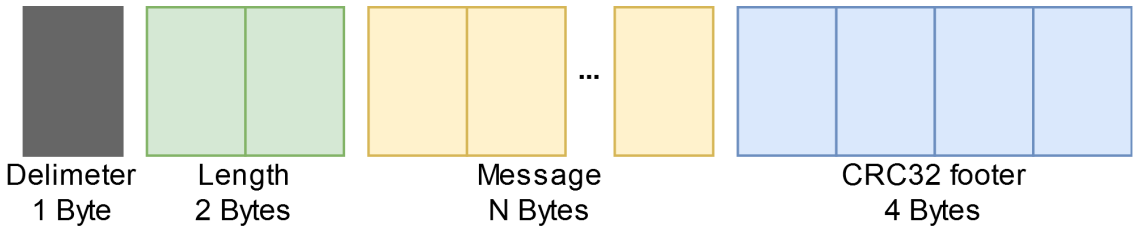
Obrázek 4.1: Připojení BMS k PC

4.2 Topologie sítě

V rámci komunikace PC a BMS vzhledem k dostupnému komunikačnímu rozhraní je vidět, že po virtuálním sériovém portu bude možná komunikace maximálně jedné aplikace a maximálně jedné BMS. Proto lze z návrhu komunikačního protokolu vynechat rozlišování zařízení a jejich případnou identifikaci v zasílaných zprávách tam a zpět.

4.3 Struktura zprávy

Pro komunikaci servisní aplikace a BMS byla navržena následující struktura zprávy. Zpráva se skládá ze 4 hlavních částí, oddělovače, délky dat, samotných dat a zápatí.



Obrázek 4.2: Struktura zprávy

- **Oddělovač** – Každá zpráva začíná oddělovačem. Oddělovač má velikost 1 byte. Hodnota oddělovače je zvolena jako náhodná hodnota(0xbc), která se nemění a je pevně daná. Oddělovač je ve zprávě obsažen z důvodu spolehlivějšího parsování příchozích zpráv.
- **Délka dat** – je 2 bytový kód jehož hodnota udává délku aplikačních dat v bytech, které následují v další části zprávy. Při velikosti 2 bytů je tedy možné zakódovat nejvyšší možné číslo 65 535. Data jsou zakódována v LSB formě. I při zakódování čísla, které je možné zakódovat do 1 bytu, se délka kódu nemění.
- **Data** – samotný datový obsah zprávy. Jeho délka je určena předchozími 2 byty a je možné zasílat 1 až 65 535 bytů dat + ostatní části zprávy.
- **Zápatí** – Po datových bytech následují 4 byty zápatí. V zápatí je obsažen hash (Cyklický redundantní součet) vytvořený funkcí CRC32 pro kontrolu integrity zprávy po přijetí.

4.4 Detekce Chyb

Pro detekci chyb při přenosu mezi aplikací a BMS byl použita funce CRC32, neboli 32bitová cyklická redundanční kontrola. CRC32 je algoritmická metoda používaná pro detekci chyb v digitálních datech. Vychází z principu polynomiálního dělení datového bloku, přičemž výsledný zbytek tohoto dělení tvoří kontrolní součet. CRC32 je efektivní v odhalování typických chyb, které se objevují při přenosu dat, jako jsou chyby v jednotlivých bitech nebo chybné bloky bitů. Díky své schopnosti generovat pevnou, 32bitovou kontrolní hodnotu z libovolně dlouhých dat je CRC32 hojně využíván v komunikačních protokolech, souborových systémech a dalších aplikacích, kde je kritická integrita dat.[2]

Funkce CRC32 byla zvolena kvůli své relativní efektivitě z hlediska výpočetní náročnosti, zejména v porovnání s komplexnějšími hashovacími funkcemi jako jsou SHA-256 nebo MD5. Na druhou stranu, kryptografické hashovací funkce provádějí složitější transformace na vstupních datech, což zvyšuje jejich odolnost proti útokům. Proto je CRC32 rychlejší, ale poskytuje méně bezpečnosti a je vhodnější pro základní kontrolu integrity dat než pro bezpečnostní aplikace. V rámci servisní aplikace na PC by byla výpočetní náročnost mezi SHA-256 a CRC32 zanedbatelná, ale hardware jednotky BMS může být omezenější a v rámci tohoto use-casu nejsou výhody jiných hašovacích funkcí využívány.

4.5 Typy zpráv

První byte v datech zprávy je rezervován pro identifikaci typu zaslané zprávy. Pro potřeby aplikace byly definovány 4 identifikátory, které jsou zakódovány jako celé číslo.

První nejdůležitější a nejpoužívanější identifikátor je **Request**. Samovypovídající název identifikuje typ zprávy, na kterou je protějším zařízením očekávána odpověď. Requesty tvoří většinu komunikace mezi aplikací a BMS.

Druhým identifikátorem zprávy je **Response**, který označuje zprávu jako odpověd na předchozí request. Komunikace vždy probíhá jako Request -> Response, kdy zařízení, které odeslalo Request, vždy čeká na odpověď(po stanovenou dobu před TimeOutem). Není možné zasílat více Requestů najednou a poté očekávat více odpovědí za sebou. Tímto BMS přijímající Request nemusí být zatěžována ukládáním Requestů do fronty a jejich následným postupným zpracováním.

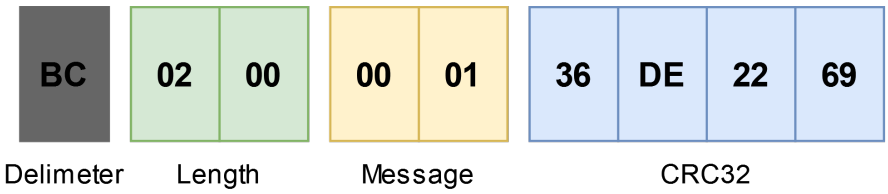
Třetím identifikátorem zprávy je **Ping**, ten se používá pokud aplikace připojená k BMS nezasílá periodicky žádné požadavky BMS, a tím neudržuje spojení otevřené. Komunikace se ukončuje tehdy pokud, BMS nedostala určenou dobu žádný další požadavek. Proto lze pomocí Pingu udržovat spojení bez přenášení velkého množství dat. U Pingu se očekává, že nebude mít kromě identifikátoru žádný obsah zprávy.

Posledním identifikátorem je **Close**.Ten při přijetí na straně BMS značí, že aplikace se chce řádně odpojit. Touto zprávou se komunikace mezi aplikací a BMS ukončí.

Typy requestů

Pokud je zpráva označena jako request, rozlišujeme také o jaký requestu se jedná, jinak řečeno, jakou operaci nebo data požaduje. Typ requestu je také zakódován v jednom bytu a jednotlivé operace jsou sekvenčně zakódovány jako celá čísla.
V následujícím výčtu jsou různé typy requestů.

Device info



Obrázek 4.3: Požadavek - Device Info

Obrázek 4.4: Odpověď - Device Info

Device info je požadavek na strukturu BMS. Na obrázku 4.3 je vidět obsah zprávy v hexadecimální reprezentaci. Ve žlutém políčku Message první byte reprezentuje typ zprávy. Typ zprávy je v tomto případě request. Druhý byte identifikuje typ requestu. Tento požadavek je zakódován hodnotou 1.

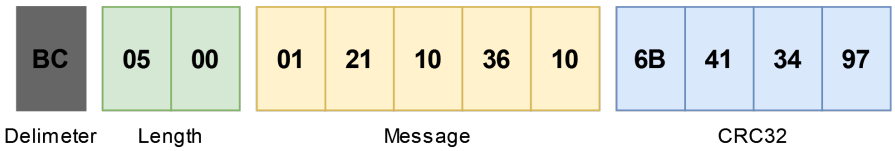
Struktura BMS je reprezentována následným způsobem. První byte značí počet modulů v BMS. Po tomto bytu následuje pro každý modul jeho počet článků. Na obrázku 4.4 je vidět odpověď na požadavek. První byte v žlutém poli Message identifikuje typ zprávy jako response. Druhý byte značí počet modulů v BMS, v tomto případě 2. Další 2 byty reprezentují počet článků v jednotlivých modulech.

Cell Voltages

Cell Voltages je požadavek na hodnoty napětí všech článků v konkrétním modulu BMS. Na obrázku 4.5 je vidět struktura požadavku. První pole zprávy byte **0x00** značí typ zprávy, tedy request. Druhý byte **0x02** značí typ requestu, tedy požadavek na hodnoty napětí článků. Třetí byte zprávy **0x00** značí modul, ze kterého mají data pocházet.

Na obrázku 4.6 je vidět příklad odpovědi na požadavek. První byte(**0x01**) zprávy značí typ zprávy(response). Dále jsou ve zprávě zakódována napětí jednotlivých článků modulu. V tomto příkladě má modul 2 články. Každá hodnota napětí je zakódována jako celé číslo do dvou bytů. Hodnota napětí je tedy v rozmezí 0-65536. Zakódovaná hodnota reprezentuje napětí v mV. Hodnoty jsou zakódovány v LSB

Obrázek 4.5: Požadavek - Cell voltages

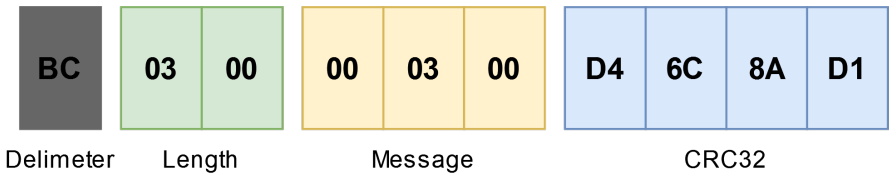


Obrázek 4.6: Odpověď - Cell voltages

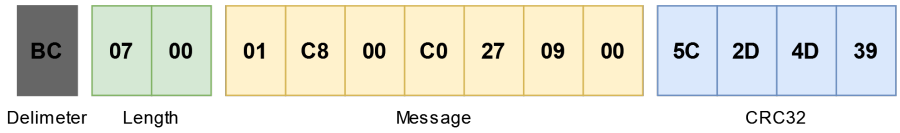
formě. To znamená, že méně významný byte je zakódován na prvním místě, stejně jako délka zprávy. Počet zakódovaných hodnot napětí pro jednotlivé články může být libovolný počet, který je ale limitován maximální délkou zprávy viz 4.3.

