FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Bakalářská práce

Kompaktní systém pro sběr a zpracování dat v prostředí internetu věcí

Studijní program: B0613A140005 Informační technologie
Studijní obor: Aplikovaná informatika

Autor práce: **David Kárník**
Vedoucí práce: Ing. Jan Kraus, Ph.D.
Ústav mechatroniky a technické informatiky

Liberec 2024

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

Zadání bakalářské práce

Kompaktní systém pro sběr a zpracování dat v prostředí internetu věcí

Jméno a příjmení:	**David Kárník**
Osobní číslo:	M21000088
Studijní program:	B0613A140005 Informační technologie
Specializace:	Aplikovaná informatika
Zadávající katedra:	Ústav mechatroniky a technické informatiky
Akademický rok:	2023/2024

Zásady pro vypracování:

1. Seznamte se s vhodnými rozhraními a protokoly pro sběr, záznam a přenos dat v prostředí internetu věcí, konkrétněji pro aplikace v chytrých budovách a energetice.
2. Na vybrané hardwarové platformě implementujte ukázkové řešení pro sběr lokálních dat (modbus, MQTT atd.), jejich lokální zpracování, archivaci a přenos do nadřazeného systému. Implementace by měla být vícevláknová, zabezpečená a responzivní.
3. Otestujte vytvořený systém a důkladně změřte jeho klíčové vlastnosti, zaměřte se zejména na stanovení limitujících parametrů výkonnosti.
4. V závěru stručně shrňte dosažené výsledky a diskutujte možnosti dalšího rozvoje vašeho řešení.

Rozsah grafických prací:	dle potreby dokumentace
Rozsah pracovní zprávy:	30 až 40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	čeština

Seznam odborné literatury:

[1] BHARATHI, P. Divya; ANANTHANARAYANAN, V.; BAGAVATHI SIVAKUMAR, P. Fog computing-based environmental monitoring using nordic thingy: 52 and raspberry Pi. In: *Smart Systems and IoT: Innovations in Computing: Proceeding of SSIC 2019*. Springer Singapore, 2020. p. 269-279.

[2] NYFFENEGGER, Alex. Connecting constrained devices to the cloud using Zephyr. 2020.

[3] GATIAL, Emil; BALOGH, Zoltán; HLUCHÝ, Ladislav. Concept of energy efficient ESP32 chip for industrial wireless sensor network. In: *2020 IEEE 24th International Conference on Intelligent Engineering Systems (INES)*. IEEE, 2020. p. 179-184.

[4] PLAUSKA, Ignas; LIUTKEVIČIUS, Agnius; JANAVIČIŪTĖ, Audronė. Performance Evaluation of C/C++, MicroPython, Rust and TinyGo Programming Languages on ESP32 Microcontroller. *Electronics*, 2022, 12.1: 143.

[5] NIKOLOV, Neven; NAKOV, Ognyan. Research of secure communication of Esp32 IoT embedded system to. NET core cloud structure using MQTTS SSL/TLS. In: *2019 IEEE XXVIII International Scientific Conference Electronics (ET)*. IEEE, 2019. p. 1-4.

[6] PARIS, Iqbal Luqman Bin Mohd; HABAEBI, Mohamed Hadi; ZYOUD, Alhareth Mohammed. Implementation of SSL/TLS Security with MQTT Protocol in IoT Environment. *Wireless Personal Communications*, 2023, 1-20.

Vedoucí práce:	Ing. Jan Kraus, 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. Ing. Josef Chaloupka, Ph.D.
garant studijního programu

V Liberci dne 12. října 2023

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou 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 bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou 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 bakalářská 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. května 2024 David Kárník

Poděkování

Rád bych upřímně poděkoval všem, kteří mě podporovali a poskytovali cenné rady během mého výzkumu a tvorby této bakalářské práce.

Kompaktní systém pro sběr a zpracování dat v prostředí internetu věcí

Abstrakt

Cílem bakalářské práce je vyvinout ukázkové řešení kompaktního systému, který umožní sběr dat z různých senzorů, jejich zpracování a přenos do centrálního bodu, kde budou data zálohována a dále analyzována. Implementace tohoto systému zahrnuje využití protokolů MQTT a Bluetooth Low Energy (BLE) na platformě ESP32 pro komunikaci mezi chytrými zařízeními a centrálním serverem. Práce se zaměřuje na optimalizaci komunikace mezi IoT zařízeními, spotřebu elektrické energie a klade důraz na bezpečnost a spolehlivost přenosu dat. V závěru celý systém otestuji, zhodnotím dosažné výsledky a uvedu možnosti dalšího rozvoje mého řešení.

Klíčová slova: IoT, MQTT, BLE, IDE, IP, QoS

Abstract

The aim of the bachelor's thesis is to develop a demonstration solution of a compact system that enables data collection from various sensors, their processing, and transmission to a central point, where the data will be stored and further analyzed. The implementation of this system includes the use of MQTT and Bluetooth Low Energy (BLE) protocols on the ESP32 platform for communication between smart devices and the central server. The thesis focuses on optimizing communication between IoT devices, energy consumption, and emphasizes the security and reliability of data transmission. In conclusion, the entire system will be tested, the achieved results evaluated, and possibilities for further development of the solution will be discussed.

Keywords: IoT, MQTT, BLE, IDE, IP, QoS

Seznam zkratek

IoT	Internet of Things, Internet věcí
MQTT	IoT komunikační protokol, Message Queuing Telemetry Transport
BLE	IoT komunikační protokol, Bluetooth Low Energy
IDE	Integrované vývojové prostředí usnadňující programování
IP	Internetový protokol, Adresa identifikující každé zařízení připojené k síti
QoS	Quality of Service, Nastavení kvality přenosu dat u protokolu MQTT

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době je internet věcí jedním z nejrychleji rostoucích technologických segmentů. S rozmachem chytrých zařízení, senzorů a také celých domácností se zvyšuje potřeba efektivního sběru, zpracování a výměny dat. Chytré (IoT) zařízení je elektronické zařízení, které je schopno komunikovat s jinými zařízeními nebo s uživatelem a sbírat a zpracovávat data. Do takové skupiny patří různá zařízení od nejmenších senzorů, chytrých domácích spotřebičů až po velká průmyslová zařízení a vozidla. Tato práce se zabývá návrhem a implementací kompaktního systému pro sběr a zpracování dat v prostředí IoT.

Cílem této práce je vytvořit ukázkové řešení kompaktního systému, který umožní lokální sběr dat, jejich zpracování a přenos do centrálního bodu, kde budou data zálohována a dále analyzována. Implementace tohoto systému zahrnuje využití reálných komunikačních protokolů MQTT, Bluetooth Low Energy (BLE) na platformě ESP32 pro komunikaci mezi chytrými zařízeními a centrálním serverem. Pro sběr lokálních dat ze senzoru používám sériové rozhraní UART. Systém by měl být responzivní na vnější podněty a implementaci musí být vícevláknová. Přidáno by také mělo být zabezpečení komunikace s okolními zařízeními.

Teoretická část práce se zabývá přehledem a představením hlavních komponent a komunikačních protokolů, které jsou v práci využity. Následuje popis návrhu a implementace systému, včetně použitých technologií, postupů a algoritmů. Poté jsou prezentovány výsledky měření a experimentů, které byly provedeny za účelem ověření funkčnosti a efektivity navrženého systému. Závěrečná část práce se věnuje zhodnocení dosažených výsledků, diskusi nad jejich významem a možnostmi dalšího rozvoje a vylepšení systému v budoucnosti.

1 Komunikační protokoly IoT

Internet věcí (IoT) se skládá z velkého množství chytrých zařízení, které mezi sebou musí komunikovat. Komunikace mezi těmito zařízeními vyžaduje použití komunikačních protokolů, což jsou soubory pravidel, která umožňují zařízením vzájemně komunikovat a předávat si mezi sebou data.

Existuje mnoho různých typů komunikačních protokolů, které se používají v různých oblastech a aplikacích. Příklady komunikačních protokolů:

- Bezdrátové protokoly – Umožňují zařízením vysílat a přijímat data bezdrátově, bez nutnosti přímé fyzické konektivity. Bezdrátové protokoly zahrnují například Bluetooth, Zigbee, Z-Wave, LoRaWAN a NB-IoT
- Kabelové protokoly – Používají se k přenosu dat mezi zařízeními pomocí kabelů, jsou to například Ethernet, RS-232, RS-485, UART a CAN
- Internetové protokoly – Tyto protokoly se používají k přenosu dat přes internet a zahrnují protokoly jako HTTP, MQTT a CoAP
- Průmyslové protokoly – Tyto protokoly jsou navrženy pro použití v průmyslových aplikacích a zahrnují protokoly jako Profibus, Modbus a DeviceNet

Každý typ komunikačního protokolu má své vlastní výhody a nevýhody a je důležité vybrat ten správný protokol pro danou aplikaci, aby se zajistila spolehlivá a efektivní komunikace mezi zařízeními. Já se rozhodoval mezi protokoly HTTP, MQTT, BLE a COaP. HTTP jsem si nezvolil jelikož jeho pakety jsou zákonitě větší než u ostatních protokolů a vyžaduje výkonnější zařízení. Není proto úplně vhodný pro můj hardware a požadavky na energeticky efektivní systém. Protokol COaP jsem si také nevybral a zvolil jsem si raději protokol MQTT, jelikož s ním mám zkušenosti již z minulého projektu. Dále jsem také zvolil pro komunikaci s okolními zařízeními protokol BLE, který je optimalizovaný pro použití v malých IoT zařízeních s omezenými prostředky. Pro sběr dat jsem si zvolil sériový protokol UART kvůli mému GPS modulu, ze kterého budu data získávat.

1.1 MQTT

MQTT (Message Queuing Telemetry Transport) komunikační protokol je ze skupiny tzv. „lightweight“ IoT protokolů, pro které je charakteristické jednoduchost,

malá spotřeba energie a přeprava paketů malých velikostí. Je navržen tak, aby byl efektivní, spolehlivý a jednoduchý na použití i v omezených sítích s nízkou silou signálu a vysokou latencí, jako jsou například senzorové sítě nebo mobilní sítě. Jednou z výhod tohoto protokolu je možnost komunikace mezi klienty a serverem oběma směry [19].

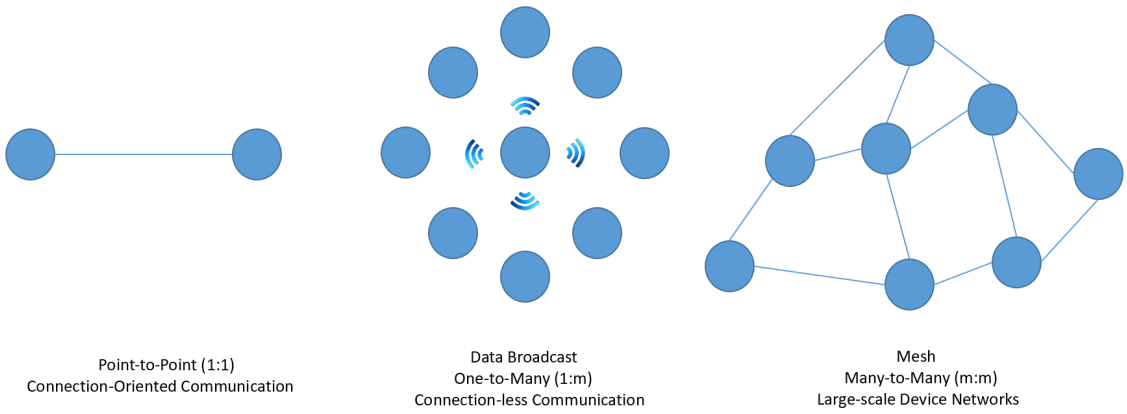
Využívá architekturu typu publish/subscribe pro přenášení paketů. Klienti se mohou připojit k brokeru (serveru) a odebírat (subscribe) nebo publikovat (publish) zprávy na základě témat (topic). Témata (Topics) si lze představit jako vlákna dat, která si server ukládá a kam je možné data vkládat (publish), či odkud je možné data odebírat (subscribe) [21]. Každá zpráva MQTT protokolu se skládá z hlavičky, která vždy zabírá 2 byty, nepovinného obsahu zprávy, který má maximální možnou velikost 256 MB, a tzv. kontrolní pole o velikosti 1 byte. První 4 bity tohoto pole říkají, o jaký typ MQTT zprávy se jedná (CONNECT, CONNACK, PUBLISH, SUBSCRIBE, nebo DISCONNECT). CONNECT - žádost klienta o připojení se k MQTT brokeru, CONNACK - potvrzení připojení, PUBLISH - odeslaná zpráva od klienta, SUBSCRIBE - žádost klienta o odebírání topicu, DISCONNECT - zpráva klienta o odpojování se od brokeru. Zbylé 4 bity nesou informace s fixními příznaky, které klient nemůže nijak měnit. Výjimkou jsou pouze zprávy typu PUBLISH, kde klient může nastavit parametry DUP (zda je zpráva duplikát), RETAIN (říká, zda má být zpráva zachována) a informaci o úrovni QoS [28]. QoS, neboli Quality of Service (Kvalita služby), je charakteristika tohoto protokolu, která určuje úroveň spolehlivosti doručení zpráv. Tento mechanismus byl do MQTT zaveden kvůli nestabilním síťovým prostředím, kde se zařízení mohou potýkat s obtížemi při zajišťování spolehlivé komunikace pouze pomocí přenosového protokolu TCP [45]. Aby se tento problém vyřešil, MQTT zahrnuje mechanismus QoS, který nabízí různé možnosti interakce se zprávami a poskytuje různé úrovně služeb. Tyto úrovně služeb jsou navrženy tak, aby vyhovovaly specifickým požadavkům uživatele na spolehlivé doručení zpráv v různých scénářích. Existují tři úrovně QoS [23]:

- **QoS 0 (At most once)**: Při této úrovni zprávy nejsou potvrzovány a jsou doručeny alespoň jednou. Toto je nejrychlejší úroveň, ale může vést k duplikaci zpráv nebo jejich ztrátě.
- **QoS 1 (At least once)**: Při této úrovni jsou zprávy doručeny alespoň jednou, ale mohou být doručeny vícekrát, takže stále hrozí duplicita dat. Po odeslání zprávy čeká odesílatel na potvrzení (*PUBACK*) přijetí od příjemce. Pokud nedojde k potvrzení, zpráva je znovu odeslána.
- **QoS 2 (Exactly once)**: Při této úrovni jsou zprávy doručeny právě jednou. Odesílatel odesílá zprávu a čeká na potvrzení (*PUBREC*) od příjemce. Poté je zpráva potvrzena (*PUBREL*) a následně je zpráva označena jako doručená. Tato úroveň poskytuje nejvyšší úroveň spolehlivosti, ale je nejpomalejší.

