Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Software na ovládání LED pásků na platformě Arduino

a jeho využití v praxi

2019

Jiří Badal

Vedoucí práce: doc. RNDr. Michal Krupka, Ph.D.

Studijní obor: Informatika, prezenční forma

Bibliografické údaje

Autor:	Jiří Badal
Název práce:	Software na ovládání LED pásků na platformě Arduino (a jeho využití v praxi)
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2019
Studijní obor:	Informatika, prezenční forma
Vedoucí práce:	doc. RNDr. Michal Krupka, Ph.D.
Počet stran:	40
Přílohy:	1 CD/DVD
Jazyk práce:	český

Bibliograhic info

Author:	Jiří Badal
Title:	Software for controlling Arduino-based LED strips (and its practical use)
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2019
Study field:	Computer Science, full-time form
Supervisor:	doc. RNDr. Michal Krupka, Ph.D.
Page count:	40
Supplements:	1 CD/DVD
Thesis language:	Czech

Anotace

Tato práce pojednává o možnostech ovládání LED pásku pomocí platformy Arduino. V první části je popsán samotný protokol pro komunikaci mezi PC a Arduinem, druhá se zabývá implementací pro čip a třetí obsahuje popis a uživatelskou dokumentaci pro skriptovací jazyk.

Synopsis

This work deals with the possibilities of controlling a LED strips using the Arduino platform. The first part describes a protocol for communication between PC and Arduino. The second one pursues an implementation for a Arduino based čip. The third section contains a description and a user documentation for a scripting language.

Klíčová slova: Arduino; ESP; Adresovatelné LED pásky

Keywords: Arduino; ESP; Addressable LED strip

Tímto bych chtěl poděkovat svému vedoucímu bakalářské práce, kamarádům a kolegům ze skupin Blackout Paradox a B4Spin a své rodině.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval/a samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam vět

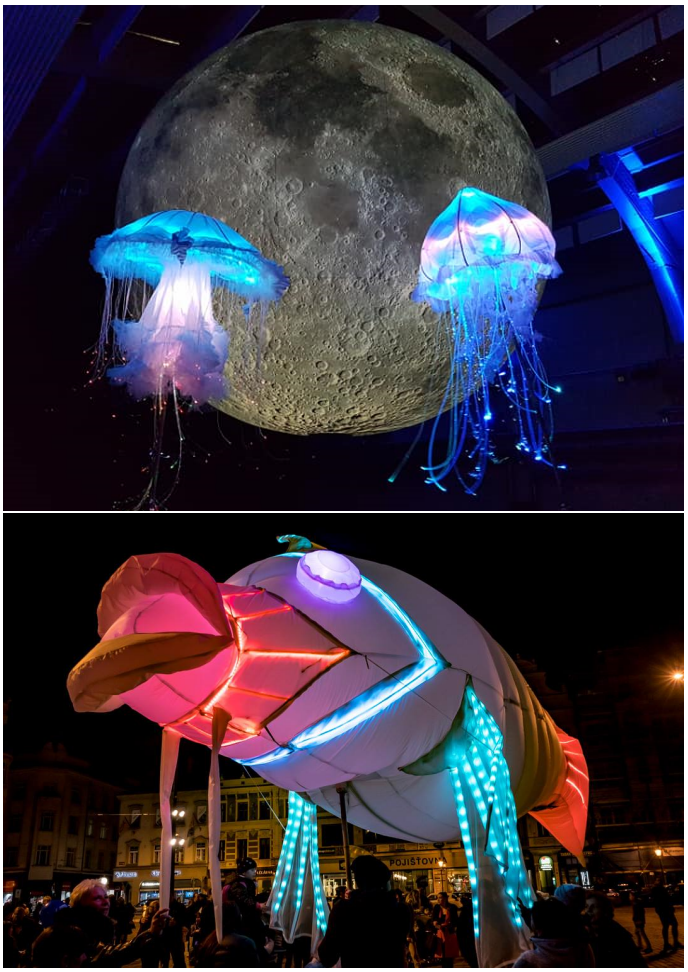
Seznam zdrojových kódů

1 Úvod

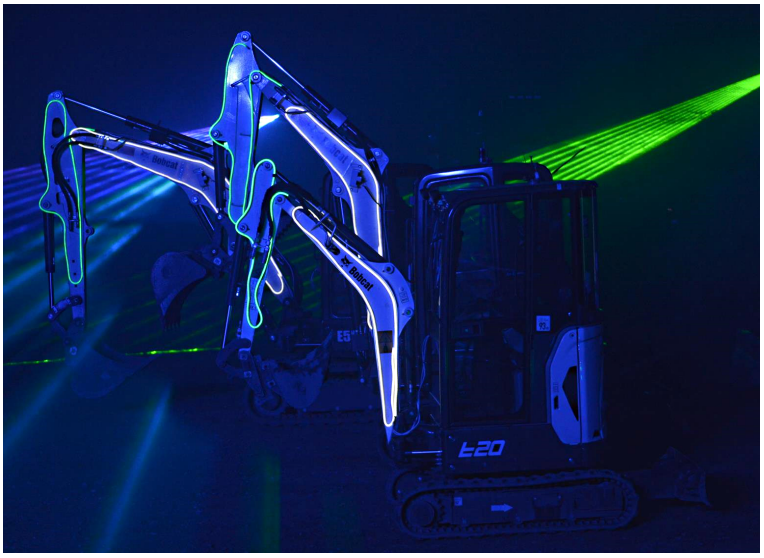
Už několik let se aktivně věnuji pohybovému divadlu a *fireshow*. V těchto oborech je často potřeba vytvářet svítící rekvizity a kostýmy pro dotvoření scény. Vhodným řešením pro tyto instalace díky ceně, jednoduché montáží a odolnosti jsou technologie pracující s elektroluminiscencí (dále LED). Většina dostupných řídících čipů umí pouze základní animace, které jsou pro účely divadla nedostačující, a proto jsem se rozhodl vytvořit systém pro ovládání LED světel.

Během vývoje systému se mi podařilo zůčastnit několika projektů, kde jsem mohl vyzkoušet některé části své dosavadní práce a ověřit její funkčnost a správnost řešení. Mezi významnější projekty patří spolupráce s firmou Bobcat CZ, a.s. na DemoShow 2018 nebo s agenturou ART Prometheus na projektu The Visitors. V současné době se objevují další možnosti uplatnění systému, a proto plánuji pracovat na vývoji i po odevzdání práce.

Ukázky světelných instalací lze nálézt na obrázcích 1 a 2, videa na odkazech Bobcat a The Visitors.



Obrázek 1: Projekt The Visitors



Obrázek 2: DemoShow pro Bobcat

2 Základní pojmy

V této kapitole jsou vysvětleny základní pojmy jako platforma Arduino a LED pásky. Problematika je rozebírána převážně z pohledu softwaru, hardwarové specifikace jsou uvedeny pouze okrajově.

2.1 Arduino

Pojem Arduino je možné chápat v několika různých souvislostech. V první řadě jde o označení pro konkretní typ jednodeskového počítače. (dále jen deska nebo čip), určeného k ovládání elektronických součástek (servo motory, LED diody...), získávání hodnot z připojených senzorů (gyroskop, teploměr...) a komunikaci s dalšími zařízeními.

Arduino lze téže možné chápat jako souhrné označení pro rodinu těchto jednodeskových počítačů, programovací jazyk a vývojové prostředí pro vytváření programů k těmto deskám. Samotní autoři Arduina na svém webu arduino.cc [1] píší: „Arduino is the world's leading open-source hardware and software ecosystem.“

Arduino vzniklo původně jako pomůcka pro výuku programování. v současné době se těší velké oblibě u nadšenců, kutilů a malých firem, zejména díky nízké ceně, spolehlivosti a jednoduchosti.

Jazyk Arduina je velice podobný jazyku C++, ze kterého přejímá syntaxi a sémantiku, a přidává funkce pro řízení elektronických součástek. Dokumentace jazyka je k dispozici rovněž na oficiálním webu *arduino.cc* [2].

Vývoj programu spočívá v definování funkcí `void setup()` a `void loop()`, které musí být v kódu proramu uvedeny, i kdyby měly mít prázdné tělo. Funkce *setup* se spustí pouze jednou po startu zařízení. Je dobrým zvykem v těle této

funkce provést veškeré nastavení čipu (nastavení rychlosti sériové linky, nastavení WiFi atd.) Po funkci *setup* se v nekonečném cyklu opakuje funkce *loop*, která by měla mít na starosti hlavní funkčnost čipu. Příklad jednoduchého programu je uveden v kódu 1.

```
1  int LOOP_COUNT = 0;          //definice globální proměnné
2
3  void setup() {
4    Serial.begin(9600);        //nastavení seriové komunikace
5    Serial.println("Setup complete"); //Výpis na seriovou linku
6  }
7
8  void loop() {
9    Serial.print("loop for: ");
10   LOOP_COUNT++;
11   Serial.print(LOOP_COUNT); // v~ nekonečném cyklu bude vypisovat
12   Serial.println(" times."); // kolikrát proběhla funkce loop
13   delay(1000);              // čip počká 1s
14 }
```

Zdrojový kód 1: Ukázka jednoduchého programu pro Arduino

Psaní a kompilace kódu probíhá na PC pomocí grafického vývojového prostředí *Arduino IDE*, poté je potřeba zkompilovaný kód nahrát do čipu. Tuto funkčnost rovněž obstarává vývojové prostředí.