Module Data

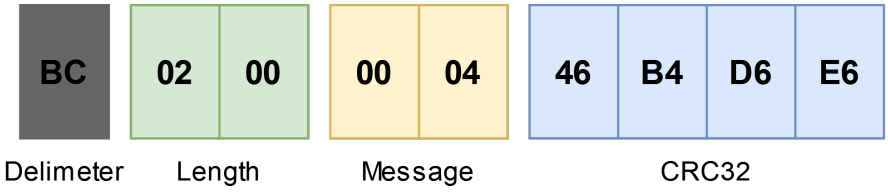
Požadavek Module Data získává data o konkrétním modulu. Odpověď požadavku(viz obrázek 4.8) obsahuje teplotu a proud konkrétního modulu. Teplota je zakódována jako první. Pro teplotu jsou vyhrazeny 2 byty, do kterých se zakódovává celé číslo v rozsahu -32,768 až 32,767 a zakódovává teplotu v desetinách stupně Celsia. Druhá zakódovaná hodnota je 4 bytová hodnota proudu v Modulu. Zakódovaná hodnota je celé číslo a představuje proud v mA.



Obrázek 4.7: Požadavek - Module data



Obrázek 4.8: Odpověď - Module data



Obrázek 4.9: Požadavek - BMS Config

Config Data

Požadavek Config Data získává konfigurační proměnné a jejich současné hodnoty v BMS. Odpověď na tento požadavek je serializované JSON pole klíčů a hodnot konfiguračních proměnných v ASCII kódu. De-serializovaná data mohou vypadat takto:

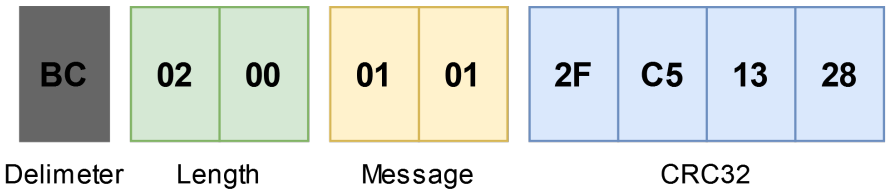
```
[
  { k: 'n-cells', v: 3 },
  { k: 't-meas', v: 1000 },
  { k: 't-ftti', v: 1000 },
  { k: 't-cyclic', v: 1 },
  { k: 'i-sleep-oc', v: 30 }
]
```

Update BMS Config

Požadavek Update BMS Config nastavuje konfigurační proměnnou BMS na novou hodnotu. Za identifikátor požadavku se vloží serializovaný JSON klíče konfigurační proměnné a nové hodnota.

```
{'k':'n-cells','v':3}
```

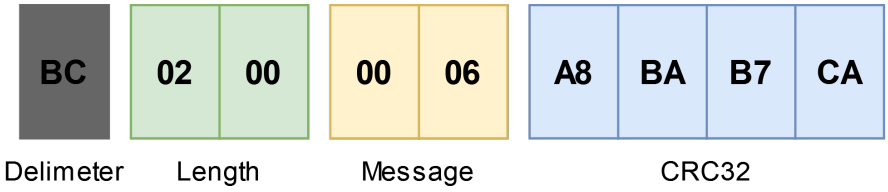
BC 17 00 00 05 7B 22 6B 22 3A 22 6E 2D 63 65 6C 6C 73 22 2C 22 76 22 3A 33 7D 65 12 F3 6F
V odpovědi na požadavek je bool hodnota značící úspěch či neúspěch operace.



Obrázek 4.10: Odpověď - Update BMS Config

BMS Data

BMS Data jsou jako 4.5 požadavek na proměnné BMS, jen s tím rozdílem, že jsou určená jen pro čtení a obsahují obecná data o BMS.

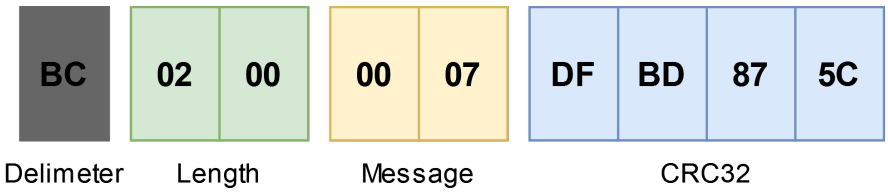


Obrázek 4.11: Požadavek - BMS Data

BMS Events

BMS Events je požadavek na data o stavu BMS. Události můžou informovat o různých překonaných limitech nebo chybných stavech potřebných k debugování BMS. Odpověď na tento požadavek je serializované JSON pole událostí a jejich stavů. Nenastavená hodnota události znamená, že je stav v pořádku.

```
[
    { k: 'Cell connection', v: 'err' },
    { k: 'FET O.T.P', v: 'err' },
    { k: 'CHG O.C.P', v: 'err' },
    { k: 'PCB O.T.P', v: 'err' },
    { k: 'Max module V delta', v: '' },
    { k: 'Max cell V delta', v: 'err' }
]
```



Obrázek 4.12: Požadavek - BMS Events

4.6 Validní zpráva

Validní zpráva je taková, která odpovídá následujícím požadavkům a je zahájena komunikace mezi aplikací a BMS.

- Každá validní zpráva začíná oddělovačem s konkrétní hodnotou.
- Každá validní zpráva obsahuje po oddělovači 2 byty obsahující délku aplikačních dat. Pokud délka i přes validitu dat a správný validní hash zprávy, neodpovídá, zpráva se vyhodnotí jako nevalidní.
- Validní zpráva má délku dat v rozmezí 1-65535 bytů.

- Validní zpráva končí zápatím o délce 4 bytů. Hash obsažený v zápatí následně musí odpovídat novému hashi dat po jejich přijetí na straně příjemce.

4.7 Zahájení komunikace

Po zahájení komunikace na úrovni sériové linky musí servisní aplikace zahájít komunikaci s BMS. Zahájení komunikace probíhá jednoduchým handshakem, kdy aplikace pošle standardní zprávu viz(4.2), která obsahuje 4 Bytovou konstantu(0x6F, 0x9A, 0x3E, 0x8D). Pokud zařízení na druhé straně odpoví stejnou zprávou, víme,že je dotazováno správné zařízení, a že implementuje stejný komunikační protokol. Pokud zařízení neodpoví, nebo odpoví špatnou zprávou. Dotaz se opakuje dle nastavení intervalu timeoutu v aplikaci nebo maximálního počtu pokusů na obdržení validní zprávy.

4.8 Komunikace po sběrnici CAN

V rámci této práce je řešena komunikace aplikace a BMS po sériové lince. V průmyslu je ale běžné, že komunikace u některých zařízení probíhá po sběrnici CAN. Proto je v následujícím textu nastíněn popis případného řešení takové komunikace.

Vzhledem k tomu, že komunikace po CAN sběrnici je velmi odlišná té po sériové lince mezi BMS a aplikací řešené v rámci této práce, komunikační model popsaný v této kapitole určený pro sériovou linku by se nepoužil. Místo toho by se využil již existující komunikační protokol v závislosti na typu zařízení. Před pokusem o komunikaci se zařízením na CAN sběrnici je nutné znát konkrétní CAN protokol, který zařízení používá. To zahrnuje identifikátory zpráv (ID), na které zařízení naslouchá, formát datových polí uvnitř CAN rámců a jaké typy odpovědí zařízení posílá. Běžné protokoly zahrnují OBD-II pro vozidla, CANopen pro průmyslové systémy a proprietární protokoly pro specifická zařízení.

Také během vývoje servisní aplikace by se měla zohlednit možnost komunikace po CAN sběrnici. Bylo by potřeba vybrat vhodné nástroje a knihovny umožňující komunikaci po CAN sběrnici. Zároveň by bylo nutné se zaobírat výběrem hardwarového CAN adaptéru, který umožňuje připojení počítače k CAN sběrnici. Běžné jsou USB-to-CAN adaptéry od výrobců jako jsou PEAK-System, Kvaser nebo Vector. S využíváním komunikace po CAN sběrnici by se mělo při implementaci komunikace počítat s chybami, které by mohly nastat, vzhledem k použití CAN protokolu, jako jsou různé kolize mezi zprávami a prioritizací zpráv. Nakonec by bylo nutné správně interpretovat a vizualizovat přijímaná data z CAN sběrnice.

5 Vývoj servisní aplikace

5.1 Případ užití servisní aplikace

Při servisu a diagnostice má uživatel BMS zařízení fyzicky u sebe. Zařízení nemusí být schopné komunikovat vzdáleně po síti, např. po ethernetu, kvůli jeho stavu nebo nedostupnosti síťového připojení. Předpokládá se, že v případě použití servisní aplikace, není zařízení ve funkčním stavu nebo je potřeba BMS zkontrolovat po servisním úkonu před jeho nasazením do produkce.

Uživatel bude mít k dispozici vlastní notebook nebo PC do kterého se BMS připojí pomocí USB a bude komunikovat přes UART.

5.2 Volba frameworku pro desktopovou aplikaci

Electron

Electron je framework pro vývoj desktopových aplikací, který využívá webové technologie jako jsou HTML, CSS a JavaScript. Tato technologie usnadňuje vývoj pro vývojáře se znalostí webových technologií. Díky tomu může být vývoj aplikací rychlejší a efektivnější. Electron nabízí také multiplatformní podporu, což znamená, že aplikace vytvořená pomocí Electronu může být spuštěna na různých operačních systémech, včetně Windows, macOS a Linuxu. Další výhodou Electronu je velké množství dostupných knihoven a pluginů, které mohou vývojáři využít při vytváření svých aplikací. Nicméně Electron může vyžadovat větší množství systémových prostředků a někdy může být náchylný na problémy s výkonem. Velikost aplikací vytvořených pomocí Electronu může být také větší kvůli balení webových technologií.[13]

Qt

Qt je cross-platformní framework pro vývoj aplikací, který je známý pro svou vysokou výkonnost a efektivitu. Qt umožňuje vývojářům vytvářet aplikace, které běží na různých operačních systémech, včetně Windows, macOS a Linuxu. Jednou z hlavních výhod Qt je jeho rozsáhlá dokumentace a silná komunita, která poskytuje podporu a pomoc při vývoji aplikací. Qt nabízí také několik nástrojů a knihoven pro práci s uživatelským rozhraním, grafikou a multimédii. Nicméně učení se jazyka Qt (Qt/C++) může být pro některé vývojáře obtížné a licenční model Qt může být omezující pro komerční projekty.[26]

JavaFX

JavaFX je framework pro vývoj desktopových aplikací, který je integrován do platformy Java. Jednou z hlavních výhod JavaFX je jeho jednoduché použití pro vývojáře, kteří znají jazyk Java. JavaFX je také cross-platformní, což znamená, že aplikace vytvořené pomocí JavaFX mohou běžet na různých operačních systémech. Další výhodou JavaFX je jeho dobrá integrace s jazykem Java a jeho ekosystémem, což umožňuje vývojářům využívat stávající znalosti a dovednosti. Nicméně je méně flexibilní než některé jiné frameworky a někdy může být jeho výkon ve srovnání s nativními aplikacemi problematický.[18]

Windows Presentation Foundation (WPF)

Windows Presentation Foundation (WPF) je framework pro vývoj desktopových aplikací pro operační systém Windows. Jednou z hlavních výhod WPF je jeho plná integrace s operačním systémem Windows, což umožňuje vývojářům vytvářet aplikace s bohatými možnostmi uživatelského rozhraní a animací. WPF také poskytuje pokročilé funkce pro práci s grafikou a multimédií. Velkou nevýhodou WPF je možnost vývoje aplikací pouze pro platformu Windows. Nevýhodou také může být nutnost znalosti značkovacího jazyka XAML a znalost .NET frameworku.[12]

GTK

GTK je cross-platformní framework pro vývoj desktopových aplikací s důrazem na Linux. Jednou z hlavních výhod GTK je, že je open-source. GTK také podporuje mnoho programovacích jazyků, včetně C, C++, Python, což umožňuje vývojářům používat jazyk, který jim nejvíce vyhovuje. Nicméně může být obtížné dosáhnout nativního vzhledu aplikace na jiných než linuxových platformách a dokumentace není vždy tak komplexní jako u jiných frameworků.[15]

5.2.1 Zvolený framework

Pro tuto aplikaci byl zvolen framework Electron kvůli následujícím vlastnostem:

Přenositelnost kódu mezi platformami: Electron umožňuje vývojářům psát aplikace jednou a spouštět je na Windows, macOS i Linux bez nutnosti zásadních úprav kódu. Toto usnadňuje správu a aktualizace aplikace napříč různými platformami.