MQTT umožňuje nastavení šifrování a ochrany přenášených dat pomocí TLS (Transportation Layer Protection) a autentifikací klientů jménem a heslem. Protokol je kompatibilní s nejnovějšími autentifikačními protokoly. Používá se například s protokolem OAuth [22].

1.2 BLE

Jednou z významných technologií v oboru IoT komunikace je Bluetooth Low Energy (BLE). Bluetooth Low Energy je technologie vyvinutá ke snížení spotřeby energie při zachování podobného komunikačního rozsahu jako v případě klasického standardu Bluetooth. Výhodou je efektivní využití spotřeby energie v zařízeních napájených z baterie, jako jsou smartphony, tablety, senzory pro monitorování zdraví a další zařízení vyžadující minimální spotřebu energie [34]. BLE je tedy energeticky úsporná varianta technologie Bluetooth [4]. Pokud se zaměříme na samotný komunikační protokol BLE, je jeho struktura a chování řízeno podle principů definovaných v rámci GAP (Generic Access Profile) a GATT (Generic ATTribute Profile). GAP profil říká, jak se zařízení mezi sebou připojují a definuje jejich role v BLE síti [37]. V kontextu protokolu BLE existují dvě základní role: centrála a periferie. Centrála je aktivní zařízení, které řídí komunikaci s periferními zařízeními. Typickým příkladem centrály je chytrý telefon, tablet nebo počítač. Centrála může sbírat data z periferních zařízení nebo je například ovládat. Periferie jsou pasivní zařízení, která poskytují informace centrále. Typickými příklady periferií jsou senzory, měřicí zařízení, ovladače a další periferní zařízení. Periferie obvykle čekají na dotazy nebo příkazy od centrály a odpovídají na ně [29]. GATT určuje jak jsou data formátována, složena a jak si je mezi sebou zařízení data předávají pomocí konceptů nazvaných služby a charakteristiky. Využívá datový protokol nazývaný Attribute Protocol (ATT), který slouží pro ukládání služeb, charakteristik a souvisejících dat do jednoduchých tabulek pomocí 16bitových identifikátorů pro každý záznam v tabulce [1]. Pro BLE sítě existují 3 základní topologie: Point-to-Point (1:1), One-to-Many (1:m) a Many-to-Many (m:m) [12].



Obrázek 1.1: Možnosti komunikace v BLE síti [12]

Já v rámci práce využívám topologii One-to-One, která je z nich nejběžnější. Ta mi poskytuje možnost obousměrné komunikace. Typ Data Broadcast (1:m) se vyu-

žívá při použití senzorů, které volně vysílají svá data do okolí pro sousední zařízení. Je to komunikace jednosměrná. Topologie Mesh umožňuje vytvořit síť o velkém počtu zařízení, kde si mezi sebou navzájem data vyměňují tvz. skoky mezi uzly. V této síti je maximální možný počet uzlů stanovený na 32,767 [12].

1.3 UART

Je důležité říci, že UART nelze řadit mezi běžné komunikační protokoly používané v chytrých budovách, tak jak jsou popsány v zadání. Je ale důležité jej zahrnout do této kapitoly, abych mohl detailněji popsat jeho účel a použití ve vlastním systému. Zatímco BLE a MQTT jsou určeny pro bezdrátovou komunikaci mezi vzdálenými zařízeními v síti, rozhraní UART mi umožňuje sběr dat z připojeného senzoru nebo jiného zařízení přímo k mikrokontroléru pomocí fyzického kabelu.

UART (Universal Asynchronous Receiver/Transmitter) se řadí mezi sériové komunikační protokoly. V sériové komunikaci jsou data přenášena bit po bitu za použití jedné linky (vodiče, kabelu). Pro obousměrnou komunikaci musíme použít dva vodiče pro přenos informací [40]. Je často používán pro jednoduché komunikační úlohy, protože je snadno implementovatelný a nepotřebuje tolik kabelů pro propojení zařízení, což se odráží na cenně. Dle definice je to hardwarový komunikační protokol, který používá asynchronní sériovou komunikaci s nastavitelnou rychlostí přenosu dat [13]. Asynchronní zde znamená, že nepotřebuje žádný společný hodinový signál pro synchronizaci výstupních bitů vysílače (transmitter - Tx) a vstupních dat přijímače (receiver - Rx). Tedy každé zařízení, které chce komunikovat pomocí UART protokolu musí mít oddělený vysílací a přijímací pin pro buď přenosu nebo příjem [42]. Vysílač je připojen k řídící sběrnici dat, která posílá data paralelně. Od tohoto (vysílače) bodu budou data přenášena na přenosové lince (vodiči) sériově, bit po bitu, k přijímači druhého zařízení. Receiver pak sériová data převede zpět na paralelní formu pro přijímací zařízení [13]. Pro UART a většinu sériových komunikací je ale třeba nastavit stejnou baudovou rychlost jak na vysílači, tak na přijímači. Baudová rychlost určuje rychlost, kterou jsou informace přenášeny do komunikačního kanálu. V kontextu sériového portu bude nastavená baudová rychlost sloužit jako maximální počet bitů přenášených za sekundu [32]. Každý přenos dat začíná startovním bitem, který označuje začátek datového rámce, a končí stopovými bity, které označují konec rámce. Toto umožňuje přijímači synchronizovat se s přenosem dat. Složení každého paketu je následující: 1. startovní bit (1 bit), 2. data frame (5 až 9 bitů), 3. paritní bit (0 až 1 bit) a 4. stop bit (1 až 2 bity). Komunikace může být dále nakonfigurován s různými parametry, jako je polarita signálu (například invertovaný nebo neinvertovaný) a formát dat (počet datových bitů, parita a stopové bity) [13].

Já protokol UART používám při komunikaci se senzorem, který mi sbírá data. Konkrétněji při testování používám Mini GPS/BDS modul. Tento GPS modul má UART v základu nastavený na 9600 baudrate, 1 start bit, 1 stop bit a nemá paritní bit [41].

2 Porovnání MQTT a BLE

Při výběru komunikačního protokolu pro IoT (Internet of Things) projekty je rozhodující zvážit různé faktory, včetně rychlosti, spolehlivosti, velikosti paketů a další. MQTT (Message Queuing Telemetry Transport) a BLE (Bluetooth Low Energy) jsou dva z nejrozšířenějších protokolů pro bezdrátovou komunikaci v IoT. Následující kapitola přináší srovnání těchto dvou protokolů v klíčových aspektech.

2.1 Rychlost

Rychlost přenosu dat je kritickým faktorem v mnoha aplikacích IoT, zejména tam, kde je potřeba přenášet data v reálném čase. MQTT je protokol navržený pro spolehlivou komunikaci přes síť TCP/IP, což znamená, že je vhodný pro aplikace, které vyžadují vyšší rychlost za cenu větší spotřeby energie. Je také ideální volbou pro robustnější systémy vyžadující přenos větších objemů dat [8].

Na druhou stranu BLE byl původně vyvinut pro spotřebiče s nízkou spotřebou energie, což ovlivňuje rychlost přenosu dat. Jeho maximální teoretická rychlost je nižší než u MQTT, zejména pokud jde o kontinuální streamování dat. Je ale mnohem praktičtější pro systémy, které vyžadují nízkou spotřebu energie [4].

U reálné aplikace těchto protokolů do systému ale musíme zohlednit spoustu omezujících faktorů. Klíčové je nastavení obou protokolů, které používám. U MQTT mohu nastavit například kvalitu služeb (QoS), která bude ovlivňovat spolehlivost a rychlost zasílání paketů. Důležitým prvkem je také samotný MQTT broker. Pokud nastavím zabezpečení pomocí hesla, šifrování dat, filtrací ip adres, nebo pomocí TLS, bude komunikace samozřejmě pomalejší, ale bezpečnější [24]. U protokolu BLE je také možnost nastavení různých parametrů. Například zabezpečení, šifrování přenášených dat, frekvence zasílání oznámení o aktualizacích [4], nebo maximální velikost datového bloku (MTU), který může být přenesen v jednom BLE paketu [44].

2.2 Spolehlivost

Spolehlivost komunikace je klíčovým aspektem v aplikacích, které vyžadují přenos dat bez ztráty nebo chyb. MQTT, nabízí spolehlivý způsob doručování zpráv, což z něj činí ideální volbu pro kritické aplikace, jako jsou průmyslové systémy nebo monitorovací zařízení. BLE byl navržen s ohledem na energetickou efektivitu, což může ovlivnit jeho spolehlivost v závislosti na konkrétních podmínkách provozu. V

situacích, kdy nízká spotřeba energie není primárním ohledem, může být spolehlivost BLE postačující. Spolehlivost obou protokolů testuji s popisuji v kapitole 7.

2.3 Velikost Paketů

Velikost paketů je důležitým faktorem zejména v prostředích s omezenými šířkami pásma. MQTT zahrnuje záhlaví TCP/IP, což může zvýšit velikost přenášených dat. Přestože existují varianty MQTT, které optimalizují velikost zpráv, mohou být v praxi stále robustnější než BLE [27]. Nicméně teoretická minimální velikost jednoho MQTT paketu jsou 2 byty [28].

BLE, naopak, je navržen tak, aby minimalizoval přenos dat a byl co nejúspornější. Jeho pakety jsou vhodné pro aplikace, které vyžadují efektivní využití šířky pásma, což je činí atraktivním pro projekty s omezenými zdroji [30]. Minimální velikost jeho paketu je však 6 bytů (Hlavička Preamble - 1 byte, Access address - 4 byty a Protocol Data Unit - 2 byty) [31].

Nás však nebude tolik zajímat minimální velikost paketů těchto protokolů, ale to jakou velikost budou mít, pokud je naplním stejnými daty, která budu sbírat a posílat na nadřazený server (viz kapitola 7).

2.4 Spotřeba elektrické energie

V kontextu energetické efektivnosti je důležité brát v úvahu spotřebu energie při použití komunikačních protokolů v konkrétních případech z reálného světa. Při využívání MQTT s Wi-Fi připojením může být energetická náročnost značná, zejména kvůli aktivnímu udržování Wi-Fi připojení, což může způsobit zvýšenou spotřebu energie v režimech, kdy ESP32 nemůže přejít do úsporného režimu. Udržování spojení s Wi-Fi je nezbytné pro komunikaci pomocí protokolu MQTT a vyžaduje neustálý příjem signálu. Může vést k vysoké spotřebě energie, což může být nevýhodné pro zařízení s omezeným napájením.

Na druhou stranu Bluetooth Low Energy (BLE) byl navržen s ohledem na nízkou spotřebu energie, což z něj činí atraktivní volbu pro aplikace s omezeným napájením. BLE by mělo být energeticky efektivnější, zejména pokud zařízení pravidelně přechází do úsporných režimů (například Advertising nebo Sleep Mode) a aktivně udržuje spojení pouze v okamžiku potřeby (přenos dat). Přenos dat pomocí BLE může být optimalizován pro nízkou spotřebu energie, což je vhodné pro bateriově napájená zařízení.