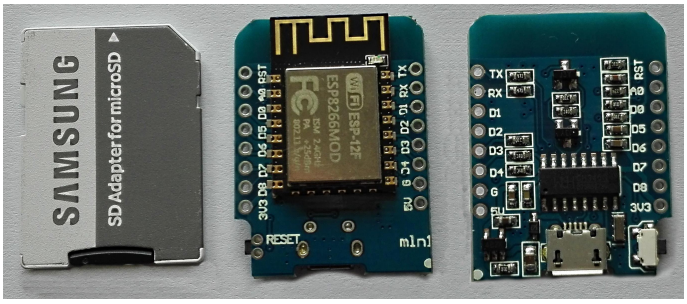
Součástí každé desky je procesor. U desek Arduino jde nejčastěji o procesory ATmega, které používají instrukční sadu RISC a pracují na frekvencích v jednotkách MHz. Pro připojení elektronických součástek slouží vstupně/výstupní piny. Jejich počty a funkce se liší v závislosti na konkrétní desce.

Arduino je open-source, proto existuje mnoho čipů, které fungující podobně. Pro tyto desky se vžilo označení klon. Implementace protokolu by po drobných úprávách kódu měla být kompatibilní s většinou těchto klonů. Tyto úpravy se nejčastěji týkají načítaných knihoven.

Pro svoji práci jsem potřeboval desku s integrovaným WiFi modulem, proto jsem si vybral desku *Wemos D1 mini* s procesorem *ESP8266* (obrázek 3). Mezi kódem pro Arduino desku a Wemos jsou drobné rozdíly, které jsou uvedeny v dokumentaci *EPS 8266* [3]. Tato deska má kromě integrovaného WiFi modulu paměť o velikosti 4 MB a 32bitový procesor pracující na frekvenci až 160 MHz.

2.2 LED pásky

LED pásek můžeme zjednodušeně chápat jako sériově zapojené LED diody. LED pásky můžeme dělit podle různých kritérií. Jedním z nich je napětí s kterým pracují (5 V, 12 V, 24 V, ...), další jsou např. výkon, počet barev. Protože Arduino pracuje s 5 V napětím, budeme pracovat s pásky stejného napětí. Po drobných hardwarových úpravách je možné řídit i pásky pracující s vyšším napětím. Výkon



Obrázek 3: Wemos D1 mini s měřítkem

je zásadním kriteriem pro napájení systému, z hlediska softwaru je zanedbatelný. LED pásky jsou nejčastěji jednobarevné nebo tříbarevné, které světlo míchají ze tří základních barev *RGB* modelu. Nejdůležitějším kritériem pro software je způsob řízení pásku. Existují dvě možnosti:

1. Analogové LED pásky (na trhu k dostání jako RGB LED pásky)
    - každá barva má svůj vodič
    - intenzita barvy a se ovládá velikostí napětí na vodiči k_a
    - do čipu vedou z každého pixelu 4 vodiče
    - sériově zapojené LED pásky nelze ovládat jednotlivě
2. Adresovatelné pásky
    - každá LED má svůj mikročip („chipset“) ovládající její barvu
    - mikročip komunikuje s okolím digitálně
    - z čipu vedou do LED pásku jen 3 nebo 4 vodiče (podle „chipsetu“)
    - mikročip ze vstupu přečte určitý počet bytů a zbytek pošle na výstup

Zajímavější z hlediska ovládání jsou adresovatelné pásky, u kterých je možné každou LED ovládat zvlášť. Pro některé světelné instalace je vhodnější vybrat klasické RGB LED pásky z důvodu jejich nižší pořizovací ceny. Tyto LEDpásky je možné ovládat pomocí *PWM* signálu a vhodného zapojení (viz. kapitola 4.10)

3 Definice protokolu a popis jeho částí

Jedním z cílů této bakalářské práce je navrhnout funkční systém zajišťující komunikaci mezi čipem a kontrolerem, a proto je nutné navrhnout protokol definující:

- Podobu komunikace mezi čipem a kontrolerem
- Podporované funkce
- Chování čipu během zpracovávání zpráv
- Strukturu souborového systému čipu

Čip může být libovolné zařízení, které je schopno kontrolovat LED pásky a komunikovat s okolím pomocí WiFi. V našem případě budeme používat Arduino. Konroler je libovolné zařízení, které je schopno komunikovat s okolím pomocí WiFi.

Funkce je základní nástroj pro řízení čipu. Kontroler posílá čipu zakódované sekvence funkcí, které nazveme zprávy.

Následující podkapitoly popisují jednotlivé části protokolu.

3.1 Funkce

Funkce jsou základním nástrojem pro ovládání čipu. Obecně lze funkci f chápat jako množinu instrukcí v programovacím jazyce čipu, které implementují nějakou funkčnost čipu. Práce čipu je založena na vykonávání funkcí. Čip musí umožňovat *multitasking*, tedy alespoň zdánlivý souběh více funkcí.

Každá funkce f má definovaný počet parametrů $n \in \mathbb{N}$. Parametru p funkce f je přiřazeno jméno, velikost a typ. Jméno slouží pouze k lepšímu zapamatování pro uživatele. Velikost $|p| \in \mathbb{N}$ parametru p udává počet bytů potřebných pro uložení parametru. Typ parametru p určuje typ hodnot, které může parametru p přiřadit.

Korektní volání funkce $f(p_1, \ldots, p_n)$ je přiřazení hodnot $a_1, \ldots, a_n$ parametrům $p_1, \ldots, p_n$, tak že typ hodnoty a_i odpovídá typu parametru p_i. Počet a pořadí parametrů u dvou různých funkcí se může lišit.

Kód volání funkce (zkráceně kód funkce) je posloupnost bytů, ve kterém je zakódováno volání funkce $f(p_1, \ldots, p_n)$. Každému volání funkce $f(p_1, \ldots, p_n)$ lze jednoznačně přiřadit kód $m_{f(p_1,\ldots,p_n)}$. Délka kódu $|m_{f(p_1,\ldots,p_n)}|$ je rovna

$$|h_f| + \sum_{i=1}^{n} |p_i|, \text{ kde } h_f \text{ je hlavička funkce } f.$$

Hlavička h_f funkce f je číslo jednoznačně určující funkci f. Pro každé dvě různé funkce f a k platí $h_f \neq h_k$ a $|h_f| = 1$. V kódu je hlavička vždy na prvním místě.

Všechny funkce mají jeden společný dvou bytový parametr, ve kterém většinou bývá uložen čas prvního spuštění funkce *start-time*, a proto pro délku všech funkcí platí $|m_{f(p_1,\ldots,p_n)}| \geq 3$. Přehled všech parametrů funkcí je v tabulce 1.

Tabulka 1: Přehled všech parametrů a jejich velikostí

Velikost	Parametry
1	*head, wait, hue, fade, density, steps, turn, mode, peak-steps, start-hue, sequence-number, group*[a]*, chipset*[b]
2	*start-time, end-time, start-index, end-index, peak-start, peak-end, pixel-count*
3	*color1, color2*
n	*SSID*[c]*, password*[c]*, sequence*[d]

a $1 \leq group \leq 8$

b nabývá hodnot z tabulky 2

c velikost je délka řetězce, řetězec je vždy zakončený '\0'

d zbytek zprávy

Funkce jsou rozděleny na 3 základní skupiny:

- příkazy
- systémové funkce
- animace

Někdy budu zaměňovat pojemy „volání funkce“ a „funkce“, například „kód funkce“ označuje „kód volání funkce“. Podrobnější popis jednotlivých funkcí je uveden v kapitole 3.6.

3.2 Komunikace mezi čipem a kontrolerem

Cílem této kapitoly je definovat jak má probíhat komunikace mezi zařízeními a upřesnit podobu zpráv.

Komunikace probíhá pomocí *UDP datagramů* na *UDP portu* 5000. Z toho vyplívá, že náš protokol nezaručuje doručení a pořadí zpráv. *UDP* protokol jsem vybral, protože má nižší režii než protokol *TCP* a méně zatěžuje pomalé zařízení jako je Arduino. Datagramy je možné odesílat na konkrétní *IP* adresu čipu i *broadcast* adresu.

3.2.1 Skupiny čipů

Čip má uložené nenulové 8 bitové číslo, které nazveme *klíč*. Bity jsou očíslováné od 1 a od nejméně významného bitu. Pokud je bit na pozici n roven 1, čip patří do skupiny n. Pokud má čip klíč např. 11000001 patří do skupin 1, 7 a 8.

3.2.2 Zprávy

Zpráva a je dvojice $\langle key_a, F_A \rangle$, kde F_a je konečná posloupnost volání funkcí a key_a klíč zprávy a, který určuje skupinu čipů a $|key_a| = 1$.

Zpráva a je určena pro čip D pokud platí: $key_D \& key_a \neq 0$, kde key_D je *klíč* čipu D, key_a je klíč zprávy a a operace & je bitový součin. Pokud je $key_a = 0$, je zpráva a zahozena každým čipem a pokud $key_a = 255$ žádný čip nezahodí zprávu a.

Kód zprávy a budeme definovat jako posloupnost bytů, kde první byte je key_a a ostatní byty jsou vyhrazeny pro kódy $f_i(p_1, \ldots, p_n) \in F_a$. Z kódu zprávy a je možné jednoznačně získat zprávu a. Ke každé zprávě a je možné jednoznačně vygenerovat její kód.

Poslání zprávy a čipu D je realizováno odesláním UDP datagramu čipu D na port 5000, který obsahuje kód zprávy a.

Uložení zprávy a do souboru „x.txt“ je uložení kódu zprávy a do souboru „x.txt“ od druhého bytu (tedy bez klíče).