Využití webových technologií: Díky Electronu mohou weboví vývojáři snadno přejít k tvorbě desktopových aplikací, aniž by se museli učit nové programovací jazyky nebo nástroje. To umožňuje rychlejší vývoj a snazší udržování aplikace.

Bohatá komunita a ekosystém: Electron je podporován bohatým ekosystémem pluginů a rozšíření, díky kterým je možné rychle přidávat nové funkce a integrace. Komunita kolem Electronu je velká a aktivní, což usnadňuje hledání řešení problémů a sdílení nejlepších praktik.

Integrace s Node.js: Electron integruje Chromium pro zobrazení webového obsahu

a Node.js pro backendové operace, což umožňuje aplikacím využívat široké spektrum Node.js modulů a knihoven pro přístup k systémovým zdrojům a provádění nízkoúrovňových operací.

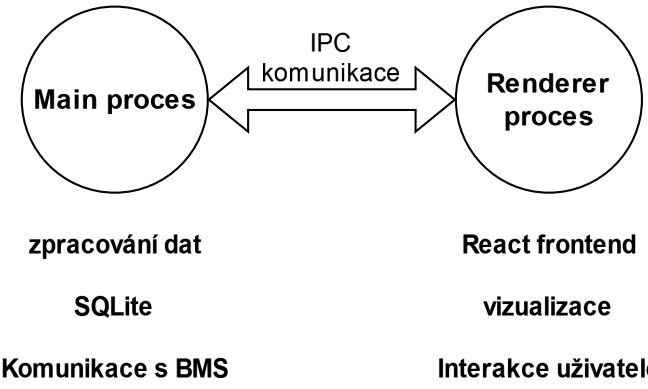
Používání v populárních projektech: Electron je využíván v mnoha populárních aplikacích jako jsou Visual Studio Code, Slack, Discord a GitHub Desktop. To svědčí o jeho spolehlivosti a škálovatelnosti pro vývoj komplexních aplikací.

Podpora pro automatické aktualizace: Electron podporuje systém automatických aktualizací, čímž umožňuje vývojářům snadno distribuovat aktualizace pro své aplikace, což zvyšuje bezpečnost a zlepšuje uživatelskou zkušenost.

Snadný přístup k systémovým funkcím: Vzhledem k integraci s Node.js, Electron umožňuje aplikacím se snadno interagovat se systémovými funkcemi jako jsou souborové systémy, síťové požadavky a další, což je často náročné v čistě webových aplikacích.

Electron tedy nabízí silnou platformu pro vývoj desktopových aplikací, která využívá známé webové technologie a zjednodušuje multiplatformní vývoj. Nicméně je také důležité zvážit potenciální nevýhody, jako jsou vyšší požadavky na paměť a velikost aplikace na disku oproti nativním aplikacím, což může být pro některé projekty limitující.

5.3 Struktura aplikace



Obrázek 5.1: Procesy Electron aplikace

Vzhledem k tomu, že aplikace je vyvíjena ve frameworku Electron, její základní struktura se odvíjí z výchozí struktury Electron aplikace. Ta se dělí na 2 hlavní procesy. Main a Renderer. Tato víceprocesová architektura vychází ze skutečnosti, že Electron využívá přibalené Chromium, které je primárně určené pro prohlížeče, kde se využívá více procesový model pro každé záložky prohlížeče v odděleném procesu. To zvyšuje bezpečnost před hrozbou ovlivnění chybou nebo cíleným útokem jiné záložky v prohlížeči.

Každá Electron aplikace má jeden Main proces, který slouží jako vstupní bod aplikace. Hlavní proces běží v Node.js prostředí, což znamená, že může importovat jeho modul a využívat Node.js API. Hlavním účelem Main procesu je vytvářet a spravovat okna aplikace pomocí modulu BrowserWindow. Každá instance třídy BrowserWindow vytváří okno aplikace, které načítá webovou stránku v samostatném Render procesu. S tímto webovým obsahem můžete interagovat z Main procesu pomocí objektu webContents window.

Hlavní proces také řídí životní cyklus aplikace prostřednictvím Electron modulu. Tento modul poskytuje rozsáhlou sadu událostí a metod, které můžete použít k přidání vlastního chování aplikace (například k programovému ukončení aplikace).

Aby aplikace mohla využívat jiné funkce než jen funkce z Chromia pro webový obsah, přidává Main proces také vlastní rozhraní API pro interakci s operačním systémem. Electron zpřístupňuje různé moduly, které ovládají nativní funkce pracovní plochy, jako jsou nabídky, dialogová okna a ikony v oznamovací oblasti. Tímto způsobem je možné přistupovat např. k USB po kterém aplikace komunikuje s BMS nebo přístup k disku kam se ukládají a exportují měřená data.

Z bezpečnostních důvodů je omezen přístup k Node.js API a modulům z Renderer procesu. Proto je potřeba nějakým způsobem zprostředkovávat komunikaci mezi Main a Renderer procesy. Na to se používají Preload skripty. Ty obsahují kód, který se spustí v Renderer procesu před zahájením načítání jeho webového obsahu. Tyto skripty se spouštějí v kontextu Rendereru, ale mají větší oprávnění díky přístupu k rozhraním API Node.js. V Preload skriptech je možné definovat jednosměrnou nebo obousměrnou komunikace mezi Procesy. To je v aplikaci hojně využíváno pro získávání dat z Main Procesu k zobrazování v GUI Renderer procesu nebo k ovládání chování aplikace interakcí uživatele na různé ovládácí prvky. [13]

5.3.1 Main proces

Main proces se stará o zpracování dat a backend aplikace, může přistupovat k různým prostředkům počítače jako je souborový systém nebo usb porty. Struktura kódu pro main proces je následující. V kořenovém adresáři ve složce main se nachází všechen kód pro main proces. První složkou je složka models. Ta obsahuje vlastní datové typy, které se využívají v aplikaci. Jednotlivé typy definují strukturu používaných objektů. Např. reprezentují entity databáze nebo různé datové struktury které se poté zasílají frontendu a vykreslují se např. napětí teploty a proudy vizualizované v grafech.

Ve složce prompts se nachází interface pro tvorbu oken, které po uživateli požadují nějaký vstup. Např. výběr portu, ke kterému se má aplikace připojit a zahájit komunikace s BMS. Viz 5.7

Další složkou jsou services. Ta obsahuje různé služby pro abstrakci funkčních celků aplikace např. Communication Handler(5.4), Databázovou službu 5.6 a službu pro Sériovou komunikaci. Kód main procesu je implementovaný proti rozhraní tak, aby různé funkční celky bylo možné měnit a nahrazovat např. kvůli testování.

5.3.2 Renderer proces

Renderer proces se stará o grafickou vizualizaci a interakci uživatele s aplikací. Pro tvorbu grafického rozhraní byl použit React s MUI. Struktura kódu pro renderer proces je následující. V kořenové složce se nachází složka renderer pro tento proces. Ve složce je následující struktura. Assets, je složka pro obrázky a ikony v aplikaci, které se např. používají v navigační liště. Další složka je Components. Složka components obsahuje funkční celky v aplikaci, jako jsou například grafy, navigační lišta, tabulky pro zobrazování dat, konfiguraci proměnných a zobrazování událostí. Tímto se dají komponenty znovu použít v jiné části aplikace. Další složkou je styles, která obsahuje kaskádové styly pro aplikaci. Poslední složka jsou views. Složka Views obsahuje všechny obrazovky aplikace, které jsou složené z již zmíněných komponent.

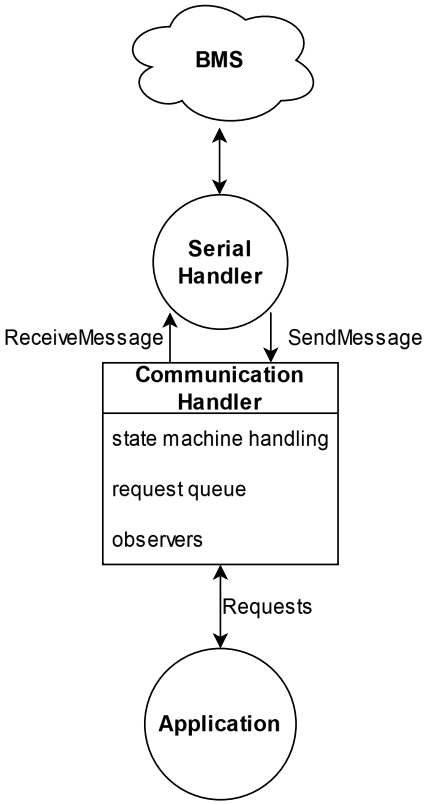
Pro vytvoření logiky a přepínání obrazovek pomocí navigační lišty byl použit React Memory Router. Ten umožňuje přepínat mezi jednotlivými obrazovkami aplikace virtuálně. To znamená, že při změně obrazovky aplikace se nemění url, jen se vykreslí jiná obrazovka. Změna url se řeší virtuálně.

5.3.3 Meziprocesová komunikace

V souboru preload jsou definovány metody, ke kterým lze přistupovat v Renderer procesu. Je možné definovat různé typy a směry komunikace. Ve všech případech se jedná o asynchronní komunikace, synchronní komunikace není možná nebo se nepoužívá kvůli možnému blokování procesu. V aplikaci se využívá jednosměrná i obousměrná komunikace. Příkladem jednosměrné komunikace může být odesílání příchozích dat z BMS z Main procesu do Renderer procesu určených k vizualizaci. Příkladem obousměrné komunikace může být požadavek na úpravu konfigurační proměnné v BMS a jeho následné zpětné potvrzení.