Tato kontrastní povaha mezi MQTT a BLE v otázce energetické efektivnosti může hrát klíčovou roli při rozhodování, který protokol použít v konkrétních aplikacích s omezeným napájením.

3 Vývojové prostředí

Vývojové prostředí (IDE – Integrated Development Enviroment) je software usnadňující práci programátorům. Většinou se zaměřují na jeden konkrétní programovací jazyk a nabízejí různé nástroje a editory kódu. Pro ESP32 je důležité vybrat prostředí, které je schopné zpracovat kód jazyku C/C++ a hlavně musí umět se zařízením komunikovat, aby bylo možné nahrát program do paměti desky ESP. IDE zajišťuje celkovou komunikaci a překlad kódu pro konkrétní procesor, tedy pro ESP32. Kompilátor v rámci IDE překládá zdrojový kód napsaný v programovacím jazyce C/C++ do strojového kódu, který je srozumitelný pro tento konkrétní procesor. Výsledkem je binární soubor, který může být nahrán do paměti mikrokontroléru a spuštěn. Mezi programovacími prostředími jsem našel dva, pro mě vhodné, kandidáty.

3.1 Visual Studio Code

VS Code je editor zdrojového kódu vyvíjený společností Microsoft pro operační systémy Windows, Linux a macOs. Dokáže pracovat s širokou škálou programovacích jazyků, včetně jazyku C. Avšak pro práci s IoT a vývojovou deskou ESP32 je potřeba si nainstalovat rozšíření. Zvolil jsem si proto profesionální vývojové prostředí pro Embeded a IoT zařízení PlatformIO. PlatformIO umožňuje pracovat s frameworkem Arduina a s platformou Espressif 32, které pro projekt potřebuji. Díky tomuto rozšíření jsem schopen používat nástroje pro práci s vývojovými deskami, jako například zobrazení sériového monitoru. I přes všechny složitosti jsem si vybral prostředí VS Code, protože mi přijde více multifunkční.

3.2 Arduino IDE

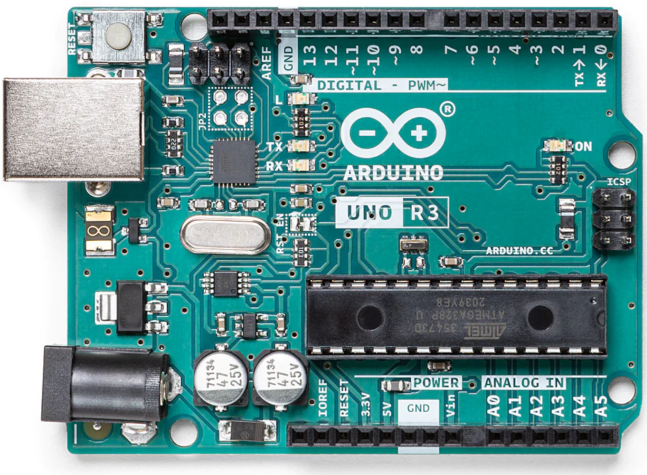
Je prostředí přímo navržené pro programování embeded systému, konkrétně pro Arduino hardware. Je proto nutné si nainstalovat knihovny pro práci s deskou ESP32. Prostředí používá jazyk C++ a je na rozdíl od VS Code méně komplexní, avšak pro spustu různorodých projektů ideální. Další veliké plus pro toto vývojového prostředí je rozsáhlá, názorná dokumentace a nepřeberné množství projektů od ostatních uživatelů. V Arduino IDE jsem již pracoval, ale nakonec jsem si ho nevybral, jelikož VS Code používám téměř denně.

4 Hardware

Nedílnou součástí práce je výběr vhodného hardwaru, který bude schopen splnit požadavky na funkcionalitu, výkon a spolehlivost. Při výběru hardwaru je nezbytné zvážit různé faktory, jako je dostupnost, cena, podpora komunikačních protokolů a dostatečný výkon pro provádění požadovaných úloh. Jedním z důležitých aspektů při výběru hardwaru pro tuto práci je dostatečný výkon a paměť mikrokontroléru. Jelikož je klíčové mít k dispozici více běžících procesů najednou, aby bylo možné efektivně zpracovávat různé úlohy, data a komunikace současně. Proto jsem zvažoval mezi mikrokontrolérem ESP32, vývojovou deskou Arduino Uno Rev3, která disponuje mikročipem ATmega328 a Raspberry Pi Pico W s mikrokontrolérem RP2040 [15] [2] [39]. Modul se také musí umět připojit k Wi-Fi pro komunikaci prostřednictvím komunikačního protokolu MQTT, který v práci chci použít. Musí mít možnost použít některý ze sériových protokolů pro sběr lokálních dat. Při výběru se zaměřuji hlavně na tyto parametry.

4.1 Arduino Uno Rev3

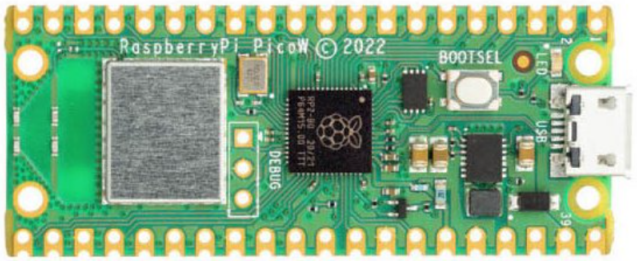
Arduino Uno Rev3 je jedním z nejoblíbenějších vývojových desek na trhu. Tato deska je založena na mikrokontroléru ATmega328P od firmy Microchip (dříve Atmel) [2]. Mikrokontrolér ATmega328P má jedno jádro s frekvencí 20 MHz a nabízí 32 kB flash paměti pro uživatelský kód, 2 kB SRAM a 1 kB EEPROM pro ukládání dat. Arduino Uno Rev3 disponuje 14 digitálními I/O piny, z nichž 6 může být použito jako PWM výstupy, a 6 analogovými vstupy [25]. Dále obsahuje USB konektor pro připojení k počítači a napájení, a sériový port pro komunikaci s periferiemi. Tato deska je ideální pro začátečníky i pokročilé uživatele [33].



Obrázek 4.1: Deska Arduino Uno Rev3 [25]

4.2 Raspberry Pi Pico W

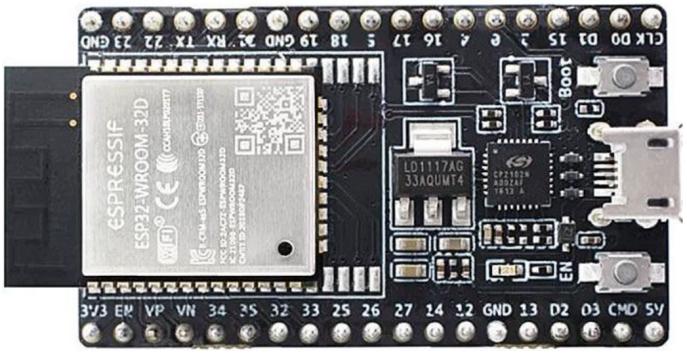
Vývojová deska Pico W je založená na mikrokontroléru RP2040 vyvinutém společností Raspberry Pi Foundation. Tento mikrokontrolér obsahuje dvě jádra ARM Cortex-M0+ s frekvencí až 133 MHz a nabízí 256 kB interní paměti RAM. Raspberry Pi Pico W poskytuje širokou škálu funkcí, včetně 28 programovatelných GPIO pinů, 3 ADC (Analog-to-Digital Converter) piny, 2 UART (Universal Asynchronous Receiver-Transmitter) rozhraní, a další [26]. Dále obsahuje USB konektor pro napájení a komunikaci s počítačem, stejně jako pro programování pomocí rozhraní USB Mass Storage Device (MSD). Tato deska je vhodná pro různé projekty od jednoduchých elektronických zařízení až po složitější IoT aplikace [39].



Obrázek 4.2: Raspberry Pi Pico W [26]

4.3 Vývojová deska ESP32

Vývojová deska ESP32, konkrétně ESP32-WROOM-32D, od společnosti Espressif Systems. Modul využívá operační systém freeRTOS pro IoT zařízení. Kapacita paměti RAM je 520KB a velikost programovatelné paměti FLASH jsou 4MB. Deska obsahuje řadu vestavěných modulů a funkcionalit, včetně [15] [14]:



Obrázek 4.3: Vývojová deska s modulem ESP-WROOM-32D [20]

- Wi-Fi a Bluetooth – Podporující standardy 802.11 b/g/n pro bezdrátové připojení přes internet a vytváření vlastních přístupových bodů (Access Point) pro připojení dalších zařízení. Modul Bluetooth umožňuje komunikovat s jinými zařízeními pomocí Bluetooth Low Energy (BLE)
- Dual-Core procesor – ESP32 WROOM 32D je vybaven duálním jádrem (dvě procesorová jádra) s nastavitelnou frekvencí 80 MHz až 240 MHz
- GPIO piny – Deska má celkem 38 GPIO (General Purpose Input/Output) pinů
- ADC a DAC – Analogově-digitální převodníky (ADC) a digitálně-analogové převodníky (DAC) umožňují ESP32 převádět analogová data na digitální a naopak
- I2C, SPI, UART – Podpora sériových komunikačních protokolů jako I2C, SPI a UART

4.4 Porovnání mikrokontrolérů

I přesto, že mám zkušenosti s vývojovou deskou Arduino Uno z minulých projektů a je dostupné velké množství návodů pro Arduina na internetu, rozhodl jsem se pro jinou volbu. Raspberry Pi Pico W by pravděpodobně byla nejjednodušší volbou díky své snadné obsluze a detailním návodům, avšak jsem se rozhodl pro složitější vývojovou desku ESP32. Tato deska se mi zdála ze všech adeptů nejlepší, protože je nejvýkonnější, splňuje všechna kritéria, avšak je zároveň nejvíce komplexní.

	ESP32-WROOM-32	Arduino Uno REV3	Raspberry Pi Pico W
Procesor	ESP32	ATmega328P	RP2040
Počet jader	2	1	2
Frekvence procesoru [MHz]	80 - 240	20	133
WiFi modul	Ano	Ne	Ano
Bluetooth modul	Ano	Ne	Ne
Počet GPIO pinů	38	14	28
RAM [kB]	520	2	256
Flash [kB]	4096	32	2048
Cena [Kč]	200	650	220
Výrobce	Espressif Systems	Arduino	Raspberry Pi Ltd

Tabulka 4.1: Porovnání ESP32, Arduino Uno Rev3 a Raspberry Pi Pico W

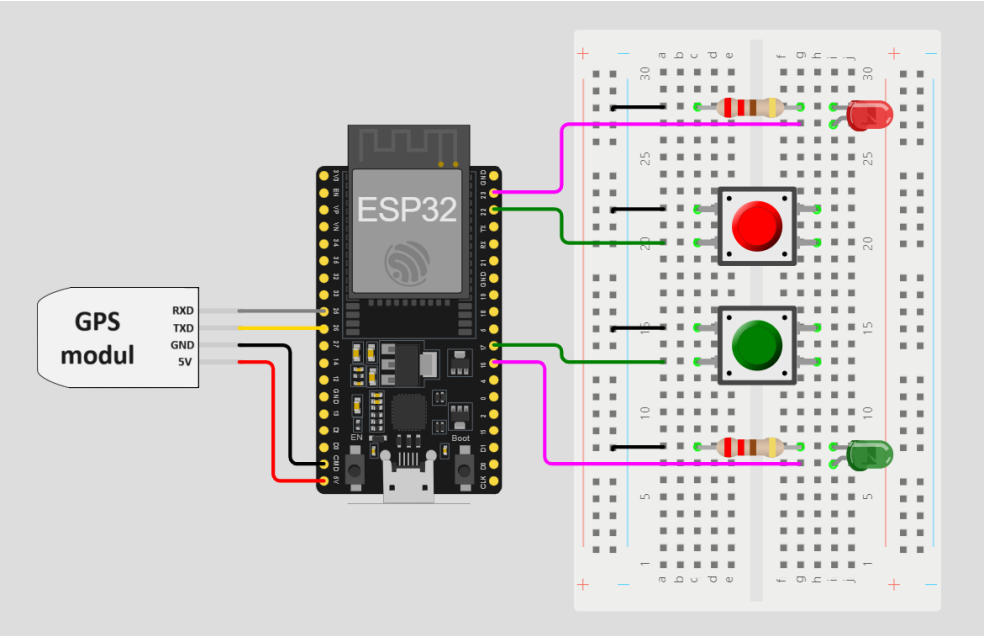
Rozhodl jsem se tedy pro použití mikrokontroléru řady ESP32. Tento mikrokontrolér nabízí dostatečný výkon, podporu různých komunikačních protokolů, jako je Wi-Fi, sériová komunikace a Bluetooth Low Energy (BLE), a především disponuje dvěma jádry, což umožňuje efektivní paralelní zpracování úloh. Má také dostatek GPIO pinů pro připojení periferií, jako například LED diod, nebo senzorů [14].

