Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Návrh hardwarového stimulátoru pro neuroinformatické experimenty

Plzeň 2020 Petr Štechmüller

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 21. května 2020

Petr Štechmüller

Abstract

There is a big amount of software stimulators for neuroinformatic experiments. All of them share one big drawback - running under the operating system. As a result of this lack there is a considerable amount of delay generated by operating systems scheduler when running experiment. Hardware stimulators are free from this drawback. On the other side they are usualy dedicated to only one experiment type. In this master thesis is designed simple hardware stimulator with support of five experiments. Part of this thesis is also software for control hardware stimulator with simple user interface.

Abstrakt

Softwarových stimulátorů pro různé neuroinformatické experimenty existuje velké množství. Všechny ale spojuje jeden zásadní nedostatek - musí běžet na počítači, tedy na operačním systému. Důsledkem tohoto nedostatku je nemalé zpoždění generované plánovačem systému při spuštěném experimentu. Hardwarové stimulátory tímto nedostatkem netrpí. Na druhou stranu jsou velmi často pouze jednoúčelové. V této diplomové práce je navržen jednoduchý hardwarový stimulátor podporující pět experimentů. Součástí této práce je také ovládací program, pomocí kterého se uživatelsky přívětivou cestou jednotlivé experimenty nastavují.

Obsah

1 Úvod

Na světě existuje velké množství softwarových stimulátorů pro neuroinformatické experimenty. Jednotlivé programy nabízí různé přístupy při tvorbě experimentu. Nejčastěji se ale jedná o tvorbu formou programování. Takový přístup se vyplatí v laboratorních podmínkách nebo při výzkumu, ale v praxi je nepoužitelný. Není možné chtít po lékaři, aby se při vyšetření či přípravě zabýval otázkou programování experimentu.

Na katedře Informatiky se v minulosti vytvořil HW stimulátor, který je bohužel už neudržovaný. Tato diplomová práce je zaměřena na znovu vytvoření HW stimulátoru, který bude odpovídat aktuálním potřebám, a přitom si zachová jednoduché ovládací rozhraní.

Na začátku práce bude čtenář seznámen obecně s myšlenkou rozhraní mozek-počítač, Elektroencefalografií a Evokovanými potenciály. Evokované potenciály budou probrány detailněji. Dále bude popsán aktuální stav se SW a HW stimulátory a jejich výhody a nevýhody. Na stimulátory naváže téma o embeded zařízení obecně, později se zaměřením na produkty od společnosti ARM holdings.

Praktická část uvede čtenáře do problematiky embeded zařízení obecně a později do popisu vybraného HW pro tvorbu stimulátoru. Budou zde popsány jednotlivé použité komponenty i s konfiguracemi. Dále seznámí čtenáře s implementací HW stimulátoru, webového serveru na Raspberry Pi a s webovou aplikací, pomocí které se bude vše ovládat.

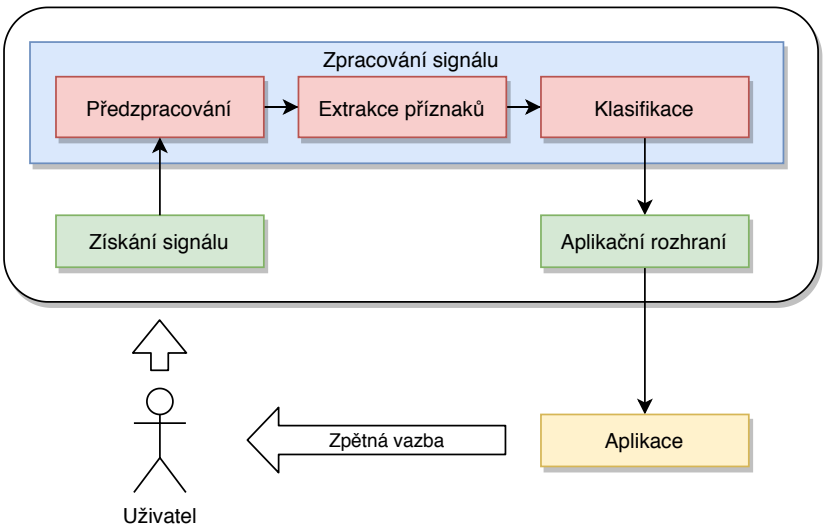
2 Rozhraní mozek-počítač

Rozhraní mozek-počítač, anglicky brain-computer interface (BCI), je systém skládající se z odpovídajícího hardwaru, doplněný softwarem, který se snaží umožnit pohodlnou interakci s dalšími počítačovými systémy. BCI vytváří nový komunikační kanál mezi subjektem a počítačem, bez použití tradičních ovládacích prvků (myš, klávesnice). Je tedy vhodný pro pacienty, kteří trpí různými dysfunkcemi těla, například ochrnutím.

Základní princip BCI je založený na analýze mozkové aktivity subjektu. BCI systém 2.1 se standardně skládá ze čtyř částí: získávání signálu, zpracování signálu, extrakce příznaků a klasifikace příznaků. Dále bude popsána pouze první část BCI systému - získávání signálu.

Mozkovou aktivitu lze snímat invazivně (electrocorticography), nebo neinvazivně (electroencefalography).

Obrázek 2.1: BCI schéma



2.1 Elektrokortikografie

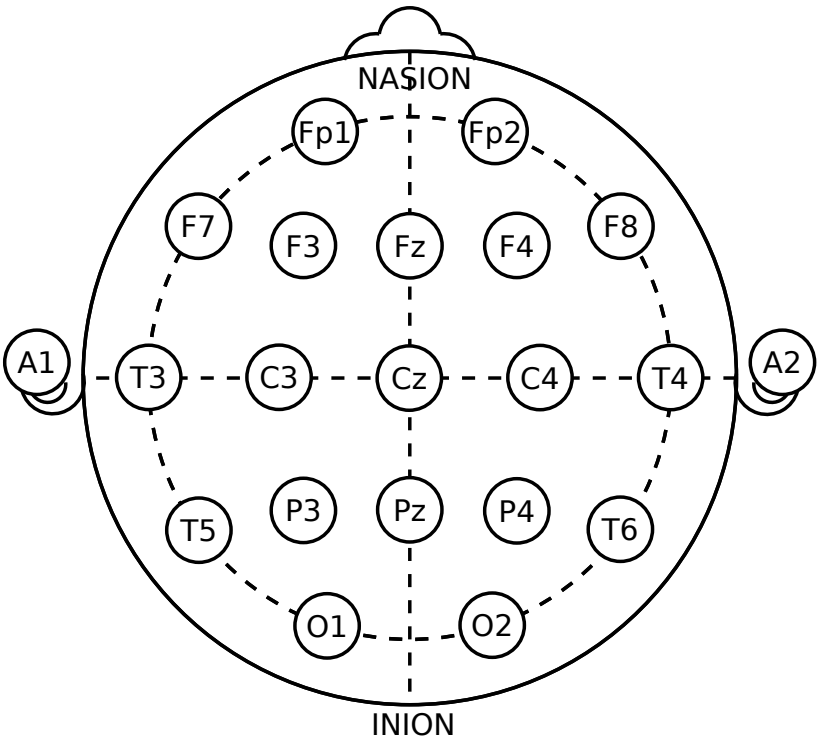
Elektrokortikografie [4] (ECoG) je invazivní metoda, při které se snímací elektrody pokládají přímo na obnažený mozek. Takto získaný signál je relativně málo zašuměný, méně náchylný k artefaktům a je dostupný ve vysokém prostorovém a časovém rozlišení. Na druhou stranu, tato metoda vyžaduje operaci, během které se zavedou elektrody a provede se vlastní měření.

Proto nebude tato metoda předmětem dalšího povídání a nebudu se touto metodou dále zabývat. Naštěstí existuje mnohem bezpečnější, neinvazivní metoda a tou je elektroencefalografie.

2.2 Elektroencefalografie

Elektroencefalografie (EEG) je neinvazivní metoda, pomocí které se zaznamenává elektrická aktivita mozku. Pro měření se obvykle používají elektrody připevněné na speciální čepici. Elektrody mají přesně definované, kde se která nachází. Nejčastěji se používá rozmístění 10-20, které je vidět na obrázku 2.2. Díky neinvazivnímu přístupu metody je EEG využíváno právě v kombinaci s BCI. V takovém případě se mluví o BCI systému založeném na EEG.

Obrázek 2.2: EEG - Systém 10-20



Výstupem této metody je elektroencefalogram, který obsahuje zaznamenanou aktivitu mozku. Z elektroencefalogramu lze vyčíst, kdy pacient mrknul očima, pohnul rukou, nebo reagoval na nějaký podnět. Právě reakce na podnět nás bude zajímat dále.

2.3 Evokované potenciály

Evokované potenciály [2] lze chápat jako malé změny napětí, které jsou generovány mozkem jako odpověď na určitou událost, nebo stimul. Na základě evokovaných potenciálů lze neinvazivní metodou studovat dobu, kterou potřebuje mozek k příjmu a interpretaci události. Pokud je testovaný subjekt v pořádku, příjem a interpretace je téměř okamžitá. V případě nemoci jedince, například roztroušené skleróze, trvá zpracování zprávy podstatně déle.

Podle zdrojů událostí můžeme evokované potenciály rozdělit na *vizuální*, *zvukové* a *motorické* evokované události.

2.3.1 Vizuální evokované potenciály

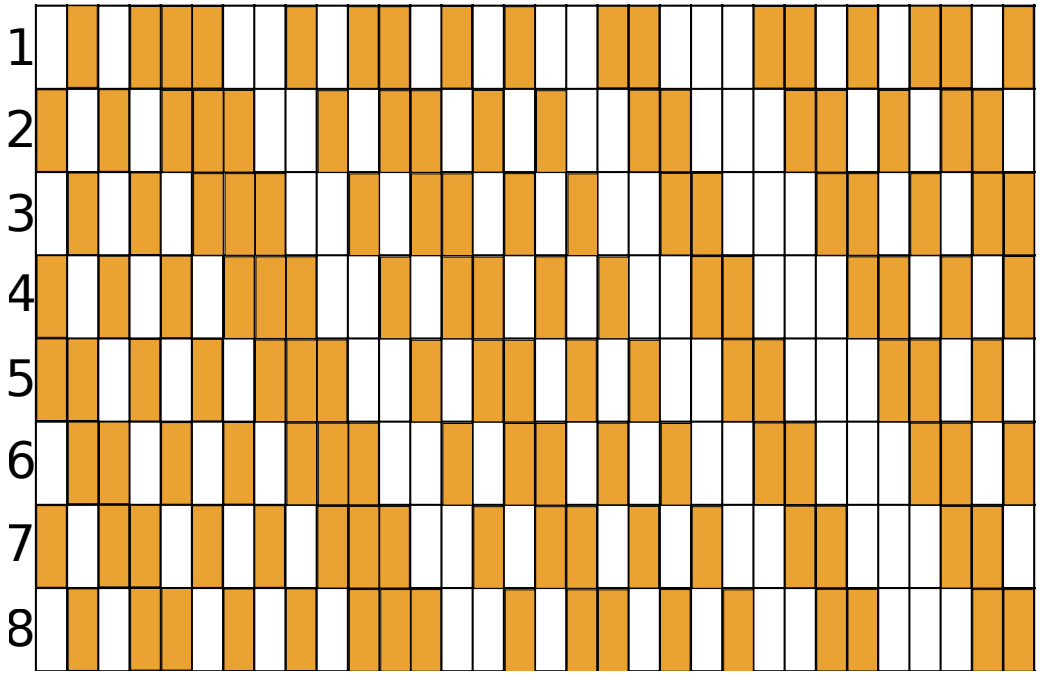
Vizuální evokované potenciály [1] (VEP) se využívají k vyšetření zraku, jestli je v pořádku. Testují mechanismy pro zpracování vizuálních informací v mozku. BCI systém založený na VEP dokáže identifikovat vizuální událost (typicky obrázek), na kterou byl pacient po dobu experimentu soustředěný. U vizuálních potenciálů se stimuluje blikajícím vzorem, nejčastěji je to šachovnice, kdy se střídají černé a bíle čtverečky.

U dalšího druhu experimentu je každá cílová událost zakódována do unikátní sekvence stimulů, která vyvolá jedinečný VEP pattern. V závislosti na druhu modulace sekvence stimulů lze vizuální evokované potenciály rozřadit do tří kategorií: *pseudonáhodně modulované* (c-VEP), *frekvenčně modulované* (f-VEP) a *časově modulované* (t-VEP).

c-VEP

C-VEP [6] experiment je založený na opakování pseudonáhodně vytvořeném patternu. Pattern má pevnou délku a definuje chování pouze pro první stimul. Další stimuly jsou odvozeny od prvního pomocí nastavitelného bitového posuvu. Na obrázku 2.3 je vizualizace patternu o délce 32 bitů pro experiment s osmi stimuly. Oranžově zvýrazněný čtvereček indikuje aktivní stimul, bílý je neaktivní stimul. Během experimentu je potřeba vysílat speciální synchronizační signál. Tento signál se vyšle vždy na začátku každého stimulačního cyklu. Slouží zejména k synchronizaci s EEG zesilovačem.

Obrázek 2.3: Pattern pro c-VEP



f-VEP

F-VEP experiment vychází z nastavení frekvencí jednotlivých stimulů. Každý stimul se aktivuje periodicky s různou frekvencí, tedy i odpovědi by měly být ve výsledku periodické. Frekvence je typicky vyšší než 6Hz, Odpovědi vyvolané aktivací targetu se můžou překrývat. Tím se vytváří periodická VEP sekvence, typicky označovaná jako steady-state visual evoked potential (SSVEP).

t-VEP

V T-VEP experimentu je možné definovat pro každý stimul jiný pattern. Každému stimulu se nezávisle na ostatních nastavuje doba, po kterou je stimul aktivní a neaktivní. Stejně jako v případě C-VEP experimentu i zde je třeba vysílat synchronizační signál. Tento signál se vyšle vždy, když se stimul aktivuje.

2.3.2 Zvukové události

Zvukové evokované potenciály [5] (AEP - Auditory Evoked Potential) jsou elektrické signály vyvolané v mozku za přítomnosti zvukových stimulů. AEP signály mají tvar pozitivního nebo negativního oblouku. Výsledné AEP signály mají mnohem menší amplitudu než EEG signály. AEP signály lze rozdělit na *přechodné* a *ustálené*.

Přechodné AEP signály se vyznačují pomalou rychlostí. Díky pomalé rychlosti je možné zabránit překrytí evokovaných potenciálů a jejich odpovědí. Naopak ustálené AEP signály jsou založeny na rychlém generování zvukových podnětů. Výsledné odpovědi se překrývají.

2.3.3 Motorické události

Motorické evokované potenciály [3] (MEP - Motor Evoked Potential) jsou elektrické signály vyvolané neinvazivní stimulací motorické kůry přes pokožku hlavy. Elektrické potenciály mohou být zaznamenány s použitím povrchové elektromyografie ze všech kosterních svalů. Pomocí MEP je možné spolehlivě detekovat abnormality šíření impulzů podél kortikospinálního traktu.

2.4 Reakční experimenty

Výše uvedené vizuální evokované potenciály vychází z "pasivní"reakce pacienta, při které se pacient pouze soustředí na jeden konkrétní stimul (vybraný obrázek či zvuk). U reakčních experimentů je vyžadována aktivita ze strany pacienta. Vždy, když přijde stimul (zobrazí se obrázek, přehraje zvuk), musí pacient zareagovat stiskem odpovídajícího tlačítka. V tomto případě se sledují dvě složky: rychlost reakce a úspěšnost výběru správného tlačítka.

2.5 Kognitivní potenciály

Experimenty zaměřené na kognitivní potenciály jsou zaměřené na dobu odezvy delší než 300ms. Během experimentu se budou například před subjektem promítat čísla od 0 do 9. Subjekt se zafixuje na jedno konkrétní číslo. Vždy, když se toto číslo zobrazí, subjekt na číslo zareaguje, například přečtením čísla v duchu. Tuto aktivitu lze dobře rozpoznat od klidové aktivity na EEG záznamu. Na konci experimentu by měl být systém schopný uhádnout, na které číslo byl subjekt zafixován.

3 Stimulátory

Stimulátory jsou zařízení sloužící ke stimulaci měřené osoby. Stimulátory lze rozdělit do dvou hlavních kategorií: softwarové a hardwarové. Obě kategorie mají své výhody i nevýhody.

3.1 Softwarové stimulátory

Softwarové stimulátory jsou realizovány programem standardně běžícím v operačním systému vedle ostatních programů. Přítomnost operačního systému je hlavní nevýhoda oproti hardwarovým stimulátorům. Operační systém musí plánovat nejenom stimulátor, ale i celou řadu dalších programů. V průběhu experimentu je třeba přesně dodržovat nastavené časování stimulů, což může být problém. Software neběží neustále, je přerušovaný. Tato přerušení jsou sice krátká, ale existují. Tím se naruší časování a může dojít k degradaci výsledku.

Mezi typické představitele softwarových stimulátorů lze zařadit software Presentation, OpenSesame či PsychoPy.

3.1.1 Presentation

V programu Presentation lze experimenty tvořit na dvou úrovních abstrakce: vysoké a nízké. Ve vysoké úrovni 3.1 má uživatel omezené možnosti při tvorbě experimentu. Může například určit, jaký bude počet stimulů, ovšem už problematicky nastavuje jejich distribuci. Navíc, pokud by měly být stimuly závislé na předchozích, tak to se v této úrovní realizovat již nedá. Nízká úroveň abstrakce přináší větší možnosti, ovšem za cenu, že uživatel už musí vědět, co přesně chce udělat a jak to naprogramovat. K programování se využívá jazyk Python. Další omezení se týká v podobě licence programu, kdy je třeba licenci prodloužit každý rok.

Zdrojový kód 3.1: Příklad jednoduchého experimentu v Presentation

```
scenario = "hello";

active_buttons = 1;
button_codes = 1;

begin;

picture {} default;

picture {
  text {
    caption = "Welcome\nto\nPresentation";
    font_size = 36;
    font_color = 100,200,200;
  };
  x = 0; y =0;
  text {
    caption = "(Press response button to continue)";
  };
  x = 0; y = -200;
} pic1;

trial {
  trial_type = first_response;
  trial_duration = forever;
  picture pic1;
  code = "pic1";
};
```

3.1.2 PsychoPy

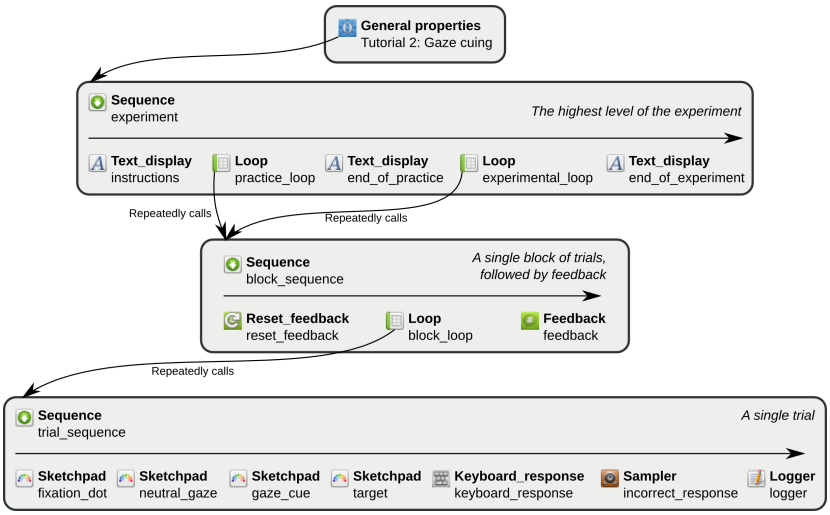
PsychoPy je program vytvořený vědci pro vědce, kteří potřebují mít absolutní kontrolu nad tvořeným experimentem. K tvorbě experimentu se používá programovací jazyk Python. Program je open-source, a k použití zdarma. Bohužel pro naše účely zcela nepoužitelný - je třeba umět programovací jazyk Python.