5.4 Komunikační třída

Komunikační třída (ve zdrojovém kódu pojmenována Communication Handler) vytváří abstrakci(viz schéma 5.2) mezi dotazy z aplikace a jednotkou BMS. Komunikační třída zastřešuje stavový automat komunikačního protokolu. Nad ním poskytuje API, které poskytuje aplikaci abstrakci pro jednoduché zasílání requestů a poskytnutí zpracované odpovědi. Pro obsluhu requestů používá třída frontu, kam se řadí requesty čekající na odeslání po sériové lince. Třída také poskytuje funkcionalitu observerů, kdy dovoluje aplikaci do svého chování implementovat funkcionalitu pracující na základě sledování současného stavu stavového automatu.



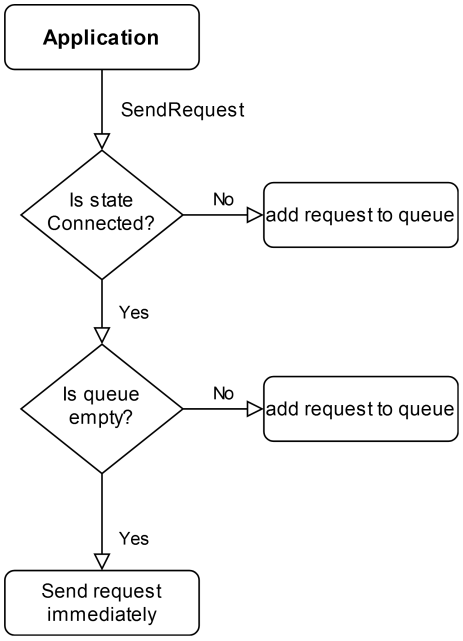
Obrázek 5.2: Obecné schéma komunikace

5.4.1 Fronta požadavků

Při používání aplikace je běžné, že aplikaci v jednu chvíli odešle více requestů BMS. Protože v komunikace není možné zasílat více požadavků najednou, je potřeba ukládat požadavky do fronty. Na obrázku 5.3 je vidět vývojový diagram řízení požadavků. Pokud je stavový automat ve stavu Connected, znamená to, že komunikační spojení s BMS je vytvořeno a čeká se na požadavky. Pokud je i fronta s požadavky prázdná, požadavek se okamžitě odešle.

5.4.2 Observers

Pokud aplikace potřebuje během svého běhu zjistit, v jakém stavu se právě nachází stavový automat, mapř. při startu zjistit, že komunikace neběží, je potřeba ji zahájit. Aplikace může do Komunikační třídy přidat observer. Observer je callback s parametrem do kterého vstupuje aktuální stav. Pokud se tedy po přihlášení observeru změní stav stavového automatu zavolají se zaregistrované observery a předá se jim současný stav. Toto se hodí pokud aplikace potřebuje řešit stavy, kdy komunikace se přepne do jednoho z chybových stavů a je potřeba obnovit komunikaci.



Obrázek 5.3: Fronta požadavků

5.5 Stavový automat komunikace

5.5.1 Stavy a přechody

Komunikace mezi aplikací a BMS byla implementována jako konečný stavový automat. Stavový automat je implementován pomocí stavů a akcí. Stav reprezentuje současný stav komunikace mezi aplikací a BMS a zároveň obsahuje všechna potřebná data ke svému účelu. Ta jsou uložena v rámci instance stavu a mají jednoznačně určenou strukturu. Stavy se používají podle návrhového stylu neměnných objektů, kdy pro změnu dat je potřeba vytvořit novou instanci stavu. Proto se data mění jen po přechodech mezi stavy, nebo při přechodu stavu na sám sebe.

V aplikaci jsou stavy reprezentovány jako třídy, které rozšiřují abstraktní třídu IState. Tato třída definuje 2 abstraktní metody. Metodu **tag** a metodu **applyAction**.

Metoda tag vrací hodnotu s type string a je určena pro debugování, kdy je potřeba při výpisu logu zjistit, o který stav se jedná. Při výpisu logu by bylo možné vypisovat přímo název třídy, z které instance stavu vznikla, ale při transpilaci typescriptu do javascriptu a jeho následného zabalení do aplikace jsou změněny názvy tříd kvůli zmenšení velikosti výsledné aplikaci na disku a obfuskaci kódu.

Abstraktní metoda applyAction je použita k aplikováním přechodů na stavy. Stavy a přechody v aplikaci jsou navrženy tak, aby bylo možné použít polymorfní volání. To znamená, že aplikace má uložený současný stav komunikace a jakýkoli přechod předává stavu pomocí metody applyAction bez jakékoliv rozhodovací logiky a znalosti konkrétního stavu. To zjednodušuje návrh kódu, zvyšuje jeho přehlednost a umožňuje stavy jednoduše testovat.

Stavy se v metodě applyAction můžou rozhodovat podle typu příchozí akce (přechodu). Každá akce má určený vlastní datový typ[1], který je mezi sebou rozlišuje a určuje, jaká data bude přechod obsahovat. Jednotlivé stavy následně akce rozlišuje pomocí typových kontrol poskytované typescriptem. Díky tomu můžeme ve stavech přistupovat bezpečněji k proměnným akcí bez rizika null pointeru při záměně objektů.

5.5.2 Přehled stavů

Initial

Stav Initial je výchozí stav při vytvoření instance komunikační třídy. Jeho přechodová funkce je příchozí požadavek aplikace na zahájení komunikace. Při aplikování přechodové funkce je odeslán požadavek na zahájení komunikace viz (4.7) a metoda vrací instanci nového stavu třídy Waiting for handshake.

Waiting for handshake

Stav Waiting for handshake setrvává, dokud nepřijde odpověď na odeslaný handshake, nebo je překročen časový limit pro odpověd. Pokud odpověd přijde ve stanoveném čase, otestuje se správnost handshake konstanty a zároveň se zkontroluje checksum v zápatí. Pokud vše odpovídá, je vrácen nový stav Connected. Pokud je překročen časový limit, nebo data nejsou korektní, je vrácen nový stav CommunicationInitFailed.

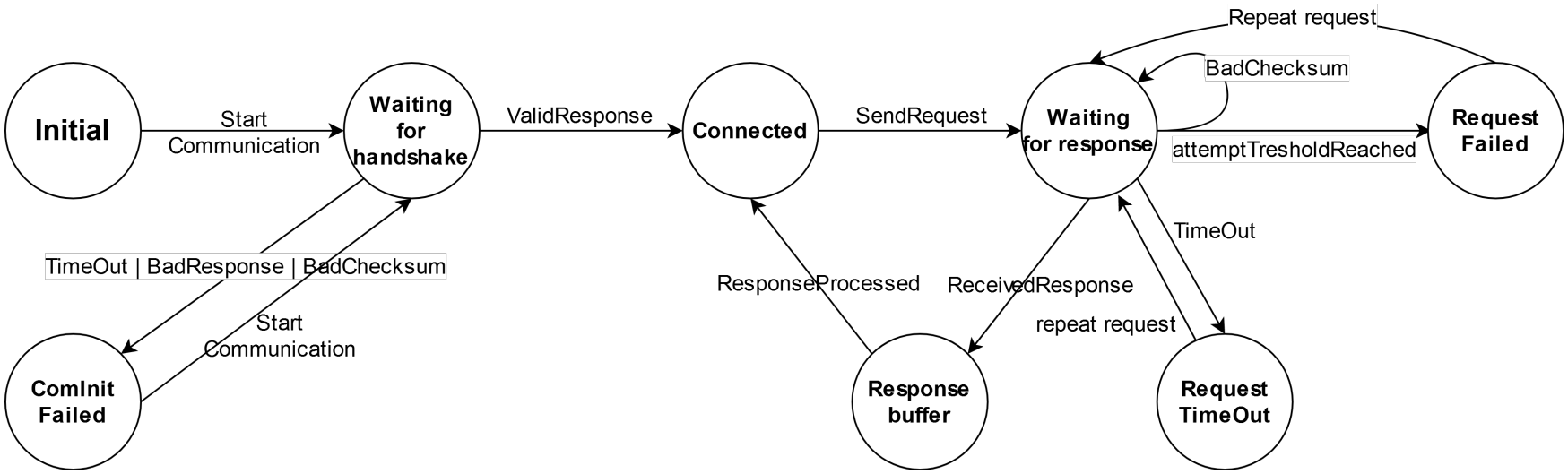
Connected

Connected je stav, ve kterém je možné posílat nové požadavky pro BMS. Pro řízení více požadavků se využívá fronta viz. 5.4.1. Při přechodové funkci na zaslání požadavku se vytvoří nová instance stavu Waiting for response s následujícími parametry. První parametr je označení pokusu o odeslání požadavku. Pokud se jedná o nový požadavek, je nastaven na nulu. Pokud byl požadavek opakován z důvodu timeoutu nebo jiné chyby, je nastaven odpovídajícím pořadím. Další parametr je outputCallback. Ten se používá při znovuodesílání požadavku. Poslední parametr je objekt časovače. Ten je zastaven po přijetí zprávy, nebo při jeho vypršení provede přechodovou funkci na stav Request TimeOut.

Waiting for response

Stav Waiting for response čeká dokud nepřijde odpověd z BMS. Tento stav řeší problém, kdy se stavový automat chová jako synchronní, ale požadavek na BMS je asynchronní událost. Tento problém je vyřešen tak, že aplikace ve stavu Connected odešle požadavek společně s callbackem, skrz který se poté vrátí odpověď.

Tento stav obsahuje 4 proměnné. Jednou z proměnných attemptCounter, stejně jako u stavu Waiting for Handshake, je při poškozených datech odpovědi vyslán



Obrázek 5.4: Stavový automat komunikace

nový požadavek, dokud nepřeteče attemptCounter. Poté se komunikace přepne do chybového stavu(Request Failed). Stejně tak pokud vyprší doba, po kterou má BMS čas odeslat odpověd, se přepne do stavu RequestTimeOut. Pokud attemptCounter nepřeteče, vynuluje se TimeOut časovač, pošle se nový request a stavový automat setrvává ve stejném stavu.

Response buffer

Response buffer je stav, jehož úlohou je uchovávat přijatá data dokud je aplikace nevyužije. Pokud aplikace data zpracuje, na stav se aplikuje přechodová funkce o přijetí dat a stav se přepne do výchozího stavu otevřené komunikace Connected. Takto probíhá komunikace stále dokola.

ComInit failed

ComInit failed je stav, který nastane po neúspěšném zahájení komunikace. Pokud se aplikace rozhodne znovu zahájit komunikaci, odešle se nový request s Handshakem a stavový automat se přepne do stavu WaitingForHandshake.