4.5 Zapojení obvodu

Obvod se skládá z desky ESP32, která řídí většinu logických operací, GPS modulu pro sběr lokálních dat, 2 signalizačních LED diod, 2 tlačítek pro mechanické ovládání přerušení (interrupt) běžících procesů a 2 předřadných rezistorů pro omezení proudu. Vše je vodivě pospojováno pomocí nepájivého pole a vodičů s konektory. Jelikož na ESP32 běží různé procesy na různých jádrech připojil jsem k obvodu signalizační LED diody. Jednu pro každé jádro. Pokud na jádře běží proces, LED dioda svítí. Pomocí tlačítka je možné reprodukovat funkci přerušení procesu na jádře. Opět jedno tlačítko pro jedno jádro. Pokud dojde ke stisku tlačítka, logická jednotka ESP32 zaznamená stisk tohoto tlačítka a provede proces přerušení na konkrétním jádře. Ve stejnou chvíli se rozsvítí signalizační LED dioda tohoto jádro, která značí, že proces na jádře je zastaven. Po uvolnění tlačítka se funkce jádra obnoví ze stavu přerušení a dojde k opětovnému spuštění programu.

Modul GPS je k ESP32 připojen pomocí sériového rozhraní UART. Piny modulu 5V a GND na příslušné odpovídající piny vývojové desky. Pin TXD na

GPIO pin 26 a signál RXD na pin 25. LED diody jsem připojil k digitálním pinům 2 a 4. Tlačítka jsem připojil na digitální piny 13 a 15. Pro připojení tlačítek na tyto GPIO piny jsem také využil funkci interně zabudovaného pull-up rezistoru mikrokontroléru, abych předešel problémům s nestabilitou vstupního signálu. Mezi každou LED diodu a pin desky ESP jsem přidal předřadný odpor pro omezení procházejícího proudu. Předřadný odpor chrání jak diodu, před přílišným proudem, tak GPIO pin mikrokontroléru.



Obrázek 4.4: Schéma zapojení ESP32 s GPS modulem, tlačítky a LED diodami [43]

5 Software

V této kapitole se zaměřím na strukturu a funkčnost softwarových prvků projektu.

Data a programování tvoří jádro softwaru. Program umožňuje zpracovávat a interpretovat informace získané z hardwarových senzorů, mikrokontroléru a dalších zařízení. Implementace softwaru je realizována v jazyce C/C++ pomocí vývojového prostředí Visual Studio Code a nástroje PlatformIO, které poskytují efektivní a uživatelsky přívětivé prostředí pro vývoj, ladění a nasazení softwarových IoT aplikací [36].

Pro programování a ovládání mikrokontrolérů v projektu využívám knihovnu Arduino přímo z platformy PlatformIO, která poskytuje bohatou škálu funkcí a jednoduché prostředky pro tvorbu programů. Díky flexibilitě a rozšířenosti této knihovny mohu snadno integrovat různé senzory a aktuátory do tohoto systému.

Kromě jazyka C++ využívám také skriptovací jazyk Python pro zpracování, ukládání a analýzu dat získaných z chytrých zařízení a jejich následnou vizualizaci a interpretaci.

5.1 Použitý program

Pro vývoj a programování mikrokontroléru ESP32 a komunikaci s vývojovou deskou používám prostředí Visual Studio Code s rozšířením PlatformIO. PlatformIO je open-source ekosystém navržený pro vývoj Internetu věcí (IoT) a vestavěných systémů. Toto rozšíření poskytuje vývojářům nástroje pro programování a nahrávání kódů pro různé mikrokontroléry, včetně ESP32. Má rozsáhlou knihovnu, podporu různých platforem a programovacích jazyků [35].

V této práci preferuji programovací jazyky C++ a Python. Jazyk C++ slouží k definici logiky a programu pro mikrokontrolér ESP32. Klíčovým stavebním kamenem mého kódu je knihovna Arduino.h, která mi umožňuje využít syntaxi a funkce z prostředí Arduino. Mohu se proto spolehnout na širokou škálu volně dostupných knihoven a nástrojů, které usnadňují vývoj a programování vestavěných systémů z tohoto balíčku. Hlavní již implementované metody, které využívám, jsou setup() a xTaskCreatePinnedToCore().

Metoda setup() má za úkol obsahovat inicializační kód, který se spustí jednou při startu mikrokontroléru. Zde probíhá nastavení různých parametrů, inicializace periferií a příprava mikrokontroléru k běhu hlavní smyčky programu.

Pro vytvoření a spuštění úlohy na konkrétním jádře mikrokontroléru ESP32 po-

užívám metodu xTaskCreatePinnedToCore(). Tato metoda umožňuje vytvořit úlohu (task) a přiřadit ji konkrétnímu jádru mikrokontroléru. Tato úloha (funkce) se provede pouze jednou a tudíž pro její opakování, například využívání komunikačního protokolu, jsem naprogramoval jako nekonečnou smyčku. Zde je umístěn hlavní kód programu, který se opakuje během celého běhu programu (v tomto případě procesu). V této metodě řeším hlavní logiku mé aplikace, komunikaci s periferiemi, sběr a odesílání dat a další opakující se úkony. Funkci xTaskCreatePinnedToCore() volám na úplném konci až po úspěšném nastavení a inicializaci všech komponent, proměnných a potřebných knihoven. Funkci volám dohromady třikrát. Jednou pro uvedení sběru dat z UARTU do provozu. Jednou pro navázání komunikace přes protokol MQTT a zasílání sbíraných dat brokeru. A jednou pro spuštění BLE serveru a zasílání dat klientům přes BLE komunikaci. Úloha pro sběr dat z UARTU a komunikace MQTT běží na prvním jádře a každý na jiném vlákně. Logika BLE serveru a komunikace běží na druhém jádře ESP32.

Pro ukládání dat z MQTT do souborové databáze na PC využívám programovací jazyk Python. Python poskytuje robustní knihovny a nástroje pro práci s MQTT protokolem a manipulaci se soubory. Konkrétně v této implementaci využívám knihovny Paho MQTT pro komunikaci s MQTT brokerem a vestavěný modul IO pro zpracování souborů [11]. V algoritmu nejprve inicializuji připojení k MQTT brokeru a definuji callback funkce, která se volá při příchodu nové zprávy. Když skript obdrží zprávu od ESP32, zpracuje ji a uloží do souborové databáze. Soubory jsou v mém případě ukládány v jednoduchém textovém formátu.

5.2 Nastavení MQTT brokeru

Od aktualizace na verzi Mosquitto 2.0.0 bylo zavedeno nastavení výchozího posluchače (listener) na localhost:

> „When the Mosquitto broker is run without configuring any listeners it will now bind to the loopback interfaces 127.0.0.1 and/or ::1. This means that only connections from the local host will be possible.“ [10]

To znamená, že broker MQTT je implicitně nakonfigurován tak, že je přístupný pouze z lokálního stroje (PC), na kterém je hostován. Tento krok byl učiněn jako bezpečnostní opatření s cílem minimalizovat riziko neoprávněného přístupu k brokeru z externích zdrojů. Takové nastavení posiluje bezpečnostní opatření a snižuje možnost neoprávněného přístupu k brokeru. V mém případě je ale nepoužitelné, jelikož k brokeru potřebuji přistupovat i z ostatních zařízení (ESP32 a mobilních zařízení). Pro potřeby testování samotného připojení bez zabezpečení jsem musel nastavil parametry konfiguračního souboru. Nastavení ”listener 1883 0.0.0.0”povolilo připojení všech IP adres (zařízení) k portu 1883, na kterém broker běží. Samozřejmě pouze z lokální sítě. Povolení ”allow_anonymous true”zrušilo potřebu jména a hesla pro připojená zařízení.

Po testování a vyzkoušení funkcionalit brokera jsem musel nastavit reálné parametry zabezpečení. Celé zabezpečení na straně brokeru spočívalo v nutnosti zadat

jména a hesla zařízení, které se chtěla připojit. Aplikoval jsem také filtraci jejich IP adres a povolil jen ty, co jsem chtěl. Více celý postup popisuji v kapitole 6.3.

5.3 Multitasking a synchronizace

Pro efektivní využití výpočetních zdrojů ESP32 byly jednotlivé úkoly implementovány jako samostatné programy běžící na různých jádrech procesoru ESP32. Funkce xTaskCreatePinnedToCore() z knihovny FreeRTOS umožňuje alokaci úloh na konkrétní jádro, což zajišťuje paralelní běh různých částí programu. [16] Po spuštění více jader však nastává mnoho komplikací a možnost přechodu do nebezpečných stavů. Jednotlivé úlohy programu potřebují často přistupovat ke sdíleným proměnný a souborové databázi. Synchronizace mezi úlohami proto probíhá pomocí vhodných mechanismů, jako jsou mutexy a semafory, aby bylo dosaženo správného a bezpečného sdílení dat a zabránilo se problémům se souběhem. Použil jsem Mutex (Mutual Exclusion) pro zablokování přístupu k sdíleným prostředkům, takže pouze jedna úloha může provádět kritickou sekci kódu a další musí počkat na povolení. To eliminuje problémy souběhu, kde dvě nebo více úloh mohou zároveň provádět operace, které by mohly způsobit nekonzistentní stav.

5.4 Zpracování a ukládání dat

V rámci práce z pracovávám a ukládám data lokálně. Konkrétně ukládám data na lokálním zařízení (notebooku) do souborové databáze. Data jsou ale také ukládána i na straně ESP32. Do programu na straně mikrokontroléru jsem implementoval ukládání do souborů s využitím SPIFFS (SPI Flash File System). Tato technologie umožňuje rychlý a modifikovatelný přístup k datům uloženým v paměti Flash [17].

Na notebooku (nadřazeném systému) běží 2 Python skripty. Jeden je pro zachytávání paketů z MQTT brokeru, tedy MQTT klient připojený k brokeru a odebírající konkrétní topic. Při každé příchozí zprávě z ESP32 na broker zašle (broker) dále data tomuto Python klientu a ten data uloží do souborové databáze (fileMQTT.txt). Druhý skript je naprogramovaný BLE klient, který se připojí přímo k desce ESP32, na které je BLE server. Tento skript také ukládá příchozí data, ale z BLE komunikace a ukládá je do souboru (fileBLE.txt). Tyto soubory poté mohu porovnat a analyzovat chyby a nedostatky v každé komunikaci z těchto rozdílných komunikačních protokolů [7]. Tato souborová databáze tedy umožňuje ukládání a správu dat získaných z připojených chytrých zařízeních. Je klíčovým prvkem pro uchování a analyzování historických dat a umožňuje efektivní práci s velkým objemem informací. Data si také ESP32 ukládá přímo do své souborové databáze ve své paměti pomocí knihovny SPIFFS [17].

Proces tedy probíhá následovně. Nejdříve dojde ke sběru dat z UART GPS modulu do souboru na ESP32. Po úspěšném nahrání dat do souboru na desce jsou zprávy periodicky odesílány na MQTT broker (proces na prvním jádře). Ten zprávy dále rozešle klientům, kteří příslušný topic, na který zprávy dorazili, odebírají. MQTT Python klient data zachytí a uloží do souborů na hostujícím notebooku.

Skript využívá knihovnu paho-mqtt [11] [7]. Tento skript se nejdříve připojí na Mosquitto broker pomocí jeho IP adresy a portu. Následně poslouchá a odebírá konkrétní MQTT topic. Z pohledu procesu na druhém jádře mikrokontroléru jsem naprogramoval také periodické odesílání dat připojeným periferiím, ale pomocí protokolu BLE. V Pythonu jsem opět vytvořil algoritmus, který běži na notebooku a dokáže se napojit k BLE serveru na ESP32, zachytávat příchozí data a ukládat je do souborové databáze. Tedy obdobný postup jako u MQTT. Tímto způsobem jsou data uchovávána a zpracovávána jak na samotném ESP32, tak i na hostujícím počítači.

Tímto způsobem mohu efektivně sbírat a ukládat data z ESP32 do souborů na PC, což usnadňuje analýzu a vizualizaci těchto dat. Python je vhodný nástroj pro manipulaci se soubory, analýzu dat a tvorbu uživatelsky přívětivých vizualizací.

6 Realizace systému

Ekosystém systému je postaven na platformě ESP32, která poskytuje prostředí pro implementaci komunikačních protokolů MQTT a BLE (Bluetooth Low Energy). Hlavním cílem je sběr a distribuce dat mezi zařízeními pomocí těchto protokolů. V zadání je také uvedeno, že veškerá komunikace musí být zabezpečena. Tudíž využívám různé postupy a metody, jak celý systém zabezpečit. Proto se žádné neautorizované zařízení (uživatel) nemůže připojit jak k MQTT brokeru, tak ani k BLE serveru.

