

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



Výukový tutoriál pro využití Arduina v robotice

Abstrakt

Tento dokument byl vytvořen jako součást bakalářské práce.

Tento tutoriál, zároveň sloužící jako referenční příručka, je určen všem, kteří zajímá robotická platforma Trilobot nebo Arduino obecně. Celý dokument je rozdělen do dvou částí. V první je vysvětlená teorie okolo Arduina: co to je, jak se používá, jak spolu mohou desky komunikovat a podobně. Druhou částí je již samotný tutoriál, rozdělený do částí, kde se každá část zaobírá jedním senzorem robota Trilobot.

Obsah

I Arduino a robotika	4
1 O mikrokontrolerech a Arduinu	5
1.1 Co to jsou mikrokontrolery a kde se používají	5
1.2 Co je to Arduino?	7
1.2.1 Historie Arduina	7
1.3 Základní Arduino desky	8
1.4 Arduino IDE a jazyk Wiring	10
1.5 Komunikace mezi více deskami	11
1.5.1 Seriová linka	12
1.5.2 I ² C sběrnice	13
1.5.3 Wi-Fi	14
1.5.4 Bluetooth	16
1.5.5 Rádiové vlny	17
2 O robotech a robotice	20
2.1 Roboti a robotika na FIT VUT	20
2.1.1 Toad	20
2.1.2 Kvadrokoptéry	20
3 Představení robota Trilobot	22
3.1 Seznam periferií a jejich popis	23
II Tutoriál	28
1 Co v tutoriu najdete	29
2 Prakticky úvod k Arduinu a OOP	30
2.1 Připojení Arduina k PC a základní nastavení	30
2.2 Arduino IDE	30
2.3 Základy programování Arduina a základy OOP	30
2.3.1 Programování Arduina a připojování senzorů	30
3 Displej, reproduktor a I²C aneb Co se může hodit	36
3.1 Displej	36
3.2 Displej 16x2	37
3.3 I ² C sběrnice	38
3.4 Reproduktor	40
4 Zjišťujeme množství světla – fotosenzory	41
4.1 Jak funguje fotosenzor – základy	41

5 Měříme vzdálenost – senzory vzdálenosti HCSR-04, SRF08 a Sharp	44
5.1 Jak se měří vzdálenost?	44
5.2 Měření senzorem HC-SR04	46
5.3 Měření senzorem SRF-08	48
5.4 Měření senzorem Sharp-GP2D120	51
6 Jezdíme – motory a enkodéry	53
6.1 Jedeme rovně	54
6.2 Zatačíme!	56
6.3 Spojujeme znalosti: Jak zastavit před překážkou?	57
7 Orientujeme se v prostoru – kompas, gyroskop a akcelerometr	59
7.1 Co je to IMU?	59
7.2 Tvoříme kompas	59
7.2.1 Kompas 1	60
7.2.2 Kompas 2	61
7.3 Detektor otřesů	63
7.4 Detektor nárazu	64
8 Závěr	66

Část I

Arduino a robotika

Kapitola 1

O mikrokontrolerech a Arduinu

V této kapitole si povíme o mikrokontrolerech, kde se používají, co s tím má společného Arduino a třeba také to, že Arduino není jenom jedno. Také si v rychlosti vysvětlíme vše okolo – jazyk Wiring a dvě vývojová prostředí: Arduino IDE a Visual Studio Code s pluginem PlatformIO.

Dále si řekneme něco o robotech a robotice a představíme si několik robotů, se kterými se můžete potkat na FIT VUT. Poslední část představí pro nás nejdůležitějšího robota, a to sice robota Trilobot, který bude použit i ve zbytku tutoriálu.

1.1 Co to jsou mikrokontrolery a kde se používají

Určitě znáte klasické počítače a notebooky. Každý má samostatný procesor, základní desku, grafickou kartu a hromadu dalších komponent, které lze jednoduše měnit. Jsou to často velmi výkonné stroje, které zvládnou opravdu hodně. Teď si ale představte, že potřebujete třeba jen ukazovat čas na digitálních hodinkách.

Jasně, to zvládne každý počítač, ale nosit s sebou laptop jen kvůli ukazování času je dost nepraktické. Proto existují speciální malinké počítače, kde je na jednom obvodu všechno – procesor, operační paměť, vstupní a výstupní piny... A přesně to je mikrokontroler: malý čip, který funguje jako samostatný malý počítač.

Kde všude se s mikrokontrolerem setkáme? Prakticky kdekoli! Ať už se jedná o chytrou žárovku, která komunikuje přes Wi-Fi, ovladač k televizi nebo fotosenzor nad dveřmi do obchodu, mozkem celého zařízení je malý jednoduchý počítač, který zpracovává signály z okolí a reaguje na ně.

Od normálních počítačů se ale mikrokontrolery i v mnohem liší.

I když (většinou) nejsou mikrokontrolery specializované na konkrétní činnost, po celou dobu jejich životního cyklu na nich běží jedený program, vykonávaný znova a znova v tzv. „hlavní smyčce“. Pro změnu programu se musí programátor napojit na rozhraní (například seriovou linku) a pomocí normálního počítače nahrát do mikrokontroleru nový program.

Mikrokontrolery také nemívají grafický výstup ani žádné uživatelské rozhraní. Posláním mikrokontroleru totiž často není komunikace s uživatelem – tedy s námi – ale pouze vykonávání svého programu stále dokola – například sledovat, zda senzor pohybu nezaznamenal návštěvníka před dveřmi do obchodu a pokud ano, tak otevřít dveře.

Mikrokontrolery mívaly také velmi nízký výkon (a tím pádem i spotřebu). Na jednu stranu to limituje jejich použití – na běžném Arduinu si v žádném případě nepřehrajete video ve FullHD nebo nevytrénujete umělou inteligenci. Na druhou stranu, nízká spotřeba umožňuje mikrokontrolerům běžet na baterii klidně i několik měsíců nebo let. A díky extrémně nízkým nárokům jsou některé mikrokontrolery schopné získávat energii z okolí – rádiových vln, rozdílu teplot a podobně. Této problematice se věnuje tzv. „energy harvesting“.

Mikrokontroler nebo vývojová platforma?

Občas se pojmy zaměňují, ale mikrokontroler je jen ten relativně malíčký černý čip na desce. Arduinu a jemu podobným jako celku (modré destičce) se správně říká vývojový kit/vývojová platforma.

Vývojové platformy kromě samotného mikrokontroleru poskytují i všechny důležité podpůrné obvody jako USB a napájecí porty, ochrany, vyvedená pole pinů, případně i nějaké zabudované senzory a indikátory.

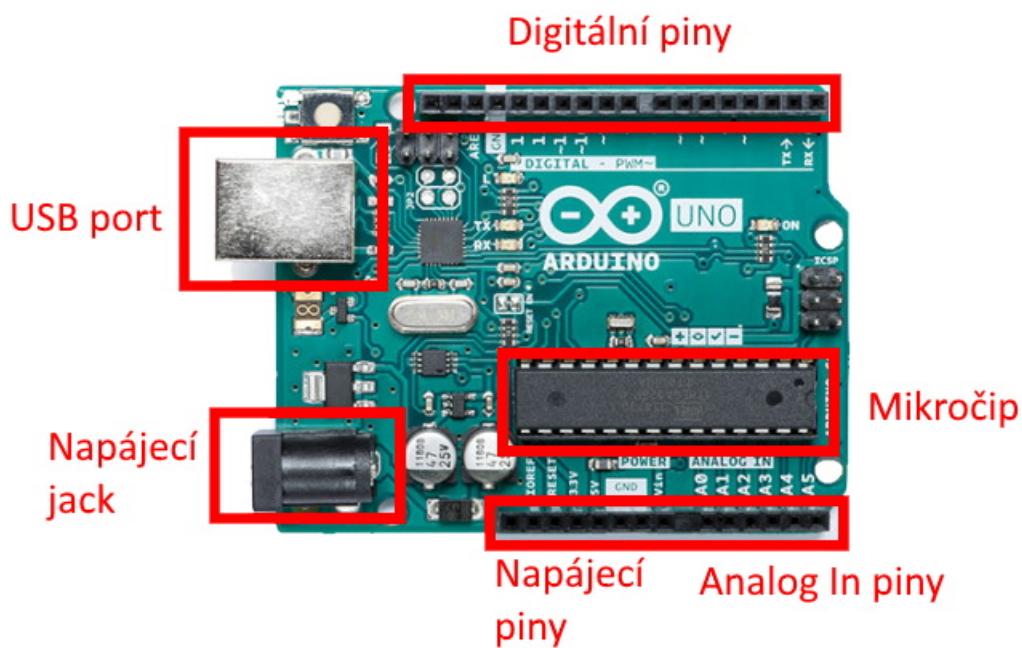


Obrázek 1.1: Ukázka několika vývojových desek s mikrokontrolery. Zleva nahoru: Pine A64+, Raspberry Pi 3B+, Odroid XU4, BeagleBone AI, Arduino Leonardo (klon), Wemos D1 UNO , Arduino UNO (klon), dvě Arduina Nano, Arduino-like klon Pro Micro. Jak je vidět, svět vývojových kitů je opravdu velice rozsáhlý od miniaturních destiček velikosti palce (Pro Micro) až po plnohodnotné počítače, které mohou zastat i funkci slabšího desktopu (Raspberry). A tohle je přitom jen naprostota náhodná a miniaturní sbírka destiček, které autor našel v šuplíku. :)

1.2 Co je to Arduino?

Arduino je podle oficiálních stránek¹ elektronická platforma, která si klade za cíl být v první řadě snadno použitelná. Co to ale vlastně ta elektronická platforma – nebo také hovorově prototypovací destička, případně vývojový kit – je? Jedna se integrovaný spoj, na kterém najeznete všechno potřebné pro tvorbu všech elektronických projektů:

- Mikrokontroler, který se stará o běh programu. Oproti procesoru v počítači je nesrovnatelně méně výkonný, ale také má mnohem menší spotřebu. Jelikož robot nebo třeba chytrý senzor nepotřebuje vysoký výkon (zpravidla nepoužívají GUI ani nevykonávají složité výpočty, maximálně přenášejí data pro tyto výpočty na servery), vůbec nám to nevadí.
- Napájecí vývody
- Paměť, kde je uložen program a data. Ta může být skutečně jen jedna nebo mohou mít data a program dvě oddělené paměti.
- Konektory pro připojení k počítači, nejčastěji USB port a integrovaný převodník.
- GPIO piny, které umožňují k destičce připojit všechno možné, od LED světel přes senzory až po rozšiřující moduly.
- A mnoho dalších, i když třeba ne tak potřebných a používaných komponent.



Obrázek 1.2: Popis základních částí Arduina

1.2.1 Historie Arduina

A teď trochu do historie. První deska typu Arduino vznikla v roce 2005 s cílem pomoci studentům a těm, kteří nikdy předtím neprogramovali zapojovat vlastní obvody a vytvářet vlastní elektronická zařízení. Jinak řečeno, cílem Arduina bylo přivést co nejvíce lidí k elektronice a – lidově řečeno – bastlení.

¹<https://www.arduino.cc/en/Main/AboutUs>

Díky tomu, že je celý projekt Arduino open-source a open-hardware, vytvořila se kolem něj mohutná komunita, která se stará například o vytváření knihoven, tvorbu podpůrných materiálů (knih, dokumentace, tutoriálů) a podobně. Open-hardware povaha také umožnila mnoha lidem vytvářet si vlastní klony Arduino desek. Toho využily hlavně čínské e-shopy, které zaplavily trh mnoha levnými klony desek Arduino².

Celá platforma Arduino se skládá z několika součástí: samotných vývojových desek, jazyka Wiring a vývojového prostředí Arduino IDE. O všech si něco řekneme dále.

1.3 Základní Arduino desky

Za zhruba 15 let existence platformy Arduino bylo vytvořeno mnoho různých desek, z nichž je ovšem zdaleka nejpoužívanější deska s názvem Arduino UNO. Kromě ní ovšem existují i verze Mega (v současné době konkrétně Mega 2560), jejímž hlavním rozdílem je větší počet GPIO pinů, nebo desky Nano a Micro, kde byla hlavním kritériem velikost. Kromě těch výše zmíněných existují i exotičtější desky, které se již tak nepoužívají, například Yún s mnohem výkonnějším čipem, díky němuž může na desce fungovat i odlehčená linuxová distribuce.

Nejběžnější desky (UNO, Mega, Nano, Micro) používají procesory od firmy Atmel, konkrétně typy ATmega328(P), ATmega32U4 nebo ATmega2560, všechny pracující na frekvenci 16 MHz. Paměť programu a dat je oddělená. Velikost pamětí se liší podle konkrétní desky.

Arduino UNO

Arduino UNO je první a dosud i zdaleka nejpoužívanější deskou platformy Arduino. V současné době se již prodává deska ve své třetí revizi (značeno „R3“ za jménem). Deska používá procesor ATmega328. Její hlavní předností je kompromis mezi velikostí, snadným přístupem ke GPIO pinům a cenou. Pro Arduino UNO je také koncipována většina tzv „shieldů“, rozšiřujících desek, které se nasadí na všechny piny a dodávají desce dodatečnou funkcionalitu, například Bluetooth konektivitu.

Programování a napájení probíhá přes vestavěný USB-B port, přičemž napájení může být řešeno i pomocí vestavěného konektoru nebo přímo přes adekvátní piny Vcc a GND.

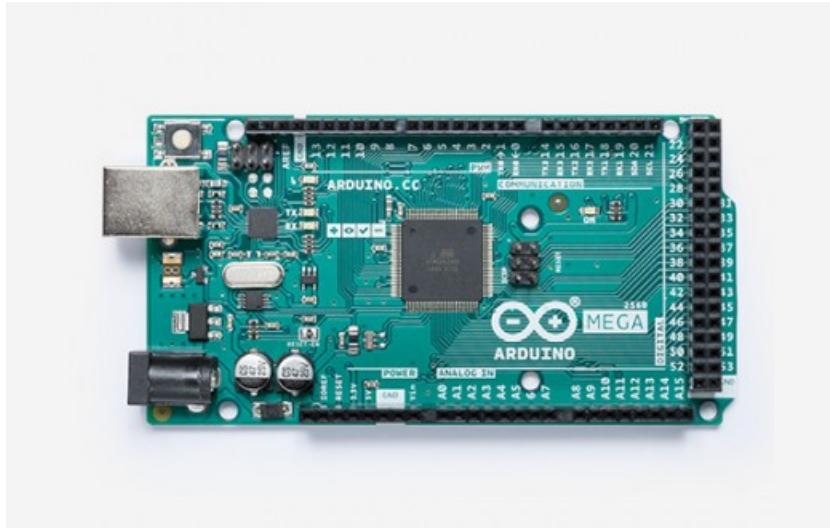


Obrázek 1.3: Deska Arduino UNO Rev3 z oficiálního obchodu [?].

²O klonech se více informací můžete dočít třeba zde: <https://blog.arduino.cc/2013/07/10/send-in-the-clones/>

Arduino Mega

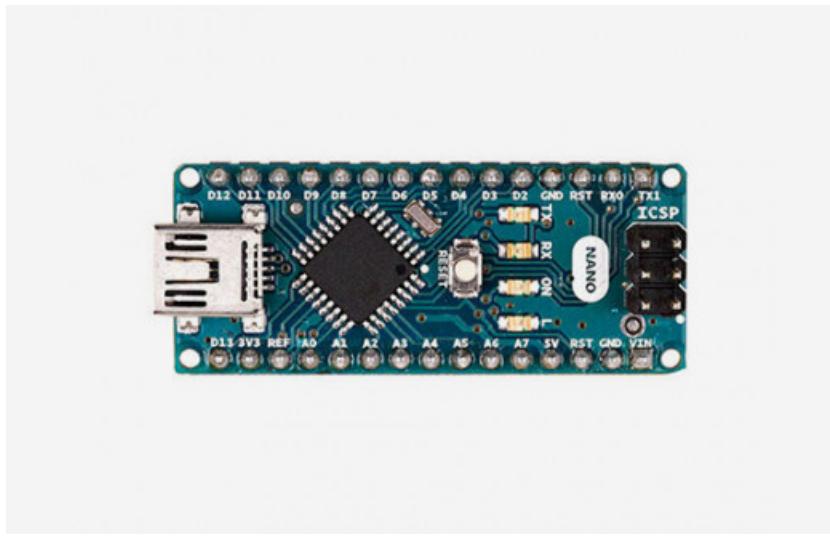
Arduino Mega je rozšířenou deskou Arduino uno s větší pamětí a větším počtem pinů, což je jeho hlavní výhodou.



Obrázek 1.4: Deska Arduino Mega Rev3 z oficiálního obchodu [?].

Arduino Nano

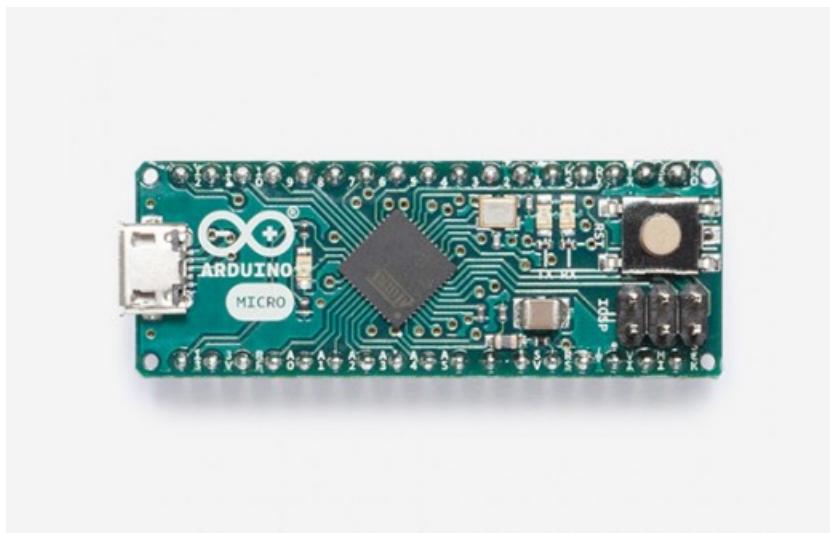
Arduino Nano Má opět prakticky totožnou funkcionalitu s deskou Arduino UNO, ale vydalo se opačným směrem než Arduino Mega: Nano je vytvořeno s cílem být co nejmenší při zachování kompatibility s nepájivými poli. Programování probíhá přes mini-USB oproti klasickému USB-B portu.



Obrázek 1.5: Deska Arduino Nano z oficiálního obchodu [?].

Arduino Micro

Srtejně jako Arduino Nano je Arduino Micro zástupcem miniaturních desek určených spíše pro trvalé zabudování do projektu, kde záleží na velikosti (IoT zařízení a podobně). Hlavním rozdílem je ovšem použitý procesor: ATmega32U4, který má zabudovaný USB převodník. Výhodou je především to, že Arduino Micro se může chovat jako standardní vstupní periferie jako myš nebo klávesnice a lze si takové zařízení naprogramovat.



Obrázek 1.6: Deska Arduino Micro z oficiálního obchodu [?].

1.4 Arduino IDE a jazyk Wiring

Jak již bylo zmíněno, Arduino je názvem projektu a zároveň platformy, sdružující pod sebou jak hardwarové prostředky, tak vlastní jazyk a vývojové prostředí. V následující části bude stručně popsán jazyk Wiring, Arduino IDE a princip programování desek Arduino.

Jazyk Wiring

Pro ujasnění terminologie je třeba uvést, že samotný jazyk Wiring byl vytvořen v roce 2003 jako diplomová práce Hernanda Barragána. Později, v roce 2005, Massimo Banzi³ naklonoval repozitář jazyka Wiring a spolu se svým týmem ho začal vyvíjet pod platformou Arduino. Samostatný jazyk Wiring sice stále existuje, ovšem je na okraji zájmu. Pojem „jazyk Wiring“, tedy odkazuje na klon původního jazyka Wiring pod platformou Arduino a v tomto smyslu zde také bude používán.

Jazyk Wiring měl za cíl při svém vytvoření zjednodušit práci umělcům a designerům, kteří používají elektroniku, kteří se do té doby museli učit relativně složité prototypovací jazyky, které v té době byly k dispozici. V současné době je používán platformou Arduino, kde díky své jednoduchosti slouží jako rychlý prototypovací a výukový jazyk. Jazyk Wiring je založený na jazyce C++⁴. Označení „jazyk“ ve spojitosti s Wiringem může být diskutabilní, po technické stránce má spíše povahu rozsáhlé knihovny nebo frameworku.

Základní filozofií tohoto jazyka je zapouzdřit složitou interakci s mikrokontrolery a nahradit je jednoduše použitelnými funkcemi. Například pro nastavení pinu jako výstupního není třeba nastavovat žádné speciální registry, postačí zavolat funkci `pinMode()`, která vše provede.

Základní struktura programu pro Arduino v jazyce Wiring se dělí do dvou hlavních funkcí: `setup()` a `loop()`.

³Massimo Banzi byl dle Hernanda Barragána vedoucím jeho diplomové práce.

⁴Při psaní programů pro Arduino lze využívat všechny dostupné konstrukce tohoto jazyka.

