Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Kybernetika a řídící technika

BAKALÁŘSKÁ PRÁCE

Hardware in the loop simulace malé robotické platformy

Autor: **Vojtěch Polívka**

Vedoucí práce: **Ing. Miroslav Flídr, Ph.D.**

Akademický rok: **2014/2015**

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci zpracovanou na závěr studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím odborné literatury a pramenů, jejichž úplný seznam je její součástí.

V Plzni dne 14. května 2015

podpis

Anotace

Hardware-in-the-loop simulace se hojně využívá v průmyslu při vývoji a testování hardwarových komponent. Tato práce popisuje možnosti využití hardware-in-the-loop simulace při programování malých robotů pro studijní účely. Součástí je návrh, popis a implementace kompletního simulačního prostředí, které umožňuje touto technikou simulovat malé robotické platformy v reálném čase pomocí obyčejného kancelářského počítače a vývojové desky Arduino UNO.

Klíčová slova:
hardware in the loop, hardware-in-the-loop, HIL, simulace, model, Arduino, UNO, V-REP, robot, robotická platforma, generování programu

Abstract

Hardware-in-the-loop simulation is widely used during research phases and testing in all kinds of industries. This paper adresses the possibility of using this technique for programming small robots for study purposes. The paper contains the design, description and implementation of complete simulation environment which enables hardware-in-the-loop real-time simulation for small robotic platforms using regular desktop PC and development board Arduino UNO.

Keywords:
hardware in the loop, hardware-in-the-loop, HIL, simulation, model, Arduino, UNO, V-REP, robot, robotic platform, code generation

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Miroslavu Flídrovi za zapůjčení veškerého potřebného hardwaru a cenné rady k vypracování této práce.

Obsah

1 HIL simulace **2**

1.1 Výhody 2

1.2 Součásti 3

1.3 Realizace 4

2 Komunikační rozhraní **5**

2.1 Předpoklady 5

2.2 Mezičlánek 5

2.3 Arduino UNO 7

2.3.1 Programování mikrokontroléru 7

2.3.2 Hardwarové rozhraní 7

2.3.3 Přerušení 8

2.4 Průběh komunikace 8

2.4.1 Hlavní smyčka programu 9

2.4.2 Obsluha sériové linky 9

2.4.3 Přerušení časovačem 10

2.4.4 Externí přerušení 10

2.5 Komunikační protokol 10

2.5.1 Zpráva protokolu 10

2.5.2 Dekódování protokolu 11

2.5.3 Kódování protokolu 12

2.5.4 Analýza komunikace 13

3 Simulace prostředí **14**

3.1 Volba simulátoru 14

3.2 Simulační software V-REP 15

3.3 Klientská aplikace 16

3.3.1 Volba programovacího jazyka 17

3.3.2 Komunikace s V-REP 17

3.3.3 Komunikace s mezičlánkem 18

3.3.4 Synchronizace komunikace 18

3.3.5 Datový model 20
3.3.6 Generátor programu pro mezičlánek 23
3.3.7 Grafické uživatelské rozhraní 25

4 Ukázková simulace robota 28
4.1 Popis robota 28
4.2 Popis úlohy 28
4.3 Zpracování 29
4.3.1 Model robota 29
4.3.2 Programování robota 31
4.3.3 HIL simulace 32

A HIL Simulation Plugin I
A.1 Minimální hardwarové nároky I
A.2 Pokyny k instalaci I
A.3 Pokyny k použití II
A.4 Základní knihovna aktuátorů a senzorů II
A.4.1 LED III
A.4.2 Stejnosměrný elektrický motor III
A.4.3 Koncový spínač V
A.4.4 Dálkový senzor HC-S04 VI

B Zdrojový kód IX
B.1 Program řídící jednotky robota IX
B.2 Ukázka vygenerovaného programu pro mezičlánek XII

Literatura XV

Úvod

S malými mobilními či stacionárními roboty se setká během svého studia prakticky každý student kybernetiky. Obvykle je jeho úkolem naprogramovat řídící jednotku předem sestaveného robota nebo i částečně robota sestavit. Při programování řídící jednotky je potřeba robota připojit k PC, nahrát kód a posléze kód otestovat v reálném prostředí, kde má robot fungovat. Motivací této práce je vyvinout nástroje, které tento vývojový cyklus zrychlí a zjednoduší.

Hlavním úkolem je tedy implementovat tzv. hardware-in-the-loop simulaci, což je technika hojně využívaná při testování a vývoji reálných řídících systémů. Tím, že jsou v modelu systému některé prvky reprezentovány skutečným hardwarem, zavádíme do simulace potřebnou míru komplexity, která může být pro zdárné navržení řízení klíčová. Zároveň tím vývoj programu můžeme výrazně urychlit.

V našem případě budeme v PC simulovat robota a prostředí, ve kterém se nachází. Testovaným hardwarem v systému bude řídící jednotka, což může být např. mikrokontrolér. Cílem práce je navrhnout způsob řešení a popsat jeho implementaci.