Softwarové stimulátory jsou univerzální, co se týká použití. Lze v nich tvořit různé sofistikované experimenty, ovšem za cenu, že tvorba různých specifických částí může být náročná na programování.

3.1.3 OpenSesame

OpenSesame je další program pro tvorbu neuroinformatických experimentů. Základní experimenty lze "naklikat"v příjemném prostředí 3.1. Jednotlivé kroky experimentu jsou graficky vizualizovány a uživatel pouze tyto bloky spojuje do jednoho velkého celku. Ovšem pro tvorbu složitějších experimentů je opět potřeba znalosti programovacího jazyka Python.

Obrázek 3.1: Tvorba experimentu v programu OpenSesame



Všechny výše uvedené softwary mají jednu společnou velmi nepříjemnou vlastnost - k vytvoření složitějších experimentů je třeba znalost použitého programovacího jazyka. Typicky se jedná o Python (PsychoPy, OpenSesame), nebo vlastní jazyk, který převzal syntaxi jazyka z Pythonu či Matlabu (Presentation). Takový způsob tvorby experimentů lze využít pro vědecké účely, ale v medicinské praxi nelze chtít po lékaři, aby vytvářel experimenty pomocí programování.

3.2 Hardwarové stimulátory

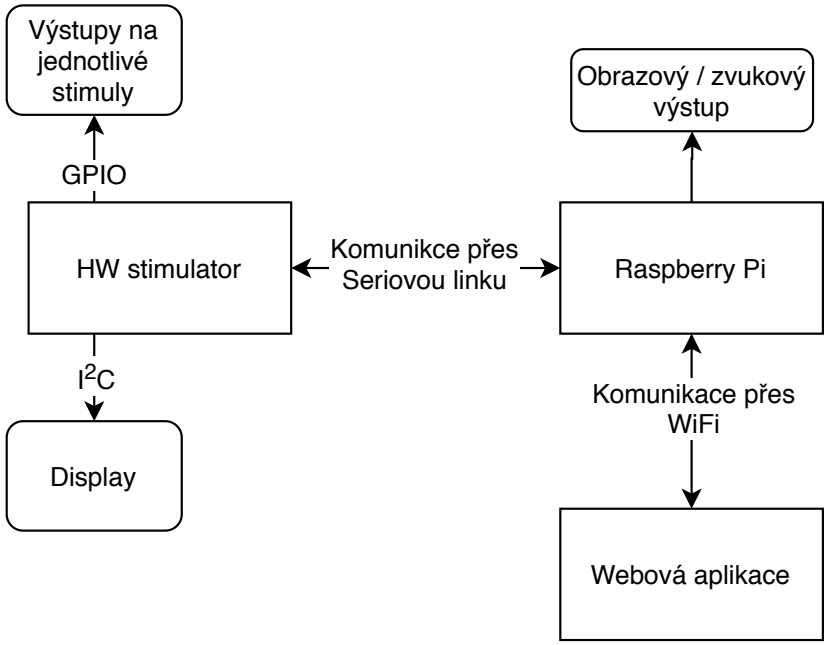
Hardwarové stimulátory řeší jeden hlavní problém: nejsou zatíženy operačním systémem. Experiment běží přímo v HW, takže se lze spolehnout, že nastavené časování bude opravdu dodrženo. Na druhou stranu takové stimulátory obvykle bývají jednoúčelové. To může být užitečné, ale nepraktické. Obvykle je třeba podpory více typů experimentů.

Jeden hardwarový stimulátor už v minulosti vznikl na katedře Informatiky fakulty Aplikovaných věd v Plzni. Podporoval pět druhů experimentů a posloužil svému účelu. Ovládal se pomocí zabudovaného displaye a uživatelské rozhraní bylo dostačující v rámci možností. Doba ale pokročila a časem se zjistilo, že rozšíření podpory o další experimenty by bylo velmi náročné. Tento stimulátor sloužil jako předloha pro tvorbu stimulátoru nového, který je výsledkem této diplomové práce.

4 Návrh struktury hardwarového stimulátoru

Celý systém se bude skládat ze tří samostatných částí (Obrázek 4.1). HW stimulátor, který bude zodpovědný za běh samotného experimentu. Stimulátor bude komunikovat po sériové lince s Raspberry Pi. Raspberry bude v tomto systému sloužit jako prostředník mezi HW stimulátorem a webovou aplikací, která poběží na koncovém klientském zařízení. Pomocí webové aplikace se bude celý experiment ovládat.

Obrázek 4.1: Blokové schéma systému



4.1 Podporované experimenty

4.1.1 Experiment ERP

ERP experiment je založen na sekvenci, podle které se aktivují nebo deaktivují příslušné stimuly. Z globálního hlediska nastavení se upravují parametry pro synchronizační puls, který se má vyslat vždy, když se stimul aktivuje. Mezi parametry pro synchronizační puls patří *doba zpoždění* a *délka synchronizačního pulsu*. Doba zpoždění se počítá od aktivace stimulu. Dále je

možné nastavit tvar synchronizačního pulsu. Puls může začínat náběžnou nebo sestupnou hranou. Z nastavení hrany se určí, zdali má dojít k přechodu z logické hodnoty $0 \rightarrow 1$, nebo opačně $1 \rightarrow 0$.

U každého stimulu je možné samostatně nastavit aktivní a neaktivní dobu (doba, kdy je LED rozsvícená a zhasnutá). Dále se nastavuje distribuční hodnota stimulu, která reprezentuje procentuální zastoupení stimulu v sekvenci. Poslední parametr je nastavení svítivosti LED.

Výskyt stimulů může být na sobě závislý. To znamená, že lze nastavit, aby se například 2. stimul zobrazil pouze za předpokladu, že se před ním vyskyte třikrát 1. stimul. Tyto závislosti velmi ovlivňují výslednou distribuci stimulů.

Sekvence

Sekvence byly založeny speciálně kvůli ERP experimentu, který je na sekvencích založený. Sekvence je reprezentována jako pole čísel reprezentující jednotlivé stimuly. Na základě distribuce jednotlivých stimulů a jejich závislostí se pole vygeneruje a uloží.

4.1.2 Experiment C-VEP

C-VEP experiment se vyznačuje nastavením jednoho patternu pro první stimul, ze kterého se odvozují další patterny pro ostatní stimuly. Pattern má pevnou délku 32 bitů, což odpovídá 32 různým hodnotám. Ostatní patterny se odvozují pomocí parametru s bitovým posuvem vzhledem ke svému předchůdci.

4.1.3 Experiment F-VEP

V F-VEP experimentu je možné nastavovat pro každý stimul různou periodu blikání. Periodu a dobu aktivního stimulu lze nastavit dvěma možnými způsoby: absolutní hodnotou délky celé periody a doby aktivního stimulu, nebo pomocí frekvence a poměru aktivního stimulu vůči neaktivnímu.

4.1.4 Experiment T-VEP

T-VEP experiment je velmi podobný C-VEP experimentu pouze s tím rozdílem, že každému stimulu se nastavuje pattern zvlášť. Také neplatí pravidlo pevné délky patternu. Uživatel si může zvolit, zdali bude délka patternu pro všechny stimuly stejná, nebo rozdílná. U rozdílné délky patternu hrozí desynchronizace stimulů.

4.1.5 Experiment REA

REA experiment bude sloužit k měření reakční doby pacienta na stimul. V konfiguraci experimentu se budou nastavovat časy pro minimální a maximální dobu aktivního stimulu. Dále se bude nastavovat doba, po kterou by měl pacient zareagovat na stimul. Pokud zareaguje později, pokus se bude počítat jako neplatný. V případě chybné reakce bude možné nastavit, zdali se experiment na definovanou dobu pozastaví, nebo bude pokračovat standardně dál. Svítivost výstupů se bude nastavovat globálně.

5 Výběr embeded zařízení

Na trhu existuje obrovské množství nejrůznějších embeded zařízení stejně jako výrobců, kteří tyto zařízení vyrábí. Zaměříme-li se na samotná zařízení, máme k dispozici výběr od málo výkonných ale úsporných zařízení až po velmi výkonné systémy, které můžou konkurovat některým stolním počítačům. Mezi výrobce embeded zařízení patří například: NPX, Renesas, Microchip, Silicon Labs, Infineon, Texas Instruments a STMicroelectronics. Volba výrobce zařízení byla pro nás velmi jednoduchá. Díky předchozím zkušenostem bylo vybráno zařízení od společnosti STMicroelectronics. Firma STMicroelectronics vytváří zařízení založená na technologii ARM.

5.1 Portfolio zařízení firmy ARM holdings

ARM holdings je Britská společnost, která pouze vyvíjí instrukční sadu společně s architekturou pro ARM zařízení. Koncová zařízení jsou vytvářena jinými firmami, mezi které patří výše zmiňovaná STMicroelectronics.

Rodinu procesorů ARM si můžeme rozdělit do tří základních kategorií:

- Cortex-A, kde **A** znamená *application*, jsou procesory určené pro výkonné aplikace. Na těchto procesorech je možné provozovat plnohodnotné operační systémy, například Linux nebo jeho derivát v podobě systému Android.

- Cortex-M, kde **M** znamená *embeded*, jsou nízko výkonné procesory určené pro použití na místě, kde je důležitá malá energetická náročnost. Nejčastěji se s těmito procesory setkáme v IoT zařízeních, případně v nejrůznější nositelné elektronice (hodinky, chytré náramky).

- Cortex-R, kde **R** znamená *real-time*, jsou procesory také určené pro embeded zařízení, ale s vysokým výkonem. U těchto procesorů se velmi dbá na minimální poruchovost, vysokou odolnost proti chybám a zajištění výsledku v reálném čase.

Pokud vezmeme v úvahu výše uvedený popis rozdělení procesorů, dojdeme ke zjištění, že nejlépe bude našim potřebám vyhovovat procesor pro embeded zařízení, tedy Cortex-M.

Řada procesorů Cortex-M se dále dělí podle výkonnosti. Vše přehledně znázorňuje tabulka v příloze A.

- M0 / M0+ - jsou procesory s nejnižším výkonem a cenou
- M3 - výkonnější procesory s větším množstvím periferií
- M4 - podobné procesory jako M3, jen přidávají digital signal processing unit a floating point unit
- M7 - nejvýkonnější procesory z této řady; výkonově připomínají Cortex-R; obsahují double decision floating point

Z výše uvedeného seznamu nejvíce vyhovuje procesor Cortex-M4, protože disponuje všemi periferiemi, které budou potřeba k tvorbě HW stimulátoru. Cortex-M7 by byl až příliš výkonný.

5.2 Požadavky na periferie desky

Než se začne vybírat konkrétní deska, je třeba si ujasnit, čím vším by měla výsledná deska disponovat. Na desce by mělo být přítomno minimálně **8 GPIO pinů** podporující **pulsně šířkovou modulaci (PWM)**. Každý z těchto pinů bude reprezentovat jeden stimul, který bude použit během experimentu. Další periferie, která bude ke stimulátoru připojena, je LCD. S tímto displayem bude HW komunikovat po sběrnici **I^2C**. Pro zajištění komunikace s Raspberry Pi musí být na desce volné piny pro **sériovou linku**. V neposlední řadě musí být na desce **8 nezávislých časovačů**. Každý časovač bude ovládat jeden stimul.

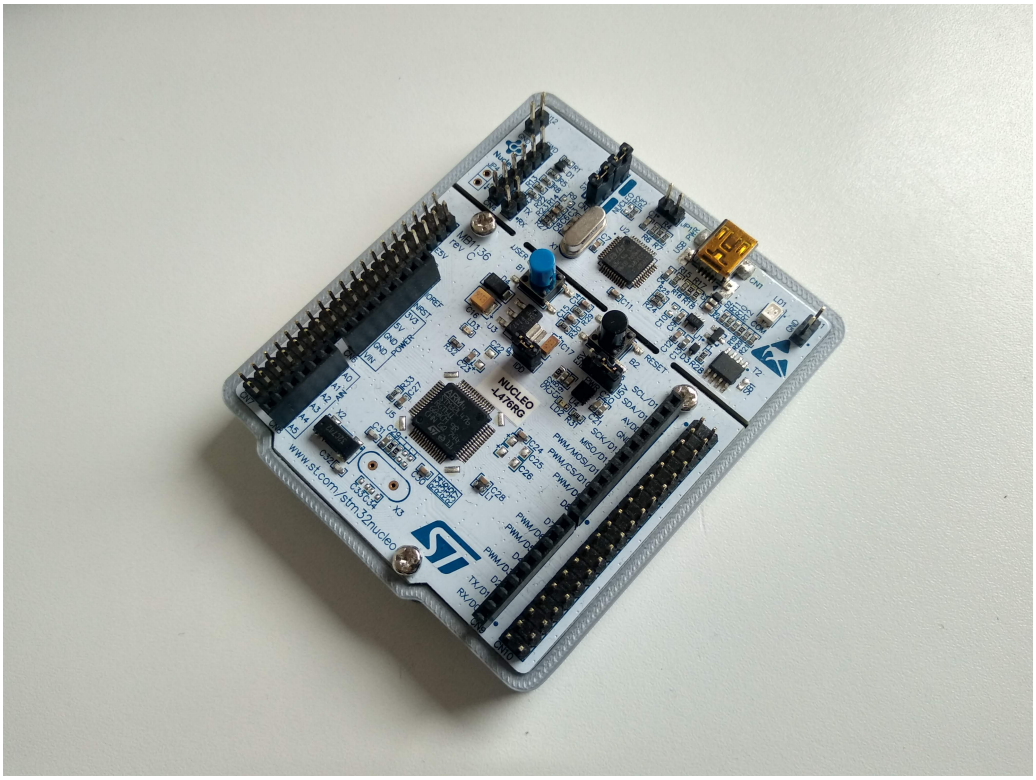
5.3 Použitá deska

Všechny výše zmíněné požadavky bezpečně splňuje deska STM32L476RGT6 (Obrázek 5.1). V příloze B je k dispozici rozložení pinů dané desky.

Tato deska obsahuje:

- 80MHz Cortex-M4 microcontroller s 1MB Flash pamětí a 128 KB SRAM
- Hodiny reálného času
- 2 32-bitové časovače pro obecné použití
- 2 16-bitové časovače určené pro ovládání motorů
- 7 16-bitových časovačů pro obecné použití
- 2 16-bitové nízko energetické časovače

Obrázek 5.1: STM32L476RGT6



5.3.1 Alternativní deska

Na katedře informatiky již byla dostupná deska s označením STM32F429ZI, o které se původně myslelo, že by se mohla použít jako základ stimulátoru. Tato deska disponuje všemi potřebnými komponentami pro vytvoření stimulátoru. Dále obsahuje velké množství dalšího užitečného hardwaru, jako je například dotykový display. V původní myšlence byl použit display k nastavování a ovládání experimentů. Z této myšlenky jsme byli nakonec nuceni ustoupit, protože jsme zjistili, že není možné provozovat paralelně display a všechny potřebné časovače v jednu chvíli. Display na desce komunikuje přes paralelní port, který zabere obrovské množství pinů, takže již nebylo možné použít časovače pro ovládání jednotlivých výstupů.

6 Implementace HW stimulátoru

Programování embeded zařízení se od standardního programování desktopových aplikací liší v několika podstatných rozdílech. Každé embeded zařízení obsahuje různé množství přídavných periferií. Aby bylo možné přistupovat k těmto periferiím, dodává výrobce embeded zařízení základní programové vybavení, pomocí kterého lze velmi elegantně k těmto periferiím přistupovat. Samozřejmě je možné přistupovat k jednotlivým periferiím na nejnižší možné úrovni - assembleru, ale to je velmi nepraktické.

Firma STM proto nabízí velmi kvalitní softwarové vybavení k veškerým embeded produktům. Základem všech zařízení (včetně embeded) je tzv. HAL (Hardware Abstraction Layer). Tato vrstva sjednocuje rozhraní napříč všemi zařízeními a jejich periferiemi. Vrstva definuje pouze rozhraní, konkrétní implementace je pak na samotném výrobci zařízení.

Firma STM vytvořila pomocný software STM32CubeF4[1], ve kterém je možné naklikat, jaké periferie se budou používat a následně vygenerovat předpřipravený projekt využívající HAL implementaci. Takto vygenerovaný projekt lze naimportovat do vývojového prostředí System Workbench for STM32[2]. V tomto IDE je následně možné využít připravené periferie v kódu. Příklad vygenerovaného kódu je v náhledu 6.1. Projekt obsahuje ještě velké množství dalších podpůrných souborů potřebných pro správný běh výsledné aplikace.

[1] https://www.st.com/en/embedded-software/stm32cubef4.html
[2] https://www.st.com/en/development-tools/sw4stm32.html

Zdrojový kód 6.1: Vygenerovaný kód aplikací STM32CubeF4

```
#include "main.h"
#include "stm32f4xx_hal.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void) {

  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1) {
    // nekonecna smycka
  }
}
```

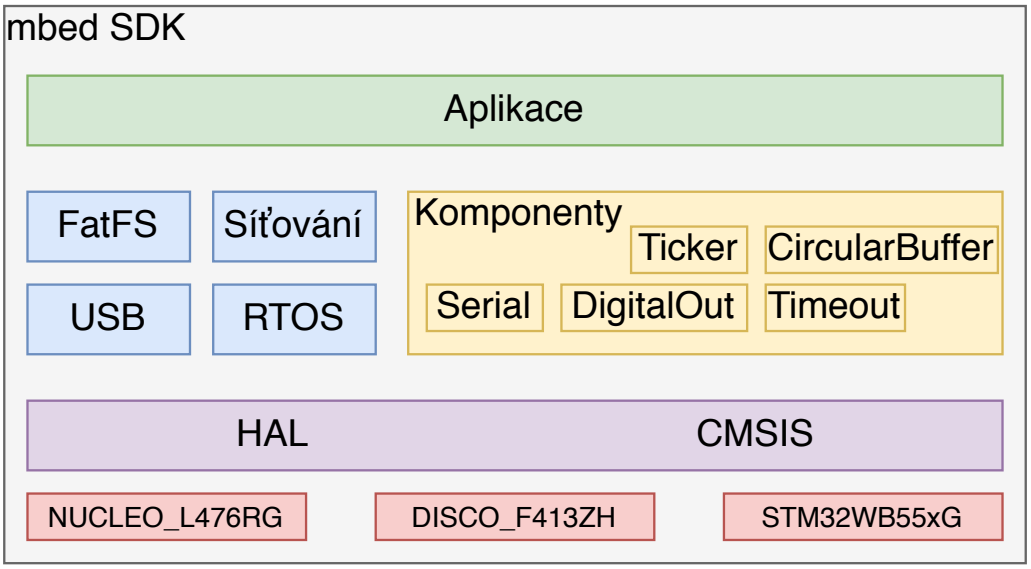
Při použití tohoto způsobu programování nebylo dosaženo dostatečné efektivity vývoje. Vedle přímého používání implementace HAL existují další možnosti, jak programovat embeded zařízení.

6.1 Mbed framework

Jedna z možností je použití frameworku Mbed[1], který velmi elegantním způsobem schovává HAL vrstvu do frameworku. Další výhoda frameworku spočívá ve využití jazyka C++. Celý framework je postavený na objektech, kde každá periferie je reprezentována jedním objektem.