Ve funkci `setup()` se nachází ta část kódu, kterou je nutné provést pouze jednou. Většinou se tedy jedná o nastavení pinů (zda–li budou dané piny použité jako vstupní nebo výstupní), inicializaci připojených komponent a podobně. Funkce `loop()`, jak již název napovídá, vykonává hlavní smyčku programu, tedy například odečet dat ze senzorů a podobně.

The screenshot shows the Arduino IDE interface with the following details:

- Title Bar:** "Blink | Arduino 1.8.12 (Windows Store 1.8.33.0)"
- Menu Bar:** File, Edit, Sketch, Tools, Help
- Toolbar:** Includes icons for Save, Undo, Redo, Open, and Print.
- Sketch Name:** "Blink §"
- Code Area:** Displays the "Blink" sketch code in C++/Arduino syntax.

```
// the setup function runs once when you press reset or power the board
void setup() {
    // initialize digital pin LED_BUILTIN as an output.
    pinMode(LED_BUILTIN, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
    digitalWrite(LED_BUILTIN, HIGH);      // turn the LED on (HIGH is the voltage level)
    delay(1000);                      // wait for a second
    digitalWrite(LED_BUILTIN, LOW);     // turn the LED off by making the voltage LOW
    delay(1000);                      // wait for a second
}
```
- Status Bar:** "Arduino Mega or Mega 2560, ATmega2560 (Mega 2560) on COM6"

Obrázek 1.7: Ukázka kódu v jazyce Wiring.

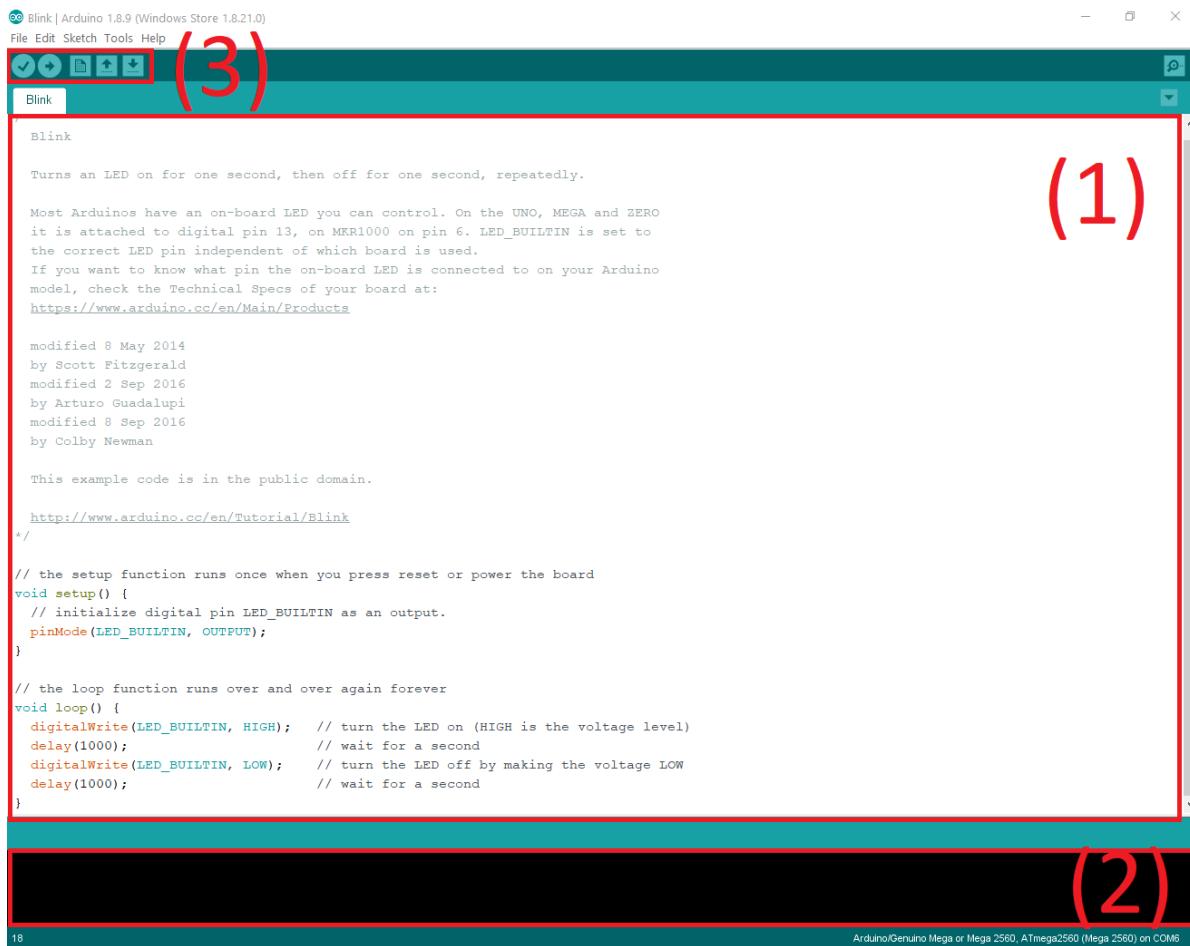
Arduino IDE a programování desek Arduino

Jak již bylo zmíněno, oficiálním vývojovým prostředím pro platformu Arduino je Arduino IDE, velmi jednoduché vývojové prostředí, které je ve své podstatě jen textový editor s funkcí programování Arduina a zvýrazňování syntaxe. Arduino IDE nepodporuje například našepťávání nebo ladící funkce. Problémem je i práce ve více souborech. Arduino IDE sice umožňuje otevřít více souborů naráz, ale není možné jednoduše přecházet mezi soubory, případně zjistit, ve kterém souboru je implementovaná daná funkce.

1.5 Komunikace mezi více deskami

Arduino velmi často najdete jako srdce jednoduchých samostatných projektů, které se svým okolím nijak komunikovat nemusí. Dříve nebo později se ovšem dostanete do situace, kdy spolu bude muset komunikovat například Arduino a počítač, dvě Arduina nebo Arduino a nějaký minipočítač – správce chytré domácnosti a podobné.

I když tento tutoriál není zaměřený na komunikaci, tak ta k robotům neodmyslitelně patří a bylo by trestuhodné, kdybyste zde nenašli alespoň základní přehled, který Arduino na poli komunikace nabízí. Proto zde ukážu několik možností, které může Arduino využít (některé z nich během tutoriálu využijeme, některé ne).



Obrázek 1.8: Na obrázku 2.1 je vidět hlavní obrazovka vývojového prostředí. Kromě hlavní části, do které se píše samotný kód (1), se ve spodní části nachází místo pro výpis ladicích informací při nahrávání (2). Vlevo nahoře pod standardní navigační lištou se nachází (zleva doprava) tlačítka pro překlad, nahrání programu, vytvoření nového souboru, otevření stávajícího souboru a uložení (3). O průběhu překladu a nahrávání je uživatel informován ve spodní části obrazovky. Připojení desky Arduino a její nastavení se provádí nejprve fyzickým připojením a poté přes nabídku „Tools“, kde se vyberou správné položky v nabídce „Board“, „Processor“ a „Port“. První dvě položky jsou dány deskou, která je aktuálně používaná. U originálních desek Arduino se port rozpozná jednoduše: v závorce za názvem portu je napsán typ desky, která je na daném portu připojená. U neoriginálních desek je třeba metody pokus–omyl.

1.5.1 Seriová linka

Komunikace přes seriovou linku se ve světě Arduina používá asi nejčastěji, jelikož jejím prostřednictvím nahráváte programy do svého Arduina a také komunikujete s počítačem, pokud si chcete vypsat nějaké informace do Seriového monitoru v Arduino IDE. Komunikace probíhá prostřednictvím dvou vodičů – Rx (receive – přijímání) a Tx (transmit – odesílání), které stačí pouze propojit (Rx zařízení A na Tx zařízení B a obráceně – to, co jedno zařízení odesílá, chce to druhé přijímat).

Jediné, na co si je třeba dávat pozor, je tzv *baudrate* neboli přenosová rychlosť, která musí být na obou zařízeních nastavena stejně. Nejčastěji se používá 9600 baudů – pomalá, ale spolehlivá rychlosť, kterou jsou schopna komunikovat prakticky všechna zařízení a používáme ji i my v tutoriálu. Pokud se ale Seriovou linkou přenáší větší kusy dat, používá se i 115200 baud. Arduino zvládá i mnoho dalších rychlosťí, ale tyto dvě jsou zdaleka nejběžnější.

Samotná komunikace v kódu vypadá velmi jednoduše:

```

/* v setup() */
/* Zahajeni komunikace ryhclosti 9600 baud */
Serial.begin(9600);
/* v loop() */
/* Vypsani dat do seriove linky - napriklad pri pouziti se Serial Monitor v Arduino IDE */
Serial.print(something);
Serial.println(something); /* navic ukonci radek */

/* Cteni probiha podobne jako u IIC. Pokud jsou k dispozici data... */
if(Serial.available() > 0)
{
    /*... precti je.*/
    byte data = Serial.read(); /* cteni po bajtech */
}

/* Pomoci tohoto prikazu zapiseme jeden bajt dat.
 * Vhodne, pokud pres Seriovou linku napriklad noco ridime. */
Serial.write(data);

```

1.5.2 I²C sběrnice

Seriová linka sloužila primárně ke komunikaci mezi dvěma zařízeními. I²C to zvládá až do počtu 128 zařízení. Kromě toho je zde ještě jeden rozdíl – u seriové komunikace byla obě zařízení rovnocenná. I²C je tzv „Master–Slave“ sběrnice. To jednoduše znamená, že na sběrnici je jedno zařízení, které řídí komunikaci, získává a zasílá data na všechna ostatní zařízení – Master. Všechna ostatní zařízení jsou Slave – nemohou začít komunikaci bez vyzvání a komunikovat mohou pouze se zařízením Master.

Samotná konstrukce sběrnice je neuvěřitelně jednoduchá: její součástí jsou pouze vodiče SCL (synchro-nizace) a SDA (data). Takže pomocí pouhých dvou vodičů můžeme propojit prakticky celý projekt (pokud to samozřejmě senzory dovolují). I²C se samozřejmě dá použít i na komunikaci mezi více deskami, ale příliš často se s tím nesetkáme.

Praktická ukázka je k vidění níže:

```

#include <Wire.h> /* Knihovna pro praci s IIC*/
... /* v setup() se vola jen Wire.begin()*/
/*Funkce aktivuje IIC sbernicu - volana v setup()*/
Wire.begin();
... /*loop()*/
/*Zapis na zarizeni s adresou <address> do registru <reg>*/
/*Zahajeni komunikace*/
Wire.beginTransmission(<address>);
/*Specifikujeme registr pro zapis*/
Wire.write(<reg>);
/*Posleme data (typu byte)*/
Wire.write(data);
/*Ukoncime komunikaci*/
Wire.endTransmission();

/*Cteni x bajtu ze zarizeni <address> od registru <reg>*/
Wire.beginTransmission(SRF08_ADDRESS);
/*Zapiseme, od ktereho registru chceme cist*/
Wire.write(<reg>);
Wire.endTransmission();

```

```

/* Od zařízení s adresou <address> */
Wire.requestFrom(<address>, x);
/* Dokud data nejsou připravena, čekáme */
while(!Wire.available());
/* Bez něj ovšem data nezahodíme :). Wire.read() se tu bude
 * volat ve smyčce xkrát*/
Wire.read();

```

1.5.3 Wi-Fi

V poslední době je Wi-Fi asi nejčastějším způsobem, jak spolu dvě zařízení komunikují. Velkou výhodou je, že zařízení není omezené jen na dosah své antény, ale pomocí internetu může komunikovat s druhým zařízením – nebo třeba vaším mobilem – na druhé straně světa. Nevýhodou je poněkud vyšší spotřeba v porovnání třeba s Bluetooth nebo rádiem.

V poslední době se také objevuje otázka bezpečnosti. Například v článku na mobilmania.cz⁵ si můžete přečíst o tom, že i pomocí obyčejné chytré žárovky napojené na internet je možné napadnout celou chytrou domácnost. Všechno má příčinu v tom, že malé mikrokontrolery nemají dostatečný výpočetní výkon pro použití vysoko zabezpečeného přenosu a proto raději používají méně zabezpečené způsoby.

Arduino samo o sobě Wi-Fi ani jinou internetovou konektivitu nemá a v zásadě se ani nepočítalo s tím, že by Arduino potřebovalo po internetu komunikovat. Postupem času ovšem vznikl Wi-Fi shield – rozšiřující deska, která se nacvakne na Arduino UNO a umožní mu po Wi-Fi komunikovat.

Na druhou stranu, ve sféře Wi-Fi „bastlení“ a prototypování je Arduino až druhým hráčem. Mnohem více se v této oblasti prosadily mikročipy ESP a zařízení postavená na nich. Arduino bylo čipy ESP převálcování na všech frontách – cenou, konektivitou (na čipech ESP může bez problémů běžet Arduino kód a nový ESP čip má v sobě rovnou i Bluetooth modul), výkonem i podporou. To všechno vedlo k tomu, že výroba Wi-Fi shieldu, jediné oficiální možnosti rozšíření Arduina o Wi-Fi, byla zastavena.

Na druhou stranu, existuje samostatná deska: Arduino UNO WiFi. Tato deska obsahuje zbrusu nový čip ATmega4809 a podporuje WiFi konektivitu již v základu. Na druhou stranu je tato deska relativně drahá a nepříliš běžná.

Základy komunikace přes WiFi se liší podle použité desky, proto zde ukážu základy komunikace pomocí nejčastěji používaného řešení – ESP32 a ESP3266. Nejprve je nutné přidat podporu ESP desek do Arduino IDE:

- Klikneme na *File->Preferences*
- Do řádku *Additional Boards Manager URLs* přidáme https://arduino.esp8266.com/stable/package_esp8266com_index.json a klikneme na OK
- Nyní přejdeme do *Tools -> Board -> Boards Manager*
- Vyhledáme **ESP8266** a nainstalujeme jedinou položku, kterou nám *Board Manager* vyhledá

Nyní je již možné programovat ESP čipy.

Jelikož WiFi toho umožňuje opravdu hodně, není cílem tohoto tutoriálu nijak detailně popisovat kompletní princip. V ukázce najdete jednoduchý příklad – vytvoření lokálního serveru v lokální síti a reakci na stisk tlačítka – po stisknutí tlačítka se rozsvítí nebo zhasne LED na pinu D1.

Princip je extrémně jednoduchý. Pokud klient vstoupí na stránku *adresa/ON*, LED se rozsvítí, při *adresa/OFF* zhasne.

```

#include <WiFi.h> /* Knihovna pro práci s WiFi přes ESP*/
/* Název site, do které se budeme připojovat/

```

⁵<https://www.mobilmania.cz/clanky/ani-chytre-zarovsky-nejsou-v-bezpeci-byly-zneuzity-k-napadeni-pocitacovych-sit-sc-3-a-1347432/default.aspx>

```
const char* ssid = "Nazev site";
/* Heslo do site*/
const char* password = "heslo";
/* Nas server */
WiFiServer server(80);

/* setup()*/
Serial.begin(9600);
/* Vypneme diodu*/
pinMode(D1, OUTPUT);
digitalWrite(D1, LOW);
/* Pripojeni k WiFi */
WiFi.begin(ssid, password);
/* Spusteni serveru */
server.begin();
/* Vypis IP adresy serveru */
Serial.println(WiFi.localIP());

/* loop() */
/* Cekani na klienta, ktery se pripoji k serveru */
WiFiClient client = server.available();
if (!client)
    return;

/* Cekame, dokud klient neposle nejaka data */
while (!client.available())
    delay(1);

/* Precteni prvniho radku URL pozadavku*/
String request = client.readStringUntil('\r');
/* Zahod vse ostatni */
client.flush();

/* Pokud se v URL vyskytuje ON, zapni LED, pokud OFF, vypni ji*/
if(request.indexOf("ON") > -1)
    digitalWrite(D1, HIGH);
if(request.indexOf("OFF") > -1)
    digitalWrite(D1, LOW);

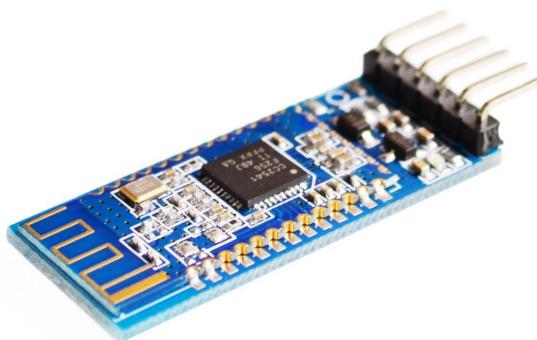
/* Odeslani stranky, ktera se zobrazí klientovi */
client.println("HTTP/1.1 200 OK");
client.println("Content-Type: text/html");
client.println("");
client.println("<!DOCTYPE html>");
client.println("<html>");
client.println("<body>");
client.print("<h1>Ukazka ESP</h1>");
client.println("<a href=\"/ON\"><button>ON</button></a>");
// Tlacitko pro odpojeni rele vede na adresu /OFF
client.println("<a href=\"/OFF\"><button>OFF</button></a>");
client.println("</body>");
client.println("</html>");

delay(1);
```

1.5.4 Bluetooth

I když je v poslední době na ústupu, Bluetooth je stále velmi využívanou technologií, zvláště mezi mikrokontrolery a IoT. Zároveň umožňuje snadnou komunikaci i s mobilními telefony, smartphony a laptopy, u kterých je Bluetooth stále prakticky nedílnou součástí výbavy.

Pro Arduino existuje množství Bluetooth modulů, jako třeba HC-05 a HC-06 pro Bluetooth v2.0 nebo HM-10, podporující BLE (Bluetooth Low Energy) a Bluetooth 4.0. Jelikož jsou ceny minimálně na čínských eshopech srovnatelné, většina z vás sáhne po modernější verzi HM-10,



Obrázek 1.9: Bluetooth modul HM-10. Obrázek z <https://navody.arduino-shop.cz/navody-k-produktum/arduino-bluetooth-4.0-ble-modul-hm-10.html>

Princip komunikace je velmi snadný. Modul HM-10 používá ke komunikaci Seriovou linku, se kterou jsme se již setkali. Zde je ovšem nutné zmínit, že Arduino UNO má k dispozici pouze jednu Seriovou linku, kterou většinou komunikuje s počítačem, je proto nutné využít knihovny `SoftwareSerial.h`. Arduino Mega má více Seriových linek, proto tento problém řešit nemusíme:

```
/* Ukazka funguje jen pro Arduino Mega,
 * Pro Arduino UNO nutno pouzit SoftwareSerial.h*/
/* v setup()*/
Serial2.begin(9600);

/*v loop()/
/* Pokud jsou dostupne nejake bajty...*/
if(Serial2.available() > 0)
{
    /*... precti je*/
    byte data = Serial2.read();
}

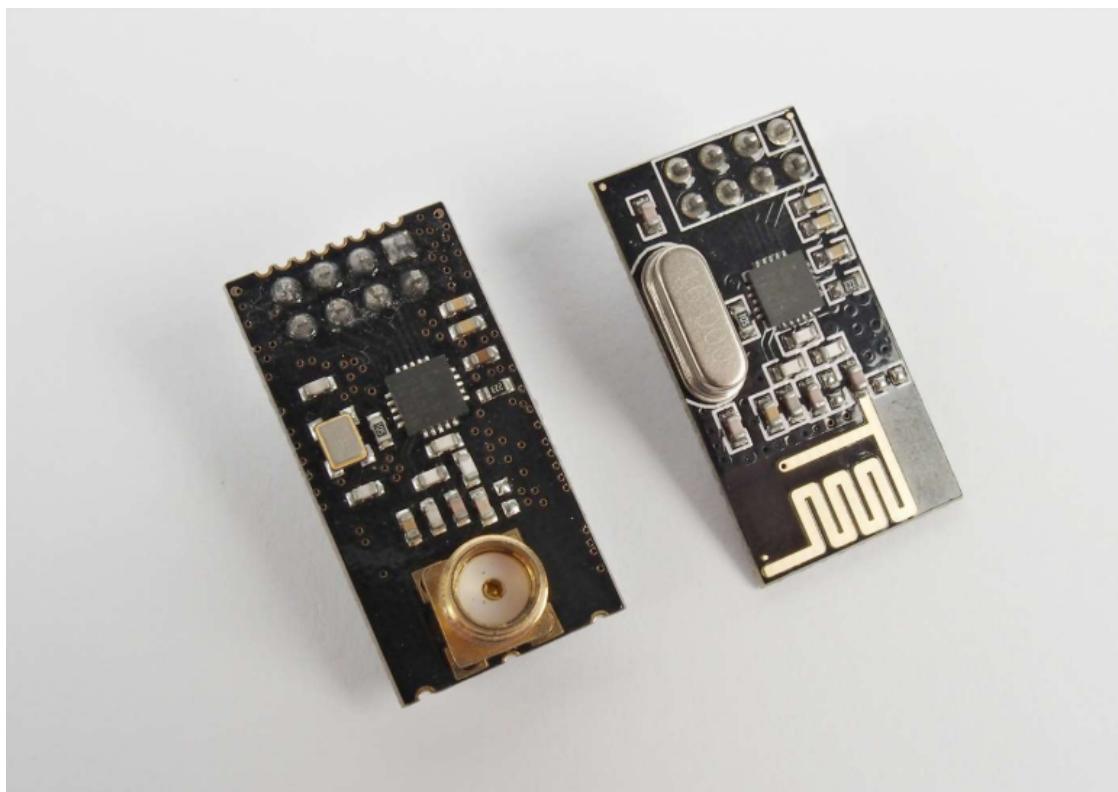
/* Odesli jeden bajt dat*/
Serial2.write(data);
```

1.5.5 Rádiové vlny

I když se o tomhle způsobu komunikace v souvislosti s Arduinem a vůbec IoT moc nemluví, přesto stojí za zmínku. Rádiové čipy jsou prakticky výlučně založené na součástkách od Nordic Semiconductors a jejích čipech nRF. Stejně čipy používají i bezdrátové periferie Microsoftu nebo třeba Logitechu.

