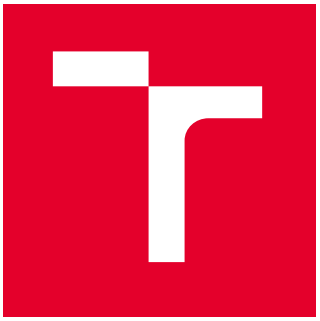


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

ZABEZPEČENÍ ASYNCHRONNÍ SÉRIOVÉ KOMUNIKACE PŘES TCP/IP

SECURITY OF ASYNCHRONOUS SERIAL COMMUNICATION VIA TCP/IP

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Jarmila Tichá**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Vladislav Škorpil, CSc.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Telekomunikační a informační technika**

Ústav telekomunikací

Studentka: Bc. Jarmila Tichá ***ID:*** 195452

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Zabezpečení asynchronní sériové komunikace přes TCP/IP

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s fungováním komunikačních protokolů, které využívají UART zařízení a standardních protokolů určených k zabezpečení síťové komunikace. Porovnejte je s ohledem na využití při šifrování komunikace UART zařízení přes TCP/IP. Podstatné náležitosti popište, rozeberte výhody a nevýhody možných řešení s ohledem na navazující implementaci. Na podkladě získaných poznatků navrhněte a implementujte aplikaci pro volně dostupný RTOS běžící na mikrokontroléru se schopností navázat spojení s externím UART zařízením a zároveň s jiným zařízením v síti podporujícím TCP/IP (možné použít stejný mikrokontrolér na obou komunikujících stranách nebo standardní PC s komplexním OS). Aplikace umožní jednoduchým uživatelským rozhraním konfigurovat alespoň základní nastavení pro komunikaci přes zmíněné protokoly (IP adresy pro síťovou komunikaci, baud rate pro UART, atp.). Pomocí vlastního řešení zprostředkujte komunikaci mezi dvěma UART zařízeními připojenými k různým zařízením v rámci LAN, přičem komunikace bude používat k zabezpečení SSL/TLS protokol. Zanalyzujte a popište časovou náročnost a dosaženou přenosovou rychlost navrhnutého systému v porovnaní s přímou komunikací UART zařízení.

DOPORUČENÁ LITERATURA:

[1] B. Amos, Hands-On RTOS with Microcontrollers: Building real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools. Birmingham, England: Packt Publishing, 2020.

[2] P. Baka and J. Schatten, SSL/TLS under lock and key: A guide to understanding SSL/TLS cryptography. Keyko Books, 2021.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: doc. Ing. Vladislav Škorpil, CSc.

Konzultant: Mgr. Dominik Mlynka, Honeywell International, s.r.o.

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

Abstrakt

Diplomová práce je rozdělena na dvě části - teoretickou a praktickou. Teoretická část popisuje různé typy sériových komunikačních zařízení s důrazem na Univerzální asynchronní přijímač a vysílač. Dále se věnuje rozdílům mezi běžným operačním systémem a operačním systémem reálného času, jejich struktuře a funkcím. Představuje volně dostupné operační systémy vhodné pro implementaci. Dále popisuje vývojové kity vhodné pro implementaci operačního systému reálného času a uvádí komunikační protokoly používané pro síťovou komunikaci.

V praktické části se podrobně zabývá tvorbou aplikací pro komunikaci UART a popisuje jejich implementaci na jednotlivých zařízeních. Dále vysvětlujeme postup pro zapojení přes Ethernet a popisuje funkce jednotlivých aplikací a prezentuje výsledky měření spolu s teoretickými výpočty, které slouží k určení očekávané hodnoty přenosu.

Klíčová slova

RTOS, mikrokontrolér, SSL/TLS komunikace, TCP komunikace, Python, C/C++, měření času

Abstract

The thesis is divided into two parts - theoretical and practical. The theoretical part describes various types of serial communication devices with an emphasis on Universal asynchronous receiver and transmitter. It also covers the differences between a regular operating system and a real-time operating system, their structure and functions. Freely available operating systems suitable for implementation are presented. It also describes development kits suitable for implementing a real-time operating system and lists the communication protocols used for network communication.

In the practical part, it deals in detail with the creation of applications for UART communication and describes their implementation on individual devices. Next, the procedure for connection via Ethernet is explain and the functions of individual applications are described. The thesis also present the measurement results together with theoretical calculations that serve to determine the expected transmission value.

Keywords

RTOS, microcontroller, SSL/TLS communication, TCP communication, Python, C/C++, timing

Bibliografická citace

Tichá, J. Zabezpečení asynchronní sériové komunikace přes TCP/IP. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023. 83 s., 3 s. příloh. Diplomová práce. Vedoucí práce: doc. Ing. Vladislav Škorpil, CSc.

Prohlášení autora o původnosti díla

Jméno a příjmení studenta:	Bc. Jarmila Tichá
VUT ID studenta:	195452
Typ práce:	Diplomová práce
Akademický rok:	2022/2023
Téma závěrečné práce:	Zabezpečení asynchronní sériové komunikace přes TCP/IP

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

.

V Brně dne: 18. května 2023 ----------

podpis autora

Poděkování

Děkuji vedoucímu diplomové práce doc. Ing. Vladislavovi Škorpilovi, CSc, za účinnou metodickou, pedagogickou a panu Mgr. Dominikovi Mlynkovi za odbornou pomoc a další cenné rady při zpracování mé diplomové práce.

V Brně dne: 18. května 2023 ----------

podpis autora

Obsah

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

Úvod

Diplomová práce je rozdělena do dvou částí: teoretické a praktické. V teoretické části se zabýváme problematikou, která je nezbytná pro splnění zadání. V praktické části se pak zaměřujeme na konkrétní zařízení a programy, které jsme použili, a popisujeme jejich funkcionalitu.

Teoretická část zahrnuje kapitolu o datové komunikaci, kde jsou popsány různé typy sériových komunikačních zařízení, s důrazem na Univerzální asynchronní přijímač a vysílač. Dále budou stručně popsány typy sběrnic, které tyto přijímače/vysílače využívají.

Další kapitola se zaměřuje na běžné operační systémy, jejich vznik, a popisuje různé typy front a dalších funkcí. Tato kapitola bude následována kapitolou o operačních systémech pracujících v reálném čase, kde budou vysvětleny různé typy těchto systémů a jejich jádrové úlohy. Dále budou uvedeny přehledy volně dostupných operačních systémů, které by mohly být vhodné pro implementaci v rámci diplomové práce.

Čtvrtá kapitola se zaměřuje na vhodné desky pro implementaci operačního systému pracujícího v reálném čase. Na základě této kapitoly budeme v praktické části vybírat konkrétní desku.

Poslední kapitola v teoretické části se bude věnovat komunikačním protokolům používaným pro síťovou komunikaci. Tyto protokoly budou následně implementovány v praktické části.

V rámci praktické části budou podrobně popsány aplikace pro UART komunikaci a jejich implementace na jednotlivých zařízeních. Budou také uvedeny výsledky měření pro tuto komunikaci.

Pro zapojení přes Ethernet bude také poskytnut popis toho, co každá aplikace provádí. Bude zde popsána volba desky na základě teoretické části. Následně bude vysvětleno, jak vytvořit spustitelný soubor pro danou desku. Jako poslední budou uvedeny výsledky měření spolu s teoretickými výpočty. Teoretické výpočty budou sloužit k určení očekávané hodnoty přenosu.

V závěru práce budou shrnuty všechny dosažené cíle a bude vyhodnoceno, zda měření splnilo očekávané teoretické výpočty.

1. DATOVÁ KOMUNIKACE

Sériová komunikace je proces, během kterého se odesílají data po jednotlivých bitech obvykle ve formě dvojstavových impulzů, přes počítačovou sběrnici nebo komunikační kanál. Jedná se o jednoduchou komunikaci mezi odesílatelem a příjemcem. Tento typ komunikace se používá pro přenos dat na velkou vzdálenost. Pro komunikaci je potřeba méně vodičů v porovnání s paralelní komunikací a je tak ekonomicky výrazně výhodnější. [1] Sériová komunikace nejčastěji probíhá mezi dvěma zařízeními, nicméně obecně může být ke sběrnici připojeno n-zařízení. Těmito zařízeními bývá obvykle počítač nebo mikrokontroler v pozici kontroléru sběrnice a například osciloskop a ostatní počítače nebo mikrokontrolery jako ovládané zařízení. [2]

1.1 Komunikační protokol

Komunikační protokol definuje pravidla, syntaxi (způsob zápisu), strukturu a prostředky synchronizace komunikace a také možné metody detekce a obnovy vzniklých chyb. Protokol může být implementován na úrovni hardware, software nebo pro obojí. [2] Komunikační protokoly se rozdělují do dvou skupin:

- a to na mezi-systémové (Intra-system) a
- interní protokoly (Inter-system).

Základní přehled komunikačních protokolů můžete vidět v Tabulka 1 a
Tabulka 2. Komunikační protokoly UART a USART budou popsány více v kapitole 1.3 a 1.4

Tabulka 1 Mezi-systémové protokoly

UART	**USART**	**USB**
Univerzální asynchronní vysílač a přijímač	Univerzální asynchronní a synchronní vysílač a přijímač	Univerzální sériová komunikace
Dvouvodičový	Dvouvodičový protokol	Dvouvodičový chránič
Přenáší datové pakety bez tříd	Přenáší datové pakety s třídami	Odesílá a přijímá data s hodinovými impulsy
Poloviční duplex	Plně duplexní	Plně duplexní
Pomalý ve srovnání s USART	Pomalý ve srovnání s USB	Rychlý ve srovnání s UART a USART

Tabulka 2 Interní protokoly

I2C	*SPI*	*CAN*
Mezi integrované obvody	Periferní rozehranní	Oblastní síť
Developer Phillips	Developer Motorola	Developer Robert Bosch
Polo duplexní	Plně duplexní	Plně duplexní
Synchronizační	Synchronizační	Synchronizační
Multi-master protokol	Single Master protokol	Multi-master protokol
Uvnitř obvodové desky	Uvnitř obvodové desky	Uvnitř dvou obvodových desek

1.1.1 Endianita bitů

Endianita určuje pořadí odvysílaných bitů v rámci komunikačního protokolu. Obecně existují dvě varianty – malý a velký Endian. Malý Endian (little endian) definuje přenos nejméně signifikantního bitu jako prvního, zatímco v případě velkého endianu (big endian) je jako první odvysílán nejvíce signifikantní bit. [3]

1.1.2 Přenosová rychlost

Přenosová nebo také modulační rychlost definuje rychlost přenosu znaku za sekundu nebo také počet pulzů za sekundu. Jednotkou přenosové rychlosti je Baud. Používaný anglický termín Baud rate je tedy synonymem přenosové rychlosti [2]

1.1.3 Adresy

Pokud aplikace potřebuje odesílat data přes společný kanál nebo zařízení, je potřeba dbát na správné adresování dat. Například aplikace sdílí stejný prostor kanálu s jinými zařízeními, která odesílají svá vlastní data. [2]

1.2 Typy komunikace

Z hlediska přenosu dat je možné sériové i paralelní sběrnice dělit:

- **Asynchronní a synchronní sběrnice** – Asynchronní sběrnice odesílá data, bez hodinového signálu, zatímco synchronní sběrnice toto synchronizační razítko obsahuje [2]
- **Simplexní komunikace** – umožnuje komunikovat pouze jedním směrem. Data jsou odesílána pouze z vysílače na přijímač. Příkladem této komunikace můžeme být televize nebo rozhlas. [2]
- **Full-duplex komunikace** – umožnuje komunikovat v obou směrech a to současně. [2]
- **Half-duplex komunikace** – spojení rovněž může probíhat v obou směrech, ale ne ve stejný čas. Pokud data jdou jedním směrem, nemůžou jít ve stejný

čas data opačným směrem. Přiklad této komunikace je HTTP, který pokud uživatel pošle žádost na webovou stránku, server žádost zpracuje, po zpracování žádostí odešle odpověď zpět uživateli. [2]

Z hlediska topologie je možné sériové i paralelní sběrnice dělit:

- **Sběrnice Master/Slave** – jedno zařízení je hlavní (tzv. Master) a ostatní jsou vedlejší (Slave). Tento typ komunikace je vždy synchronní, jelikož hlavní zařízení poskytuje čas pro všechny zařízení a odesílaná data, které jsou posílány na sběrnici. [4]
- **Multi-master sběrnice** – rozšíření sběrnice Master/Slave o možnost více hlavních zařízení. Kolize přístupu se řeší naslouchání a systémem priorit. [5]
- **Mezibodová komunikace** – nebo také Point-to-Point (P2P). Jedná se o vzdálené spojení mezi dvěma uzly, které mohou být použity pouze ke komunikaci tam a zpět. Příkladem využiti této komunikace je třeba telefonní hovor, při kterém jeden telefon připojen k druhému a oba uzly mohou odesílat a přijímat zvuk. Jedná se tedy o jednosměrné propojení. Mezi každým spojení existuje vyhrazené spojení a kapacita celého kanálu je výhradně pro přenos paketů mezi odesílatelem a příjemcem.[6]
- **Multi-drop komunikace** – jeden vysílač a několik přijímačů. Hlavní zřízení odesílá informace v kterémkoliv bodě. Ostatní zařízení naslouchají. [2]
- **Více-bodová** (Multi-point) **komunikace** – spojení je komunikační kanál sdílen mezi mnoha zařízeními nebo uzly mezi kterými může docházet ke komunikaci. Mnoho zařízení sdílí jedno spojení ve multi bodovém připojení. V důsledku je možné říct, že všechna zařízení připojená k lince dočasně sdílejí kapacitu kanálu. V této komunikaci je odesílaný paket přijat a zpracován každým zařízením na lince. Příjemce však dokáže vyhodnotit, zda mu paket patři či ne, a to na základě adresního pole v paketu. Například pokud příjemce zjistí, že pro něj není paket určen, zahodí jej. V opačném případě je paket zpracován a zaslána odpověď odesílateli [6]

1.3 Univerzální Asynchronní Sériová Komunikace

Univerzální asynchronní přijímač a vysílač (zkráceně UART – z anglického názvu Universal Asynchronous Reciever-Transmitter) je dedikovaný hardware používaný pro sériovou komunikaci se subsystémem v mikroprocesorech, mikrokontrolerech a osobních počítačích. Pojem Univerzální je použit, protože je možné nakonfigurovat datový formát (počet bitů) a přenosovou rychlost použitou pro komunikaci. Vysílač rozhraní UART, mění paralelní data na sériová. Naopak část UART, která představuje přijímač, zpracovává sériová data a konvertuje je na paralelní. Komunikace může probíhat v simplexním, polo-duplexním nebo plně-duplexním režimu. [2]

1.3.1 Line Control Registr (LCR)

Line Control Registr je vnitřním registrem rozhraní UART a používá se pro nastavení komunikačních parametrů, jakými jsou počet datových bitů, paritní bit a počet stop bitů. Jeho obsah je možné číst i měnit programově. Není tak potřeba ukládat nastavení UART do paměti, ale v případě potřeby lze zjistit z obsahu toho registru. Mimo nastavení parametrů komunikace je registr využit pro kontrolu přístupu k registrům DLL a DLM. Tyto registry jsou mapovány na stejné adresy jako registry RBR, THR a IER, ale jsou přístupné pouze při inicializaci, kdy neexistuje žádná komunikace a nemůže dojít o ovlivnění již fungující komunikace. [2]

1.3.2 Ostatní registry

Mimo LCR registr je definováno dalších 6 registrů rozhraní UART. Jsou to registry sloužící jako vysílací/přijímací paměť RBR/THR, registr pro povolení přerušení IER, registr identifikace zdroje přerušení IIR, speciální kontrolní FIFO registr FCR, kontrolní registr modemového přenosu MCR zajišťující potvrzování doručení data a registr signalizace stavu linky LSR. Pro signalizaci stavu modemu slouží registr MSR. Přenosové rychlost je definována pomocí tzv. dělícího čísla uloženého v registru DLL. [7]

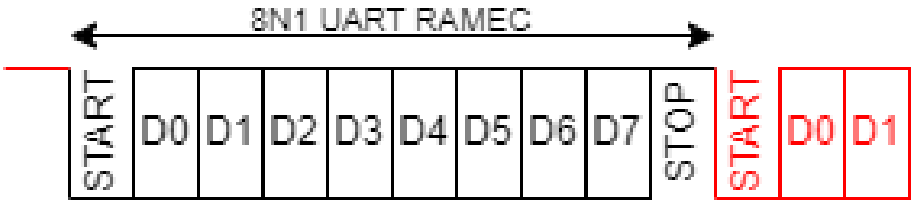
1.3.3 Časování UART

V rámci asynchronního přenosu dat není nutné vysílat hodinový signál. Stačí, když se odesílatel i příjemce dohodnou na časových parametrech komunikace společně s použitím synchronizačních prostředků. Těmi jsou speciální bity START a STOP.

Pokud je linka nečinná, tedy se neposílají žádná data, je stav nastaven na log. 1. Zahájení přenosu datového rámce začíná „START“ bitem. Tento bit slouží pouze k upozornění příjemce, že se vysílač chystá odeslat paket. START bit také slouží pro synchronizaci systémových hodin příjemce s vysílačem. Odchylka těchto dvou hodinových domén by měla být během přenosu zbývajících bitu paketu nižší než 10 %. Jednotlivé bity jsou odeslány teprve poté co je odeslán start bit nejméně významného bitu. Například vysílatel neví, kdy příjemce přečetl čas bitu. Vysílač tedy začne posílat další část slova na další hraně tiku a pokračuje, dokud vysílač nedokončí přenos bitů. Přiklad komunikace můžeme vidět na obr č. 1.

Paritní bit je volitelný, který lze přidat, pokud bude odesláno cele datové slovo. Tento bit může být použit pro detekci chyb na straně přijímače. Jestliže vysílač odešle jeden ukončovací bit, který indikuje konec datových bitu.

Před zahájením vysílání se příjemce a odesílatel musí dohodnout na počtu datových bitů v paketu, jestli bude vysílán paritní bit a počtu stop bitů. K tomuto účelu se používá linkový kontrolní registr (LCR), který na základě nastavení odešle startovací bit. [2]



Obr č. 1 UART komunikace

1.3.4 Přenosová rychlost a časování

V kapitole 1.1.2 byl vysvětlen obecný pojem Baud rate. Protože při asynchronní komunikaci není přenos synchronizován hodinovým signálem, je použití správné přenosové rychlosti na obou stranách (vysílače i přijímače) nezbytné.

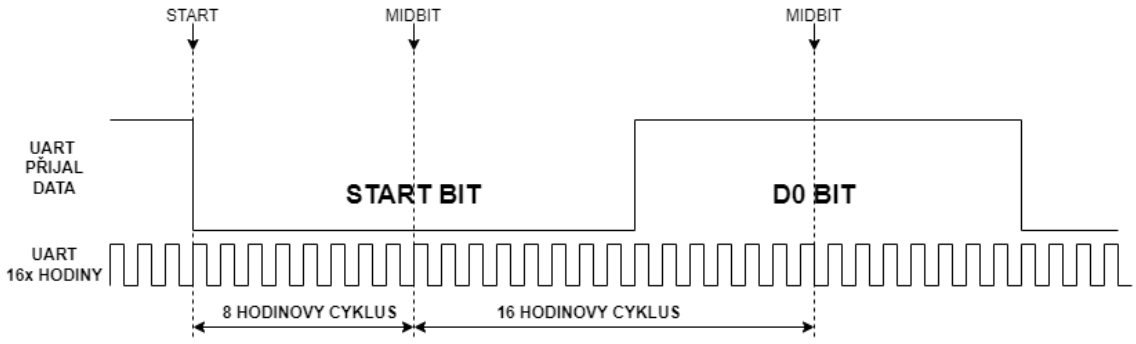
V principu je možné použít libovolnou přenosovou rychlost, nicméně v praxi se používají standardizované přenosové rychlosti odvozené od násobků 1200, tzn. 2400, 4800, 9600, 19200, 38400, atd.

Aby byl přenos úspěšný, musí být dodrženo správné časování v rámci přenosu rámce. Proto se hodinový signál přijímače UART synchronizuje s vysílačem pomocí START bitu. Díky je vznik chyby omezen pouze na jeden rámec. [2]

1.3.5 Synchronizace rámců a vzorkování dat

Vysílač i přijímač znají přenosovou rychlost, ale příjemce neví, kdy bude paket odeslán. Příjemce musí rozpoznat začátek rámce, aby se synchronizoval a určil nejlepší okamžik pro vzorkování dat.

Na Obr č. 2 je zobrazen běžný způsob komunikace používaným přijímačem UART pro synchronizaci s přijímaným rámcem. UART používá 16krát vyšší rychlost hodinového signálu, než je použitá rychlost přenosu dat. Nový rámec je detekován se sestupnou hranou START bitu. Linkový signál přejde z aktivní úrovně (log. 1) do log. 0. Rozhraní UART restartuje své čítače a očekává, že se v 8 cyklech objeví START bit a po dalších 16 cyklech se objeví střed přenášeného bitu. START bit je obvykle vzorkován uprostřed doby bitu, aby se zkontrolovalo, zda je úroveň stále nízká a potvrdilo se, že detekovaná sestupná hrana byla skutečně START bitem a ne šum. [2]

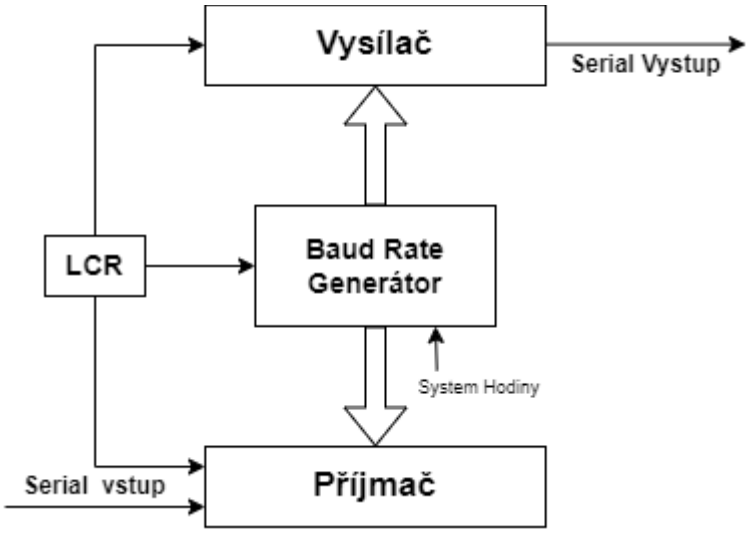


Obr č. 2 Synchronizace dat

1.3.6 Blokové schéma UART

Blokové schéma závisí na zvoleném zařízení a na jeho schopnostech. Nicméně všechny rozhraní UART se skládají ze třech hlavních částí: generátor hodinového signálu, vysílací a příjímací sekce, viz obr č. 3.

Vysílací i přijímací části se skládají z několika registrů, z nichž každý má svou vlastní funkci a řídicí logické obvody. Rozhraní UART obsahuje paměť FIFO a řadič DMA. Tyto dva bloky jsou řízeny programovatelným generátorem přenosové rychlosti a generátorem času. Systémový hodinový signál je pomocí generátoru Baud rate dělen od 1 do 65536. Tak vznikne referenční čas pro interní vysílač a logiku přijímače, který je 16krát nebo 13krát rychlejší než přenosová rychlost. [2]



Obr č. 3 Blokové schéma UART

1.4 Univerzální Synchronní/Asynchronní Sériová komunikace

Univerzální synchronní a asynchronní přijímač-vysílač (USART) je typ sériového rozhraní, které může být naprogramované jak pro synchronní, tak asynchronní komunikaci. USART je také označován jako sériové komunikační rozhraní (SCI).

USART je podobný univerzální asynchronní komunikaci. Oba totiž pracují tak, že přijímají paralelní data z procesoru (CPU), převádí je na sériová data, a odesílají je vybraným způsobem sériovou komunikací. Stejným způsobem se zachází s přijatými sériovými daty. Tyto data jsou převedena na paralelní načtením do přijímacího registru. USART může být řešen jako integrovaný obvod osazený na základní desce nebo integrován na křemíkovém čipu mikrokontroleru. [2]

1.4.1 Operace USART

Operace USART souvisejí s protokolem, který je použit. Protože asynchronní komunikace již byla popsána v kapitole 1.3, bude další popis zaměřen na komunikaci synchronní. V rámci synchronní operace se musí za daný čas doručit celý paket. Pokud se tak nestane, vznikne chyba, která je označena jako „underrun error“ a přenos je přerušen.

Linka není nikdy prázdná. I když fyzická vrstva indikuje, že modem je aktivní, USART bude posílat stálý proud pro zařízení a protokoly. [2]

1.5 Standardy sériové komunikace

V této kapitole jsou popsány standardy sériové komunikace implementované rozhraním UART.

1.5.1 Sběrnice RS-232

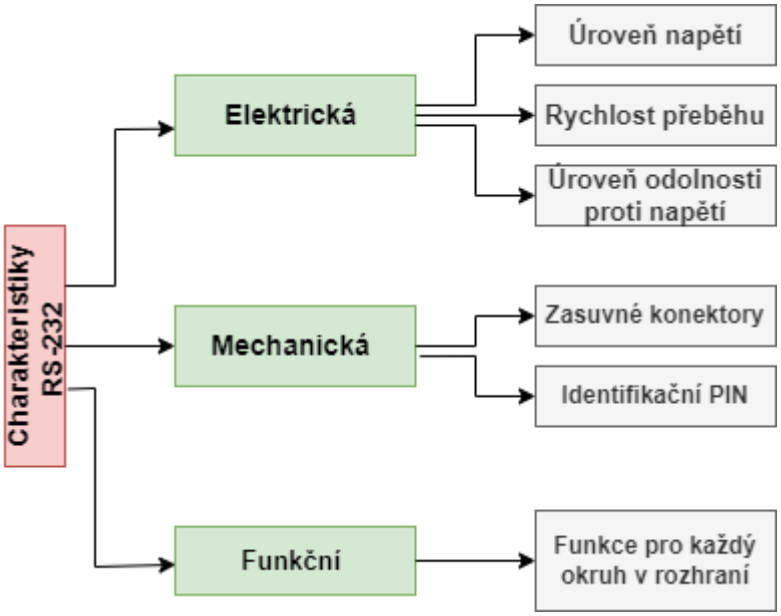
RS-232 je nejrozšířenější a nejpoužívanější standard pro posílání sériových dat a zároveň je to první standard, který byt vytvořen, a to v roce 1962 pro vysílání dat modemem skrz telefonní linky. Byl implementován do počítačů a jejich periferií, např. do tiskáren. Sběrnice RS-232 je velmi citlivá na rušení, proto je maximální vzdálenost mezi zařízeními omezena na 50 m. Tento standard se používá pro připojení point-to-point. Existuje i mnoho dalších verzí, které byly vyvinuty s RS-232.