Na obrázku 6.1 je grafické znázornění Mbed frameworku a jeho komponent. V nejvyšší vrstvě se nachází samotná uživatelská aplikace. Pod aplikační vrstvou se nachází jednotlivé komponenty, kterými Mbed framework disponuje. Tyto komponenty jsou hardwarově nezávislé - algoritmy jsou stejné pro všechny zařízení. Jediná podmínka je, aby daná komponenta byla fyzicky přítomna v zařízení. Vedle komponent se nachází ještě implementace virtuálních komponent, které lze v rámci frameworku také použít. Vedle HAL se nachází ještě jedna vrstva: CMSIS[1] CORE. Tato vrstva je dodávaná ke každé desce, která je založená na procesoru ARM. Obsahuje (mimo jiné) mapování registrů a mapování pinů na desce. Dále obsahuje informace o periferiích, které jsou na desce dostupné. Zajišťuje k těmto periferiím přístup a jejich konfiguraci.

Obrázek 6.1: Komponenty Mbed frameworku



V kódu 6.2 je srovnání inicializace jednoho výstupního pinu.

Zdrojový kód 6.2: Srovnání inicializace LED výstupu HAL a Mbed

```
1  // HAL
2  __HAL_RCC_GPIOA_CLK_ENABLE();
3
4  GPIO_InitTypeDef GPIO_InitStruct;
5
6  GPIO_InitStruct.Pin = GPIO_PIN_5;
7  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
8  GPIO_InitStruct.Pull = GPIO_PULLUP;
9  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
10 HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
11
12 // Mbed
13 DigitalOut led(PC_7);
```

Při použití HAL je potřeba nadefinovat veškeré vlastnosti dané struktury a nakonec zavolat `HAL_GPIO_Init()`. Mbed toto dělá také, ale na pozadí. Framework Mbed je využit k naprogramování celého HW stimulátoru.

[1]https://www.mbed.com/en/
[1]Cortex Microcontroller Software Interface Standard

6.1.1 Možnosti vývoje s Mbed

Vývoj software pro embeded zařízení není úplně triviální záležitost. Je velmi pracné založit prázdný projekt, naimportovat veškeré potřebné soubory související s vybranou deskou a začít tvořit software, jako v případě tvorby standardní desktopové aplikace. Proto existují pluginy, pomocný software, či rovnou celá IDE, které vyřeší veškeré importy a nastavení při zakládání nového projektu. V rámci projektu se ale používají další externí knihovny, nebo celé frameworky. Při výběru pluginu či IDE je třeba dávat si pozor nejenom na možnosti tvorby projektu ale také jeho udržovatelnost.

System Workbench for STM32

Základní IDE, které firma STM doporučuje pro tvorbu software se jmenuje *System Workbench for STM32*. IDE je založeno na Eclipse. Eclipse je vývojové prostředí umožňující vývoj ve velkém množství jazyků díky velkému množství vytvořených pluginů. Po pár testech bylo IDE zavrženo jako nepoužitelné k tvorbě většího projektu. Jakákoliv interakce s programem trvala nepřirozeně dlouho.

Mbed Online Compiler

Mbed Online Compiler dostupný na webové adrese ide.mbed.com je dostupné zdarma všem uživatelům, kteří jsou zaregistrováni na stejnojmenné stránce. IDE je tvořeno přímo firmou Mbed, takže veškeré tvořené projekty jsou založeny na Mbed frameworku. Výhodou tohoto vývojového prostředí je snadná dostupnost a integrace knihoven třetích stran. HW část této práce byla ze tří čtvrtin napsaná za pomoci tohoto online IDE. Projekt je možné zkompilovat a výsledný binární soubor stáhnout. Nahrání souboru do STM desky je velmi jednoduché. Po připojení desky přes USB se v počítači jeví jako malé externí úložiště. Stačí tedy vzít binární soubor a nakopírovat ho do tohoto úložiště. O vše ostatní se již postará deska sama. Tento způsob programování lze používat pouze do doby, než se začnou odstraňovat bugy. Bohužel, to je i největší nevýhoda online IDE - není možné debugovat program, tedy krokovat řádek po řádce. Na místo debugování bylo potřeba vyvinout kvalitní logování, pomocí kterého lze snadno zjistit, kterými místy program prošel a kterými už ne. Problém nastal, když program přistoupil mimo přidělenou paměť. Takové chyby se velmi těžko hledaly.

Platform.io

Další a zřejmě asi i nejuniverzálnější způsob je využití platformy Platform.io. *Platform.io* nabízí celý ekosystém pro tvorbu programů nejenom pro embeded zařízení. Do ekosystému patří IDE s debuggerem, které je dostupné pro všechny tři hlavní operační systémy. Dále je k dispozici analyzátor kódu, podpora unit testování, multiplatformní build systém s podporou sestavování aplikací na více architektur. V neposlední řadě jsou k dispozici nástroje k inspekci paměti v zařízení. Celá platforma je dostupná pro všechny zdarma pod volnou licencí Apache 2.0.

Platform.io je dostupné jako plugin do IDE VSCode vyvinuté firmou Microsoft. Nabízí podporu pro téměř 20 frameworků a 785 embeded zařízení. Mezi podporovanými frameworky se nachází i Mbed framework.

Platform.io je naprogramováno v jazyku Python a ke svému běhu nepotřebuje žádné další knihovny. Veškeré potřebné programy pro kompilaci, spuštění a debugování výsledného produktu si platforma zajistí sama přesně pro potřeby operačního systému, na kterém běží. To je velká výhoda. Tímto způsobem lze vyvíjet software napříč operačními systémy s podporou velkého množství embeded zařízení. Díky přechodu vývoje na tuto platformu bylo odhaleno nemalé množství zanesených chyb při používání Mbed Online Kompileru.

6.1.2 Použité komponenty

Na následujících řádcích budou popsány jednotlivé komponenty/periferie použité v aplikaci.

DigitalIn a DigitalOut

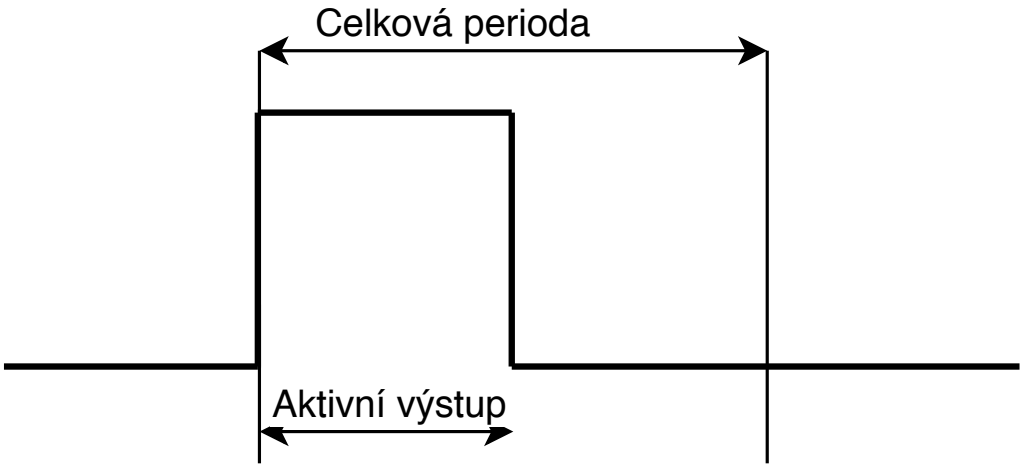
Komponenty `DigitalIn` a `DigitalOut` patří mezi základní komponenty pro komunikaci s vnějšími periferiemi. Komponenta `DigitalIn` reprezentuje jeden PIN v zařízení, pomocí kterého lze číst data z periferie. `DigitalOut` umožňuje naopak zapsat data do periferie. Přenášená data můžou mít pouze dvě různé hodnoty: 0 a 1. Pro využití více hodnot je třeba použít jinou komponentu.

PwmOut

`PwmOut` umožňuje ovládat daný PIN pomocí pulsně-šířkové modulace. Pomocí `DigitalOut` je možné při ovládání LED, buď LED rozsvítit úplně, nebo zhasnout. `PWM` umožňuje ovládat svítivost LED. Může se tedy rozsvítit jenom na 50%.

Mezi další vymoženosti `PwmOut` patří nastavení periody blikání s přesností od vteřin až po mikrosekundy. V tomto režimu je možné nastavit dobu, po kterou bude LED svítit a kdy bude zhasnuta. Dobu trvání rozsvícené LED lze nastavit vzhledem k celkové době periody relativně nebo absolutně. Na obrázku 6.2 je znázorněna jedna perioda impulsu. PWM výstup začíná vzestupnou hranou do aktivního stavu. Po definovanou dobu se drží aktivní stav, poté přejde do neaktivního stavu.

Obrázek 6.2: Návrh zapojení stimulátoru



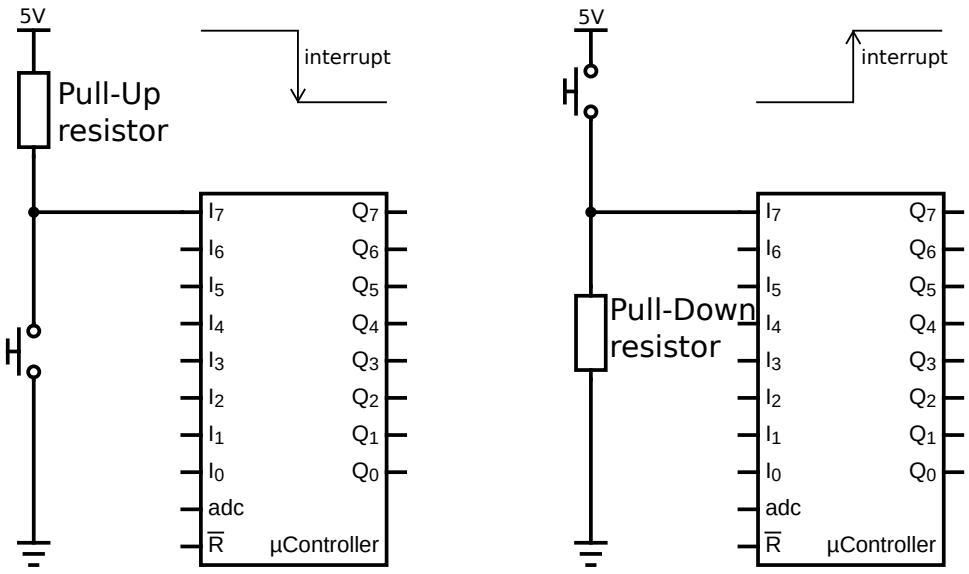
InterruptIn

Typickým vybavením aplikací jsou tlačítka. Jak ale zjistit, kdy bylo tlačítko stisknuto? Odpovědí je více. Nejjednodušší metoda je tzv. pooling. V nekonečné smyčce se ptáme tlačítka, zda náhodou nebylo stisknuto. V případě stisku se vykoná kód. Tato metoda je velmi problémová. Jednak musí být dotazování velmi rychlé, dále se může stát vlivem rušení, že bude přečtena jiná hodnota, která nemusí vyústit v rozpoznání stisku tlačítka. Mnohem lepší řešení je reakce na tlačítko pomocí přerušení. K tomu slouží `InterruptIn`.

U přerušení je třeba si ujasnit, v jaké situaci se má vygenerovat. Nabízí se dvě možnosti: při náběžné hraně nebo při sestupné hraně. Na obrázku 6.3 je příklad zapojení tlačítka s pull-up resistorem (vlevo) a s pull-down resistorem (vpravo).

Zásadní rozdíl v zapojení je ve výchozí hodnotě napětí na připojeném pinu. V zapojení s pull-up resistorem je základní hodnota `HIGH` neboli logická 1. Zapojení s pull-down resistorem se chová přesně opačně, základní hodnota je `LOW` neboli logická 0. Při stisku tlačítka dojde ke změně hodnoty a $1 \rightarrow 0$

Obrázek 6.3: Rozdíl mezi zapojením s Pull-Up a Pull-Down resistorem



případně $0 \rightarrow 1$. V prvním případě mluvíme o změně za pomocí sestupné hrany, v případě druhém se jedná o změnu pomocí náběžné hrany.

Deska použitá pro vývoj stimulátoru STM32L476RGT6 má v interním zapojení u každého pinu resistor. Ve zdrojovém kódu je možné nastavit, jaký typ resistoru se použije, zdali pull-up nebo pull-down.

Ticker

`Ticker` umožňuje periodicky spouštět zvolenou funkci. Spouštění funguje na principu přerušení. Na desce se vezme časovač, kterému se nastaví perioda generování přerušení. V každém přerušení se vykoná požadovaná funkce. Na funkci se kladou určitá omezení:

- funkce musí být co možná nejkratší,
- funkce nesmí být blokující,
- ve funkci se nesmí použít žádné alokátory paměti (`malloc` a `new`)
- ve funkci není možné uspat mikrokontroler do hlubokého spánku.

Timeout

`Timeout` dává možnost odložit spuštění zvolené funkce do budoucna. Na funkci se vztahují stejná pravidla jako v případě `Tickeru`.

Serial

Objekt `Serial` reprezentuje komunikační sériovou linku. V sériové komunikaci se přenáší data postupně bit po bitu. Mezi důležité parametry patří:

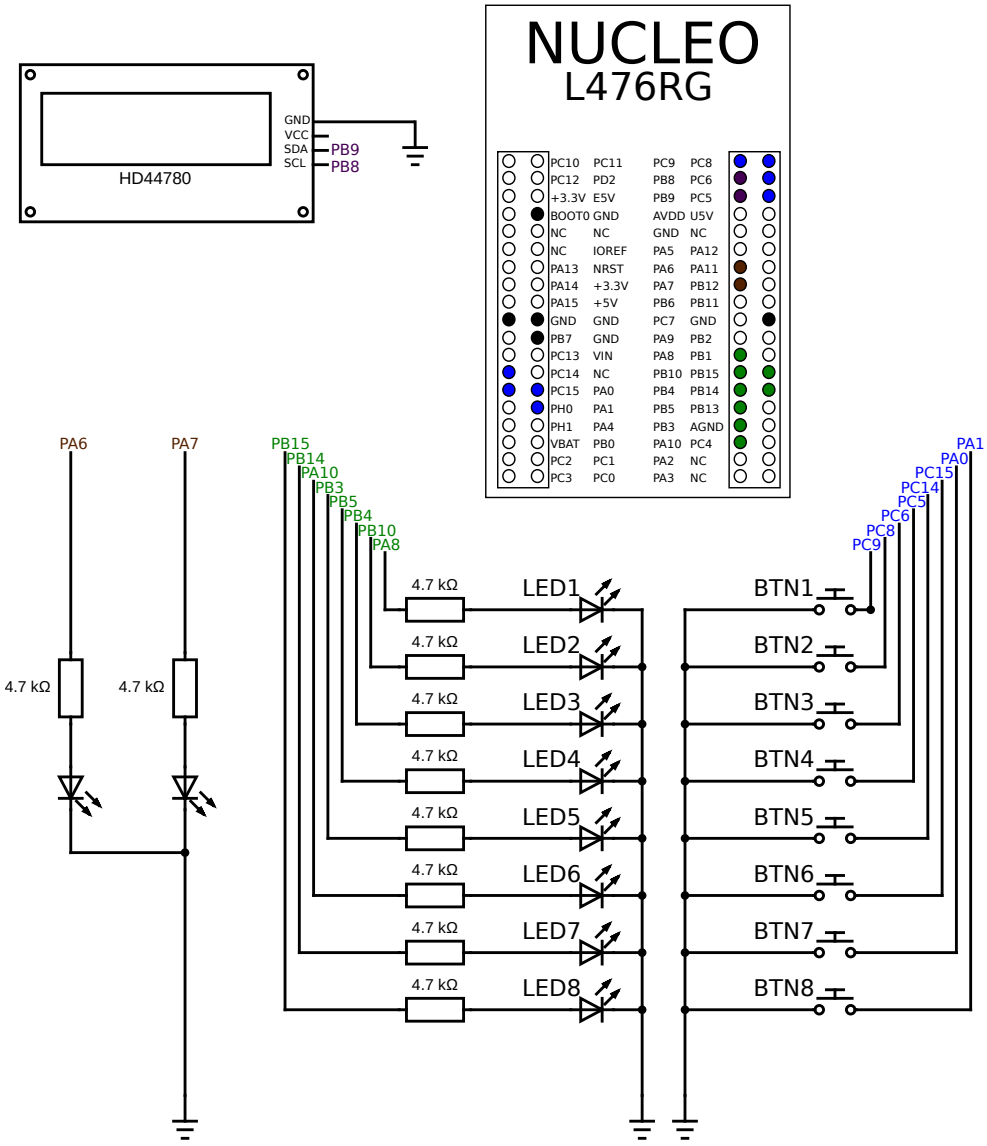
- baudrate - určuje přenosovou rychlost v bitech za sekundu,
- dataLength - velikost jednoho rámce dat; 7 nebo 8 bytů,
- parity - na konec datového rámce lze vložit paritní bit; v závislosti na konfiguraci (lichý nebo sudý) nabude paritní bit hodnoty tak, aby v případě liché parity byl celkový počet jedniček ve slově lichý, případně pro sudou paritu sudý,
- stopBits - za paritním bitem můžou následovat jeden až dva stop bity, sloužící k synchronizaci linky.

6.2 Návrh zapojení stimulátoru

Na obrázku 6.4 je vidět návrh zapojení stimulátoru. Celý stimulátor se skládá z 8 LED a 8 tlačítek. Ke každé LED je přiřazeno jedno tlačítko. Dále se v zapojení nachází dvě indikační LED. První dioda reprezentuje připravenost stimulátoru. Rozsvítí se v momentě, kdy je stimulátor připraven k použití. Druhá LED se rozsvítí vždy, když probíhá datový přenos ve směru ze stimulátoru do kontrolního zařízení. Každá LED je připojena přes odpor o hodnotě 220Ω. Piny pro stimulační LED jsou vyznačeny zelenou barvou. Modrou barvou jsou vyznačeny piny pro reakční tlačítka. Dále je počítáno s připojením dvouřádkového displaye. Display by plnil funkci informativního charakteru. Bude zobrazovat zejména stav, ve kterém se stimulátor aktuálně nachází. Ve výsledné práci ale display není připojen, pouze je vytvořeno základní komunikační rozhraní.

Ke stimulátoru je možné připojit dvouřádkový display s 20 znaky na řádce. Display bude plnit funkci informativního charakteru. Zobrazovat bude zejména stav, ve kterém se stimulátor aktuálně nachází.

Obrázek 6.4: Návrh zapojení stimulátoru



Každá LED je připojena k pinu, který podporuje pulzně šířkovou modulaci (PWM). Díky PWM je možné regulovat intenzitu jasu LED. PWM je řešena pomocí časovačů dostupných na desce. Je důležité, aby se o každou LED staral jeden časovač. Tabulka 6.1 přehledně zobrazuje, který časovač ovládá kterou LED.