Request TimeOut

Tento stav nastává pokud vyprší definovaná doba pro získání odpovědi z BMS.

Request Failed

Tento stav nastane, pokud sice přijde odpověd z BMS, ale data byla během přenosu poškozena.

5.6 Ukládaní měřených dat

5.6.1 SQLite

Pro ukládání získaných dat během běhu aplikace bylo zvolena databáze SQLite[28], konkrétněji knihovna pro NodeJS BetterSQLite3[10].

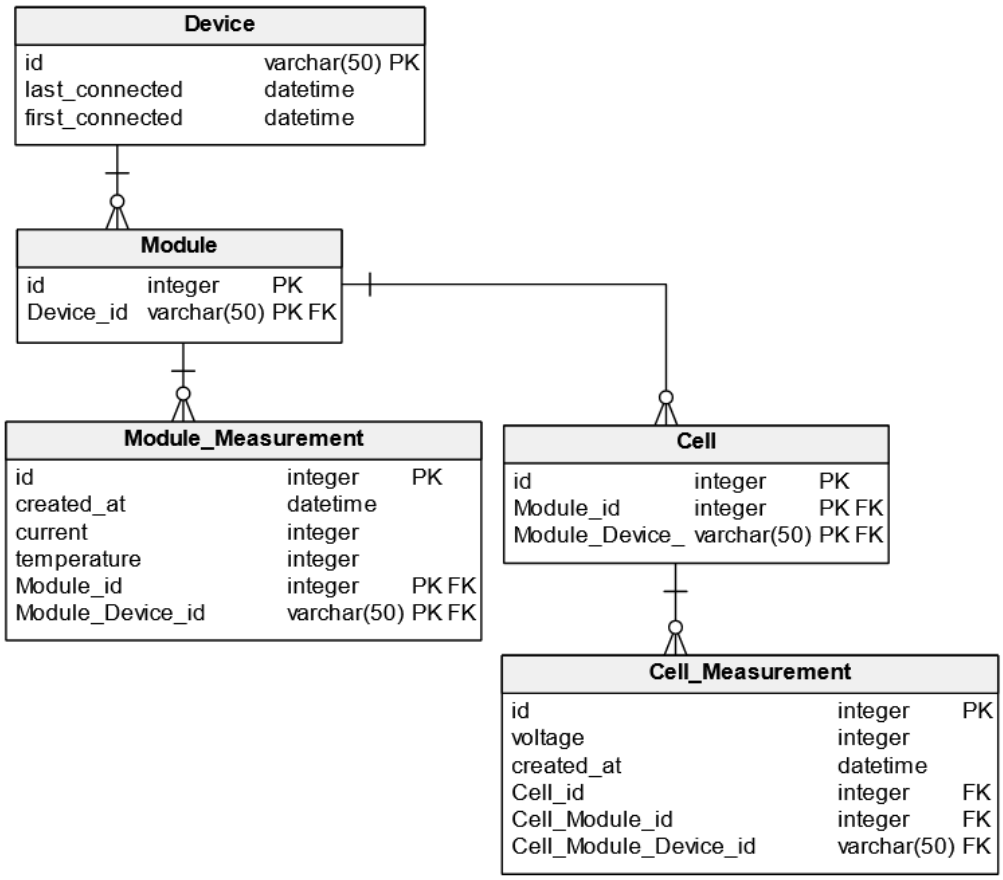
SQLite je in-process knihovna, která implementuje v sobě vnořený transakční SQL databázový engine, který neběží na serveru a nevyžaduje žádnou konfiguraci, správu a údržbu. Kód pro SQLite je open-source a je možné ho použít jak komerčně, tak soukromě bez jakéhokoli omezení a license. SQLite je po světě velmi rozšířený.

SQLite je vestavěný databázový engine. Na rozdíl od většiny ostatních databází SQL nemá SQLite samostatný serverový proces. SQLite čte a zapisuje přímo do obyčejných souborů na disku. Kompletní databáze SQL s několika tabulkami, indexy, spouštěči a pohledy je obsažena v jediném souboru. Formát databázového souboru je multiplatformní. Databázi můžete libovolně kopírovat mezi 32bitovými a 64bitovými systémy nebo mezi big-endian a little-endian architekturou. Díky

těmto vlastnostem je SQLite oblíbenou volbou jako aplikační formát souborů. Databázové soubory SQLite jsou doporučeným formátem pro ukládání dat Kongresovou knihovnou USA. SQLite nemá za cíl nahradit velké databázové systémy, ale nahrazuje vlastní implementace zápisu dat do souborů a k tomu poskytuje výhodu SQL databází.

SQLite je kompaktní knihovna. Při zapnutí všech funkcí může být velikost knihovny menší než 750 kB, v závislosti na cílové platformě a nastavení optimalizace kompilátoru. (64bitový kód je větší). A některé optimalizace kompilátoru, jako je inlining a loop unrolling, mohou způsobit, že kód bude mnohem větší). Existuje kompromis mezi využitím paměti a rychlostí. SQLite obecně běží tím rychleji, čím více paměti má k dispozici. Nicméně výkon je obvykle poměrně dobrý i v prostředích s malou pamětí. V závislosti na způsobu použití může být SQLite rychlejší než přímý I/O souborového systému.[28]

5.6.2 Databázový model



Obrázek 5.5: Databázový model

Databázový model byl zvolen tak, aby reprezentoval strukturu BMS. Na obrázku 5.5, je vidět databázový model. V současném stavu modelu je vidět, že tabulky

nemají větší množství sloupců a bylo by možné je sloučit a zmenšit jejich počet. Model byl ale navržen s předpokladem, že bude potřeba měřit i jiné veličiny, a tím by se tabulky značně rozrostly.

Tabulka Device ukládá jednotlivé BMS. Ty se rozlišují na základě COM portu, ke kterému jsou připojeny přes USB. V tabulce jsou 3 sloupce. Id je název COM portu. Last_connected, jak již název značí poslední připojení zařízení.First_connected ukazuje, kdy zařízení bylo poprvé připojeno.

Tabulka Module ukládá moduly pro jednotlivé BMS. BMS můžou mít různou strukturu, nebo některé jejich části nemusí být zapojeny, proto musí být model flexibilní pro tento požadavek. Sloupec id identifikuje konkrétní modul BMS a spolu s ID BMS tvoří primární klíč.

Tabulka Cell uchovává data o jednotlivých článcích modulů a zařízení. Každé zařízení může mít různý počet modulů a ten může mít různý počet článků.

Tabulka Module_Measurement uchovává data o měření na úrovni modulu. Uchovává proud procházející modulem a jeho teplotu.

Tabulka Cell_Measurement uchovává měření o jednotlivých článcích. V tomto případě uchová jen napětí a timestamp měření. Pokud by ale bylo potřeba měřit i jiné veličiny, lze strukturu jednoduše upravit a uchovávat další data.

5.6.3 Databázové rozhraní

Na obrázku 5.6 jsou vidět metody, které aplikace implementuje k práci s databází.

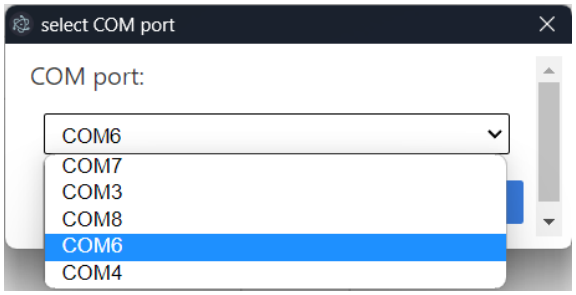
```
7   export interface IDatabase {
8     name: string;
9     connect(): boolean;
10    close(): void;
11    getAllDevices(): Device[];
12    getAllModules(): Module[];
13    getAllCells(): Cell[];
14    getAllCellMeasurements(): CellMeasurement[];
15    getAllModuleMeasurements(): ModuleMeasurement[];
16
17    insertDevice(newDevice: Device): boolean;
18    insertModule(newModule: Module): boolean;
19    insertCell(newCell: Cell): boolean;
20    insertCellMeasurement(newMeasurement: CellMeasurement): void;
21    insertModuleMeasurement(newMeasurement: ModuleMeasurement): void;
22
23    deleteDevice(deviceId: string): void;
24    deleteModule(deviceId: string, moduleId: number): void;
25    deleteCell(deviceId: string, moduleId: number, cellId: number): void;
26    deleteCellMeasurement(id: number): void;
27    deleteModuleMeasurement(id: number): void;
28
29    getCellMeasurementsAscending(
30      limit: number,
31      id: number,
32      deviceId: string,
33      moduleId: number,
34    ): CellMeasurement[];
35    getCellMeasurementsDescending(
36      limit: number,
37      id: number,
38      deviceId: string,
39      moduleId: number,
40    ): CellMeasurement[];
41    getCellMeasurementsCount(deviceId: string, moduleId: number): number;
42
43    getModuleMeasurementsAscending(
44      limit: number,
45      id: number,
46      deviceId: string,
47      moduleId: number,
48    ): ModuleMeasurement[];
49    getModuleMeasurementsDescending(
50      limit: number,
51      id: number,
52      deviceId: string,
53      moduleId: number,
54    ): ModuleMeasurement[];
55    getModuleMeasurementsCount(deviceId: string, moduleId: number): number;
56  }
```

Obrázek 5.6: Databázové rozhraní

5.7 Části aplikace

5.7.1 Připojení BMS

Po kliknutí na tlačíko zástrčky v navigační liště aplikace se uživateli zobrazí dialog s výběrem COM portu. Uživatel vybere port, na kterém je připojená BMS. Po vybrání portu aplikace naváže s BMS komunikaci.



Obrázek 5.7: připojení BMS

5.7.2 Live view

Live view(viz obrázek 5.8) je hlavní obrazovka aplikace. Po připojení BMS se zobrazí nejdůležitější hodnoty BMS. V horní části okna se zobrazuje panel s přehledem obecných informací o BMS, např. teplota minimálního a maximálního napětí článků, celkový součet napětí, počet událostí v BMS a dalších. Tento panel se dynamicky načítá z BMS, dle dat, která nám poskytne. Panel je plně responzivní a dokáže reagovat na měnící se počet zobrazovaných hodnot a šířku okna aplikace.