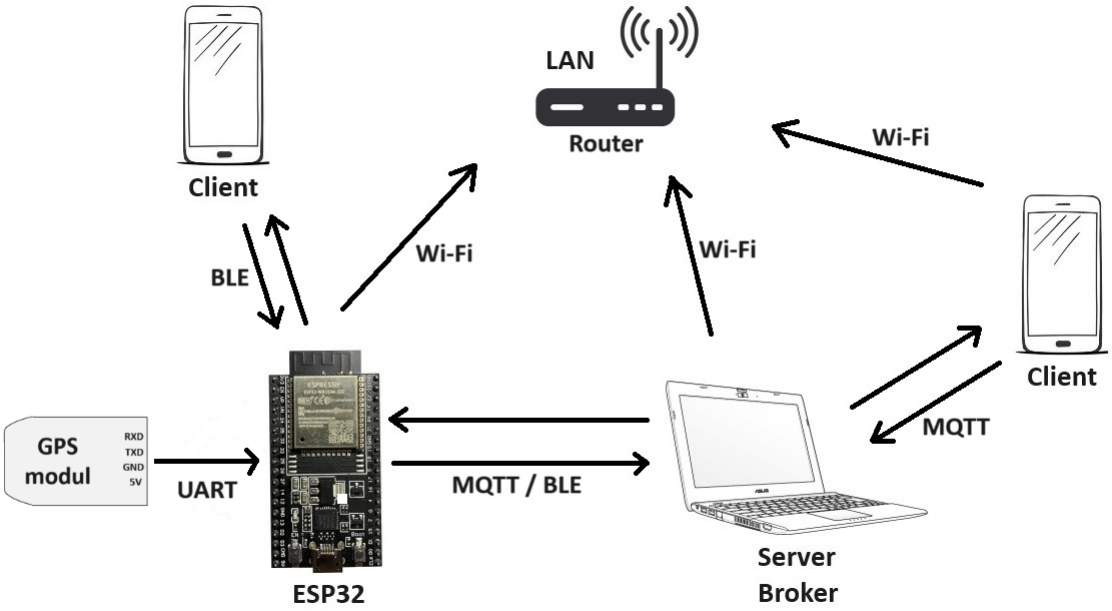
Na ESP32 je implementován MQTT klient, který běží na jednom jádře. Tento klient slouží k odesílání dat na MQTT server (broker), který běží na notebooku. MQTT broker je zodpovědný za distribuci příchozích zpráv připojeným periferiím (subscribers), jako je mobilní aplikace pracující s MQTT, či samotný modul ESP32, který je nastavený jako subscriber i jako publisher. Pro úspěšnou komunikaci musí být všechna zařízení připojena ke stejné Wi-Fi síti, která je v mém případě zajištěna jedním společným routerem.

Na notebooku je spuštěn Python skript, který funguje jako MQTT klient připojený k brokeru. Tento skript zachytává příchozí zprávy z ESP32 a ukládá je do databáze na notebooku pro další zpracování a případnou analýzu.

Druhé jádro ESP32 hostí BLE server, který slouží k odesílání dat pomocí protokolu Bluetooth Low Energy. K tomuto BLE serveru je připojena mobilní aplikace, která pracuje s tímto protokolem a je schopna zobrazit příchozí zprávy i odeslat zprávu na samotný server. K této komunikaci je také připojen notebook s Python skriptem, který slouží jako připojená periferie k BLE serveru a zprávy zachytává a ukládá do databáze pro další zpracování.

Celkově je ekosystém systému navržen tak, aby umožňoval spolehlivý, zabezpečený a efektivní sběr a distribuci dat mezi zařízeními pomocí protokolů MQTT a BLE. K systému a následné komunikaci se může připojit každé autorizované chytré zařízení.

6.1 Návrh systému



Obrázek 6.1: Komunikace mezi prvky v systému

Na obrázku můžeme vidět demonstrační komunikaci mezi jednotlivými prvky ekosystému, který jsem sestavil. ESP32 sbírá data o poloze z GPS modulu přes UART. Dále na ESP32 běží BLE server, který zasílá zpracovaná data přes BLE protokol. K serveru jsou připojeni dva klienti. Python script běžící na notebooku pro archivaci dat a mobilní aplikace klienta BLE, ve které vidím živě přicházející data. Mikrokontrolér zasílá zpracovaná data také přes MQTT protokol na broker, který data přeposílá dalším připojeným zařízením, jako je taktéž mobilní aplikace, nebo samotné ESP32. V diagramu je znázorněný router, ke kterému musejí být připojena všechna zařízená, která chtějí komunikovat přes MQTT protokol.

6.2 Problém s pamětí flash

Na mikrokontroléru ESP32 jsem vyzkoušel a odladil všechny potřebné algoritmy i komunikační protokoly, ale vždy jako samostatné programy. Když jsem však všechny technologie sloučil do jednoho finálního programu došlo k přetečení paměti flash. Mé ESP32 má velikost paměti 4 MB. Výsledný program by potřeboval kapacitu úložiště 125%. Celý kód jsem se proto snažil co nejvíce optimalizovat a upravit pro co nejmenší objem. Odebral jsem nepotřebné technologie a sloučil a upravil funkce. Dokázal jsem kód optimalizovat až na 114% paměti flash. Tento postup však nikam nevedl, jelikož největší složkou programu byly knihovny (hlavně ty pro protokol

BLE) [3]. Nahradil jsem všechny knihovny pro komunikaci pomocí BLE novými a aktuální vytíženost vnitřní paměti klesla na optimálních 72%.

6.3 Zabezpečení

Zabezpečení a šifrování jakékoliv komunikace je dnes již běžnou praxí. V mém systému využívám několik opatření pro zajištění bezpečnosti komunikace přes komunikační protokoly MQTT a BLE.

Pro MQTT komunikaci jsem nastavil několik bezpečnostních prvků v konfiguračním souboru mého Mosquitto brokera. To zahrnuje povinnost autentizace každého klienta pomocí přednastaveného uživatelského jména a hesla, filtrace IP adres zařízení, zakázání anonymních připojení pomocí volby ”allow_anonymous false”. Veškeré zabezpečení komunikace pomocí MQTT leží na straně brokera. Každý klient se pro úspěšné připojení ke komunikaci musí autorizovat.

Ochranné mechanismy jsem nastavil pomocí úpravy konfiguračního souboru ”mosquitto.conf”, který je implicitně umístěný přímo v lokálním adresáři Mosquitto brokera. Zde jsem nastavil parametr ”listener 1883 <adresa_IP_zařízení>”, který umožní připojení zařízení s konkrétní IP adresou k portu 1883, na kterém broker běží. Dále konfigurace ”allow_anonymous false”pro nastavení nutnosti pro připojené klienty autorizovat se jménem a heslem. Jména a hesla jsem nastavil následujícím způsobem:

1. Založení textového souboru passwords.txt ve stejném adresáři, jako je Mosquitto broker
2. Vepsání jména a hesla do souboru ve formátu ”jmeno:heslo”
3. Zašifrování hesla mechanismem pomocí příkazu ”mosquitto_passwd -U ./passwords.txt”, který zašifruje hesla do požadovaného stavu a Mosquitto broker je bude schopný zpracovat [9]
4. Přidání parametru pro nastavení souboru, z kterého bude broker čerpat jména a hesla, ”password_file ./password.txt”do konfiguračního souboru

Šifrovat hesla v ”./password.txt”souboru není nutností. Broker je i tak dokáže přečíst a použít, ale je to doporučované. Příkaz ”mosquitto_passwd -U <file>”hesla v souboru zašifruje podle šifrovací funkce crypt(3) [6].

Pro Bluetooth Low Energy (BLE) jsem implementoval statický PIN kód, který je vyžadován pro připojení k BLE serveru. Tímto způsobem zajišťuji, že pouze oprávněná zařízení jsou schopna připojit se a komunikovat s mým ESP32 zařízením (BLE serverem) pomocí BLE protokolu.

V rámci dalšího zabezpečení sítě bylo vyžadováno zadání hesla pro připojení k jedné konkrétní Wi-Fi síti, což zajišťuje, že pouze autorizovaná zařízení jsou připojena k naší síti (v mém případě ke společnému routeru). Toto zabezpečení je zvláště důležité v kontextu MQTT protokolu, který vyžaduje stálé připojení k Wi-Fi síti. Dodatečný ochranný mechanismus by bylo možné zohlednit pouze v případě,

že bychom hostovali MQTT broker u sebe na lokálním zařízení v naší lokální síti. V opačném případě, pokud bychom hostovali broker přes cloudovou službu, mohli bychom se k němu připojit odkudkoli, přičemž bychom potřebovali pouze stabilní připojení k internetu a znalost adresy tohoto serveru. V mém případě je realizace celého systému čistě lokální.

6.4 Praktické využití

Díky nepřebernému množství senzorů, které lze k mikrokontroléru připojit, a jeho schopnosti napájení z přídavné baterie, nabízí ESP32 široké možnosti využití v různých oblastech. Systém, díky svým malým rozměrům a energetické efektivitě, se hodí pro bezdrátové monitorování zařízení, chytré domácnosti, sledování prostředí a mnoho dalších aplikací. Připojení k Bluetooth Low Energy (BLE), stejně jako k MQTT serveru, umožňuje komunikaci s mobilními zařízeními, což umožňuje vzdálený přístup a sledování pomocí mobilních aplikací. Tato bezdrátová konektivita usnadňuje integraci ESP32 do existujících ekosystémů chytrých zařízení. Díky podpoře pro vícevláknové zpracování (multithreading) může ESP32 provádět více úloh paralelně, což je výhodné pro různé úlohy zpracování dat a senzorické sběry. Tato schopnost umožňuje efektivní využití zdrojů a zajišťuje plynulý běh systému i při komplexních operacích. Pro optimalizaci výkonu je možné upravit periferie, nastavení portů a algoritmů v závislosti na konkrétních požadavcích dané aplikace. Z tohoto důvodu je systém vhodný pro širokou škálu projektů, od jednoduchých senzorických měření po náročné aplikace v internetu věcí (IoT).

Monitorování a Kontrola Senzorů v Domácnosti:

V domácnosti můžeme nainstalovat různé senzory (např. teplotní senzory, senzory pohybu) v různých místnostech. Propojíme je s ESP32, ať už bezdrátově, či pomocí kabelu. Přičemž jedno jádro mokrokontroléru je schopno posílat data přes protokol BLE na mobilní aplikaci, a druhé jádro zasílá data přes MQTT na server na PC.

V reálné situaci mobilní aplikace s BLE může poskytovat uživatelům okamžitý přístup k informacím o teplotě, pohybu, vlhkosti atd. v každé místnosti přímo na mobilní zařízení. MQTT data mohou být ukládána na PC a analyzována pro dlouhodobé sledování, vytváření grafů nebo vytváření historie údajů. Je možné i opačné provedení, kde budou na počítači zpracovávána data z protokolu BLE a data z MQTT budou zasílána z ESP32 klienta na broker server a dále mobilnímu klientu do aplikace na chytré zařízení. Pokud bude potřeba vzdálený přístup k datům a ne jen lokální za použití Bluetooth, je možné data zpracovat na serveru a zasílat přes internet.

Monitorování a Ovládání zařízení v Průmyslovém Prostředí:

V průmyslovém prostředí mohou být různá zařízení (senzory, aktuátory) rozmís-

těna na velkém území. ESP32 jednotky mohou být umístěny poblíž zařízení, které je potřeba monitorovat, s tím, že jedno jádro je schopno zajišťovat komunikaci a zasílat data na místo A a druhé jádro může data zpracovávat a zasílat na místo B kompletně rozdílným způsobem.

Konkrétní praktické využití protokolu BLE na jednom jádře umožní přístup technikům nebo obsluze k aktuálním údajům a stavu zařízení pomocí chytrých zařízení. MQTT umožní centralizovaný monitorovací systém, který shromažďuje data ze všech zařízení pro analýzu, plánování údržby a rychlou reakci na události.

Využití na Venkově:

Na venkově můžeme mít různá zařízení pro monitoring a řízení zavlažování, sledování stavu hospodářských zvířat apod. Na mikrokontroléru ESP32 můžeme využít obě jádra, kde jedno jádro zajišťuje komunikaci přes BLE a druhé jádro přes MQTT. Komunikaci protokolem BLE můžeme přijímat mobilní aplikací, která umožní sledovat a ovládat zařízení přímo na místě, kvůli omezení dosahu signálu BLE. Řádově je dosah BLE do několika desítek metrů v závislosti na výkonu konkrétního zařízení, které signál přijímá i vysílá, na překážkách v prostředí, anebo na elektromagnetickém šumu okolí. MQTT data mohou být odesílána na server, který umožní sledovat historii, provádět analýzy dat, vzdáleně řídit zařízení a dále data zpracovávat dle požadavků.