Tabulka 6.1: Mapování časovačů a LED

#	LED	TIMER
1	PA_8	TIM1_CH1
2	PB_10	TIM2_CH3
3	PB_4	TIM3_CH1
4	PB_5	TIM3_CH2
5	PB_3	TIM2_CH2
6	PA_10	TIM1_CH3
7	PB_14	TIM1_CH2N
8	PB_15	TIM1_CH3N

6.3 Stavy stimulátoru

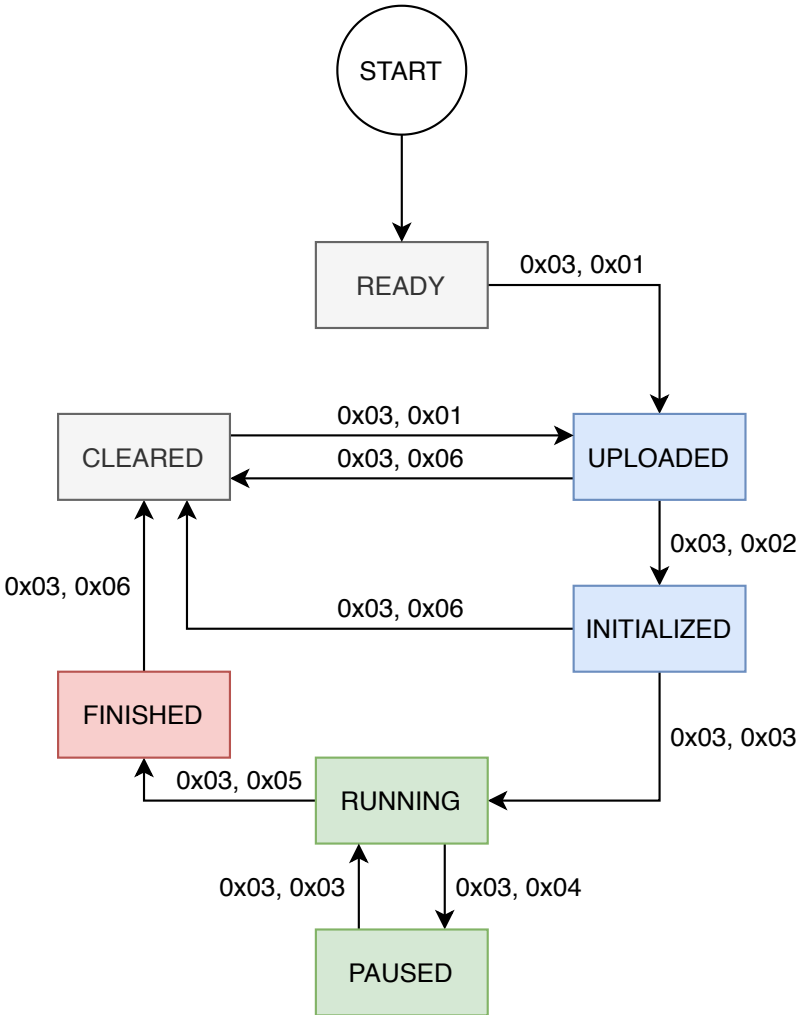
Stimulátor je navržen jako stavový automat. Diagram stavů stimulátoru je zobrazen na obrázku 6.5.

Stavů existuje celkem sedm:

1. READY - do tohoto stavu se dostane stimulátor po zapnutí napájení, jakmile je připraven k použití
2. UPLOADED - jakmile se do stimulátoru nahraje konfigurace experimentu
3. INITIALIZED - konfigurace byla inicializována
4. RUNNING - experiment běží
5. PAUSED - experiment se pozastavil
6. FINISHED - experiment se ukončil
7. CLEARED - experiment byl vymazán

Přechodu mezi stavy se dosáhne posláním příkazu na stimulátor, který se přeloží na zavolání konkrétní funkce.

Obrázek 6.5: Diagram stavů stimulátoru



6.4 Datové struktury

Při programování stimulátoru bylo potřeba dbát na přehlednost kódu. Velkou výzvu tvořilo uložení konfigurace jednotlivých experimentů. Stimulátor podporuje pět různých experimentů, což odpovídá pěti různým konfiguracím. Každá konfigurace obsahuje parametry pro daný experiment.

První možnost, jak tyto konfigurace uložit se nabízí pole, ve kterém budou parametry lineárně za sebou. Přínos tohoto řešení spočívá v jeho jednoduchosti. Při nahrání experimentu do stimulátoru se uloží do paměti pole bytů, které reprezentuje konfiguraci a víc se nemusí řešit. Nevýhoda tohoto řešení se projeví až při čtení pole. Ne všechny parametry lze uložit do hodnoty o velikosti jednoho bytu. V takovém případě by se musely vytvořit funkce pro přečtení vícebytové hodnoty z pole.

Místo použití pole byly zavedeny struktury, které přímo popisují konfigurace experimentů. Struktury a pole jsou v jazyce C totéž z pohledu paměti. V obou případech se jedná o souvislý blok paměti. U pole se k prvkům přistupuje pomocí indexů. Ve struktuře jsou jednotlivé prvky již pojmenované a s předem známou velikostí.

V kódu 6.3 je vidět struktura konfigurace pro T-VEP experiment. Konfigurace je rozdělena celkem do tří samostatných struktur. První struktura `experiment_tvep_head_s` obsahuje obecné parametry pro T-VEP konfiguraci. Struktura `experiment_tvep_output_s` popisuje parametry jednoho výstupu, který je v experimentu použit. Poslední struktura `experiment_tvep_s` zabaluje předchozí dvě struktury pod jednu hlavní strukturu. Takto jsou vytvořeny struktury i pro ostatní konfigurace experimentů.

V kódu 6.3 je použito makro `#pragma pack(push/pop)`. Kompilátor během překladu v rámci optimalizace kódu zarovnává jednotlivé proměnné do bloku. To znamená, že struktura o dvou prvcích s datovým typem **byte** by se například zarovnala na čtyřbytovou hodnotu. Toto chování je ale potřeba potlačit, protože pak by nebylo možné přistupovat do paměti v poli pomocí struktur. Toto chování je právě potlačeno pomocí makra `#pragma pack(push /pop)`.

Zdrojový kód 6.3: Struktura konfigurace pro T-VEP experiment

```
#pragma pack(push, 1)
typedef struct experiment_tvep_head_s {
    experiment_type_t type;
    experiment_output_count_t outputCount;
} experiment_tvep_head_t;
#pragma pack(pop)

#pragma pack(push, 1)
typedef struct experiment_tvep_output_s {
    experiment_tvep_output_pattern_length_t
       patternLength;
    experiment_output_type_t outputType;
    experiment_output_out_t out;
    experiment_output_wait_t wait;
    experiment_output_brightness_t brightness;
    experiment_tvep_output_pattern_t pattern;
} experiment_tvep_output_t;
#pragma pack(pop)

#pragma pack(push, 1)
typedef struct experiment_tvep_s {
    experiment_tvep_head_t head;
    experiment_tvep_output_t outputs[
       TOTAL_OUTPUT_COUNT];
} experiment_tvep_t;
#pragma pack(pop)
```

Všechny prvky ve struktuře používají vlastnoručně nadefinované datové typy. Tímto krokem bylo docíleno snadnější úpravy kódu. V případě, kdy bude potřeba změnit datový typ jednoho parametru, stačí udělat změnu pouze na jednom místě v kódu. V kódu 6.4 je vidět příklad definice vlastních datových typů použitých ve stimulátoru.

Zdrojový kód 6.4: Příklad definice vlastních datových typů

```
typedef uint8_t experiment_type_t;
typedef uint8_t experiment_output_count_t;
typedef uint8_t experiment_output_type_t;
typedef uint8_t experiment_output_brightness_t;
```

6.4.1 Uložení struktur konfigurací experimentů

Všechny konfigurace experimentů sdílejí jeden vymezený paměťový prostor. Toto sdílení je založeno na předpokladu, že v jednu chvíli může být ve stimulátoru nahrán a spuštěn pouze jeden experiment. Jazyk C má pro takovéto sdílení prostoru datový typ `union`. `Union` bude mít vždy pevně stanovenou velikost paměti, kterou bude zabírat. Velikost struktury odpovídá největšímu datovému prvku uloženému v `union`u. To znamená, že když bude `union` obsahovat dva prvky, kde první prvek bude typu `byte` a druhý typu `float`, celková velikost `union`u budou 4 byty.

V kódu 6.5 je vidět uložení konfigurací experimentů. `Union` obsahuje celkem šest atributů. Pět konfigurací (pro každý typ experimentu jedna) a atribut *type*. Pomocí atributu *type* se definuje, ke které konfiguraci v rámci `union` struktury se má přistupovat. Aby vše fungovalo správně, je nutné zajistit, že tento atribut *type* se bude vždy vyskytovat i v jednotlivých konfiguracích, a to na prvním místě.

Zdrojový kód 6.5: Struktura pro uložení všech experimentů

```
1 union ExperimentConfig {
2     experiment_type_t type;
3     experiment_erp_t  experimentERP;
4     experiment_cvep_t experimentCVEP;
5     experiment_fvep_t experimentFVEP;
6     experiment_tvep_t experimentTVEP;
7     experiment_rea_t  experimentREA;
8 };
```

6.5 Implementace experimentů

V této sekci bude podrobně popsána implementace jednotlivých experimentů na hardwarovém stimulátoru. Vždy, když příjde příkaz *správy experimentu* s typem **INIT** zavolá se inicializace vybraného experimentu. Typicky se aktivují časovače s vypočítanou periodou pro celý experiment, nebo konkrétní výstup. Časovačům se nastavují callback funkce, které se zavolají vždy, když uběhne požadovaný interval. Tyto funkce jsou nazvány *ticker*. Každý experiment je složen z jedné až 8 funkcí *ticker*. Počet funkcí je ovlivněn počtem stimulů, který bude v experimentu použit. CVEP a REA používají pouze jednu *ticker* funkci.

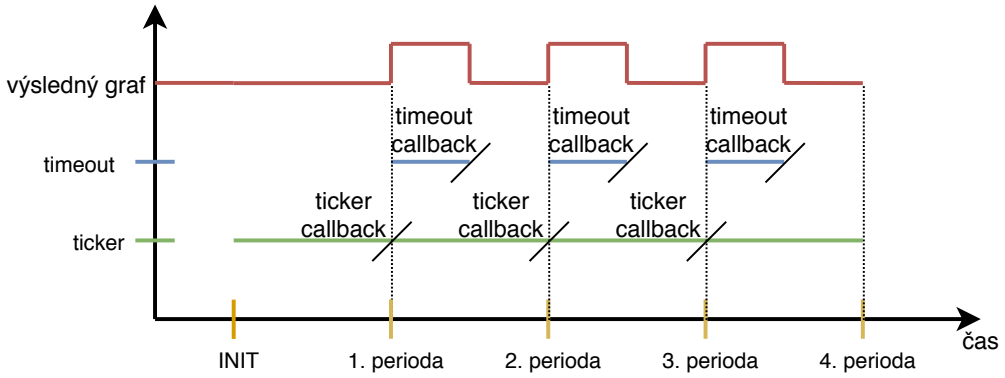
Ticker funkce reprezentuje aktivaci stimulu. To znamená, že se fyzicky stimul aktivuje a informace o aktivaci stimulu se odešle na server. Dále se ve

funkci získá doba, po kterou musí být stimul aktivní. Ve funkci se provedou další akce specifické pro konkrétní experiment. Vždy se ale spustí *timeout*, s dobou aktivního výstupu.

Timeout je opět reprezentován funkcí, která se zavolá po vypršení zadaného času. V této funkci se výstup deaktivuje a zároveň se odešle informace o vypnutí výstupu na server. Pokud je potřeba, vykonají se akce specifické pro příslušný experiment.

V grafu 6.6 je graficky znázorněno, jak se spouští *ticker* a *timeout* funkce v čase.

Obrázek 6.6: Graf znázorňující spouštění funkcí ticker a timeout



6.5.1 Experiment CVEP

CVEP experiment 6.6 obsahuje pouze jednu *ticker* a *timeout* funkci.

V *ticker* funkci se projdou všechny stimuly a aktivují se podle hodnoty v patternu, která odpovídá jednotlivým stimulům. Dále se zvýší interní counter indexu pro pattern.

Při zavolání timeout funkce se všechny stimuly deaktivují.

Zdrojový kód 6.6: Pseudokód CVEP experimentu

```
void init() {
    startTicker();
}
void ticker() {
    readCurrentPattern();
    activateSelectedStimulus();
    sendInformationToServer();
    startTimeout();
}

void timeout() {
    deactivateAllStimulus();
    sendInformationToServer();
}
```

6.5.2 Experiment FVEP

FVEP experiment 6.7 používá až osm *ticker* a *timeout* funkcí. Každá funkce ovládá jeden stimul.

V *ticker* funkci se aktivuje odpovídající stimul a nastaví *timeout*, který stimul opět deaktivuje.

Zdrojový kód 6.7: Pseudokód FVEP experimentu

```
void init() {
    startTickers();
}
void ticker() {
    activateOneStimul();
    sendInformationToServer();
    startTimeout();
}

void timeout() {
    deactivateOneStimul();
    sendInformationToServer();
}
```

6.5.3 Experiment TVEP

TVEP experiment 6.8 je velmi podobně implementován jako FVEP experiment. Také používá až osm *ticker* a *timeout* funkcí pro nezávislé ovládání jednotlivých stimulů.

Rozdíl je v *ticker* funkci. Na začátku funkce se na základě patternu a interního čítače určí, zdali se bude stimul aktivovat či nikoliv. Pokud se stimul nebude aktivovat, žádný timeout nebude nastaven.

Při zavolání timeout funkce deaktivuje pouze vybraný stimul.

Zdrojový kód 6.8: Pseudokód TVEP experimentu

```
1  void init() {
2      startTickers();
3  }
4  void ticker() {
5      readPattern();
6      if (activateStimul) {
7          activateOneStimul();
8          sendInformationToServer();
9          startTimeout();
10     }
11 }
12
13 void timeout() {
14     deactivateOneStimul();
15     sendInformationToServer();
16 }
```

6.5.4 Experiment REA

REA experiment 6.9 využívá pouze jednu *ticker* a *timeout* funkci. Ticker funkce volána periodicky s časem použitím z parametru pro maximální dobu aktivního stimulu.

V *ticker* funkci se aktivuje náhodně vybraný stimul. Reálná doba aktivního výstupu se určí náhodně pomocí následujícího postupu:

$$delta = waitTimeMax - waitTimeMin$$

$$period = waitTimeMin + rand()\ \%\ delta$$

Nejdříve se vypočte tzv. dynamická složka, pomocí které se bude upravovat čas aktivního výstupu. Výsledná doba timeoutu se vypočítá jako součet

minimální doby a náhodného čísla vymodulovaného o deltu. Díky této náhodnosti bude doba aktivního výstupu trvat různou dobu.

Timeout funkce deaktivuje výstup a oznámí to serveru. Dále zkontroluje, zdali bylo stisknuto správné tlačítko včas. Pokud nebylo stisknuto správné tlačítko, měl by se stimulátor zachovat podle nastavení. Buď bude pokračovat dále v experiementu, nebo počká definovanou dobu po které bude experiment pokračovat. V odevzdané práci je tento parametr vynechán a experiment pokračuje standardně za každé situace.

Zdrojový kód 6.9: Pseudokód REA experimentu

```
1  void init() {
2      startTicker();
3  }
4  void ticker() {
5      selectRandomStimul();
6      activateSelectedStimul();
7      sendInformationToServer();
8      startTimeout();
9  }
10
11 void timeout() {
12     deactivateStimul();
13     sendInformationToServer();
14 }
```

6.5.5 Experiment ERP

ERP experiment 6.11 má odlišně implementován časování. V tomto experimentu se nevolá žádná funkce periodicky. Vše je řízeno přes timeouty. Toto řešení bylo nutné zvolit kvůli rozdílným periodám jednotlivých stimulů. Další odlišnost spočívá v použití pouze jedné *ticker* funkce a až osmi *timeout* funkcí. Název pro ticker funkci je zachován pouze kvůli kompatibilitě. Reálně se funkce *tickerERP* a *timeoutERP* volají při timeoutu.

ERP experiment vyžaduje ke svému běhu sekvenci stimulů, podle které se aktivují jednotlivé stimuly. Struktura pro ERP experiment obsahuje kromě konfigurace experimentu ještě interní pomocná data, do kterých jsou uloženy důležité pracovní hodnoty experimentu. Struktura je v kódu 6.10

Zdrojový kód 6.10: Struktura interních dat pro ERP experiment

```
#pragma pack(push, 1)
typedef struct experiment_erp_sequence_data_s {
    uint16_t pointer;
    uint16_t accIndex;
    uint16_t accOffset;
    uint16_t requestIndex;
    uint16_t requestOffset;
} experiment_erp_sequence_data_t;
#pragma pack(pop)

#pragma pack(push, 1)
typedef struct experiment_erp_runtime_data_s {
    ushort randomBase;
    uint8_t currentOutput;
    experiment_erp_sequence_data_t sequence_data;
    uint32_t accumulators[TOTAL_OUTPUT_COUNT];
} experiment_erp_runtime_data_t;
#pragma pack(pop)
```

Sekvence experimentu je uložena v proměnné *accumulators*. Jedná se o pole, které odpovídá velikosti maximálnímu počtu výstupů. Jednotlivé položky v poli jsou typu **uint32_t**, tedy 32 bitů. Do jedné položky v poli lze uložit informaci o deseti stimulech v sekvenci. Sekvence je složena z čísel reprezentující jednotlivé stimuly. Tato čísla budou vždy z uzavřeného intervalu $< 0; 7 >$. Nejvyšší číslo sedm lze binárně zapsat jako 111, proto stačí tři byty na reprezentaci jednoho stimulu. Dohromady lze do pole *accumulators* uložit sekvenci o celkové délce 80 stimulů. Pokud by experiment obsahoval více stimulů, budou se další hodnoty postupně nahrávat v průběhu experimentu.

V *ticker* funkci se nejdříve získá číslo stimulu, který se bude aktivovat. Toto číslo se získá na základě proměnných *accIndex* a $accOffset$. První proměnná představuje index do pole *accumulators*. Offset říká na jaké pozici se číslo stimulu nachází. Výsledné číslo stimulu se tedy získá pomocí následujícího vzorce:

$$output = ((accumulators[index] >> (offset * 3)) \ \& \ 0xF)$$

Číslo zároveň slouží jako index do pole výstupů v konfiguraci experimentu. Z konfigurace výstupu se získá informace o svítivosti výstupu a hlavně době aktivního výstupu. Když jsou známy veškeré potřebné parametry, aktivuje se stimul a informace se odešle na server. Zároveň se spustí *timeout* ve kterém se stimul deaktivuje.

Nakonec se aktualizují údaje o sekvenci. To znamená inkrementovat hodnotu proměnné $accOffset$. Pokud tato proměnná nabude hodnoty 10 (počet stimulů v jedné položce), tak se hodnota vynuluje a inkrementuje se hodnota v proměnné $accIndex$. Pokud proměnná $accIndex$ nabude hodnoty $TOTAL\_OUTPUT\_COUNT$, hodnota se také vynuluje. Vždy, když se aktualizuje hodnota proměnné $accIndex$, vyšle se příkaz k získání další části sekvence.

V *timeout* funkci se deaktivuje dříve aktivovaný stimul. Dále se zkontroluje, zdali experiment nedošel na konec sekvence a pokud ano, experiment bude ukončen. V opačném případě se nastaví nový timeout, tentokrát s callbackem funkce $tickerERP$. Tímto se vytvoří virtuální smyčka, která zajistí přehrávání experimentu.

Zdrojový kód 6.11: Pseudokód ERP experimentu

```
1  void init() {
2      startTicker();
3      requestSequencePart();
4  }
5  void ticker() {
6      readCurrentStimul();
7      activateSelectedStimulus();
8      sendInformationToServer();
9      startTimeout();
10     updateSequenceCounters();
11 }
12
13 void timeout() {
14     deactivateStimul();
15     sendInformationToServer();
16     checkSequenceEnd();
17     startTicker();
18 }
```

6.6 Komunikace stimulátoru a Raspberry Pi

Aby mohl server a stimulátor mezi sebou komunikovat, bylo potřeba navrhnout komunikační protokol. V této sekci budou popsány vlastnosti, které musí protokol splňovat. Dále bude znázorněno schéma komunikace mezi stimulátorem a serverem. Výsledný komunikační protokol je k dispozici v příloze E. Komunikace s Raspberry Pi probíhá pomocí sériové linky v asynchronním módu.