- **EIA/TIA – 232:** varianta odkazuje na RS-232. Je založena na jménech společností, které sponzorovali organizace Electronic Industries Alliance (EIA) a Telecommunications Industry Alliance
- **RS-232-C:** varianta byla aktualizována v roce 1960, aby zahrnovala mnoho specifických charakteristik zařízení
- **V24**: Standard vyvinula společnost se jménem Mezinárodní telekomunikační Unie (ITU) Consultative Commiter for International Telephone and Telegraph (CCITT). Tento standard je kompatibilní s RS-232 a jeho cílem bylo umožnit výrobcům vyhovět globálním standardům a umožnit produkci zařízení, které by fungovaly ve všech zemích po celém světě. [2]

RS-232 je „úplný“ standard. Jeho cílem je zajistit dokonalou kompatibilitu mezi hostitelským a klientským systémem, tím že se specifikuje

- úrovně napětí logických stavů signálu
- konfiguraci zapojení pinů v konektorech
- řídicí signály a informace mezi hostitelským a klientským zařízením

Na obr č. 4 je zobrazena standardní specifikace RS-232 s jejich rozdělením. [2]



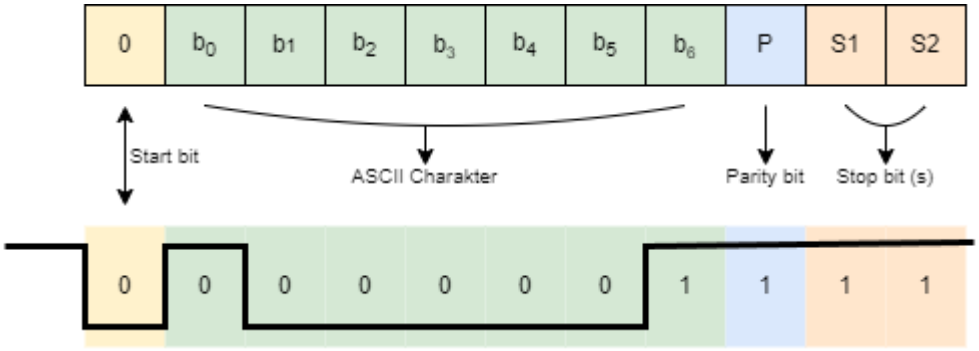
Obr č. 4 Specifikace RS-232

Limity sběrnice:

- Tím, že se jedná o signalizaci se společnou zemí (single ended) je méně odolná proti šumu což omezuje použitelnou přenosovou vzdálenost.
- Není definována možnost připojení více klientských zařízení (multi-drop)
- Sběrnice RS-232 nedefinuje možnost využití řídicích signálů DTE přímo k DTE nebo DCE k DCE. Pokud je toto řízení výhodné využít je nutné použít modemový kabel, který není definován standardem. [2]
- 25 pinový konektor Dsub definovaný ve standardu je v dnešní době zbytečně velký [2]

Časování dat

Tím, že je sběrnice RS-232 asynchronní komunikační protokol je snadno implementovatelná pomocí rozhraní UART. V rámci protokolu je použit 1 START bit, 8 datových bitů, volitelně paritní bit a 1, 1,5 nebo 2 STOP bity. Příklad odeslaného rámce je zobrazen na obr č. 5. [2]



Obr č. 5 Přenos rámce na sběrnici RS-232

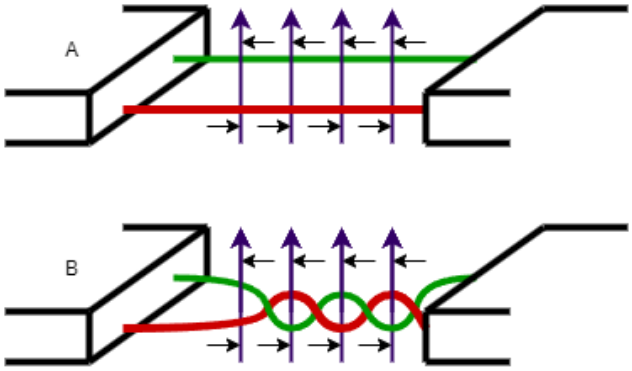
Sériové rozhraní (V24)

Rozhraní RS-232 bylo historicky využíváno v IBM kompatibilních počítačích pro připojení myši, tiskárny nebo průmyslového vybavení. I když se jedná o relativně nízko-rychlostní komunikační standard, pro tyto periferie to nepředstavovalo problém. Jako reakce na toto omezení bylo rozšíření standardu známé jako V24 definované společností EIA.

Rozhraní RS-232/V24 je určeno hlavně pro provoz na menší vzdálenosti, maximální vzdálenost může být patnáct metrů. Nepředpokládá se, že by modem nebo obdobné zařízení byl umístěn dále od počítače. I v tomto případě je přenosová rychlost omezena. Maximální přenosová rychlost pro verzi RS-232C je 19 200 Bd.

1.5.2 Sběrnice RS-422

Standard RS-422 je známý jako TIA/EIA RS-422-A nebo X.27. Historicky byl používán na počítačích Apple Macintosh. Standard definovala společnost Electronic Industries Alliance (EIA). Na rozdíl od RS-232 využívá sběrnice RS-422 diferenční pár vodičů pro přenos informace. Tím je značně potlačen vliv rušení indukovaného v zemním vodiči. Díky použití kroucené dvojlinky (viz obr č. 6) je eliminován vliv indukovaného napětí vzniklého parazitním magnetickým polem.



Obr č. 6 indukce napětí vlivem magnetického pole a) přímá linka b) kroucená linka

Diferenční signalizace umožňuje přenášet data rychlostí až 10 Mbit/s na vzdálenost 12 m (40 stop) nebo komunikovat po kabelech dlouhých až 1500 m. Standard definuje

pouze úroveň signálu. Mechanické vlastnosti jako např. použité konektory nebo jejich pinové zapojení jsou součásti jiných standardů. Maximální délka použitého kabelu také není specifikována.

Mezi parametry, které se musí vzít do úvahu limitující maximální délku kabelu, jsou datová rychlost, vyvážení a zakončení kabelu a individuální instalace kabelu. Proto se uvádí maximální délka právě 1500 metrů, která je ale platná pouze pro aplikace, které mohou tolerovat větší zkreslení času a amplitudy.

Kabeláž je vyrobena ze dvou sad krocených párů, přičemž každý pár bývá stíněný, a společným zemnícím vodičem. Dvou-párový kabel může být praktický pro mnoho aplikací, specifikace definuje pouze jednu signálovou cestu a nepřiřazuje žádné funkce. [8]

Limitace RS-422

Standard je definován pro jedeno zařízení kontrolující komunikace a jedno kontrolované zařízení. Standard může spolupracovat s rozhraními, které jsou navrženy pro MLT-STD-188114 B, ale nejsou úplně stejné. Jmenovité hodnoty napětí signálů se pohybují od 0 V do 5 V, zařízení MLT-STD-188114 B používají signál symetrický kolem 0 V. Tolerance pro napětí je ve společném režimu v obou zařízeních, to umožnuje vzájemnou spolupráci. Každá kompletní kabelová sestava s konektory by měla být označena štítkem se specifikací, která definuje funkce signálu a mechanické rozložení konektoru. [8]

1.5.3 Sběrnice RS-423

Standard je jedním z méně známých. Jeho starší verze RS-232 je široce známá, protože sériové porty s tímto rozhraním jsou přítomny na téměř všech počítačových systémech. Standard využívá jednostrannou signalizaci a nachází se někde mezi standardy rozšiřující sběrnici RS-232 o možnost použití delších kabelů a vyšší podporované přenosové rychlosti.

Sběrnice RS-423 není v současné době používána. Na konci osmdesátých let minulého století byla nicméně široce používaná, kvůli své kompatibilitě se standardem RS-232. Společnost Hewlett Packard dodávala své počítače s rozhraním schopným komunikovat sběrnicemi RS-232 i RS-423. Společnost Digital Equiment Corporation používala úroveň signálu pro standard RS-423 na svém standardu DEC Connect MMJ. [8]

1.5.4 Sběrnice RS-449

Úplný název standardu je „*EIA-449 general purpose 37-Position and 9-position interface for data terminal Equipment and Data circuit-Terminating Equipment Employing Serial Binary Data Interchannge*“. [2]

Standard RS-449 byl vyvinutý za účelem rozšíření standardu RS-232. Byl zaměřen na poskytování sériového přenosu dat rychlostí až 2 Mbps a zachování kompatibility s RS-232.

Sběrnice nebyla nikdy použita na osobních počítačích, ovšem našla uplatnění na některých síťových komunikačních zařízeních. Na standard RS-449 navazují další standardy jako EIA-449, TIA-449 a ISO 4902. Ty definují funkce a mechanické vlastnosti rozhraní mezi datovým koncovým zařízením, například počítačem, a datovým zařízením jako je třeba modem nebo terminálový server. Cílem bylo nahradit sběrnici RS-232 C, prostředkem nabízející mnohem vyšší výkon a delší kabely. Nicméně se ukázalo, že jde o velmi nepraktický systém, který vyžaduje konektory DC-37 a DE-9. Proto byl nahrazen standardem RS-350, který využíval konektor DB-25. [2]

1.5.5 Sběrnice RS-485

Tento standard je nejuniverzálnější ve standardní řadě nadefinované EIA. Proto je v dnešní době široce používán v aplikacích pro sběr dat, kde spolu komunikuje více uzlů. Přestože nikdy nebyl zamýšlen pro domácí použití, našlo se mnoho aplikaci, kde bylo vyžadováno vzdálené získávání dat. Například pro komunikaci v řídicích aplikacích, kde spolu komunikuje více uzlu. [2] Sběrnice RS-485 podporuje až 32 zařízení připojených do jednoho uzlu a definuje elektrické charakteristiky nezbytné pro zajištění adekvátního napětí signálu při maximální zátěži.

Standard je nadmnožinou RS-422 a proto podporuje všechna zařízení RS-422, která mohou být ovládána pomocí RS-485. Maximální přenosová rychlost je 10 Mb/s na vzdálenost až 50 stop. Je zde omezeni, hardware nelze použít pro sériovou komunikaci s kabelem o délce 4000 stop a nižší rychlostí 100 kb/s. Standard používá diferenční signály, což má za následek delší vzdálenost a vyšší přenosovou rychlost. [8]

2. OPERAČNÍ SYSTÉM

Operační systém (OS) je v současné době základní softwarové vybavení každého počítače. Jeho hlavním úkolem je abstrakce hardwarových prostředků počítače. Díky tomu je umožněna pseudo-paralelizace běhu programů na jednom procesoru (takzvaný multitasking). Využití operačního systému není doménou pouze osobních počítačů, ale je možné jej nalézt i v mnoha jiných zařízeních – od mobilních telefonů přes fotoaparáty, kamery, videoherní konzole až po inteligentní televize. Operační systém se skládá z jádra (angl. kernel) a specializovaných modulů. Jádro OS se zavádí do paměti bezprostředně po startu počítače v chráněném režimu procesoru a má za úkol zejména správu paměti a přidělování systémových prostředků běžícím procesům. Běžný operační systém je z hlediska přidělování výpočetního času jednotlivým procesům obvykle zcela nedeterministický. Při plánování procesorového času často vychází z požadavku spravedlnosti nebo optimalizace vytížení procesoru. Není tak možné zaručit, že proces dostane přidělen procesorový čas ve chvíli, kdy by jej potřeboval [9]. Toto naopak zaručují operační systémy reálného času.

2.1 Operační systémy a podpora multitaskingu

První IBM kompatibilní počítače (IBM 5150) byly uživatelům dodávány s operačním systémem CP/M, který podporoval spuštění pouze jedné úlohy. Operační systémy tohoto typu jsou nazývány jedno-úlohové (angl. single-task). Jejich velkou výhodou je jednoduchost a determinovanost. S postupným rozšířením osobních počítačů přestaly jedno-úlohové OS vyhovovat a objevily se systémy umožňující spustit několik programů současně. Pro uživatele se tyto programy jeví, jako by byly zpracovávány souběžně (paralelně). Toto ovšem platí pouze, pokud je počet dostupných jader procesoru vyšší než počet běžících procesů, což je v praxi extrémně raritní stav. V opačném případě musí být mezi jednotlivými programy procesor sdílen tak, aby bylo dosaženo zdání souběžného zpracování tzv. pseudo-paralelismus. Multitaskingový systém lze rozdělit na preemptivní, kdy systém přiděluje procesorový čas procesu na definovanou dobu a po uplynutí této doby je procesu procesor odebrán a kooperativní (nepreemptivní), kdy se operační systém spoléhá na vlastní program, že předá systémové prostředky dalšímu programu podle předem definovaného způsobu.

2.1.1 Kooperativní multitasking

V případě kooperativního (nepreemptivního) multitaskingu běží pouze jeden proces na popředí. Pokud běžící proces dospěje do stavu, kdy už procesor nepotřebuje (například čeká na vstup z klávesnice nebo na nějakou jinou událost), může předat přidělený procesor některému procesu na pozadí – a to buď přímo, nebo prostřednictvím OS. Výhodou kooperativního multitaskingu je jeho jednoduchost

a přehlednost. Nižší režie OS a lepší využití prostředků systému - například paměť a čas procesoru. Mezi nevýhody patří problémy s bezpečností a stabilitou. Při chybně běžícím procesu dochází k selhání celého systému. [10]

2.1.2 Preemptivní multitasking

U preemptivního multitaskingu OS pravidelně přepíná mezi procesy. Procesy vzájemně nespolupracují a po znovu přidělení procesoru ani nejsou informovány o tom, že jeho činnost nebyla souvislá. Výhodou preemptivního multitaskingu je možnost spuštění více procesů souběžně bez nutnosti aktivní správy přístupu k prostředkům počítače. Tuto správu zajišťuje samotný OS, díky čemuž jsou lépe využity prostředky systému (například paměť, čas procesoru atd.). Mezi nevýhody patří vyšší režie operačního systému a s tím spojené vyšší nároky na hardware. [10]

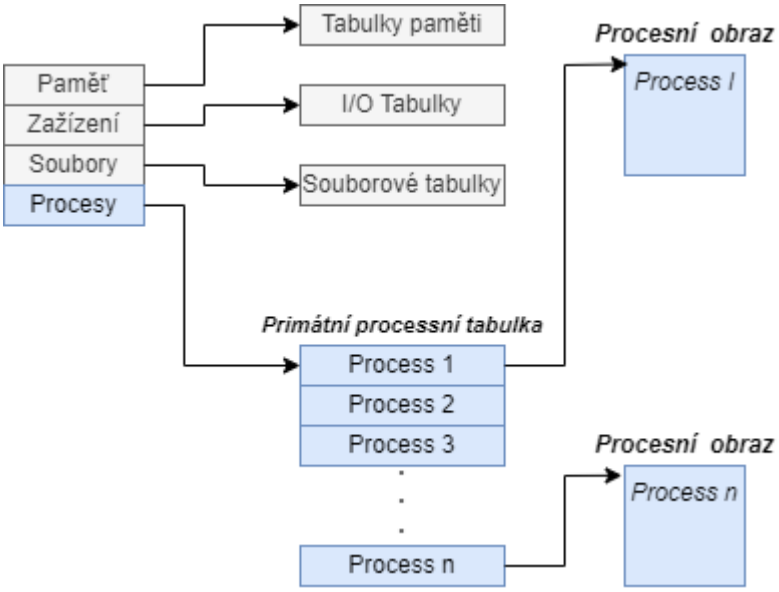
2.2 Jádro operačního systému

Hlavní funkcí jádra operačního systému je zajišťovat běžícím programům (procesům) systémové prostředky tzn. procesorový čas, paměť a V/V zařízení. Zároveň musí operační systém udržovat informace o aktivních procesech a jejich stavu.

Pro tyto účely vytváří OS speciální tabulky:

- Paměti
- V/V prostředků
- Souborů
- Procesů (process table)

Pro bližší představu jsou tyto tabulky rozkresleny na následujícím Obr č. 7



Obr č. 7 Části jádra operačního systému

2.2.1 Zavádění jádra OS

Aby mohl mít operační systém plnou kontrolu nad hardwarem, je zavedeno jádro OS do paměti počítače jako první v tzv. chráněném režimu procesoru (tento režim je v případě moderních procesorů plně podporován). V tomto režimu má OS přístup ke všem instrukcím procesoru a jádro je tak zavedeno do paměti od adresy 0. Má tím pod kontrolou adresní prostor vektorů přerušení procesoru. Ve chvíli, kdy je jádro zavedeno do paměti, dochází k načtení dalších částí OS, jako jsou databáze konfiguračních registrů, výběr hardwarového profilu určujícího načtení ovladačů zařízení (tzv. drivery) a následné načtení ovladačů. Posledním krokem může být autentizace uživatele v případě víceuživatelského systému.

2.3 Tabulka procesů

Tabulka procesů (Primary Process Table) je tvořena záznamy obsahujícími kontrolní bloky procesu (anglicky Process Control Block – PCB). Kontrolní blok procesu je speciální datová struktura obsahující informace o stavu procesu, obsahu vnitřních registrů procesoru, informace pro správu paměti a přístupu k V/V zařízením. Obsah této datové struktury se dle operačního systému mění. Typická struktura PCB obsahuje následující informace:

- ukazatel na následující proces,
- stav procesu (běžící, čekající, připravený),
- priorita procesu,
- časové kvantum,
- využití předchozích časových kvant,
- identifikace procesu,
- aktuální stav počítadla programu (PC),
- obsah vnitřních registrů CPU,
- odkaz na záznamy v tabulce správy paměti,
- informace pro správu souborů,
- účtovací informace

2.4 Správa paměti

Modul správy paměti umožňuje OS přidělovat paměťový prostor běžícím procesům. Dále pak zajišťuje

- přidělování a uvolňování (ochranu) paměti,
- adresace fyzické paměti,
- segmentace paměti,
- defragmentace paměti,
- monitorování obsazenosti adresního prostoru.

Fyzická (skutečná) adresa v paměti se získá sečtením obsahu segment registru

a logické adresy (tzn. adresy použité v programu). Segment registr nastavuje OS a pro program je nepřístupný. Díky tomu adresový prostor každého programu začíná adresou 0 a odpadají tak problémy s relokací.

2.5 Plánování procesů

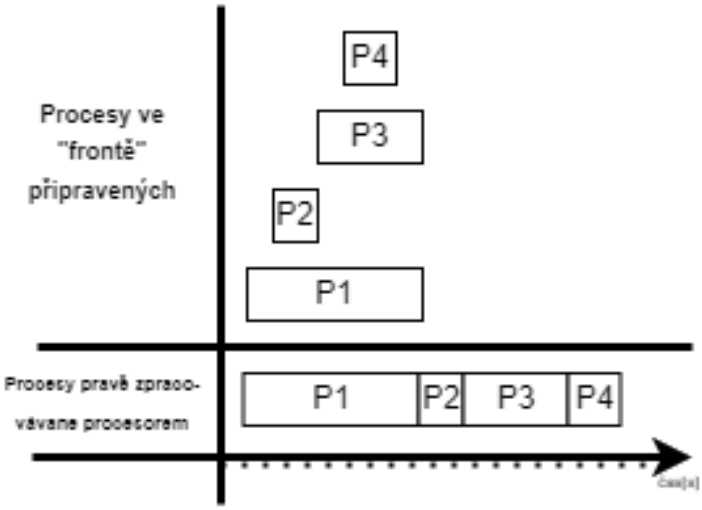
Jednou z nejdůležitějších služeb, kterou jádro OS zajišťuje, je plánování procesů. Plánování je založeno na definici priorit jednotlivým procesům. Procesy jsou pak spouštěny podle časového plánu, a to takovým způsobem, aby byla zohledněna různá kritéria pro spouštění procesů (a tím docíleno optimálního vytížení procesoru). [11]
Mezi kritéria, ke kterým se přihlíží, patří

- spravedlnost (každý proces dostane spravedlivý díl času na zpracování),
- efektivita (maximální vytíženost procesoru a popřípadě i pro jiné časti systému),
- minimální čas odezvy (pro interaktivnost uživatele a pro zpracování každého procesu),
- průchodnost (maximalizace množství úloh, které se mohou zpracovat za jednotku času). [12]

Z pohledu aplikace (procesu) jde o krátkodobé plánování operačního systému.

2.5.1 Plánování typu FIFO

Nejjednodušším způsobem, jak plánovat zpracování procesů, je nepreemptivně pomocí fronty. Tento přístup je sice jednoduchý, ale rozhodně není optimální z hlediska využitelnosti systémových prostředků. V případě, že je požadováno zpracování procesu, který si vyžádá procesor na signifikantně dlouhou dobu, dojde po tuto dobu k „zamrznutí“ operačního systému.

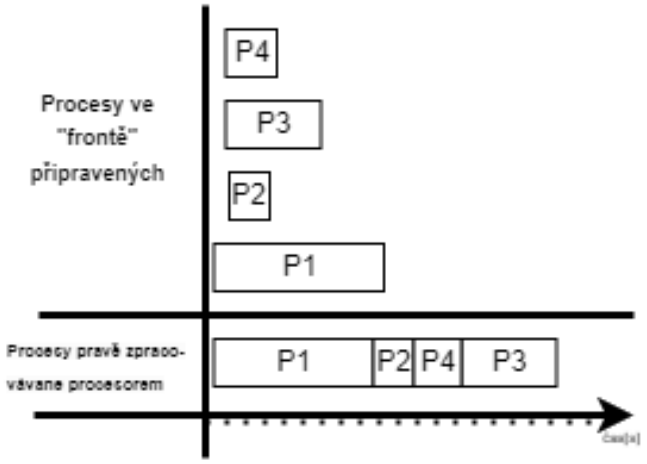


Obr č. 8 Plánování procesů typu FIFO

2.5.2 Plánování s preferencí nejkratší úlohy

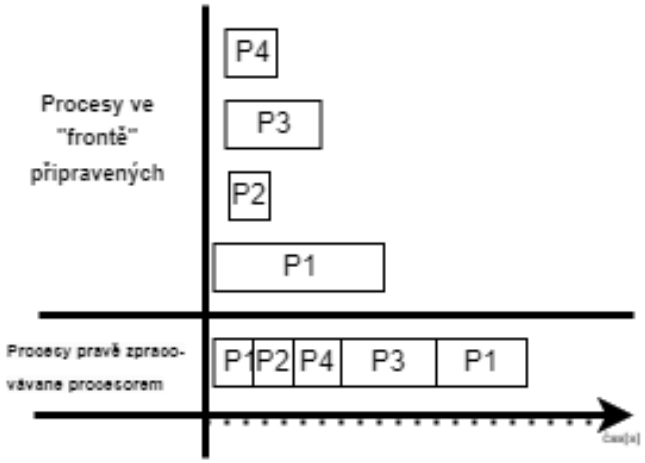
Plánování s preferencí nejkratší úlohy (anglicky Shortest-Job-First) může využívat

kooperativní i preemptivní způsob plánování. V tomto typu plánování se připravené procesy řadí od nejkratšího po nejdelší. Protože ovšem v případě kooperativního plánování nejsou v čase nula známy požadavky všech procesů, může se stát, že bude upřednostněn proces s delší dobou provádění.



Obr č. 9 Plánování procesů s preferencí nejkratší úlohy bez přerušení běhu

Lepší situace nastává při preemptivním plánování procesů touto metodou. Pokud je proces s kratší dobou vykonání zařazen do fronty, delší proces, který je aktuálně zpracováván, je přerušen operačním systémem a systémový čas dostane přidělen tento rychlejší proces. Proto bývá tento typ plánování označován jako s preferencí nejkratšího zbývajícího procesu (angl. Shortest Remaining Time First).



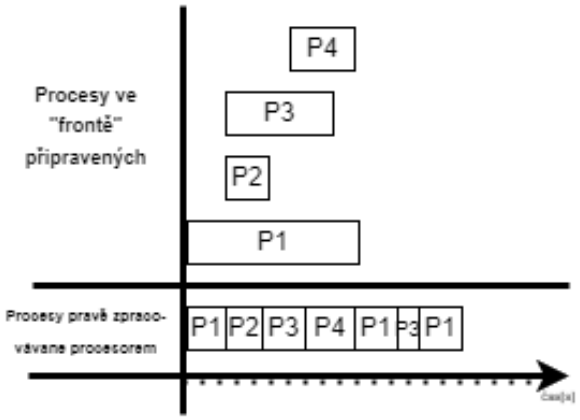
Obr č. 10 Plánování procesů s preferencí nejkratší úlohy s přerušením běhu

Tento typ plánování dosahuje nejnižší průměrné doby čekání procesu, nicméně dochází zde k diskriminaci dlouhých procesů a s tím spojenému problému stárnutí procesu (anglicky aging).