Pro osobní použití se nejlépe hodí čipy nRF24L01/nRF24L01+, které sice sám Nordic Semiconductors nepodporuje, ale komunita kolem nich vytvořila velmi kvalitní knihovny. Velkou výhodou je také naprostě minimální spotřeba (zhruba 15 mA při aktivním používání oproti 20 mA u HM-10 a zhruba 100 mA u WiFi) v kombinaci s velmi slušným dosahem – pokud má obvod dodatečnou anténu, je teoretický dosah až jeden kilometr! V reálném světě se ovšem dostaneme o řad níže, maximálně na nižší stovky metrů⁶.

Velkou výhodou těchto čipů je také cena, která je v přepočtu na jeden modul nejnižší ze všech tří technologií. Pokud potřebujeme komunikovat jen např. v rámci pokoje nebo bytu, můžeme použít modul s integrovanou anténou, kde sice dosah dramaticky klesne maximálně na desítky metrů, ale samotný modul bude menší než poštovní známka.



Obrázek 1.10: Modul nRF24L01. Obrázek z <https://www.zive.cz/clanky/pojdme-programovat-elektroniku-radiovy-cip-ktery-ma-skoro-kazda-bezdratova-mys/sc-3-a-201436/default.aspx>.

Programování nRF24L01 není nic složitého. Komunikace probíhá po sběrnici SPI, která zde sice doposud zmíněná nebyla, ale to nevadí, jelikož nRF24 využívá vlastní knihovnu, kterou je ovšem třeba importovat:

- Přejdeme do *Sketch-> Library Manager*
- Vyhledáme „RF24“ a mezi několika dalšími najdeme stejnojmennou knihovnu, kterou nainstalujeme

Samotné základní již poněkud složitější, ale jsou tu vysvětleny. Pokud vám není něco jasné, pravděpodobně stačí jen zkopirovat kód – zadané hodnoty jsou nastavené pro co nejlepší výsledek i za cenu vysoké

⁶Je to kvůli zahlcení 2.4GHz pásmá mnoha dalšími radiovými zařízeními.

spotřeby – ale nejlepší bude trochu prohledat internet. Knihovna RF24 pracuje objektově, proto bude náš rádiový modul zastupovat objekt třídy RF24.

```
/* Nutne knihovny */
#include <SPI.h>
#include <RF24.h>
/*****************/
*****Nastaveni vysilace*****
/*****************/

/* Inicializace objektu. Na piny 9 a 10 jsou pripojeny vyyvody CE a CSN - ridici piny sbernice SPI */
RF24 radio(9,10);
/* Adresa zarizeni - libovolnych 5 znaku nebo cisel*/
uint8_t address[5] = {0, 1, 2, 3, 4};

/* setup() */
radio.begin(); /* start vysilace */
radio.setChannel(1); /* kanal v rozsahu 0-125*/
/* rychlost komunikace 2 MB/s, je tu moznost spolehlivejsi
 * ale pomalejsich, ale spolehlivejsich rychlosti pomoc
 * RF24_1MBPS a RF24_250KBPS*/
radio.setDataRate(RF24_2MBPS);
/* Dosah na maximum */
radio.setPALevel(RF24_PA_MAX);
/* Za jak dlouho se ma vysilac pokusit odeslat zpravu znova,
 * pokud neuspeje (0 = 25 ms, 1= 50 ms, 2 = 75 ms) a kolikrat
 * se o to ma pokusit */
radio.setRetries(2,15);
/* Delka kontrolniho souctu */
radio.setCRCLength(RF24_CRC_8);
/* Nastaveni komunikace pro zapis s danou adresou */
radio.openWritingPipe(address);
/* Pri jednom vysilani se odesle 32 B. Pri delsi zprave nutno rozdelit */
radio.setPayloadSize(32);

/* loop () */
/* Probuzeni radia*/
radio.powerUp();
/* Priprava zpravy*/
char zprava[] = "Trilobot";
/* Odeslani*/
radio.write(zprava, strlen(zprava));
/* Uspani radia*/
radio.powerDown();
```

```
/* Nutne knihovny */
#include <SPI.h>
#include <RF24.h>
/*****************/
*****Nastaveni prijemace*****
/*****************/

/* Inicializace objektu. Na piny 9 a 10 jsou pripojeny vyyvody CE a CSN - ridici piny sbernice SPI */
RF24 radio(9,10);
```

```
/* Adresa zarizeni - libovolnych 5 znaku nebo cisel*/
uint8_t address[5] = {0, 1, 2, 3, 4};

/* setup() */
radio.begin(); /* start vysilace */
radio.setChannel(1); /* kanal v rozsahu 0-125*/
/* rychlost komunikace 2 MB/s, je tu moznost spolehlivejsi
 * ale pomalejsich, ale spolehlivejsich rychlosti pomoc
 * RF24_1MBPS a RF24_250KBPS*/
radio.setDataRate(RF24_2MBPS);
/* Dosah na maximum */
radio.setPALevel(RF24_PA_MAX);
/* Za jak dlouho se ma vysilac pokusit odeslat zpravu znova,
 * pokud neuspeje (0 = 25 ms, 1= 50 ms, 2 = 75 ms) a kolikrat
 * se o to ma pokusit */
radio.setRetries(2,15);
/* Delka kontrolniho souctu */
radio.setCRCLength(RF24_CRC_8);
/* Nastaveni komunikace pro cteni od vysilace s danou adresou. * Prvni cislo je index prijemce od 0 do 5 a u
radio.openWritingPipe(1, adresa);
/* Zahajeni naslouchani */
radio.startListening();

/* loop () */
while(radio.available())
{
    /* Priprava pole data, kam ulozime prijata data*/
    int len = radio.getPayloadSize();
    byte data[len];
    /* Precteni tolika bajtu, kolik se jich vejde od pole data
     * a jejich ulozeni tamtez */
    radio.read(&data, sizeof(data));
}
```

Kapitola 2

O robotech a robotice

Některí z vás možná vědí, že slovo robot vymysleli bratři Čapkovi kolem roku 1920. Přesněji, vymyslel ho Josef a Karel ho použil ve své hře R.U.R. Co ale vlastně toto slovo znamená?

Žádná exaktní definice neexistuje, ale obecně je robotem něco, co mechanicky interaguje s okolím a samostatně se rozhoduje. „Rozhodováním“ samozřejmě není míněno, že robot bude přemýšlet, co si dá k obědu. Mnohem blíže pravdě bude rozhodování například na vstupu z nějakého senzoru.

Roboty a roboti – životná koncovka často značí to, že robot alespoň vzdáleně připomíná něco živého – mají celou řadu využití. Od průmyslových robotů, kteří montují auta, přes zdravotnictví, kde díky robotům může lékař v Americe operovat pacienta v Evropě, přes armádu, logistiku, výuku až po malé roboty na hraní, jako třeba autíčko na ovládání.

Všemi těmito roboty se zabývá robotika, která zahrnuje vše od návrhu robota, přes jeho tvorbu, programování, až po jeho nasazení. Kombinuje v sobě mnoho oborů, jako mechaniku, senzoriku, elektrotechniku, ale i umělou inteligenci nebo počítačové vidění.

2.1 Roboti a robotika na FIT VUT

Kromě jiného probíhá na FIT VUT i výzkum a vývoj v oblasti robotiky a robotů a kromě jiného zde funguje i celá výzkumná skupina ROBO@FIT¹, která se v poslední době zaměřuje hlavně na výzkum interakce robotů s člověkem, využití rozšířené reality a podobně.

Kromě výzkumné skupiny je na FIT VUT i několik robotů, vzešlých z výzkumné činnosti. Níže si několik z nich ukážeme. Občas jsou některé roboty k vidění na akcích jako je Den otevřených dveří.

2.1.1 Toad

Venkovní robot vybavený 3D lidarem, určený pro mapování a lokalizaci ve velkém měřítku, průzkum nebo třeba mapování terénu.

2.1.2 Kvadrooptéry

Na FIT se nachází i několik kvadrooptér, které jsou určené pro průzkum, mapování a mnoho dalšího.

Tohle byl jen velmi stručný přehled, robotů je na FIT mnohem více, ovšem pro základní ukázku stačí. Záměrně je ovšem vynechaný jeden robot, který vás bude provázet i celým tutorialem. Pro jeho představení je vyhrazena celá následující kapitola.

¹<https://www.fit.vut.cz/research/group/robo/.cs>



Obrázek 2.1: Robot Toad. Obrázek převzat z učebních materiálů k předmětu ROBa²



Obrázek 2.2: Jedna z několika kavdrokoptér. Obrázek převzat z učebních materiálů k předmětu ROBa³

Kapitola 3

Představení robota Trilobot

Robot Trilobot původně vyráběla firma Arrick robotics z USA¹, odkud také bylo před zhruba dvaceti lety několik kusů dovezeno na tehdy zbrusu novou fakultu informačních technologií. Původním zaměřením Trilobota bylo použití v akademickém vývoji, případně ve výuce. Předpokládalo se, že bude použit i s vlastními senzory a proto je celé šasi navrženo jako modulární a upravitelné.

Současná verze trilobota má s jeho původní verzí společný již jen podvozek a modulární kostra. Jádrem současného Trilobota je kombinace desek Arduino Mega a Odroid XU4. Jednotlivé komponenty budou podrobněji popsány dále.



Obrázek 3.1: Původní podoba robota Trilobot, obrázek převzat z <https://www.arrickrobotics.com/trilobot/>

¹<https://www.arrickrobotics.com/trilobot/>

3.1 Seznam periferií a jejich popis

V této části budou krátce popsány jednotlivé komponenty robota Trilobot, jejich účel, možnosti a případně odkaz na podrobnější dokumentaci.

Také zde zmíním, jak lze jednotlivé komponenty ovládat pomocí vestavěných metod nad jejich objekty.

SRF-08

SRF-08 je ultrazvukový měřič vzdálenosti kombinovaný se světelným senzorem. Komunikuje prostřednictvím I²C sběrnice. Podrobnější informace o jeho použití lze nalézt v dokumentaci výrobce².

K ovládání slouží objekt `srf08` třídy `SRF08`. Umožňuje změřit hladinu světla a vzdálenost od objektu, na obě tyto vlastnosti jsou vytvořené i kapitoly v tutoriálu.



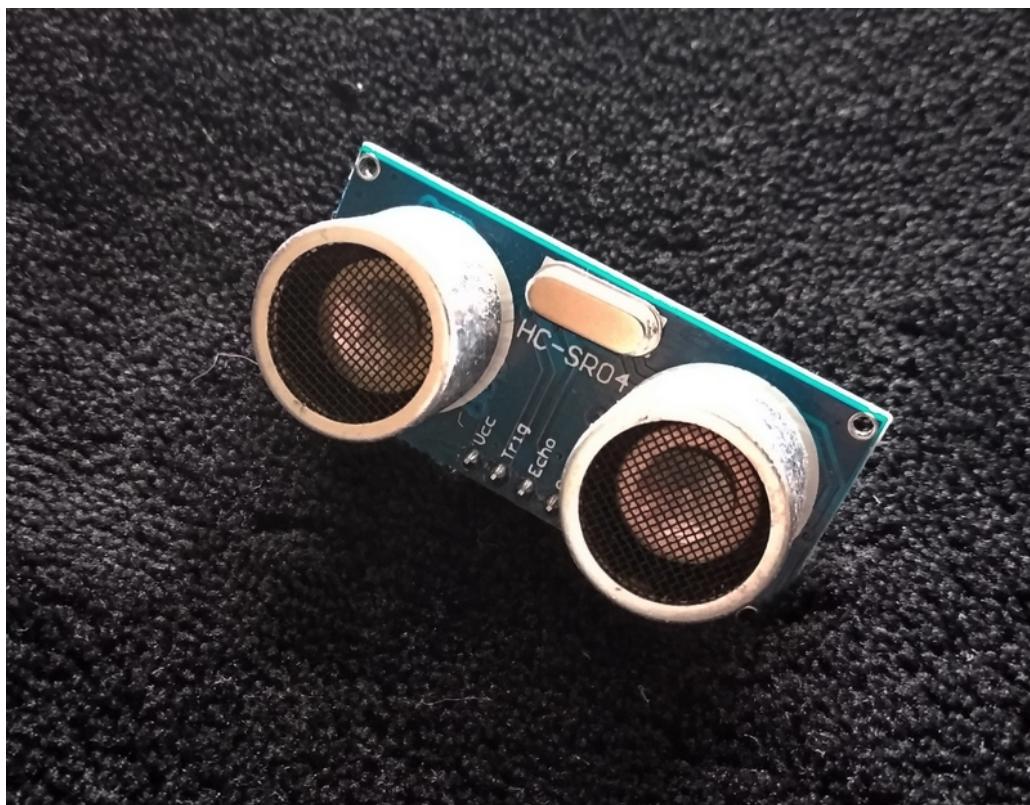
Obrázek 3.2: Senzor SRF-08

HC-SR04

HC-SR04 je stejně jako předchozí senzor ultrazvukový měřič vzdálenosti. Na rozdíl od SRF-08 ale nekomunikuje po žádné sběrnici ale aktivuje se přímo aktivačním vodičem. Naměřená hodnota v milisekundách je odečítána také na starně Arduina jako délka impulzu na jiném vodiči. Jeho hlavní předností je mnohem nižší cena. Na druhou stranu ale není tak přesný, Výrobce Sparkfun k němu dodává i dokumentaci: <https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf>

K ovládání slouží objekt `hcsr04` třídy `HCSR04`. Umožňuje měřit vzdálenost v centimetrech nebo vracet čas, za který se zvuková vlna vrátila. Na měření vzdálenosti v centimetrech je k dispozici kapitola tutoriálu.

²<https://www.robot-electronics.co.uk/htm/srf08tech.html>



Obrázek 3.3: Senzor HC-SR04

Sharp 1994 (GP2Y0A41SK0F)

Poslední ze skupiny senzorů měřících vzdálenost. Od předchozích dvou se liší hned několika věcmi. Především pracuje na principu odrazu infračerveného světla, ne zvuku. Vzdálenost se poté nepočítá jako délka od vyslání k přijetí vlny (tentot čas by byl ve většině případů extrémně krátký a pravděpodobně neměřitelný s naším vybavením), ale k jeho výpočtu se používá hodnota napětí na datovém vodiči. Zajímavé (ale u senzorů naprostě běžné) je i to, že závislost, s jakou naměřené napětí závisí na vzdálenosti, není lineární funkce a na straně mikrokontroleru je třeba poměrně složitého výpočtu.

K ovládání slouží objekt `sharp` třídy Sharp. Umožňuje měřit vzdálenost v centimetrech. Na tuto vlastnost také existuje kapitola tutoriálu.

I²C displej 16x2

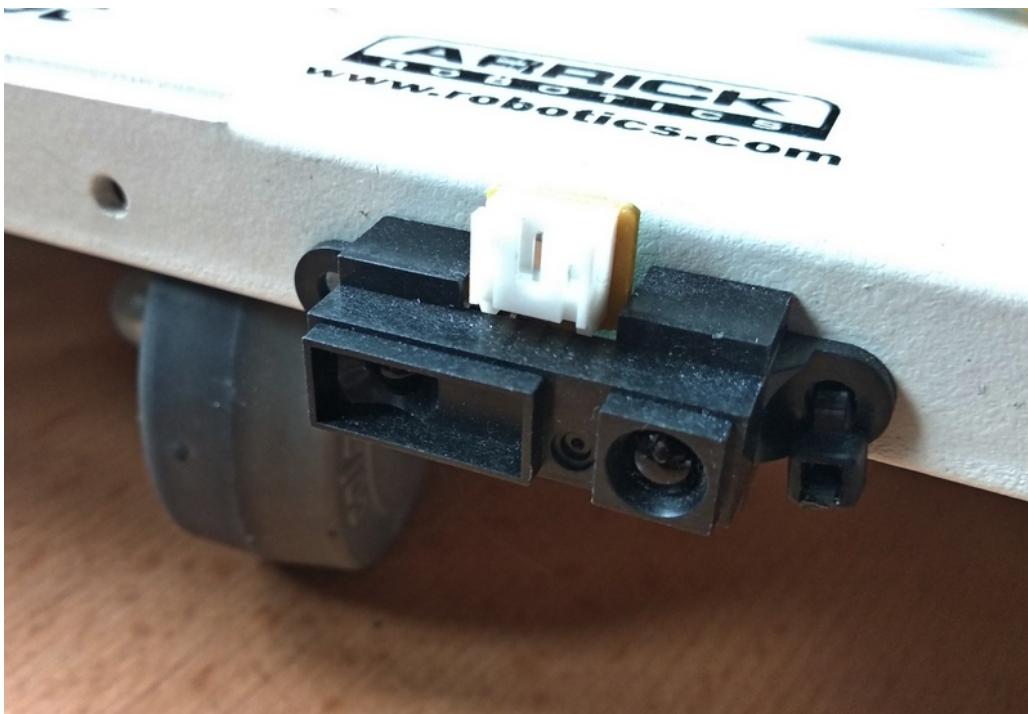
Jedná se o standardní displej 16x2, tedy dvouřádkový se 16 znaky na každém řádku. Pro ušetření vodičů je k němu připájené rozhraní pro komunikaci přes I²C sběrnici. Slouží pro zobrazování naměřených údajů a případně pro podrobnější zpětnou vazbu.

K ovládání slouží objekt `display` třídy Display. Umožňuje pouze tisknout celý jeden řádek displeje nebo kompletně smazat celý displej. Na displej není k dispozici tutoriál, ale pouze vysvětlení, aby bylo možné displej během tutoriálu používat.

Motory Faulhaber 16002, řídící deska Sabertooth 2x5

Pohonná soustava robota Trilobot se skládá ze dvou motorů Faulhaber 16002 a řídící desky Sabertooth 2x5.

Deska Sabertooth je schopná pracovat v mnoha různých režimech, pro robota Trilobot jsou implementovány dva: analogový vstup (s filtrovaným PWM) a zjednodušené ovládání po sériové lince („Analog Input“ a „Simplified Serial“).



Obrázek 3.4: Senzor Sharp 1994 (GP2Y0A41SK0F)

Motory Faulhaber poskytují i zpětnou vazbu v podobě enkodérů, podle kterých se dají určit jejich otáčky. Toho je využíváno při způsobu řízení pomocí analogového vstupu.

K ovládání slouží objekt `motors` třídy `Motors`. Motory umožňují pohyb rovně nebo zatáčení, vždy lze nastavit, zda se pohyb zastaví po určité vzdálenosti, úhlu, nebo zda bude pohyb zastaven explicitně pomocí příkazu `zastavení`.

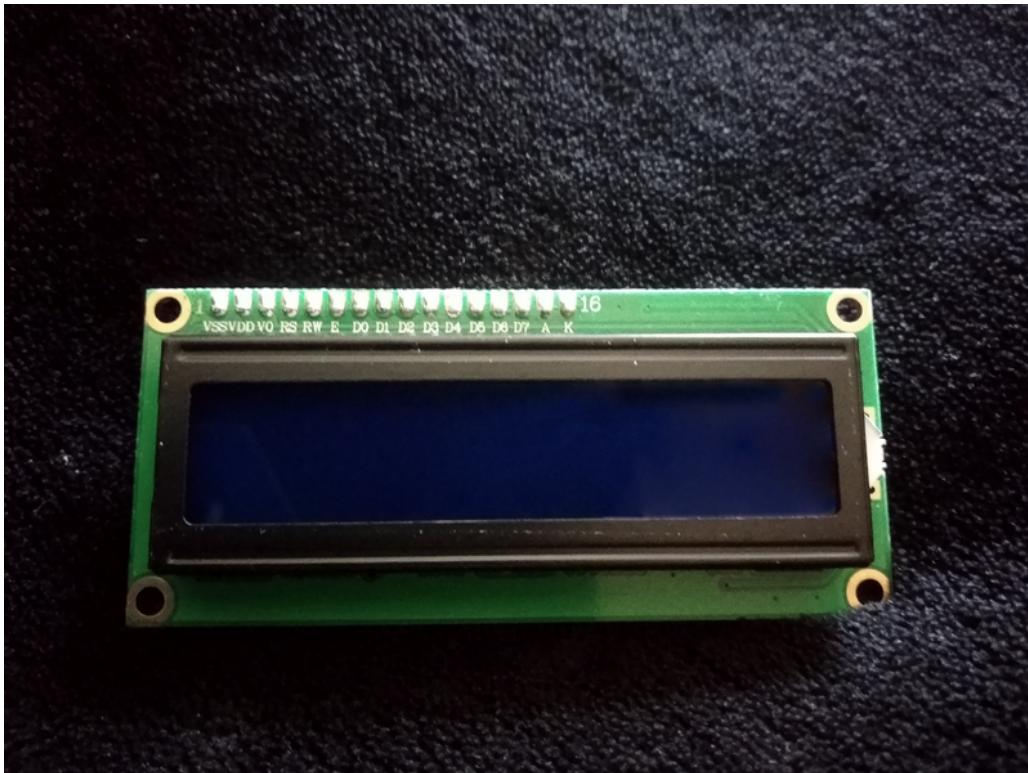
Reprodukтор

Jedná se o obyčejný piezo reproduktor o neznámém výkonu. Primárním účelem je akustická zpětná vazba některých akcí, je ale také plně programovatelný a lze na něm zahrát libovolnou melodii.