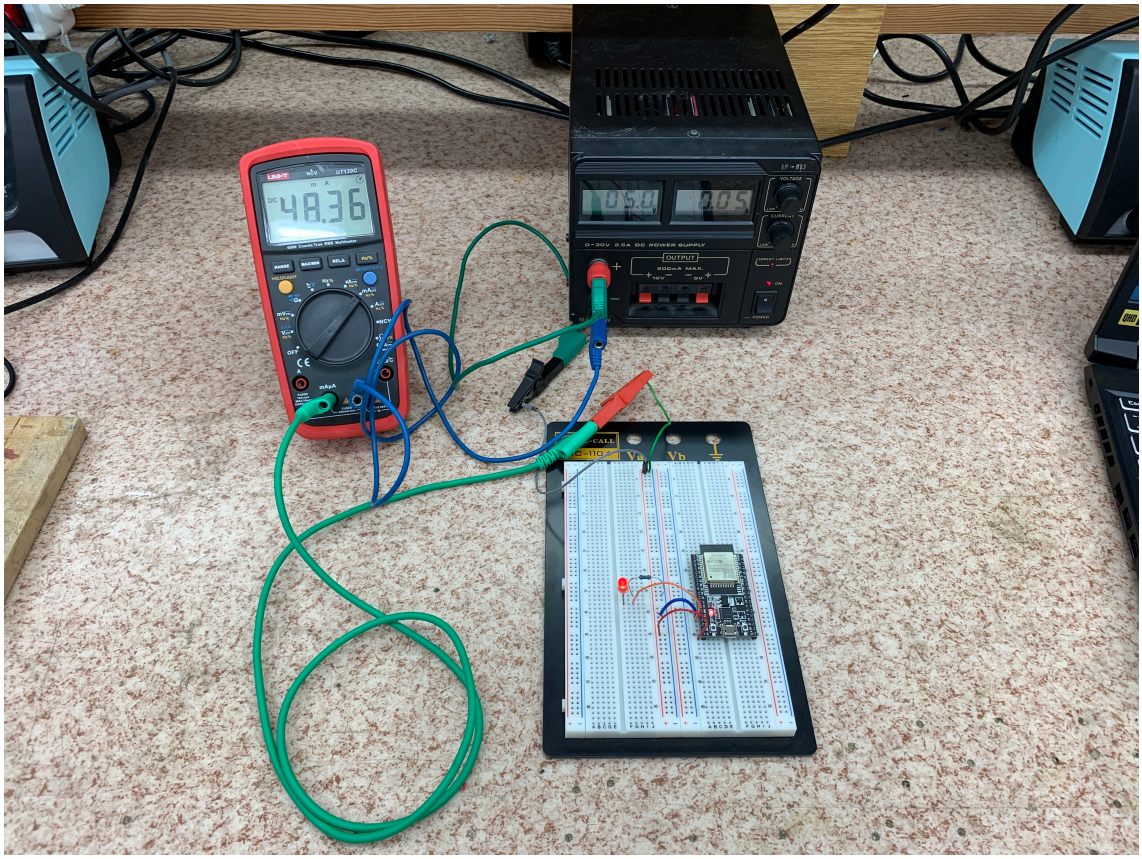
7 Testování

7.1 Spotřeba energie

Jako jeden z hlavních parametrů, které při práci měřím, je spotřeba elektrické energie mikrokontroléru. Vývojovou desku jsem vystavil různým situacím a měřil jsem odebíraný proud při použití různých programů. ESP32 se dle očekávání chovalo různorodě při použití rozdílných algoritmů a při připojení periferií. Během testování napájím desku přímo z portu počítače a pomocí USB micro portu na desce ESP. Toto zapojení mi umožňuje rychlou výměnu programů a sledování sériového monitoru pro kontrolu a výpis stavů. Pro přiblížení se reálnému provozu však musím zvolit jiný typ napájení. Je možné napájet mikrokontrolér přímo ze sítě, ale poté postrádájí smysl malé rozměry a přenosnost tohoto zařízení. Proto pro reálné testování používám powerbanku s kapacitou 20000 mAh. Při přímém měření s laboratorním zdrojem byla vývojová deska ESP32 napájena přímo z externího zdroje elektrické energie. Tato metoda umožnila přesné měření odebíraného proudu a napětí pomocí multimetru. Laboratorní zdroj poskytoval stabilní napětí a umožňoval měření spotřeby v různých scénářích. V průběhu měření byly získány údaje o spotřebě elektrické energie v různých režimech a situacích. Výsledky ukazují, jak mikrokontrolér reaguje na různé algoritmy, připojení periferií a běžné činnosti. Tyto informace jsou klíčové pro optimalizaci výkonu a efektivního využití mikrokontroléru v konkrétních aplikacích.

7.1.1 ESP32 bez programu a blikání LED

Prvním testovaným scénářem bylo měření spotřeby energie mikrokontroléru ESP32 v tzv. idle stavu, tedy když nebyl nahrán žádný program. Tato situace sloužila k získání referenční hodnoty o minimální spotřebě, které lze prakticky dosáhnout. Spotřeba energie v idle stavu je zhruba 47,82 mAh. Další testovací situací bylo naprogramování ESP32 tak, aby periodicky blikala LED dioda. Tím bylo sledováno, jak se spotřeba mění při neaktivním stavu a krátkých obdobích aktivity. V této situaci se spotřeba pohybovala mezi 47,30 mA (LED nesvítí) a 48,72 mA (LED svítí).



Obrázek 7.1: Měření spotřeby při blikání LED

Na obrázku výše vidíme měření spotřeby elektrické energie pomocí ampérmetru (multimetru) a laboratorního zdroje na ESP32 s programem pro blikání LED diody. Citlivost multimetru je nastavena na mA a deska je napájena ze zdroje 5V stejnosměrného napětí. Stejné podmínky platí i pro všechna další měření následujících scénářů. Na obrázku je dioda rozsvícena, proto se hodnota proudu vyšplhala k hodní hranici naměřených dat.

7.1.2 ESP32 Booting a Setup

Při tomto měření jsem zkoumal, kolik proudu ESP32 odebírá při startu a nastavování potřebných technologií. Měřil jsem prvotní bootování desky a průměr odběru proudu z 36 pokusů je 150 mA. Průměr při inicializaci a připojování k Wi-Fi routeru je 122,5 mA za dobu 36 sekund. A při připojování ESP32 k MQTT brokeru je průměrná spotřeba 76 mA za dobu 36 sekund. S jistotou tedy mohu říci, že největší odběr má mikrokontrolér při startu a prvotním nastavování všech modulů a technologií. V situaci při startu BLE serveru na ESP32 jsou odběry proudu i podmínky totožné s běžným již aktivním chodem tohoto serveru. Bluetooth modul na desce se totiž prvotně aktivuje již při bootování a po jeho spuštění (pro potřeby BLE) jsem tedy hodnoty neměřil.

7.1.3 ESP32 a MQTT

V této části testu byl mikrokontrolér naprogramován tak, aby se připojil k MQTT brokeru a periodicky odesílal a přijímal zprávy. Tato simulace reálné komunikace přes MQTT ukázala, jaký vliv má síťová činnost na spotřebu energie. Spotřeba energie zákonitě rapidně vzroste při použití Wi-Fi modulu, při připojení k routeru a při odesílání zpráv MQTT brokeru. Můžeme si všimnou, že průměrná hodnota je zhruba 71 mA, což je pouze o 5 mA méně než při samotném nastavování MQTT komunikace (připojování k brokeru). Směrodatná odchylka naměřených hodnot je 11,7 mA.

7.1.4 ESP32 a BLE

Testování s Bluetooth Low Energy protokolem bylo zaměřeno na sledování energetické náročnosti tohoto komunikačního protokolu. Průměrná hodnota odběru je 71 mA a směrodatná odchylka 0,13 mA. Zkoumal jsem, jak se spotřeba mění při aktivním BLE serveru a připojení periferních zařízení (mobilní aplikace a notebooku). Hodnota proudu nijak významně nekolísala. Byla prakticky až konstantní, což jsem neočekával. Odběr proudu se nezměnil ani při zasílání dat připojeným zařízením, ani v klidovém stavu serveru, ani při zvyšování počtu připojených klientů a ani při příchodu dat od klienta na server.

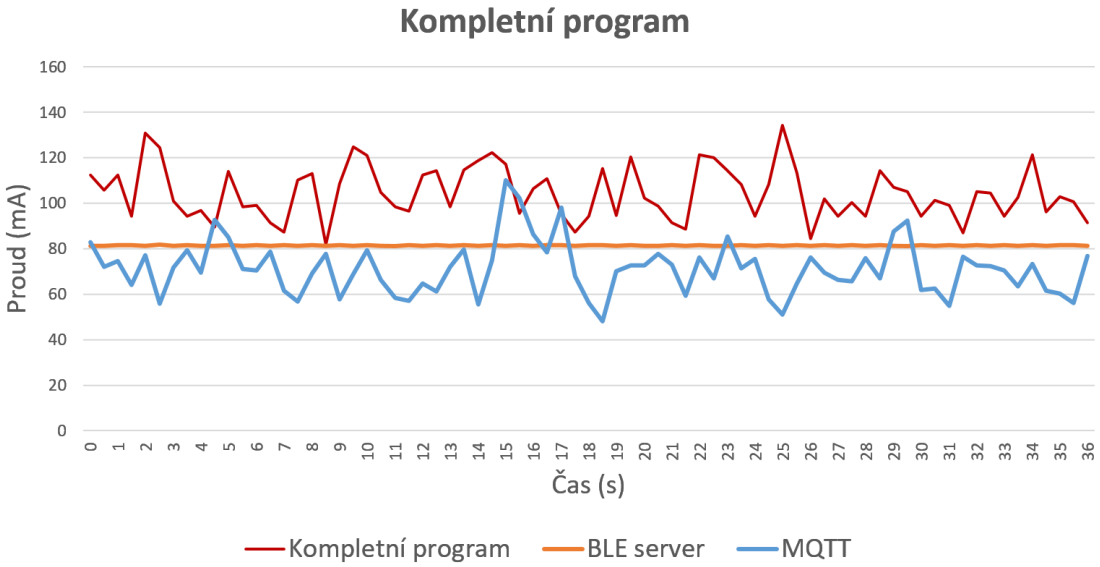
7.1.5 MQTT vs. BLE

Pro porovnání spotřeby elektrické energie na desce ESP32 při použití komunikačního protokolu MQTT a při použití protokolu BLE jsem data zpracoval do grafu níže. Z grafu je na první pohled patrné, že protokol MQTT při práci potřebuje v průměru menší přísun elektrického proudu než protokol BLE. Naměřené hodnoty u MQTT kolísaly mezi hodnotami 48 mA a 110 mA. Hodnoty proudu u protokolu BLE byly téměř konstantní a to mezi hodnotami 81 mA a 82 mA. Průměrná hodnota u MQTT je zhruba 71 mA a u BLE je 81 mA. Prokol MQTT si tedy stojí v průměru o 12% lépe ve spotřebě elektrické energie než protokol BLE. Což je obráceně než mé očekávání dle zjištěných informací. Avšak musíme brát v potaz, že nijak nenastavuji velikost zasílaných paketů a mechanismus QoS u MQTT je nastavený na nejmenší (nejúspornější a nejrychlejší) úroveň 0 [5]. Také nijak blíže nespecifikuji vnitřní nastavení BLE serveru. Navíc na vývojové desce ESP32 běží pro protokol BLE přímo server i klient a pro protokol MQTT realizuji na straně ESP32 pouze MQTT klienta a ne celý MQTT broker.