2.5.3 Planovač Round-Robin

Kruhový plánovač Round-Robin (zkráceně RR) je jedním z nejstarších, nejjednodušších a nejrychlejších algoritmů. Poskytuje každému připravenému procesu stejně dlouhé

časové kvantum procesoru a patří tak do kategorie preemptivních plánovačů. Všechny procesy v rámci plánovače RR mají stejnou prioritu. Plánovač je možné si představit jako kruhový zásobník, do kterého se procesy řadí postupně podle toho, jak v čase přicházejí. Ve chvíli, kdy je proces dokončen je místo posunu na konec kruhu z fronty uvolněn. [11]



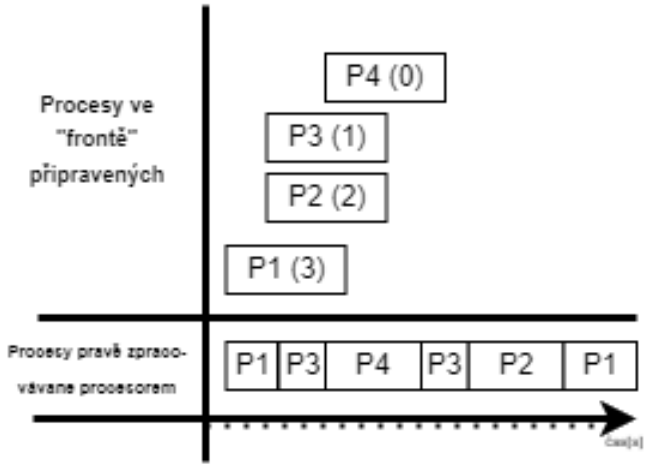
Obr č. 11 Zpracování procesů s využitím plánovače Round-Robin

Lepší představu o činnosti plánovače Round-Robin poskytne Obr č. 11. Každý proces dostane přidělen procesor nejdéle za (n-1).t jednotek času, kde *n* je počet procesů ve frontě a *t* je délka časového kvanta. Existují zde dvě extrémní konfigurace – příliš krátké časové kvantum a s ním spojená vysoká režie OS a neúměrně dlouhé časové kvantum, díky kterému se Round-Robin mění v plánovač typu FIFO. Běžně se časové kvantum volí v rozsahu 10 až 100 ms. Díky tomu, že proces má jistotu přiděleného časového kvanta procesoru, splňuje tento typ plánování požadavky na systém reálného času.

2.5.4 Prioritní planovač

V případě prioritního plánování jsou procesy před zařazením do fronty na zpracování označeny svojí prioritou tzv. prioritním číslem. Je zažitým axiomem, že procesu s nejvyšší prioritou je přiřazeno prioritní číslo nula a procesům s nižší prioritou pak čísla vyšší. Záleží přitom na konkrétním operačním systému, jak vysoké prioritní číslo podporuje. Stejně jako výše prioritních čísel se podle OS liší i strategie jejich určení.

Při souběhu procesů se stejnými prioritními čísly se postupuje dle pořadí zařazení procesů do fronty. [9]



Obr č. 12 Příklad prioritního plánování

I tento druh plánovače lze využít pro kooperativní i preemptivní multitasking. Pokud má nový proces, který dorazil do fronty připravenosti, vyšší prioritu než aktuálně běžící proces, je v případě kooperativního prioritního plánovacího algoritmu, zařazen do čela fronty připravenosti, což znamená, že po provedení aktuálního procesu bude zpracován. V případě preemptivního plánování dojde, při příchodu nového procesu s vyšší prioritou, k odebrání CPU běžícímu procesu. To znamená, že zpracování aktuálního procesu se zastaví a příchozí nový proces s vyšší prioritou získá CPU ke svému provedení. Nevýhodou tohoto typu plánování je fakt, že pokud do fronty připravenosti přicházejí stále nové procesy s vyšší prioritou, mohou procesy čekající ve frontě uváznout. Příkladem může být zjištění inženýrů z MIT, kteří v roce 1973, kdy byl na univerzitě ukončen provoz sálového počítače IBM 7904, objevili v jeho paměti proces s nízkou prioritou, který byl spuštěn v roce 1967 a to té doby nebyl zpracován. 20 Z těchto důvodů byl přidán do prioritního plánovače mechanismus stárnutí procesu (aging), který mění prioritu procesu v závislosti na délce času, po který proces čeká na své zpracování.

3. OPERAČNÍ SYSTÉM REÁLNÉHO ČASU

V případě osobního počítače v domácnostech či firmách je přípustné, aby si uživatel na dokončení některých operací musel počkat. Nicméně existují případy, kde by čekání na dokončení jedné operace negativně ovlivnilo spolehlivost provedení jiné úlohy. Dobrým příkladem mohou být průmyslové aplikace, kde zpoždění čtení informace ze senzorů může mít za následek nepřesnost výroby a s tím spojenou vyšší zmetkovitost, zpomalení výrobního procesu nebo dokonce ohrožení zdraví či života obsluhy.

„Například od počítače, který řídí výtah, vyžadujeme, aby se signál od čidla, oznamující, že dosáhl požadovaného patra, zpracoval ihned a výtah se zastavil (tj. není například možné, aby „zaneprázdněnost“ počítače zobrazováním čísla patra na displeji způsobila, že výtah v daném patře nestihne zastavit a zastaví až v následujícím).“ [15]

V těchto případech je řešením použít operační systém reálného času (angl. Real-Time Operating System, zkráceně RTOS). RTOS je typ operačního systému, který umožňuje procesům reagovat na události bezprostředně po jejich výskytu. Tím je vytvořena iluze výhradnosti běhu a proces tak reaguje v reálném čase. Tohoto chování je dosaženo deterministickým chováním systému v čase. To znamená, že služby operačního systému spotřebují známé množství času a je tak možné čas služeb jádra OS vyjádřit pomocí matematických vztahů. Tyto vztahy musí být přesně dány a nesmí obsahovat žádné náhodné složky.[9], [16]

I přes to, že běžné OS nechávají uživatele čekat na dokončení procesu, jsou jejich plánovače optimalizovány k dosažení nejkratšího času čekání a nejvyšší průchodnosti procesů. Naopak RTOS tím, že garantuje splnění odlišných požadavků, nemusí být nutně pro desktopové počítače výhodný a stejně výkonný jako běžné OS. Už jen proto, že zpravidla spotřebovává větší část procesorového času. To je důvodem, proč je používán tam, kde je na jednu stranu k dispozici dostatek výkonu procesoru, ale současně se od zařízení neočekává vysoký výpočetní výkon [17]. Příkladem mohou být vestavěné systémy, robotické systémy, průmyslová automatizace, elektronické měřicí přístroje nebo telekomunikační zařízení

Mimo jednotný požadavek na deterministické chování existují i další požadavky na RTOS, které se liší zpravidla podle oblasti nasazení systému. Jako příklad mohou sloužit dvě sady definic požadavků na RTOS. První sada byla definována výrobcem mikrokontrolerů a předpokládá jejich použití především v průmyslových aplikacích. [15] Druhá sada je definována americkou vesmírnou agenturou (NASA), která v porovnání s průmyslem klade důraz na vývoj bezpečného softwaru a s tím spojenou verifikaci.

Podle společnosti Texas Instruments [18] by měl RTOS splňovat následující cíle:

- Nízká latence (limitně nulová).
- Determinovanost – pro úspěšné splnění požadavku procesu musí operační
  - systém znát dobu obsluhy procesu i dobu vlastní režie. Není přípustné nepředpokládané chování.
  - Strukturovanost software – díky strukturovanosti programu je snadné přidat další části do aplikace.
  - Škálovatelnost – RTOS musí být schopný rozdělit procesy dle využití systémových prostředků.

Podle NASA [15] by měl splňovat následující požadavky:

- RTOS je preemptivní multitaskový OS pro real-time aplikace,
- musí podporovat plánování tak, aby byla garantována doba odezvy procesu,
- musí podporovat prioritu procesů (staticky i dynamicky) a mechanismus pro dědění priorit při odštěpení vlákna procesu,
- RTOS musí podporovat mechanismus pro předvídatelnou synchronizaci procesů,
- spravovat hardwarové a softwarové prostředky (systémové zdroje),
- být plně deterministický – tzn. garantovat dokončení úlohy před definovaným časovým milníkem (systém je deterministický tehdy, pokud pro každý možný **stav** procesu **a hodnoty vstupních proměnných** je možné určit **následující stav** procesu a **hodnoty výstupů**),
- Časová omezení chování OS by měla být známa a minimalizována
  - Latence přerušení (čas od požadavku na obsluhu přerušení ke spuštění úlohy navázané na toto přerušení)
  - Minimalizace doby přepnutí mezi procesy (přepínání kontextu).

V závislosti na tom, jak závažná vznikne chyba při nesplnění časového požadavku procesu, se RTOS dělí na typy Hard, Firm a Soft. [19], [20]

3.1 Hard RTOS

Hard RTOS je plně deterministický a poskytuje časově zaručené služby. V praxi to znamená, že pokud proces zpracovává data ze senzoru a na jejich základě upravuje parametry regulačního algoritmu každých 100 ms, je operačním systémem zaručeno, že tyto parametry (výstup procesu) budou k dispozici právě každých 100 30 ms. Není tak přípustné, aby došlo ke zpoždění nebo předstihu vůči této periodě. Zmeškání tohoto časového okna je považováno za selhání systému. Dokonce přesnost termínu dokončení procesu je pro Hard RTOS důležitější než korektní funkce vlastního jádra systému.

Příkladem využití Hard RTOS je protiblokovací brzdící systém (ABS). Časové omezení je zde nastavené zpravidla tak, aby se brzdný tlak vyvinul a uvolnil 12

- 16krát za sekundu. Toto přerušované brždění má zajistit, aby nedošlo k zablokování kol během brždění, resp. napomoci tomu, aby se zablokované kolo opětovně začalo točit a tím i brzdit. Katastrofálním důsledkem nedodržení reakce tohoto systému v požadovaném čase je možná ztráta lidských životů. [19]

3.2 Firm RTOS

Firm nebo také pevný systém reálného času toleruje příležitostný výskyt nedodržení termínu pro zpracování úlohy. Několik zmeškaných termínů tak nepovede k úplnému selhání, nicméně dlouhodobé nedodržení požadovaných termínů může vést k úplnému a katastrofickému selhání systému. [16]

Příkladem, kde je výhodné použít Firm RTOS, je videokonferenční hovor. Pokud během hovoru vypadne několik snímků, zpravidla pocítíme pouze horší kvalitu přenosu, ale jednání může probíhat. Nicméně pokud je ztráta snímků rozsáhlejší, není možné v jednání pokračovat a dochází ke katastrofálnímu selhání služby. [20]

3.3 Soft RTOS

Soft RTOS uděluje prioritu real-time procesům a ty jsou tak upřednostněny před ostatními. Snížení výkonu je tolerováno při nedodržení několika časových oken (doby, kdy je očekáván výsledek) se sníženou kvalitou služeb, ale bez kritických následků.

I když nedodržení termínu dokončení úlohy nezpůsobí katastrofické následky, užitečnost výsledku po uplynutí časového intervalu může být nulová. Dochází tedy ke zhoršení kvality a spolehlivosti RTOS. Příkladem zařízení využívající Soft RTOS je záznamové zařízení trasy vozidla pomocí navigačního systému (GPS). Pokud dojde z nějakého důvodu ke zpoždění při výpočtu pozice, následkem je pouze vynechání jednoho průjezdního bodu.[19]

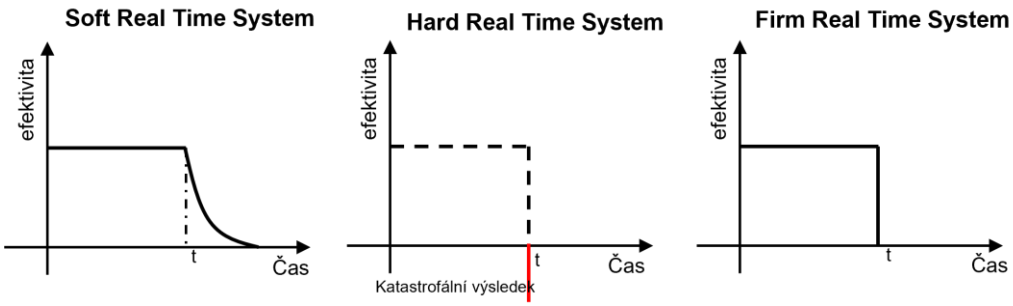
3.4 Jádro RTOS

Jak už bylo uvedeno výše, jádro OS je ta jeho část, která poskytuje základní služby aplikacím běžícím na procesoru. Jádro RTOS nám vytváří abstraktní úroveň, která je skryta aplikačnímu software. Může obsahovat např. sady procesorů, detaily hardware procesoru v době běhu úlohy. Tímto způsobem nám vytváří interface mezi jednotlivými úlohami a hardware.

Jádro rovněž zahrnuje nástroje pro podporu reálného času. Mezi tyto nástroje patří rychlý multitasking, podpora přerušení, preemptivní a Round Robin plánovač. [19] Každé jádro RTOS nám rovněž nabízí mnoho dalších komponent operačního systému, které ovšem patří mezi vyšší úroveň služeb – například řízení souborů, komunikace v síti, řízení sítě atd. Jádro je navrženo tak, aby potřebovalo minimální režii systému a umožňovalo rychlou a deterministickou odezvu na jakoukoliv externí událost.

Díky výkonnému mechanismu mezi procesové komunikace je umožněna nezávislá koordinace úloh. [9] V základu nám jádro RTOS poskytuje služby, které jsou nezbytné pro funkčnost aplikačního softwaru. Mezi tyto služby patří následujících pět hlavních skupin: řízení procesu, mezi-procesová komunikace a synchronizace, řízení jednotek I/O, dynamická alokace paměti a časovače, které zde budou popsány. [9]

Většina RTOS nabízí další volitelné služby, které řadíme na vyšší úroveň. Mezi ně patří systém řízení souborů, komunikace a řízení sítě, řízení databáze a jiné. Mnoho těchto přídavných služeb je obvykle mnohem větších a komplexnějších než samotné jádro RTOS. Všechny se opírají o jeho přítomnost a využívají jeho základní služby. Každá z těchto volitelných komponent je zahrnuta do RTOS pouze tehdy, pokud je splněna podmínka minimální spotřeby paměti při využití daného RTOS.



Obr č. 13 Srovnání reakcí různých úrovní RTOS

3.4.1 Řízení procesu

Řízení procesu patří mezi základní služby jádra. Tato služba umožňuje vývojářům vytvořit aplikační software, ve kterém existuje několik oddělených částí programu. Každá z těchto částí má svůj vlastní cíl a v řadě případů i časové omezení, které musí být splněno (v reálném čase). Jednotlivé části se nazývají úlohy a po spuštění se mění na proces, který musí být splněn do požadovaného časového omezení. Služby z této skupiny umožňují přiřadit jednotlivým procesům příslušnou prioritu. Více o plánování procesů a jejich principech v kapitole 2.5. [9]

3.4.2 Komunikace mezi procesy a jejich synchronizace

Komunikace mezi procesy zajišťuje předávání zpráv mezi jednotlivými procesy a chrání komunikaci před poškozením nebo ztrátou informace. Zároveň umožňuje koordinaci procesů, aby vzájemně spolupracovaly. Bez použití této služby u RTOS mohou procesy komunikovat s nesprávnými informacemi nebo v jiném případě se vzájemně rušit mezi sebou. [9] K rušení procesů mezi sebou dochází, pokud proces neoprávněně přistupuje do paměti jiného procesu. Vzájemné rušení procesů znamená, že při špatné synchronizaci může dojít k tomu, že jeden proces dostane mylnou informaci v porovnání se skutečností.

3.4.3 Dynamická alokace paměti

Existují RTOS, které poskytují službu dynamického alokování paměti. Ovšem pro některé kritické úlohy je vhodnější a bezpečnější využívat pouze statické přidělování paměti. [1]

Statickým přidělováním se rozumí, že paměť je pevně rozdělena na sekce při spuštění RTOS. S vytvořením požadavku poskytnutí paměti se vybere blok paměti, který je dostatečně velký pro jeho splnění. [21]

Dynamické přidělování paměti umožňuje jednotlivým procesům vypůjčit si dočasně oblast pamětí RAM pro daný software. Vytvořené bloky paměti bývají sdíleny mezi jednotlivými procesy a umožňují tedy rychlou komunikaci i s velkým množstvím dat. [9]

3.4.4 Řízení jednotek I/O

Většina jader RTOS nabízí službu pro řízení jednotek I/O. Jednotky se označují jako FLC (Field Logic Controllers). Velmi volně přeloženo se jedná o distribuované logické řídící jednotky, které mají za úkol realizaci jednoduchých logických programovatelných aplikací. [22]

Jestliže je služba dostupná, umožňuje pomocí ovladačů standardizovaný přístup k mnoha hardwarovým jednotkám, které jsou pro RTOS typické. Příkladem jednotek I/O jsou LED diody, display atd. [22]

3.4.5 Časovače

RTOS má přesné požadavky na časové omezení. I přes tento požadavek tato služba poskytuje základní časovače, například možné držení nebo kontrola prodlevy. [9]

3.5 Přehled RTOS pro vestavěné systémy

V této kapitole jsou popsány oficiální distribuce RTOS, jejich výhody a omezení. Využití oficiální distribuce, přináší výhodu široké podpory včetně příkladů pro řešení různých problémů spojených s jejich implementací do HW. Další nespornou výhodou oficiálních distribucí je dostupnost ovladačů zařízení a knihovny, které zjednodušují tvorbu kódu. Na základě srovnání dostupných RTOS bude vybrán jeden, který bude použit v praktické části diplomové práce.

3.5.1 FreeRTOS

FreeRTOS je operační systém určený pro vestavěná zařízení. Systém je navržen tak, aby byl malý a jednoduchý. Celý koncept je možné rozdělit do tří vrstev. První vrstva obsahuje hardware (ovladače), druhá vrstva je již nezávislá na hardwaru a třetí obsahuje základ operačního systému. Většina souborů je napsaná v jazyce C, aby se usnadnila údržba a ovládání I/O zařízení. Operační systém nám poskytuje metody pro více vláken, procesů nebo softwarových časovačů. Pro aplikace, které vyžadují nízkou spotřebu

energie, je k dispozici režim bez časovače. V operačním systému je podporovaná priorita vláken. Lze alokovat paměť pomocí pěti schémat pro její správu. Systém může buď paměť pouze přidělit, nebo přidělit a zároveň uvolnit pomocí velmi jednoduchého, rychlého algoritmu. Dále existuje algoritmus, který je náročnější, ale rychlý a obsahuje coalescenci paměti. Alternativou je složitější schéma, které umožňuje rozběhnutí více oblastí paměti a knihovny v jazyce C, které nám alokují a uvolní paměť s určitou ochranou proti vzájemnému vyloučení.

Neexistují zde žádné složitější funkce, které se obvykle vyskytují například v operačním systému Linux nebo Windows, jako jsou ovladače zařízení nebo uživatelské účty.

FreeRTOS je nejrozšířenější operační systém na trhu - a to zejména kvůli vysokému počtu podporovaných architektur a pestré škále překladačů. Tento systém byl vyvinut a je nadále udržován Richardem Barrym ve společnosti Real Time Engineers. [23]

3.5.2 BitThunder

Je operační systém reálného času, který jako open-source tvoří James Walmsley. Operační systém vychází z FreeRTOS a jeho aktuální verze je dostupná pod licencí GPL verze 2.0. Jelikož se jedná o systém, který je stále vyvíjen, poskytuje pouze základní funkce, a zatímco velká část dalších funkcí je stále vytvářena. Systém umožňuje spuštění více úloh najednou pro zadané frekvence ze zadání. Dále nabízí funkce – třeba pro práci s rozhraním I2C, GPIO nebo s časovači. Operační systém má sloužit jako náhrada v aplikacích, u kterých je nyní používán OS Linux.

Jedná se tedy o náhradu operačního systému Linux a jeho zavaděče U-boot. OS BitThunder zkrátí čas bootování a vylepší realtimové vlastnosti systému. Snahou vývojáře BitThunder je vytvoření OS, který bude co možná nejkompatibilnější a který by pracoval na všech zařízeních. [24]

3.5.3 ChibiOS/RT

ChibiOS obsahuje mnohem více funkcí než FreeRTOS, které nám zajišťují přehlednost a přenositelnost na jiné platformy. Tento systém je volně dostupný pro soukromé použití. ChibiOS nám nabízí dvě různá jádra. RT jako rychlý operační systém, který se snaží být co nejmenší z hlediska velikosti kódu. K operačnímu systému je dodáván modul HAL, který umožňuje abstrakci mezi hardwarem (jako je USB, Ethernet) a aplikací. Velkou předností je vývojové prostředí, které je dodáváno s oficiálními demo projekty a je volně dostupné ke stažení. [25] Operační systém obsahuje funkce i pro možnost přidělení priority jednotlivých vláken, pro tvorbu více vláknových aplikací nebo správu časovačů a mnoho dalších modulů. [24]

3.5.4 Systém reálného času pro multiprocesorové systémy

Real Time Executive for Multiprocessor Systems - zkráceně RTEMS. Jak už vyplývá z názvu, jde o RT OS, který je navíc open source. Systém se využívá v letectví,

medicíně, armádě nebo třeba ve vesmírných programech. Je podporovaný pro širokou škálu architektur ARM, PowerPC, Intel a jiné. Díky standardu, který se používaná hlavně pro unixové operační systémy (POSIX – Porable Operating System Interface) je kombinovaný a může být využit pro širokou řadu GNU nástrojů, které se využívá pro generování spustitelných souborů například podpora pro různé programovací jazyky. Avšak primárními programovacími jazyk jsou C, C++ a ADA95. Systém RTEMS je široce škálovatelný. Nejmenší verze systému, která je funkční může mít velikost pod 20 kB. Funkce, které systém poskytuje, jsou především dynamická alokace paměti, vlákna kompatibilní s normou POSIX 1003.1b, vnitřní procesorovou komunikaci, synchronizaci, preemptivní plánování a volitelné RMS plánování. Systém je vytvořený z více projektů, proto každá jeho část spadá pod jinou licenci. Primární licence je GPL2 pod kterou patří většina systému. [26]

3.5.5 RISC OS

RISC OS je plnohodnotný operační systém, který se specializuje na Acorn RISC Machine (ARM) architekturu. Operační systém byl navržen společností Acorn Computers, která sídlí v Cambridge v Anglii. Tato společnost navrhla i architekturu ARM. [27] Systém RISC OS má velmi malé a rychlé jádro. Tento systém je daleko jednodušší, než moderní OS jako je například Linux. Je velmi jednoduchý na pochopení a má velmi dobře zdokumentovanou komunikaci mezi moduly. Grafické rozhraní operačního systému může mít velikost pod 6 MB. Systém využívá kooperativní multitasking, který umožnuje vytvářet aplikace s potřebou mít celý operační systém pod kontrolou. Příkladem je komunikace s hardwarem, který vyžaduje přesné časování. [28] Nejedná se ovšem o systém reálného času v pravém slova smyslu.

3.5.6 Xenomai

Xenomai je operační systém, který spolupracuje s jádrem OS Linux a je určený pro real-time aplikace. Operační systém je typu open-source a v současné době pracuje pod licencí GPL v2. Systém se stále vyvíjí. Současná stabilní verze, na které systém funguje ve Xenomai 3, která odporuje konfiguraci jednoduchého i duálního jádra. Ukončená architektura Xenomai 2 podporovala pouze konfiguraci duálního jádra. Architektura Xenomain 3 je rozdělena na dva typy jader. První se nazývá Xenomai 3 Cobalt, která je výstupem vývoje verze 2. Představuje nám situaci dvou jader, které běží vedle sebe na jednom hardwarovém zřízení. Stejný princip představovala i verze 2. Hlavním rozdílem mezi verzemi je přemisťování emulátoru API z prostoru jádra do uživatelského prostředí. Druhý typ jádra je Xenomai 3 Mercury, jedná se o jedno jádrové pojetí, kde je jádro operačního systému vylepšené o Mercury. Hlavní výhoda dané architektury je vemulaci POSIX, které je nekompatibilní s API v reálném čase. [29]

3.5.7 Azure RTOS ThreadX

Azure RTOS ThreadX je plně deterministický operační systém vytvořený pro zařízení IoT využívající 32bitové procesory. Existuje více verzí jádra včetně extrémně malého označovaného jako picoKernel. Je vyvíjen společností Microsoft a jako jediný z výčtu není open-source. [30]

4. MIKROKONTROLER

Pro realizaci aplikace jsem zvolila řadu kontrolérů ESP 32, která je velmi rozšířená a podporuje implementaci operačního systému reálného času. Jedná se řadu vývojářských desek a čipu s podporou různých periferií. V této kapitole jsou popsány jednotlivé nabízené moduly, z nichž bude vybrán nejlepší pro realizaci aplikace. Samotný výběr vývojového kitu bude probíhat na základě získaných informacích v kapitole 6.2.1

4.1 Společnost Espressif

Čínská společnost Espressif je producentem 32-bitových mikrokontrolerů známých pod označením ESP. Prvním systémovým řešením byl modul pojmenovaný ESP8089, který se na trhu objevil v roce 2013. Tento modul byl určen pro tablety a set-top boxy. Cílem společnosti je uvedení špičkových řešení pro umelou inteligenci věcí (AIoT) na trh s podporou bezdrátové technologie s nízkou spotřebou. [33]

V následujících kapitolách bude uveden výčet základních typů modulů [32], ze kterých bude zvolen vhodný pro realizaci aplikace.

4.2 Řada ESP32-S

Řada modulů založených na 32bitových mikroprocesorech typu Xtensa LX7 pracující až na 240 MHz s integrovanou podporou 2.4 GHz WiFi.

4.2.1 Modul ESP32-S2

Řada ESP32-S2 je založena na jedno-jádrovém mikroprocesoru. Jsou zde bezpečnostní funkce: eFuse, Flash šifrování, zabezpečené zavádění, ověřování podpisu a hardwarová podpora algoritmů AES, SHA a RSA. [32]

Mezi integrované periferie se řadí 43 V/V pinů, plnohodnotné USB OTG rozhraní, SPI, I2S, UART, I2C, LED PWM, rozhraní pro připojení LCD zobrazovače, rozhraní kamery, ADC, DAC, dotykový senzor nebo teplotní senzor. [32]

4.2.2 Modul ESP32-S3

Řada ESP32-S3 má dvoj-jádrový mikroprocesor. Podporuje vektorové instrukce v MCU, které poskytují zrychlení pro výpočetní úlohy v neuronové síti a zpracování signálů. Mezi periferie patří 45 V/V pinů, které podporují SPI, I2S, I2C, PWM, RMT, ADC, DAC a UART, SD/MMC host a TWAITM. Zabezpečení paměti programu díky spouštění s využitím šifrování na bázi RSA, šifrování Flash AES-XTS, digitálním podpisem a periferním zařízením HMAC. [32]

4.3 Řada ESP32-C

Řada modulů založených na jedno-jádrových 32bitových mikroprocesorech typu RISC-V pracující až na 120 MHz (C2) nebo 160 MHz (C3 a C6) s integrovanou podporou 2.4 GHz WiFi a Bluetooth 5 (LE).