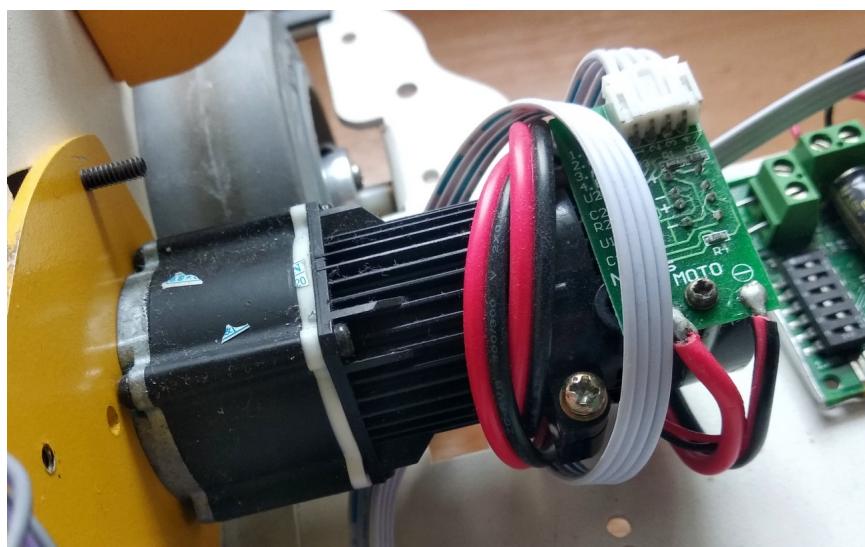
K ovládání slouží objekt `speaker` třídy `Speaker`. Umožňuje hrát konkrétní noty (zhruba oktávu na obě strany od komorního „a“) pomocí funkce `beep()`.

minIMU-v3

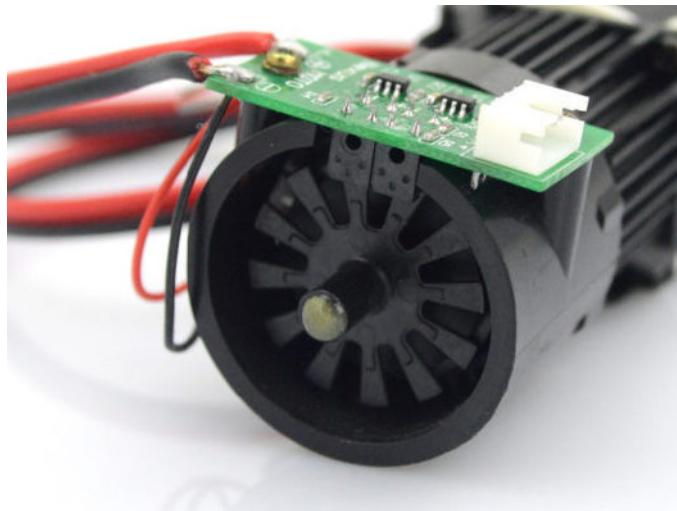
Tento senzor v sobě skrývá akcelerometr, gyroskop a magnetometr. S Arduinem komunikuje přes sběrnici I²C. Jeho uchycení je vyřešeno „věžičkou“ s montáží v horní části Trilobota. Toto řešení je zvolené z toho důvodu, že magnetometr, jedna ze součástí senzoru minIMU-v3, je rušen tělem Trilobota a pokud by byl přimontován blíže, rušení by bylo příliš velké a nešlo by kompenzovat.



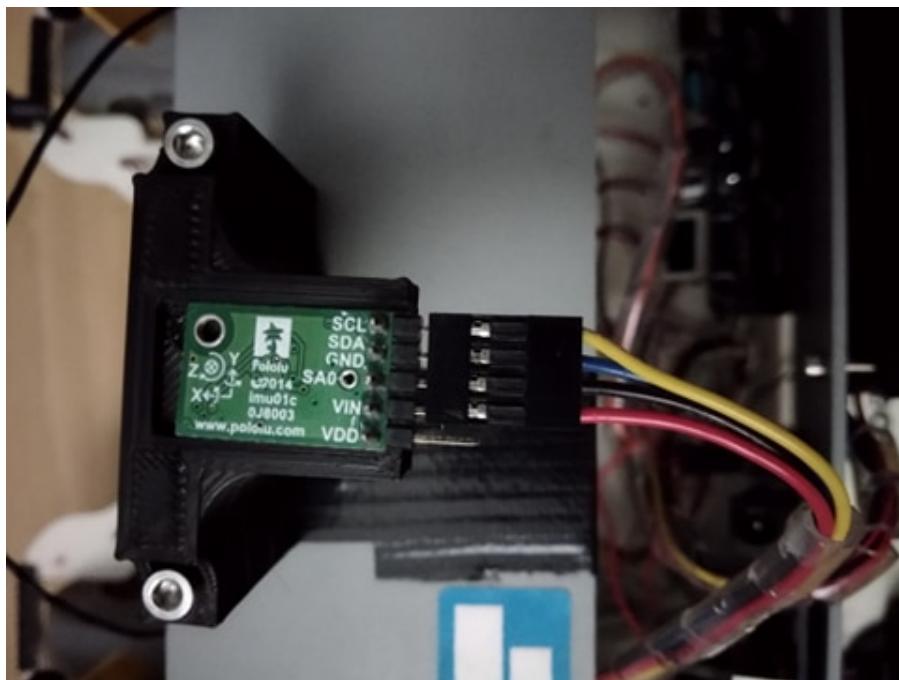
Obrázek 3.5: I²C displej 16x2



Obrázek 3.6: Motor Faulhaber



Obrázek 3.7: Enkodér motoru Faulhaber



Obrázek 3.8: minIMU-v3

Část II

Tutoriál

Kapitola 1

Co v tutoriálu najdete

Nyní se už dostáváme k samotnému tutoriálu. Tutoriál je rozdělený na několik částí, které se mohou vzájemně libovolně kombinovat a pokud už nějakou kapitolu ovládáte, není problém ji přeskočit, každá kapitola je co nejvíce nezávislá.

V tutoriálu si ukážeme:

- Jak zapojit Arduino do počítače a zdroji energie
- Jak programovat (nejen) Arduino
- Jak připojovat senzory
- Programování jednotlivých komponent Trilobota:
 - Fotosenzor
 - Senzory vzdálenosti
 - Motory
 - Displej
 - IMU (polohový senzor)

Kapitola 2

Prakticky úvod k Arduinu a OOP

2.1 Připojení Arduina k PC a základní nastavení

Fyzické připojení Arduina k počítači a jeho nastavení v Arduino IDE není nic složitého, postačí se držet několika jednoduchých kroků:

- Propojíme Arduino a PC vhodným kabelem (stejným se připojují například tiskárny).
- Otevřeme Arduino IDE
- Vybereme správný port přes „Tools->Ports“. Pokud je na výběr několik portů a nevíte, který je správný, nejlepší je najet myší na nabídku portů a Arduino odpojit. Ten, který zmizí, bude ten správný. U originálních desek se také u COM portu objeví i název desky, což usnadňuje výběr.

2.2 Arduino IDE

Arduino IDE jsme si teoreticky popsali v teoretickém představení Arduina, zde se na něj podíváme trochu více prakticky. Na obrázku 2.1 vidíme základní obrazovku Arduino IDE a popisky nejdůležitějších komponent.

Níže si ukážeme, jak nahrát zkušební program do našeho Arduina:

- Nejprve otevřeme zkušební program Blink v nabídce *File -> Examples -> Basics -> Blink*
- Poté klikneme na zelenou šipku doprava vlevo nahoře: tím nahrajeme kód dó Arduina
- Pokud je vše v pořádku, Na Arduinu by měla začít blikat vestavěná LED. U Trilobota začne blikat i LED vedle displeje.

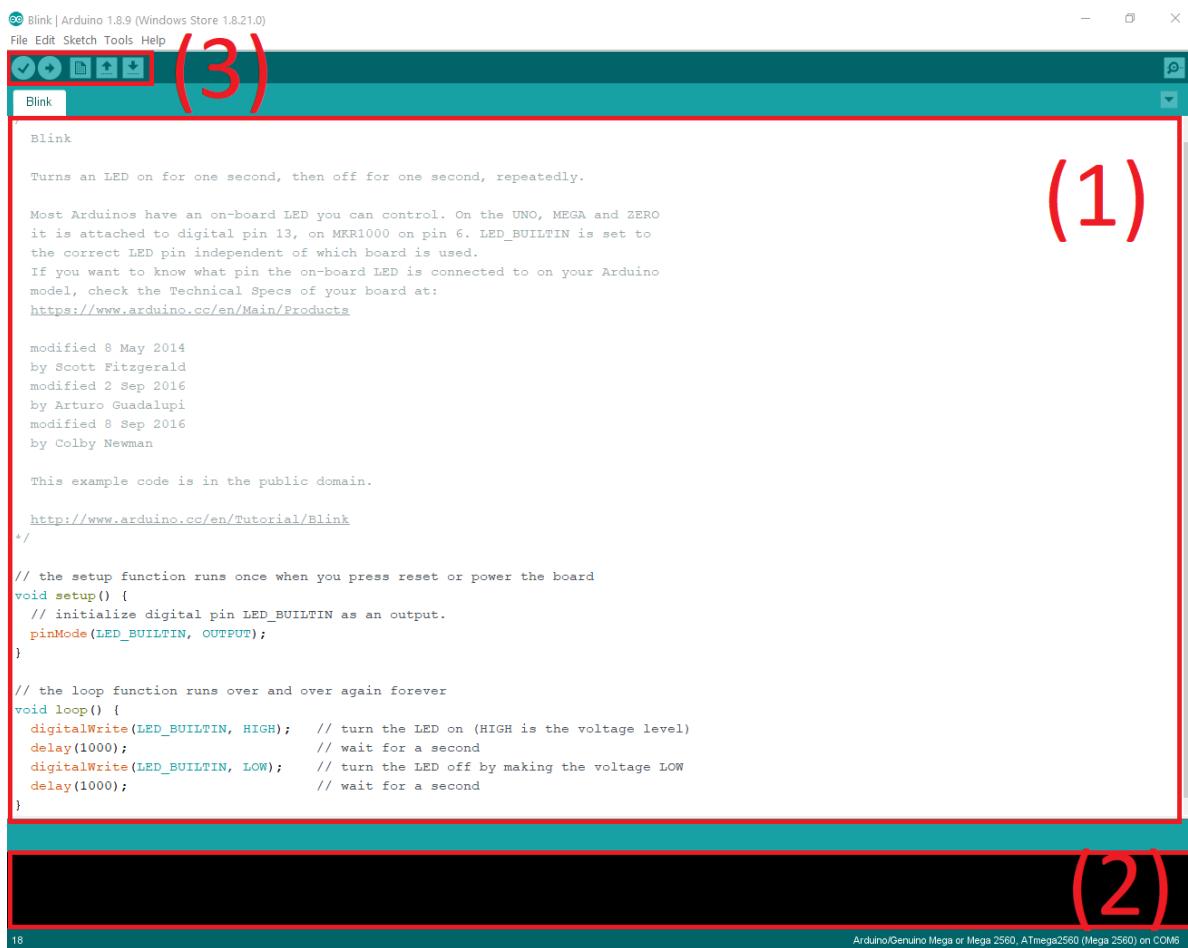
2.3 Základy programování Arduina a základy OOP

V této části si představíme, jak se programuje Arduino a co je to objektově orientované programování. Cílem není poskytnout plnohodnotný tutoriál na OOP, ale pouze nastínit základy a vysvětlit koncept všem, kteří již někdy programovali, ale s OOP se dosud nesetkali.

2.3.1 Programování Arduina a připojování senzorů

Jak již bylo řečeno, Arduino se programuje pomocí jazyka Wiring, což je vlastně obří knihovna pro C++ a je tedy s C++ plně kompatibilní.

Samotný program pro Arduino (nazývaný *sketch*, se skládá z několika částí, které jsou většinou typické i pro běžný program, ovšem je zde i pár rozdílů:

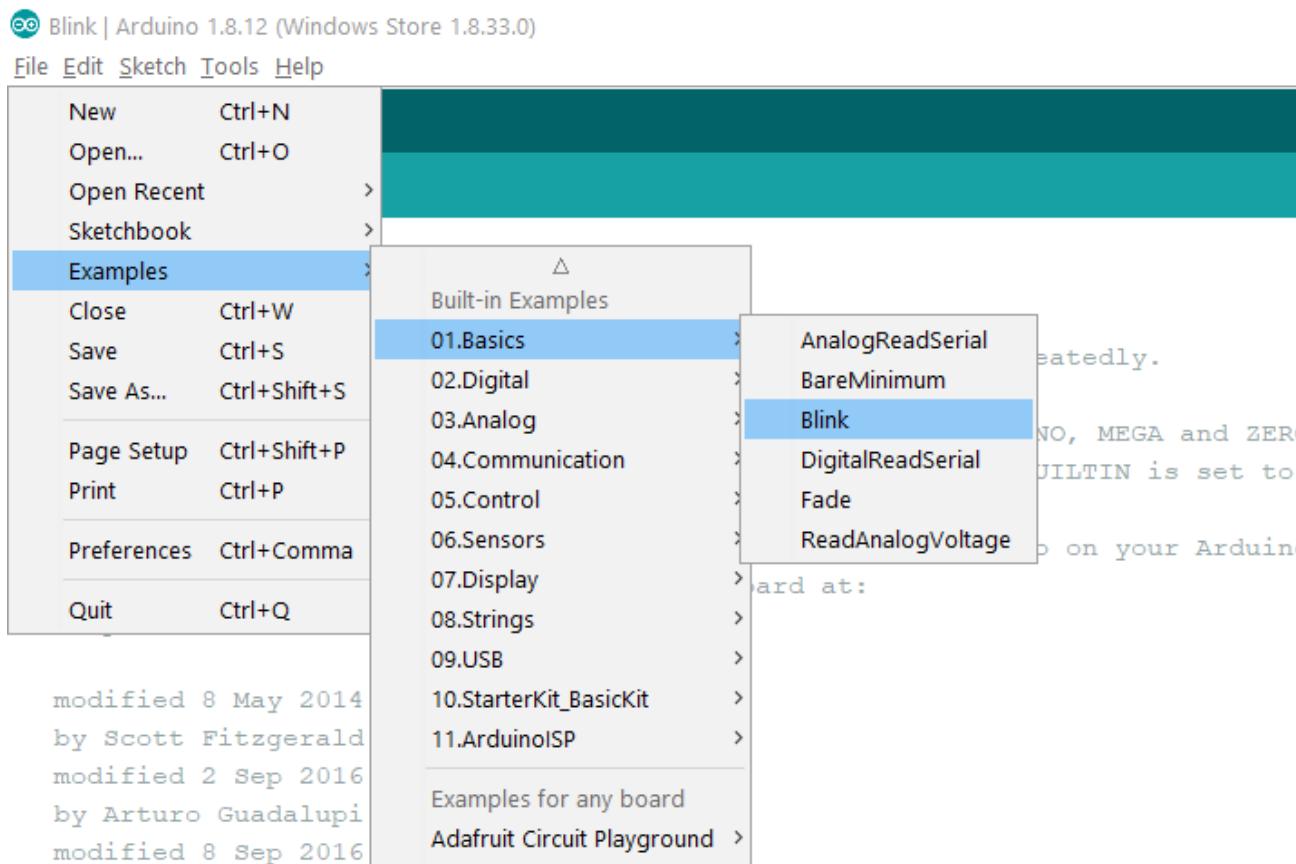


Obrázek 2.1: Na obrázku 2.1 je vidět hlavní obrazovka vývojového prostředí. Kromě hlavní části, do které se píše samotný kód (1), se ve spodní části nachází místo pro výpis ladicích informací při nahrávání (2). Vlevo nahoru pod standardní navigační lištou se nachází (zleva doprava) tlačítka pro překlad, nahrání programu, vytvoření nového souboru, otevření stávajícího souboru a uložení (3). O průběhu překladu a nahrávání je uživatel informován ve spodní části obrazovky. Připojení desky Arduino a její nastavení se provádí nejprve fyzickým připojením a poté přes nabídku „Tools“, kde se vyberou správné položky v nabídce „Board“, „Processor“ a „Port“. První dvě položky jsou dány deskou, která je aktuálně používaná. U originálních desek Arduino se port rozpozná jednoduše: v závorce za názvem portu je napsán typ desky, která je na daném portu připojená. U neoriginálních desek je třeba metody pokus–omyl.

- Stejně jako v běžném programu v C++, i zde můžeme přidávat knihovny pomocí `#include`, definovat makra a konstanty pomocí `#define` a vytvářet globální proměnné.
- Na rozdíl od běžných programů chybí vstupní funkce `main()`, která je nahrazena dvěma funkcemi: `setup()` a `loop()`

Funkce `setup()` se provede pouze jednou při spuštění nebo resetu Arduina. Provádí se v ní startování komunikace přes seriovou linku nebo I²C, inicializace komponent a v případě Trilobota i instanciace objektů, kterými se řídí jednotlivé senzory.

Ve funkci `loop()` se již nachází vlastní kód, který bude Arduino vykonávat stále dokola, dokud mu ne-vypneme napájení nebo nespustíme reset. Může se tedy jednat například o měření ze senzorů, čekání na nějakou akci a podobně. Tato funkce ovšem nemá garantované, jak rychle se vykoná. Pokud tedy potřebujeme, aby se něco vykonávalo pravidelně, je třeba funkci `loop()` nějak synchronizovat. Pro potřeby tutoriálu to ovšem potřeba nebude a pro jiné potřeby na to Trilobot má speciální část kódu uvnitř smyčky `loop()`.



Obrázek 2.2: Kde najdeme program Blink

Pokud jsme dříve potřebovali spojit dva kusy elektroniky do funkčního celku, měli jsme dvě možnosti: pouze „spojit dráty“ a upevnit je třeba lepící páskou nebo komponenty připájet. První přístup sice umožňuje jednoduše výsledek rozebrat, ovšem není vůbec spolehlivý. Druhý je sice mnohem spolehlivější, ale zařízení jsou spojena napevno a je velmi těžké je opět rozpojít.

V současné době ovšem většina prototypovacích desek používá princip nepájivého pole a DuPont kabelů, právě kvůli problémům uvedeným výše. Nepájivá pole zásadě umožňují relativně stabilně a spolehlivě spojit různé komponenty a opět je od sebe odpojit – vše si můžete prohlédnout na obrázku.

Samotné HW propojení se pak realizuje tak, že se kabel pomocí konektoru nasadí na pin – drátek vyvedený ze senzoru nebo příslušná zdírka podle typu samec/samice. Na druhé straně se poté propojí bud' přímo s prototypovací deskou nebo se využije nepájivého pole. Druhá možnost se využívá například v situaci, kdy se na jeden pin připojuje více zařízení (například při napájení nebo I²C sběrnici).

OOP

OOP je jedno z tzv. „Programovacích paradigm“ – způsobů, jak psát kód.

Nosná myšlenka je jednoduchá. V reálném světě máme objekty: kočky, auta, senzory a mnoho dalších. Proč bychom tedy nemohli psát v kódu jejich zjednodušené verze – modely – i s jejich vlastnostmi?

A přesně to dělá OOP. K reálnému objektu – třeba kočce – vytvoří její model v kódu – třída – i s jejími charakteristikami. Ty jsou v zásadě dvojího druhu: atributy (barva, stáří), a schopnosti (mňoukání nebo loudění o kapsičku), kterým se v OOP říká metody. Tím jsme vytvořili třídu `Kocka`, která popisuje objekt reálného světa – kočku.