3.3 Zpracování zpráv a plánování běhu

Zpracování zpráv je proces čipu, který se odehrává po přijetí zprávy. Způsob zpracování zprávy ze síťové komunikace se trošku líší od způsobu zpracování zprávy ze souboru.

Pokud zpráva a pochází ze síťové komunikace, tak se všechny funkce z F_a přidají do plánovače. Plánovač je struktura, ve které jsou uloženy všechny funkce, které má čip vykonávat.

V případě je-li zpráva a ze souboru, tak se funkce $f_i \in F_a$ přidá do plánovače pokud splní podmínky

1. Všechny funkce f_j z množiny $\{f_j \in F_a | j < i\}$ už jsou v zařazeny v plánovači.

2. Funkce f_i je příkaz nebo parametr *start-time* funkce f_i je menší nebo roven lokálnímu času čipu.

Plánovač uchovává frontu všech funkcí. Základní operace s plánovačem jsou *addFunction* pro přidávání funkcí, *flushAnimation* pro smazání všech animací a *executePlan*. Operace *executePlan* postupně projde všechny funkce ve frontě a

- Vymaže všechny funkce, kterým vypršel čas
- Spustí funkce, které mají být spuštěny
- Přeskočí všechny funkce, které mají status *idle*

Tabulka 2: Přehled podporovaných „chipsetů“

chipset	Hodnota	Piny
apa102	0	4
ws2811	1	3
ws2801	2	4
ws2812b	3	3

3.4 Souborový systém čipu

Zprávu je možné ukládat do souboru, který musí být pojmenován n.txt, kde $0 \leq n \leq 255$ a $n \in \mathbb{N}^0$. Některé soubory mají speciální význam. Soubor *255*.txt je konfigurační soubor, otevírá se při startu čipu a načítají se z něj informace o WiFi, systémové funkce, které mají běžet atd. Soubor *254*.txt se spustí pokaždé, když není dostupný Wi-Fi signál. Soubor *253*.txt se naopak spustí, když se čip úspěšně připojí k WiFi. Existují dva způsoby jak nahrát soubor do paměti čipu

1. přes Arduino IDE pomocí *Sketch data upload*
2. pomocí protokolové funkce *save-sequence* (viz. kapitola 4.7)

3.4.1 Struktura souboru

Funkce v souboru musí být vzestupně setřízeny podle parametru *start-time*. Pokud tomu tak není, protokol nezaručuje, že čip bude přidávat funkce v korektním čase. Příkazy lze rovněž ukládat. Pokud od nich vyžadujeme konkrétní čas spuštění a neexistuje systémová funkce, která funguje stejně, využijeme „triku“. Trik spočívá, že před příkaz přidáme nějakou jinou funkci s parametrem *start-time* rovným startovacímu času příkazu.

3.5 Jednotky času

Samotné arduino počítá čas v milisekundách. Určovat parametry *start-time* nebo *end-time* s takovou přesností je zbytečné plýtvání místem. Parametry *start-time* a *end-time* mají velikost 2 B a jsou uvedeny v jednotkách $1j = 100ms$. Pokud bychom nesnížili přesnost, maximální trvání funkce by bylo 2^{16} ms (1,09 min), takto můžeme mít funkce trvající $2^{16} \cdot 100ms = 1,82h$. Další informace a omezení ohledně času jsou uvedeny v kapitole 4.9.1.

Někdy může být výhodné mít k dispozici funkci, která není časově omezená. Toho lze docílit tak, že nastavíme parametr *end-time* $\leq$ *start-time*. Plánovač pak tuto funkci neodstraní v důsledku vypršení času.

3.6 Podporované funkce

3.6.1 Příkazy

První parametr není využit jako *start-time*, ale jako obecný parametr. Některé příkazy ho vůbec nevyužívají, nicméně v kódu funkce musí být uveden. Pokud je v tomto parametru uvedena jedno bytová informace uloží se do prvního bytu. Například kód příkazu *open-sequence* může vypadat takto *7 - 0 - 5*, (otevři soubor *5.txt*). Přehled všech příkazů je uveden v tabulce 3.

3.6.2 Systémové funkce

Některé systémové funkce jsou jenom příkazy, které jdou zařadit do plánovače. V kódu je první vždy hlavička a poté parametr *start-time*. v tabulce 4 jsou uvedené pouze parametry, které jsou navíc.

3.6.3 Animace

Všechny animace mají společné parametry *start-time, end-time, start-index, end-index*. Parametry *start-time* a *end-time* určují časové ohraničení funkce. Množina LED, na které animace operuje, je určená jako $\{x \in LEDs | start - index \leq index(x) \leq end{-}index\}$. Kde $LEDs$ je množina všech pixelů na pásku a $index(x)$ je index pixelu v pásku (číslováno od 0).

Dalšími často se vyskytujícími parametry jsou *color1, color2, wait, turn*. Parametry popisující barvu (*color1, color2*) mají velikost 3 bytů (na každou barvu 1 B). Parametr *wait* udává kolik snímků bude funkce čekat, než ji plánovač znovu pustí. Pokud tedy má funkce nastavený $wait = 1$, plánovač ji v jednom snímku přeskočí a v dalším snímku ji již pustí. Pokud během čekání vyprší funkci čas, plánovač ji již nespustí. Parametr *wait* má velkost 1 bytu.

Přehled všech parametrů animací a jejich velikostí je uveden v tabulce 1. Všechny podporované animace, lze nalézt v tabulce 5

Tabulka 3: Přehled všech příkazů

Příkaz	**H**	**Parametry**	**Popis**
restart-time	1	-	Vynuluje lokální čas čipu.
flush-animation	2	-	Vymaže z plánovače všechny animace.
pause	3	-	Zastaví počítání času a vykreslování animací, ostatní funkce fungují normálně.
unpause	4	-	Pokud má čip pozastaveno vykreslování znovu ho spustí, jinak nedělá nic.
restart-čip	5	-	Nejdříve zavře otevřené soubory a poté restartuje zařízení.
turn-off	6	-	Bezpečně vypne čip.
open-sequence	7	*sequence-number*	Otevře sekvenci uloženou v souboru *sequence-number.txt*.
close-sequence	8	-	Zavře aktuálně otevřenou sekvenci.
send-info-message	9	-	Pošle informace o sobě *broadcastem*.
print-info-message	10	-	Vytiskne informace o sobě na seriovou linku.
set-connection-parameters	11	*SSID, password*	Nastaví SSID a heslo k WiFi, mezi *head* a *SSID* jsou dva byty nevyužité.
set-pixel-count	12	*pixel-count*	Nastavuje počet pixelů.
save-sequence	13	*sequence-number, mode, sequence*	Viz kapitola 4.7 .
set-configuration	14	*pixel-count, SSID, password*	Nastaví *pixel-count*, *SSID* a *password*.
initialize-drawing	15	-	Initializuje potřebné struktury pro vykreslování.
initialize-communication	16	-	Initializuje potřebné funkce pro síťovou komunikaci.
set-group	17	*group*	Nastaví skupinu čipu na *group*.
add-group	18	*group*	Přidá *group* do skupin čipu.
set-chipset	21	*chipset*	Nastaví *chipset* LED pásku. Přípustné hodnoty pro *chipset* jsou v tabulce 2

Tabulka 4: Přehled systémových funkcí

Funkce	**H**	**Parametry**	**Popis**
restart-time-function	32	-	Vynuluje čítač času
flush-animations-function	33	-	Vymaže z plánovače všechny animace
open-sequence-function	34	*sequence-number*	Otevře soubor sekvence určený parametrem *sequence-number*
close-sequence-function	35	-	Zavře soubor sekvence
print-info-message-function	58	-	Pravidelně tiskne info na seriovou linku
check-message-box	59	-	Kontroluje jestli není dostupná zpráva na UDP portu 5000, jestli ano zpracuje ji.
actualize-clock	60	-	Funkce pro aktualizaci času
check-wifi-statu	61	-	Pravidelně kontroluje stav WiFi, pokud se změní na připojeno otevře se soubor *253.txt*. Když zařízení ztratí signál otevře se soubor *254.txt*
read-file	62	-	Je vždy v plánovači, přidává funkce z otevřeného soubou
draw-pixel	63	-	Funkce která má nastarosti pravidelné vykreslování pásku

Tabulka 5: Podporované animace

Animace	**H**	**Parametry**	**Popis**
periodic-solid-color	64	*color1, wait*	periodicky vykresluje *color1*
periodic-rainbow-fill	65	*start-hue*	periodicky vykresluje duhu
strobo	66	*color1, color2, wait*	přepíná mezi *color1* a *color2*
solid-color	67	*color1*	vybarví pásek barvou *color1* a skončí
rainbow-fill	68	*start-hue*	zaplní pásek duhou
random-dots	69	*color1, fade, density, wait*	generuje náhodný šum v barvě *color1*
pulse	70	*color1, color2, wait, steps*	přechází mezi *colors* v *steps* krocích
rainbow-run	71	*hue, wait, turn*	kreslí duhu, každý snímek posune *hue* o *turn*
wawe	72	jako *peak-fade* + *peak-step, wait*	kreslí *peak-fade,* který každý snímek posouvá o *peak-steps*
fading	73	*fade, wait*	snižuje jas pixelu o *fade*
peak-fade	74	*color1, peak-start, peak-end, fade*	mezi *p-s* a *p-e* vykreslí *color1*, ostatní pixely ztmavuje o *fade*
peak-fade-aditive	75	jako*peak-fade*	funguje stejně jako *peak-fade,* akorát k pixelům barvu přičítá
add-color	76	*color1*	přičte k pixelům *color1*
wawe-aditive	77	jako *wawe*	jako *wawe*, jen barvu k pixelům přičítá
periodic-add-color	78	*color1, wait*	periodická funkce *add-color*
sub-color	79	*color1*	odečte od pixelů *color1*
perodic-sub-color	80	*color1, wait*	periodická funkce *sub-color*

4 Implementace protokolu

Implementace protokolu pro čip je rozdělená do několika souborů. Hlavní soubor má koncovku *ACID.ino* a jsou v něm napsané funkce *setup, loop*. Dále pak tento soubor implementuje zavádění zpráv do systému. Knihovna *RunManager.h* poskytuje strukturu pro plánovač a operace plánovače. *ComManager.h* obstarává posílání a příjmání zpráv, *FileManager.h* přidává funkčnost zpracování souborů, jejich ukládání a čtení, a knihovna DrawingLibrary poskytuje funkce pro kreslení na LED pásek. Nejnižší komunikaci s LED páskem zajišťuje knihovna *FastLED*.