7.1.6 ESP32 a kompletní program

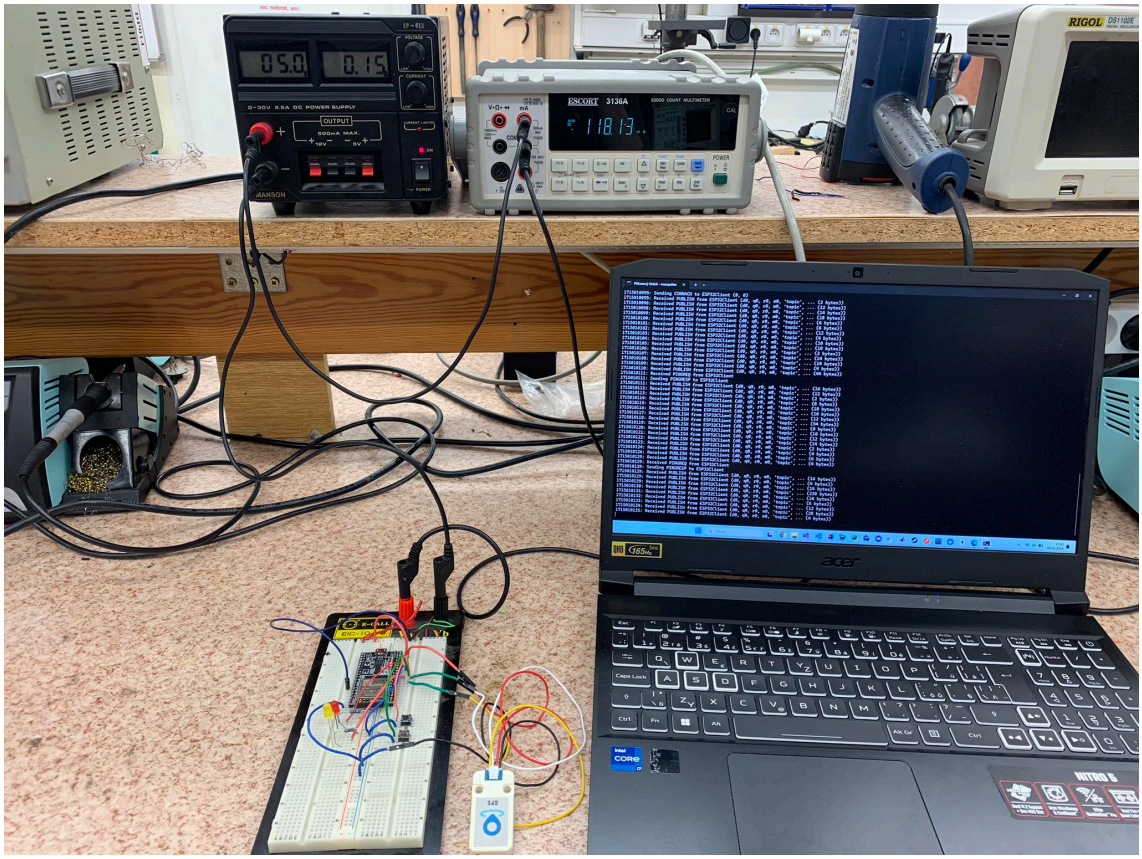
V této části byl na mikrokontrolér nahrán kompletní programem, který kombinoval všechny zmíněné činnosti, jako například čtení a zápis dat do paměti, komunikaci přes komunikační protokoly MQTT a BLE. Na mikrokontroléru ESP32 jsou spuštěna obě dvě jádra a na každém běží jiná úloha. Program a hardware jsem se snažil

co nejvíce přizpůsobit k co nejmenší spotřebě elektrické energie. Vypnul jsem nepotřebné funkce a komunikace, optimalizoval jsem kód a našel jsem jiné knihovny pro co nejlepší výsledky. Tato situace poskytla komplexní pohled na celkovou spotřebu v reálných podmínkách. Udělal jsem dvě různá měření. Jedno jsem měřil veškerou spotřebu desky od prvotního spuštění po dobu jedné minuty. Nejdříve se vytvoří a inicializují potřebné proměnné. Inicializuje se mutex, piny pro LED diody, piny pro přerušení a další funkcionality. Při prvních pár vteřinách ESP32 spotřebuje nejvíce proudu. Poté se mikrokontrolér bude snažit připojit k Wi-Fi, k MQTT brokeru, nastaví a spustí BLE server. Všechny tyto procesy proběhnou jeden po druhém, jelikož jeden bez druhého nemohou fungovat. Každý má při svém vykonávání jinou hodnotu spotřeby energie. Po úspěšném připojení a nastavení se spustí paralelně funkce obou jader, spustí se komunikace a sběr dat. Data pro graf spotřeby s kompletním programem jsem naměřil až po úspěšné inicializaci všech komponent a funkcionalit. Měřil jsem tedy až poté, co jsem na MQTT brokeru (konzole PC) a na BLE klientu (mobilní aplikaci) viděl příchozí data z ESP32. Což znamená pouze výslednou komunikaci a sběr dat.



Obrázek 7.2: Graf spotřeby s jednotlivými programy

Spotřeba elektrické energie se pohybovala mezi hodnotami 82,4 a 138,5 mA při plném využití všech požadovaných funkcionalit. Průměrná hodnota spotřeby dosáhla 104,6 mA. Pro získání komplexnějšího pohledu jsem do grafu zahrnul také data z předchozích měření. Porovnání průměrných hodnot ukazuje, že nejvyšší spotřeba nastává v situaci, kdy je program kompletní. V této konfiguraci jsou aktivní obě jádra ESP32, běží většina modulů (Wi-Fi, Bluetooth) a je připojeno více periferií, jako jsou LED diody a tlačítka.



Obrázek 7.3: Měření spotřeby použití kompletního programu

Na obrázku výše můžeme vidět měření spotřeby mikrokontroléru při použití kompletního programu, kdy ESP32 sbírá data z připojeného GPS senzoru přes rozhraní UART. V konzoli notebooku můžeme živě vidět příchozí data z ESP32 na MQTT broker. Deska je napájena z laboratorního zdroje 5V a procházející proud je měřen digitálním multimetrem Escort 3136A.

7.1.7 Výsledky měření spotřeby

Z výsledků měření bylo zřejmé, že spotřeba mikrokontroléru závisí na konkrétních činnostech, technologiích a periferiích, které jsou v daném okamžiku aktivní. Nejvíce nás ale zajímá spotřeba energie při plném aktivním stavu. V něm tedy používám kompletní program s veškerou popisovanou logikou. Průměrná spotřeba kompletního aktivního systému je 104,6 mA. Pro demonstraci reálného kompaktního systému používám powerbanku s kapacitou 20000 mAh. Při napájení z této baterie by celý systém vydržel v plně aktivním stavu zhruba 191 hodin, tedy necelých 8 dní.

Použitý program	Spotřeba elektřiny [mA]	Směrodatná odchylka [mA]	Výdrž baterie [dny]
Blikání LED	47,51	0,39	17,6
MQTT	71	11,7	11,7
BLE	81	0,13	10,3
Kompletní	104,6	19,6	7,9

Tabulka 7.1: Tabulka spotřeby elektrické energie

V tabulce jsou uvedeny naměřené hodnoty aktivního běhu systému při použití různých programů a jejich směrodatné odchylky. Uvedl jsem také vypočtenou dobu výdrže systému při napájení z baterie o kapacitě 20000mAh. Například malá svítilna může spotřebovat 100mA až 300mA. Pro porovnání mobilní telefon při aktivním používání spotřebuje 300mA až 2000mA v závislosti na typu aktivity [18]. Při stejném odběru (105mA) by mobilní telefon, který má průměrnou kapacitou baterie (4000mAh) vydržel aktivně běžet 1,6 dne.

7.2 Zasílání paketů

ESP32 četlo pravidelně každých 10 ms data z GPS modulu pomocí rozhraní UART a velikost jedné zprávy je přibližně 320 bytů. Průměrný bit rate této komunikace, který odpovídá přenosové rychlosti dat, byl přibližně 32 kbps. Následně je mikrokontrolér ukládal do souboru na svém úložišti.

ESP32 periodicky odesílal data z tohoto souboru přes protokol MQTT na zadanou adresu brokeru každé 2 sekundy, což umožňovalo jejich monitorování a zpracování na vzdáleném serveru. Pro testování jsem zasílal vždy aktuální uloženou informaci z GPS modulu ze souboru o velikosti zmíněných 320 bytů.

Současně ESP32 vysílal stejná data prostřednictvím Bluetooth Low Energy (BLE) také každé 2 sekundy, což umožňovalo jejich příjem na zařízeních s podporou BLE. Zde jsem zasílal stejná data jako přes komunikaci MQTT.

7.3 Chybovost

I přes poměrně jasně definované a zadané instrukce může dojít k jisté chybovosti systému. Algoritmy a funkce používané v programu jsou velmi komplexní a přirozeně může dojít k chybám, či neočekávanému chování. Například nerovnoměrné zpracování jádrových úloh představuje problém, který může negativně ovlivnit spolehlivost a stabilitu systému. I přes použití synchronizačních mechanismů, jako jsou mutexy, může dojít k asynchronnímu chování mezi jádry, kdy jedno jádro stihne dokončit svou úlohu a provést zápis do sdílené paměti dříve než druhé jádro. Tento problém často vzniká v důsledku nerovnoměrného zpracování úloh jádry, přestože

mají nastavena stejná zpoždění pomocí funkce delay(). Pro eliminaci této chyby je nezbytné implementovat dodatečné mechanismy pro řízení a synchronizaci spouštění jádrových úloh tak, aby bylo zajištěno rovnoměrné zpracování úloh a minimalizace konfliktů při přístupu k sdíleným datům.

Tuto chybovost mohu demonstrovat pomocí programu, při kterém obě jádra zapisují do jedné sdílené souborové proměnné. Jádro číslo jedna bude zapisovat do souboru "1", číst z něj a zasílat data přes MQTT na server. Druhé jádro bude zapisovat do souboru "0", číst z něj a zasílat data pomocí protokolu BLE všem připojeným zařízením. Problém s nekonzistencí dat a souběhem jader je vyřešený. Může však nastat situace, při které jedno jádro předstihne to druhé a zapíše do souboru dvakrát. Při tomto scénáři může vypadat soubor uvnitř následovně: "01010101001010101010010...". I přes synchronizaci zapisování do sdíleného souboru a nastavení stejně dlouhé prodlevy na obou jádrech dochází k občasné chybě, kde jádro číslo 0 stihne svou úlohu vykonat rychleji a předběhne jádro číslo 1. Po zkoumání tohoto jevu jsem došel k závěru, že důvodů pro toto chování může být více. Nejsem si jistý zda by problém mohl být i přímo v hardwaru, respektive v architektuře mikrokontroléru. Je možné, že první jádro je schopno pracovat rychleji než to druhé i při stejných podmínkách. Příčinou mohou být také samotné algoritmy komunikačních protokolů. Nedokáži změřit, jak dlouhý je interval doby od zaslání paketů zprávy z desky ESP32 do přijmutí dat Serverem/Klientem. Ať už u protokolu MQTT, nebo u protokolu BLE. Nejsem schopný změřit přesnou dobu příchodu zprávy na MQTT server, nebo na připojená zařízený k BLE serveru. Mohu ale zjistit rozestupy dob mezi jednotlivými zaslanými daty skrze protokol BLE. Pro připojení k BLE serveru na ESP32 používám chytrý telefon s aplikací navrženou na připojení a odchytávání BLE komunikace [38]. Po připojení k BLE serveru vidím v aplikaci příchozí zprávy s daty a časy doručení. Rozestup, i přes nastavení v programu ESP32 na 2 sekundy, se pohybuje mezi 1,9 až 2,2 sekundy. Rozpětí 300 ms opravdu není zanedbatelné, proto také může dojít k opoždění a následné nekonzistenci. Oproti tomu protokol MQTT má různé stupně nastavení QoS (Quality of Service), kdy může dojít ke ztrátě některých dat, či ke zpoždění pro zaručení kvality. Dále má rozdílné mechanismy zabezpečení než protokol BLE. Použití například filtrace IP adres, nastavení jména a hesla, či přidání zabezpečení pomocí TLS (Transport Layer Security) může a nejspíše dojde ke zpoždění, které však nejsem schopný rozumně naměřit. Proto v tomto scénáři hraje roli spoustu faktorů a opravení této chyby je nad rámec mých dosavadních schopností. Mohl bych ale například vyvinou algoritmus, který by tento soubor hlídal a případně by upravoval chyby v něm. To by bylo ale za předpokladu, že data budou mít konstantní hodnoty a v praxi by to nedávalo smysl.

Dalším významným problémem může být ztráta zpráv při komunikaci mezi zařízeními pomocí komunikačních protokolů. Tento problém může nastat v důsledku nedostatečné spolehlivosti síťového spojení, nedostupnosti cílového zařízení nebo rušení signálu v prostředí s mnoha rušivými faktory. Ztráta zpráv může vést k narušení integrity dat, chybnému vyhodnocení stavu zařízení nebo nesprávné funkčnosti celého systému. Pro řešení tohoto problému je důležité implementovat mechanismy pro kontrolu spolehlivosti komunikace, jako jsou potvrzovací zprávy, opakování zpráv a mechanismy pro detekci a obnovení ztracených zpráv. To umožní zajištění spoleh-

livé a robustní komunikace mezi zařízeními, i při výskytu nepříznivých podmínek nebo rušení signálu v prostředí. Pro kontrolu komunikace na jádře 0 při použití protokolu MQTT je možné implementovat ověření doručení zprávy přímo pomocí prvků a metod tohoto protokolu. Pokud ESP32 tzv. publikuje zprávu přes MQTT na konkrétní topic, mohu tento topic také odebírat a MQTT broker mi zašle zpět to, co ESP32 na příslušný topic poslalo. Takto jsem naprogramoval mechanismus pro ověření, zda zpráva byla skutečně úspěšně zaslána a dorazila na broker. Co se týče kontroly zasílání pomocí protokolu BLE mohu použít funkci přímo z knihovny ArduinoBLE - bleCharacteristic.writeValue(). Ta mi vrátí hodnotu 0 při neúspěšném odeslání zprávy připojenému zařízení a hodnotu 1 při úspěšném zaslání dat [3]. Bohužel ale nikde nedostanu informaci, že tato připojená periferie data skutečně přijala. Data se nemusela dostat až k tomuto připojenému koncovému zařízení, ale mohla se jen správně odeslat z mikrokontroléru ESP32, tedy z BLE serveru. Příchozí zprávy však mohu sledovat přímo v BLE aplikaci na mém mobilním zařízení, tedy na BLE periferii. Kontrolu mohu tedy zde provést manuálně a nikoli pomocí programu. Pokud bych chtěl napodobit kontrolu jako při použití protokolu MQTT musím upravit tento BLE server na desce ESP32, aby byl přístupný i pro data zasílaná z periferií a nesloužil jen jako vysílač. Tento přístup, kdy ESP32 zašle zprávu přes protokol BLE připojené periferii a periferie zašle zprávu zpět, se mi povedlo plně uvést do provozu. BLE server na ESP32 zaslal zprávu mému notebooku, který je připojený k tomuto BLE serveru a přes Python script jsem schopný tyto zasílaná data zachytit a automaticky poslat zpět serveru (ESP32), který je nastavený jako vysílač i jako přijímač [7]. Takto mohu automatizovaně ověřil, že data na připojenou periferii skutečně dorazila. Příchozí i odchozí zprávy z ESP32 mohu kontrolovat pomocí výpisů přímo na sériovém monitoru v IDE.