Ovšem `Kocka` popisuje kočku obecně – jak popsat třeba naši kočku Rose? Jednoduše – vytvoříme si objekt Rose třídy `Kocka`, které nastavíme vlastnosti skutečné kočky Rose, tedy šedou barvu a věk tři roky.

```

/* Include knihoven*/
/* Definovani maker, konstant a podobne */
/* Globalni promenne */

void setup()
{
    /* Kod, ktery se provede jednou*/
}

void loop()
{
    /* Kod, ktery se bude provadet opakovane */
}

```

Obrázek 2.3: Struktura programu pro Arduino

Nyní opustíme reálný svět a ponořme se do toho počítačového. V kódu si neprve ukážeme třídu Kocka:

```

class Kocka
{
public:
    /* Konstruktor - priradi vstupni hodnoty do atributu */
    Kocka(jmeno, vek);

    /* metoda */
    void mnoukni();

    /* atributy */
    String jmeno;
    int vek;
}

```

Pokud nechápete syntaxi, nevadí, není to potřeba. Důležité jsou tři položky v těle třídy Kocka:

- Konstruktor: Zkráceně, konstruktor se stará o konstrukci konkrétního objektu. Mimo jiné, své argumenty přiřadí do atributů konkrétního objektu třídy Kocka.
- Metoda: Je to vlastně funkce, spojená s nějakým objektem.
- Atributy jsou vlastnostmi objektu.

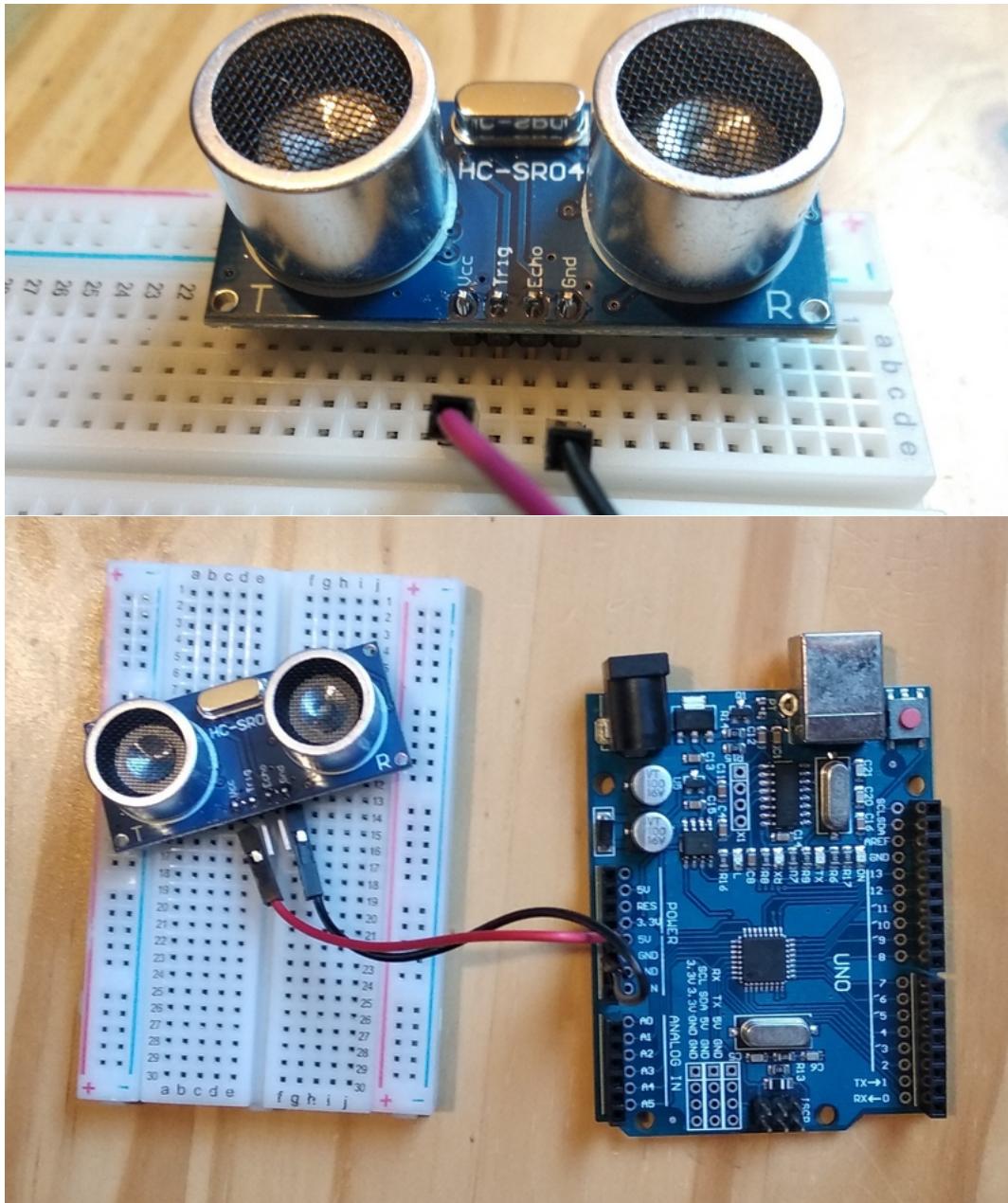
Tím jsme tedy do kódu přepsali obecnou charakteristiku kočky. Jak ale vytvoříme naši Rose? Snadno:

```
Kocka *rose = new Kocka("Rose", 3);
```

Tím jsme vytvořili objekt `rose` třídy `Kocka` – kódovou reprezentaci jedné konkrétní kočky. Hvězdičky si nemusíte všímat¹.

A jak naše Rose zamňouká? Jednoduše:

¹Ve skutečnosti se jedná o ukazatel na objekt a proto jsou metody volané přes šipku. Pokud by byla `rose` přímo objektem, používali bychom tečku. Všechny objekty senzorů na Trilobotovi jsou ovšem ukazatele a budeme pro ně tedy potřebovat šipku.



Obrázek 2.4: Na obrázcích vidíme, jak se propojují senzory. Nahoře je detail propojení přes nepájivé pole, dole potom napojení přímo na Arduino.

```
rose->mnoukni();
```

Metodu `mnoukni()` jsme tzv. „zavolali“ na objektu `rose` pomocí šípky `->`.

Na závěr si dovolím ukázat reálný příklad z kódu Trilobota, konkrétně použití I²C LCD displeje:

```
/* setup() */
Display *display = new Display();

/* loop() */
display->print_first_line("OOP je super!");
```

Ve funkci `setup()` jsme si vytvořili objekt `display` třídy `Display` a ve funkci `loop` jsme zavolali metodu `print_first_line()`, která vytiskne na první řádek displeje text „OOP je super!“.

Možná se toho na vás najednou sesypalo trochu více, ale netřeba se bát. Pro tento tutoriál si stačí zapamatovat jen to, že existují nějaké objekty a že jejich schopnosti se zavolají pomocí šípky.

Kapitola 3

Displej, reproduktor a I²C aneb Co se může hodit

V téhle kapitole si probereme pár věcí, které se prolínají všemi ostatními kapitolami, ale samostatná kapitola by se na ně tvořila dost špatně. O čem je řeč? Hlavně o displeji, který nám slouží k zobrazování různých dat bez toho, aniž by musel být Trilobot připojený Seriovou linkou k počítači.

Dále tu padne pár slov o I²C sběrnici, která je hojně využívaná všude, kde to senzory dovolily, a v neposlední řadě si povíme pár slov o reproduktoru.

3.1 Displej

Nejjednodušší způsob zobrazení informací je jejich odeslání seriovou linkou přes USB do počítače, kde se údaje zobrazí v terminálu. Co kdybychom ale chtěli zobrazovat informace bez připojení k počítači? K tomu nám slouží široká škála displejů!

Nejjednoduššími displeji jsou maticové LED displeje, které znáte například z vozů MHD. Nejedná se o nic složitého, je to skutečně jen hromada diod speciálně pospojovaná, aby pro každou samostatnou diodu nemusel existovat samostatný vodič. Většinou je tedy jeden vodič pro každý řádek a každý sloupec. Problémem tohoto řešení je, že v jednu chvíli nemůžeme na displej vysvitit celý text naráz, proto se používá malý trik. Cyklicky se zapínají jednotlivé řádky a vysvití se diody, které mají v tom daném snímku svítit. Pokud takhle přepínáme řádky dostatečně rychle, lidské oko nic nepozná.

Výhodou těchto displejů je obrovská odolnost a jednoduchost, nevýhodou je to, že mají hodně nízké rozlišení.

Tím se dostáváme k asi nejpoužívanějším displejům: LCD. Tyhle displeje můžete vidět například v kalkulačkách. Základem těchto displejů jsou tekuté krystaly. Ve zkratce, tekuté krystaly reagují na elektrický proud, který jimi prochází, tím, že změní svou orientaci. A světlo poté buď přes krystaly prochází nebo ne. U našich jednoduchých monochromatických displejů skutečně o nic víc nejdeme a operujeme tu jen s jedinou proměnnou: světlo prochází nebo ne. U složitějších LCD (velmi pravděpodobně na LCD čtete i tento text) vstupují do hry i barvy, rychlosť obnovy a tak dále. Těmi se tu ale zabývat nemusíme, protože u osmibitovým mikrokontrolerů s nimi moc často nepracujeme (vyžadují mnohem složitější řízení).

Posledním typem, který stojí za zmínku, jsou takzvané e-ink displeje, které můžete znát ze čteček knih. Jednotlivé body displeje (nemusí se nutně jednat o pixely, ale o body ještě menší nebo naopak větší, podle rozlišení) jsou tvořeny tzv „mikrokapslemi“, ve kterých je pigmentační medium, obsahující dva pigmenty: černý a bílý. Jedny částečky pigmentu jsou nabité záporně, druhé kladně. Podle potřeby se generuje kladné nebo záporné napětí na spodní části kapsle a tím jsou přitahovány částice pigmentu s opačnou polaritou a odpuzovány částice se stejnou polaritou. Odpuzované částice se poté nahromadí na vrchní vrstvě displeje, kde vytvoří obraz. Výhodou je, že displej nevyžaduje trvalé napájení, protože částice zůstávají na svých místech. Na druhou stranu, displeje jsou velmi pomalé a nejsou použitelné pro přehrávání videa nebo dynamicky proměnný obsah. Občas také při překreslení vznikají stíny.

Po tomto rychlém přehledu se můžeme vrhnout na LCD displej 16x2, který používá i Trilobot.

3.2 Displej 16x2

Displej 16x2 je jedním z nejpoužívanějších displejů pro Arduino a obecně pro osmibitové kontrolery. Nevyžaduje žádné složité řízení, je pro něj velmi kvalitní knihovna a v neposlední řadě je také velmi levný. Drobou nevýhodou je, že pro jeho řízení je zapotřebí 6 datových vodičů. Proto je i v případě Trilobota využit I²C adaptér, který umožňuje řídit displej právě přes tuto sběrnici. Obsluha displeje se provádí pomocí funkcí z knihovny LiquidCrystal_I2C.h, kterou si tychle vysvětlíme.

Pokud ovšem potřebujete jen jednoduše vypisovat základní údaje na řádek displeje (například naměřenou hodnotu ze senzoru), lze tak učinit i funkcí `display->print_first_line()`, určenou právě pro tento účel.

Použití knihovny `display.h`

Tato knihovna je pouze velmi zjednodušeným rozhraním pro výpis jedné hodnoty na jeden řádek displeje. Hlavním účelem je její použití při práci s ostatním vybavením Trilobota bez znalosti knihovny LiquidCrystal_I2C.h.

Použití je snadné. Podle toho, na jaký řádek chceme tisknout, zavoláme funkci `display->print_first_line()` nebo `display->print_second_line()`. O mazání se starat nemusíme, o to se stará sám objekt. Pokud si ale přejeme pouze smazat displej bez tisknutí ničeho jiného, poslouží nám k tomu funkce `display->print_clear()`. Podporované typy jsou všechny, které vytiskneme i přes seriovou linku, takže mimo jiné znak, řetězec, číslo.

```
#include <display.h> /* Knihovna pro praci s displejem*/
...
/** Funkce smazou dany radek a vytisknou data*/
display->print_first_line(to_print);
display->print_second_line(to_print);
display->clear(); /* Smaze cely displej */
}
```

Použití knihovny `LiquidCrystal_I2C.h`

Protože je knihovna `LiquidCrystal_I2C.h` velmi dobře napsaná a jedná se o de facto standard pro práci s LCD displeji na Arduinu, nebylo třeba vytvářet kompletní knihovnu plně podporující displej jen pro Trilobota. Zde tedy budou vysvětleny základy knihovny `LiquidCrystal_I2C.h`.

```
#include<LiquidCrystal\_I2C.h>
/* Deklarujeme si globalni promennou pro nas displej */
LiquidCrystal_I2C *display;
void setup() {
    /* Inicializace displeje - pokud bychom pouzivali
     * jiny displej, zmenime udaje */
    display = new LiquidCrystal_I2C(DISPLAY_ADDRESS, 16, 2);
    /* Zakladni nastaveni */
    display->init();
    /* Zapnuti podsviceni, ktere je v zakladu vypnute.
     * Muze setrít energii. */
    display->backlight();
    /* Nastaveni kurzoru na zacatek displeje.
     * Muzeme pouzivat i v loop(), první je hodnota sloupce,
     * druhá hodnota radku. Pozor na indexovani od 0 */
    display->setCursor(0,0);
}

void loop() {
    /* Vymaze displej */
```

```

display->clear();
/* Vytiskne libovolnou hodnotu na displej.
 * Pro cisla ma druhý parametr - pro celi soustavu,
 * pro desetinnna pocet mist*/
display->print(something);
}

```

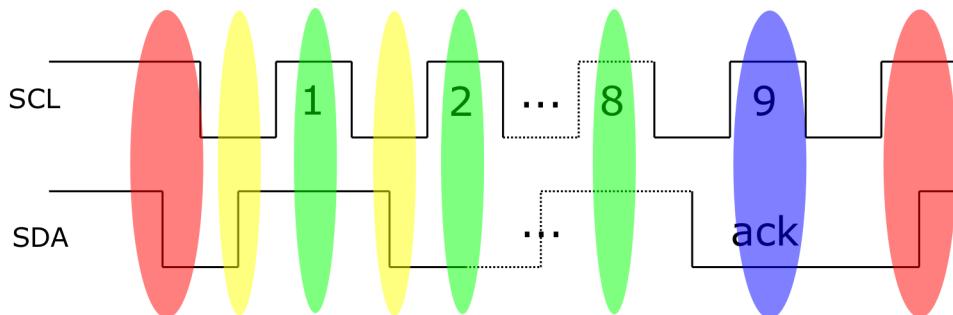
3.3 I²C sběrnice

Ted' si vysvětlíme I²C sběrnici (můžete se setkat i se zkratkami TWI nebo I²C, ale jedná se o totéž). Jedná se o jednu z nejjednodušších komunikačních sběrnic, která může propojit až 128 zařízení. Jedno ze zařízení, které je na I²C sběrnici, je tzv „master“. Master ,v našem případě Arduino, je zařízení, které řídí komunikaci, požaduje data po senzorech (zařízeních typu „slave“) nebo jim zasílá vlastní data nebo příkazy.

Jak se ale zařízení vzájemně poznají? K tomu nám slouží adresa, která je v případě I²C sedmibitová (rozsah je tedy 0 až 127). Každé zařízení má potom unikátní adresu, kterou často zapisujeme v hexadecimální soustavě, například 0x70.

A jak je I²C sběrnice realizovaná ve skutečnosti? Stačí nám k tomu dva vodiče: SCA a SCL. Přes SCL se přenáší hodinový signál a přes SCA samotná data. Není třeba se děsit nějakých hodin, jelikož pro praktické použití stačí propojit stejně označené piny na Arduinu a na senzoru, ideálně přes nepájivé pole. Správně se totiž musí každý ze dvou vodičů propojit přes tzv pull-up rezistor na napětí 5V. O důvodech si povíme v pokročilé sekci.

Samotná komunikace s Arduinem probíhá velmi jednoduše přes knihovnu `Wire.h`. Níže si ukážeme, jak přes I²C odeslat data na zařízení a jak je ze zařízení získávat.



Obrázek 3.1: na obrázku je znázorněn princip komunikace přes sběrnici I²C . START a STOP podmínky jsou znázorněny červeně. Při SCL 0 dochází ke změně signálu na SDA (žlutě), jehož hodnota je čtená při SCL 1 (zeleně). ACK podmínkou potvrzuje Slave příjem zprávy. Poznámka: v reálných podmínkách vidíme při ACK fázi chvílkové „vystřelení“ SDA do 1. Je to způsobeno chvílí, kdy Master již sběrnici SDA neovládá a Slave ji ještě ovládat nezačal.

Nejdříve odesílání:

```

#include <Wire.h> /*Díky knihovné muzeme používat funkce Wire*/
Tu...
Wire.begin();
...
/*Zahajíme komunikaci se zařízením dane adresy*/
Wire.beginTransmission(adresa_zarizeni);

```

```
Wire.write(data); /*Posleme data*/
Wire.endTransmission(); /*Ukoncime komunikaci*/
```

Proměnná `adresa_zarizeni` je, jak jsme si už řekli, sedmibitová identifikace daného zařízení, vždycky jedinečná v rámci dané sběrnice. Proměnná `data` jsou data, která chceme na senzor odeslat. Důležitá poznámka: `data` jsou typu `byte`. Pokud chceme poslat data větší než jeden byte, třeba číslo nebo řetězec, musíme funkci označit, kolik bajtů zabírají daná data. `Wire.write(data)` by tedy vypadal takhle:

```
Wire.write(data, sizeof(data)); /*Posleme data o dane velikosti*/
```

A teď i získávání:

```
#include <Wire.h>
...
Wire.begin();
...
/*Od zarizeni na dane adrese chceme tento pocet bajtu*/
Wire.requestFrom(adresa_zarizeni, kolik_chceme_bajtu);
if(Wire.available()) /*Pokud jsou data k dispozici...*/
{
    data = Wire.read(); /*...tak je precteme*/
}
```

Kód je opět velmi jednoduchý. Nejprve určíme, od kterého zařízení budeme chtít data a kolik jich bude. Funkce `Wire.available()` zjišťuje, zda dané zařízení vůbec má tolik dat, aby je mohlo poslat. Pokud je dat dost, v cyklu budeme číst bajt po bajtu (a pro příklad je budeme ukládat do předem deklarovaného pole bajtů).

I²C do detailu

Pokud vás zajímá, jak probíhá komunikace po I²C bit po bitu, zde se to dozvíte :). V základní části je hodně teorie přeskročeno, jelikož zabere nějaký čas ji vysvětlit. Na druhou stranu, není to nijak složité a rozhodně se hodí vědět, co těmi dráty teče.

Komunikace vždycky začíná tzv START podmínkou, kterou vždycky vytváří Master (Slave nemůže začít komunikovat z vlastní vůle). Master na SDA nastaví LOW, zatímco je SCL ve stavu HIGH (v klidovém stavu jsou oba vodiče ve stavu HIGH díky pull-up rezistorům, o kterých si něco povíme na konci). Poté začne vysílat hodinový signál na SCL a zároveň data po SDA. Slave čte data vždy při náběžné hraně SCL (když signál SCL „nabíhá“ do stavu HIGH) a Master mění hodnotu na SDA vždy při SCL ve stavu LOW.

První přenášený byte je adresa zařízení o délce 7 bitů a osmý bit, který určuje, jestli bude Master vysílat nebo přijímat. Říká se mu read/write bit, 0 pro vysílání, 1 pro přijímání. Po přenesení celého bytu dojde k jeho potvrzení ze strany Slave zařízení, které rozpoznalo svou adresu: Master stále generuje SCL signál, ale SDA nechá volný (a díky pull-up rezistoru bude ve stavu HIGH). Slave, pokud přijal všech 8 bitů, SDA „stáhne do nuly“ (nastaví na SDA stav LOW) a tím potvrdí příjem. Pokud zůstane SDA ve stavu HIGH, známená to, že nikdo svou adresu nerozpoznal a Master nemá s kým komunikovat.

Pokud je ale adresa úspěšně rozpoznána, začíná vlastní komunikace, která probíhá úplně stejně. Master odešle 8 bitů a v devátém si nechá potvrdit příjem. Na Závěr posílá Master STOP podmínku, kterou ukončí komunikaci: oba vodiče ponechá ve stavu HIGH.

Pokud by chtěl Master data přijímat, funguje to velmi podobně. SCL signál stále generuje Master, ale data na SDA vkládá Slave a Master je potvrzuje. Komunikaci ukončí jednoduše: prostě nepotvrdí přijatá data, čímž Slave zastaví vysílání.

A co že to jsou ty pull-up rezistory? Ti, co umí anglicky už asi tuší. Pull-up rezistor dělá přesně to, co má v názvu, „vytahuje“ signál na vodiči na úroveň HIGH ve chvíli, kdy by byl vodič jinak odpojený

(správně by se mělo říkat „stav vysoké impedance“).

3.4 Reproduktor

Reproduktor je jen doplňkovým zařízením a vzhledem k tomu, že nelze nastavovat hlasitost, nedoporučuji ho používat tam, kde byste jeho zvukem mohli rušit ostatní – třeba v laboratoři.