4.3.1 Modul ESP32-C2

Moduly obsahují 14 V/V pinů, které podporují sběrnice SPI, UART, I2C, řadič LED PWM, obecný řadič DMA (GDMA), SAR ADC, teplotní senzor. [32]

4.3.2 Modul ESP32-C3

Řada ESP32-C3 podporuje bezpečnostní funkce pro spuštění na bázi RSA-3072, flashové šifrování pro AES-128-XTS, digitální podpis a periférie HMAC s podporou hardwarové akcelerace kryptografických algoritmů. Dále rozsáhlou sadu periférii a PIO pinů. Modul je ideální pro různé scénáře a složité aplikace. [32]

4.3.3 Modul ESP32-C6

Moduly ESP32-C6 obsahují jedno-jádrový procesor RISC-V, který může pracovat až s frekvencí 160 MHz. Nalezne zde 30 nebo 20 pinů GPIO, které podporují sběrnice SPI, UART, I2C, I2S, RMT, TWAI a PWM. [32]

4.4 Řada ESP32-H2

Řada disponuje jedno-jádrovým 32bitový procesor RISC-V, který pracuje s frekvencí až 96 MHz. Je zde k dispozici 19 V/V pinů, které podporují UART, SPI, I2C, I2S, periferie dálkového ovládání, LED PWM, řadič USB Serial/JTAG s plnou rychlostí, GDMA, MCPWM. Může být použit pro koncové zařízení. [32]

4.5 Řada ESP32

Řada ESP32 je založena na jedno-jádrovém nebo dvoj-jádrovém 32bitovém mikroprocesoru Xtensa LX6 s nastavitelnou frekvencí v rozsahu 80 MHz až 240 MHz. Stejně jako v předchozích případech obsahuje modul radiový front-end s výstupním výkonem až 19,5 dBm. Dále je podporován Bluetooth 5 - Low Energy včetně L2CAP, GAP, GATT, SMP a profilů založených na GATT, jako je SPP-like atd. Bluetooth se připojuje k chytrým telefonům a vysílá nízkoenergetické signály pro snadnou detekci. Mezi periférie, které jsou podporovány patří – dotykové senzory, Hallův senzor, rozhraní pro SD karty, Ethernet, vysokorychlostní SPI, UART, I2S a I2C. [32]

4.6 Řada ESP8266

Obdobně jako řada ESP32 je řada ESP8266 založena na jedno-jádrovém 32bitovém mikroprocesoru Xtensa LX6, který může pracovat až s frekvencí 160 MHz. Napájecí proud může být menší než 20uA. Je tedy vhodný pro bateriové napájení a nositelné mobilní elektronické zařízení. Mezi základní periférie patři UART, GPIO, I2C, I2S, SDIO, PWM, ADC a SPI. [32]

4.7 Vývojové kity s ESP32

Pro realizaci zadání bylo nutné vybrat konkrétní vývojový kit, který disponuje univerzálním asynchronním přijímačem/vysílačem a je dostatečně výkonný pro zabezpečení pomocí SSL/TLS.

Dostupných vývojových desek založených na mikroprocesorech ESP32 je velká řada. V následujících kapitolách bude popsáno několik z nich.

4.7.1 ESP32 – Core Board

ESP32-Core Board nebo také ESP32-DevKitC byla navržena tvůrci procesoru ESP32 Espressif Systems. Deska byla prototypem pro nový ESP32SoC a je dnes již zastaralá.[34]

Základní specifikace kitu:

- WiFi + BLE včetně ESP32-WROOM-32
- Rozhraní USB
- Dvě tlačítka
- Stavové LED
- Rozměry 54,32 x 27,9 mm

4.7.2 ESP32 – Gateway

Vývojová deska ESP32-GATEWAY je založena na modulu ESP32-WROOM-32D. Byla vytvořena firmou OLIMEX jako open source projekt. Jde o vývojový kit se 100 Mbit Ethernetovým rozhraním, Bluetooth LE a WiFi ideální pro aplikace IoT. [34]

Základní specifikace kitu:

- Mikro USB konektor
- CH340 USB převodník
- Vestavěný program pro Arduino a ESP-IDF
- Wifi a BLE konektivitu
- Rozhraní pro Ethernet 100 Mbit
- MicroSD karta
- 20 pinový konektor GPIO se všemi porty ESP32
- Rozměry 50x62 mm

4.7.3 ESP32-DevKit-Lipo

Deska vznikal jako další open source hardware projekt. Existuje zde kompatibilita s Espressif ESP32 – CoreBoard, ale má přidanou LiPo nabíječku a schopnost pracovat z připojených baterií, pokud chybí externí zdroj napájení, což činí modul ideální pro přenosné aplikace. Dodává se s modulem ESP32 s konektorem a připojenou externí anténou. [34]

Základní specifikace kitu:

- Modul ESP32-WROOM-32D WiFi/BLE nebo modul ESP32-WROVER
- Uživatelské tlačítko
- Mikro USB konektor
- Vestavěný USB-Seriál program
- Vestavěná LiPo nabíječka
- Konektor LiPo baterie
- Rozměry: 48x28 mm

4.7.4 ESP32-EVB

Kit je opět open source hardware projekt. Jedná se desku s Ethernetovým rozhraním, Bluetooth LE, Wi-Fi, dálkovým ovládáním IR a CAN sběrnicí. Je tak ideálním základem pro IoT aplikace. Kit může pracovat s jednou LiPo záložní baterií, proto má integrovánu nabíječku. Dvě relé umožnují zapínat a vypínat spotřebiče. Deska je dodávána s EPS32-WROOM-32Ue a externí anténou. Tato deska byla původně navržena společností OLIMEX Ltd a je výsledkem neustálého zlepšování. [34]

Základní specifikace kitu:

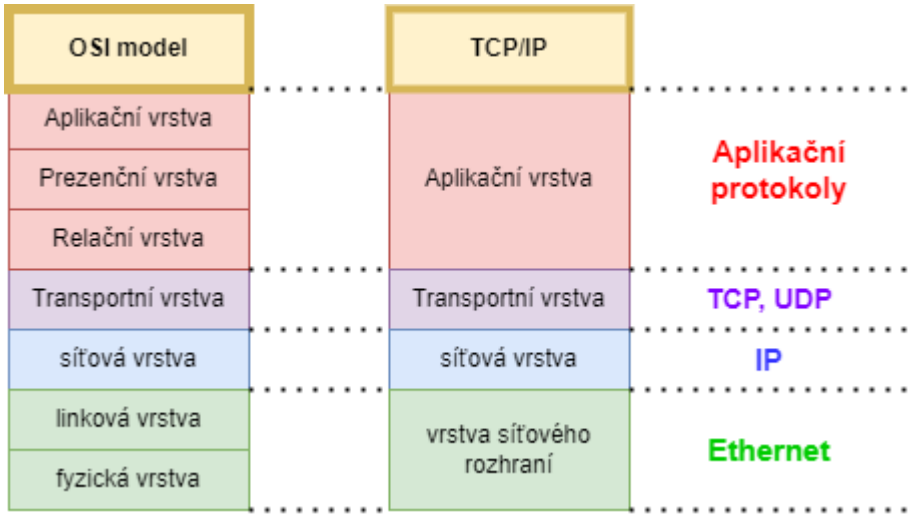
- Modul ESP32-WROOM-32E nebo ESP32-WROOM-32UE
- Vestavěný program pro Arduino a ESP-IDF
- WiFi a BLE konektivita
- Rozhraní Ethernet
- Mikro SD karta
- 2 x 10A/250VAC (15A/120VAC 15A/24VDC) relé
- Rozhraní CAN
- IR přijímač a vysílač, až na vzdálenost 5 metrů
- LiPo nabíječka pro samotný provoz během výpadku napájení se 4 stavovými LED
- Napájecí konektor pro externí napájení 5V DC
- Konektor UEXT pro připojení modulu
- Reset a uživatelská tlačítka
- 40 pinový GPI konektor se všemi porty ESP32
- Volitelná externí anténa
- Rozměry 75x75 mm

5. KOMUNIKAČNÍ PROTOKOLY PRO KOMUNIKACI PO SÍTI

5.1 TCP protokol

Protokol TCP – Transmission Control Protocol je protokol transportní vrstvy v rámci ISO OSI modelu. Společně s protokolem IP (Internet Protocol) síťové vrstvy tvoří tato dvojice základ celého dnešního internetu. Tím, že je tato dvojice protokolů velmi známá, není ji nutné detailně popisovat. V této kapitole budou proto popsány pouze základní informace.

Zjednodušením OSI ISO modelu vznikl TCP/IP model. Na rozdíl od OSI modelu má pouze čtyři vrstvy. Porovnání vrstev těchto dvou modelů je na obr č. 14.



Obr č. 14 Porovnání modelů ISO OSI a TCP/IP

Následující kapitoly popisují jednotlivé vrstvy modelu TCP/IP.

5.1.1 Vrstva síťového rozhraní

Vrstva síťového rozhraní TCP/IP modelu představuje nejnižší vrstvu zodpovědnou za přístup k fyzickému médiu a řízení komunikace mezi zařízeními. Její implementace závisí na typu fyzického média, které používá síť. Protokol TCP/IP tak může být v principu implementován na libovolném typu fyzického média a typu sítě. Model tak předpokládá, že fyzická a linková vrstva je řešena separátně od TCP/IP komunikace. [38]

5.1.2 Síťová vrstva

Síťová vrstva TCP/IP modelu je klíčovou vrstvou, jejíž hlavním cílem je zajištění rychlé komunikace. Proto je takto zajišťovaná komunikace nespolehlivá. Pokud komunikující strany požadují spolehlivý přenos dat, musí si ho zajistit

svépomocí ve vyšších vrstvách modelu. Tento protokol dostal jméno IP (Internet Protocol).

Klíčové vlastnosti IP protokolu:

- nespolehlivý – při nesprávném doručení paketu se nestará o nápravu a
- nespojovaný – cesta paketů není deterministická, proudí rozdílnými cestami.

5.1.3 Transportní vrstva

Transportní vrstva TCP/IP modelu je klíčovou vrstvou, která zajišťuje komunikaci mezi dvěma koncovými body v síti. Tato vrstva má za úkol rozdělit data, která mají být poslána mezi koncovými body, na menší jednotky, které se nazývají segmenty. Každý segment obsahuje informace o zdrojové a cílové adrese, aby mohl být správně doručen.

Na úrovni transportní vrstvy se používají dva hlavní protokoly: TCP (Transmission Control Protocol) a UDP (User Datagram Protocol). TCP je spojovaně orientovaný protokol, který zajišťuje spolehlivý přenos dat mezi koncovými body. Protokol TCP má mechanismy pro řízení přetížení sítě, řízení toku dat a kontrolu chyb přenosu. Tyto mechanismy zajišťují, že jsou data přenášena spolehlivě a bez ztrát.

Protokol UDP je nepotvrzovaný protokol, který sice neposkytuje spolehlivý přenos dat, ale je díky tomu rychlejší než TCP. Protokol UDP je často využíván aplikacemi, které nevyžadují spolehlivost, jako jsou například hlasové hovory nebo videohovory.

Transportní vrstva také poskytuje kontrolu toku dat, což znamená, že při přenosu dat mezi dvěma koncovými body může být zajištěno, aby rychlost přenosu byla optimální a aby nedocházelo k zahlcení sítě.

V rámci transportní vrstvy jsou také implementovány mechanismy pro řízení přenosu dat mezi aplikacemi, které jsou běžně používány na síti. Tyto mechanismy zahrnují správu spojení, kontrolu chyb, řízení toku dat a řízení přetížení sítě. Celkově lze říci, že transportní vrstva TCP/IP modelu je klíčová pro správné fungování sítě. [38]

5.1.4 Aplikační vrstva

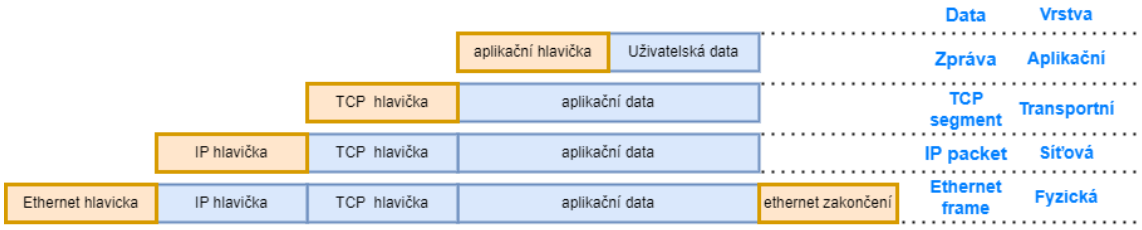
Aplikační vrstva je nejvyšší vrstvou TCP/IP modelu a poskytuje uživatelským aplikacím rozhraní k síťovému protokolu. Tato vrstva zahrnuje mnoho protokolů, které se používají pro komunikaci mezi aplikacemi. Každý protokol této vrstvy je navržen tak, aby poskytoval určitou funkčnost.

Mezi nejznámější protokoly této vrstvy patří HTTP, který se používá pro komunikaci webových prohlížečů s webovými servery, FTP, který umožňuje přenos souborů mezi různými počítači, a DNS, který překládá názvy domén na IP adresy. [38]

Aplikační vrstva se často označuje jako "uživatelská" vrstva, protože zde jsou implementovány protokoly, které jsou viditelné pro uživatele. Tyto protokoly využívají nižších vrstev (transportní, síťové a linkové), aby zajistily bezpečnost, spolehlivost a rychlost přenosu dat mezi počítači v síti. [38]

5.1.5 Formát rámce TCP/IP modelu

Základním prvkem rámce je jeho struktura, která se skládá z hlavičky (header), dat (payload) a zakončení (trailer). Data přenášená v rámci jsou složena z informací, které jsou zapouzdřené v datových jednotkách vyšších vrstev. Protože v rámci modelu TPC/IP existují čtyři základní vrstvy, každá z nich má svou specifickou strukturu hlavičky. Obr č. 15 zobrazuje tyto vrstvy spolu s formátem dat, jak jsou vzájemně zapouzdřována odshora dolů. Příklad je uveden pro protokol TCP, ale tento princip je platný pro všechny protokoly v TCP/IP modelu. Obsah jednotlivých hlaviček bude popsán v textu dále. [38]



Obr č. 15 Rámec TCP/IP modelu

Ethernetová hlavička

Ethernetová hlavička obsahuje zdrojovou a cílovou MAC adresu (fyzickou adresu zařízení) a typ/délku rámce. Typ v hlavičce určuje, jaký protokol je použit na vyšší vrstvě (IP, ARP, atp.). K synchronizaci komunikace se na začátku vysílá speciální úvod (preamble), který je složen ze sekvence střídajících se jedniček a nul následovaných oddělovačem SFD (Start Frame Delimiter), který označuje začátek rámce. Data v rámci jsou složena ze zapouzdřených dat vyšších vrstev. [38]

IP hlavička

IP hlavička se skládá ze zdrojové a cílové IP adresy, které slouží k určení logické adresy komunikujících stran. Dále obsahuje informaci o použitém protokolu pro další vrstvu (TCP, UDP, ICMP atd.), délku datagramu a kontrolní součet hlavičky. Standardní velikost IP hlavičky pro IPv4 je 20B, ale může obsahovat i další volitelné položky jako například „*Options*“ nebo „*Padding*“. Struktura IP hlavičky se liší podle použité verze IP protokolu.

TCP hlavička

TCP hlavička obsahuje zdrojový a cílový port, číslo sekvence a potvrzení (pro zajištění spolehlivosti), příznaky (k nimž patří SYN a ACK pro navázání spojení), velikost okna

pro potvrzení a další údaje. Standardní velikost TCP hlavičky je opět 20 bajtů. 80[38]

5.1.6 Navázání a ukončení spojení

Pro vyslání dat pomocí protokolu TCP je nejprve potřeba navázat spojení, což se obvykle provádí pomocí třícestného handshaku. Během tohoto procesu se obě strany dohodnou na čísle sekvence, což jsou 32bitové hodnoty uvedené v TCP hlavičce. Pro navázání spojení se odešle TCP segment s nastavenými příznaky v TCP hlavičce. Tyto příznaky jsou 8bitové hodnoty, které zahrnují CWR (Congestion Window Reduced), ECE (ECN – Echo), URG (Urgent), ACK (Acknowledgement), PSH (Push), RST (Reset), SYN (Synchronize) a FIN (Finish). [39]

Navázání spojení

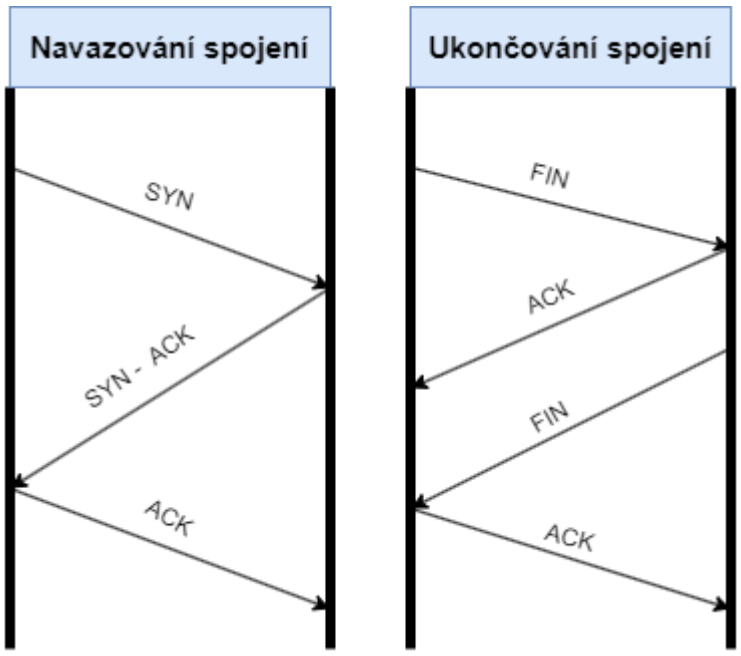
Navázání spojení probíhá během třech kroků:

1. Klient odešle paket SYN s uvedeným číslem sekvence.
2. Druhá strana si uloží sekvenční číslo a odpovídá paketem SYN-ACK. Jako číslo sekvence nastaví svoje číslo a do čísla odpovědi vloží číslo, které odpovídá sekvenčnímu číslu klienta +1.
3. Klient odpovídá paketem AKC s číslem sekvence.

Níže vidíme grafické znázornění navazování spojení. [39]

Ukončování spojení

Proces ukončování spojení je obdobný jako jeho navazování. Nejčastěji se používá čtyřcestný handshake, kdy každá strana samostatně uzavře spojení. Používají se pakety, které nesou hodnoty FIN s odpovědí ACK. Grafické znázornění tohoto procesu je uvedeno na obr č. 16. [39]



Obr č. 16 Ukončování a navazování TCP spojení

5.2 Protokol SSL/TLS

Secure Sockets Layer (SSL) a Transport Layer Security (TLS) jsou protokoly pro šifrování, které umožňují zabezpečený přenos dat mezi klientem a serverem. Protokol TLS je novější verze protokolu SSL. Protokol SSL byl vyvinut v první polovině 90. let společností Netscape Communications Corporation. Tato společnost tehdy prodávala software pro webové servery, které využívají šifrovaný přenos dat.

Protokol TLS byl představen v roce 1999 s cílem vytvořit otevřený a všem dostupný standard určený pro jakoukoli společnost nebo projekt. Protokol SSL byl vytvořen pouze pro HTTP spojení, zatímco v případě TLS se počítalo s protokoly POP3, SMTP a IMAP. Původně se nepředpokládalo, že by mimo banky bylo nutné šifrovat data. [41]

SSL certifikát je použitelný všemi verzemi SSL i TLS protokolu.

5.2.1 Základní charakteristika protokolů SSL a TLS

SSL nebo TLS protokol slouží k vytvoření bezpečného tunelu pro přenos dat mezi klientem a serverem, přičemž ostatní aplikace mohou využívat tento tunel bez nutnosti dodatečného zabezpečení každého používaného protokolu. Například HTTPS je zabezpečená verze protokolu HTTP pomocí SSL/TLS a FTPS je zabezpečená verze protokolu FTP.

Protokol SSL/TLS je nezávislý na informacích, které jsou přenášeny, a nezohledňuje, zda jsou stavové nebo bez stavové. Obvykle je konfigurace serveru nastavena na nejnovější a nejbezpečnější verze protokolu SSL/TLS a šifrování, ale stále podporuje omezený počet starších protokolů v závislosti na kompromisu mezi bezpečností a kompatibilitou se staršími klienty.

Tento protokol využívá komunikaci s vysokým stupněm asymetrického šifrování pro poskytnutí symetrického klíče, což zvyšuje rychlost přenosu dat v probíhající komunikaci. [41]

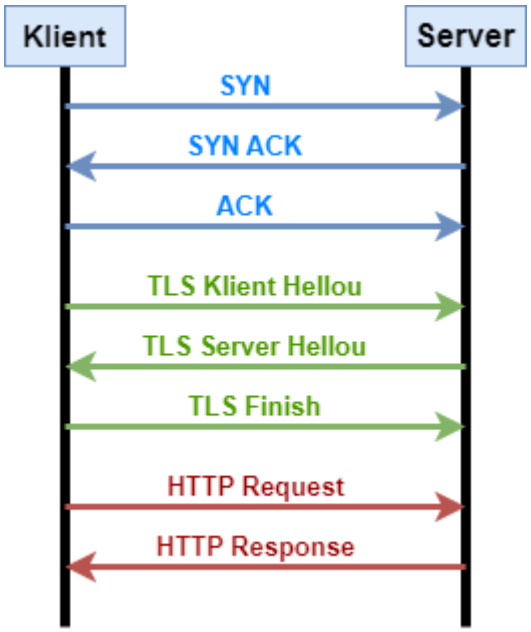
5.2.2 Funkčnost protokolu.

Jak už bylo uvedeno výše SSL/TLS používá asymetrické šifrování, aby se vytvořil kanál, přes který lze předávat symetrické klíče mezi serverem a klientem.

Komunikace pomocí SSL nebo TLS vytváří bezpečné šifrované spojení. Na začátku komunikace dochází k tzv. handshake, během kterého si obě strany dohodnou parametry spojení, včetně použité šifry. Vždy se používá certifikát na straně serveru, což umožňuje ověření identity serveru. Pokud je identita ověřena, může se použít i klientský certifikát. Klient následně odešle zprávu „*pre-master secret*“, kterou zašifruje veřejným klíčem z certifikátu a odešle ji serveru. Zde se využívá

asymetrické kryptografie a rozšifrovat data může pouze server, který vlastní privátní klíč. [41]

Klient i server provedou stejný postup nad „*pre-master secret*“ a tím vytvoří „*master secret*“. Na něj aplikují dohodnutou hashovací funkci a vytvoří **session key**, což je symetrický klíč, který se používá pro šifrování a dešifrování celého spojení. Vlastní komunikace tedy probíhá s využitím symetrické kryptografie pomocí **session key** a asymetrický kanál se již nepoužívá. Na obr č. 17 je graficky znázorněn průběh komunikace přes SSL/TLS.



Obr č. 17 Vytvoření TLS spojení

6. PRAKTICKÁ ČÁST

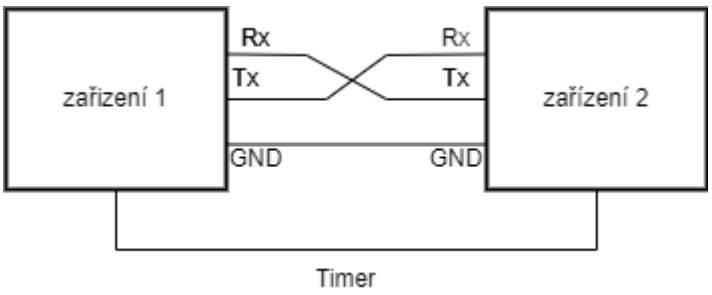
V rámci diplomové práce bylo zapotřebí vytvořit dvě zapojení. Výsledky prvního měření slouží jako reference, díky které je možné určit výsledek zpoždění při druhém zapojení. V této kapitole jsou popsána jednotlivá zapojení, jejich funkce a výsledky měření.

6.1 Přímá komunikace přes UART

Prvním úkolem bylo změřit rychlost komunikace přímé sériové linky mezi dvěma libovolnými mikrokontrolery. V kapitole bude popsáno, jaké zařízení bylo zvoleno, implementaci kódu pro zařízení a jaké jsou výsledky měření. Dále bude rozebráno, kde může nastat chyba měření. Tento kód se již nezmění ani ve druhém zapojení. Schéma zapojení je zachyceno na obr č. 18.

Pro vytvoření aplikace pro první zařízení nejsou kladeny žádné speciální požadavky. Proto jsem použila aplikaci Arduino IDE, které je velmi populární a snadno dostupné. Toto vývojové prostředí je vhodné pro programování vývojových desek s mikrokontrolery a obsahuje mnoho knihoven, které usnadňují práci s různými periferiemi a senzory.

Pro správné fungování druhého zařízení bylo potřeba pracovat na úrovni jednotlivých bitů. Z tohoto důvodu jsem zvolila vývojové prostředí CodeVisionAVR, které umožňuje ovládat integrované periferie mikrokontroleru na nejnižší úrovni. Detailní informace o nutných nastaveních bude uvedeno později.



Obr č. 18 Schéma prvního zapojení

6.1.1 Aplikace prvního zařízení

Jako první zařízení jsem si zvolila vývojovou desku ESP32, která je vysílačem dat. Tato deska obsahuje integrovanou LED, která je využita jako signalizace zahájení komunikace.