6.6.1 Analýza komunikačního protokolu

Komunikace bude probíhat na bázi packetů. Každý packet bude obsahovat právě jeden příkaz. Packet se bude skládat z hlavičky, těla a ukončovacího znaku/znaků. Hlavička se vždy bude skládat z jednoho bytu, pomocí kterého se bude určovat, o který příkaz se jedná. Následuje tělo, které nemá pevnou délku a obsahuje data, která se mají odeslat. Tělo může být prázdné. Packet je ukončen ukončovacím znakem. Limitující je přítomnost ukončovací sekvence. Tato ukončovací sekvence se nesmí vyskytnout v datech samotných. Další možností by bylo stanovit pevnou délku příkazu, čímž by odpadla potřeba ukončovacích znaků. Toto řešení má ale tu nevýhodu, že pro velmi krátký příkaz se bude vždy posílat zbytečně mnoho dat. Další problém by mohl nastat v případě, že by se do jednoho příkazu nevešla všechna data. Bylo by potřeba implementovat rozdělení jednoho příkazu do dvou čí více packetů.

6.6.2 Typy příkazů

Příkazy lze rozdělit do tří kategorií:

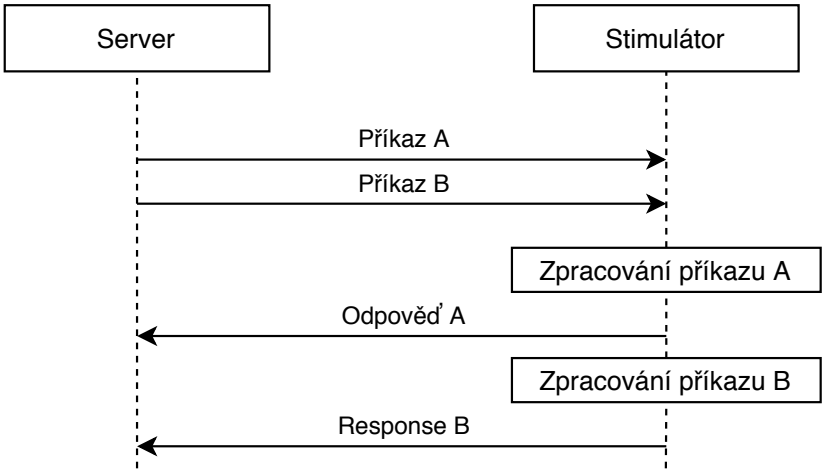
- **ovládání experimentu** - do této kategorie spadají příkazy, pomocí kterých se bude ovládat experiment (start, stop),
- **získání stavu experimentu** - zde se nachází příkazy pro získání aktuálního stavu stimulátoru, dále sem lze zahrnout podpůrné příkazy, které dokážou vypsat vybrané části paměti ze stimulátoru,
- **podpora pro sekvence** - do této kategorie patří příkazy pro podporu sekvencí, které jsou součástí ERP experimentů.

Do příkazů, které ovládají experiment je zahrnuto nahrávání konfigurací experimentů.

6.6.3 Schéma komunikace

Protokol je založen na schématu dotaz-odpověď. Server odešle do stimulátoru příkaz a stimulátor odešle na server odpověď. Protože se komunikuje v asynchronním módu, čekání na odpověď není blokující. Asynchronní mód je důležitý zejména pro stimulátor, na kterém může běžet experiment a zároveň komunikovat se serverem. Jediný příkaz, který může být blokující ze strany serveru je příkaz pro získání stavu experimentu. Pokud stimulátor na příkaz neodpoví do definovaného času, tak to znamená, že je se stimulátorem něco v nepořádku. Taková situace by ale nikdy neměla nastat. Na obrázku 6.7 je příklad schématu asynchronní komunikace mezi serverem a stimulátorem.

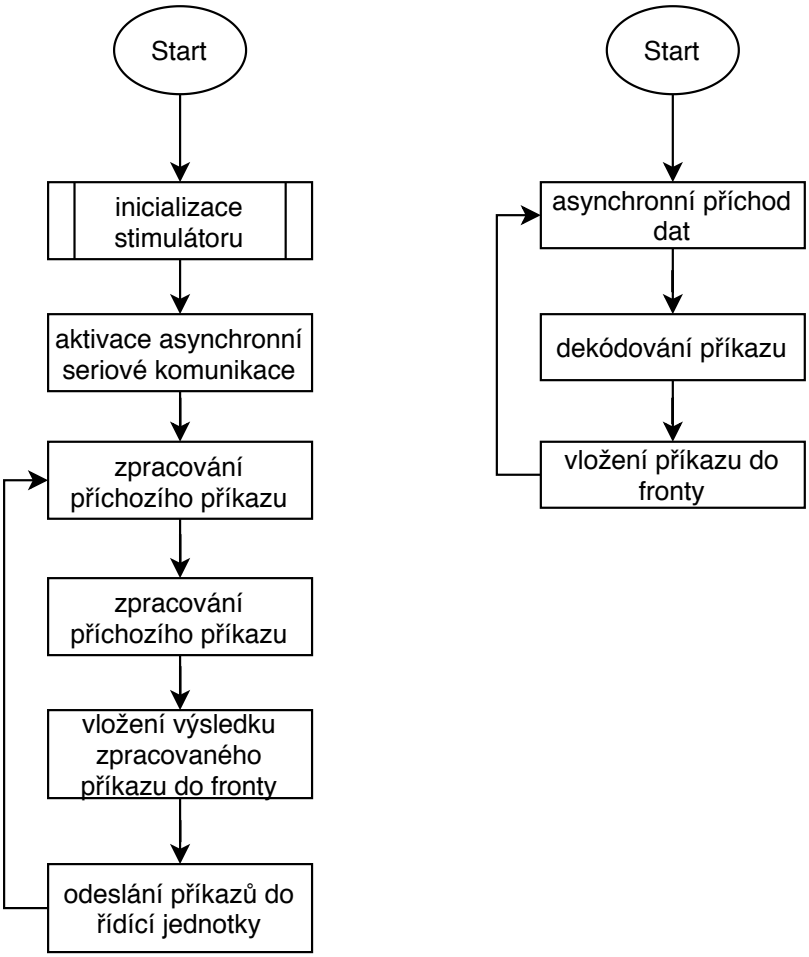
Obrázek 6.7: Schéma asynchronní komunikace mezi serverem a stimulátorem



6.7 Životní cyklus aplikace

Životní cyklus aplikace včetně zpracování příkazů je vidět na obrázku 6.8. Při startu stimulátoru se nejdříve inicializují veškeré periferie, jako jsou LED a reakční tlačítka. Po úspěšné inicializaci se aktivuje sériová komunikace. Nakonec se spustí nekonečná smyčka, ve které se neustále čtou příchozí příkazy. Příkazy se po vyjmutí z fronty zpracují. Pokud je výsledkem zpracovávaného příkazu odpověď ve formě dalšího příkazu, vloží se do výstupní fronty. Z výstupní fronty se opět přečte příkaz a odešle se do Raspberry Pi. Z každé fronty se při jednom průchodu zpracovává vždy jeden příkaz. Každá fronta má pevnou velikost a dokáže uložit maximálně 16 příkazů. Jedná se o cyklickou frontu implementovanou polem. Mbed framework obsahuje tuto frontu pod třídou `CircularBuffer`.

Obrázek 6.8: Životní cyklus aplikace a zpracování příkazů



7 Server

Server slouží jako prostředník mezi klientskou aplikací a HW stimulátorem. Je napsán v jazyce TypeScript s využitím frameworku NestJS a používá pro svůj běh prostředí NodeJS.

7.1 NodeJS

NodeJS je běhové prostředí vzniklé v roce 2009, založené na Chrome V8 enginu, který je napsaný v jazyce C++. JavaScript se do té doby považoval primárně za klientský jazyk, běžící pouze v prohlížeči. V roce 2009 se ale vše změnilo a JavaScript se dostal také na serverovou stranu. NodeJS umožňuje spuštění serverových aplikací napsaných v JavaScriptu. Hlavní myšlenka projektu je dát vývojářům k dispozici jednoduché prostředí pro tvorbu robustních a snadno škálovatelných síťových aplikací. NodeJS nabízí v základu velmi kvalitní softwarové vybavení v podobě modulů. Mezi základní moduly patří: fs (práce se souborovým systémem), path (práce s cestami k souborům a složkám), os (poskytuje informace o hostitelském systému), http (základní modul pro tvorbu web serveru) a events (práce s událostmi).

Všechny aplikace tvořené v NodeJS jsou událostně řízené. Při spuštění aplikace se vytvoří jedno hlavní vlákno, ve kterém běží hlavní smyčka. Veškeré asynchronní I/O operace, síťové a konkurentní operace jsou prováděny v separátním vlákně. Naštěstí je tohle všechno již vyřešeno "pod pokličkou"v enginu V8 za pomoci knihovny libuv.

7.1.1 NPM

Společně s NodeJS byl vytvořen systém pro správu knihoven (balíčků) třetích stran NPM (Node Package Manager). Pro aplikace psané v JavaScriptu je typické, že obsahuje stovky i tisíce závislostí. Takto enormní množství je způsobeno dlouhou historií jazyka JavaScript. Stalo se tradicí, že než se začne vyvíjet nějaká nová funkce, podívá se člověk na internet, zdali už na podobnou funkci někdo nenapsal knihovnu, která by šla využít.

Všechny projekty psané v NodeJS jsou definovány pomocí jediného souboru `package.json`. Ten mimo jiné obsahuje také závislosti, které projekt potřebuje ke svému běhu. Závislosti lze rozdělit na produkční a vývojové. Bez produkčních závislostí nelze aplikaci spustit. Vývojové závislosti slouží

zejména k ulehčení práce, případně k testování výsledné aplikace. Jazyk TypeScript je typický příklad vývojové závislosti.

7.1.2 JavaScript

Javascript je jazyk vytvořený na přelomu devadesátých let, kdy firma Netscape Communications potřebovala do svého prohlížeče Netscape Communicator přidat základní prvky dynamického webu, jako jsou například animace a interakce s DOM (Document Object Model). Za tímto účelem vznikla první verze jazyka JavaScript.

V JavaScriptu existují základní datové typy: **řetězce**, **čísla**, **boolean**, **pole**, **funkce** a **objekty**. Dále ještě existují datové typy pro prázdné hodnoty: **null** a **undefined**.

Na objekty lze nahlížet i jako na hash-mapy, neboli slovníky, kdy jednomu klíči náleží jedna hodnota z výše jmenovaných.

JavaScript používá objektově orientovaný model založený na prototypech. Z jazyka Java je známé, že všechny objekty ve výsledku dědí z jednoho společného předka, třídy `Object`. V JavaScriptu je dědičnost vyřešena pomocí prototypů. Každý neprázdný objekt či řetězec obsahují atribut `prototype`, pomocí kterého se lze odkázat na předka.

7.1.3 TypeScript

JavaScript je dynamický jazyk bez typové kontroly. Právě míchání datových typů v JavaScriptu je nejčastější příčina chyb. TypeScript, jazyk navržený a vyvíjený firmou Microsoft, zavádí striktní datovou kontrolu, takže již není možné uložit například do řetězce číslo, pokud má proměnná definovaný datový typ. TypeScript je tzv. superset nad JavaScriptem. To znamená, že vše, co je možné zapsat v JavaScriptu, lze také zapsat i v TypeScriptu.

TypeScript přidává syntaktický cukr kolem prototypové dědičnosti. Díky tomu existují v TypeScriptu *třídy*, *rozhraní* a *anotce* tak, jak jsou známy z ostatních programovacích jazyků.

7.2 NestJS framework

Tvorba webových serverů v NodeJS je možná, ovšem náročná. Pro tvorbu jednoduchého serveru, který poskytuje staticky obsah, plně dostačuje. V případě rozsáhlejší aplikace je lepší sáhnout po knihovně či celém frameworku. Další potenciální problém se může skrývat pro méně zkušené vývojáře v určení struktury projektu. Při špatném návrhu se bude aplikace špatně udržovat i testovat. První knihovna, která je dodnes používaná se jmenuje Express.js. Hlavní výhoda této knihovny je v usnadnění definice rout, neboli adres dostupných na webu. Nad `Express.js` knihovnou bylo vybudováno mnoho frameworků, z nichž jeden je zde použit: NestJS.

NestJS framework je výsledek snahy vývojářů vytvořit kvalitní ekosystém pro jednoduchou tvorbu komplexních aplikací. Velkým přínosem frameworku je integrovaná podpora jazyka TypeScript.

NestJS má kromě integrace TypeScriptu další výhody. Významný díl tvoří implementace *dependency-injection*, neboli automatická správa závislostí. Typická webová aplikace se skládá z kontrolerů sloužících jako vstupní bod, dále služeb, které obsahují business logiku a nakonec databáze, která udržuje data. Propojení jednotlivých komponent mezi sebou může být často dost problematické. Dependency-injection slouží právě k jednoduchému propojení těchto komponent mezi sebou. Postará se o správné vytvoření instancí a jejich správu. V neposlední řadě je dependency-injection nedocenitelné při tvorbě testů jednotlivých komponent.

Značně ulehčená je tvorba endpointů. Každý endpoint je reprezentován třídou s kontrolerem. Třída je označena anotací `Controller('endpoint-base-path')`. Tím se řekne frameworku, že se jedná o kontroler a že má v třídě vyhledat endpointy.

7.2.1 Komponenty NestJS

NestJS framework je rozdělen do samostatných komponent. Díky tomuto rozdělení si každý projekt stáhne pouze takové komponenty, které potřebuje ke svému běhu. Základní komponenta, v které je napsané jádro frameworku, se nachází v balíku `@nestjs/core` a `@nestjs/common`.

@nestjs/cli

Samostatná komponenta určená pro vývojáře, která dává k dispozici pomocí příkazové řádky možnosti tvorby celého projektu, modulů aplikace, kontrolerů, služeb a dalších komponent. Také zajišťuje spuštění aplikace.

@nestjs/typeorm

Nedílnou součástí velkých aplikací je databáze. Komponenta `@nestjs/typeorm` obsahuje nástroje pro komunikaci s databází, tvorbu databáze a dotazy do databáze. Podporováno je velké množství databázových enginů. V případě nepodporované databáze je relativně snadné takovou podporu doprogramovat.

Veškeré dotazy na databázi lze psát dvěma způsoby: klasicky pomocí SQL jazyka, nebo pomocí API, které úplně odstraní potřebu SQL dotaz psát. V případě, že dotaz je velmi složitý, se vyplatí použít SQL jazyk, protože je vyšší šance, že programátor napíše dotaz efektivněji. Ve všech ostatních případech je lepší použít API. V kódu 7.1 je ukázka zápisu dotazu pro vyhledání experimentu podle ID za použití SQL jazyka (nahoře) tak API (dole).

Zdrojový kód 7.1: Rozdíl mezi použitím SQL jazyka (nahoře) a API typeorm (dole)

```
1 private repository: Repository<ExperimentEntity>;
2
3 // SQL dotaz
4 const experimentEntity=await this.repository.query(
5   'SELECT * FROM experiment_entity WHERE id = ?',
6   [id]);
7
8 // Pouziti API
9 const experimentEntity=await this.repository.findOne
    (id);
```

@nestjs/platform-express

Jak již bylo řečeno, NestJS framework je postavený na základech knihovny `Express.js`. Integrace knihovny je implementovaná právě v této komponentě. Alternativní možnost ke knihovně `Express.js` je fastify.io.

@nestjs/websockets

V této komponentě jsou implementovány websockety. Ty se využívají zejména v dynamických aplikacích, kdy je potřeba udržovat otevřené spojení mezi klientem a serverem. Výhodou tohoto spojení je možnost odesílat zprávy ze serveru na klienta, aniž by klient o data žádal.

7.3 Databáze

Hlavní funkcí serveru je perzistentní úložiště pro experimenty. Jako úložiště je použita SQLite databáze řadící se mezi relační databáze. To znamená, že základní jednotou databáze je tabulka. V modelu server je celkem 12 tabulek. Přehled všech tabulek a vazeb mezi nimi je shrnut v ERA (Entity-relationship) modelu v příloze C. Základ tvoří tabulka `experiment_entity`, která obsahuje záznamy všech dostupných experimentů. Informace dostupné v této tabulce jsou společné pro všechny experimenty. Jedná se zejména o unikátní identifkátor experimentu (id), název a popis experimentu a další. Na tuto tabulku navazují přes cizí klíče tabulky pro jednotlivé typy experimentů, tabulka pro výsledky experimentů a tabulka pro sekvence.

V databázi je zaregistrováno celkem 11 triggerů, jejichž přehled je vidět v tabulce 7.1. Triggery zaregistrované před vytvořením nového záznamu a po smazání záznamu (ERP, F-VEP, T-VEP) slouží primárně k automatické tvorbě a mazání dat v tabulkách závislých na rodičovské tabulce s konkrétním experimentem. Například pro experiment T-VEP existuje ještě tabulka se záznamy o jednotlivých výstupech. Při založení nového experimentu se aktivuje trigger a vygeneruje potřebné údaje do tabulky o výstupech. Na stejném principu pracuje trigger pro smazání experimentu. Při smazání experimentu se odstraní záznamy o výstupech daného experimentu. Trigger pro aktualizaci slouží k aktualizaci hodnoty `outputCount` v rodičovské tabulce všech experimentů.

Tabulka 7.1: Přehled zaregistrovaných triggerů

Typ experimentu		Trigger	
	create	update	delete
ERP	ano	ano	ano
C-VEP	ne	ano	ne
F-VEP	ano	ano	ano
T-VEP	ano	ano	ano
REA	ne	ano	ne

NestJS framework obsahuje samostatný modul zaměřený na databáze. Modul se stará o mapování jednotlivých entit na sloupečky v tabulkách. Každá tabulka, která se má v databázi vytvořit je reprezentována třídou. Třída obsahuje anotaci `@Entity()`, která říká frameworku, že se jedná o třídu obsahující mapování na tabulku v databázi. Jednotlivé atributy třídy, které se mají mapovat na tabulku jsou označeny anotací `@Column()`. Tato ano-

tace přijímá jako atribut (mimo jiné) datový typ, pod kterým bude fyzicky uložena hodnota v databázi. Mezi další dostupné anotace patří například `@PrimaryColumn()`, která označí atribut jako primární klíč. Vazby mezi tabulkami lze vytvořit za pomoci tří pomocných anotací: `@OneToMany()`, `@ManyToOne()a @ManyToMany()`. Pomocí těchto vazeb se lépe udržuje konzistence celé databáze. V ukázce kódu 7.2 je příklad definice entity pro rodičovskou tabulku experimentů.

Takto vytvořené entity slouží pouze k mapování atributů v rámci ORM, jinak nejsou používané nikde jinde, kromě místa, kde dochází ke konverzi na kompletní datovou strukturu experimentu. Toto dvojí mapování bylo potřeba vytvořit, protože reálná datová struktura obsahuje více atributů, než se fyzicky používá v databázi. Další důvod je, že struktura pro experimenty a další sdílené struktury je uložena v samostatném projektu.

Zdrojový kód 7.2: Příklad definice entity pro rodičovskou tabulku experimentů

```
1  @Entity() export class ExperimentEntity {
2    @PrimaryGeneratedColumn() id: number;
3    @Column({ length: 255, type: 'text', nullable:
       false, unique: true }) name: string;
4    @Column({ length: 255, type: 'text', nullable:
       true }) description: string;
5    @Column({ length: 255, type: 'text' }) type:
       string;
6    @Column({ type: 'integer', default: 1 })
       usedOutputs: number;
7    @Column({ type: 'integer' }) created: number;
8    @Column({ type: 'integer', default: 1})
       outputCount: number;
9    @Column({ type: 'text', nullable: true}) tags:
       string;
10 }
```

7.3.1 Migrace databáze

Technika migrace databáze je důležitá zejména v produkčním prostředí. Aplikace prochází během svého životního cyklu změnami. Mezi změny se dá počítat také úprava na databázové vrstvě. Při změně struktury databáze je velice důležité neztratit již existující data.