Pokud ho ale budete chtít použít, je to velmi snadné. Před použitím je třeba reproduktor aktivovat funkcí `speaker->enable()` a po použití je nutné reproduktor opět vypnout funkcí `speaker->disable()`, jinak bude stále potíchu pískat. Další funkce můžete vidět níže:

```
#include <speaker.h>
/* Pred pouzitim zapneme reproduktor */
speaker->enable();

/* Note: nazev noty - podivejte se do speaker.h pro seznam
 * Duration: cas trvani noty v ms
 * */
speaker->beep(<note>, <duration>);
/* Zahraje Imperial March ze Star Wars : */
speaker->imperial_march();

/* Po pouziti ho vypneme, jinak bude vydavat piskani */
speaker->disable();
```

Funkce `tone()` a `beep()`

Jak vidno z oficiální dokumentace¹, funkce generuje „pravoúhlý signál o dané frekvenci a střídě 50 %“ a „může sloužit pro generování tónů na reproduktorech“. Přeloženo do češtiny, pokud na daném pinu budeme mít zapojený reproduktor, funkce `tone` na něm zahraje tón o dané frekvenci a délce. Její syntaxe je velice jednoduchá:

```
void tone(int pin, unsigned int frequency, unsigned long duration)
```

kde `pin` je číslo pinu, na kterém je reproduktor (v kódu makro `SPEAKER_PIN`), `frequency` je frekvence tónu v hertzích a `duration` je délka tónu v milisekundách. Abychom nemuseli dohledávat frekvence jednotlivých tónů, v knihovně `speaker.h` jsou makra pro základní noty (c až aH), což nám samozřejmě nebrání použít i libovolnou jinou frekvenci.

Funkce `tone()` přitom program nepozastaví a chcete-li nechat hrát několik not za sebou, je nutné po každé notě explicitně zastavit program funkcí `delay()`. Pro zjednodušení práce obsahuje knihovna `speaker.h` funkci `beep(int note, int duration)`, která všechno tohle řeší za vás.

Pro ukázkou je v knihovně `speaker.h` připravená funkce `Speaker::imperial_march()`, hrající známou melodii z filmové série Star Wars.

¹<https://www.arduino.cc/reference/en/language/functions/advanced-io/tone/>

Kapitola 4

Zjišťujeme množství světla – fotosenzor

V této kapitole si ukážeme, jak můžeme pomocí senzorů zjistit množství světla v místnosti a vyzkoušíme, jestli všechno funguje, jak má.

4.1 Jak funguje fotosenzor – základy

Fotosenzor na SRF-08 je kompletní obvod, který změří údaje a pošle je přes I²C sběrnici do Arduina. V základě se ovšem jedná o obyčejný fotorezistor, který jde koupit i samostatně doslova za pár korun. Co je to ale fotorezistor?

Je to polovodičová součástka, podobně jako dioda nebo tranzistor, ale s tím rozdílem, že nemá ani jeden PN přechod a ani samotný polovodič není dotovaný. Ve zkratce, za normálních podmínek takový polovodič povede jen velmi špatně. Jenže při vystavení světlu dochází k tzv. „vnitřnímu fotoelektrickému jevu“: foton předá energii elektronu ve valenční vrstvě a ten se utrhne ze své pozice a začne se volně pohybovat v látce. Tím nám vznikne dobré známý pár elektron–díra, který může vést proud.

Práce s fotosenzorem je opravdu velmi jednoduchá. Jelikož se jedná o součást senzoru SRF-08, i k jeho obsluze nám poslouží objekt `srf08`. Komunikaci přes I²C sběrnici si dopodrobna vysvětlíme v dalších krocích, stejně jako práci s displejem nebo seriovou linkou. Naším jediným úkolem bude načítat údaje z fotosenzoru a vypsat je na displej Trilobota.

Nejdřív si zažádáme o data ze senzoru a přečteme si je. Jelikož SRF-08 neumí jen změřit hodnotu osvětlení, ale provádí měření spolu se vzdáleností, trochu nelogicky použijeme funkci `this->set_measurement()`. Velmi stručně, funkce zašle do senzoru informaci o tom, v jakých jednotkách chceme měřit. Jelikož nám jde ale pouze o vedlejší data, samotný výsledek měření nás až tak nezajímá.

V dalším kroku si implementujeme funkci `zmer_svetlo()`, která zkонтaktuje po I²C sběrnici senzor SRF08 a přečte hodnotu ze senzoru světla. Kostru a její popis můžeme vidět níže. Samotná kostra se nachází v souboru `srf08.cpp`.

```
byte SRF08::zmer_svetlo()
{
    this->set_measurement(); /* Zahajení merení */
    /*Pozadáme SRF-08 o informaci
    pocinající REGISTREM FOTOCIDLA*/
    Wire.beginTransmission(SRF08_ADDRESS);
    /* 0x00 je jen zastupny znak, aby sel kod preložit */
    Wire.write(0x00/*TODO REGISTR FOTOCIDLA*/);
    Wire.endTransmission();

    /*Pozadáme pouze o jeden bajt...*/
    Wire.requestFrom(SRF08_ADDRESS, 1);
    /*... a počkáme, dokud nebude dostupný*/
    while(Wire.available() < 0);
```

```
/*TODO: precist hodnotu a vratit ji.  
 * Nasledujici radky pouze zajisti, ze se kod  
 * zkompiluje, nakonec vracejte namerenou hodnotu.*/  
 byte light_intensity = 0;  
 return light_intensity ;  
}
```

Z dokumentace k SRF-08 (strana 4) vidíme správné číslo registru, které můžeme doplnit do kódu. V druhé části pomocí funkce `Wire.requestFrom()` požádáme senzor SRF-08 o jeden bajt dat.

Ten podivný cyklus je něco, čemu se říká aktivní čekání. Senzor totiž potřebuje nějaký čas, aby se připravil a byl schopen nám data poskytnout a my mu tento čas musíme dát právě tímhle cyklem, kde se stále dokola ptáme, „jestli už máme nějaká data“.

Nyní je už senzor připravený a nám stačí si data jen přečíst, což provedeme pomocí funkce `Wire.read()`. Tato funkce nám vrátí jeden bajt přečtených dat, což je přesně tolik, kolik potřebujeme. Posledním krokem je data vrátit.

Posledním krokem je zavolat naši novou funkci ve smyčce `loop()` a její hodnotu vypsat jak na seriovou linku (data uvidíme na počítači) pomocí `Serial.println()`, tak na displej pomocí `display->print_first_line()`. Dále ještě doporučuji přidat rádek `delay(1000)`, který smyčku vždycky na vteřinu pozastaví. Jinak dojde dost pravděpodobně k výpisu chaotických znaků na displej, který nebude stíhat příval dat.

Pokud je vše napsané správně, projekt přeložíme a nahrajeme do Arduina a to by mělo po chvíli začít vysílat naměřená data. Pokud se někde vyskytla chyba, na další straně bude ukázkový kód toho, jak by to mohlo vypadat.

Pokud vám přišel příklad jako poněkud umělý a „neužitečný“, zkuste třeba při snížené intenzitě světla rozsvítit vestavěnou LED. Jelikož ta je spřažená s pinem 13, rozsvítí se i LED vedle displeje, netřeba rozebírat celého Trilobota pro kontrolu. Pomoci vám může třeba příkladový kód Blink nebo internet. :)

Řešení

```
byte SRF08::zmer_svetlo()
{
    this->set_measurement(); /* Zahajeni mereni */
    /*Pozadame SRF-08 o informaci
    pocinajici REGISTREM FOTOCIDLA*/
    Wire.beginTransmission(SRF08_ADDRESS);
    Wire.write(0x01);
    Wire.endTransmission();

    /*Pozadame pouze o jeden bajt...*/
    Wire.requestFrom(SRF08_ADDRESS, 1);
    /*... a pockame, dokud nebude dostupny*/
    while(Wire.available() < 0);

    byte light_intensity = Wire.read();
    return light_intensity;
}
/* V loop()*/
display->print_first_line(srf08->photosensor());
```

Kapitola 5

Měříme vzdálenost – senzory vzdálenosti HCSR-04, SRF08 a Sharp

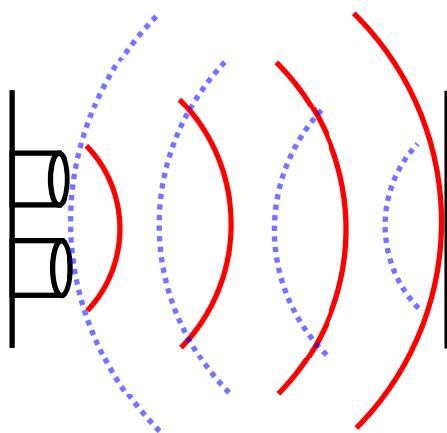
V této části si představíme celkem tři senzory: infračervený senzor Sharp a dva ultrazvukové senzory: HC-SR04 a SRF-08, jehož fotosenzor jsme používali v první kapitole. Kromě toho si také vysvětlíme, jak jednotlivé senzory mohou s Arduinem komunikovat a ukážeme si to na dvou případech: se senzory Sharp a HC-SR04 budeme komunikovat „na přímo“ – pomocí samostatného datového vodiče pro každý senzor zvlášť – kdežto u senzoru SRF-08 využijeme I²C sběrnici.

5.1 Jak se měří vzdálenost?

Vzdálenost potřebujeme měřit velmi často: například autonomní vozidla využívají různé senzory vzdálenosti¹ pro zaparkování nebo udržování bezpečné vzdálenosti. Vlastně téměř vždy, když se robot pohybuje v prostředí, které nezná nebo se mění, potřebuje nějak zjistit, co je kolem něj. A nejdůležitější údaj je právě to, v jaké vzdálenosti se kolem něj nacházejí různé objekty.

Vzdálenost můžeme měřit různě, ale nejčastěji senzory pracují tak, že vyšlou nějakou energetickou vlnu a počítají čas, za který se vlna vrátí. Tahle vlna může být mechanická – zvuk – nebo elektromagnetická – světlo nebo radarové vlny. Senzor pak buď sám spočítá výslednou hodnotu nebo pošle mikrokontroleru data. Princip se liší senzor od senzoru, kde se mu budeme věnovat podrobněji.

¹Zde se často jedná o tzv lidar: ten vysílá laserový paprsek a počítá dobu, za kterou se tento paprsek vrátí do snímače.



Obrázek 5.1: Na obrázku je znázorněn princip, na jakém funguje senzory HC-SR04, SRF-08 a také senzor SHARP. Senzor nejprve vysíle vlnu (červeně), která se následně odrazí od překážky (modře). Změřením času mezi odesláním a příchodem vlny a jeho vydělením rychlosť získáváme vzdálenost mezi senzorem a překážkou.

Měření podrobněji

Přesný vzorec pro získání vzdálenosti v centimetrech je vlastně vcelku jednoduchý:

$$s = t * v / 2 \quad (5.1)$$

kde:

- s je vzdálenost v cm
- t je čas mezi vysláním a příjemem pulzu/vlny
- v je rychlosť šíření vlny

Jelikož vlna letí oběma směry, je nutné celkový čas podělit dvěma.

Problém ovšem nastává s přesnou rychlosťí vlny, zvlášť pokud senzor používá k měření ultrazvuk. Rychlosť zvuku se totiž mění podle vlhkosti a teploty vzduchu: rozdíl mezi rychlosťí vzduchu při teplotě -10°C a $+40^{\circ}\text{C}$ je zhruba 10 % (325 vs 355 m/s). Rychlosť se také nepatrně zvyšuje při vlhčím vzduchu.

Budu musíme senzor doplnit i například o teploměr a vlhkoměr, kterým bychom použitou rychlosť ve vzorci korigovali pro správnou teplotu a vlhkost, nebo zvolíme nějakou střední hodnotu (například pro $+20^{\circ}\text{C}$) a budeme počítat s odchylkami při jiné teplotě. I v praxi se řeší podobné problémy a vývojáři si kladou otázky, zda jim podobné nepřesnosti vadí. Pokud bychom podobný senzor použili třeba v robotovi na hraní, vadit to pravděpodobně nebude. Ovšem třeba pro autonomní vozidlo by to už problém být mohlo.

Do Arduina se poté hodnoty přenáší třemi způsoby. HC-SR04 vyšle pulz v okamžiku, kdy se k němu vrátí odražená vlna. Arduino si tedy musí pamatovat, kdy poslalo senzoru příkaz k měření aby zjistilo celkovou dobu šíření vlny. Senzor SRF-08 komunikuje s Arduinem po sběrnici I²C (bude o ní řeč dále) a Arduino dostane přímo hodnotu v cm (nebo jiných jednotkách). A Sharp-GP2D120 trvale udržuje na datovém pinu hodnotu napětí, odpovídající měřené vzdálenosti podle grafu funkce. Zkuste sami přijít na plus a minusy jednotlivých řešení (odpovědi dole^a).

^aPrincip senzoru SRF-08 je pro uživatele asi nejpohodlnější, ale jsou třeba 4 kabely na propojení a samotný senzor je relativně drahý. HC-SR04 má výhodu v absolutní jednoduchosti (a také v ceně), ale na druhou stranu musí Arduino počítat čas – například příchod přerušení by mohl celé měření znehodnotit (o přerušení budeme mluvit dále, prozatím stačí vědět to, že je to situace, Arduino začne okamžitě vykonávat jiný kus kódu). No a Sharp-GP2D120 má výhodu například v tom, že není třeba nic počítat, stačí jen snímat hodnotu na daném pinu. Problém ovšem nastává, pokud chceme hodnotu napětí převést na vzdálenost, jelikož závislost nemí lineární.

5.2 Měření senzorem HC-SR04

Po nezbytné teorii můžeme konečně přejít k programování, a to konkrétně programování senzoru HC-SR04. Je to ten jednodušší z dvojice ultrazvukových senzorů (I²C zatím nebude potřeba), je jednoduchý, levný a pro běžné použití i dostatečně přesný. Používá ho většina doma postavených robotů, kteří nějak potřebují měřit vzdálenost a dá se s ním dělat opravdu mnoho zajímavého. Od měření vody v nádrži až po domácí radar. My si zprovozníme základní měření, na kterém si ukážeme, jak přesně se se senzorem pracuje.



Obrázek 5.2: Ukázka senzoru HC-SR04.

Dle dokumentace jsou zapojení a komunikace relativně snadné. V první části dokumentace (Product fea-

tures) je popis činnosti popsán jednoduše:

1. Alespoň 10 mikrosekund uvést pin Trig do stavu HIGH
2. Změřit délku trvání pulzu HIGH na pinu Echo

To zní dobře. Připravíme si tedy funkci, která nám bude měřit vzdálenost. V kódu je již připravená kostra, kterou najdete v souboru `hcsr04.cpp` a jmenuje se `long HCSR04::zmer_vzdalenost()`.

```
long HCSR04::zmer_vzdalenost()
{/* Pro jistotu na chvíli nastavíme stav LOW */
digitalWrite(TRIGGER_PIN, LOW);
delayMicroseconds(2);
/* TODO: Nastavit TRIGGER_PIN do stavu HIGH na 10
 * mikrosekund, pote ho vrátit do LOW*/
/*TODO: pomocí funkce pulseIn(pin, stav) zjistit, jak
 * dlouho bude na ECHO_PIN stav HIGH a prevest
 * cas na vzdalenost*/
long return_value = 0;
return return_value;
}
```

Ze všeho nejdřív musíme vyslat 10mikrosekundový pulz. Nejjednodušší cesta je takováto:

```
long HCSR04::zmer_vzdalenost()
{/* Pro jistotu na chvíli nastavíme stav LOW */
digitalWrite(TRIGGER_PIN, LOW);
delayMicroseconds(2);
digitalWrite(TRIGGER_PIN, HIGH);
delay(10);

/*TODO: pomocí funkce pulseIn(pin, stav) zjistit, jak
 * dlouho bude na ECHO_PIN stav HIGH a prevest
 * cas na vzdalenost*/
long return_value = 0;
return return_value;
}
```

Nyní už stačí jen změřit délku impulzu na pinu Echo. To nebudeme programovat ručně, ale využijeme funkci `PulseIn()`, která slouží přesně k tomuhle účelu. Měření délky pulzu by totiž nebylo úplně snadné². Funkce `PulseIn()` potřebuje dva parametry: pin, kde se má pulz objevit a stav, který má počítat. V našem případě to bude `ECHO_PIN` (opět makro, podobně jako `TRIG_PIN` a stav HIGH). Mimochodem, počet mikrosekund může být relativně velký, proto místo běžného typu `int` použijeme větší typ `long`, který vrací i funkce `PulseIn()`. Náš kód teď tedy vypadá takto:

```
long HCSR04::zmer_vzdalenost()
{/* Pro jistotu na chvíli nastavíme stav LOW */
digitalWrite(TRIGGER_PIN, LOW);
delayMicroseconds(2);
```

²Zkuste se zamyslet, jak byste to udělali vy. Asi vás napadne ve smyčce kontrolovat hodnotu pinu a sotva se dostane do stavu HIGH, začít ve druhé smyčce zvyšovat nějaký čítač. Ted' byste museli převést hodnotu čítače na časový údaj. Museli byste zjistit frekvenci procesoru (což je zrovna docela známý údaj, 16 MHz :)), jak dlouho trvá, než se čítač zvýší a podobně. Funkce `PulseIn` funguje podobně, nepoužívá funkce jako `digitalRead()`, které jsou docela pomalé, ale pracuje přímo s registry. Pokud pracujete ve Visual Studio Code, stačí podržet Ctrl a kliknout na funkci `PulseIn()`. Ukáže se vám její implementace.

```

digitalWrite(TRIGGER_PIN, HIGH);
delayMicroseconds(10);
digitalWrite(TRIGGER_PIN, LOW);

long echo = pulseIn(ECHO_PIN, HIGH);
return return_value;
}

```

Pokud je všechno tak, jak by mělo být, funkce by už měla něco vracet. Můžete si ji zavolat ve hlavní smyčce a její hodnotu si vytisknout na displej, třeba takhle:

```

void loop()
{
    long echo = hcsr04->zmer_vzdalenost();
    display->print_first_line(echo);
    delay(1000); /*pockame vterinu*/
}

```

Funkce zatím rozhodně nevrací vzdálenost. Vrací totiž počet mikrosekund od vyslání po návrat pulzu (vzpoříme si na obrázek s vlnami). Tedy tedy potřebujeme převést počet mikrosekund na vzdálenost v centimetrech. Jak na to? To je vcelku jednoduchá matematika. Máme čas, potřebujeme vzdálenost, jediné, co musíme udělat, je tedy vynásobit hodnotu echo rychlosťí zvuku. Ta je zhruba 340 m/s, pro naše potřeby se ale bude hodit jednotka 0.0343 cm/us. Nesmíme zapomenout ještě vydělit dvěma, chceme vzdálenost jen v jednom směru.

Zpět do kódu. Upravme tedy naši funkci:

```

long HCSR04::zmer_vzdalenost()
{/* Pro jistotu na chvíli nastavíme stav LOW */
    digitalWrite(TRIGGER_PIN, LOW);
    delayMicroseconds(2);

    digitalWrite(TRIGGER_PIN, HIGH);
    delayMicroseconds(10);
    digitalWrite(TRIGGER_PIN, LOW);

    long echo = pulseIn(ECHO_PIN, HIGH);
    long return_value = echo * 0.0343;
    return return_value;
}

```

Znovu si zkusíme naši funkci. Tedy by už měla vracet správné výsledky v centimetrech. Gratuluji, zprovoznili jste svůj první senzor! Mimochodem, pokud vám něco nefunguje nebo se chcete podívat na referenční implementaci, podívejte se do souboru hcsr04.cpp nebo vyzkoušejte funkci hcsr04->get_distance(). Možná vás napadne i další použití senzoru.

Když máme první kolo za sebou, můžeme se vrhnout na trochu složitější verzi: SRF-08.

5.3 Měření senzorem SRF-08



Obrázek 5.3: Ukázka senzoru SRF-08.

SRF-08 je velmi podobné senzoru HC-SR04, také používá pro měření vzdálenosti ultrazvuk. Liší se ovšem tím, jak komunikuje s Arduinem a co všechno umí. Kromě měření vzdálenosti v různých jednotkách a modech zvládá díky vestavěnému fotosenzoru (kterému je věnovaná samostatná kapitola) měřit i úroveň světla. My si probereme jen základní mod – jednoduché měření vzdálenosti.

Když se opět podíváme do dokumentace³, zjistíme, že senzor má celých 35 registrů! Dobrou zprávou je, že zapisovat můžeme jen do prvních třech, ostatní jsou jen pro čtení. Důležitá je také jejich velikost, pouze jeden byte, což nám bude v pozdějších fázích trochu komplikovat získávání údajů (ale nejde o nic těžkého, jen je třeba o tom vědět).

Pokud chceme zapsat do konkrétního registru, nejprve přes I²C odešleme jeho adresu a poté data o velikosti jeden byte. Takže například pro započetí měření chceme zapsat do registru s adresou nula. Nula je všude nula, ale pro „osmibitovou nulu“ můžeme použít i zápis 0x00, který nám přímo říká, že se jedná o osmibitové hexadecimální číslo. U nenulových čísel je tenhle zápis vůbec nejlepší (a také nejvíce používaný). Co se týče zapisované hodnoty, my požadujeme „Ranging Mode“, což znamená, že chceme měřit vzdálenost. Jednotku si můžeme vybrat: centimetry(0x51), palce (0x50), nebo mikrosekundy(0x52). Všechno je i v dokumentaci, strana 4, spodní tabulka. Mimochodem, ono „0x“ pře číslem značí, že je číslo zapsané v hexadecimální soustavě, která se v informatice používá velmi často. Pokud chcete, můžete si číslo převést do desítkové soustavy (nebo se podívat do dokumentace), ničemu to nevadí.