6.1.2 Funkce prvního zařízení

V rámci funkce *setup()* bylo nutné nastavit funkci V/V bran mikrokontroleru. Nastavila jsem pin 2 jako výstup pro LED, a pin 4 jako výstup generovaní synchronizačního pulsu *sencorTime*. Dále jsem pro sériovou komunikaci použila RX2 a TX2, které se nacházejí na pinech 16 a 17.
V rámci smyčky *loop()* se vykonává zapnutí a vypnutí LED a to konkrétně pomocí funkce *digitalWrite()*. Dále se používá funkce *delay()* k vytvoření pauzy mezi jednotlivými operacemi. Jestliže LED svítí, je odeslán řetězec „ON\n“ na sériové rozhraní pomocí funkce *Serial.write()*. Zároveň s odvysíláním zprávy na sériovou linku se vytvoří synchronizační puls pro měření.

6.1.3 Funkce druhého zařízení

Jako druhé zařízení jsem zvolila desku Arduino UNO, která je v roli přijímače dat. Aby bylo možné přesně měřit dobu přenosu dat, byl pro toto zařízení napsán program nižší úrovni než u prvního zařízení.

6.1.4 Aplikace druhého zařízení

Přijímač UART je napsán v jazyce C, což vyžaduje nastavení konfiguračních registrů použitých V/V bran. Kromě toho jsou v programu definována makra pro testování různých registrů, která se používají k detekci různých chyb. Například makro DATA_OVERRUN (1 << OVR) označuje bit pro detekci přetečení datového registru při přenosu dat. Pokud je tento bit nastaven na hodnotu 1, znamená to, že došlo k přetečení datového registru a některá data byla ztracena. Byly definovány hodnoty, které přijímač bude podporovat, a ty jsou uloženy v poli s názvem "baudrate". Každá hodnota je uložena v hexadecimální reprezentaci.

Dále je v kódu definována velikost bufferu pro příjem dat (RX_BUFFER_SIZE) a pole pro uložení přijatých dat "rx_buffer". Následující kód zpracovává data přijatá na rozhraní UART a ukládá je do bufferu. Pokud je buffer plný, při dalším přijetí dat se nastaví příznak pro detekci ztráty dat. Při úspěšném přečtení bufferu se nastaví příznak pro označení, že data jsou připravena k zpracování.

Dále byla vytvořena funkce, která slouží k odeslání celého čísla přes sériovou komunikaci. Číslo je nejprve rozděleno na jednotlivé číslice a uloženo v poli "text“ o délce 5 bajtů. Poté se jednotlivé číslice převedou na ASCII znaky pomocí operace sčítání s ASCII hodnotou nuly ('0') a postupně jsou vypsány na sériovou linku pomocí funkce *putchar*(). Na konci funkce odešle také znak nového řádku ('\n').

Jako hlavní funkce programu byla vytvořena nekonečná smyčka, která neustále provádí tyto operace:

1. Kontroluje, zda je připravena nová přijatá data (‚rx_ready‘). Jestliže ano, vynuluje tuto proměnnou a zhasne LED diodu.

2. Zavolá se funkce *Send_int*(time), která odešle aktuální hodnotu proměnné ‚timer‘, která představuje interval od posledního pulsu, jako sériová data připojeným zařízením.
3. Následně se načtou všechna přijatá data z bufferu přijímače (rx_buffer) do pole ‚text‘ a uloží se počet přijatých bytů do proměnné ‚i‘.
4. Jestliže se přijatý první znak rovná ‚O‘ a druhý ‚N‘, odešle se odpověď „ACK“, což znázorňuje zkratku pro potvrzení přijetí zprávy.
5. Ovšem pokud je první znak „B“ a druhý znak je číslo v rozmezí od 0 do 5, změní se rychlost sériové komunikace podle předdefinovaného pole ‚baudrate‘ a přijatého čísla. Následně se buffer přijímače a vysílače vynuluje.

6.1.5 Výsledky měření prvního zapojení

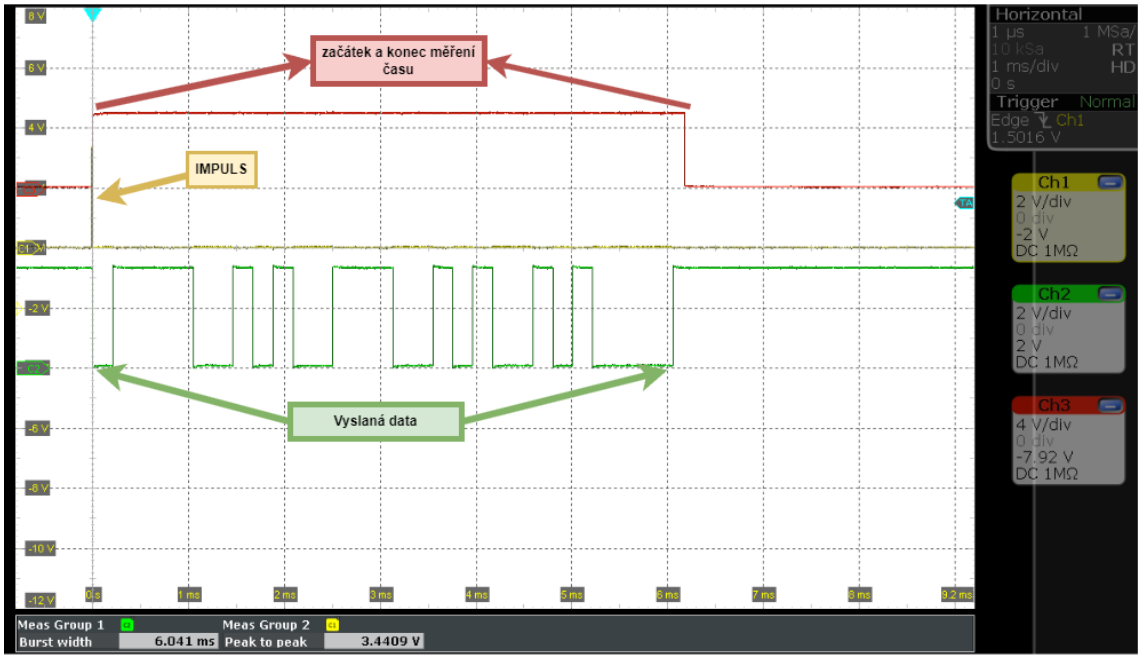
Pro každé měření bude stanovena průměrná hodnota zpoždění, která byla změřena, a také přiložen výstup z osciloskopu. Na obrázcích z osciloskopu jsou zobrazeny sledované čtyři kanály. Každý kanál má následující význam:

- Žlutý kanál ukazuje impuls, který určuje, kdy má být spuštěno měření času.
- Červený kanál ukazuje, kdy byla započata doba měření. A kdy bylo následně měření ukončeno
- Zelený kanál znázorňuje samotná přenášená data, v mém případě zprávu "ON\n".

Na obr č. 19 jsou zobrazeny jednotlivé signály zvýrazněné a popsané kanály. Všechny ostatní výstupy mají tytéž kanály.

Přenosová rychlost 4800 baudů

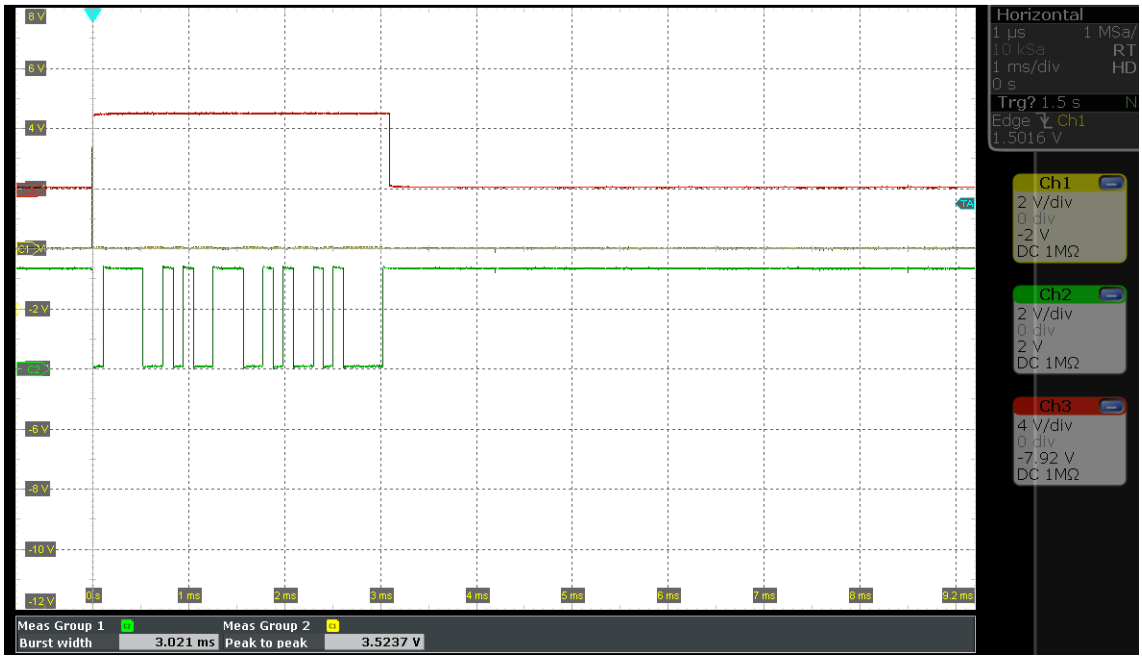
Pro přenosovou rychlost 4800 bylo změřeno zpoždění okolo 6 ms. Výstup z osciloskopu je zobrazen na obr č. 19.



Obr č. 19 Výstup z osciloskopu pro 4800 baudů

Přenosová rychlost 9600 baudů

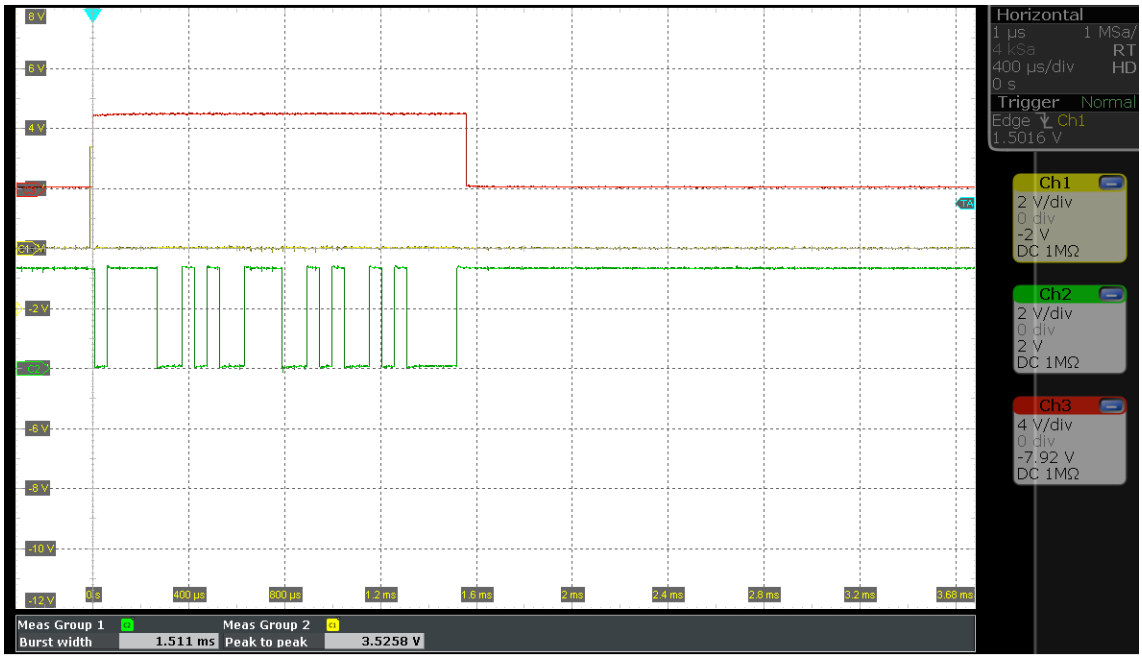
Pro přenosovou rychlost 9600 bylo změřeno zpoždění okolo 3 ms. Výstup z osciloskopu je zobrazen na obr č. 20.



Obr č. 20 Výstup z osciloskopu pro 9600 baudů

Přenosová rychlost 19200 baudů

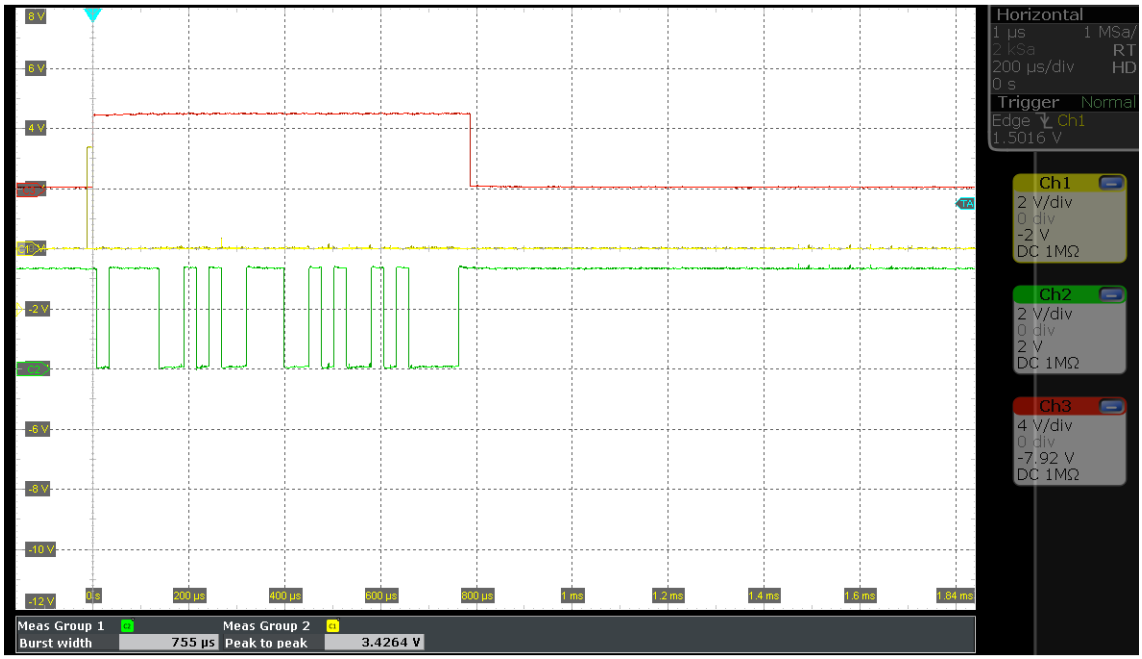
Pro přenosovou rychlost 19200 bylo změřeno zpoždění okolo 1,5 ms. Výstup z osciloskopu je zobrazen na obr č. 21.



Obr č. 21 Výstup z osciloskopu pro 19200 baudů

Přenosová rychlost 38400 baudů

Pro přenosovou rychlost 38400 bylo změřeno zpoždění okolo 750 µs. Výstup z osciloskopu je zobrazen na obr č. 22.

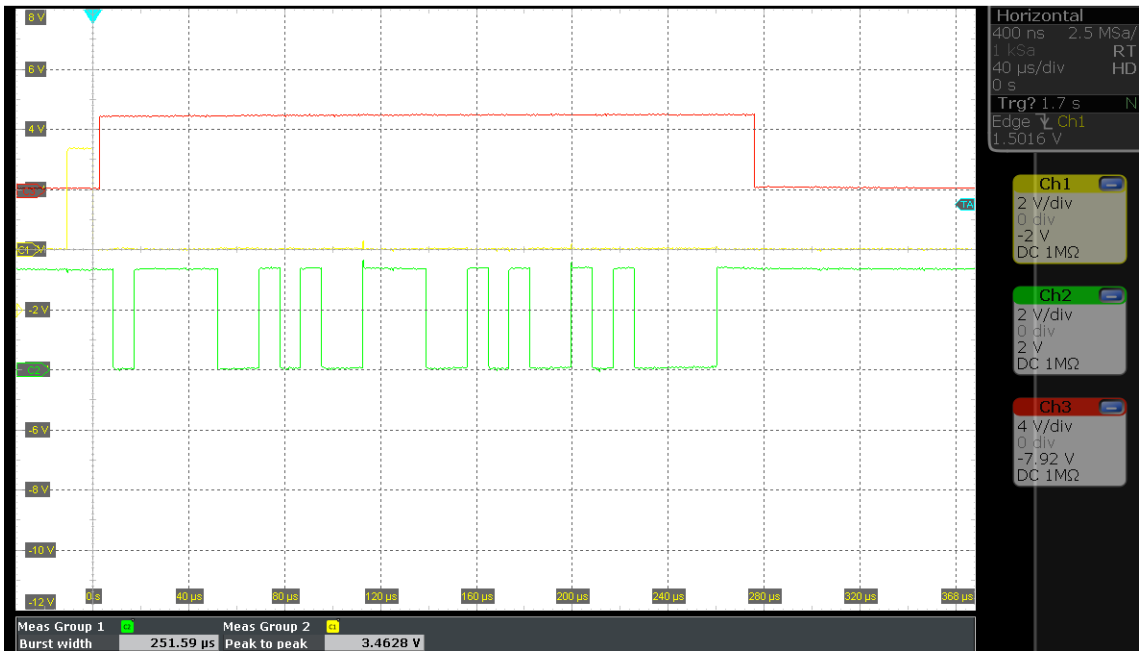


Obr č. 22 Výstup z osciloskopu pro 38400 baudů

Přenosová rychlost 115200 baudů

Při vývoji aplikace pro druhé zařízení jsem zjistila, že zvolené zařízení není vybaveno vhodným oscilátorem, pro generování hodinového signálu nutného k nastavení zvolené

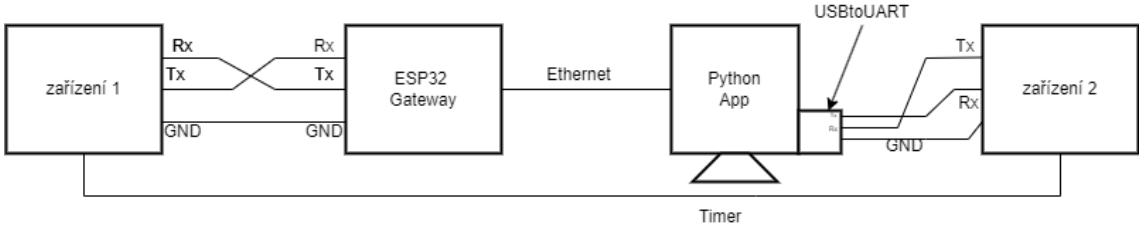
přenosové rychlosti. To má za následek chybu nastavení rychlosti komunikace okolo 4 %, což je hodnota mimo přípustný limit a může tak docházet k pravidelným výpadkům komunikace. I přesto se podařilo zachytit a změřit tuto komunikaci. Při přenosové rychlosti 115200 bylo změřeno zpoždění okolo 500 µs, jak je vidět na výstupu z osciloskopu na obr č. 23.



Obr č. 23 Výstup z osciloskopu pro 115200 baudů

6.2 Komunikace přes TCP a SSL/TLS protokoly

Druhým úkolem diplomové práce bylo původní zapojení rozšířit o komunikaci skrze lokální síť Ethernet s šifrováním protokolem SSL/TLS. Schéma rozšířeného zapojení je zobrazeno na obr č. 24. Programy zařízení 1 a 2 jsou stejné jako při prvním měření, aby bylo možné porovnat nárůst zpoždění přenosu informací s komunikací bez šifrování. V kapitole jsou podrobně popsány obě aplikace a zařízení, které jsem na základě teoretické části vybrala. Na závěr zde jsou uvedeny výsledky měření.



Obr č. 24 Schéma druhého zapojení

6.2.1 Volba zařízení pro vývoj aplikace v RTOS

Na základě rešerše, provedené v teoretické části, byl vybrán modul ESP32, protože podporuje implementaci operačního systému reálného času, který je nutný pro výslednou aplikaci. Jako vývojové kity byli vybráni dva kandidáti, kteří splňují kritéria pro splnění zadání diplomové práce – ESP32 Gateway a ESP32-EVB. Při konečném rozhodování bylo zjištěno, že deska ESP32-EVB má možnost připojení sběrnice CAN, což pro účely diplomové práce není nezbytné, a proto byla zvolena deska ESP32 Gateway, která disponuje dostatečným počtem UART rozhraní a RJ-45 konektorem.

6.2.2 Aplikace na zařízení ESP32-Gateway

Funkce této aplikace je příjem dat ze sériové linky a jejich předání dál přes lokální síť Ethernet.

Sestavení aplikace

Pro sestavení aplikace na zařízení EPS32 se používá terminálová aplikace nazvaná „ESP-IDF installer tools“.

Pro správu sestavení, nasazení a ladění projektu se používá front-end s názvem **idf.py**, který je součástí terminálové aplikace ESP-IDF. Všechny potřebné příkazy začínají prefixem **idf.py**.

Pro vytvoření nového projektu je nutné nejprve spustit příkaz

```
idf.py create-projekt <název projektu>
```

, případně s parametrem *--path*, který určuje cestu ke složce, kde bude projekt vytvořen. Poté je třeba se přesunout do složky s vytvořeným projektem, kde se nachází soubor main.c.

Prvním příkazem je

```
idf.py set-target esp32
```

Tento příkaz vymaže adresář sestavení a znovu vygeneruje soubor konfiguračního souboru *sdkconfig*. Předchozí konfigurační soubor bude uložen jako ***sdkconfig.old***. Příkaz provede tyto kroky:

- Vymaže adresáře, kde se nachází sestavený projekt.
- Odstraní konfigurační soubor a nahradí ho novým.
- Nakonfiguruje projekt s novým cílem, v tomto případě je to ESP32.

Další konfiguraci desky lze provést pomocí příkazu

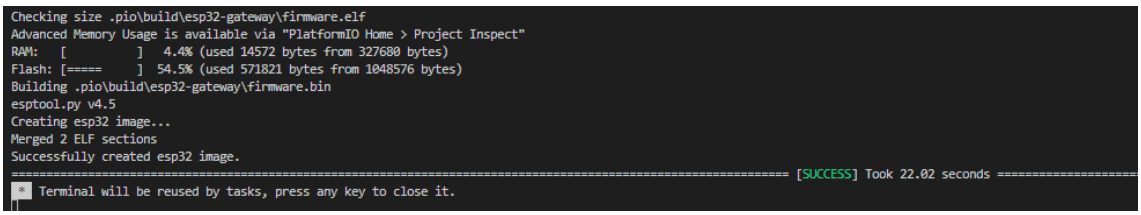
```
idf.py menuconfig
```

,který spustí grafický konfigurační nástroj, kde je možné nastavit další potřebné volby pro vytvoření aplikace.

Pokud je konfigurace dokončena a splňuje všechny požadavky, tzn. je vytvořený konfigurační soubor a správná nastavení z grafického nástroje, je možné se přesunout k samotnému sestavení aplikace pomocí následujícího příkazu:

```
idf.py build
```

Po spuštění příkazu se v aktuálním adresáři spustí sestavování projektu. Pokud nebyly ve zdrojových souborech nebo v konfiguraci žádné změny od posledního



Obr č. 25 Úspěšné sestavení aplikace

sestavení, při spuštění příkazu se nic nezmění. V terminálovém okně se zobrazí status sestavování, podobný tomu, který je vidět na obr č. 25.

Sestavování probíhá přírůstkově. Pokud je žádoucí sestavit pouze zavaděč nebo tabulku oddílů, může být přidán parametr *app*. Pro nahrání aplikace do vývojového kitu je nutné jen nejprve připojit k počítači pomocí USB kabelu a zjistit, jaké číslo COM portu bylo kitu přiděleno operačním systémem. Po zjištění čísla COM portu je možné nahrát aplikaci do kitu pomocí následujícího příkazu v terminálové aplikaci:

```
idf.py -p COM_PORT flash
```

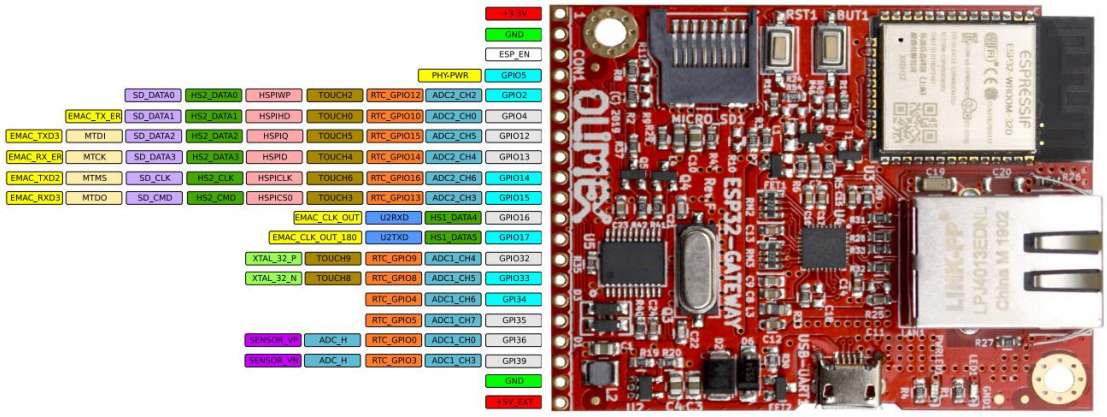
Pro sledování chování aplikace se používá příkaz

```
idf.py monitor
```

Při práci na aplikaci jsem narazila na plugin „PlatformIO", který byl vytvořen pro aplikaci Visual Studio Code. Tento plugin umožňuje velmi přívětivě a intuitivně vytvářet nové projekty, sestavovat je a nahrávat do kitu přímo v rámci tohoto programu. Proto jsem se rozhodla používat tento plugin. Jediná nevýhoda spočívá v neúplném zobrazování monitoru pro sledování běhu aplikace, a proto jsem pro monitorování používala aplikaci ESP-IDF.

6.2.2.1 Aplikace na ESP32 Gateway

Aplikace je psána v jazyce C/C++. Jak bylo uvedeno výše, pro vývoj byl použit program Visual Code.



Obr č. 26 V/V brány (GPIO piny) desky ESP32 GATEWAY[34]

Jako první jsem si zjistila hardwarové informace o desce, například jaké V/V brány se používají pro přijímání dat ze sériové komunikace. Rozložení V/V bran je zobrazeno na obr č. 26.