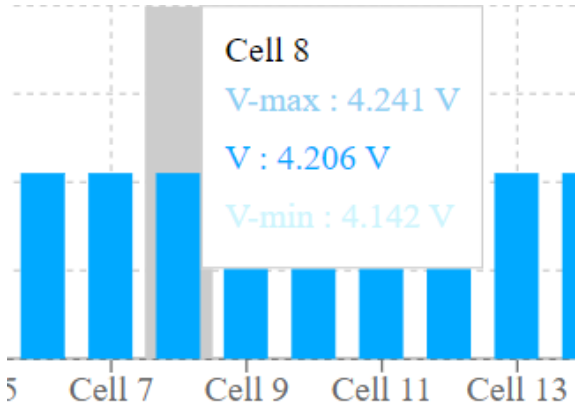
Prostřední panel ukazuje napětí jednotlivých článků ve vybraném modulu. Pokud chce uživatel vidět hodnotu konkrétního modulu, může najetím kurzoru na konkrétní článek vidět. Jeho minimální, současnou a maximální hodnotu napětí. Minimální a maximální hodnota se zaznamenává od poslední změny zobrazeného modulu nebo od připojení zařízení. Graf je schopný zobrazovat libovolné množství článků i pro moduly s ruznými počty článků. Ve výchozím stavu se data v grafu aktualizují každou sekundu.

Ve spodní části obrazovky jsou umístěny 2 grafy. Levý graf se žlutými uzly zobrazuje proud konkrétního modulu. Pravý graf se zelenými uzly zobrazuje současnou teplotu monitorovaného modulu.

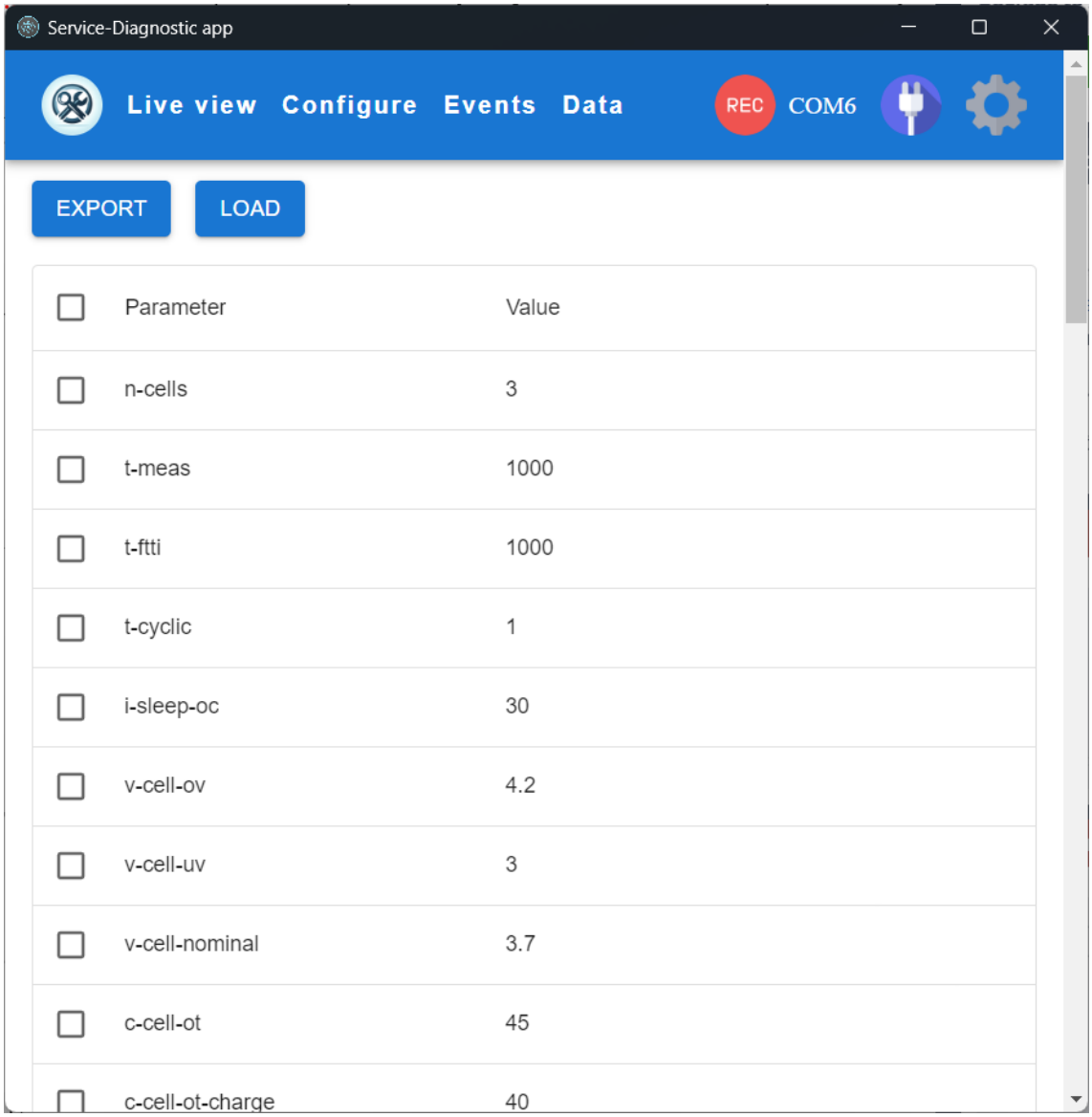
5.7.3 Configure

Configure(viz obrázek 5.10) je obrazovka pro konfiguraci vnitřních proměnných BMS. Obsahuje tabulka se dvěma sloupci. Parameter a Value. Sloupec parameter obsahuje názvy všech konfiguračních proměnných v BMS a sloupec value jejich současnou hodnotu. Při zobrazení této obrazovky se vyšle požadavek BMS a všechna

Obrázek 5.8: Live view



Obrázek 5.9: Zobrazení napětí konkrétního článku



Obrázek 5.10: Konfigurace BMS

tato data se načtou a jsou aktuální. Pokud uživatel chce nějakou proměnnou změnit. Klikne na konkrétní políčko a může hodnotu přepsat. Pokud chce změnu zrušit stiskne klávesu escape. Pokud chce hodnotu uložit, musí stisknout klávesu enter. Po potvrzení změny enterem se odešle požadavek BMS o změně proměnné. Pokud vše proběhne v pořádku, BMS pošle odpověď o úspěšném či neúspěšném uložení. Pokud se proměnná uloží správně. Buňka v tabulce se v tabulce s hodnotou uloží. Pokud se uložení neprovede, zobrazí se chybová hláška a buňka zůstane otevřená.

Pokud uživatel chce použít již použitou konfiguraci i na jiné BMS nebo ji jen zálohovat, může využít tlačítek import a export. Tlačítka umožnují exportovat současnou konfiguraci BMS do souboru a zároveň ji zpět importovat. Díky tomu lze velmi rychle nastavit parametry BMS bez nutnosti přepisování většího množství hodnot ručně.

5.7.4 Events

Events (viz obrázek 5.11) je obrazovka pro zobrazení přehledu událostí v BMS. Jednotlivé události můžou značit chybové stavy nebo upozornění o nesprávném chování BMS a díky nim se dá diagnostikovat porucha BMS. Události jsou přehledně zobrazeny do sloupce od největší důležitosti po události, které jsou v normálním stavu.

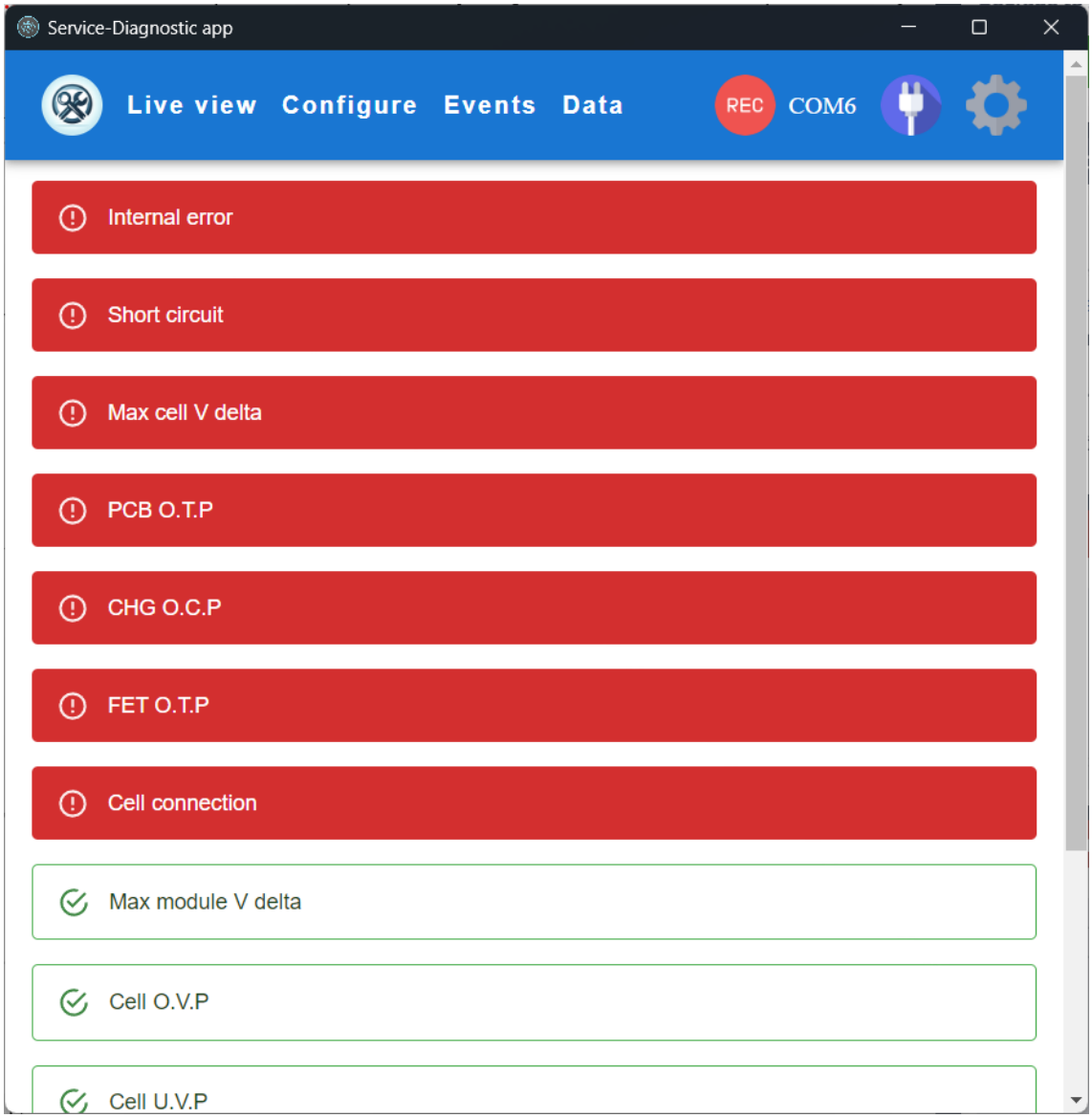
5.7.5 Data

Obrazovka Data (viz obrázek 5.12) je určena pro zobrazení naměřených dat z BMS.Uživatel může procházet všechna data od začátku až po konec měření a to pomocí přepínání stránek. Každá stránka zobrazuje až 100 řádků dat v jednu chvíli. Přepínání mezi stránkami je implementováno z důvodu využití metody Paging(stránkování), kdy v jednu chvíli je v paměti načteno jen omezené množství dat. Pokud by se načítala data všechna, při větším množstvím by to mohlo způsobovat problémy s odezvou GUI a aplikace by využívala velké množství procesorového času. Tímto způsobem má aplikace stejnou odezvu jak s malým množstvím dat, tak i s velkými objemy zaznamenávané po delší časové období.