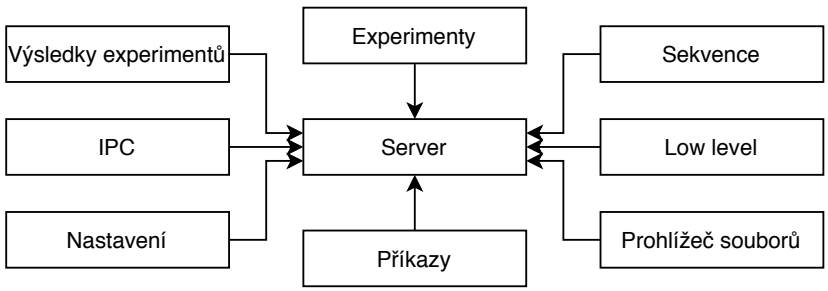
TypeORM modul obsahuje podpůrné funkce pro tvorbu a spuštění migrací při startu aplikace. Migrace jsou uloženy v třídě, která obsahuje dvě

funkce: `up` a `down`. Funkce `up` obsahuje nové změny, které do databáze přidají novou funkčnost. Typickým příkladem nové funkčnosti může být upravení nebo přidání sloupečku v tabulce. Funkce `down` slouží k odstranění změn, které se provedly ve funkci `up`. To může být užitečné v případě, že nově zavedené změny nějakým způsobem poškodili aplikaci.

7.4 Struktura serveru

Server je rozdělen do samostatných částí, kde každá část je reprezentována kontrolerem. Kontrolery definují přístupové body do dané části aplikace. S několika částmi aplikace lze vytvořit stálí spojení pomocí websocket technologie. Toto spojení je vytvořeno zejména na místech, kde je potřeba využít obousměrnou komunikaci. Standardní http spojení je pouze jednosměrné, pracuje na principu požadavek-odpověď. Při vytvoření websocket spojení lze komunikovat oběma směry po celou dobu spojení. Takovéto spojení je využito například v přehrávači experimentů, ve kterém se vizualizuje právě přehrávaný experiment. Na obrázku 7.1 je přehled modulů serveru.

Obrázek 7.1: Přehled modulů serveru



Mezi nejdůležitější moduly patří:

- Modul Experimenty - obsahuje logiku pro správu experimentů
- Modul Sekvence - obsahuje logiku pro správu sekvencí pro ERP experiment
- Modul Výsledky experimentů - spravuje výsledky experimentů
- Modul Low level - zajišťuje spojení se stimulátorem (komunikační protokol se zde nenachází)
- Modul IPC - zajišťuje spojení s externím programem pro zobrazení obrázků a přehrávání zvuků

7.4.1 Experimenty

V modulu je zajištěn nejenom přístup k experimentům, jejich tvorbě a úpravě, je zde také řešena validace, za pomoci JSON schémat. Více o validaci je v sekci 7.5.

7.4.2 Prohlížeč souborů

Na serveru bylo potřeba naimplementovat jednoduché rozhraní k přístupu na souborový systém serveru za účelem správy obrázků a zvuků, které se budou používat během experimentu. Pomocí rozhraní je možné vykonávat základní operace se soubory a složkami: *obsah složky*, *vytvořit novou složku*, *nahrát soubor* a *smazat soubor nebo složku*.

7.4.3 Low level

Tento modul definuje rozhraní, přes které se komunikuje přímo se stimulátorem. V modulu jsou k dispozici funkce pro *získání seznamu dostupných sériových portů*, *otevření* a *zavření* komunikačního portu. Dále obsahuje funkci pro zápis dat na sériovou linku. Veškerá komunikace tedy probíhá pomocí tohoto modulu. Komunikační protokol samotný je definovaný na jiném místě. Díky tomu není problém vyměnit způsob přenosu dat mezi serverem a stimulátorem.

7.5 Validace požadavků

K zajištění absolutní bezpečnosti z pohledu dat jsou veškeré konfigurace experimentů, sekvencí a výsledků experimentů, které přichází na server, validovány. Při validaci se využívá faktu, že veškeré objekty lze v JavaScriptu snadno převést na řetězce s formátem JSON. Pro JSON dokument lze vytvořit tzv. schéma, sloužící k popisu a validaci struktury.

7.5.1 JSON schema

Schéma je popsáno opět pomocí JSON dokumentu. Není tedy vždy snadné určit, co je a co není schéma. Tento problém je vyřešen přidáním nepovinného klíče `$schema` na začátek dokumentu.

Ve schématu lze pro každý atribut zvolit jeden ze šesti datových typů: `string, number, object, array, boolean a null`. Žádné jiné datové typy podporovány nejsou.

Základní schéma se skládá z následujících atributů:

- `$schema` slouží k rozlišení od JSON dat,
- `description` popis schématu,
- `type` - datový typ popisovaného atributu
- `properties` - tabulka všech dostupných atributů
- `required` - pole povinných atributů z tabulky `properties`
- `dependencies` závislosti mezi jednotlivými atributy

Ve zdrojovém kódu 7.3 je příklad jednoduchého JSON schematu.

Zdrojový kód 7.3: Příklad JSON schema

```
1  {
2    "$id": "https://example.com/address.schema.json",
3    "$schema": "http://json-schema.org/draft-07/schema
       #",
4    "description": "An address similar to http://
       microformats.org/wiki/h-card",
5    "type": "object",
6    "properties": {
7      "locality": {"type": "string" },
8      "region": {"type": "string" }
9    },
10   "required": [ "locality", "region", "country-name"
       ],
11   "dependencies": {
12     "post-office-box": [ "street-address" ],
13     "extended-address": [ "street-address" ]
14   }
15 }
```

7.6 Obrázky a zvuky

Paralelně se serverem může běžet aplikace, která se bude starat o zobrazení obrázků a přehrávání zvuků. HW stimulátor nemá grafický čip, pomocí kterého by se snadno obrázky zobrazovaly. Komunikace s touto aplikací probíhá na dvou úrovních.

Na softwarové úrovni komunikuje server s aplikací prostřednictvím named pipe (pojmenované roury). V prostředí Linuxu se za pojmenovanou rourou skrývá obyčejný soubor, do kterého jedna aplikace zapisuje a druhá z něj čte. Komunikace je pouze jednosměrná. To znamená, že server může pouze vydávat příkazy a aplikace se musí přizpůsobit. O založení pojmenované roury se stará server.

Na této úrovni informuje server aplikaci, v jakém stavu se aktuálně stimulátor nachází. Před spuštěním experimentu se například odešle seznam všech použitých obrázků a zvuků. Aplikace musí tento seznam zpracovat, a když dostane signál, zobrazí obrázek, nebo přehraje zvuk.

Další úroveň, na které musí umět aplikace komunikovat, je HW úroveň. Při komunikaci mezi stimulátorem a serverem vzniká nemalé zpoždění. Kdyby se zprávy pro aplikaci měly posílat přes server, zpoždění by narostlo ještě více. Za tímto účelem musí být aplikace schopna vytvořit vlastní sériové spojení, přes které bude dostávat příkazy, který obrázek má zobrazit, případně přehrát zvuk. Zpoždění je v tomto případě minimální, ovšem stále nezanedbatelné.

Aplikace pro zobrazení obrázků a přehrávání zvuků během experimentu není předmětem této diplomové práce. Existuje pouze základní implementace, na které je názorně demonstrován komunikační protokol. Implementovaný příklad nezobrazuje obrázky, ani nepřehrává žádné zvuky.

8 Klientská aplikace

Poslední aplikace, která byla vyvinuta v rámci této práce, je klient, pomocí kterého se vše ovládá. Cílem bylo vytvořit přehlednou intuitivní aplikaci snadnou na ovládání. Ve své bakalářské práci [7] jsem tvořil aplikaci pro vzdálené ovládání původního stimulátoru. První myšlenka logicky vedla k znovupoužití již existující aplikace. Doba ale pokročila a aplikaci by bylo potřeba značně přepsat, aby byla propojitelná s vytvořeným serverem. Mezi další nevýhody původní aplikace patří svázání pouze na mobilní operační systém Android. Proč omezovat uživatele pouze na jednu platformu, když už existují možnosti, jak tvořit aplikace pro celou řadu platforem v jednom projektu?

Jedna z možností je vydat se cestou implementace webové aplikace. Ke svému běhu potřebuje pouze webový prohlížeč, který je v současné době dostupný na všech zařízeních.

K tvorbě webové aplikace je potřeba použít programovací jazyk JavaScript, případně jazyk, který se do JavaScriptu transpiluje, například TypeScript. Angular, framework od společnosti Google, používající programovací jazyk TypeScript obsahuje veškerou funkcionalitu potřebnou k tvorbě webových aplikací.

8.1 Angular

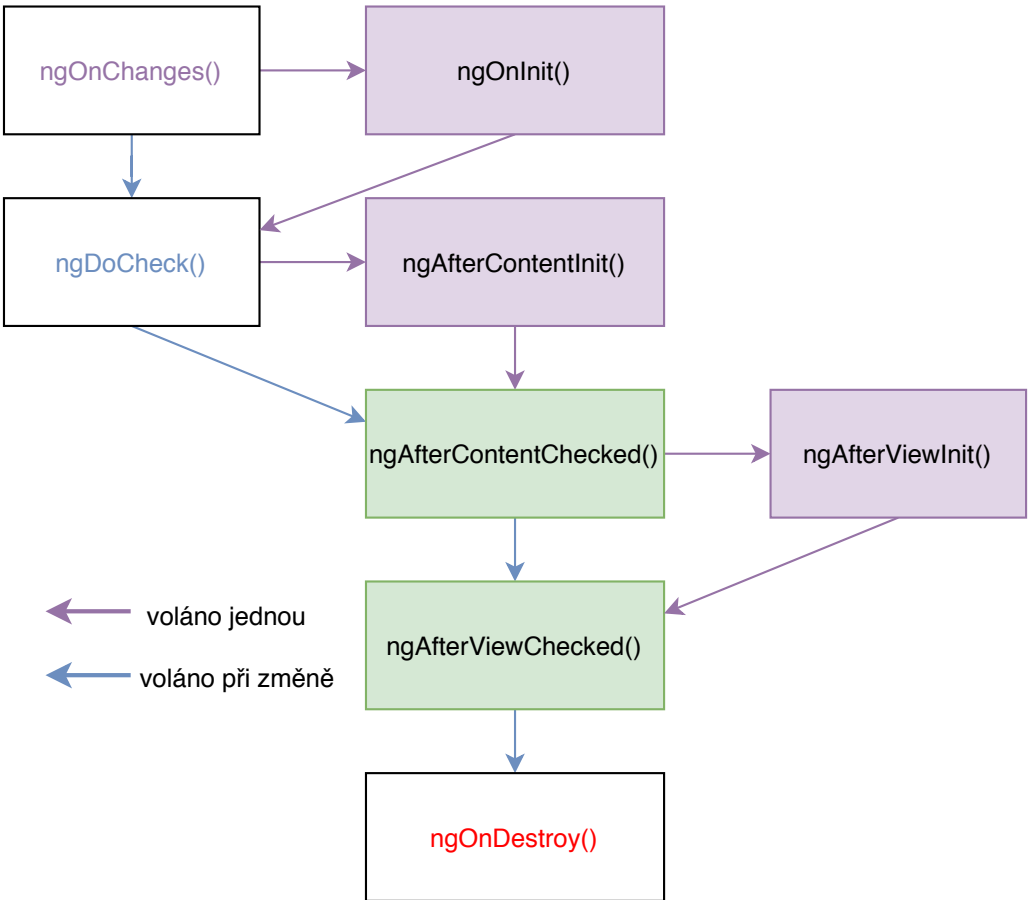
Angular, framework od společnosti Google byl vytvořen v roce 2016 jako přepsání původní AngularJS knihovny. Framework nabízí širokou škálu připravených nástrojů k ulehčení tvorby aplikace. Mezi přední výhody patří: vlastní šablonovací systém, pomocí kterého se tvoří rozložení prvků v aplikaci, striktní oddělení logiky aplikace od grafické prezentace, integrovaná správa závislostí instancí a v neposlední řadě obsahuje výborné nástroje pro testování výsledné aplikace.

Angular je založený na MVC (**M**odel **V**iew **C**ontroller) architektuře. Model, typicky reprezentován třídou, obsahuje nosná data. View se stará o vizualizaci dat v modelu. Kontroler propojuje model a view dohromady. O zdroj dat se obvykle starají služby (**S**ervices). Služby jsou zodpovědné za vlastní business logiku a komunikují s webovým serverem.

8.1.1 Životní cyklus

Angular projekt se skládá z komponent. Každá komponenta typicky obsahuje jeden kontroler, jedno view a data, která dokáže vizualizovat. Komponenty prochází v rámci aplikace definovaným životním cyklem. Na jednotlivé fáze lze reagovat pomocí implementace funkcí představující jednotlivé fáze. Na obrázku 8.1 jsou vizualizovány jednotlivé fáze životního cyklu komponent.

Obrázek 8.1: Životní cyklus komponent



Na každou fázi lze reagovat stejnojmennou funkcí. Nejčastěji se využívá funkce `ngOnInit()`, která se spustí pouze jednou a slouží k inicializaci komponenty. V této funkci je doporučeno například inicializovat formuláře na stránce, nebo se přihlásit k odběru událostí na pozorovatelných objektech.

8.1.2 Zpracování formulářů

Mezi další velké přednosti Angular frameworku patří zpracování formulářů. Angular nabízí dva přístupy, jak s formuláři pracovat:

- reactive forms - robustní, škálovatelný a lépe testovatelný přístup; používaný zejména v případě, kdy jsou formuláře klíčovou částí aplikace,
- template-driven forms - používané v jednoduchých aplikací, například k tvorbě kontaktního formuláře.

Velký rozdíl mezi jednotlivými přístupy je v použitém datovém modelu. Zatímco u reaktivního přístupu se musí přesně sepsat výsledná struktura datového modelu, v template-driven přístupu toto není potřeba, protože formulář a jednotlivé atributy modelu se definují až v HTML šabloně.

Ke zpracování formulářů byl vybrán reaktivní přístup.

Reactive forms

Pro reaktivní formuláře je nejdříve potřeba nadefinovat datový model, který bude formulář generovat. Datový model je nejčastěji rozhraní. Dále se vytvoří instance formuláře s formulářovými prvky, které přesně odpovídají datovému modelu. Tato instance se typicky definuje v kontroleru, ve kterém se formulář vyskytuje. V kódu 8.1 je příklad vytvoření jednoduchého formuláře pro experiment.

Zdrojový kód 8.1: Tvorba struktury formuláře

```
this.form = new FormGroup({
    id: new FormControl(null),
    name: new FormControl(null,
        {
            validators: [Validators.required],
            updateOn: 'blur'
        }),
    description: new FormControl(),
    usedOutputs: new FormGroup(
        {
            led: new FormControl(null),
            audio: new FormControl(null),
            audioFile: new FormControl(null),
            image: new FormControl(null),
            imageFile: new FormControl(null)
        }),
    tags: new FormControl([])
});
```

Základ vždy tvoří třídy `FormGroup`, která přijímá v konstruktoru jako parametr objekt s jednotlivými formulářovými prvky. Může se jednat o obyčejný formulářový prvek `FormControl`, prvek s polem `FormArray`, nebo vnořený formulář `FormGroup`. Počet zanořených formulářů není omezen.

8.2 Popis aplikace

Výsledná aplikace je v základu rozdělena do stejných modulů, jako tomu je na serveru. Díky tomuto rozdělení se lze lépe orientovat v kódu a v případě přidávání nových funkcí je snadnější implementace jak na straně klienta, tak i serveru.

Základní rozložení aplikace se skládá z horní lišty a postranního menu. Zbytek vyplňuje samotný obsah podle právě zobrazované stránky.

8.2.1 Experimenty

Hlavní část aplikace se věnuje experimentům a jejich správě. Na hlavní stránce se zobrazuje přehled všech dostupných experimentů. Na obrázku 8.2 je vidět výchozí stránka aplikace. Experimenty jsou seřazeny podle abecedy. Na seznam experimentů lze aplikovat filtry s nastavením jiného řazení.

Dále lze experimenty seskupovat do skupin podle typu experimentů. Toto seskupení se ukázalo jako velmi užitečné z hlediska přehlednosti. Vyhledávání mezi experimenty je samozřejmostí. Jednotlivé typy experimentů jsou od sebe barevně odlišeny.

Každý záznam o experimentu obsahuje název a popis experimentu, datum vytvoření, druhy a počet stimulů. V pravé části každého záznamu jsou tlačítka pro spuštění/editaci/smazání experimentu. Smazání experimentu je chráněno potvrzovacím dialogem k zabránění nechtěného smazání.

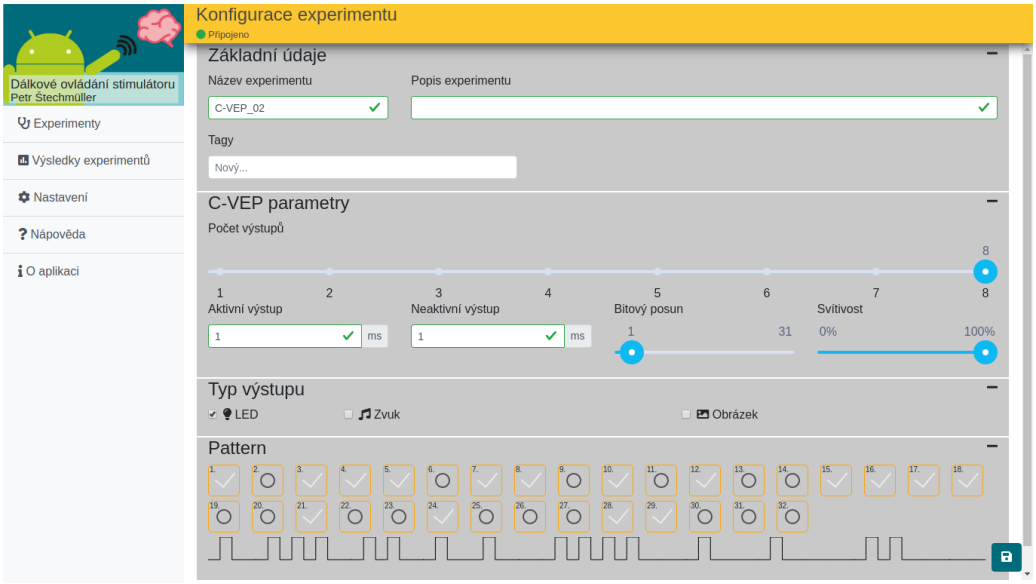
Obrázek 8.2: Hlavní stránka aplikace - přehled všech experimentů

V pravém dolním rohu je tlačítko pro založení nového experimentu. Po kliknutí se zobrazí nabídka typu experimentu, který se má založit. Po výběru typu experimentu se zobrazí editor příslušného experimentu. Příklad editoru C-VEP experimentu je na obrázku 8.3.

Zvláštní péče byla zaměřena na validaci všech vstupů. Díky **Reactive forms** bylo možné implementovat velmi kvalitní validaci, takže se nemůže stát, že by byla na server odeslána nevalidní konfigurace experimentu. V rámci validace byly implementovány nejrůznější informační hlášky ke každému vstupu. Uživatel je tedy velmi dobře informován v situaci, kdy vstup není validní.

Nejčastějším problémem, který se v konfiguraci může vyskytnout, je nulová hodnota. Například u nastavení výstupů lze ovlivnit intenzitu svítivosti jednotlivých LED. Když ale uživatel nastaví hodnotu na 0, tak LED sice bude blikat, ale vizuálně uživatel nic neuvidí. Chybám tohoto typu je snaha

Obrázek 8.3: Editor C-VEP experimentu



v aplikaci předejít právě pomocí informačních hlášek. V případě nulového typu je pod vstupem hláška, že hodnota je nulová. Nejedná se o validační chybu, takže formulář lze uložit a odeslat, ale následky můžou být i tak někdy fatální.