Závěr

Podařilo se mi vytvořit kompaktní systém napájený baterií stavěný na platformě ESP32. Mikrokontrolér ESP32 sbírá lokální data z modulu GPS přes sériový protokol UART. Data jsem v reálném čase zpracovával a úspěšně přenášel do nadřazeného systému pomocí komunikačních protokolů MQTT a BLE, které se běžně používají v chytrých budovách. Archivaci dat jsem prováděl jak na samotném mikrokontroléru, tak i na připojeném notebooku pomocí souborové databáze (viz 5.4).

Sestavil jsem ekosystém, ke kterému jsem připojil více prvků (klientů). Klienti taktéž dostávali aktuální data v reálném čase přímo z ESP32. Komunikaci jsem zabezpečil dodatečnými mechanismy. Nikdo neautorizovaný se nemůže k síti připojit. Implementace systému je vícevláknová a synchronizovaná, takže se nijak neruší žádný z probíhajících procesů - sběr dat, archivace, přenášení zpráv přes MQTT a přes BLE. Ošetřil jsem rizikové situace, souběhy a možnou duplicitu dat na straně ESP32 (viz 5.3). Systém je také responzivní na externí podněty díky možnosti vyvolat přerušení komunikace na daném jádře pomocí připojených tlačítek (viz 4.5).

Celý systém jsem otestoval, zda dokáže data přenášet, archivovat, duplicitu dat a případnou chybovost (kapitola 7). Popsal jsem neošetřené chyby a jejich možné řešení (viz 7.3). Otestoval jsem a změřil elektrickou spotřebu tohoto kompaktního systému, která při aktivním chodu byla v průměru 105 mA (viz 7.1). Uvedl jsem také příklady využití v praxi a možnosti dalšího rozvoje tohoto ukázkového systému . S dosaženými výsledky a měřeními jsem spokojen. Překvapilo mě, jak rychle dokáže mikrokontrolér data přijímat a odesílat, a to dokonce i paralelně. Také při přidání mechanismů zabezpečení se spotřeba energie nijak výrazně nezvyšovala. Dle očekávání se nejvíce zvýšila při zapnutí a aktivním využívání více modulů ESP32 najednou (viz 7.1).

V návaznosti na tuto práci bych rád zdůraznil důležitost dalšího zabezpečení sítě. Skvělé by bylo přidat zabezpečení pomocí TLS (Transport Layer Security), aby data byla šifrovaná při přenosu. V mém případě zabezpečuji celé připojení k brokeru a sdílené komunikaci, ale data nejsou při přenášení šifrována. Je také možné zaměřit se na spotřebu energie a optimalizovat procesy, které na ESP32 běží. To však již záleží na konkrétní požadované aplikaci tohoto systému. Pokud nebude potřeba číst data každých 10 ms a zasílat je každých 2 sekund, jako je tomu v mém případě, je možné mikrokontrolér uvést do režimu spánku a aktivovat jej pouze v případě potřeby, aby se maximalizovala úspora energie.

Další možnost rozšíření je například implementace automatické detekce událostí, která také ovlivní spotřebu energie, nebo přidání senzorů. Případně i vytvoření vlastního uživatelského rozhraní (GUI) pro monitorování příchozích dat a v neposlední řadě skutečná integrace do systému chytré budovy.

Bibliografie

Odkazy

[1] Adafruit learning system. *GATT*. 8.03.2024. URL: https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gatt.

[2] Arduino. *Arduino UNO R3 ATmega328P Manual*. 01.02.2024. URL: https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf.

[3] Arduino SA. *ArduinoBLE Library*. 2019. URL: https://www.arduino.cc/reference/en/libraries/arduinoble/.

[4] Bluetooth SIG. *Bluetooth Technology Overview*. [Navštíveno 28.12.2023]. 2023. URL: https://www.bluetooth.com/learn-about-bluetooth/tech-overview/.

[5] Communication Systems. *How can you optimize MQTT for low-power devices?* Navštíveno 8.03.2024. URL: https://www.linkedin.com/advice/1/how-can-you-optimize-mqtt-low-power-devices-p2sze.

[6] *crypt(3) - Linux man page*. Navštíveno 05.04.2024. URL: https://linux.die.net/man/3/crypt.

[7] David Kárník. *GitHub repozitář kódu k bakalářské práci, ESP32withMQTTandBLE*. 2023-2024. URL: https://github.com/DavidKarnik/ESP32withMQTTandBLE.

[8] Dr. Andy Stanford-Clark, Arlen Nipper. *MQTT*. 1999. URL: https://mqtt.org/.

[9] Eclipse Foundation. *mosquitto_passwd man page*. Navštíveno 24.10.2023. URL: https://mosquitto.org/man/mosquitto_passwd-1.html.

[10] Eclipse Foundation. *Version 2.0.0 released*. 12.03.2020. URL: https://mosquitto.org/blog/2020/12/version-2-0-0-released/.

[11] Eclipse, Roger Light. *paho-mqtt library*. 21.10.2021. 2021. URL: https://pypi.org/project/paho-mqtt/.

[12] Embeded Centric. *BLE profiles, services, characteristics, device roles and network topology*. Navštíveno 3.05.2024. URL: https://embeddedcentric.com/lesson-2-ble-profiles-services-characteristics-device-roles-and-network-topology/.

[13] Eric Peňa, Mary Grace Legaspi. *UART: A Hardware Communication Protocol*. Navštíveno 20.04.2024. URL: https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html.

[14] ESPRESSIF. *ESP-WROOM-32 Datasheet*. Navštíveno 12.09.2023. URL: https://html.alldatasheet.com/html-pdf/1179101/ESPRESSIF/ESP-WROOM-32/571/1/ESP%02WROOM-32.html.

[15] Espressif Systems. *ESP32-DevKitC V4 Getting Started Guide*. Navštíveno 08.04.2024. URL: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/hw-reference/esp32/get-started-devkitc.html.

[16] Espressif Systems. *FreeRTOS*. 2021. URL: https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/api-reference/system/freertos.html.

[17] Espressif Systems. *SPIFFS Filesystem*. 2021. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html.

[18] Everphone. *Smartphone battery capacity*. Navštíveno 29.04.2024. URL: https://everphone.com/en/wiki/battery-capacity-smartphone/.

[19] Gaurav Kunal. *BLightweight IoT Protocols for Seamless Device Integration*. 20.08.2023. URL: https://www.softobotics.com/blogs/lightweight-iot-protocols-for-seamless-device-integration/.

[20] Hadex s.r.o. *Vývojová deska s modulem ESP-WROOM-32D*. Navštíveno 28.04.2024. URL: https://www.hadex.cz/m432n-esp32-devkitc-vyvojova-deska-s-modulem-esp-wroom-32d.

[21] HiveMQ Team. *MQTT Publish/Subscribe Architecture (Pub/Sub)*. 6.06.2023. URL: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/.

[22] HiveMQ Team. *OAuth 2.0 & MQTT - MQTT Security Fundamentals*. 6.03.2024. URL: https://www.hivemq.com/blog/mqtt-security-fundamentals-oauth-2-0-mqtt/.

[23] HiveMQ Team. *What is MQTT Quality of Service (QoS) 0,1, & 2?* 20.02.2024. URL: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

[24] HW group s.r.o. *MQTT - univerzální protokol pro cloudové a IoT aplikace*. Navštíveno 19.03.2024. URL: https://www.hw-group.com/cs/podpora/mqtt-univerzalni-protokol-pro-cloudove-a-iot-aplikace.

[25] LaskaKit s.r.o. *Deska Arduino Uno Rev3*. Navštíveno 28.04.2024. URL: https://www.laskakit.cz/arduino-uno-rev3--original/.

[26] LaskaKit s.r.o. *Raspberry Pi Pico W*. Navštíveno 28.04.2024. URL: https://www.laskakit.cz/raspberry-pi-pico-w/.

[27] Laurenz Dallinger. *Mastering MQTT Packet Guide*. 25.05.2023. URL: https://cedalo.com/blog/mqtt-packet-guide/.

[28] Laurenz Dallinger. *Understanding an MQTT Packet: Ultimate Guide*. 25.05.2023. URL: https://cedalo.com/blog/mqtt-packet-guide/.

[29] MOHAMMAD AFANEH. *Mastering BLE: A Guide to Peripherals and Centrals*. 7.06.2023. URL: https://novelbits.io/ble-peripherals-centrals-guide/.

[30] Mohammad Afaneh. *A Deep Dive into BLE Packets and Events*. 30.03.2020. URL: https://novelbits.io/deep-dive-ble-packets-events/.

[31] Nordic Developer Academy. *Advertisement packet*. Navštíveno 5.05.2024. URL: https://academy.nordicsemi.com/courses/bluetooth-low-energy-fundamentals/lessons/lesson-2-bluetooth-le-advertising/topic/advertisement-packet/.

[32] Northwestern University - Electrical and Computer Engineering. *BaudRate*. Navštíveno 3.05.2024. URL: http://www.ece.northwestern.edu/local-apps/matlabhelp/techdoc/matlab_external/baudrate.html.

[33] Pájeníčko s.r.o. *Modul Arduino Uno Rev3*. Navštíveno 28.04.2024. URL: https://pajenicko.cz/arduino-uno-rev3-original.

[34] Panasonic. *Panasonic Support*. [Navštíveno 27.12.2023]. URL: https://support-cz.panasonic.eu/app/answers/detail/a_id/7185/~/co-je-%E2%80%9Eble%E2%80%9C-bluetooth-low-energy%3F.

[35] PlatformIO Labs. *PlatformIO*. 2014. URL: https://platformio.org/.

[36] PlatformIO Labs. *PlatformIO IDE Extension for VSCode*. Navštíveno 28.04.2024. URL: https://marketplace.visualstudio.com/items?itemName=platformio.platformio-ide.

[37] Punch Through. *How GAP and GATT Work — Bluetooth Low Energy Basics*. 1.04.2024. URL: https://punchthrough.com/how-gap-and-gatt-work/.

[38] Punch Through. *LightBLue - The Go-To BLE Development Tool*. Navštíveno 10.11.2023. URL: https://punchthrough.com/lightblue/.

[39] Raspberry Pi Ltd. *Raspberry Pi Pico W Datasheet*. Navštíveno 28.04.2024. URL: https://datasheets.raspberrypi.com/picow/pico-w-datasheet.pdf.

[40] Rohde & Schwarz, Inc. *Understanding UART*. Navštíveno 3.05.2024. URL: https://www.rohde-schwarz.com/us/products/test-and-measurement/essentials-test-equipment/digital-oscilloscopes/understanding-uart_254524.html.

[41] RPishop.cz. *M5Stack Mini GPS/BDS modul (AT6558)*. Navštíveno 25.04.2024. URL: https://rpishop.cz/504086/pouzity-m5stack-mini-gps-bds-modul-at6558/#tab-description.

[42] UNIVERSITY of WISCONSIN–MADISON. *UART Basics*. Navštíveno 3.05.2024. URL: https://ece353.engr.wisc.edu/serial-interfaces/uart-basics/.

[43] Uri Shaked. *Online ESP32 Simulator Wokwi*. Navštíveno 12.02.2024. URL: https://wokwi.com/esp32.

[44] viewtool. *One minute to understand BLE MTU data package*. 15.12.2017. URL: https://devzone.nordicsemi.com/nordic/nordic-blog/b/blog/posts/one-minute-to-understand-ble-mtu-data-package.

[45] Zibo Zhou. *MQTT QoS 0, 1, 2 Explained*. 12.01.2023. URL: https://www.emqx.com/en/blog/introduction-to-mqtt-qos.

Přílohy

Zdrojové kódy

https://github.com/DavidKarnik/ESP32withMQTTandBLE