4.1 Knihovna FastLED

Tato openSource knihovna poskytuje abstrakci LED pásku, základní barevné modely, rychlé funkce náhodných čísel, rychlé matematické funkce a jednoduché kreslící funkce. Funkce, u kterých je kritický výkon jsou psány v asembleru.

Pro ovládání pásku se používá pole `CRGB* PIXELS[n]`, kde n je počet pixelů na pásku. `CRGB` je datový typ z knihovny *FastLED* pro barvu. Pro vykreslení pole je třeba zavolat funkci `FastLED.show()`. Bližší informace je možno nalézt na webové stránce *fastled.io* [4].

4.2 Implementace plánovače

Čip si udržuje posloupnost struktur, ve které jsou uloženy informace o funkci a její paměti. Tuto posloupnost nazveme jako *plán*. Snímek je spuštění všech funkcí uložených v *plánu*.

V implementaci je tato struktura pojmenována *node* (zdrojový kód 2) a obsahuje v sobě přímo odkaz na další *node* v pořadí. Funkce uložené v této struktuře musí být definované jako `char jménoFunkce (node* a)`. Musí skončit po provedení své práce na jednom snímku. Pokud už nemají běžet v dalších snímcích vrací 0, jinak vrací číslo n ($n \neq 0$). Čip implementuje kooperativní multitasking.

```
1 typedef struct node {
2   char (*function)(node*); // funkce uložená ve struktuře
3   char* parameters;  // pamět' přidělená funkci
4   unsigned char size; // velikost paměti
5   node* next;        // odkaz na další prvek ve struktuře
6 } node;
```

Zdrojový kód 2: Struktura *node*

4.3 Příznaky

Jsou to jedno bitové informace o některých vlastnostech funkce.

- typ funkce - 0 pro systémové funkce, 1 pro animace
- waitForStart tag (WFSTag) - 0 již spuštěná funkce, 1 čeká na spuštění
- hasNoEnd tag - 0 funkce má časové omezení, 1 funkce nemá časové omezení

4.4 Správa paměti

Velikost operační paměti se u čipů arduino pohybuje ve stovkách kilobytů. S pamětí je tedy potřeba nakládat obezřetně.

4.4.1 Přidělování paměti funkcím

Každá funkce potřebuje mít přidělenou paměť pro uložení svých parametrů a proměných, aby bylo možné přepínání mezi jednotlivými funkcemi.

Paměť, která je přidělená nějáké funkci f, je implementovaná jako pole bytů velikosti n. Jednotlivým skupinám bytů jdoucí po sobě říkáme funkční proměnné. Funkční proměné mají určené pořadí. Proměnnou, která má první ze svých bytů uložených na indexu 0 v poli, nazveme první.

Standardní uspořádání paměti je takové uspořádání, kdy v prvním proměnné jsou uložené příznaky, druhá proměnná je vyčleněna pro počet snímků, kdy nemá funkce běžet tzv. *sleeping-steps*, třetí proměnná je vyčleněna pro uložení času prvního spuštění funkce *start-time*. Jakmile je funkce spuštěna, toto místo může být využito pro lokální proměnné. Pořadí a význam dalších proměnných většinou odpovídá parametrům funkce. Některé systémové funkce z důvodu šetření paměti nevyžadují žádnou paměť.

Všechny animace mají standardní uspořádání a dále mají stejné pořadí těchto proměnných:

- 4. proměnná *end-time*
- 5. proměnná *start-index*
- 6. proměnná *end-index*

4.4.2 Garbage collector

V jazyce C++ není defaultně implementovaný garbage collector a arduino je pomalé zařízení, u kterého by režie plnohodnotného garbage collectoru byla výrazně znát. Proto jsem se rozhodnul implementovat vlastní jednoduší systém.

V tomto systému se všechny zahozené *node* ukládají do seznamu `SCRAP_HEAP`. Pokud systém potřebuje funkci f přiřadit nový node, nejdříve prohledá `SCRAP_HEAP` a hledá *node* který má stejnou velikost přidělené paměti jako potřebuje funkce f. Jestliže takový node existuje, systém ho přidělí funkci f, jinak funkci f přidělý nově alokovaný node. Jelikož existuje mnoho funkcích vyžadujících stejné množství paměti, tento systém funguje v praxi dostatečně dobře.

Možným vylepšením tohoto systému by mohlo být využití jiné struktury pro uložení vyřazených *nodu* než je seznam (např. skutečně haldu, nějáký binární vyhledávací strom). Dále by bylo možné implementovat hlídač, který když bude seznam plný, uvolní všechny nody ze seznamu. Další možná úprava je, že algoritmus pro vyhledávání recyklovaného nodu místo podmínky $size_{hledana} = size_x$ použije podmínku

$$size_{hledana} \leq size_x \text{ a současně } size_{hledana} - size_x \leq \varepsilon,$$

kde x je *node* $\in$ `SCRAP_HEAP` a $\varepsilon \in \mathbb{N}$ je maximální odchylka.

4.5 Spouštění funkcí ze struktury

Spouštění funkce f nodu A zajišťuje funkce `char startFunction (node* A)` (pseudokód 3). Pokud $startFunction(A) = false$ je funkce f z A určena k zahození. Funkce *startFunction* v těchto případech nespustí funkci f z A:

1. Je pozastaveno vykreslování a f je animace.
2. Funkce f ještě nebyla spuštěna a *start-time* $\geq$ *actual-time*.
3. Funkce f má $sleepingSteps \neq 0$.

V prvním a druhém případě je návratové hodnota funkce *startFunction* vždy $startFunction(A) = true$. Ve třetím pak

$$startFunction(A) = hasNoEnd(A) \vee isTimeToEnd(A).$$

Tedy pokud má f časové omezení a vypršel jí čas, je f určena k vyhození.

4.6 Zpracování zpráv

Zprávy obdržené přes udp komunikaci zpracovává funkce **`void`** `udpResolver ( Stream *udpMessage)`. Postup zpracování je následující:

1. *UdpResolver* přečte z *udpMessage* klíč a ověři jestli je zpráva pro něj.
2. pokud ano *udpResolver* volá `messageSolver(udpMessage)`, dokud nenarazí na konec zprávy.

Funkce messageSolver je přetížená metoda, `messageSolver(Stream* message )` pouze přečte z *message* první byte jako hlavičku, dva byty jako *start-time* a zavolá funkci **`void`** `messageSolver(char head, short startTime, Stream * message)`. Tato funkce podle hlavičky rozhodne jako initializační funkci spustit. Hlavička $0 \leq h \leq 255$. Tento interval je rozdělen na 8 stejných částí. Prvním třem částím jsou přiděleny inicializační funkce, zbylé jsou volné pro další rozšiřování. Initializační funkce přečtou zbytek kódu funkce. Podle druhu funkce pak vytvoří *node* a přidají ho do plánovače nebo vykonají příkaz.

Základní inicializační funkce jsou:

```
1  char startFunction (node* a) {
2    f = getFunction(a);
3    if(isAnimation(a) && GLOBAL_PAUSE) { // pozastaveno vykreslování
4      return 1;                        // a~f je animace
5    }
6
7    if(waitingForStart(a)) {           // f ještě nebyla spuštěná
8      if(isTime(getStartTime(a))) {    // je čas spustit f poprvé
9        setStartTime(a, 0);            // uvolnění místa pro lk
10       result = f(a);                 // samotné zavolání funkce
11       setWaitingForStartTag(a, 0);   // nastavení WFS tagu
12       return result;                 // pokud je 0, bude vyřazena
13     }
14     return 1;                        // funkce dál čeká na start
15
16   } else {                           // f již byla někdy spuštěna
17     sleep = getSleepingStep(a);
18     endTime = getEndTime(a);
19     hasTimeEnd = getHasEndTag(a);
20     if(sleep) {                      // funkce "spí" sleep snímků
21       setSleepingStep(a, sleep - 1); // sniž počet snímků o~1
22       return !hasTimeEnd || !isTime(endTime);// kontrola času
23     } else {
24       if(!hasTimeEnd){               // funkce není časově ohraničena
25   return f(a);                 // o~jejím vyřazení rozhoduje pouze f(a)
26       }
27       return !isTime(endTime) && f(a); // funkce je časově omezena
28     }
29   }
30 }
```

Zdrojový kód 3: Pseudokód funkce *startFunction*

- inicializační funkce pro příkazy
  - **void** `runCommand(char head, short parameter, Stream *msg)`
  - nepřidává žádné node do plánovače, rovnou spouští kód funkce
  - hlavička 0 - 31
- Inicializační funkce pro systémové funkce
  - **void** `initializeSystemFunction (char head, short startTime, Stream *msg)`
  - některým funkcím nepřiděluje pamět, objevují se tu různé zvláštnosti a pořušení předchozích pravidel (nestandartní uložení proměných)
- inicializační funkce pro animace

- `void initializeBasicAnimation (char head, short startTime, Stream *msg)`
- vždy přečte z msg minimálně endTime (2 byty), startIndex (2 byty) a endIndex(2 byty)
- `BASIC_ANIM_SIZE - 1` je index posledního bytu funkční proměné, kterou mají všechny animace společné

4.7 Ukládání zpráv

Zprávy ukládá funkce *save-sequence*. Ta má tři parametry, *sequence-number, mode, sequence*. Parametr *sequence* je posloupnost funkcí, která se uloží do souboru „*sequence-number*.txt“. Pokud soubor již existuje a parametr $mode \neq 0$, soubor je přepsán. v opačném případě je *sequence* připojena na konec souboru.