Práce se skládá z několika logických celků řazených od hardwarové vrstvy po softwarovou. Po úvodní části, která se zabývá hardware-in-the-loop simulací obecně, následuje popis připojení mikrokontroléru k simulaci, dále část o simulaci robota a jeho prostředí v PC a nakonec ukázka řízení a simulace konkrétního robota.

Kapitola 1

HIL simulace

Reálné systémy jsou tvořeny soustavou hardwarových zařízení, která mezi sebou přenášejí informace nebo energii prostřednictvím různých datových protokolů či elektrických signálů. V kybernetice nás zajímá řízení těchto systémů, což se v praxi provádí pomocí regulátorů nebo digitálních řídících systémů. Úlohou inženýra může být navrhnout určitou část hardwarového vybavení (např. aktuátor, senzor nebo řídící člen). Při řešení této úlohy se často využívá počítačová simulace.

Hardware-in-the-loop simulace (dále „HIL simulace") je technika, při které je část hardwarového vybavení zapojena do počítačové simulace systému [4]. HIL simulace tedy integruje fyzická zařízení do virtuálního modelu a tvoří tím uzavřenou řídící smyčku. To přináší mnoho výhod, které jiné testovací mechanismy nemají.

1.1 Výhody

HIL simulace se využívá při vývoji i testování v mnoha průmyslových odvětvích. Oproti testování v reálných systémech umožňuje:

- snížit náklady,
- zkrátit dobu vývoje,
- zlepšit bezpečnost.

Většinu těchto výhod přináší i klasické modelování pomocí počítačových modelů, to ale neumožňuje testování skutečných hardwarových komponent jako HIL simulace. Například při vývoji a testování brzdového systému ABS můžeme použít skutečnou rozhodovací jednotku a její vstupy a výstupy simulovat a zobrazovat na PC. Tím zamezíme opotřebování skutečných brzdových disků a snížíme tak náklady, nicméně kvalita testování bude blízká testování na reálných vozidlech.

V průmyslu je samozřejmě kladen velký důraz na časovou efektivitu a HIL simulování nám může pomoci výrazně zkrátit vývojový cyklus. To uvítáme obzvlášť

v případě, kdy uvedení reálného systému do počátečního stavu zabere dlouhý čas, nebo je omezena jeho dostupnost. Při HIL simulaci může hardwarové součásti vyvíjet najednou i několik týmů, které by jinak musely sdílet svůj přístup k systému.

Bezpečnostní hlediska se musí zohledňovat ve všech oblastech průmyslu. Velmi důležité jsou třeba v jaderném průmyslu, kde se na reálném systému testovat prakticky nedá. Po sestavení virtuálního modelu skutečného systému můžeme řídící části hardwaru vyvíjet pomocí HIL simulace bez bezpečnostních rizik.

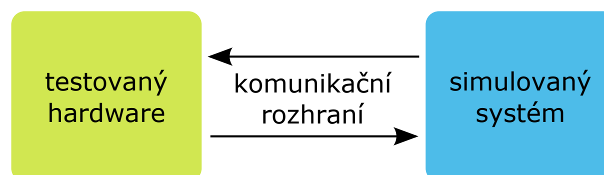
Některá experimentální řešení také nemusí být za daných podmínek proveditelná, přesto však může být přínosné taková řešení implementovat a simulovat. Tým pracovníků může také navrhovat a experimentovat s různými částmi hardwaru, i když zbytek systému ještě není realizován. To zvyšuje flexibilitu při plánování vývoje.

HIL simulace se může rovněž uplatnit ve výuce, kde umožňuje studentům bezpečně experimentovat se systémy, které jsou jim z bezpečnostních či finančních důvodů nedostupné. Na tuto oblast je tato práce zaměřena.

1.2 Součásti

Simulační prostředí HIL (dále obvykle jen „simulační prostředí“) se skládá ze tří hlavních částí: testovaného hardwaru, simulovaného systému a komunikačního rozhraní, které tyto dvě části spojuje (Obrázek 1.1).

Testovaným hardwarem je obvykle řídící jednotka realizovaná např. mikrokontrolérem, může to být však i celý subsystém skládající se z více hardwarových částí. Testovaný hardware má vstupy a výstupy, se kterými v reálné situaci komunikuje s okolním prostředí. V HIL simulaci musíme tyto vstupy a výstupy emulovat generováním patřičné signálové odezvy.



Obrázek 1.1: Části simulačního prostředí

Simulovaný systém modeluje prostředí, ve kterém by se v reálném světě testovaný hardware normálně nacházel. Existují různé nároky na přesnost a detailnost modelovaného prostředí. Pokud hardware např. reaguje jen na jednoduché podněty, může jako simulovaný systém posloužit i generátor signálů. Pokud jsou ale vstupní signály komplikovanější, např. obrazový vstup, je nutno simulovat prostředí

co nejvěrněji včetně simulace světla, pohybu apod. Výstupy hardwaru mohou klást na simulovaný systém podobné požadavky.