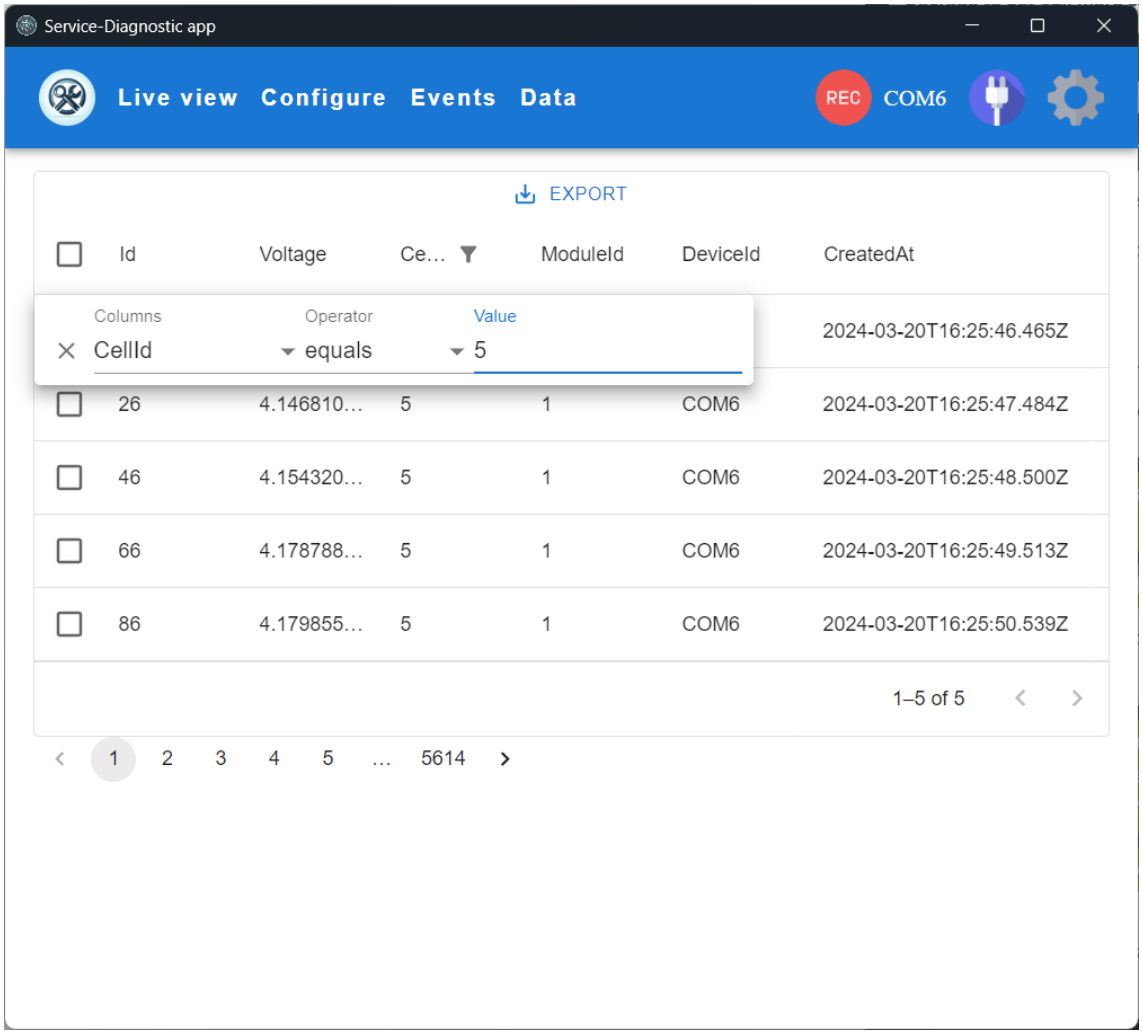
Uživatel může filtrovat jednotlivé sloupce, ale také i řádky. Řádky se dají filtrovat pomocí různých podmínek jako <,>,= atd. Uživatel také pokud potřebuje, může exportovat uložená data pomocí tlačítka export. To vyvolá dialog pro zvolení formátu a místa na disku, kam chce exportovaná data uložit. V tomto stavu aplikace může uživatel exportovat data do csv,pdf nebo k tisku.

5.8 Simulace BMS

Při vývoji servisní aplikace bylo potřeba nějakým způsobem napodobit BMS. To by usnadnilo vývoj a testování funkčnosti aplikace. Tím, že by zařízení fyzicky napodovalo BMS a bylo připojeno do USB zařízení se servisní aplikací. By bylo možné s velkou pravděpodobností říci, že aplikace by byla funkční i s pravou BMS,



Obrázek 5.11: Obrazovka událostí BMS



Obrázek 5.12: Prohlížení zaznamenaných dat

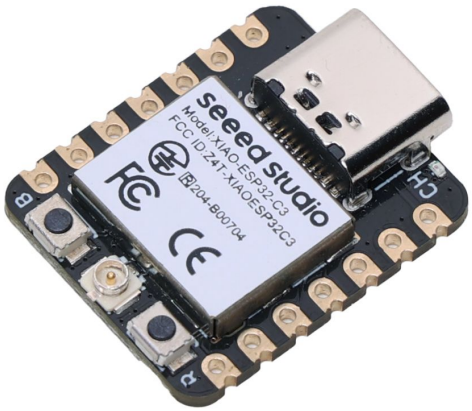
protože by z aplikace nebylo možné určit, že se nejedná o pravé zařízení. Simulační zařízení by tvořilo iluzi BMS tím, že by komunikovalo stejným způsobem přes USB-UART převodník jako skutečná BMS.

Pro tento účel byla vybrána vývojová deska Seeed Studio XIAO ESP32C3. Vývojová deska Seeed Studio XIAO ESP32C3 je kompaktní, výkonný nástroj navržený pro projekty IoT, nositelná zařízení a další aplikace, kde je klíčová velikost a účinnost spotřeby energie.

Na vývojové desce je SoC ESP32-C3, 32bitový jednojádrový procesor RISC-V, který pracuje až na frekvenci 160 MHz. Obsahuje 2,4GHz Wi-Fi subsystém, podporuje různé režimy jako režim stanice a SoftAP režim, spolu s Bluetooth 5.0 a Bluetooth mesh.

Deska je vybavena 400 kB SRAM a 4 MB Flash pamětí. Podporuje různá rozhraní, včetně UART, IIC a SPI. Také je osazena 11 GPIO piny, které mohou sloužit jako PWM výstupy a 4 z nich jako ADC piny.

S rozměry pouhých 21 x 17,5 mm je deska velmi skladná, a také podporuje správu nabíjení lithiových baterií, což dále zvyšuje její přenositelnost a univerzálnost. [27]



Obrázek 5.13: Vývojová deska s MCU ESP32C3[27]

Pro simulátor BMS, nebylo mnoho požadavků na vývojovou desku kromě podpory UARTu. Většina vývojových desek na trhu tuto funkci podporuje a pro účely simulátoru je jejich procesor dostatečně výkonný. Tato deska byla spíše zvolena kvůli její dostupnosti, malé pořizovací ceně 5$, standardnímu USB-C konektoru a její malé velikosti. To se hodí k jejímu přenášení a skladnosti. Během vývoje aplikace s vývojovou deskou nebyl žádný problém a dobře plnila svou funkci.

6 Testování

Jednou z důležitých součástí vývoje softwaru je otestování jeho funkčnosti, spolehlivosti a uživatelské přívětivosti. Testování umožňuje identifikovat a opravit chyby, zlepšit kvalitu kódu, ověřit splnění specifikací a zajistit, že software bude správně fungovat v reálných podmínkách.

V rámci testování bude ověřena funkčnost jednotlivých komponent aplikace, otestování aplikace po výkonové stránce a interakce uživatele s grafickým rozhraním.

6.1 Stavový automat komunikace

Jedna z nejdůležitějších komponent aplikace je stavový automat pro komunikaci s BMS (5.4). U této komponenty je potřeba ověřit, že jsou stavy a přechody implementovány správně, aby při komunikaci nedošlo k neočekávanému chování.

```
test('Initial state + StartCommunication => WaitingForHandshake', () => {
  const callback: (data: Uint8Array) => void = function (data) {};
  let timeout = setTimeout(() => {}, 0);
  expect(
    reducer(
      <StartCommunication>{
        tag: 'StartCommunication',
        timeOutTimer: timeout,
        outputCallback: callback,
      },
      new Initial(),
    ),
  ).toBeInstanceOf(WaitingForHandshake);
});
```

Obrázek 6.1: Testování stavů

Výhoda stavového automatu a implementace jeho stavů i akcí v této aplikaci je ta, že se dá velmi efektivně a jednoduše testovat pomocí jednotkových testů. V jednotkových testech lze jednoduše vytvořit instanci některého ze stavů a na něj aplikovat příslušnou akci (přechodovou funkci). Následně se porovná výsledný stav po aplikování akce s očekávaným výsledkem. Takto se dá jednoduše otestovat každý stav

s každou možnou akcí. Pro toto otestování byla použita knihovna JEST(JavaScript Testing Framework)[19], která přidává velké množství funkcí pro testování aplikací a zároveň usnadňuje implementaci testů. Na obrázku 6.1 je vidět příklad testu pro Initial stav, na který se aplikuje akce StartCommunication a očekávaný nový stav je WaitingForHandshake(viz 5.4). V testu není potřeba žádný složitý kód nebo jiná příprava dat pro testování. Je potřeba jen vytvořit požadovanou akci a stav na který bude akce aplikována. Poté pomocí JEST funkce expect se otestuje výsledek s očekávanou hodnotou. Tímto způsobem se definují i další testy pro ostatní stavy a akce.

6.2 Sériový port

Pro testování funkčnosti samotného připojení aplikace s BMS přes sériový port bylo použita tzv. technika Mocking. Mocking je technika používaná při testování aplikací, která umožňuje izolaci testovaného kódu od jeho závislostí. To znamená, že místo skutečných objektů, s kterými by kód normálně komunikoval, se používají objekty nahrazující, které simulují chování těchto skutečných objektů. To umožňuje testování kódu nezávisle na externích službách, databázích nebo jiných komponentech, které mohou být nestabilní nebo nejsou dostupné během vývoje.