4.8 Zpracovávání zpráv ze souboru

Soubory zpracovává funkce `car readFile (node* a)`. Tato funkce postupuje následovně:

1. Jestli není otevřený soubor, funkce končí `return true;`
2. Ověří jestli se od posledního běhu změnil otevřený soubor

   ano načte *head* a *start-time* z právě otevřeného souboru

   ne načte *head* a *start-time* ze své paměti
3. Dokud *start-time* $\geq$ *local-time* nebo *head* určuje příkaz
   (a) zavolá `messageSolver(head, startTime, &SEQUENCE_SOURCE)`
   (b) jestli je na konci souboru, zavře soubor a vyskočí z cyklu
   (c) načte *head* a *start-time* ze souboru
4. Uloží si poslední *head* a *start-time* do své paměti a `return true;`

4.8.1 Zavádění konfiguračního souboru

Po startu čipu je v plánovači zařazena jenom funkce *read-file* a příkaz k otevření konfiguračnímu souboru. Ostatní funkce se načtou právě z konfiguračního souboru. Pokud tedy není v konfiguračním souboru funkce pro vykreslování, čip nebude vykreslovat na pásek. Minimální a doporučená konfigurace zařízení je uvedena v tabulce 6.

Minimální konfigurace
draw-pixel
initialize-drawing
*drawing-parameters**
*setting-chipset***

* není-li uvedena je nastavena hodnota 10

** není-li uvedena je nastavena **apa102**

Doporučená konfigurace
actualize-clock
print-info-message
check-wifi-status
check-message-box
draw-pixel
initialize-communication
initialize-drawing
set-configuration
setting-chipset

Tabulka 6: Minimální a doporučená konfigurace čipu

4.9 Počítání času a jeho aktualizace

Arduino poskytuje funkci `unsigned long millis()`, která vrací počet milisekund od spuštění zařízení. Tento časový údaj by neměl být používán v žádné jiné funkci než je *actualize-time*. Pro ostatní funkce je stěžejní aktuální čas.

Aktuální čas je uložen v globální proměnné `unsigned long LOCAL_TIME`. Počátek tohoto času je počítán od posledního zavolání funkce **`void`** `resetTime()`. Aktuální čas obnovuje každý snímek protokolová funkce *actualize-time*.

Tuto funkčnost implementuje funkce `char timer (node* a)`, která zároveň počítá průměrnou délku mezi snímky (proměnná `TICK_TIME`). Proměnná `TICK_TIME` se spočítá:

$$TICK\_TIME = \frac{tick\_time_{last} + (local\_time_{new} - local\_time_{last})}{2}.$$

4.9.1 Omezení času

Počítání času má několik omezení vyplývající z omezené paměti a velikosti čísla. Samotná proměnná `LOCAL_TIME` přeteče po 2^{32} ms (≈ 50 dnů). Stejné omezení má funkce *millis()*. Všechny časové parametry funkcí mohou nabývat maximálně $100 * 2^{16}$ ms ($\approx 1,82$ h). V praxi většinou jsou tyto limity nedosažitelné, uvádím je zde pro úplnost.

4.10 Řízení neadresovatelných LED

Pro řízení analogových LED diod je potřeba změnit připojení pásku k čipu (viz obrázek 4). Pro detaily zapojení doporučuji navštívit webovou stránku [**?**], odkud je převzaný i obrázek 4. Takto zapojený pásek je možné ovládat jenom jako jeden pixel. Jestliže deska má více *PWM* pinů, je možné zapojit i více segmentů. Na *ESP8266* je možné zapojit dva segmenty.

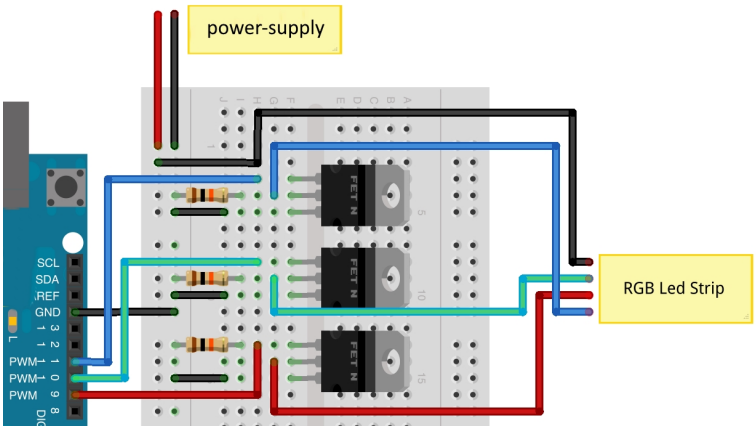
```
1  char timer (node* a) {
2   unsigned int newLocalTime;
3   if (GLOBAL_PAUSE) {              // vykreslování je pozastaveno
4     TIMER_BREAK = true;            // čas se neaktualizuje
5     return 1;
6   } else {
7     if (TIMER_BREAK) {             // první spuštění po pauze
8       if(LOCAL_TIME) {             // během pauzy neproběhlo
9         TIMER_BREAK = false;       // restartování času
10        TIME_START = millis() - LOCAL_TIME; // posunutí TIME_START
11        LOCAL_TIME = LOCAL_TIME + TICK_TIME;// drobná heuristika
12      } else {
13        resetTime();               // během pauzy došlo
14        LOCAL_TIME++;              // restarování času
15      }
16      return 1;
17    }
18
19    newLocalTime = millis() - TIME_START; // nejběžnější aktualizace
20    TICK_TIME = ((newLocalTime - LOCAL_TIME) + TICK_TIME) / 2;
21    LOCAL_TIME = newLocalTime;
22    return 1;
23  }
24 }
```

Zdrojový kód 4: Funkce *actualize-time*

Pro řízení se využívá PWM modulace [6]. Knihovna *FastLED.h* nepodporuje ovládání těchto pásků. Nicméně poměrně jednoduchou úpravou kódu je možné si tuto funkčnost doprogramovat a zároveň i ponechat možnost souběžného ovládání adresovatelného pásku.

Úprava spočívá v rozdělení pole pixelů na dvě části. První část odpovídá RGB páskům a druhá adresovatelnému pásku. Pixely náležící do části pro adresovatelné pásky, budeme zobrazovat jako doposud, rgb pixely zobrazíme pomocí funkce *showRGBSegments*. Tato funkce používá k zobrazování barevné složky c_i segmentu k funkci `analogWrite(pin, value)`, kde *pin* je číslo pinu náležící barvě c_i segmentu k a *value* je velikost složky c_i.

Důležité je zmínit, že některé desky pracují z různým rozsahem *value*. Arduino používá standardně rozsah od 0 do 255, ale třeba *ESP8266* používá rozsah od 0 do 1023.



Obrázek 4: Zapojení RGB LED pásku

5 Skriptovací jazyk

Úkolem skriptovacího jazyku je poskytnout nástroj pro vytváření sekvencí funkcí a jejich následný překlad do kódu a uložení do souboru. Skriptovací jazyk nezprostředkoává komunikaci mezi čipem a PC.

5.1 Implementace jazyka

Skriptovací jazyk je implementovaný jako knihovna pro programovací jazyk Common Lisp. Skriptovací jazyk je testovaný v implementaci SBCL, nicméně by měl fungovat i v ostatních implementacích Common Lispu.

Pro každou funkci (vyjma *save-sequence*) je napsána třída, která zajišťuje veškerou potřebnou funkcionalitu pro nastavování, hlídání rozsahu a typu parametrů. Přiřazení parametrů $\{p_1, \ldots, p_n\}$ funkci f je realizováno jako vytvoření instance třídy f a nastavení vlastností $\{p_1, \ldots, p_n\}$. Všechny třídy funkcí dědí z třídy *basic-function*, která poskytuje všechny základní metody a vlastnosti.

Pro seskupování funkcí je určena třída *program*, která rovněž zajišťuje vzestupné uspořádání funkcí podle parametru *start-time* ve výsledném kódu.

5.1.1 Generování kodu

Generování kódu z programu zajišťuje metoda `(get-code program)`, kde *program* musí být instance třídy *program*. Tato metoda zavolá `(get-code function)` na všechny funkce, které patří do instance *program*. Argument *function* je instance třídy nějáké funkce.

Metoda `(get-code function)` získá hodnoty a velikost parametrů funkce *function* (viz kapitola 5.1.2) a pomocí lispové funkce *split-into-bytes* získané hodnoty rozdělí do jednotlivých bytů.