Využita je také asynchronní validace u názvu experimentu, kdy se kontroluje, že název je unikátní napříč všemi existujícími experimenty. Při opuštění vstupního pole pro název se automaticky vyšle dotaz na server. V závislosti na odpovědi se vstupní pole zabarví červeně nebo ne.

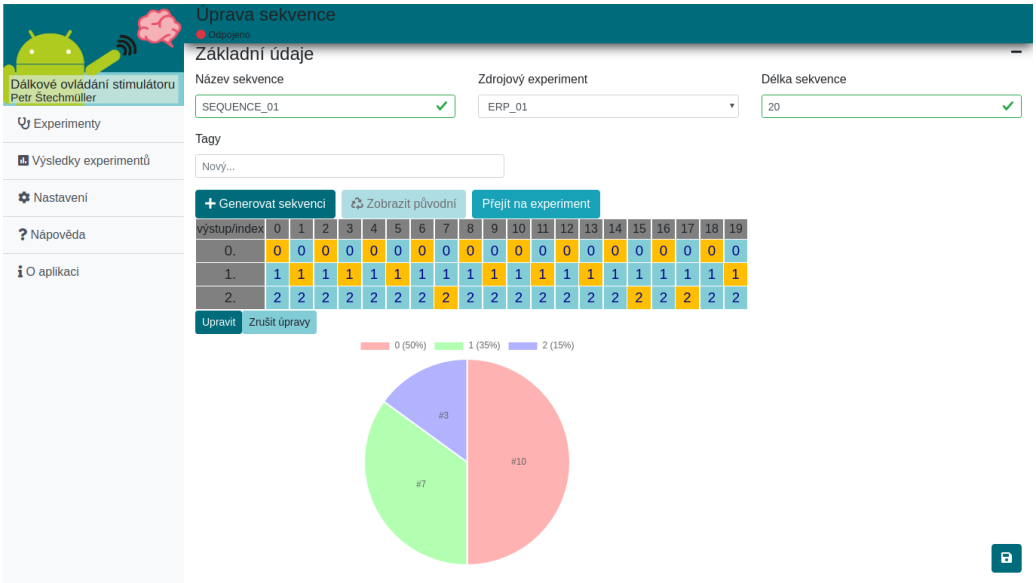
U každého experimentu v nastavení stimulu je možné určit typ výstupu. Na výběr je vždy ze tří podporovaných možností: *LED*, *zvuk*, nebo *obrázek*. V případě LED se při aktivním stimulu rozsvítí přiřazená LED na stimulátoru. Pro podporu *zvuku* a *obrázku* musí být spuštěna aplikace na přehrávání zvuků a obrázků. Při zaškrtnutí zvuku nebo obrázku je třeba vybrat zvukový soubor nebo obrázek. K tomu slouží integrovaný prohlížeč souborů.

Prohlížeč souborů zobrazuje soubory, které jsou uloženy ve speciální složce na serveru.

8.2.2 Sekvence

Na přehled sekvencí se lze dostat pouze z ERP experimentů. Stránka s přehledem sekvencí je po obsahové stránce totožná s přehledem experimentů. Na obrázku 8.4 je vidět náhled editoru sekvencí. V editoru je možné upravovat vlastnosti sekvence. Po vygenerování sekvence je možné upravovat jednotlivé stimuly v případě, že nevyhovují očekávání. Dále se v editoru nachází koláčový graf, který vizualizuje procentuální zastoupení jednotlivých stimulů v sekvenci. Při úpravě sekvence se graf automaticky přepočítá, takže je vždy vidět aktuální procentuální zastoupení stimulů. Výsledek v grafu by měl odpovídat distribuční hodnotě stimulu nastavené ve zvoleném ERP experimentu v případě, že nemá žádné závislosti. Pokud má stimul závislosti, výsledná hodnota bude menší než nastavená.

Obrázek 8.4: Editor sekvencí



8.2.3 Přehrávač experimentů

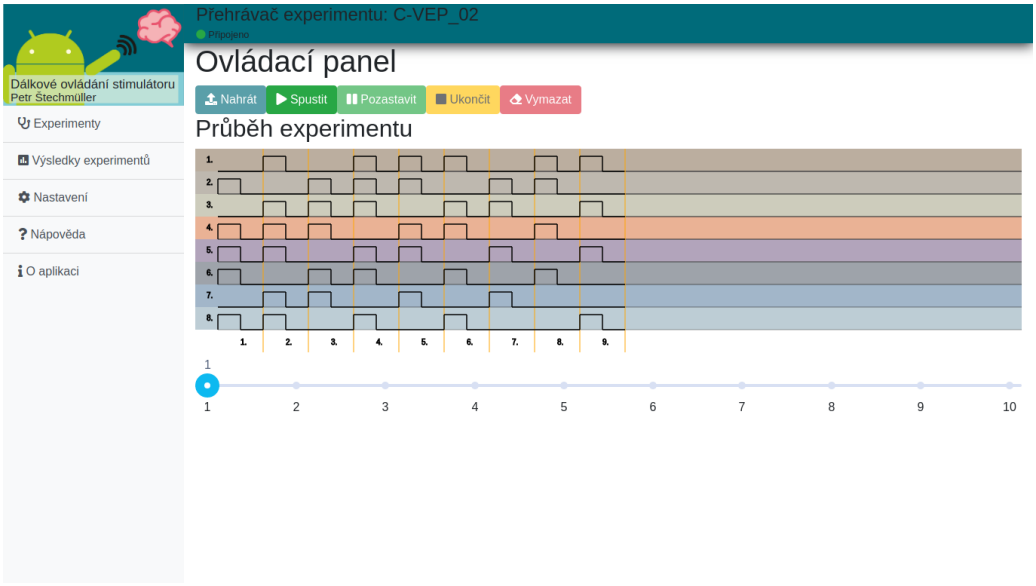
Přehrávač experimentu zobrazený na obrázku 8.5 je dostupný z hlavní stránky s experimenty. Přehrávač obsahuje v horní části tlačítka pro ovládání experimentu. Pod tlačítky se nachází graf, na kterém se vizualizuje průběh aktuálně spuštěného experimentu. V grafu je vidět, kdy se aktivoval, který stimul. Dále je zobrazeno zaznamenání vstupu od uživatele.

Aby bylo možné experiment spustit, je potřeba experiment nejdříve do stimulátoru nahrát. K tomu slouží první tlačítko *nahrát*. Po stisku tlačítka

se odešle příkaz na server. Server příkaz zpracuje a odešle na stimulátor dva příkazy. V rámci zpracování příkazu se na serveru vygeneruje struktura pro výsledek experimentu, do které se bude zaznamenávat průběh experimentu. První příkaz, který se odešle do stimulátoru, je s konfigurací experimentu, která se uloží do paměti. Druhý příkaz se do stimulátoru odešle pro inicializaci experimentu. Toto rozdělení bylo nutné zejména v ranných časech vývoje, kdy se velmi často stávalo, že konfigurace se zapsala mimo přidělenou paměť, a proto se inicializace neprovedla.

Zbylá tlačítka odpovídají vlastnímu popisu. Tlačítko pro zastavení experimentu ukončí experiment a přesune uživatele na stránku s výsledkem experimentu.

Obrázek 8.5: Přehrávač experimentu



8.2.4 Výsledky experimentů

Stránka s výsledky experimentů je dostupná z bočního menu. Na stránce je opět k dispozici přehled všech provedených experimentů. Po vybrání zvoleného experimentu se zobrazí informace o experimentu a graf s průběhem.

Data o průběhu experimentu je možné z aplikace stáhnout ve formátu JSON a použít k dalšímu zpracování. Každá událost obsahuje strukturu, která je znázorněna v kódu: 8.2

Zdrojový kód 8.2: Struktura dat s průběhem experimentu

```
1 class EventIOChange {
2     ioType: 'input' | 'output';
3     state: 'on' | 'off';
4     index: number;
5     timestamp: number;
6 }
```

Každá akce provedená na stimulátoru (aktivace/deaktivace stimulu, reakce na stimul) je reprezentována jedním záznamem se strukturou 8.2. `IoType` atribut určuje, zdali se jedná o akci provedenou na stimulu nebo na tlačítku. `State` atribut určuje, zdali se stimul (de)aktivoval / tlačítko zmáčklo(uvolnilo). Uvolnění tlačítka už není podstatné z pohledu měření. Poslední atribut `timestamp` obsahuje časovou značku vytvoření události. Nejedná se přímo o čas, ale o číselnou hodnotu generovanou časovačem na stimulátoru.

8.2.5 Nastavení

Celá stránka s nastavením 8.6 je rozdělena na tři hlavní sekce: *stav služeb*, *konfigurace parametrů* a *konzole*.

V sekci *stav služeb* je vizualizované, zdali je připojený server, stimulátor a program pro zobrazování obrázků/zvuků. Dále je zde možné zvolit, na který port se má aplikace připojit, aby mohla komunikovat se stimulátorem. Tato volba není automatická, takže je na uživateli, aby správně zvolil sériový port. Lze zaškrtnout, aby si server zapamatoval poslední použitý sériový port a při příštím restartu se na něj automaticky připojil. V sekci je také k dispozici tlačítko pro aktualizaci firmware stimulátoru.

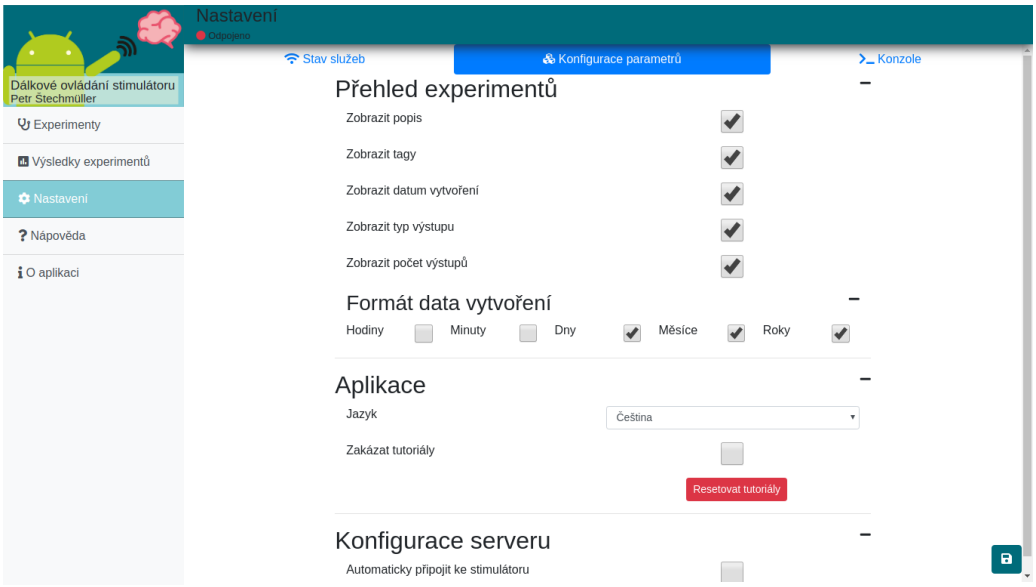
Sekce *konfigurace parametrů* se zabývá nastavením nejrůznějších parametrů celé aplikace. Sekce je rozdělena podle částí aplikace, kterých se týká. V obecném nastavení lze nastavit jazyk aplikace - na výběr je zatím mezi češtinou a angličtinou. Dále je možné zakázat zobrazení tutoriálů na stránkách, které ještě uživatel nenavštívil. Část s přehledem experimentu obsahuje zejména nastavení vizuálních prvků, které se zobrazí u každého řádku

s experimentem na hlavní stránce aplikace.

K některým částem nastavení je možné přistoupit přímo ze stránky, které se nastavení týká. Na stránce s výpisem všech experimentů se nachází tlačítko pro zobrazení nastavení viditelnosti prvků v položkách seznamu experimentů. V přehrávači experimentů se lze díky rychlému nastavení snadno připojit ke stimulátoru.

V nastavení se také nachází záložka s *konzolí*. Pomocí konzole lze ovládat experiment. Konzole lze také využít k výpisu paměti stimulátoru. Lze vypsat paměť, kde je uložena samotná konfigurace a dále paměť s různými čítači použitými v některých experimentech. U ERP experimentu lze pomocí výpisu paměti získat informaci, která část sekvence je aktuálně nahraná ve stimulátoru.

Obrázek 8.6: Nastavení aplikace



9 Testování

Testování aplikace je nedílnou součástí jejího vývoje. V této kapitole budou popsány druhy testů, které jsou použity.

9.1 Způsoby testování

Aplikaci lze testovat manuálně, nebo automatizovaně. Manuálního testování se typicky zúčastňují dobrovolníci v rámci betatestování aplikace. Automatické testy slouží zejména k nalezení chyb či nesrovnalostí vůči zadání. Mezi nejznámější nástroje pro automatické testování patří Selenium.

9.2 Přístupy testování

White Box testování je přístup, u kterého je znám zdrojový kód testované aplikace. Testy přímo využívají zdrojový kód a ověřují funkcionalitu na nejnižší vrstvě aplikace.

Black Box testování je pravým opakem **white box testování**. V tomto přístupu se k aplikaci přistupuje jako k hotovému celku. V rámci těchto testů se kontroluje, zdali systém správně reaguje na vstupy a generuje odpovídající výstupy. Také se ověřuje správná funkctionalita uživatelského rozhraní. Správné zobrazení chybových hlášek a další.

Grey Box testování představuje kombinaci obou výše zmíněných přístupů.

9.3 Úrovně testů

9.3.1 Jednotkové testy

Jednotkové testy spadají do kategorie **white box testů**. Ověřují funkcionalitu samostatných částí nezávisle na zbytku aplikace. Jednotkové testy se snaží pokrýt co největší množství cest, kterými je možné testovanou část projít. Při testování funkce, je třeba ověřit veškeré parametry, které vstupují do funkce. Všechny podmínky, které funkce obsahuje. S velkým množstvím podmínek roste komplexnost testu. Je tedy důležité, aby kód ve funkci byl co nejkratší a dělal pouze jednu věc. Každý test by měl kontrolovat pouze jednu věc.

9.3.2 Integrační testy

Integrační testy volně navazují na **jednotkové testy**. V integračních testech se ověřuje, že jednotlivé moduly v aplikaci dokážou mezi sebou správně komunikovat. Stále se jedná o typ **white box testů**.

9.3.3 Systémové testy

Systémové testy neboli end-to-end testy se používají k ověření funkcí výsledné aplikace jako celku. Zde se ověřuje, že systém funguje ve všech podporovaných prostředí. Systémové testy spadají do kategorie **black box testů**. Ověřují, že zadané vstupy generují odpovídající výstupy. Testy se také používají k ověření správného chování uživatelského rozhraní.

V rámci ověřování uživatelského rozhraní se může jednat od malých detailů, jako je například zobrazení validační hlášky při chybně zadaném vstupu až po celé scénáře, kdy se testuje, že aplikace je schopná například vytvořit novou konfiguraci experimentu, která se následně zobrazí v přehledu všech experimentů na první pozici.

10 Ověření funkcionality

V rámci této diplomové práce vznikla široká škála testů zaručující funkcionalitu aplikace. Díky velkému počtu testů bylo odhaleno a odstraněno velké množství chyb, zanesených většinou nepozorností při kopírování jednou napsaného kódu.

10.1 Testování klientské aplikace

Jedna z výhod použitého frameworku Angular v klientské části aplikace je integrovaná podpora testování aplikace. V projektu byly vytvořené testy zejména zaměřené na end-to-end testování. V testech je kontrolována aplikace z pohledu uživatelského rozhraní. Velký důraz byl kladen na základní účel aplikace - tvorba a editace experimentů. Jednotkové testy pokrývají minoritní část aplikace. Vždy, když byla odhalena chyba (a bylo to možné), byl napsán test, který ověřoval odstranění nalezené chyby. Tímto způsobem bylo možné eliminovat vznik jednou nalezených chyb.

10.2 Testování na serveru

Stejně jako Angular nabízí integrované nástroje pro testování, NestJS framework nabízí podobné nástroje pro testování na serveru. Na serveru bylo naopak vytvořeno velké množství jednotkových testů s cílem pokrýt zejména validaci vstupních dat.

10.3 Testování stimulátoru

Funkcionalita stimulátoru byla ověřena pouze manuálním testováním.

10.4 Automatizace spouštění testů

Spouštět testy ručně pokaždé, když se provedou nějaké změny není z pohledu času efektivní. Proto bylo v rámci repositáře s projektem nakonfigurováno automatické spouštění testů vždy, když se změny v kódu nahrály do repositáře.

11 Možnosti rozšíření

Výsledný systém takových to rozměrů obsahuje velké množství míst, které lze rozšířit, nebo vylepšit. V této kapitole bude popsáno pouze několik důležitých návrhů na rozšíření, na které se přišlo během testování aplikace.

Vylepšení vizualizace patternů - Aktuální vizualizace patternů není dostatečně uživatelsky přívětivá. V současné verzi zabírají jednotlivé checkboxy reprezentující jednu jednotku v patternu příliš mnoho místa.
U C-VEP patternu by bylo dobré přidat vizualizaci také odvozených patternů, aby bylo vidět, jak bude vypadat průběh ostatních stimulů a ne jenom prvního.

Plugin systém - Experiment typu REA byl implementován až když byly veškeré ostatní experimenty naimplementovány a odladěny. Během této implementace bylo zjištěno, že přidání podpory nového experimentu do stávajícího systému není úplně triviální záležitost a zabere nepřiměřeně velké množství času. Plugin systém by mohl tento problém vyřešit. Na druhou stranu taková změna by vyžadovala velké úpravy v kódu.

Automatické odstranění zpoždění na stimulátoru - Doba, než se přenese informace o (de)aktivaci stimulu ze stimulátoru do aplikace pro přehrávání zvuků / zobrazení obrázků není nezanedbatelná. V rámci rozšíření by se mohla implementovat funkce, která by na základě analýzy zpoždění upravila vyslání impulzu s (de)aktivací stimulu. Momentálně se o tomto zpoždění ví, ale nijak se neodstraňuje.

Export/Import experimentů - V ovládací aplikaci na stránce s přehledem všech dostupných experimentů by bylo vhodné přidat možnost exportovat experimenty do souboru. Touto malou modifikací by aplikace získala jednoduchý způsob přenosu experimentů mezi zařízeními.

Nastavení jednotky času v experimentech - V editoru experimentů se velmi často vyskytují parametry nastavující čas. Časová jednotka byla přímo zadána do kódu a nelze změnit. Přidání možnosti nastavit, v jakých jednotkách se čas zadává by opět mohlo přinést lepší uživatelský zážitek.

12 Závěr

Cílem této práce byla reimplementace hardwarového stimulátoru pro neuroinformatické experimenty. Čtenář byl nejprve seznámen s obecnou terminologií elektroencefalografie a později i evokovaných potenciálů. Další kapitoly se věnovaly softwarovým a hardwarovým stimulátorům a jejich výhodám a nevýhodám. Na základě získaných informací byl vybrán HW, na kterém byl stimulátor naprogramován.

Aby se stimulátor snadno ovládal, byla k němu vytvořena klientská aplikace. Pomocí aplikace se nastavují parametry experimentů, řídí se jednotlivé experimenty a také prohlíží výsledky. Prohlížeč výsledků experimentů je pouze informativní a neobsahuje žádné podrobné analýzy. Mezi klientem a stimulátorem běží server, který spravuje databázi experimentů a jejich výsledků. Server slouží jako prostředník mezi klientem a stimulátorem.

V závěru práce byly diskutovány možnosti rozšíření výsledného systému, které stojí za pozornost a budou implementovány později.