Nastavení V/V bran pro sériovou komunikaci

Pro splnění zadání bylo nutné inicializovat a vytvořit úlohu pro příjem dat ze sériové linky. Inicializace sériového rozhraní probíhá pomocí několika příkazů, které jsou zde stručně popsány. Vytvořená funkce pro inicializaci byla pojmenována *uart_init()* a je volána několikrát během běhu aplikace.

Funkce nejprve kontroluje, zda je UART již inicializován. Pokud již inicializován je, tak funkce uvolní strukturu ovladače pro toto UART rozhraní pomocí funkce

```
uart_driver_delete(UART_NUM_2)
```

, která uvolní alokované prostředky pro toto rozhraní.

```
uart_config_t uart_config = {
    .baud_rate = baudrate_global,
    .data_bits = UART_DATA_8_BITS,
    .parity    = UART_PARITY_DISABLE,
    .stop_bits = UART_STOP_BITS_1,
    .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
    .source_clk = UART_SCLK_DEFAULT
    };
    uart_param_config(UART_NUM_2, &uart_config);
    uart_driver_install(UART_NUM_2, 1024, 0, 0, NULL, 0);
```

Jako první se vytvoří struktura *uart_config_t* sloužící k inicializaci a nastavení parametrů sériové komunikace pro UART, aby bylo možné přenášet data. Tato struktura je inicializována s následujícími parametry:

- Baud rate – rychlost přenosu v bit/s, ve struktuře je nastavena pomocí parametru baudrate_global, která má defaultní hodnotu 9600 baudů.
- Data_bits – počet datových bitů v rámci jednoho přenosu, hodnota je nastavena na 8 bitů
- Parity – parita, hodnota je nastavena na DISABLE (vypnuta)
- Stop_bits – počet stop bitů v rámci jednoho přenosu, zvolená hodnota je 1
- Flow_control – řízení toku, které je nastaveno na DISABLE (vypnuto)
- Source_clk – zdroj hodin pro UART, který je nastaven na výchozí hodnotu

```
uart_driver_install(UART_NUM_2, RX_BUF_SIZE * 2, 0, 0, NULL, 0);
```

Funkce inicializaci ovladače UART je pojmenována *uart_driver_install()*. Funkce inicializuje ovladač pro určité rozhraní a nastavuje buffer pro příjem. Pro správné nastavení ovladače je zapotřebí nastavení šesti parametrů:

- UART_NUM_2 - určuje číslo UART rozhraní, pro které se má inicializace provést
- RX_BUF_SIZE * 2 - velikost bufferu pro příjem dat ze sériového portu, v našem případě je buffer nastaven na dvojnásobek velikosti 1024

- 0 - počet bajtů v bufferu pro vysílání dat, v našem případě je nastaven na 0, což znamená, že buffer pro vysílání nebude použit
- 0 - nastavení timeoutu pro blokující operace, v našem případě je timeout nastaven na 0, což znamená, že blokující operace bude čekat nekonečně dlouho na dokončení operace
- NULL - fronta pro RX data, v našem případě není použita
- 0: - délka fronty pro RX data, v našem případě je délka fronty nastavena na 0, což znamená, že fronta pro RX data nebude použita

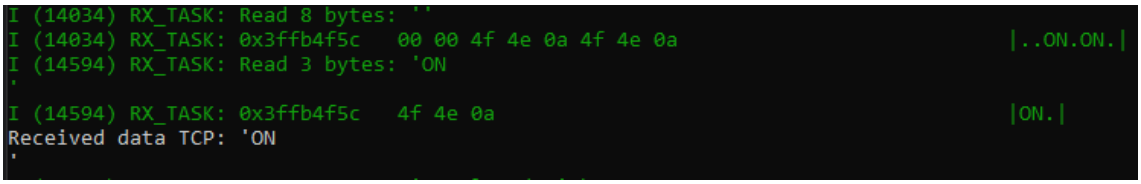
Funkce *uart_param_config()* slouží k nastavení parametrů UART pro zvolené číslo rozhraní. Pro účely této práce je funkce volána s argumentem UART_NUM_2, který určuje číslo rozhraní. Druhým argumentem je ukazatel na předem nadefinovanou strukturu.

Poslední funkcí při inicializaci UART komunikace je *uart_set_pin*(). Tato funkce slouží k nastavení pinů pro přenos dat pomocí UART. Konkrétně se funkce používá k nastavení TX a RX pinů, které jsou propojeny s příslušnými piny na zařízení, se kterým probíhá komunikace.

Takto inicializovaná periferie UART je připravena pro příjem dat. Dále byla vytvořena funkce *rx_task*(), která přečte data a zpracuje přijatá data. Tato třída obsahuje mnoho metod, avšak hlavní funkcí je *uart_read_bytes*(), která slouží k asynchronnímu příjmu dat. Pro správné přečtení dat jsou potřeba 4 parametry:

- UART_NUM_2 - identifikátor UART linky, ze které se mají data přijímat
- &data[index] - adresa v paměti, kam se má přijatý byte uložit
- 1 - počet bajtů, které se mají přijmout (v tomto případě jeden byte)
- pdMS_TO_TICKS(100) / portTICK_PERIOD_MS - timeout pro čekání na data, v tomto případě 100 milisekund.

Na obr č. 27 jsou zobrazena přijatá data ze sériové komunikace



Obr č. 27 Příjem dat z UART linky

Inicializace Ethernetu

Ethernet komunikaci je nutné inicializovat hned při připojení mikrokontroleru a získaní IP adresu pomocí síťového prvku, například routeru. Pro Ethernet byl vytvořen dedikovaný soubor *ethernet.c* obsahující všechny potřebné funkce pro vytvoření spojení a získání IP adresy. Samotná inicializace je volána v hlavním souboru aplikace. Inicializace probíhá pomocí tří funkcí, které jsou popsány níže. V rámci monitorování mikro kontrolérů byla vytvořena proměnná:

```
static const char *TAG = "eth";
```

, která vypisuje informace o stavu Ethernetového připojení.

První funkcí v souboru *ethernet.c* je

```
static void eth_event_handler(void *arg, esp_event_base_t event_base,
int32_t event_id, void *event_data)
```

, která slouží jako obslužná rutina pro události související s Ethernetovým rozhraním. Tato funkce zpracovává události, které jsou vyvolány frameworkem, jako například připojení nebo odpojení Ethernetového kabelu. Funkce má několik argumentů, které jsou potřeba dodržet. Prvním argumentem je ukazatel na data, která jsou předávaná jako parametr při registraci. Dalšími argumenty jsou event_base, event_id a event_data, které označují základní typ události, ID konkrétní události a ukazatel na data, která jsou spojená s událostí. Všechny vyvolané události jsou zpracovány pomocí struktury switch, kde je každá událost identifikována odpovídajícím příkazem case. Následně se z těchto case volají další funkce pro zpracování nastalé události.

Druhou funkcí v souboru ethernet.c je

```
void got_ip_event_handler(
void *arg, esp_event_base_t event_base, int32_t event_id, void
*event_data)
```

, která vytváří obslužnou rutinu pro přidělení IP adresy síťovému rozhraní. Tato funkce zpracovává události, které jsou vyvolány ESP-IDF frameworkem po úspěšném připojení zařízení k síti a získání IP adresy. Podobně jako u předchozí metody i tato má argumenty, které odpovídají funkcím *esp_event_handler_register()*.

Nejdůležitější funkcí ze souboru je ovšem funkce *ethernet_init()*, která inicializuje Ethernetové rozhraní a vrátí příznak, zda je Ethernet aktivní či nikoliv. Pro zajištění správného chodu programu některé funkce používají makro *ESP_ERROR_CHECK()*, které zkontroluje, zda funkce vrátila korektní hodnotu a v případě chyby ukončí program. V případě úspěchu funkce vrátí hodnotu *ESP_ERR_OK* a v případě nedostatku paměti pro inicializaci síťového rozhraní vrátí *ESP_ERR_NO_MEM*. V rámci tohoto makra je volána funkce *esp_netif_init()*, která má za úkol inicializovat všechny potřebné komponenty a prostředky pro správný chod síťového rozhraní. Tento krok je velmi důležitý, aby bylo možné používat různé síťové protokoly, jako jsou například TCP/IP nebo UDP.

```
esp_netif_config_t cfg = ESP_NETIF_DEFAULT_ETH();
    esp_netif_t *eth_netif = esp_netif_new(&cfg);

    // Init MAC and PHY configs to default
    eth_mac_config_t mac_config = ETH_MAC_DEFAULT_CONFIG();
    eth_phy_config_t phy_config = ETH_PHY_DEFAULT_CONFIG();
```

Funkce *esp_event_loop_create_default()* také využívá makro pro kontrolu úspěšnosti. Slouží k vytvoření výchozí smyčky pro zpracování událostí. Jestliže tato smyčka není vytvořena, není možné přijímat události od různých periférií a modulů připojených k ESP32. Smyčka se používá pro svou činnost nízko úrovňové funkce z knihovny „*esp_event_loop.h*“. Funkce vrací ukazatel na vytvořenou smyčku událostí.

Pokud výsledek funkce není úspěšný, vrátí hodnotu NULL. Po úspěšném vytvoření smyčky je nutné registrovat obslužné funkce pro jednotlivé události, které budou zpracovány.

Poté následuje vytvoření konfigurační struktury *esp_netif_config_t* s výchozími hodnotami pro Ethernet, které získá pomocí funkce *ESP_NETIF_DEFAULT_ETH()*. Následně se vytvoří nová instance síťového rozhraní pomocí funkce *esp_netif_new()* a nastaví se MAC a PHY konfigurace na výchozí hodnoty pomocí funkcí *ETH_MAC_DEFAULT_CONFIG()* a *ETH_PHY_DEFAULT_CONFIG()*.

Dále se provádí inicializace konfigurační struktury pro Ethernet MAC a PHY na jejich výchozí hodnoty pomocí funkcí *ETH_MAC_DEFAULT_CONFIG()* a *ETH_PHY_DEFAULT_CONFIG()*. Tyto hodnoty jsou součástí hardwaru, který se stará o přenos dat po Ethernetové síti. MAC (Media Access Control) řídí přístup k médiu a PHY (Physical Layer) zajišťuje fyzické spojení s Ethernetovým kabelem a nastavuje jeho přenosové parametry.

Po této konfiguraci následuje funkce

```
ESP_ERROR_CHECK(esp_eth_driver_install(&config, &eth_handle));
```

, která inicializuje Ethernetový ovladač. Jejími vstupními parametry jsou ukazatel na konfigurační strukturu *esp_eth_mac_t* a proměnná typu *esp_eth_handle_t*, která slouží k manipulaci s Ethernetovým rozhraním. Po zavolání této funkce se nejprve inicializuje MAC podle zadané konfigurace a poté se inicializuje Ethernetové rozhraní s touto konfigurací. Pokud inicializace proběhne bez problémů, funkce vrátí hodnotu *EPS_OK*. V případě chyby funkce vrátí odpovídající kód chyby a vyvolá se výjimka s odpovídajícím chybovým hlášením.

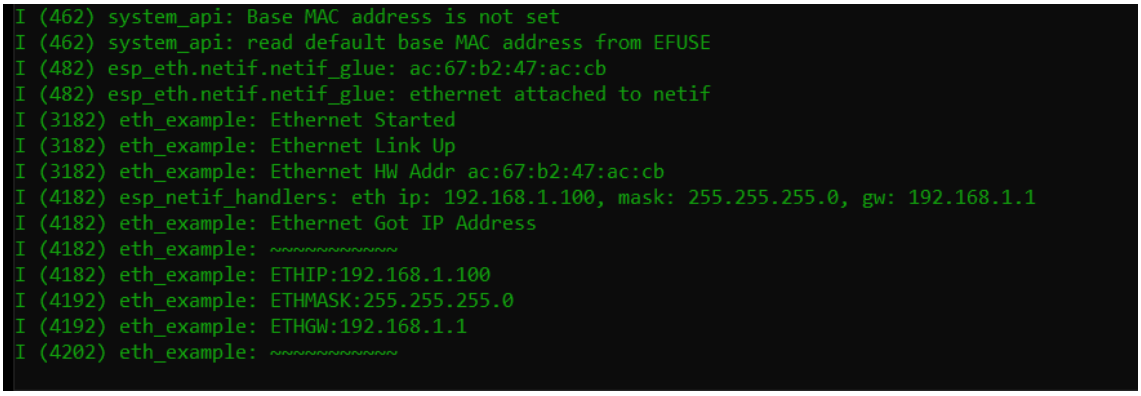
Následuje řádek

```
ESP_ERROR_CHECK(
esp_netif_attach(eth_netif, esp_eth_new_netif_glue(eth_handle)));
```

, kterým se aplikace snaží připojit rozhraní k TCP/IP stacku. Tato funkce přijímá dva parametry: ukazatel na Ethernetové rozhraní a ukazatel na funkci, která vrací instanci struktury. Tento krok je nezbytný pro používání TCP/IP komunikace s rozhraním. Další dva řádky slouží k registraci uživatelsky definovaných event handlerů pro události spojené s Ethernetovým rozhraním.

Funkce *esp_event_handler_register()* se používá k registraci funkce, která bude zpracovávat události daného typu a identifikátoru. V tomto případě jsou zaregistrovány funkce *eth_event_handler()* pro události ETH_EVENT a *got_ip_event_handler()* pro události IP_EVENT_ETH_GOT_IP. Následující řádek spouští stavový automat ethernetového rozhraní pomocí funkce *esp_eth_start()*. Tento krok je důležitý pro aktivní připojování a odpojování rozhraní ze sítě a pro možnost odesílání a přijímání dat."

Přiklad úspěšného inicializování Ethernetu a přidělení IP adresy je zobrazen na obr č. 28.



Obr č. 28 Inicializace Ethernetu a získání IP adresy

Funkce pro SSL/TLS komunikaci

Popisuje funkce, která vytváří vlákno, jak už bylo výše uvedeno. Vlákno pravidelně odesílá TLS zabezpečený paket na určenou IP adresu a port. Výchozí číslo portu je 8080. Nicméně číslo portu se muže změnit, dle nastavení uživatele.

Nejprve se nastaví konfigurace pro TLS spojení, aby se neprováděla kontrola Common Name v rámci certifikátu. Dále jsou popsány jednotlivé metody, které byly použity při inicializaci TLS spojení.

Jako první probíhá inicializace struktury pro konfiguraci TLS spojení pomocí příkazu:

```
esp_tls_cfg_t config = {
        .skip_common_name = true};
```

Tato proměnná se používá pro nastavení různých parametrů spojení. Konkrétně v tomto případě nastavuje parametr *skip_common_name* na hodnotu *true*. Tento parametr určuje, zda bude TLS knihovna ověřovat Common Name (CN) certifikátu serveru. Pokud je tento parametr nastavený na hodnotu *true*, nebude TLS knihovna kontrolovat, zda se CN v certifikátu shoduje s očekávaným serverem. Toto nastavení se používá pro případy, kdy CN v certifikátu nemusí být stejná jako adresa serveru, se kterým se spojujeme. Například může být použita IP adresa místo doménového jména.

Dále je použita funkce *esp_tls_init(),* která inicializuje SSL/TLS knihovnu a vytváří nový SSL/TLS kontext, který se následně používá pro komunikaci se vzdáleným serverem. Funkce vrací ukazatel na strukturu *esp_tls_t*, která obsahuje informace o daném SSL/TLS kontextu. Tato struktura se následně používá pro další volání knihovny pro odeslání či příjem dat přes SSL/TLS spojení.

Pro inicializování nového spojeni použijeme funkci

```
int result = esp_tls_conn_new_sync(APP_PYTHON, 14, PORT_TLS, &config,
esp_tls_handle);
```

, která jako vstupní parametry přijímá IP adresu (tzn. adresu python aplikace) a port, který je defaultně nastaven na hodnotu 8080. Tato funkce je blokující, což znamená, že pokud se spojení nenaváže, program se zastaví a čeká. Pokud se spojení úspěšně

naváže (obr č. 29), funkce vrátí hodnotu 0. Pokud však nastane chyba (obr č. 30), vrátí se záporná hodnota. V této hodnotě je uložena informace o druhu a příčině chyby, která je dále zpracována a logována pomocí funkce *esp_tls_conn_error()*. Výstup metody je uložen do proměnné *result* a dále vyhodnocován.

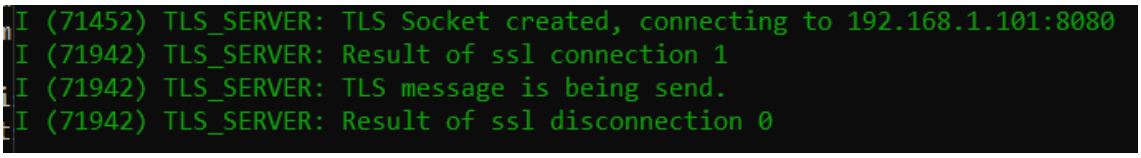
Pokud je výsledek záporný, vytvoří se Log zpráva s chybovým hlášením:

```
ESP_LOGE(TAG, "TLS with %s:%d could not be established", APP_PYTHON, SERVER_PORT_SSL);
```

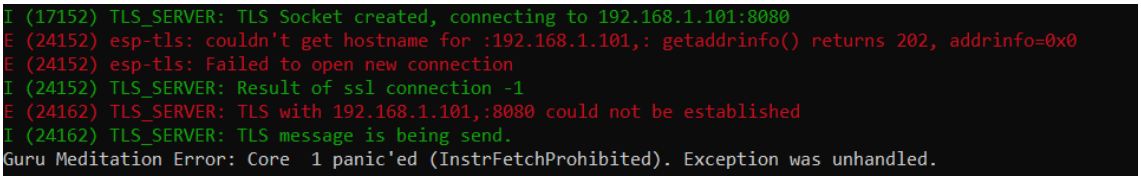
Jeli, ale vrácena jiná hodnota, komunikace pokračuje dále a přechází na funkci, která již posílá zašifrovanou zprávu. Procedura je podobná jako při UART komunikaci, ale probíhá pomocí TLS spojení. Odesílá se řetězec „ON\n", který je uložen do proměnné *message*. Funkce *esp_tls_conn_write()* odesílá data přes TLS spojení a má následující parametry: *esp_tls_handle* jako ukazatel na strukturu, která obsahuje informaci o TLS spojení, *message* jako ukazatel na buffer odesílající data a strlen(message), který udává velikost odesílaných dat.

```
char * message = "ON";
        esp_tls_conn_write(esp_tls_handle, message, strlen(message));
```

Po odeslání zprávy se volá funkce *esp_tls_conn_destroy*, která ukončí TLS spojení. Po dokončení komunikace je nutné spojení uzavřít, aby se uvolnily alokované prostředky a zabránilo se nežádoucímu chování programu. Volaná funkce toho docílí a výsledek uloží do proměnné *result*, která je následně předána do funkce *ESP_LOG* pro zaznamenání informace.



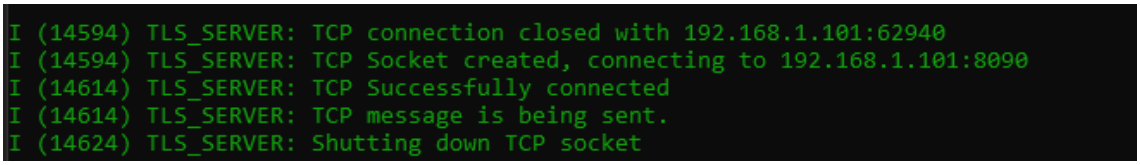
Obr č. 29 Úspěšné poslání TLS paketu



Obr č. 30 Neúspěšné zaslání TLS zprávy

Funkce pro odesílání TCP komunikace

Pro možnost volby komunikace pomocí protokolu TLS nebo TCP byla vytvořena funkce *tcp_send_task()*. Tato funkce nastavuje podobné parametry jako funkce pro inicializaci UART komunikace, s tím rozdílem, že komunikace probíhá výchozím portem 8090 a neslouží k příjmu dat, pouze k odesílání. Funkce *tcp_send_task()* spouští TCP komunikaci, připojuje se k zadané IP adrese a portu, odesílá zadaná data a následně ukončuje komunikaci. Na obr č. 31 je výpis konzole při úspěšném vytvoření, odeslání a ukončení TCP komunikace.



Obr č. 31 Úspěšné zaslání TCP zprávy

Funkce pro zpracování příchozí TCP komunikace

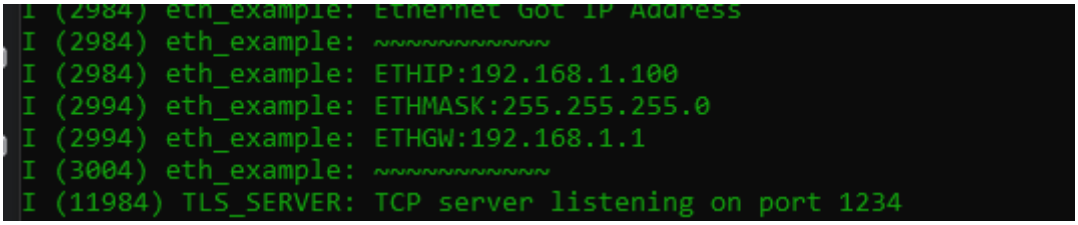
V předchozích kapitolách byla popsána inicializace Ethernetu, rozhraní UART a vytvoření funkcí pro TCP a TLS komunikaci. Nyní je potřeba všechny funkce spojit do jednoho tak, aby byla zajištěna funkčnost celého programu a bylo tak splněno zadání diplomové práce.

V hlavním programu jsme tedy vytvořili novou úlohu pomocí funkce *xTaskCreate,* která volá funkci s názvem *tcp_receive_task* a je spuštěna pouze tehdy, jestliže je inicializování Ethernetu úspěšné a připojení k Ethernetu aktivní.

Pro úspěšné vytvoření úlohy je zapotřebí zavolat funkci *xTackCreate* s pěti parametry:

- tcp_receive_task: název funkce, která bude provádět danou úlohu
- "tcp_receive_task": název úlohy, která bude vytvořena
- 8192: velikost zásobníku v bytech, který bude alokován pro tuto úlohu
- NULL: ukazatel na argumenty, které budou předány funkci při jejím spuštění (v tomto případě nejsou argumenty potřebné, takže je nastaven na NULL)
- 5: priorita úlohy (od 0 do 24, kde nižší číslo značí vyšší prioritu)
- NULL: ukazatel na proměnnou, do které bude uloženo ID vytvořené úlohy (v tomto případě není potřeba, takže je nastaven na NULL)

Úloha naslouchá na portu 1234, který je nastaven pomocí funkce PORT_SET. Na obr č. 32 je snímek konzole úlohy čekající na příchozí zprávy. Funkce dokáže zpracovat dvě zprávy: SET a RUN.



Obr č. 32 Úspěšné vytvoření úlohy

Pomocí následujícího kódu bude popsán způsob implementace tohoto procesu.

```
struct sockaddr_in remote_addr;
    socklen_t remote_addr_len = sizeof(remote_addr);
```

Následující kód inicializuje strukturu *remote_addr* typu *sockaddr_in,* která slouží k uchování adresy vzdáleného klienta, který se připojí k serveru. Tato struktura obsahuje informace o síťové adrese v IPv4 formátu. Dále je vytvořena

proměnná *remote_addr_len* typu *socklen_t*, která uchovává délku struktury. Tuto proměnnou lze použít jako parametr pro funkce, které vyžadují délku struktury, například pro funkci *lwip_accept()*. Tímto způsobem se zajistí, že funkce bude pracovat s platnými daty a nedojde k překročení hranic paměti.

```
int sock = lwip_socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Tento řádek kódu vytváří nový síťový socket pro TCP spojení. Konkrétně se vytváří socket pro rodinu protokolů IPv4 (*AF_INET*), pro streamový typ socketu (*SOCK_STREAM*) a s protokolem TCP (*IPPROTO_TCP*). Funkce *lwip_socket()* je funkce z knihovny LWIP, která se používá pro vytvoření nového socketu. Návratovou hodnotou této funkce je deskriptor nového socketu, který se ukládá do proměnné sock. Po uložení následuje jeho vyhodnocení. Pokud je hodnota menší než 0, vytvoření socketu selže.