Kromě toho je ještě dobré zmínit jednu věc. Senzor je součástka jako každá jiná a ultrazvukové měření může chvíli trvat. Proto mezi zápisem příkazu a čtením požadovaných dat musíme chvíli počkat. Pro náš senzor je to nejméně 65 milisekund.

Když jsme si vysvětlili základní principy, můžeme se pustit do programování. Naším cílem je jednoduché měření v centimetrech. S nastavováním senzoru se tu nemusíme zdržovat, všechno za nás zařídí funkce void SRF08::set_measurement(byte unit) a na nás zbude pouze přečíst data z registrů, spojit je a vrátit. Kostru najdete v souboru srf08.cpp

```
int SRF08::zmer_vzdalenost()
{
    /* Navazeme komunikaci */
    Wire.beginTransmission(SRF08_ADDRESS);
    /* Napiseme, kam chceme zapisovat... */
    Wire.write(REG_CMD); /* expanduje do 0x00 */
    /* ... a co chceme zapisovat... */
    Wire.write(0x00/*TODO spravna jednotka k nalezeni ve slajdech*/);
    /* Nakonec ukoncime prenos */
    Wire.endTransmission();

    /* TODO: pockat 100 ms */

    /* Zajimaji nas data od registru 0x02*/
    Wire.beginTransmission(SRF08_ADDRESS);
    Wire.write(0x02);
    Wire.endTransmission();
    /* TODO precist data, spojit je a vratit*/

    uint16_t return_value = 0;
    return return_value;
}
```

Čtení nebude nijak těžké. Potřebujeme data z registrů 2 a 3 (0 není ke čtení a 1 je senzor světla). Kostra už za nás nastavila potřebný start čtení, stačí data jen přečíst, třeba takto:

```
int SRF08::zmer_vzdalenost()
{
```

³<http://coecsl.ece.illinois.edu/ge423/devantechsr08ultrasonicranger.pdf>, strana 4

```

Wire.beginTransmission(SRF08_ADDRESS);
Wire.write(REG_CMD); /* expanduje do 0x00 */
Wire.write(0x51); /* centimetry*/
Wire.endTransmission();
delay(100); /* cekani */
Wire.beginTransmission(SRF08_ADDRESS);
Wire.write(0x02);
Wire.endTransmission();

Wire.requestFrom(SRF08_ADDRESS, 2);
while(Wire.available() < 0);
byte high = Wire.read();
byte low = Wire.read();

uint16_t return_value = 0;
return return_value;
}

```

Ted' už nám stačí jen spojit dva bajty do jednoho čísla. Jak? Použijeme operátor bitového posuvu. Ten dělá přesně to, co bychom od něj čekali, posouvá číslo v rámci kapacity jeho datového typu „doprava“ nebo „doleva“. Nejprve si připravíme proměnnou typu `uint16_t`, 16bitové celé číslo bez znaménka. Spodních 8 bitů bude zabírat číslo `low`, zbytek číslo `high`, které dostaneme na horních 8 bitů právě bitovým posunem o 8. Nesmíme zapomenout přetypovat číslo `high` na správný datový typ, aby se mělo kam posouvat.

```

/*Připravíme si číslo na výsledek/
uint16_t value = 0;
/*A pak do nej vložíme spojené hodnoty.*/
uint16_t number = (uint16_t)(high << 8)+low;

```

Ted' už stačí jen spojenou hodnotu vrátit a máme hotovo.

```

int SRF08::zmer_vzdalenost()
{
    Wire.beginTransmission(SRF08_ADDRESS);
    Wire.write(REG_CMD); /* expanduje do 0x00 */
    Wire.write(0x51); /* centimetry*/
    Wire.endTransmission();
    delay(100); /* cekani */
    Wire.beginTransmission(SRF08_ADDRESS);
    Wire.write(0x02);
    Wire.endTransmission();

    Wire.requestFrom(SRF08_ADDRESS, 2);
    while(Wire.available() < 0);
    byte high = Wire.read();
    byte low = Wire.read();

    uint16_t return_value = (uint16_t)(high << 8)+low;
    return return_value;
}

```

Nyní si zkuste porovnat hodnoty ze senzorů HC-SR04 a SRF-08. Oba by měly vypisovat podobnou, i když asi ne úplně stejnou vzdálenost. Dokážete odhadnout, čím by to mohlo být?

Odpověď je jednoduchá: HC-SR04 je určený pro kutily a jeho hlavní předností je cena (na různých čínských eshopech stojí v přepočtu méně než dvacku). Na přesnost tedy nijak nehraje a většinou nevadí, že je naměřená

vzdálenost o pár centimetrů na metr jiná než ve skutečnosti. Řešením by mohlo být třeba to, že bychom zprůměrovali několik hodnot a zkusili senzor zkalibrovat – do pevně dané vzdálenosti bychom umístili překážku a sledovali, co by naměřil. Nejjednodušší je ovšem prostě se s tím smířit.

5.4 Měření senzorem Sharp-GP2D120



Obrázek 5.4: Ukázka senzoru SHARP.

Senzor Sharp-GP2D120 na rozdíl od předchozích senzorů nepracuje se zvukem, ale s infračerveným světlem. Samotné měření je naprostě jednoduché. Sotva senzor začneme napájet, začne s měřením a na svůj jedený datový pin nastaví napětí, které odpovídá naměřené vzdálenosti. Důležitým detailem je i to, že minimální vzdálenost, kterou je schopen změřit, jsou zhruba 3 centimetry. Proč, to si vysvětlíme dále.

Samotné získávání údajů je tedy velmi jednoduché, stačí přečíst analogovou hodnotu z datového výstupu senzoru. Problém ovšem nastává, pokud chceme z této hodnoty získat smysluplná data. Podívejme se do dokumentace⁴ senzoru, konkrétně na strany 4 a 5. Graf závislosti naměřené hodnoty totiž nejen že není lineární, ale dokonce dvěma vzdálenostem odpovídá stejná hodnota napětí! Proto jsou výsledky měření z tohoto senzoru validní až od vzdálenosti 3 centimetrů.

Jak tedy konvertovat získanou hodnotu na smysluplný údaj? Obecně se používají dva postupy. Bud' se snažíme najít funkci, která má stejný průběh jako naše závislost. Tomu se říká analytický postup. Druhému způsobu se říká numerický. Ten funguje tak, že se průběh snažíme co nejlépe popsat, ale již pracujeme s chybou. Jelikož jsou oba postupy poměrně zdlouhavé, použijeme funkci `approximate()`. Ta implementuje tzv. lineární splajn. Pod tímhle podivným souslovím se ale skrývá naprostě jednoduchá věc. Závislost je rozdělena na několik menších úseků a jejími krajními body je poté proložena přímka, která na dostatečně malém úseku popisuje závislost dostatečně přesně.

Ted' se můžeme pustit do kódování. Kostru funkce najdete v souboru `sharp1994v0.cpp`.

```
float Sharp::zmer_vzdalenost()
{
/*TODO:
 * precist hodnotu na SHARP_PIN
 * ziskanou hodnotu prevest do rozsahu 0-5
 * Pomoci funkce this->approximate ziskat hodnotu v cm a vratit ji
 */
    float approximated_value = 0;
    return approximated_value;
}
```

Nejprve si načteme hodnotu z analogového pinu, kde je připojený senzor. Funkce `analogRead()` ovšem nevrací přímo hodnotu napětí na daném pinu, ale hodnotu z intervalu 0–1024 (maximální hodnota se obecně může lišit, proto je tu makro `ANALOG_MAX`). Následně ještě podíl vynásobíme makrem `DEFAULT_VOLTAGE`, čímž se dostaneme na rozsah 0 až `DEFAULT_VOLTAGE` voltů. v našem případě 0 až 5. Některá Arduina ovšem pracují na 3.3 voltech. Toto makro nám poté umožní upravit základní voltáž na jednom místě.

Na závěr stačí approximovat získanou hodnotu a máme výsledek! Níže si můžete prohlédnout hotovou funkci.

⁴<https://www.pololu.com/file/0J157/GP2D120-DATA-SHEET.pdf>

```
float Sharp::zmer_vzdalenost()
{
    int value = analogRead(SHARP_PIN);

    /*ANALOG_MAX = 1024. DEFAULT_VOLTAGE = 5*/
    float normalized_value = 0;
    normalized_value = (value/ANALOG_MAX)*DEFAULT_VOLTAGE;

    float approximated_value = this->approximate(normalized_value);
    return approximated_value;
}
```

Kapitola 6

Jezdíme – motory a enkodéry

Tato kapitola bude poněkud zajímavější. Trilobot je vybavený dvěma motory Faulhaber 16002 s rotačními enkodéry pro zpětnou vazbu o pohybu.

Pod pojmem rotační enkodér si představte senzor, který nějak periodicky mění signál (třeba napětí na vodiči) podle toho, jak se mění nějaký mechanický pohyb – třeba podle toho, jak se otáčí motor. V našem případě je senzorem dvojitý senzor IR světla (IR dioda na jedné stráně kotouče a senzor IR světla na druhé. Kotoučem je skutečně kotouč, který má po obvodu výrezy. Jak se kotouč pohybuje, postupně jednotlivé zářezy propustí světlo do prvního a poté i do druhého senzoru a stejně tak světlo zase zastaví, viz obrázek.

Změnami logické úrovně na kanálech/senzorech A a B a znalostí předchozího stavu můžeme zjistit jak směr otáčení, tak například o kolik se už motor otočil a podobně. V našem příkladu budeme mít řízení směru řešené jinak a enkodér budeme používat jen jako zpětnou vazbu o ujeté vzdálenosti.

Samotný motor se ovládá pomocí desky Sabertooth2x5. Tato deska je určena speciálně pro řízení motorů pomocí mnoha různých módů, my budeme používat mód 3 aneb „Simplified Serial“¹. Tento mód funguje jednoduše. Přes seriovou linku posíláme jednobajtové zprávy s číslem z intervalu 0–255:

- 0 znamená „oba motory stop“
- 1–127 ovládá levý motor následovně:
 - 1 je „naplno vzad“
 - 64 je „levý motor stop“
 - 127 je „naplno vpřed“
- 128–255 ovládá pravý motor (jsou to čísla pro levý motor, ke kterým je přičteno 127):
 - 128 je „naplno vzad“
 - 192 je „pravý motor stop“
 - 255 je „naplno vpřed“

Pokud tedy chceme rozjet oba motory zároveň, pošleme dva samostatné bajty. Nula je speciální a zastaví oba motory naráz.

A co tedy bude naším úkolem?

- Rozjet robota rovně
- Zatočit s ním
- Zastavit v reakci na překážku

¹Podrobnější informace k nalezení v dokumentaci, hledejte „Simplified Serial“: <https://www.dimensionengineering.com/datasheets/Sabertooth2x5.pdf>

6.1 Jedeme rovně

Jízda rovně má jednu výhodu: oběma motorům udělujeme stejnou rychlosť a jediné, co potřebujeme spočítat, je jak rychle a jak dlouho mají motory fungovat. K tomu budeme potřebovat dva údaje: rychlosť a vzdálenosť. Rychlosť si můžeme zvolit například tak, že ji budeme zadávat v procentuálním rozsahu -100 až +100. Záporná čísla nám umožní kromě rychlosti ovládat i směr. Vzdálenost může být v libovolných jednotkách, ovšem nejlepší budou centimetry.

Nastavení rychlosti bude vcelku jednoduché, stačí číslo z intervalu -100 až 100 přepočítat do intervalu 1 až 127 (pro druhý motor jen přičteme 127).

Vzdálenost bude trochu komplikovanější a poslouží nám k tomu zmiňované enkodéry (konkrétně jen jeden kanál) a něco, čemu se říká přerušení. Celý princip zase až tak složitý není. Enkodér vykoná za jednu otočku motoru (a tím i kola) nějaký konkrétní počet kroků, který je pro naši potřebu uložený v makru STEPS_ONE_CHANNEL. Pokud známe obvod kola (opět můžeme použít makro WHEEL_CIRCUIT), můžeme z těchto dvou hodnot vypočítat potřebný počet kroků pro ujetí konkrétní vzdálenosti.

Zpět k programu! Kostru najeznete v souboru `motors.cpp`. Naším úkolem bude spočítat, kolik kroků musíme ujet, následně odešleme kontrolní bajt a ve chvíli, kdy dosáhneme požadovaného počtu kroků, oba motory opět zastavíme.

```
void Motors::jed_rovne(int distance, int speed)
{
    int steps_to_go = 0/*TODO potrebne kroky*/;
    byte driving_byte;
    driving_byte = Motors::get_speed_from_percentage(speed);
    attach_interrupts();
    /*TODO odeslat kontrolní bajt do obou motoru*/
    /*Toto si vysvetlime mimo kod*/
    while(1)
    {
        if(steps_left == steps_to_go ||
           steps_right == steps_to_go)
            break;
    }
    /*TODO zastavit motory*/
    detach_interrupts();
}
```

Na první pohled vás možná překvapí nekonečný cyklus. Za prvé, nikde nejsou definované proměnné `steps_left` a `steps_right`. Za druhé, žádná proměnná se v cyklu nemění, takže by se tu program měl zacyklit nebo okamžitě skončit. A právě zde nám pomohou přerušení!

Proměnné `steps_left` a `steps_right` jsou deklarované jako globální proměnné s identifikátorem `volatile`. Možná jste o něm nikdy neslyšeli a pravděpodobně jste ho ani nikdy nepotřebovali. Slouží totiž k identifikaci takové proměnné, která může svou hodnotu změnit naprostě kdykoli, nehledě na právě běžící kód. Běžně se tento identifikátor nepoužívá a použití spolu s globální proměnnou, která je měněná v obslužné rutině přerušení – jednoduše v situaci, kdy se přerušení vyvolá – je jeden z velmi mála případů, kdy se jeho použití nevyhneme.

A tím jsme vlastně vysvětlili i druhou podivnost cyklu. Proměnné nejsou měněny v právě běžícím kódu, ale právě v obslužné rutině přerušení. Tedy v té části programu, která se vyvolá ve chvíli, když se na pinech, ke kterým jsou připojené enkodéry, detekuje nástupná hrana.

Doplňení zbytku kódů už pro vás bude hračka. Jen pozor, ovladač motorů komunikuje s Arduinem po jeho druhé seriové lince, `Serial1`!

Pokud vám náhodou něco nefunguje, můžete se podívat do ukázkové funkce níže nebo do referenční implementace:

```

void Motors::jed_rovne(int distance, int speed)
{
    int steps_to_go = (int)(distance/
        WHEEL_CIRCUIT*STEPS_ONE_CHANNEL);
    byte driving_byte;
    driving_byte = Motors::get_speed_from_percentage(speed);
    attach_interruptions();
    Serial1.write(driving_byte);
    Serial1.write(driving_byte+127);
    while(1)
    {
        if(steps_left == steps_to_go ||
            steps_right == steps_to_go)
            break;
    }
    Serial1.write(STOP_BYTE);
    detach_interruptions();
}

```

Přerušení v Arduinu

V základní části jsme si o přerušených řekli jen docela málo informací, hlavně jejich konkrétní použití jsme odbyli jen pár řádky s tím, že „to tak prostě funguje“. Tady tohle malé opomenutí napravíme.

Arduino Má k dispozici několik pinů, které umožňují, aby na nich bylo tzv. napojeno přerušení. Konkrétně u modelu Mega jsou to piny 2, 3, 18, 19, 20 a 21. Naše enkodéry (konkrétně jejich A kanály) dlí na pinech 2 a 3. Jak tedy řekneme Arduinu, že pokud se na pinech 2 nebo 3 objeví nástupná hrana (přechod 0 – 1), maj provést jakou akci?

Celý proces se skládá z několika kroků, které si postupně projdeme:

1. Vytvoření tzv. „obslužných rutin“
2. Připojení/povolení přerušení na daném pinu a uvedení obslužné rutiny, která se má provést
3. V této fázi se bude vykonávat kód, do kterého mají zasahovat přerušení
4. Odpojení přerušení ve chvíli, kdy již nejsou potřeba

Vytvoření obslužných rutin

Obslužná rutina není nic jiného než funkce, která obslouží přerušení. Jinak řečeno, která se provede, pokud je splněna podmínka. Nic jiného v tom nehledejte. Funkce by měla mít návratový typ void, protože stejně nemá kam co vracet, a také by měla být co nejkratší. Obslužná rutina přerušuje hlavní běh programu a tím ho zpomaluje. Hlavní program během vykonávání obslužné rutiny nemůže na nic reagovat. Extrémním příklad: představte si auto, které má zastavit v situaci, kdy před sebou uvidí překážku. Pokud ale bude obslužná rutina přerušení z předního senzoru moc dlouhá (bude například logovat na centrální server přes internet, ať jsme hodně extrémní), auto už třeba nestihne zastavit. Zpět do reality. Skutečná obslužná rutina pro pravý motor vypadá takhle:

```

void motor_right_interrupt_handler()
{
    steps_right++;
}

```

Jak vidíte, je skutečně krátká. Co se týče jména, je dobré značit obslužné rutiny jako „some_name_interrupt_handler“.

Povolení přerušení

Když už víme, co bude na přerušení reagovat, můžeme ho na daném pinu povolit. To se v případě Arduina děje funkcí `attachInterrupt(digitalPinToInterrupt(pin), handler, TYPE)`. Jako první parametr jde samotné číslo pinu, druhým parametrem je obslužná rutina a třetím je TYP přerušení. `RISING` znamená náběžnou hranu neboli přechod z 0 do 1. Kromě něj existuje třeba jeho opak, `FAILING`, a další.

```
attachInterrupt(digitalPinToInterrupt(right_A), motor_right_interrupt_handler, RISING);
```

Odpojení přerušení

Ve chvíli, kdy už přerušení nepotřebujeme, je dobré je odpojit. To zajistí, že se nebudou zbytečně provádět obslužné rutiny v době, kdy to není žádoucí. Konkrétně v případě našeho Trilobota může dojít k pohybu a změně signálu například při posunutí rukou. V takovou chvíli asi nemá smysl přerušovat program. V kódu stačí pouze oznámit, od jakého pinu přerušujeme přerušení:

```
detachInterrupt(digitalPinToInterrupt(left_A));
```

Poznámka: Kromě odpojení přerušení jsou ještě v našem případě nulovány volatilní proměnné, které s přerušením přímo nesouvisí. Samozřejmě se mohou nulovat i kdekoli jinde.

6.2 Zatáčíme!

Na zatáčení samotném by nemělo být nic těžkého. Dejme tomu, že pokud bude úhel kladný, budeme zatáčet doprava, pokud záporný, budeme zatáčet doleva. Druhým parametrem by asi mělo být, jak rychle se celá otočka odehraje.

Co se týče samotného otáčení, bud' se může Trilobot točit tak, že jedno kolo zablokuje a druhé se začne pohybovat, nebo se bude otáčet na místě: kola se budou točit obráceně. My si ukážeme jednodušší variantu, kde se bude točit jen jedno kolo.

Nejtěžší bude asi spočítat, kolik kroků má pohybující se kolo urazit, aby samotný robot opsal daný úhel. Není to zase až tak těžké. Chceme spočítat část kruhu, které přísluší úhel, který dostaneme jako první parametr funkce. Přepočet z centimetrů na kroky je už otázkou trojčlenky.

Sotva víme, o kolik má jedno z kol popojet, podle znaménka u úhlu se rozhodneme, které kolo to bude. A to je vše! Zbytek kódu bude velmi podobný tomu u jízdy rovně.

Kostra naší funkce může vypadat třeba takhle:

```
void Motors::zatoc(int angle, int speed)
{
    int steps_to_go = /*TODO pocet kroku, ktere musime urazit */;
    attach_interrupts();
    byte driving_byte;
    driving_byte = Motors::get_speed_from_percentage(speed);
    /*TODO zatoceni podle znamenka u parametru angle*/

    while(1)
    {
        if(steps_left == steps_to_go || steps_right == steps_to_go)
            break;
    }
}
```

```

    }
    /*TODO zastavit dany motor */
    detach_interruptions();
}

```

Návod: Pokud máte problémy s určením počtu kroků, můžete se inspirovat hotovou funkcí níže nebo se podívat do hotové implementace:

```

void Motors::zatoc(int angle, int speed)
{
    int steps_to_go = (int) (((2 * TURN_RADIUS * PI * angle)
        / 360) / WHEEL_CIRCUIT * STEPS_ONE_CHANNEL);
    attach_interruptions();
    byte driving_byte;
    driving_byte = Motors::get_speed_from_percentage(speed);
    Serial1.write(driving_byte);

    while(1)
    {
        if(steps_left == steps_to_go ||
            steps_right == steps_to_go)
            break;
    }
    Serial1.write(STOP_BYTE);
    detach_interruptions();
}

```

6.3 Spojujeme znalosti: Jak zastavit před překážkou?

Ted' už toho umíme dost, abychom mohli dát dohromady velmi jednoduchou verzi robota, který se bude vyhýbat překážkám. V konečném důsledku by mohl i projíždět bludištěm, ale k tomu nemá náš Trilobot zrovna ideální umístění senzorů. Pokud vás to ale zajímá, podívejte se do pokročilé části.