Funkce *split-into-bytes* má jako argumenty seznam hodnot *values* a seznam velikostí *sizes*. Funkce rekurzivně prochází seznam *values*, dokud nenarazí na konec seznamu. v každé iteraci odebere jeden prvek seznamu *sizes* a *sizes* prvků

ze seznamu *values*. Prvky z *values* rozděluje na jednotlivé byty a přidává je do seznamu *accum*, který je na konci rekurze vrácen jako výsledek. Zrojový kód funkce split-into-bytes 5

```
(defun split-into-bytes (values sizes &optional (accum nil))
  (if (not sizes) (reverse accum) ;; zastavení rekurze
     (cond ((equal (first sizes) 1) ;; 1 bytový parametr
      (split-into-bytes (cdr values)
                        (cdr sizes)
           (cons (cut (first values))
                                  accum))) ;; ořezání
     ((numberp (first sizes)) ;; více bytový parametr
      (split-into-bytes-bigger-value values sizes accum))
     ((equal (first sizes) 'n) ;; zpracování stringu
      (split-into-bytes-string values (cdr sizes) accum)))))

;; do accum přidá rozdělenou hodnotu z values, sníží (first sizes)
;; a~zavolá zpět funkci split-into-bytes

(defun split-into-bytes-bigger-value (values sizes &optional (accum
    nil))
  (let* ((actual-size (- (first sizes) 1))
   (actual-value (cut (ash (first values)
        (* -8 actual-size)))))
    (split-into-bytes values
         (cons actual-size
         (cdr sizes))
         (cons actual-value accum))))
```

Zdrojový kód 5: Funkce *split-into-bytes*

5.1.2 Získávání jmen, velikostí a hodnot parametrů

Objekty funkcí mají vlastnosti *parameters-size, parameters-name*, obě tyto vlastnosti jsou pouze ke čtení. Metody sloužící jako „reader“ mají stejné jméno jako vlastnosti a vrátí seznam velikostí/jmen parametrů funkce v pořadí, které odpovídá pořádí v kódu.

Implementace těchto metod využívá dědičnosti. Ve třídě *basic-function* jsou defnované metody, které vrací seznam `'(n1 n2)`, kde $n1, n2 \in \mathbb{N}$ pro *parameters-size* nebo jsou symboly (ve smyslu Lispu) pro *parameters-name*. Ve třídách dalších funkcí pak stačí přepsat tyto metody jak je uvedeno ve zdrojovém kódu 6

Seznam hodnot všech parametrů poskytuje metoda *get-all-parameters-values*. Díky vlastnostem lispu je kód funce elegantní, krátký a společný pro všechny funkce. (viz. kód 7)

```
(defmethod parameters-name ((func func-class))
   (append (call-next-method) '(name1 ... nameN)))

(defmethod parameters-size ((func func-class))
   (append (call-next-method) '(num1 ... numN)))
```

Zdrojový kód 6: Přepsání *parameters-name* a *parameters-size*

```
(defmethod get-all-parameters-values ((func basic-function))
  (mapcar (lambda (parameter) (funcall parameter basic-function))
        (parameters-name basic-function)))
```

Zdrojový kód 7: Metoda *get-all-parameters-values*

5.1.3 Přídávání nových funkcí

Pro přidání funkce je potřeba upravit zdrojový kód ACIDScript.lisp. Pro přidání funkce je potřeba vytvořit novou třídu. Podle parametrů funkce si zvolíme, z které třídy bude nová třída funkce dědit (viz. tabulky 5, 4 a 3) Pokud funkce má nějáké nové parametry, přepíšeme metody *parameters-size* a *parameters-size* jak bylo uvedeno v kapitole 5.1.2. Parametry je potřeba přidat jako vlastnost objektu. Důležité je, aby souhlasilo jméno vlastnosti i metoda pro čtení vlastnosti. Aby bylo možné nastavovat hodnotu vlastnosti pomocí `(setf (vlastnost objekt) value)`, musí metoda *vlastnost* definovat místo. Je nutné přepsat metodu *head*. Nakonec zavoláme jednu z funkcí:

- `(add-suported-animation 'name)` pro animace
- `(add-suported-system-function 'name)` pro systémové funkce
- `(add-suported-command 'name)` pro příkazy

Argument *name* je jméno třídy funkce.

5.1.4 Reverzní překlad

Reverzní překlad je překlad z kódu funkce do instance třídy funkce. K vyřešení tohoto problému je potřeba míti tabulku s dvojicemi *hlavička, funkce*. Funkce pro překlad si nejdříve vyhledá podle hlavičky funkci, vytvoří instanci této funkce a nastaví ji všechny parametry. Jelikož je tato funkcionalita trošku navíc, skriptovací jazyk zatím nepodporuje reverzní překlad celého dokumentu. Překlad jednotlivých funkcí využívá v metodě `(copy-function instance-tridy-funkce)`.

5.1.5 Přizpůsobení jazyka pro konkrétní světelnou instalaci

Výhoda ponechání jazyka ACIDScript na úrovni knihovny Lispu je snadná rozšiřitelnost znalými uživateli. Například při častém opakování stejné sekvence animací, kde jsou různé jenom parametry, je dobré si na to napsat funkci. Skriptovací jazyk poskytuje třídu *device*, jejím úkolem je ulehčit zdlouhavé nastavovaní parametrů *start-index* a *end-index* u zařízení které mají pásek rozdělený na více segmentů. Definice segmentů se provádí během initializace instance. Napsat si třídu pro konkrétnější zařízení, která dědí z *device* a má rovnou definované segmenty je další příklad vhodného přizpůsobení.

5.2 Uživatelská příručka jazyka ACIDScript

Program v jazyce ACIDscript je posloupnost funkcí z protokolu. Zdrojové kódy skriptovacího jazyka ACIDscript mají příponu *.lisp*. V těchto souborech je možné definovat i více programů. Pro překlad ze zdrojového kodu jazyka do kódu pro čip je potřeba mít spuštěný interpret programovacího jazyka lisp a načtený soubor ACIDscript.lisp `(load "file/path/ACIDscript.lisp)`.

5.2.1 Vytvoření a uložení programu

syntax `(generate-program path f1 ... fn)`

sémantika do souboru určeného *path* ulož sekvenci funkcí $f_1 \dots f_n$.

Funkce $f_1 \dots f_n$ nemusí být v zápisu seřazeny v zestupně podle parametru *start-time*. o setřízení se postará sám generátor. Pokud mají některé funkce f_i, f_j kde $i < j$ stejnou hodnotu parametru *start-time* bude v programu první f_i (ta, která je první uvedena v zápisu).

Vygenerování a uložení kódu pro čip proběhne až po zkopírování vámi definovaného výrazu `(generate-program path f1 ... fn)` do interpretu jazyka Common Lisp.

Příklad kódu pro jednoduchý program je uveden ve zdrojovém kódu 8. Tento jednoduchý skript rozsvítí zeleně pixely 0 až 10 a otevře soubor „2.txt“.

```
1 (generate-program "1.txt"
2               (make-instance 'solid-color
3                         :color1 '(0 255 0)
4                         :start-index 0 :end-index 10)
5               (make-instance 'open-sequence :sequence-number 2))
```

Zdrojový kód 8: Jednoduchý skript

5.2.2 Definice funkcí

sytax `(make-instance 'f :parametr1 hodnota1 ... :parametrN hodnotaN).`

sémantika vytvoř volání funkce *f* s uvedenými hodnotami parametrů.

Ve výčtu parametrů $parametr_1$ až $parametr_n$ nemusí být uvedeny všechny parametry funkce a mohou být uvedeny v libovolném pořadí. Ty které nejsou uvedeny, se nastaví na defaultní hodnotu.

Existují dvě možnosti jak zadat parametr barvy. První možnost je pomocí klíčového slova *:colorN*, kde *N* je pořadí barvy. Druhou možností je použít klíčové slova *:redN*, *:blueN* a *greenN*, kde *N* je pořadí barvy. Obě možnosti jsou použity v kódu 9

```
1 (make-instance 'wawe
2             :peak-step 1 :peak-fade 127
3             :end-index 100
4             :start-time 0 :end-time 0
5             :color1 '(0 255 127))
6 (make-instance 'solid-color
7             :red1 255
8             ;;:green1 se automaticky nastaví na 0
9             :blue1 127)
```

Zdrojový kód 9: Příklad definicí funkcí

5.2.3 Skriptování konfiguračního souboru

Konfigurační soubor lze rovněž programovat. *ACIDscript* poskytuje několik možností jak generovat kod pro konfigurační soubor:

1. Pomocí předpřipravených generátorů

   \+ Snadné použití, není potřeba znát podrobněji funkce čipu

   \- Omezené možnosti nastavování

2. Pomocí funkce *generate-program*

   \+ Dává naprostou volnost, co bude v konfiguračním souboru

   \- Dává naprostou volnost, co bude v konfiguračním souboru

Skriptovací jazyk poskytuje 2 základní generátory konfiguračního souboru. Generátor *generate-standard-config* vygeneruje konfigurační soubor, který odpovídá doporučené konfiguraci z tabulky 6. Syntax generátoru vypadá následovně: `(generate-standard-config pixel-count ssid password chipset path),`

kde *pixel-count* je přirozené číslo menší než 2^{16}, *ssid, password* jsou řetězce, určující parametry pro připojení k wifi, *chipset* určuje chipset připojeného pásku (viz tabulka 2) a *path* je řetězec popisující cestu pro uložení souboru. Argumenty generátoru musí být uvedeny všechny a ve stejném pořadí.