```
struct sockaddr_in server_addr = {
        .sin_family = AF_INET,
        .sin_addr.s_addr = htonl(INADDR_ANY),
        .sin_port = htons(PORT_SET),
    };
```

Kód inicializuje strukturu sockaddr_in s názvem server_address, která se používá k určení adresy a portu, na kterém bude naslouchat serverová aplikace. Inicializují se tři položky:

- *sin_family* - určuje rodinu protokolu pro použití serverové aplikace.
- *sin_addr.s_addr* - určuje IP adresu, na které bude naslouchat server. Nastavena je na INADDR_ANY, což znamená, že server naslouchá na všech dostupných IP adresách.
- *sin_port* - určuje číslo portu, na kterém bude naslouchat server.

Po inicializaci struktury *server_addr*, je nutné provést přiřazení socketu k určitému IP a portu pomocí

```
funkcelwip_bind(sock,       (struct       sockaddr       *)&server_addr,
sizeof(server_addr))
```

Tato funkce přijímá jako parametry *socket*, strukturu obsahující IP adresu a port pro naslouchání. Jelikož tato funkce akceptuje libovolnou adresu *socketu*, je nutné jako poslední parametr předat velikost této struktury v bytech. Pokud se navázání nezdaří a funkce vrátí zápornou hodnotu, aplikace vypíše chybovou hlášku, uzavře *socket* a ukončí běžící úlohu.

Nakonec je potřeba zavolat funkci *lwip_listen(sock, backlog),* která způsobí, že se začne naslouchat příchozím spojením na daném socketu „*sock*“. Backlog určuje maximální počet připojení, které mohou být ve frontě na zpracování. Jestliže operace selže, funkce vrátí zápornou hodnotu a vypíše se chybová hláška. Pokud vše proběhne v pořádku, aplikace je připravena přijímat připojení od klientů.

Následuje vytvoření nekonečné smyčky, která bude zpracovávat příchozí TCP pakety a nastavovat hodnoty nebo posílat zpětnou komunikaci. Jednotlivé příkazy TCP paketů s hodnotami a následným zpracováním jsou zde popsány. Pokud přijde

paket, který nebude obsahovat žádnou z hlídaných hodnot, není zpracován a nevykoná se žádná akce.
Následující pakety jsou zpracovány:

- První paket musí na začátku obsahovat slovo „SET“. Součástí přijaté zprávy je IP adresa a port, Baud rate a typ komunikace. IP adresa určuje, kam se mají posílat zprávy. Port je určen pro danou komunikaci a je rovněž součástí přijatého paketu. Nakonec se nastaví Baud rate pro UART komunikaci. Pro potvrzení, že aplikace obdržela zprávu, se odešle odpověď s potvrzením, že hodnoty byly nakonfigurovány.
- Druhý zpracovávaný packet musí obsahovat zprávu „RUN“. Po přijetí této zprávy se spustí úloha, která přijme data z UART komunikace a vytvoří komunikaci podle nastaveného typu. Jestliže před touto zprávou nebyla přijata zpráva „SET“, použijí se defaultní nastavené hodnoty. Poté se odešle odpověď „Communication started“ pro potvrzení odeslání a spuštění úlohy.

6.2.3 Aplikace na PC v Pythonu

Pro vytvoření aplikace v jazyce Python byla použita verze 3.8.6. Pro vytvoření aplikace bylo nezbytné doinstalovat balíčky, jako například pro vytvoření grafického rozhraní nebo pro komunikaci se sériovým portem. Pro správu balíčků a vytvoření izolovaného virtuálního prostředí byl použit nástroj pipenv.

6.2.3.1 Vytvoření API rozhraní

V rámci zadání práce byl stanoven požadavek, aby měl uživatel možnost při komunikaci pomocí Ethernetu nastavit parametry jako je přenosová rychlost či IP adresa python aplikace.

Pro splnění požadavků na GUI byla použita knihovna PyQt5 a vytvořena třída s názvem „*App*“, která dědí metody z třídy *QWidget*. V konstruktoru třídy „*App*“ byl volán

```
super()._init__()
```

, což znamená, že byl volán konstruktor rodičovské třídy. Tento příkaz vytváří proxy objekt, který umožňuje volání metod z třídy *QWidget*. Pokud by byla potřeba specifikovat konkrétní třídu, můžete být předán název třídy a *self* jako argumenty. Použití *super()* je v Pythonu běžné při metodách tříd, které dědí z jiných tříd. Toto volání způsobí přepsání konstruktoru rodičovské třídy, ale přitom si zachová některé části konstruktoru. Všechny metody se budou volat pomocí *self* například *self.title*.

V konstruktoru třídy „*App*“ jsem vytvořila potřebné proměnné, které jsou volány i v rámci jiných funkcí.

Pro vytvoření *GUI* jsem využila metody z knihovny *PyQt5* a vytvořila funkci s názvem „*GUI*“, která obsahuje všechny potřebné prvky pro splnění požadavků zadání práce. V této funkci se tvoří a umísťují jednotlivé prvky, které slouží ke vstupu

a zobrazení dat.

Každá metoda vytváří jeden prvek a určuje jeho umístění a vlastnosti, jako například velikost, popisek, šířku okna a další. Po vytvoření prvku se nastavují jeho vlastnosti a přidávají se do okna aplikace. Např. pro vytvoření tlačítka se použije metoda *QPushButton*, která je vytvoří s popiskem a nastaví jeho velikost a pozici v okně.
Metoda *self.setWindowTitle(self.title)* z třídy *QWidget*, slouží k nastavení titulku okna. V tomto konkrétním případě se používá hodnota *self.title*, která byla definována jako vlastnost třídy „*App*“.
Metoda *self.setGeometry(self.left, self.top, self.width, self.height)* je rovněž ze třídy *QWidget*, která slouží k nastavení umístění a rozměrů hlavního okna. K tomuto účelu byly použity proměnné *self.left, self.top, self.width* a *self.height*, které byly definovány jako vlastnosti třídy „App“. Tuto metodu je potřeba použít, aby se grafické okno zobrazilo s požadovanými rozměry a umístěním na obrazovce.

Následně byly vytvořeny jednotlivé popisky pomocí prvku QLabel. Příklad zobrazení popisku pro rychlost přenosu (Baud rate) je v následujícím kódu.

```
self.baudrate_label = QLabel(self)
self.baudrate_label.setText('Baudrate:')
      self.baudrate_label.move(50, 50)
```

První řádek vytvoří nový popisek a uloží jej do proměnné *self.baudrate_label*. Argument *self* určuje, že tento popisek je potomkem hlavního okna aplikace. Metoda *setText()* nastaví testový řetězec, který bude zobrazen na popisku. V tomhle příkladu se nastaví text „Baudrate:“. Poslední metoda je *move(),* která nastaví pozici popisku na okně. V tomto případě se nastaví (50, 50), což znamená, že popisek bude umístěn 50 pixelů od levého horního rohu ona a 50 pixelů od horního okraje okna. Na obr č. 33 je zobrazení jednotlivých popisků v aplikaci.

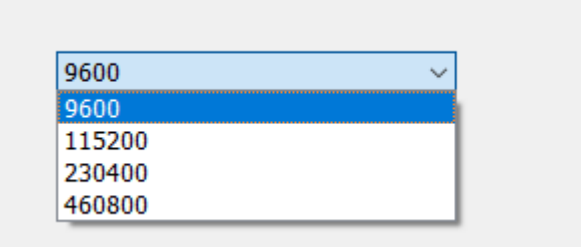
Obr č. 33 Použití *QLabel* widget

Dále byl použit widget QComboBox pro výběr přenosové rychlosti.

```
self.baudrate_entry = QComboBox(self)
self.baudrate_entry.setGeometry(150, 50, 200, 20)
self.baudrate_entry.addItems(["9600", "115200", "230400", "460800"])
```

Metoda setGeometry() nastavuje pozici a velikost tohoto widgetu v rámci hlavního okna. Argumenty této metody jsou x-ová a y-ová souřadnice levého horního rohu widgetu, jeho šířka a výška. Metoda addItems() přidává položky do tohoto combo boxu.

V našem případě jsou to konkrétní rychlosti pro sériovou komunikaci: 4800, 9600, 19200, 38400 a 115200. Tento prvek bude sloužit k výběru požadované rychlosti a následně bude tato hodnota použita pro inicializaci sériového portu. Výsledné rozložení widgetů je zobrazeno na obr č. 34.



Obr č. 34 Použití QComboBox widget

Pro vytvoření prvků pro zadávání IP adresy byl použit widget QLineEdit. Níže je ukázka vytvoření prvku pro zadávání prvního bytu IP adresy:

```
self.ip_address_entry = QLineEdit(self)
self.ip_address_entry.setFixedWidth(40)
self.ip_address_entry.setValidator(QIntValidator(0, 255, self))
self.ip_address_entry.move(150, 100)
self.ip_address_entry.setText("192")
```

Metoda *setFixedWidth(40)* nastavuje pevnou šířku pole na 40 pixelů. Metoda *setValidator(QIntValidator(0, 255, self))* nastavuje validátor pro toto pole, který omezuje uživatele na zadávání čísel v rozsahu od 0 do 255. Poslední řádek *setText("192")* nastavuje výchozí hodnotu na 192. Jelikož se IP adresa skládá ze 4 bytů, byly tyto prvky vytvořeny 4, kde každý z nich znázorňuje jeden byte. Výsledné rozložení prvků pro zadávání celé IP adresy je zobrazeno na obr č. 35.

Obr č. 35 Použití QLineEdit windget

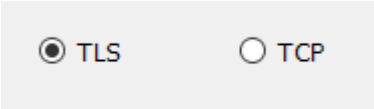
Obdobě je nastavený prvek pro zadání komunikačního portu. Jedná změna je, že prvek je omezený od 0 do 100 000.

Jako poslední parametr v GUI jsem se rozhodla přidat typ komunikace, která má být měřena. I když v zadání bylo řečeno, že bude práce zaměřena pouze na šifrovanou komunikaci, z důvodů vyšší variability jsem přidala dva radio buttony pomocí třídy QRadioButton. Tyto tlačítka umožňují uživateli vybrat mezi komunikací typu TCP a TLS. Pro každé radio tlačítko byl použit následující kód:

```
self.radio_button1 = QRadioButton("TLS", self)
self.radio_button1.move(50, 200)
self.radio_button1.setChecked(True)
```

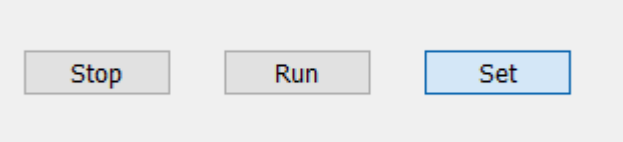
Tento kód vytváří radio tlačítko s popiskem „TLS“ a umisťuje ho na pozici x = 50 a y = 200 v okně, které je prezentováno objektem *self*. Metodou *setChecked(True)* bylo tlačítko označeno jako výchozí výběr, který se zobrazí při spuštění aplikace. Stejným způsobem bylo přidáno radio tlačítko pro komunikaci typu „TCP“.

Nakonec byly oba radio buttony spojeny do skupiny pomocí třídy QButtonGroup, metodou *addButton(),* která zajišťuje, že uživatel může zvolit pouze jednu z několika možností. Argumenty metody jsou objekt tlačítka a celočíselná hodnota, kterou tlačítko reprezentuje v rámci skupiny. Tato hodnota se použije jako identifikátor pro dané tlačítko v rámci skupiny. Celý výsledek můžeme vidět na obr č. 36.



Obr č. 36 Použití *QRadioButton* a *QButtonGroup* widget

Nyní je potřeba všechny hodnoty uložit a zpracovat, aby s nimi mohlo být dále pracováno. Proto vznikla tři tlačítka. Pro vytvoření tlačítek byl použit QPushButton, který nastavuje tlačítka na „SET“, „RUN“ a „STOP“. Po stisku jednotlivých tlačítek jsou volány různé funkce. Tlačítka jsou na obr č. 37.

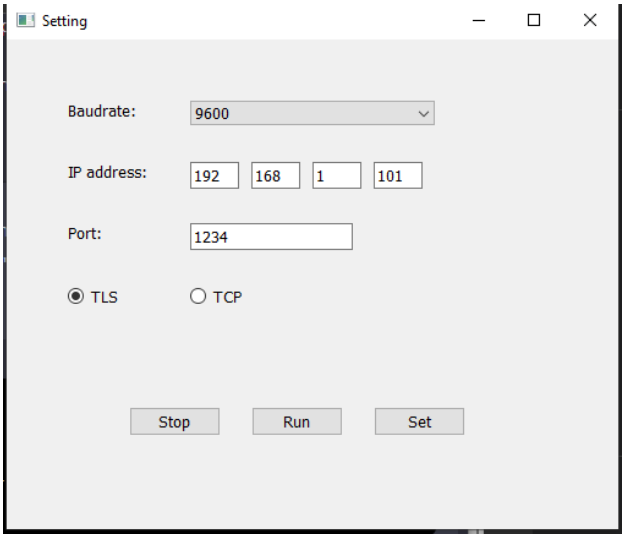


Obr č. 37 Použití *QPushButton* widget

Jednotlivé funkce, které jsou volány, jsou následující:

- Funkce „configuration“: Tato funkce se spustí po stisknutí tlačítka „SET“ a slouží k nastavení parametrů pro komunikaci. Nejprve se zkontroluje hodnota IP adresy. Pokud je zadaná hodnota větší než 256 a zároveň je hodnota rovna „“, tak se doplní nulou. Výsledná IP adresa se uloží do proměnné *self.ip_address*. Dále se zkontroluje, zda zadaný port není prázdný řetězec a uloží se do proměnné *self.port*. Ukládá se i hodnota Baud rate do proměnné self.baudrate a vybraný radio button do proměnné *self.type_comunication*. Všechny tyto proměnné jsou následně používány v ostatních funkcích. Zároveň funkce odešle zprávy i na ostatní zařízení, že se mají překonfigurovat. Funkce budou popsány později.
- Funkce „run“: Tato funkce se spustí po stisknutí tlačítka „RUN“. Pokud uživatel nezadal žádné hodnoty pro komunikaci, aplikace ho varuje, že bude pracovat s výchozími. Poté spustí komunikaci s ostatními vývojovými kity v souladu s vybraným komunikačním protokolem. Podrobnosti budou popsány níže v rámci popisu komunikace s ostatními deskami.
- Funkce „stop“: Tato funkce se zavolá po stisknutí tlačítka „STOP“ a slouží k ukončení celé aplikace.

Výsledné GUI je zobrazeno na obr č. 38.



Obr č. 38 Výsledné GUI ovládací aplikace

UART komunikace

Pro komunikaci pomocí UART existují dvě funkce, které umožňují odesílat data. V kódu je nastaveno, že všechna data se budou posílat na COM8, kde se nachází zařízení „*USB to UART*“, které data přijímá pomocí sériové komunikace.

První funkce, která byla vytvořena, se spouští při stisknutí tlačítka „SET“ a nazývá se *self.configuration_UART*. Jako parametr přijímá novou hodnotu přenosové rychlosti. Po jejím zavolání se otevře port COM8 s přenosovou rychlostí 9600 a s časem 1 s, kdy čeká na odpověď a nová hodnota se pošle na tento port. Tuto hodnotu bude následně používat pro přenos informací. Nakonec se port zavře.

Druhá funkce se nazývá *self.send_to_com_port* a reaguje na stisknutí tlačítka „RUN“. Tato funkce přijímá data, která má odeslat na COM port s aktuální přenosovou rychlostí. Po odeslání dat na straně Arduina se měření času ukončí.

Ukázku Kódu pro UART komunikaci níže

```
def send_to_com_port(self, data):
        com_port = serial.Serial('COM8', self.baudrate)
        com_port.write((data+"\n").encode())
        com_port.close()
```

TCP komunikace

V rámci TCP komunikace byly vytvořeny celkem 4 funkce. Jedna z nich reaguje po stisknutí tlačítka „SET“. Pokud k tomu dojde, pošle se konfigurační paket s novou IP adresou a portem a poté port uzavře.

Všechny další TCP pakety jsou odeslány až po stisknutí tlačítka „RUN“. Nejprve se odešle paket s výzvou „RUN" na ESP32. Podle nastavení typu komunikace se ověří socket pro TCP nebo TLS komunikaci a čeká se na příchozí zprávu. Po přijetí se komunikace uzavře. Zde je ukázka tvorby TCP paketu v jazyce Python:

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((TCP_IP, self.port))
```

```
        message = "RUN"
        s.send(message.encode())

        data = s.recv(1024)
        print(data.decode())
        data = 0
        s.close()
```

Příklad kódu vytváří TCP soket pro komunikaci s danou IP adresou a portem. Poté se odesílá zpráva „RUN“ na server pomocí metody *socket.send()*. Program čeká na odpověď ze serveru pomocí *socket.recv()* a uloží ji do proměnné data, kterou vypíše. Nakonec se soket uzavře pomocí *socket.close()*.

TLS komunikace

Než je možné otevřít port pro TLS komunikaci, je nutné vygenerovat certifikát a privátní klíč. Způsob, jakým byl certifikát vygenerován, je popsán níže. Pro příjem šifrovaného paketu bylo nejprve zapotřebí použít funkci *ssl.create_default_context()*, která vytváří SSL/TLS kontext pro komunikaci s SSL/TLS servery. Kontext je objekt, který obsahuje informace o zabezpečené komunikaci, jako jsou certifikáty a klíče, protokoly, šifry atd. Pro vytvoření kontextu byl použit jako parametr *ssl.Purpose.CLIENT_AUTH*, což značí, že tento kontext je určen pro klienta, který se autentizuje u serveru pomocí certifikátu. Kontext bude obsahovat výchozí nastavení, které lze dále upravovat. V našem případě je tento kontext uložen do proměnné *context*.

Následuje volání metody *context.load_cert_chain()*, která slouží k načtení certifikátu a privátního klíče pro použití šifrované komunikace. Soubory budou načteny a použity pro autentizaci serveru a šifrované komunikace s klientem.

Po načtení certifikátu a privátního klíče následuje nastavení kontextu pomocí atributu context.verify_flags na hodnotu ssl.VERIFY_X509_STRICT. Toto znamená, že ověřování certifikátu se provede pomocí přísných pravidel pro ověření platnosti certifikátu. Tato volba zahrnuje ověření platnosti certifikátu, platnosti řetězce certifikátů a ověření podpisu certifikátu. Pokud ověření selže, spojení bude ukončeno.

Následuje skupina kódu, který vytváří soket pro naslouchání na určitém síťovém portu a přiřazuje mu daný port. Konkrétně je vytvořen soket typu „AF_INET“, který značí použití IPv4 protokolu a typ „SOCK_STREAM“, což znamená, že je použit streamový protokol TCP pro spolehlivou komunikaci.

Poté je volána metoda *bind()* na tomto soketu, které jsou předány dva parametry. Prvním je prázdný řetězec pro vazbu na všechna síťová rozhraní a druhým parametrem je číslo 8080, které specifikuje číslo portu, na kterém bude soket naslouchat příchozím spojením.

Nakonec je volána metoda *listen()* s argumentem 5. Argument značí, že soket bude naslouchat až na 5 spojení ve frontě, pokud již bude soket obsazen dalšími spojeními. Následuje tento kód, který představuje zpracování nového připojení TCP klienta na daném portu:

```
newsocket, fromaddr = bindsocket.accept()
        connstream = context.wrap_socket(newsocket, server_side=True)
        data = connstream.recv(1024)
        print (data)
        connstream.shutdown(socket.SHUT_RDWR)
        connstream.close()
        self.send_to_com_port(data)
```

Nejprve se pomocí metody *accept()* na řádku 1 přijme nové připojení a čeká se na vstup dat. Funkce *accept()* vrací nový soket a adresu klienta, které jsou uloženy do proměnných newsocket a fromaddr. Na řádku 2 se pomocí metody *wrap_socket()* vytvoří nový kontext s použitím SSL a vytvoří se šifrovaný soket pro komunikaci s klientem (connstream). Poté se na řádku 3 pomocí metody *recv()* přijmou data ze soketu do proměnné data a vypíší se na obrazovku pomocí funkce *print()*. Poté se spojení uzavře pomocí metod *shutdown()* a *close()* a získaná data se předají na sériovou komunikaci pomocí metody *send_to_com_port()*.

Vygenerování certifikátu a klíče

Pro vygenerování certifikátu a primárního klíče jsem použila Pythonovský skript a knihovny OpenSSL. Nyní budou popsány jednotlivé kroky, které vedly k úspěšnému vygenerování certifikátu a primárního klíče.

Nejprve jsem vytvořila nový objekt pomocí metody PKey z knihovny crypto, který bude reprezentovat RSA klíč. Poté byla volána metoda generate_key() na tomto objektu a předán typ klíče crypto.TYPE_RSA s délku 2048 bitů. Tímto krokem jsem vytvořila nový RSA klíč, který byl uložen do proměnné „key“.

Následující kód slouží k vytvoření samo podepsaného certifikátu X.509, který je standardem pro digitální certifikáty používané k zabezpečení webových stránek, emailových komunikací, šifrování dat atd.:

```
cert = crypto.X509()
cert.get_subject().CN = "My Server"
cert.set_serial_number(1000)
cert.gmtime_adj_notBefore(0)
cert.gmtime_adj_notAfter(365 * 24 * 60 * 60)
cert.set_issuer(cert.get_subject())
cert.set_pubkey(key)
cert.sign(key, 'sha256')
```

V první řadě se vytvoří objekt cert, který reprezentuje certifikát. Metoda *get_subject()* vrátí objekt s informacemi o vlastníkovi certifikátu, do kterého se následně vloží Common Name (CN), který v tomto případě byl nastaven na „My Server“. Další řádky kódu nastavují sériové číslo certifikátu, platnost certifikátu (od okamžiku vytvoření v délce až jednoho roku) a vydavatel certifikátu (v tomto případě se nastaví stejné jako vlastník). Poté se do certifikátu vloží veřejný klíč pomocí metody *set_pubkey()*. Nakonec se certifikát podepíše pomocí privátního klíče zadaného metodou *sign()* s použitím algoritmu SHA-256.

Nakonec byl vygenerovaný certifikát a primární klíč uložen do souborů server_cert.pem *a* server_key.pem.

To bylo provedeno pomocí následujících příkazů:

```
with open("server_key.pem", "wb") as f:
    f.write(crypto.dump_privatekey(crypto.FILETYPE_PEM, key))

with open("server_cert.pem", "wb") as f:
    f.write(crypto.dump_certificate(crypto.FILETYPE_PEM, cert))
```

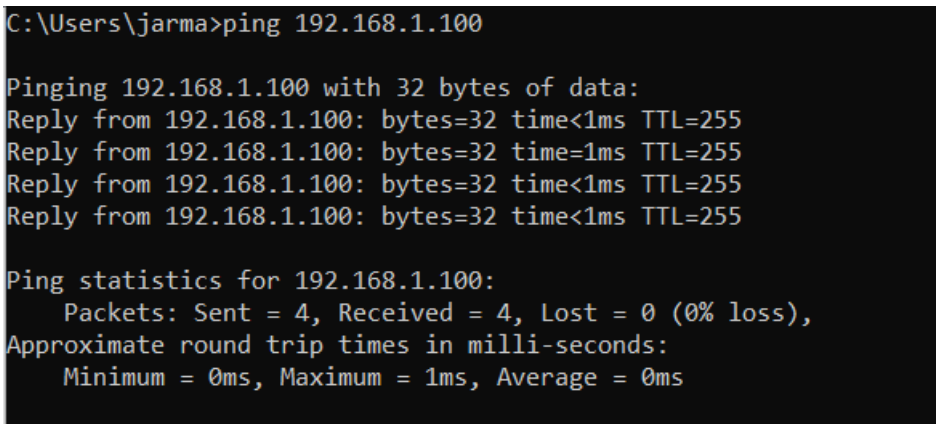
V prvním kroku byl otevřen soubor „server_key.pem“ pro zápis binárních dat, a pomocí metody *dump_privatekey* zapsána do souboru binární podoba privátního klíče v PEM formátu. V druhém kroku byl otevřen soubor „server_cert.pem“ pro zápis binárních dat, a pomocí metody *dump_certificate* do souboru zapsána binární podoba certifikátu v PEM formátu.

6.3 Výsledky měření

6.3.1 Inicializace měření

Každé zařízení či deska, které má být použito pro měření, musí být nejprve vráceno do továrního nastavení, aby se předešlo zkreslení měření díky nežádoucímu nastavení z předchozího měření. Desky ESP32-gateway a Arduino Uno obsahují fyzická tlačítka, která jsou využita pro reset mikrokontrolerů a tím i aplikací. V této kapitole je popsán průběh komunikace, tj. na kroky, které musí proběhnout při komunikaci protokoly TCP a TLS, následně na předání dat na sériovou linku a výpis. Pro zobrazení jednotlivých zpráv v rámci TCP a TLS komunikace byl použit SW Wireshark. Pro ostatní zprávy byl použit výpis do konzole.

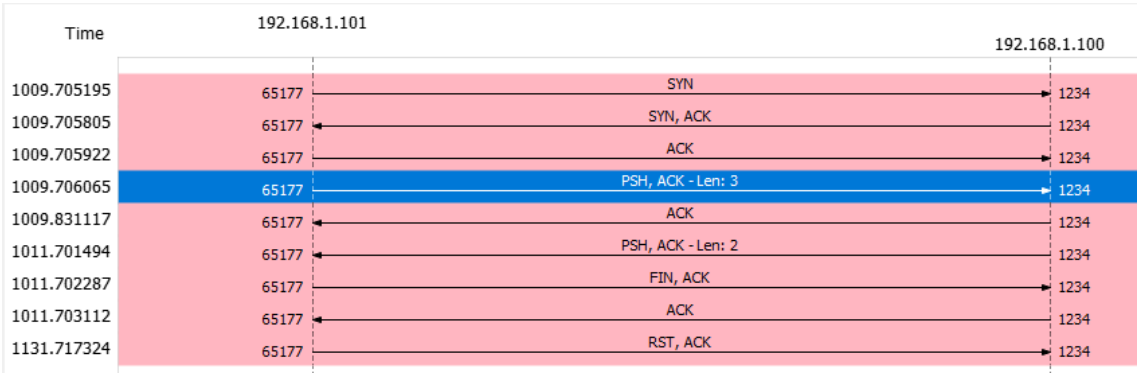
Po restartování desek je dobré si ověřit navázání spojení mezi Python App a RTOS aplikaci. To lze např. v příkazovém řádku spuštěním příkazu *ping „192.168.1.100“*, který testuje přítomnost zařízení na dané IP adrese a jako odpověď vrátí dobu zpoždění příjmu testovacího paketu (viz obr č. 39).



Obr č. 39 Ping na desku ESP32

6.3.2 Měření rychlosti komunikace s využitím TCP

Ukázka vytvoření komunikace protokolem TCP, která byla popsána v teoretické části, je zobrazena na obr č. 40. Komunikace je vytvořena mezi RTOS aplikací a Python aplikací, které mají IP adresy 192.168.1.100 a 192.168.1.101. Tato výměna probíhá při každém TCP spojení. Dále jsou uvedeny výsledky pro jednotlivé přenosové rychlosti.



Obr č. 40 Monitorování TCP komunikace

6.3.3 Výsledky měření rychlosti komunikace s využitím TCP

Měření rychlosti komunikace s využitím protokolu TCP bylo provedeno vždy pro stejnou přenosovou rychlost, která byla změřena při přímé komunikaci skrz UART. Aby bylo možné naměřené výsledky vyhodnotit, byla nejprve stanovena teoretická doba přenosu v ideálním případě. Tato doba je stanovena vzorcem:

$$T = 2 * doba\ přenos\ po\ UART + \frac{RRT}{2} * počet\ rámců\ ,$$

kde *RRT* (round-trip time) je také známá jako doba zpoždění zpáteční cesty. Je to definovaná metrika, která měří v milisekundách potřebnou dobu k odeslání datového paketu plus dobu, za kterou se datový paket odešle. Toto časové zpoždění zahrnuje dobu šíření pro cesty mezi dvěma komunikačními koncovými body.[42]

Cesta jednoho rámce byla změřena pomocí příkazu ping a vysledována ve wiresharku, kde bylo zjištěno, že jeden rámec potřebuje na zpracování zhruba 527 µs. Původní vzorec tedy může být upraven na

$$T = 2 * přenos\ po\ UART + čas\ přenosu\ jednoho\ rámce * počet\ rámců.$$

Aby se minimalizovala chyba měření, bylo provedeno 15 měření za sebou a ze získané hodnoty proveden aritmetický průměr. Všechny naměřené hodnoty jsou zaznamenány v tabulce v příloze A.1.