Naším cílem bude vytvořit robota, který při detekci překážky zatočí a pokusí se jí vyhnout. Celý projekt si rozdělíme do tří částí:

1. Upravení předchozích funkcí tak, aby robot nezastavoval
2. Pokud jste přeskočili kapitolu o měření vzdálenosti, implementace funkcí pro práci se senzory
3. Spojení předchozích dvou částí dohromady!

Úprava funkcí

Ve své podstatě jen odstraníme všechn kód, který se stará o zastavení. Jistě to zvládnete sami, pro ukázku přidávám i hotové řešení. Jelikož teď ale robot nezastaví, budete potřebovat funkci, která mu zastavení nakáže. Ta bude sama o sobě také velmi jednoduchá:

```

void Motors::move(int speed)
{
    byte control_byte = get_speed_from_percentage(speed);
    Serial1.write(control_byte);
    Serial1.write(control_byte+127);
}

```

```

void Motors::turn_left(int speed)
{
    byte control_byte = Motors::get_speed_from_percentage(speed);
    Serial1.write(control_byte+127);
}

void Motors::turn_right(int speed)
{
    byte control_byte = Motors::get_speed_from_percentage(speed);
    Serial1.write(control_byte);
}

void Motors::stop()
{
    Serial1.write(STOP_BYTE);
}

```

Měříme vzdálenost – zkráceně

Pokud jste přeskočili sekci o měření, zde je jen pár základních informací: Trilobot má celkem tři senzory vzdálenosti: dva ultrazvukové a jedna infračervený. Všechny tři senzory si můžete naprogramovat buď sami podle kapitoly nebo využijte předpřipravených funkcí. My budeme používat senzor SRF-08, komunikující přes I²C sběrnici. Jeho použití je velmi snadné, zde jen krátké shrnutí:

- Před každým měřením je třeba zavolat funkci `set_measurement(unit)`, kde parametr `unit` bude pravděpodobně makro `CM`.
- Pro získání dat stačí zavolat funkci `get_distance()`, která vrátí naměřenou hodnotu v daných jednotkách.

Uhýbáme překážkám

Samotný kód je teď ve své podstatě velmi jednoduchý. Robot se rozjede kupředu a ve chvíli, kdy před sebou uvidí překážku, zatočí. Následně bude opět pokračovat rovně. Fantazii se meze nekladou, takže Trilobot může pořád uhýbat doprava, může střídat strany i úhel otočky... Cokoli, co vás napadne.

Pro ukázkou si spolu můžeme udělat verzi, Trilobot u překážky vždy zatočí doprava o úhel 90 stupňů. Pokud překážky rozmístíte správně, začne Trilobot jezdit ve čtverci:

```

void loop()
{
    /*Nechceme, aby pro záčtek jel Trilobot moc rychle.*/
    motors->(30);
    while(1)
    {
        if(srf08->get_distance() < 10)
        {
            stop();
            turn_right(90);
            /*Vyskocíme z cyklu, abychom se opět rozjeli*/
            break;
        }
    }
}

```

Kapitola 7

Orientujeme se v prostoru – kompas, gyroskop a akcelerometr

Předešlé kapitoly byly věnované mnoha různým senzorům a částečně Trilobotovi, z nichž některé umožnily Trilobotovi alespoň částečně zjistit, kde se nachází. Enkodéry poskytovaly informaci o tom, kolik Trilobot ujel, senzory vzdálenosti ho varovaly před překázkou. Stále to ovšem byly dosti relativní ukazatele polohy, které se nedaly vztáhnout na nějakou souřadnicovou síť, nemluvě o zásahu zvenčí – Trilobot například neměl šanci poznat, zda se skutečně pohybuje, nebo se mu kola protáčejí ve vzduchu. A právě ke zjišťování absolutní polohy bude sloužit poslední senzor: jednotka minIMUv3, kombinující gyroskop, akcelerometr a magnetometr¹.

7.1 Co je to IMU?

IMU (angl. inertial measurement unit) slouží k určování polohy ve 3D světě a je velmi důležitá například pro kvadrokoptéry a jiné létající stroje. IMU využívá data ze tří senzorů: gyroskopu, akcelerometru a magnetometru.

Gyroskop měří úhlovou rychlosť ve třech osách (X, Y, Z). Úhlovou rychlosť většinou nevyužíváme přímo, ale můžeme ji integrovat v čase, čímž získáme natočení robota ve třech osách. Problém je, že běžně dostupné gyroskopy trpí kromě obecné chyby měření i tzv. driftem. Ten se zjednodušeně projevuje tak, že při „obyčejné“ integraci se bude úhel natočení měnit i za situace, kdy bude robot v klidu. Ke kompenzaci driftu se využívají údaje z ostatních senzorů – akcelerometru a magnetometru – čímž vzniká AHRS – poziciální systém. Te je už silně nad rámec tohoto tutoriálu.

Akcelerometr, jak již název napovídá, měří akceleraci, tedy zrychlení, opět ve všech třech osách. Některé možná napadlo data z akcelerometru integrovat podle času, čímž bychom získali rychlosť. To má ovšem několik problémů. Akcelerometr v IMU, které používá Trilobot, není natolik přesný, aby udávané údaje dávaly smysl. Odchylka při měření je už za několik vteřin tak vysoká, že jsou hodnoty rychlosti naprostě nepoužitelné. Dalším problémem je, že akcelerometr nerozezná situaci, kdy se robot pohybuje rovnoramenně přímočara a kdy stojí (v obou případech na něj působí nulové zrychlení ve směru pohybu). V konečném důsledku se akcelerometr používá pro dva účely: měření svislého směru a jako jeden ze vstupů do AHRS.

Posledním ze senzorů je magnetometr. Zjednodušeně řečeno, měří intenzitu magnetického pole, čímž můžeme poměrně jednoduše určit, kde je sever.

7.2 Tvoříme kompas

Začneme něčím jednoduchým, třeba kompasem/ukazatelem severu. Ukážeme si rovnou dvě možnosti, jak takový sever najít. První bude extrémně jednoduchá, ale nebude úplně přesná. Druhá bude přesnější, ale zase se budeme muset ponořit do vektorové aritmetiky.

¹My si ukážeme jen použití každého senzoru samostatně

7.2.1 Kompas 1

Naším úkolem bude doplnit funkci `float Magnetometer::kompas_1()`. Cíle jsou následující:

- Přečteme data z magnetometru
- Zahodíme složku Z (použijeme jen data pro osu x a y – vodorovnou plochu)
- Vypočítáme úhel mezi vektorem dat z magnetometru a vektorem `front`. Bude nám stačit úhel, který vrací funkce `acos()` – od 0 do 180. sice nebudeme vědět, jestli se otáčíme na západ nebo na východ, ale nám nyní nevadí.

Kostru si můžeme prohlédnout níže, nachází se v souboru `magnetometer.cpp`:

```
float Magnetometer::kompas_1()
{
    /*TODO ziskat data intenzity z magnetometru do prom. nize*/
    vector<int16_t> mag_values = {0,0,0};
    /*Udava, jaký směr je "predek" (osa X - první pozice).
     * Jelikož je pro lepsi přístup senzor "opacne",
     * je hodnota zaporna.*/
    vector<int> front = {-1,0,0};

    mag_values.x -= (mag_min.x + mag_max.x) / 2;
    mag_values.y -= (mag_min.y + mag_max.y) / 2;
    /*TODO vynulovat souřadnici Z*/
    vector<float> norm_mag_values;
    norm_mag_values = vector_normalize(mag_values);

    float angle = 0; /*TODO spočítat uhel dle navodu*/
    return angle;
}
```

Celý postup práce bude velmi jednoduchý:

- Data přečteme pomocí:

```
this->get_intensity();
```

- Vynulování souřadnice je prostě... vynulování souřadnice :)

```
mag_values.z = 0;
```

- Úhel mezi vektory známe ze střední školy: $\alpha = \arccos\left(\frac{\mathbf{mag_values} \cdot \mathbf{front}}{|\mathbf{mag_values}| \cdot |\mathbf{front}|}\right)$
- V kódu stačí použít funkce `vector_dot(u, v)` pro skalární součin a `vector_abs(v)` pro velikost vektoru. Arkus kosinus je jednoduše `acos()`, ovšem hodnotu vrací v radiánech – stačí vynásobit výsledek zlomkem $\frac{180}{\pi}$

A výsledek může vypadat třeba takto:

```
float Magnetometer::kompas_1()
{
    vector<int16_t> mag_values = this->get_intensity();
    /*Udava, jaký směr je "predek" (osa X - první pozice).
     * Jelikož je pro lepsi přístup senzor "opacne",
     * je hodnota zaporna.*/
}
```

```

vector<int> front = {-1, 0, 0};

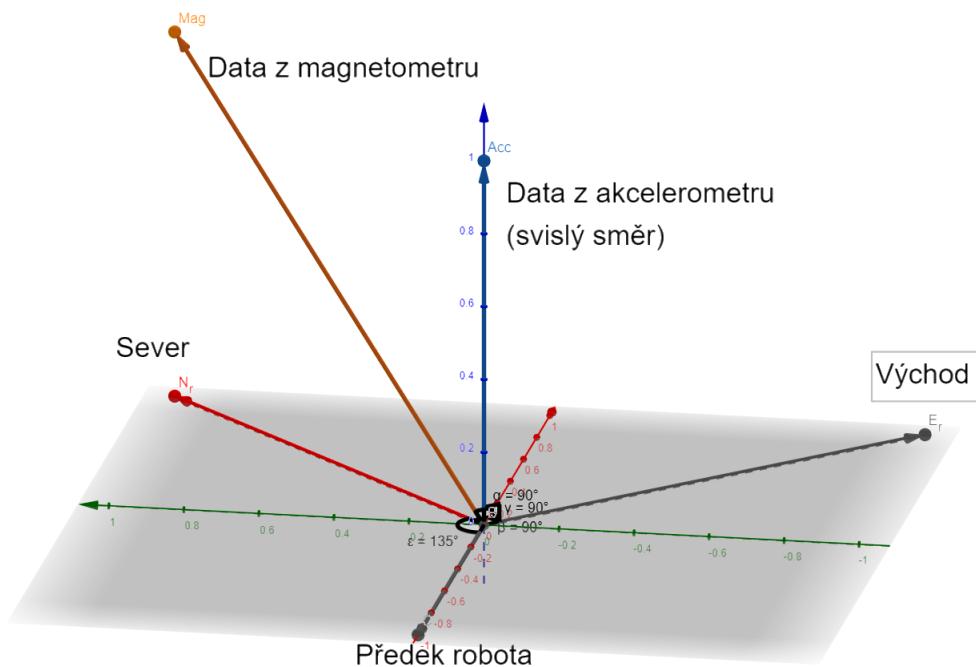
mag_values.x -= (mag_min.x + mag_max.x) / 2;
mag_values.y -= (mag_min.y + mag_max.y) / 2;
mag_values.z = 0;
vector<float> norm_mag_values;
norm_mag_values = vector_normalize(mag_values);

float angle = acos(vector_dot(norm_mag_values, front)
    /
    (vector_abs(norm_mag_values)*vector_abs(front)))
    * 180 / PI;
return angle;
}

```

7.2.2 Kompas 2

První kompas nefunguje zle, ale ve 3D prostoru začne ztrácat dech. Proto vytvoříme poněkud lepší verzi, která bude zohledňovat i třetí rozměr.



Obrázek 7.1: Vektory použité pro určení severu a úhlu mezi ním a předkem robota

Na obrázku vidíme několik vektorů, vstupem jsou ten modrý a oranžový – data z akcelerometru a magnetometru – a směr, kterým je předek robota. Pomocí vektorového součinu dat z magnetometru a akcelerometru (nebojte se, máme na to funkci `vector_cross()` – jen nesmíte poplatit pořadí) získáme vektor, leží v rovině a zároveň je kolmý na směr severu.

Proč? Vektorový součin nám vrací vektor kolmý na oba vektory vstupní. Víme, že data z akcelerometru nám udávají svislý směr – kolmý vektor tedy bude ležet v horizontálním směru nebo v rovině. Výstup magnetometru nám zase udává směr severu – a kolmý vektor na sever je vektor směrem na východ (nebo na západ, ale to bychom museli prohodit vektory při výpočtu vektorového součinu).

Sever už získáme jednoduše – třeba jako vektorový součin dat z akcelerometru a východu.

Ted' už zbývá jen poslední krok – určit úhel mezi vektorem, který ukazuje na sever, a vektorem, udávajícím předešek robota. Toho docílíme například vzorcem:

$$\alpha = \arccos\left(\frac{\text{vector_dot}(\text{sever}, \text{predek})}{\text{abs}(\text{sever}) * \text{abs}(\text{predek})}\right)$$

Poznámka: `vector_dot()` je funkce pro skalární součin vektorů, `abs()` je absolutní hodnota vektoru.

Tak a ted' po vyčerpávající teorii můžeme přistoupit k praxi. V souboru `namagnetometer.cpp` na vás čeká kostra funkce `float Magnetometer::kompas_2(Accelerometer *accel)`:

```
float Magnetometer::kompas_2(Accelerometer *accel)
{
    /*Udava, jaký směr je "predek" (osa X - první pozice).
     * Jelikož je pro lepsi přístup senzor "opacne",
     * je hodnota zaporna.*/
    vector<int> front = {-1, 0, 0};
    vector<int16_t> mag_values = this->get_intensity();
    vector<float> accel_values = accel->get_acceleration();
    /** odčtení offsetu */
    mag_values.x -= (mag_min.x + mag_max.x) / 2;
    mag_values.y -= (mag_min.y + mag_max.y) / 2;
    mag_values.z -= (mag_min.z + mag_max.z) / 2;

    vector<float> E; /* Východ */
    vector<float> N; /* Sever*/
    /* TODO: spočítat sever podle navodu */

    float angle = 0; /*TODO uhel mezi vektory N a front*/
    return angle;
}
```

I když se ted' asi děsíte vektorového součinu, můžeme využít síly počítače (a matematické knihovny) a pouze říct, že chceme vektorový součin dvou vektorů:

```
E = vector_cross(mag_values, accel_values);
```

Nyní máme vektor, ukazující na východ. stejným způsobem získáme i sever:

```
N = vector_cross(accel_values, E);
```

Posledním krokem je určit úhel mezi vektory `front` a `N` podle vzorce výše. Pokud se vám nechce skalární součin vypisovat, můžete použít funkci `vector_dot()`, která vrátí výsledek.

A to je vše! Funkce `float Magnetometer::kompas_2(Accelerometer *accel)` může vypadat třeba takto:

```
float Magnetometer::kompas_2(Accelerometer *accel)
{
    /*Udava, jaký směr je "predek" (osa X - první pozice). Jelikož je pro lepsi přístup senzor "opacne",
     * je hodnota zaporna.*/
    vector<int> front = {-1, 0, 0};

    vector<int16_t> mag_values = this->get_intensity();
    vector<float> accel_values = accel->get_acceleration();

    // subtrahuj (průměr min a max) z magnetometru čtení
    mag_values.x -= (mag_min.x + mag_max.x) / 2;
```

```

mag_values.y -= (mag_min.y + mag_max.y) / 2;
mag_values.z -= (mag_min.z + mag_max.z) / 2;

vector<float> E; /* Vychod */
vector<float> N; /* Sever*/
E = vector_cross(mag_values, accel_values);
E = vector_normalize(E);
N = vector_cross(accel_values, E);
N = vector_normalize(N);

float angle = acos(vector_dot(N, front)
                    /(vector_abs(N)*vector_abs(front)))
                    * 180/PI;
return angle;
}

```

Poznámka na závěr: funkce `acos()` vrací výsledek pouze z intervalu $(0, 180)$ stupňů. Pokud bychom tedy chtěli hodnotu od 0 do 360 , musíme ještě rozhodnout, jestli má být výsledný úhel větší nebo menší než 180 stupňů.

7.3 Detektor otřesů

Ted' se můžeme přesunou k dalšímu senzoru – gyroskopu. jak už jsme si řekli, měřit natočení jen s pomocí gyroskopu není úplně snadné, proto si vytvoříme něco jednoduššího – detektor otřesů.

Detektory otřesů se používají na spoustě míst, třeba:

- Seismometry měřící zemětřesení
- Kontrola stavu strojů: pokud se stroj moc otřásá, je pravděpodobně někde chyba (uvolněná hřídel, opotřebený díl...)
- Kontrola křehkých zásilek spolu s detektory nárazu

Naším úkolem bude takový měřič otřesů naprogramovat a ve chvíli, kdy s Trilobotem (přiměřeně) zařešeme, rozsvítíme v `loop()` diodu. Princip bude jednoduchý: načteme si dvoje data s nějakým časovým rozestupem a zjistíme, jak hodně se změnila. Pokud bude změna velká, interpretujeme to jako zatřesení.

Kostru můžeme vidět níže (v `k=odu` ji najdete v `gyroscope.cpp`):

```

bool Gyroscope::detektor_otresu(float threshold)
{
    /* TODO ziskat dvoje data s urcitou casovou prodlevou
     - treba 20 ms */
    vector<float> first = {0,0,0};
    vector<float> second = {0,0,0};
    vector<float> shake = {0,0,0};

    /*Ziskame jejich rozdíl */
    shake.x = abs(first.x-second.x);
    shake.y = abs(first.y-second.y);
    shake.z = abs(first.z-second.z);

    /* TODO pokud bude nektery z rozdilu v ose x,y,z vetsi nez
     threshold - hranice "klidu", vratime true, jinak false */
    return false;
}

```



Obrázek 7.2: Senzor nárazu na balení jedné nejmenované 3D tiskárny :)

Funkci doplníme jednoduše: data z gyroskopu přečteme funkcí `this->get_angular_velocity()` a porovnávání není nic těžkého. Výsledek můžeme vidět níže:

```
bool Gyroscope::detektor_otresu(float threshold)
{
    vector<float> first = this->get_angular_velocity();
    delay(20);
    vector<float> second = this->get_angular_velocity();
    vector<float> shake = {0, 0, 0};
    shake.x = abs(first.x - second.x);
    shake.y = abs(first.y - second.y);
    shake.z = abs(first.z - second.z);
    if(shake.x > threshold || shake.y > threshold
       || shake.z > threshold)
        return true;
    else
        return false;
}
```

7.4 Detektor nárazu

Detektor otřesů doplníme ještě detektorem nárazu – ten se používá například u přepravních služeb, kdy hlídá, zda se k balíku všichni chovali dobře. V tomhle případě většinou senzory neobsahují MCU a akcelerometr, ale jedná se o mechanická zařízení, která se deformují při daném zrychlení (třeba praskne ampulka s barvou).

Na závěr tutoriálu to nebudeme vůbec komplikovat – jediným naším úkolem bude přečíst data z akcelerometru a pokud některá z hodnot přesáhne hranici (`threshold`), podáme o tom zprávu – třeba rozsvítíme LEDku.

Kostra je k nalezení níže nebo v souboru `accelerometer.cpp`:

```
bool Accelerometer::detektor_narazu(float threshold, bool reset)
{
    /* static promenna si zachova ulozenou hodnotu i pri dalsim
    volani funkce */
    static bool impact = false;
    if(reset) /* pri resetu vyresetujeme i pamet dopadu */
    {
        impact = false;
    }
    else
    {
        if(shake.x > threshold || shake.y > threshold
           || shake.z > threshold)
            impact = true;
        else
            impact = false;
    }
}
```

```

    }
    /* TODO precteme data*/
    vector<float> a = {0,0,0};
    /* TODO pokud nekterá z os prekročí threshold, nastavíme
     * impact na true */
    return impact; /* vracíme aktuální hodnotu impact */
}

```

- Čtení hodnoty je jednoduché:

```
vector<float> a = this->get_acceleration();
```

- No a porovnávat jednotlivé složky vektoru zvládneme taky – jen nesmíme zapomenout u osy Z přičítat k thresholdu 10 (neboli gravitační zrychlení). Také se nám vyplatí použít funkci `abs()` – zajistí, že se náraz zaznamená i pokud příjde ze druhé strany

```

/* 10 +-= g*/
if(... data.y > threshold || data.z > threshold+10)
    {/* Udelej...*/}

```

Výsledné řešení může vypadat třeba takhle. V `main.ino` stačí podle vrácené hodnoty rozsvítit nebo zhasnout LED.

```

bool Accelerometer::detektor_narazu(float threshold, bool reset)
{
    static bool impact = false;
    if(reset)
    {
        impact = false;
    }
    vector<float> a = this->get_acceleration();
    /* +10 u osy z je kompenzace gravitace */
    if(abs(a.x) > threshold || abs(a.y) > threshold
       || abs(a.z) > threshold+10)
    {
        impact = true;
    }
    return impact;
}

```

Kapitola 8

Závěr

A jsme u konce tohoto tutoriálu. Prošli jsme si vše od teoretického základu – co je to Arduino, jak se používá, jak spolu může komunikovat více desek a podobně – přes praktický úvod do programování Arduina až po konkrétní příklady, jak zprovoznit některé senzory robota Trilobot.

Doufám, že se vám tutoriál líbil a přinesl vám alespoň nějaké nové informace. Pokud budete mít jakékoli nápady na zlepšení, ozvěte se s nimi prosím na e-mail xberan43@stud.fit.vutbr.cz. Budu moc rád za každou zpětnou vazbu!