Literatura

[1] Bin, G. et al. VEP-based brain-computer interfaces: time, frequency, and code modulations [Research Frontier]. *IEEE Computational Intelligence Magazine*. 2009, 4, 4, s. 22–26.

[2] Blackwood D. H. R, M. W. J. Cognitive brain potentials and their application. *The British Journal of Psychiatry*. 1961, s. 96–101.

[3] G.AbbruzzeseC.Trompetto. Motor Evoked Potential. *Reference Module in Neuroscience and Biobehavioral Psychology*. 2017.

[4] Hill, G. D. B. P. G. A. A. M. R. A. S. G. N. Recording Human Electrocorticographic (ECoG) Signals for Neuroscientific Research and Real-time Functional Cortical Mapping. *Journal of Visualized Experiments*. 2012. doi: 10.3791/3993.

[5] M. P. Paulraj, S. B. Y. A. H. B. A. K. S. – Hema, C. R. Auditory evoked potential response and hearing loss: a review. *The open biomedical engineering journal*. 2015, 9, s. 17–24. doi: 10.2174/1874120701509010017.

[6] Spüler, M. A high-speed brain-computer interface (BCI) using dry EEG electrodes. *PLoS ONE*. 02 2017, 12, s. e0172400. doi: 10.1371/journal.pone.0172400.

[7] Štechmüller, P. *Vytvoření aplikace na bázi OS Android pro ovládání stimulátoru pro neuroinformatické experimenty* [online]. [cit. 2020/03/05]. Dostupné z: `https://dspace5.zcu.cz/bitstream/11025/27713/1/Stechmuller_BPINI.pdf`.

A Tabulka procesorů Cortex-M4

Procesor/Funkce	Cortex-M0	Cortex-M1	Cortex-M3	Cortex-M4	Cortex-M7
Instrukční sada Architektura	Armv6-M Thumb, Thumb-2	Armv6-M Thumb, Thumb-2	Armv7-M Thumb, Thumb-2	Armv7-M Thumb, Thumb-2	Armv7-M Thumb, Thumb-2
DMIPS[1]/MHz	0.87 - 1.27	0.8	1.25 - 1.89	1.25 - 1.95	2.14 - 3.23
Jednotka DSP[2]	Ne	Ne	Ne	Ano	Ano
FP[3] jednotka	Ne	Ne	Ne	Ano	Ano
NVIC[4]	Ano	Ano	Ano	Ano	Ano
Max. externích interuptů	32	32	240	240	480

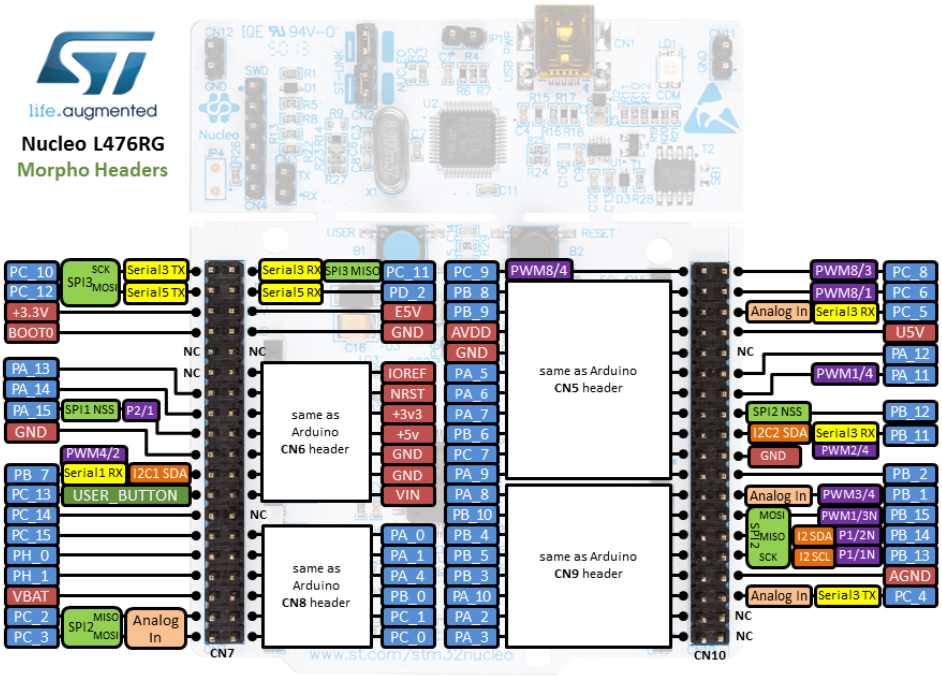
[1]DMIPS/MHz je číslo, které udává, jakého výkonu DMIPS daný mcu dosahuje pokud je taktován na 1 MHz. Čím vyšší číslo, tím lepší výkon mcu

[2]DSP = Digital Signal Processing

[3]FP = Floating point

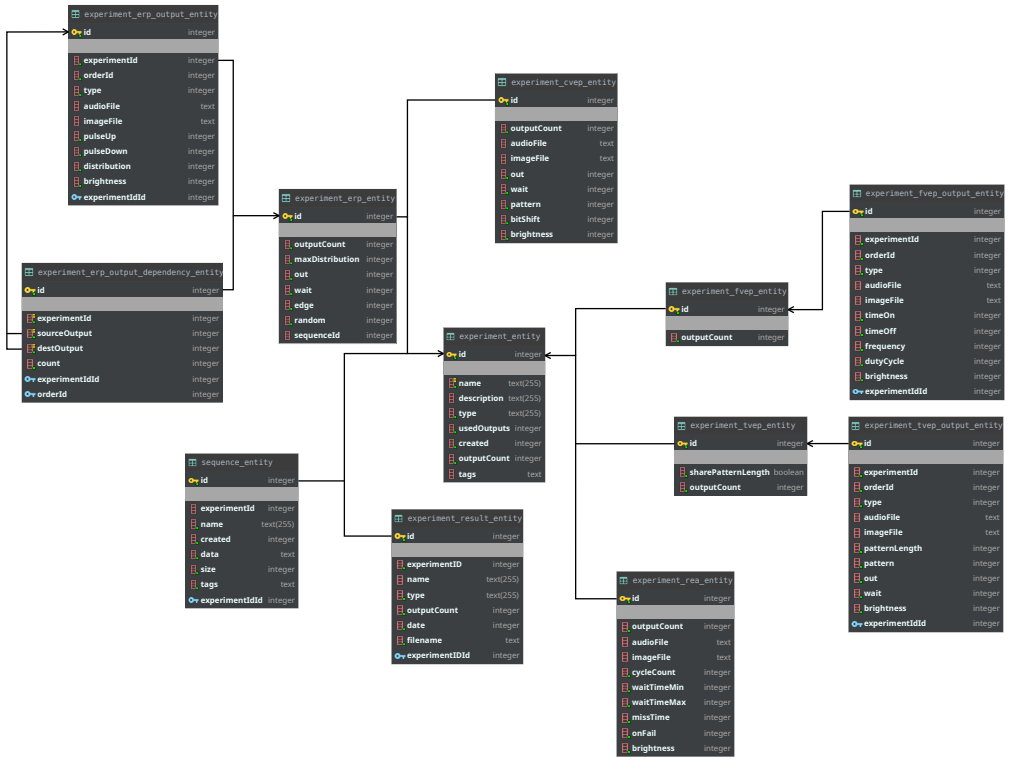
[4]NVIC = Nested Vector Interrupt Control

B Přehled pinů na desce STM32L476RG



C ERA model databáze

Obrázek C.1: ERA model databáze



D Instalace aplikace

V přiloženém DVD se nachází zdrojové kódy výsledné aplikace a text práce v TeXu. Zdrojové kódy jsou uloženy ve složce s názvem `"zdrojové kódy aplikace"`. Text práce se nachází ve složce `"text práce"`. Na DVD se dále nachází soubor `"README.txt"`, který obsahuje tento text.

Ve složce se zdrojovými kódy aplikace se nachází adresáře pro klientskou část, serverovou část a hardwarovou část. K instalaci a spuštění klienta a serveru je nutné mít na provozovaném zařízení nainstalovaný program `NodeJS` alespoň ve verzi 12.6. Současně je vyžadován balíčkovací systém NPM, který je součástí NodeJS. K vytvoření firmware pro stimulátor je potřeba mít na zařízení nainstalovaný program `PlatformIO Core`.

Vedle adresářů se zdrojovými kódy aplikace a firmwaru se nachází soubor `install.sh`, který se postará o kompletní přeložení všech tří aplikací a vytvoření firmwaru pro stimulátor.

Přeložené soubory pro server s klientskou aplikací se po instalaci budou nacházet ve složce `"/zdrojové kódy aplikace/application"`. Firmware bude umístěn také do složky `application`, kde bude k nalezení pod názvem `firmware.bin`.

Soubor `run.sh`, který je umístěn ve stejném adresáři jako `install.sh` slouží ke spuštění serveru pomocí kterého se zpřístupní uživatelské rozhraní.

Po spuštění serveru stačí přejít do webového prohlížeče a otevřít webovou stránku. Server naslouchá ve výchozím nastavení na portu `3005`. Adresu je třeba zjistit například pomocí linuxového nástroje `ifconfig`, případně ve Windows `ipconfig`.

Firmware pro stimulátor je možné do zařízení nahrát dvěma způsoby. Všechny STM zařízení se po připojení USB jeví v počítači jako úložiště. Doporučený způsob je zkopírovat vytvořený firmware přímo do zařízení. Druhou možností je nahrání firmware přímo z klientské aplikace. V nastavení serveru se nachází tlačítko pro nahrání firmware do stimulátoru. Tato možnost funguje pouze na linuxu a vyžaduje dodatečné nastavení. Je třeba *namountovat* diskovou jednotku stimulátoru do složky `"/mnt/stm"`. Příkaz pro *mount* by mohl vypadat například takto: `sudo mount /dev/disk/by-label/NUCLEO_l476rg /mnt/stm`. Před použitím příkazu *mount* je třeba se ujistit, že cílová složka `/mnt/stm` existuje a případně ji vytvořit.

E Komunikační protokol

V této příloze bude popsán komunikační protokol mezi serverem a stimulátorem. Nejdříve budou uvedeny příkazy, které se odesílají ze serveru na stimulátor. V druhé části budou uvedeny příkazy odesílané ze stimulátoru na server. Všechny příkazy jsou uloženy v souboru *src/commands/protocol/functions.protocol.ts*

E.1 Příkazy ze serveru na stimulátor

E.1.1 Příkaz STIMULATOR_STATE

Význam: Příkaz slouží k získání aktuálního stavu stimulátoru.

index	délka (byte)	hodnota	význam
0	1	0x01	hlavička příkazu
1	1	0x53	oddělovač příkazu

Tabulka E.1: Struktura příkazu pro získání stavu stimulátoru

E.1.2 Příkaz MANAGE_EXPERIMENT

Význam: Pomocí parametrů se ovládá experiment. Upravuje vnitřní stav stimulátoru.

index	délka (byte)	hodnota	význam
0	1	0x03	hlavička příkazu
1	1	?	co se má stát s experimentem
2	1	0x53	oddělovač příkazu

Tabulka E.2: Struktura příkazu pro ovládání experimentu

Na 1. index se vloží hodnota na základě činnosti, která se má stát. Hodnota může být z uzavřeného intervalu <2;6>. Význam hodnot je popsán v další tabulce.

hodnota	význam
2	inicializace experimentu
3	spuštění experimentu
4	pozastavení experimentu
5	ukončení experimentu
6	vymazání experimentu

E.1.3 Příkaz MANAGE_EXPERIMENT - Upload

Význam: Příkaz, který nahraje experiment do stimulátoru. Při serializaci experimentu se zároveň provádí serializace nastavení výstupů, pokud to experiment vyžaduje. V takovém případě se odešle jeden příkaz s nastavením experimentu následovaný příkazy s nastavením jednotlivých výstupů.

Experiment ERP

ERP experiment obsahuje samostatné nastavení výstupů. Proto budou uvedeny dvě tabulky. První tabulka obsahuje základní informace o experimentu. Druhá tabulka popisuje strukturu příkazu pro jeden výstup. Těchto příkazů se odešle takový počet, aby odpovídal nastavení počtu výstupů v experimentu. Příkazy pro sekvence ERP experimentu jsou uvedeny samostatně.

index	délka (byte)	hodnota	význam
0	1	0x03	hlavička příkazu
1	1	0x01	nahrání experimentu
2	1	?	typ experimentu
3	1	?	počet výstupů
4	8	?	doba aktivního výstupu [us]
12	8	?	doba neaktivního výstupu [us]
20	1	?	nastavení náhodnosti
21	1	?	nastavení náběžné/sestupné hrany
22	2	?	celková délka sekvence
24	1	0x53	oddělovač příkazu

Tabulka E.3: Struktura příkazu pro nahrání ERP experimentu

Experiment C-VEP

C-VEP experiment neobsahuje žádné samostatné nastavení výstupů.

index	délka (byte)	hodnota	význam
0	1	0x11	hlavička příkazu
1	1	?	index výstupu
2	1	?	typ výstupu
3	8	?	doba aktivního výstupu [us]
11	8	?	doba neaktivního výstupu [us]
19	1	?	intenzita svítivosti výstupu (pro LED)
20	1	0x53	oddělovač příkazu

Tabulka E.4: Serializace výstupu ERP experimentu

index	délka (byte)	hodnota	význam
0	1	0x03	hlavička příkazu
1	1	0x01	nahrání experimentu
2	1	?	typ experimentu
3	1	?	počet výstupů
4	1	?	typ výstupu
5	8	?	doba aktivního výstupu [us]
13	8	?	doba neaktivního výstupu [us]
21	1	?	bitový posun patternu vůči originálu
22	1	?	intenzita svítivosti výstupu (pro LED)
23	4	?	pattern 1. výstupu
27	1	0x53	oddělovač příkazu

Tabulka E.5: Struktura příkazu pro nahrání C-VEP experimentu

Experiment F-VEP

F-VEP experiment obsahuje samostatné nastavení výstupů. Proto budou uvedeny dvě tabulky. První tabulka obsahuje základní informace o experimentu. Druhá tabulka popisuje strukturu příkazu pro jeden výstup. Těchto příkazů se odešle takový počet, aby odpovídal nastavení počtu výstupů v experimentu.

Experiment T-VEP

T-VEP experiment obsahuje samostatné nastavení výstupů. Struktura příkazu pro nastavení experimentu je totožná s F-VEP experimentem. Nastavení výstupů se liší pouze přidaným parametrem "pattern". Proto bude uvedena pouze tabulka pro nastavení výstupu.

index	délka (byte)	hodnota	význam
0	1	0x03	hlavička příkazu
1	1	0x01	nahrání experimentu
2	1	?	typ experimentu
3	1	?	počet výstupů
4	1	0x53	oddělovač příkazu

Tabulka E.6: Struktura příkazu pro nahrání F-VEP experimentu

index	délka (byte)	hodnota	význam
0	1	0x11	hlavička příkazu
1	1	?	počet výstupů
2	1	?	index výstupu
3	1	?	typ výstupu
4	8	?	doba aktivního výstupu [us]
12	8	?	doba neaktivního výstupu [us]
20	1	?	intenzita svítivosti výstupu (pro LED)
21	1	0x53	oddělovač příkazu

Tabulka E.7: Serializace výstupu F-VEP experimentu

index	délka (byte)	hodnota	význam
0	1	0x11	hlavička příkazu
1	1	?	počet výstupů
2	1	?	index výstupu
3	1	?	typ výstupu
4	8	?	doba aktivního výstupu [us]
12	8	?	doba neaktivního výstupu [us]
20	1	?	intenzita svítivosti výstupu (pro LED)
21	4	?	pattern výstupu
25	1	0x53	oddělovač příkazu

Tabulka E.8: Serializace výstupu T-VEP experimentu

Experiment REA

REA experiment neobsahuje žádné samostatné nastavení výstupů.

index	délka (byte)	hodnota	význam
0	1	0x03	hlavička příkazu
1	1	0x01	nahrání experimentu
2	1	?	typ experimentu
3	1	?	počet výstupů
4	1	?	typ výstupů
5	1	?	počet cyklů
6	8	?	minimální délka periody [us]
14	8	?	maximální délka periody [us]
22	8	?	validní doba reakce [us]
30	1	0/1	co se má dít při neúspěšné reakci (čekat na timer/pokračovat)
31	1	?	intenzita svítivosti výstupu (pro LED)
32	1	0x53	oddělovač příkazu

Tabulka E.9: Struktura příkazu pro nahrání REA experimentu

E.1.4 Příkaz NEXT_SEQUENCE_PART

Význam: Odešle na stimulátor požadovanou část sekvence ERP experimentu.

index	délka (byte)	hodnota	význam
0	1	0x20	Hlavička příkazu
1	1	?	Index
2	4	?	Část sekvence
6	1	0x53	Oddělovač příkazu

Tabulka E.10: Struktura příkazu s částí požadované sekvence

E.2 Příkazy ze stimulátoru na server

E.2.1 Příkaz STIMULATOR_STATE

Význam: Příkaz obsahuje aktuální stav stimulátoru. Je odeslán vždy, když se zavolá stejnojmenný příkaz ze serveru nebo když přijde příkaz MANAGE_EXPERIMENT.

E.2.2 Příkaz OUTPUT_ACTIVATED

Význam: Příkaz se odešle vždy, když se aktivuje výstup.

index	délka (byte)	hodnota	význam
0	1	0x01	Hlavička příkazu
1	1	0x08	Délka dat
2	1	?	Stav stimulátoru
3	1	1/0	Aktivuje postprocessing na serveru
4	4	?	Časová známka
9	1	0x53	První oddělovací znak
10	1	0xFF	Druhý oddělovací znak

Tabulka E.11: Struktura příkazu s informací o stavu stimulátoru

index	délka (byte)	hodnota	význam
0	1	0x10	Hlavička příkazu
1	1	0x07	Délka dat
2	1	?	Index výstupu/tlačítka
3	4	?	Časová známka
7	1	0x53	První oddělovací znak
8	1	0xFF	Druhý oddělovací znak

Tabulka E.12: Struktura příkazu s aktivovaným výstupem

E.2.3 Příkaz OUTPUT_DEACTIVATED

Význam: Příkaz se odešle vždy, když se deaktivuje výstup.

index	délka (byte)	hodnota	význam
0	1	0x11	Hlavička příkazu
1	1	0x07	Délka dat
2	1	?	Index výstupu/tlačítka
3	4	?	Časová známka
7	1	0x53	První oddělovací znak
8	1	0xFF	Druhý oddělovací znak

Tabulka E.13: Struktura příkazu s deaktivovaným výstupem

E.2.4 Příkaz INPUT_ACTIVATED

Význam: Příkaz se odešle vždy, když se zmáčkne tlačítko.

index	délka (byte)	hodnota	význam
0	1	0x12	Hlavička příkazu
1	1	0x07	Délka dat
2	1	?	Index výstupu/tlačítka
3	4	?	Časová známka
7	1	0x53	První oddělovací znak
8	1	0xFF	Druhý oddělovací znak

Tabulka E.14: Struktura příkazu s aktivovaným vstupem

E.2.5 Příkaz NEXT_SEQUENCE_PART

Význam: Příkaz se odešle na server ve chvíli, kdy je inicializovaný experiment typu ERP. Pomocí tohoto příkazu se požádá server, aby dodal část ERP sekvence do stimulátoru. Příkaz se bude odesílat také v průběhu experimentu.

index	délka (byte)	hodnota	význam
0	1	0x20	Hlavička příkazu
1	1	0x09	Délka dat
2	1	?	Offset sekvence
3	1	?	Index bufferu ve stimulátoru
4	4	?	Časová známka
9	1	0x53	První oddělovací znak
10	1	0xFF	Druhý oddělovací znak

Tabulka E.15: Struktura příkazu pro získání části ERP sekvence