Rovněž je k dispozici graf výsledků měření v příloze A.2, kde na ose X je počet měření a na ose Y je čas naměřený při jednotlivých měřeních. Jednotlivé průběhy představují testované přenosové rychlosti UART.

Výsledky pro 4800 baudů

Teoretická hodnota: 2 * 6 + 0.527 * 8 =16,296 ms

Naměřená hodnota: 47,70 ms

Výsledky pro 9600 baudů

Teoretická hodnota: 2 * 3 + 0.527 * 8 =10,216 ms

Naměřená hodnota: 29,95 ms

Výsledky pro 19200 baudů

Teoretická hodnota: 2 * 1,5 + 0.527 * 8 = 7,216 ms

Naměřená hodnota: 22,23 ms

Výsledky pro 38400 baudů

Teoretická hodnota: 2 * 0,754 + 0.527 * 8 =5,724 ms

Naměřená hodnota: 17,494 ms

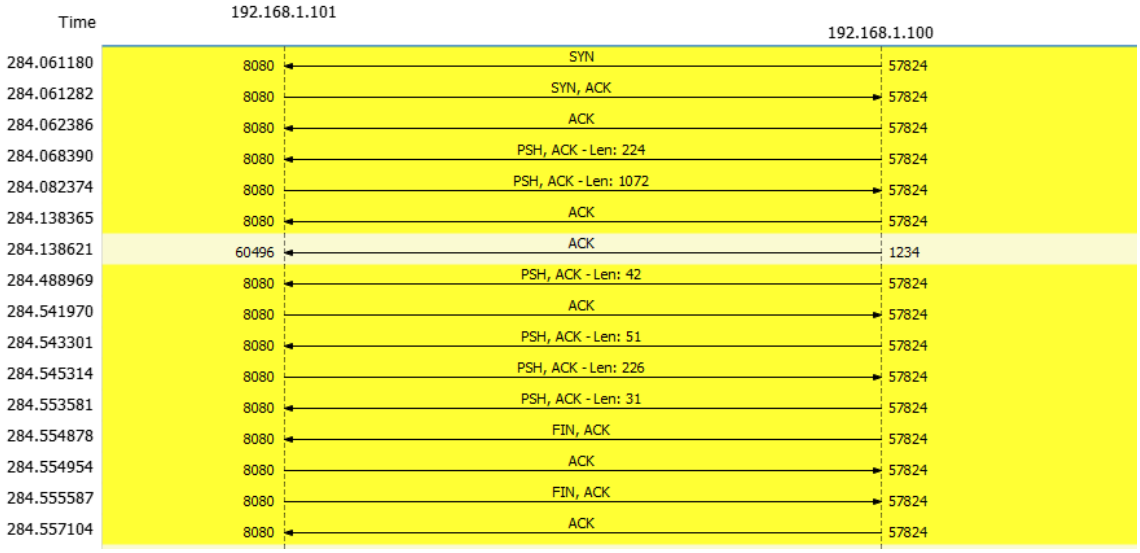
Výsledky pro 115200 baudů

Teoretická hodnota: 2 * 0,350 + 0.527 * 8 =4,916 ms

Naměřená hodnota: 6,337 ms

6.3.4 TLS monitorování

Z následujícího obrázku je patrné, že komunikace pomocí protokolu TLS je náročnější v porovnání s protokolem TCP, neboť dochází k výměně mnohem většího množství dat. Detailní popis výměny paketů je uveden v teoretické části této diplomové práce. Na Obr č. 41 je zachycen proces navazování spojení v rámci praktické



části.

Obr č. 41 Monitorování TLS komunikace

6.3.5 Výsledky měření rychlosti komunikace s využitím protokolu TLS

Obdobně jako v případě protokolu TCP i pro protokol TLS bylo provedeno měření rychlosti přenosu pro každou přenosovou rychlost, které byly změřeny pro přímou komunikaci UART. Rovněž byl proveden vypočet teoretické ideální doby přenosu. Vzorec pro výpočet je stejný jako v předchozím případě. Jak už bylo uvedeno v teoretické části, spojení pomocí protokolu TLS je náročnější, proto bude počet přenesených paketů navýšen na 15 rámců.

Měření vždy probíhalo 15krát za sebou a výsledky byly zaneseny do tabulky v příloze A.3. Výsledná hodnota je opět průměrem všech hodnot.

V příloze A.4 jsou uvedeny časy v závislosti na grafu, přičemž přenosová rychlost 115200 baudů je zobrazena na vedlejší ose. Toto uspořádání bylo zvoleno s cílem lepší rozlišitelnosti jednotlivých výsledků.

Výsledky pro 4800 baudů

Teoretická hodnota: 2x6 + 0.527 * 15 =19,905 ms
Naměřená hodnota: 623,65 ms

Výsledky pro 9600 baudů

Teoretická hodnota: 2x3 + 0.527 * 15 =13,905 ms
Naměřená hodnota: 604,09 ms

Výsledky pro 19200 baudů

Teoretická hodnota: 2x1,5 + 0.527 * 15 =10,905 ms
Naměřená hodnota: 595,176 ms

Výsledky pro 38400 baudů

Teoretická hodnota: 2x0,754 + 0.527 * 15 =9,413 ms
Naměřená hodnota: 585,138 ms

Výsledky pro 115200 baudů

Teoretická hodnota: 2x0,350 + 0.527 * 15 =8,605ms
Naměřená hodnota: 573,64 ms

7. ZÁVĚR

V rámci diplomové práce byla studována sériová komunikace včetně jejích výhod a nevýhod a byly popsány její různé typy. Dále byl porovnán běžný operační systém s operačním systém pracující v reálném čase. Byly prozkoumány úlohy, které jsou součástí jádra operačního systému, a byly popsány různé typy operačních systému, které mohou být použity pro vývoj praktické části. Také byla vyhledána a prozkoumána vhodná deska pro splnění zadaní. Nakonec v teoretické části byly popsány komunikační protokoly TCP a SSL/TLS, které se používají pro komunikaci přes lokální síť Ethernet.

V praktické části mé práce jsem prozkoumala různé platformy pro vývoj aplikací. Bylo nezbytné pečlivě promyslet, jaké konkrétní požadavky jsou kladeny na jednotlivá zařízení a jak je nejlépe splnit. Jedním z hlavních cílů práce bylo vytvoření aplikace, která by běžela v reálném čase. Proto bylo důležité pečlivě naplánovat všechny úlohy tak, aby se vzájemně neovlivňovaly a aby nedocházelo například k resetování aplikace.

Vytvořená aplikace funguje tak, že iniciuje komunikaci po přijetí TCP paketu obsahujícího zprávu "RUN". Poté, co aplikace přijme tuto zprávu, vytvoří spojení, odešle jednu zprávu a spojení ukončí. I když pro splnění zadání je tato funkce dostatečná, myslím si, že právě zde je prostor pro vylepšení o průběžnou komunikaci po přijetí zprávy "RUN". Tento přístup by byl uživatelsky přívětivější.

Dalším úkolem bylo vytvoření Python aplikace, kde jsem se seznámila s procesem vytváření certifikátů pro TLS komunikaci. Také jsem se naučila vytvářet grafické uživatelské rozhraní (GUI) pomocí knihovny PyQt5. Dále jsem se seznámila s knihovnou pyserial, která umožňuje ovládat sériový port a posílat a přijímat přes něj data.

Posledním úkolem mé diplomové práce bylo porovnat přímou komunikaci po UART s komunikací po Ethernetu. Při komunikaci po UART dosahovalo zařízení velmi přesných výsledků. Tyto výsledky jsem si ověřila pomocí osciloskopu. V případě komunikace po Ethernetu jsem rozšířila testy o porovnání mezi TCP a TLS komunikací. Vypočítala jsem teoretickou dobu přenosu jako referenční hodnotu pro následné porovnání s měřením. Očekávaně se potvrdilo, že se reálná změřená hodnota od této hodnoty lišila. To je způsobeno tím, že teoretický výpočet nepředpokládá žádné přidané zpoždění, zatímco ve skutečnosti k němu dochází.

Při odeslání dotazu na měření z Python aplikace je nejprve nutné vytvořit TCP spojení, což zabere určitý čas. Jakmile deska ESP32 přijme dotaz, okamžitě začne naslouchat na UART a ihned po přijetí začne odesílat TCP nebo TLS komunikaci. Skript v Python přijme data, ale musí je zpracovat na aplikační vrstvě. To zahrnuje dešifrování dat a následné odeslání přes UART, kde již čeká přijímač, který data vypisuje. Ke zpoždění dochází na několika místech. Prvním faktorem je, že jedna z aplikací neběží v reálném čase (RTOS), ale na běžném operačním systému. Dalším faktorem, který přispívá ke zpoždění, je skutečnost, že posílaná data se neposílají

na nejnižší úrovni, ale musí být vždy zpracována na aplikační vrstvě a daná aplikace se musí podle toho chovat.

Při měření pomocí TLS jsem zaznamenala zpoždění v řádu stovek milisekund ve srovnání s měřením pomocí TCP. Jak již bylo zmíněno, při přenosové rychlosti 115200 baudů má přijímač až 4% chybovost, což patrně způsobilo ztrátu dat a některé výsledky měření jsou proto neplatné.

Navzdory těmto chybám lze měření považovat za validní, protože se zpoždění snižuje s vyšší přenosovou rychlostí.

LITERATURA

[1] BOLTON, William. Programmable Logic Controllers. 6. Newnes, 2015. ISBN 978-0-12-802929-9.

[2] DAWOUND, Peter. Serial Communication Protocols and Standards. 1. Dennark: River Publishers, 2020. ISBN 9788770221535.

[3] Endianness [online]. [cit. 2023-05-16]. Dostupné z: https://helpful.knobs-dials.com/index.php/Endianness

[4] ČSN EN ISO 80000-2: Veličiny a jednotky - Část 2: Matematické znaky a značky užívané v přírodních vědách a technice. Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví, 2020.

[5] *ČSN ISO 690: Informace a dokumentace - Pravidla pro bibliografické odkazy a citace informačních zdrojů*. Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví, 2011.

[6] Difference Between Point-to-Point and Multi-Point Communication [online]. [cit. 2023-02-18]. Dostupné z: https://www.javatpoint.com/point-to-point-vs-multi-point-communication#:~:text=A%20point-to-point%20communication%20is%20also%20known%20as%20P2P.,may%20be%20used%20to%20communicate%20back%20and%20forth.

[7] Serial UART information [online]. [cit. 2023-05-16]. Dostupné z: https://www.lammertbies.nl/comm/info/serial-uart

[8] Sběrnice RS-422, RS-423 a RS-485 [online]. [cit. 2023-05-16]. Dostupné z: https://www.root.cz/clanky/sbernice-rs-422-rs-423-a-rs-485/

[9] ČERNOHORSKÝ, Jindřich a Vilém. Základní koncepce operačních systémů pracujících v reálném čase (RTOS) [online]. [cit. 2022-05-08]. Dostupné z: https://www.atpjournal.sk/buxus/docs/atp-2005-12-54.pdf

[10] Principy operačních systémů. Lekce 5: Multiprogramming a multitasking, vlákna [online]. [cit. 2022-05-08]. Dostupné z: https://docplayer.cz/6024948-Principyoperacnich-systemu-lekce-5-multiprogramming-a-multitasking-vlakna.html

[11] LI, Qing a YAO, Carolyn. Real-Time Concepts for Embedded Systems. San Francisco: CMP Books, 2003. ISBN 1-57820-124-1

[12] KOLÁŘ, Petr. Operační systémy [online]. 1.2.2005 [cit. 2021-10-28]. Dostupné z: e

[13] KOUTNÝ, Jiří. Krátodobé plánovací algoritmy v operačních systémech [online].

[14] Brno, 2006 [cit. 2021-12-04]. Dostupné z: https://theses.cz/id/d8cxw2/. Bakalárska práca. Masarykova univerzita, Faculty of Informatics. Vedoucí práce Ing. Mgr. et Mgr. Zdeněk Říha, Ph.D.

[15] Operační systém reálného času [online], [cit. 5.12.2021] Dostupné z: https://1url.cz/CKIMP

[16] KOWALSKI, Richard a Frank HUY. Real-Time Operating Systems (RTOS) 101. Nasa.gov [online]. NASA Independent Verification and Validation Facility

Fairmont, West Virginia, 2010, 9.9.2010 [cit. 2021-12-04]. Dostupné z: https://www.nasa.gov/sites/default/files/482489main_4100_-_RTOS_101.pdf
[17] Electronics Hub, Real Time Operating System (RTOS), [online], [cit. 5.12.2021] Dostupné z: https://www.electronicshub.org/real-time-operating-system-rtos/
[18] Texas Instruments, Simple Link Academy [online], [cit. 5.12.2021] Dostupné z: https://1url.cz/hKIzq
[19] FAN, Xiaocong. Real-Time Embedded Systems. Elsevier Books, 2015. ISBN 978-0-12-801507-0
[20] Difference between Firm Real-time Tasks and Soft Real-time Tasks [online]. 16.5.2020 [cit. 2021-10-23]. Dostupné z: https://www.geeksforgeeks.org/difference-between-firm-real-time-tasks-and-softreal-time-tasks/
[21] Pridelovani pameti po sekcich [online]. [cit. 2021-11-28]. Dostupné z: https://zcu.arcao.com/kiv/zos/zos/OdSobi/Materialy/Buris/dalsimaterialy/operacni_systemy-02/pamet/sekce.html
[22] Programovatelné I/O jednotky FLC přes webové prostředí AGREE [online]. [cit. 2021-11-28]. Dostupné z: https://automatizace.hw.cz/programovatelne-jednotkyflc-pres-webove-prostredi-agree.htm
[23] FreeRTOS Real-time operating system for microcontrollers [online]. [cit. 2021-11-02]. Dostupné z: https://www.freertos.org/RTOS.htmlPOSCH, Maya.
[24] BitThunder [online]. 25.6.2017 [cit. 2021-11-06]. Dostupné z: https://www.osrtos.com/rtos/bitthunder
[25] ChibiOS Homepage [online]. [cit. 2021-11-06]. Dostupné z: https://www.chibios.org/dokuwiki/doku.php
[26] RTEMS [online]. [cit. 2021-11-12]. Dostupné z: https://www.osrtos.com/rtos/rtems/
[27] What is RISC OS? [online]. [cit. 2021-11-13]. Dostupné z: https://www.riscosopen.org/content/
[28] STICKY: New to RISC OS? Read this! [online]. 05.11.2012 [cit. 2021-11-13]. Dostupné z: https://forums.raspberrypi.com/viewtopic.php?f=55&t=22093
[29] Xenomai: Home [online]. [cit. 2021-11-14]. Dostupné z: https://source.denx.de/Xenomai/xenomai/-/wikis/home
[30] Microsoft, Azure RTOS [online]. [cit. 2021-11-13] Dostupné z: https://azure.microsoft.com/cs-cz/services/rtos/#overview
[31] I2C BUS: MultiMaster [online]. [cit. 2023-02-18]. Dostupné z: https://www.i2c-bus.org/multimaster/
[32] Modules [online]. [cit. 2023-03-03]. Dostupné z: https://www.espressif.com/en/products/modules
[33] What Is the ESP32? [online]. [cit. 2023-03-04]. Dostupné z: https://www.elektormagazine.com/articles/what-is-the-esp32
[34] Olimex [online]. 1997 [cit. 2023-03-05]. Dostupné z: https://www.olimex.com/

[35] What is a DLL [online]. [cit. 2023-03-05]. Dostupné z: https://learn.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library
[36] Data lifecycle management (DLM) By [online]. [cit. 2023-03-05]. Dostupné z: https://www.techtarget.com/searchstorage/definition/data-life-cycle-management
[37] *ČSN ISO 7144: Dokumentace - Formální úprava disertací a podobných dokumentů*. Praha: Úřad pro technickou normalizaci, metrologii a státní zkušebnictví, 1997.
[38] TCP/IP - model, encapsulace, paket vs. rámec [online]. [cit. 2023-05-14]. Dostupné z: https://www.samuraj-cz.com/clanek/tcpip-model-encapsulace-paketu-vs-ramec/
[39] TCP/IP - navázání a ukončení spojení [online]. [cit. 2023-05-14]. Dostupné z: https://www.samuraj-cz.com/clanek/tcpip-navazani-a-ukonceni-spojeni/
[40] Practical TCP/IP and Ethernet Networking for Industry. British: Newnes, 2003. ISBN 9780080473826.
[41] SSL/TLS Under Lock and Key. 1. Australia: Keyko Books, 2020. ISBN 9780648931614.
[42] What is RTT (Round-Trip Time) and How to Reduce it? [online]. [cit. 2023-05-16]. Dostupné z: https://www.stormit.cloud/blog/what-is-round-trip-time-rtt-meaning-calculation/

SEZNAM SYMBOLŮ A ZKRATEK

Zkratky:

ACK	Acknowledgement Code
ADC	Analog to Digital Convertor
AES	Advanced Encryption Standard
API	Application Programming Interface
BLE	Bluetooth Low Energy
CAN	Cosmopolitan Area Network
CCITT	Consultative Commiter for International Telephone and Telegraph
CMD	Command PromptF
CN	Common Name
COM	Communications Ports
CPU	Central processing Unit
DAC	Digital to Analog Convertor
DCE	Data Communications Equipment
DLL	Divisor Latch Registers
DLM	Divisor Latch Registers
DTE	Data Terminal Equipment
EIA	Electronic Industries Alliance
FCR	Fifo Control Register
FLC	Field Logic Controllers
FIN	Finish
FTP	File Transfer Protocol
FTPS	File Transfer Protocol -Secure
GAP	Generic Access Profile
GATT	Generic Attribute Profile
GPIO	General-Purpose Input/Output
GUI	Graphic User Interface
HMAC	Hashed Authentication Message Code
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol -Secure
I/O	Input/Output
I2C	Two Wire Interface
I2S	Inter-IC Sound
IER	Interrupt Enable Register
IIR	Interrupt identification
IMAP	Internet Message Access Protocol
IP	Internet Protocol

IR	InfraRed
ITU	International Telecommunication Union
LCR	Line Control Register
LSR	Line Status Register
MCR	Modem Control Register
MSR	Modem Status Register
OS	Operating System
OSI	Open Systems Interconnection
P2P	Point-to-Point
PCB	Proces Control Block
POP3	Post Office Protocol 3
PWM	Pulse Width Modulation
RBR	Richard Burns Rally
RSA	Rivest, Shamir, Adleman
RTOS	Real Time Operating System
RX	Receiver
SHA	Secure Hash Algorithm
SD	Secure Digital
SDIO	Secure digital Input Output
SMTP	Simple Mail Transfer Protocol
SPI	Serial Peripheral Interface
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TIA	Telecommunications Industry Alliance
TLS	Transport Layer Security
TX	Transmit
UART	Universal Asynchronous Reciever-Transmitter
UDP	User Datagram Protocol
USB	Universal Serial comunication
USART	Universal Synchronous/Asynchronous Reciever-Transmitter
WiFi	Wireless Fidelity

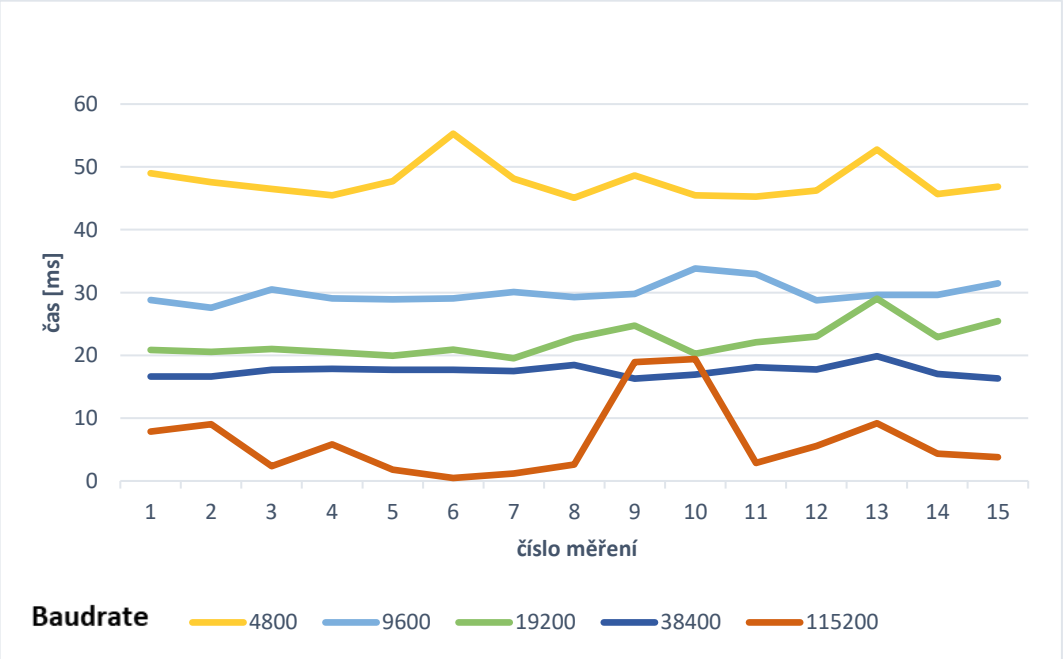
SEZNAM PŘÍLOH

Příloha A - Naměřené hodnoty

A.1 Tabulka naměřených hodnoty pro TCP

Čas[ms]	Přenosová rychlost				
Číslo měření	**4800**	**9600**	**19200**	**38400**	**115200**
1	49,01	28,80	20,84	16,62	7,84
2	47,57	27,57	20,54	16,62	9,01
3	46,51	30,49	21,02	17,68	2,34
4	45,45	29,08	20,52	17,83	5,82
5	47,73	28,93	19,95	17,68	1,82
6	55,29	29,05	20,89	17,72	0,47
7	48,11	30,10	19,53	17,49	1,16
8	45,08	29,25	22,72	18,45	2,61
9	48,62	29,76	24,75	16,29	18,91
10	45,48	33,83	20,25	16,92	19,41
11	45,25	32,96	22,10	18,12	2,86
12	46,23	28,76	22,98	17,75	5,57
13	52,74	29,60	29,03	19,86	9,16
14	45,70	29,62	22,92	17,04	4,32
15	46,87	31,46	25,42	16,34	3,76

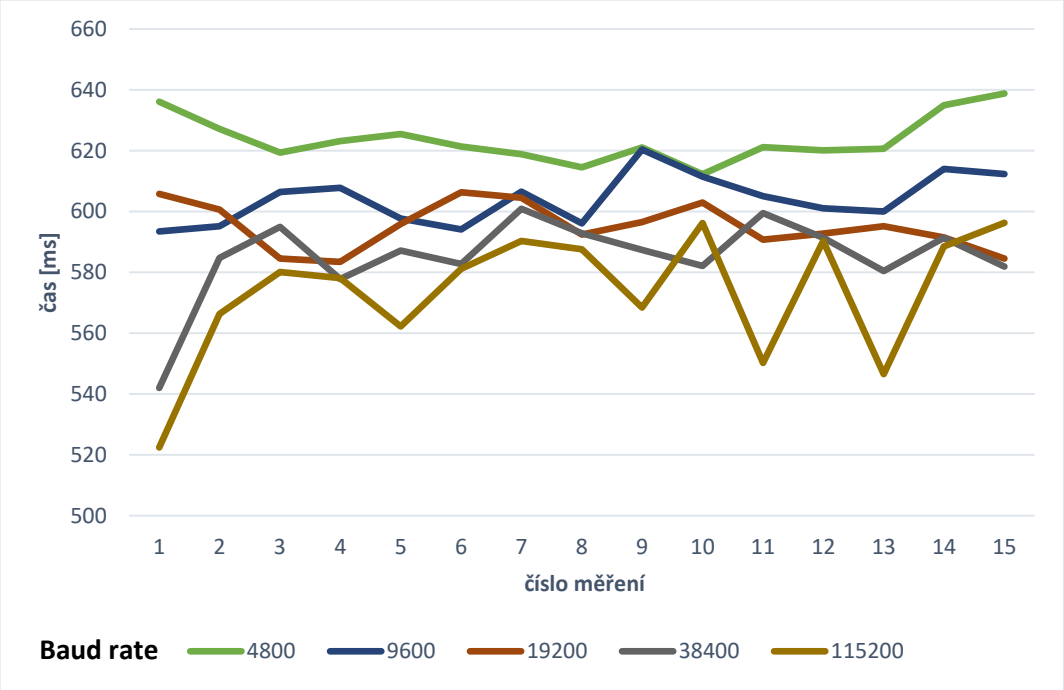
A.2 Graf naměřených hodnot pro TCP



A.3 Tabulka naměřených hodnot pro TLS

Čas[ms]	Přenosová rychlost				
Číslo měření	4800	9600	19200	38400	115200
1	636,08	593,45	605,82	542,00	522,44
2	627,14	595,20	600,63	584,74	566,35
3	619,39	606,44	584,48	594,92	580,12
4	623,16	607,79	583,44	577,77	578,12
5	625,46	597,65	595,88	587,20	562,20
6	621,32	594,12	606,32	582,71	581,20
7	618,79	606,54	604,51	600,90	590,30
8	614,51	596,12	592,42	592,82	587,53
9	621,09	620,37	596,57	587,36	568,42
10	612,25	611,44	602,96	582,08	596,20
11	621,16	605,05	590,70	599,52	550,20
12	620,15	601,02	592,78	591,32	590,30
13	620,63	599,98	595,15	580,46	546,50
14	634,97	614,03	591,50	591,39	588,51
15	638,79	612,26	584,48	581,88	596,28

A.4 Graf naměřených hodnot pro TLS



Příloha B - Zdrojové kódy