Druhým základním generátorem je *generate-minimal-config*. Tento generátor generuje minimální konfiguraci z tabulky 6. Syntax generátoru je podobná tomu předchozímu:

```
(generate-minimal-config pixel-count sequence-number chipset path).
```

Příklady volání těchto generátorů nalezneto v kódu 10.

```
1 ;; standard configuration
2 (generate-standard-config 300 "MujO2Internet" "2312231224ad"
3                     *apa102* "./255.txt")
4
5 ;; minimal configuration
6 (generate-minimal-config 250 1 *ws2801* "./255.txt")
```

Zdrojový kód 10: Příklad volání generátorů

6 Aplikace pro PC

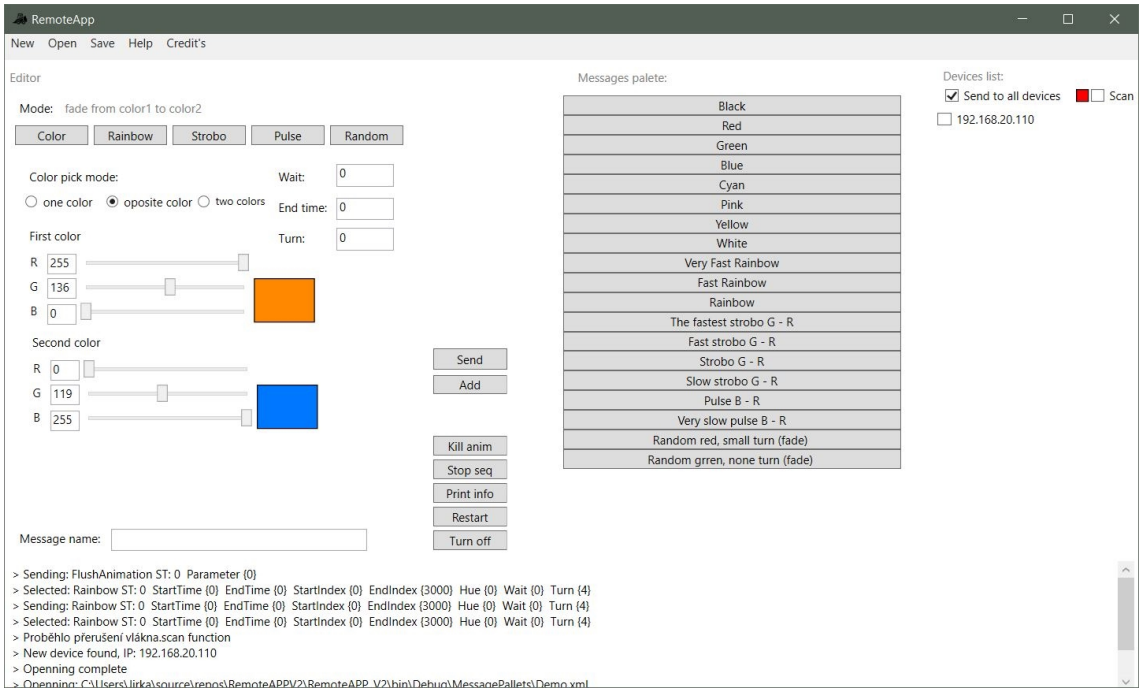
Aplikace *RemoteApp* by měla být jednoduchý ovladač. Původně pracovala jenom se dvěma rgb segmenty a se starší verzí protokolu, který nepodporoval souběh animací, měl méně funkcí a jiné hlavičky. Aplikace je napsána v jazyce C#. Aplikace poskytuje jednoduchý editor zpráv, paletu zpráv a scaner zařízení.

6.1 Uživatelské rozhraní

Uživatelské prostředí se skládá ze čtyř hlavních částí.

1. Editor
2. Paleta
3. Seznam dostupných zařízení
4. Stavový řádek

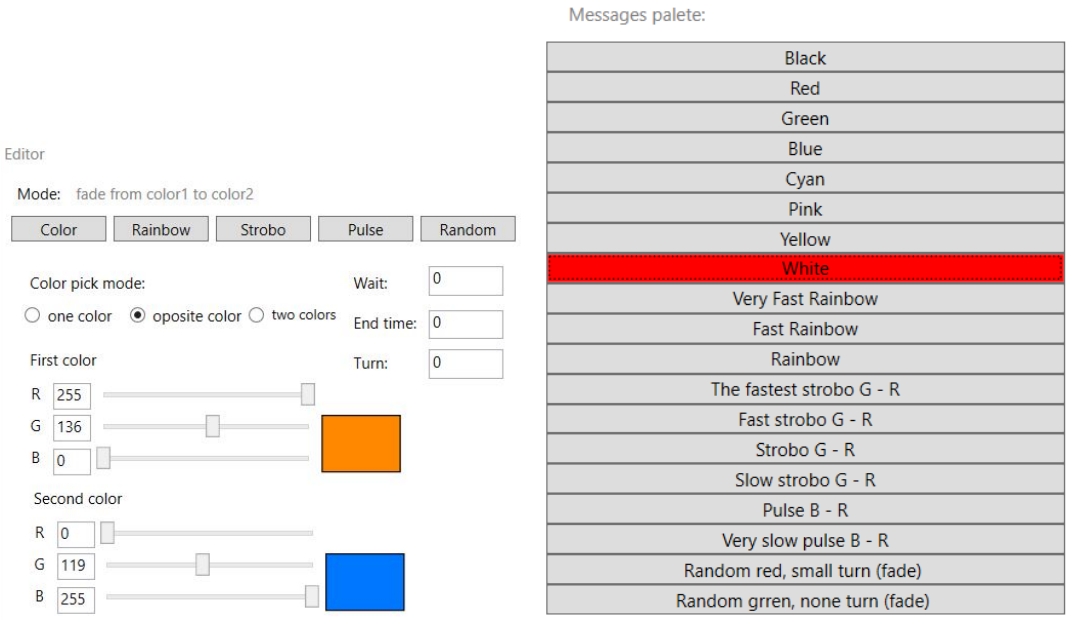
Editor slouží k přiřazování parametrů k funkcím. Tlačítka *Color, Rainbow, Strobo, Pulse, Random* slouží k výběru funkce. v prostoru pod tlačítky se v závislosti na funkci zobrazují dostupné parametry. Tlačítko *Send* pošle zprávu aktuálnímu adresátovi. Tlačítko *add* přidá zprávu do palety. Tlačítka *Kill anim, Stop seq, Print info, Restart, Turn off* okamžitě pošlou zprávu, která odpovídá jejich jménu.



Obrázek 5: Vzhled aplikace

Paleta umožňuje rychle posílat dříve vytvořené zprávy. Aktuálně vybraná zpráva je zvýrazněna červenou barvou. Zprávu z palety je možné poslat dvojklikem nebo zmáčknutím klávesy „*x*“. Označenou zprávu můžeme přejmenovat stisknutím klávesy *F2*. Kombinace kláves *ctrl* a ↑/↓ přesune zprávu v paletě nahoru/dolů. Odstranění zprávy z palety umožňuje klávesa *Delete*.

Aktuální adresát je určen v seznamu dostupných zařízení. Pokud je zaškrnuto políčko *Send to all devices*, zpráva se odešle *broadcastem*. Jinak je zpráva poslána všem zařízením, které jsou zaškrnuty v seznamu pod *Send to all devices*. Zaškrnutí *checkBoxu Scan* zapne scanner, který rozpošle *broadcastem* zprávu *send-info-message*. Všechny zařízení, od kterých následně dostane zprávu, přidá do seznamu dostupných zařízení. Zařízení, které jsou v seznamu z dřívějška a neodpoví do určité doby, jsou označena hvězdičkou. Hvězdičky jsou smazány, začne-li zařízení opět komunikovat nebo dosáhne 4 hvězdiček, tehdy je zařízení smazáno ze seznamu dostupných zařízení.



Obrázek 6: Editor a paleta

Obrázek 7: Seznam dostupných zařízeních

Závěr

Podařilo se mi vytvořit funkční systém pro ovládání LED pásků, který je poměrně snadno upravitelný pro různé světelné instalace. Rovněž jsem se snažil, aby všechny části mé práce byly snadno rozšiřitelné pro nové protokolové funkce. Samotný systém však stále vyžaduje mnoho práce v ladění detailů, čištění principů a rozšiřování funkčnosti.

Protokol bych časem chtěl obohatit o interaktivní funkce, které budou svoji činnost vykonávat v závislosti na hodnotě proměnných. Tyto funkce by bylo možné využít k interaktivnímu zobrazování naměřených hodnot na pásek, například z gyroskopu či zvukového čidla. Dále bych chtěl přidat funkce pracují s náhodou, například „otevři náhodný soubor se sekvencí“ nebo „s určitou pravděpodobností, spusť funkci x“. Tyto funkce se mohou hodit v situacích, kdy je potřeba, aby zařízení nejevilo stopy generičnosti, a samotné psaní negenerických programů pro více různých zařízení by bylo náročné. Posledním typ, který bych chtěl přidat, jsou funkce umožňující komunikaci s ostatními čipy. Tímto rozšířením by bylo možné dosáhnout určité synchronizace mezi zařízeními.

Řídící aplikaci v PC plánuji přepracovat a vytvořit podobným stylem, jakým jsou navrženy aplikace pro řízení světel na stagi. Uživatel definuje scénu jako množinu dvojic *čip - funkce*, kterou pak může hromadně odeslat všem zařízením. Zároveň bych chtěl, aby aplikace umožňovala důkladněji zobrazit informace o čipu. Tohoto lze docílit pomocí funkce *send-info-message*, která zatím pouze ohlačuje připojení čipu k síti.