V případě testování této aplikace bylo vhodné použít Mocking na nahrazení fyzického připojení BMS simulátoru k aplikaci. To je vhodné, pokud je potřeba aplikace testovat na jiném, či vzdáleném testovacím stroji, který nemůže mít vlastní fyzicky připojený simulátor BMS nebo je potřeba odizolovat případné chyby způsobené fyzickým připojením nebo hardwarem simulátoru.

Pro tento účel byla použita funkcionalita knihovna Node SerialPort, která je použita v této aplikaci pro používání sériového portu. Ta umožňuje vytvoření virtuálního sériového portu, který má všechny vlastnosti a chování skutečného fyzického připojení.[23]. Díky tomu, jakým způsobem byla v této aplikaci implementována třída pro obsluhu sériového portu. Je možné jednoduše v testovacím prostředí předat třídě virtuální instanci portu, která se následně bez problému používá jako ta skutečná a lze poté otestovat aplikace bez fyzického simulátoru.

6.3 Výkon

Jednou z často zmiňovaných nevýhod Electron frameworku jsou problémy s výkonem, tedy jeho vyšším využíváním procesoru. Je pravda, že Electron využívá více prostředků než nativní aplikace, kvůli jeho využití chromia, ale to nutně nemusí znamenat, že aplikace napsané v Electronu poběží špatně. Příkladem může být např. velmi rozšířený Visual Studio Code, který je vytvořen v Electronu. Většina výkonových problémů spíše pochází z nevhodně napsaného kódu nebo synchronního zpracování asynchronních událostí. [25]

V rámci této aplikace nejsou znát žádné problémy s výkonem, které by ovlivňovaly

uživatelský komfort nebo využívaly příliš mnoho prostředků. I při přenášení vyššího množství dat z BMS je limitující přenosová rychlost sériové linky. Nejvíce systémových prostředků v této aplikaci využívá renderer proces vykreslující uživatelské rozhraní. Při vývoji aplikace byla nalezena slabá stránka vykreslování grafického rozhraní. Ta spočívala v renderování animací při vykreslovaní dat do grafů. Již v řádu stovek vykreslovaných bodů v grafu, který se periodicky překresluje, byl znát vyšší nárůst využití procesoru při běhu aplikace. Jelikož ale animace nebyly pro aplikaci ani uživatelský zážitek potřeba, aplikace po jejich vypnutí spotřebovávala minimálně prostředků a chovala se předvídatelně i s větším počtem vykreslovaných dat.

6.4 Uživatelské rozhraní

Uživatelské rozhraní v rámci této aplikace se testuje čistě manuálně a mohlo by být jedním z budoucích vylepšení, které by zrychlovalo vývoj a snižovalo pravděpodobnosti vnesení chyby do aplikace pozdější úpravou kódu.

Pro otestování grafického rozhraní a v podstatě i fungování celé aplikace je využit simulátor BMS zmíněný v předchozí kapitole. Díky simulátoru dostává aplikace data, která může vizualizovat, ukládat a uživatel může vidět fungování aplikace. Díky simulátoru také může vyzkoušet konfiguraci BMS, kdy simulátor odpovídá na všechny požadavky aplikace jako skutečná BMS. Bez simulátoru by aplikaci nebylo možné příliš testovat, protože bez připojené BMS, je v podstatě jen možné zobrazovat již historická zaznamenaná data.

Grafické rozhraní aplikace je responzivní a dobře reaguje na změny rozlišení monitoru, či škálování okna aplikace. Uživatelské rozhraní je čisté a vizualizovaná data jsou přehledně prezentována v adekvátní formě. Pro navigaci v aplikaci slouží navigační panel, díky kterému se lze jednoduše a rychle pohybovat v aplikaci. Dostat se od spuštění aplikace až k monitorování a konfiguraci BMS lze velmi rychle a uživateli stačí velmi malý počet kroků k dosažení jeho cíle v aplikaci. Navigace je intuitivní a jednotlivá okna aplikace sdílí konzistentní vzhled.

7 Závěr

V rámci této diplomové práce jsem se zabýval vývojem servisní aplikace pro BMS. Nejprve jsem se seznámil s problematikou BMS. Prozkoumal jsem praktická užití BMS, jejich vlastnosti a směr, kterým se tyto technologie vyvíjí. Také jsem prozkoumal, jakým způsobem uživatel interaguje s BMS, jaká data se vizualizují a konfigurují a jakou funkcionalitu by měla servisní aplikace obsahovat.

Na základě rešerše komunikačních standardů jsem navrhl a detailně zdokumentoval vlastní komunikační protokol vytvořený na míru dané aplikaci pro potřeby komunikace mezi BMS a servisní aplikací. V rámci komunikačního protokolu jsem navrhl vhodnou strukturu zpráv, zajišťující robustnost proti chybám v přenosu a flexibilitu v zasílání požadavků a odpovědí mezi servisní aplikací a BMS. Také jsem navrhl stavový automat pro řízení chování komunikačního cyklu BMS a servisní aplikace.

Pro servisní aplikaci jsem zvolil vhodné technologie umožňující implementaci požadovaných vlastností a funkcí aplikace. Výsledkem práce je funkční servisní aplikace splňující požadavky zadání. Servisní aplikace umožňuje připojení BMS přes USB a následně komunikuje pomocí navrženého komunikačního protokolu. Aplikace umožňuje uživateli interagovat s BMS a poskytuje funkcionalitu pro vizualizaci, konfiguraci a ukládání měřených dat.

Během testování nebyly nalezeny žádné problémy s aplikací, která fungovala spolehlivě. Byly použity jak automatizované testy, tak manuální testování uživatelem. Grafické rozhraní je přehledné a intuitivní.

Možností dalšího rozvoje aplikace by mohla být podpora více komunikačních rozhraní pro připojení BMS, jako např. sběrnice CAN nebo implementace automatizovaných testů pro testování grafického rozhraní.

Na závěr mohu říct, že během tvorby této diplomové práce jsem získal cenné zkušenosti ve vývoji softwaru, v problematice BMS a zároveň v organizaci a plánování práce, které se určitě hodí pro můj další profesní rozvoj.

Použitá literatura

[1] *Advanced types* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://www.typescriptlang.org/docs/handbook/advanced-types.html.

[2] *An Algorithm for Error Correcting Cyclic Redundance Checks* [online]. 2003. [cit. 2024-04-08]. Dostupné z: https://www.drdobbs.com/an-algorithm-for-error-correcting-cyclic/184401662.

[3] ANDREA, Davide. *Battery Management Systems for Large Lithium-Ion Battery Packs*. Artech, 2010. ISBN 978-1608071043.

[4] *Back to Basics: The Universal Asynchronous Receiver/Transmitter (UART)* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://www.allaboutcircuits.com/technical-articles/back-to-basics-the-universal-asynchronous-receiver-transmitter-uart/.

[5] *Basics of UART Communication* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://www.circuitbasics.com/basics-uart-communication/.

[6] *Basics of UART Communication* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://www.electronicshub.org/basics-uart-communication/.

[7] *Battery Management System (BMS)* [online]. 2022. [cit. 2024-02-20]. Dostupné z: https://www.futavis.de/en/products/battery-management-systems-bms/.

[8] *Battery Management System Integrated with CAN BUS Safety Control Environment for Electric Vehicle* [online]. 2020. [cit. 2024-04-04]. Dostupné z: https://www.researchgate.net/publication/346073347_Battery_Management_System_Integrated_with_CAN_BUS_Safety_Control_Environment_for_Electric_Vehicle.

[9] *Battery management systems: design by modelling* [online]. 2001. [cit. 2024-02-20]. Dostupné z: https://www.researchgate.net/publication/254858275_Battery_management_systems_design_by_modelling.

[10] *better-sqlite3* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://www.npmjs.com/package/better-sqlite3.

[11] *BU-908: Battery Management System (BMS)* [online]. 2024. [cit. 2024-02-20]. Dostupné z: https://batteryuniversity.com/article/bu-908-battery-management-system-bms.

[12] *Desktop Guide (WPF .NET)* [online]. 2024. [cit. 2024-02-20]. Dostupné z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-8.0/.

[13] *Electron* [online]. 2024. [cit. 2024-02-20]. Dostupné z: https://www.electronjs.org/.

[14] *Fooling the Master: Exploiting Weaknesses in the Modbus Protocol* [online]. 2020. [cit. 2024-04-12]. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1877050920312576.

[15] *GTK* [online]. 2024. [cit. 2024-02-20]. Dostupné z: https://www.gtk.org/.

[16] *Introduction to Modbus* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://control.com/textbook/digital-data-acquisition-and-networks/modbus/.

[17] *Introduction to the Controller Area Network (CAN)* [online]. 2008. [cit. 2024-02-20]. Dostupné z: https://www.academia.edu/23928983/Introduction_to_the_Controller_Area_Network_CAN.

[18] *JavaFx* [online]. 2024. [cit. 2024-02-20]. Dostupné z: https://openjfx.io/.

[19] *JavaScript Testing Framework* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://jestjs.io/.

[20] *Lithno-sirné baterie z VUT na dlouhé cestě do praxe* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://www.technickytydenik.cz/rubriky/energetika-teplo/lithno-sirne-baterie-z-vut-na-dlouhe-ceste-do-praxe_53200.html.

[21] *Modbus* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://modbus.org/.

[22] *MODBUS over serial line* [online]. 2006. [cit. 2024-04-08]. Dostupné z: https://modbus.org/docs/Modbus_over_serial_line_V1_02.pdf.

[23] *Node Serial Test* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://serialport.io/docs/guide-testing.

[24] *Parameters of the BMS* [online]. 2022. [cit. 2024-02-20]. Dostupné z: https://nxp.gitbook.io/rddrone-bms772/software-guide-nuttx/untitled-1.

[25] *Performance* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://www.electronjs.org/docs/latest/tutorial/performance.

[26] *Qt* [online]. 2024. [cit. 2024-02-20]. Dostupné z: https://www.qt.io/.

[27] *Seeed Studio XIAO ESP32C3* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://www.seeedstudio.com/Seeed-XIAO-ESP32C3-p-5431.html.

[28] *SQLite* [online]. 2024. [cit. 2024-04-08]. Dostupné z: https://www.sqlite.org/.

[29] *Understanding CAN* [online]. 2024. [cit. 2024-03-22]. Dostupné z: https://www.circuitbread.com/tutorials/understanding-can-a-beginners-guide-to-the-controller-area-network-protocol.

[30] *What is a Battery Management System?* [online]. 2024. [cit. 2024-02-20]. Dostupné z: https://www.synopsys.com/glossary/what-is-a-battery-management-system.html.

[31] *What is a Solid-state Battery?* [online]. 2024. [cit. 2024-05-01]. Dostupné z: https://www.samsungsdi.com/column/technology/detail/56462.html?listType=gallery.