Conclusions

I managed to create a functional system for controlling the LED strip. The adjustment for different lightning installations is relatively easy in this system. I also tried to have all parts of the work extended to new protocol functions. There is still a lot of work in detail harmonizing, principle cleaning and expansion functionality.

I would like to upgrade the protocol with interactive functions which will do their operations depending on the value of some variables. These features could be used for interactive imaging of measured values for example from gyroscope or acoustic sensor on the strip.

As next step, I want to add features that work with circumstances, e.g. open a random sequence file or run function x with a certain probability. These features may be useful in situations when it is necessary that the device does not show any traces of generality and the non-generic writing for a multi-device program would be difficult.

The last type of features I would like to add are those that allow us to communicate with other chips. This extension could make possible some synchronizations.

I am planning to rework the PC control application and designed it more like applications for stage light control. The user defines the scene as a set of chip-function pairs that can be sent to all devices.

I would like the application to be able to show the chip information more thoroughly. This can be done using the send-info-message function, which only works as a notification of a chip connection to the network.

A Obsah přiloženého CD/DVD

bin/
: Aplikace *RemoteApp* se nachází v podsložce RemoteApp. Spustit aplikaci je možné pomocí souboru *RemoteAPP_V2.exe*. v druhé podsložce je skriptovací jazyk ACIDScript. Pro jeho spuštění je potřeba mít spuštěný REPL Common Lispu a vyhodnotit výraz `(load "cesta k~souboru")`.

doc/
: Složka obsahuje text práce ve formátu PDF (soubor „kidiplom.pdf“ a zdrojový kód textu (archiv „kidiplom.zip“.y

src/
: Kompletní zdrojové texty programů *RemoteApp, ACIDScript* a *ACIDControler* (implementace protoklu pro čip). Knihovna FastLED je zde uvedena pouze pro úplnost. v Arduino IDE, které je ve složce *install/*, už je naimportována.

readme.txt
: Tento soubor obsahuje postup pro bezproblémové spuštění všech částí bakalářké práce.

Navíc CD/DVD obsahuje:

data/
: Obsahuje příklady skriptů (vygenerovaných i zdrojových kódů), které lze nahrát do zařízení.

install/
: Obsahuje instalační soubor Arduino IDE pro Windows.

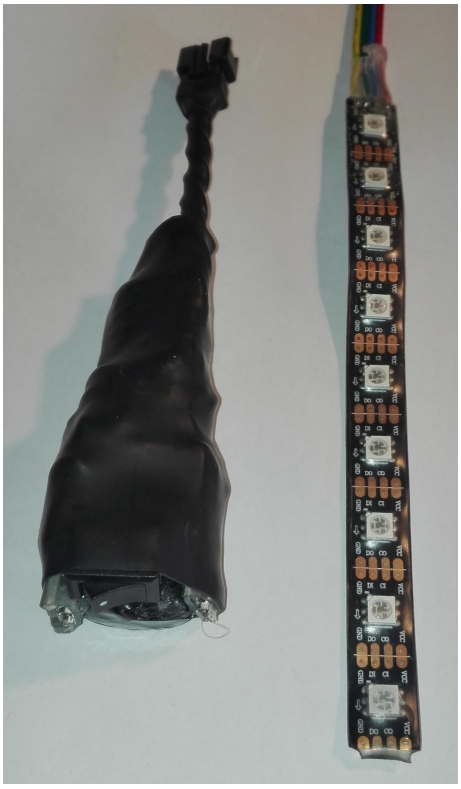
literature/
: Obsahuje odkazy na dokumentaci pro Arduino, ESP8266, knihovnu FastLED a další.

B Stručný popis testovacího modulu

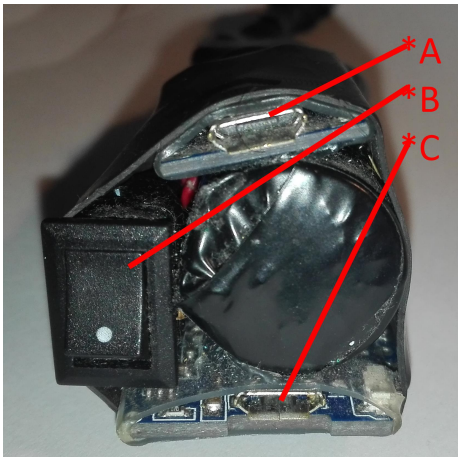
Tato příloha slouží jako návod a popis jednoduchého testovacího modulu, který je zobrazen na obrázku 8.

Modul obsahuje desku Wemos D1 mini, Li-Pol baterii a čip hlídající podvybití a přebití baterie. Adresovatelný pásek se připojuje pomocí čtyř pinového konektoru.

USB konektor, který je na obrázku 9 označen písmenem *A* slouží k nabíjení baterie. Pokud se při nabíjení rozsvítí kontrolní dioda modře, baterie je plně nabita, jinak kontrolka svítí červeně. Pro nahrávání programu a souborů do čipu



Obrázek 8: Testovací modul

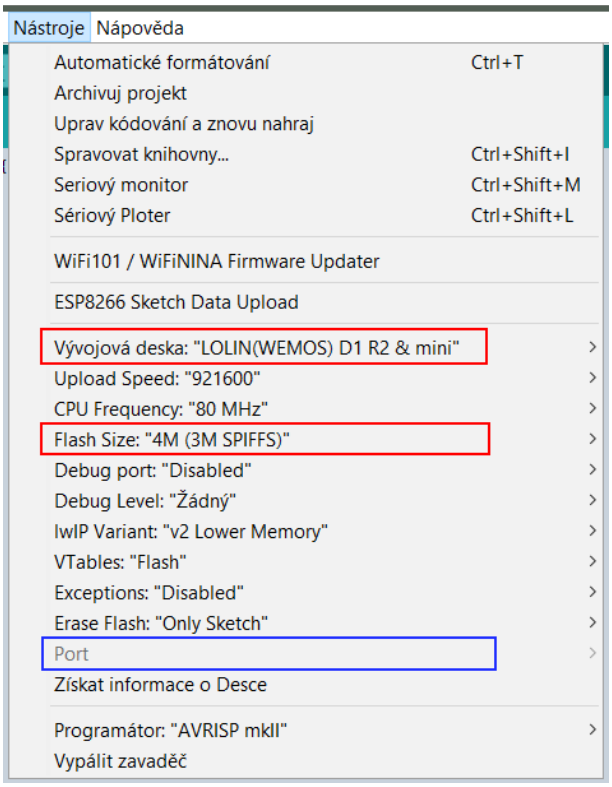


Obrázek 9: Detail modulu

slouží konektor *C*. Pro připojení napájecího či datového USB kabelu se spustí i čip. Spínač *B* slouží k připojení/odpojení baterie.

Nahrát program či soubory do zařízení je možné pomocí Arduino IDE. Instalátor je dostupný na přiloženém DVD nebo na oficiálním webu *arduino.cc* [7]. Postup nahrávání:

1. Nainstalujte Arduino IDE
2. Spusťte Arduino IDE a v záložce „Soubor > Vlastnosti“ a do textového pole „Správce dalších desek URL“ přidejte odkaz (bez uvozovek) „http://arduino.esp8266.com/stable/package_esp8266com_index.json“
3. Přes „Nástroje > Vývojová deska > Manažer Desek“ nainstalujte desku „ESP8266“. Stačí tento název zadat do vyhledovacího pole, a kliknout na tlačítko „install“
4. Nainstalujte Arduino ES8266 filesystem uploader dostupný na adrese esp8266fs
   (a) Nástroj je stažený na přiloženém DVD ve složce „install“ (složka „DVD:\install\ESP8266FS“)



Obrázek 10: Nastavení desky, velikosti SPIFFS a portu

(b) V domovském adresáři pro Arduino IDE vytvořte složku „tools“. Domovskou složku lze zjistit v „Soubor > Vlastnosti > Umístění projektů“

(c) Do vytvořené složky „tools“ zkopírujte složku „ESP8266FS“.

(d) Restartujte Arduino IDE

5. Připojte modul pomocí USB kabelu, vyberte desku a velikost souborového systému jak je znázorněno na obrázku 10 (červeně zvýrazněné pole), nastavte sériový port odpovídající desce (modře zvýrazněné pole).

6. Tlačítkem „Nástroje > ESP8266 Sketch Data Upload“ nahrajete všechny soubory, ze složky „~\domovskáSložkaArduino\složkaAktuálníhoProjektu\data“ do paměti chipu. Samotné nahrávání může chvilku trvat.

Literatura

[1] Oficiální stránky projektu Arduino, dostupné na odkaze: https://www.arduino.cc/

[2] Dokumentace k jazyku Arduina: https://www.arduino.cc/reference/en/

[3] Dokumentace pro ESP 8266: https://arduino-esp8266.readthedocs.io/en/latest/index.html

[4] Oficiální webové stránky knihovny FastLED: http://fastled.io/

[5] Návod pro zapojení a ovládání RGB segmentů: http://www.jerome-bernard.com/blog/2013/01/12/rgb-led-strip-controlled-by-an-arduino/

[6] Stránka na Wikipedii, která vysvětluje pulsně šířkovou modulaci: https://cs.wikipedia.org/wiki/PWM

[7] Odkaz na stažení Arduino IDE: https://www.arduino.cc/en/Main/Software

[8] Návod k instalaci SPIFFS pro ESP8266: https://github.com/esp8266/arduino-esp8266fs-plugin