Úkolem **komunikačního rozhraní** je propojit simulovaný systém s testovaným hardwarem pokud možno tak, aby se přenos informace co nejméně lišil od reality. Správně navržené simulační prostředí by mělo umožnit testovaný hardware po otestování připojit do reálného systému, kde by měl fungovat beze změny. Nároky jsou tedy kladeny na rychlost a přesnost přenosu informace oběma směry.

1.3 Realizace

V následujícím textu budou analyzovány všechny výše uvedené části HIL simulačního prostředí a způsob jejich realizace. Důraz je kladen především na snadné použití a přívětivé uživatelské rozhraní s ohledem na zaměření na výuku. Bude prezentován ucelený způsob, jak HIL simulaci využít k usnadnění programování malých robotických platforem pro studenty nebo hobby amatéry. Z tohoto důvodu se práce zaměřuje na cenově dostupné a dobře dokumentované komponenty. Cílem práce není implementovat simulační prostředí, které by se dalo nasadit v průmyslu, a to kvůli náročnosti takového problému.

Kapitola 2

Komunikační rozhraní

Aby se dal testovaný hardware zapojit do virtuální simulace, musí s ní nějakým způsobem komunikovat. Počítač, ve kterém simulace probíhá, ale nemá potřebné hardwarové vstupy ani výstupy. Proto je nutno prozkoumat možnosti jejich zprostředkování.

V této kapitole je podrobně popsána volba komunikačního mezičlánku a jeho vlastnosti, poté jsou specifikovány možnosti a způsoby jeho naprogramování. Nakonec je popsán komunikačním protokol, kterým budou informace přenášeny.

2.1 Předpoklady

Komunikační rozhraní mezi PC a testovaným hardwarem je důležitá část simulačního prostředí HIL, protože na něm závisí kvalita simulace. Rozhraní musí být přizpůsobeno určitému typu hardwaru, a přestože má tato práce za cíl pojmout co nejširší škálu různých kontrolerů, předpokládá se, že testovaný hardware:

- komunikuje se vstupy pomocí napěťových digitálních signálů,
- komunikuje s výstupy pomocí napěťových digitálních, analogových a PWM signálů.

Tyto vlastnosti jsou společné pro velkou část hardwarového vybavení použitého v průmyslu. Je však nutno poznamenat, že návrh komunikačního rozhraní zcela zanedbává přenosy energie a soustředí se pouze na přenos informace.

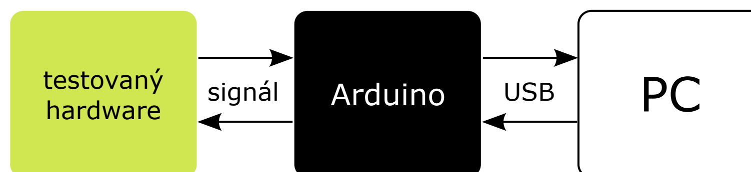
2.2 Mezičlánek

Osobní počítač nemá napěťové vstupy ani výstupy pro přenos signálů, tudíž je přímá komunikace testovaného hardwaru s PC nemožná. Navíc operační systém neumožňuje odezvu v reálném čase, takže by nemohl včas reagovat na některé signálové podněty.

Proto je potřeba do systému zařadit prvek, který zprostředkuje komunikaci na hardwarové vrstvě. Jeho úkolem je překládat datové signály přicházející z PC na napěťové signály a obdobně v opačném směru.

Obrázek 2.1: Arduino UNO Rev3

Jako tento prvek byla zvolena deska Arduino UNO Rev3 (Obrázek 2.1). Jedná se o vývojovou platformu pro programování AVR mikroprocesorů, která má jak hardwarové vstupy a výstupy, tak USB konektor pro připojení k počítači. Proto může v simulačním prostředí HIL tvořit chybějící mezičlánek (Obrázek 2.2).



Obrázek 2.2: Mezičlánek

Existuje celá řada vývojových desek, které Arduino UNO překonávají ať již ve výpočetním výkonu, v počtu hardwarových vstupů a výstupů či v možnostech připojení k PC. Velkou výhodou tohoto mikrokontroléru je však jeho nízká cena, kvalitní dokumentace a velká rozšířenost v cílové skupině této práce. HIL simulační prostředí je navíc navrženo tak, aby bylo možno implementovat i jiné typy mezičlánků bez nutnosti velkých zásahů do programového kódu.

2.3 Arduino UNO

V dalších částech budou popsány funkce Arduina, které byly využity při navrhování simulačního prostředí.

2.3.1 Programování mikrokontroléru

Hlavní výpočetní jednotkou Arduina je mikrokontrolér ATmega328[1]. Jednotka může být programována několika způsoby:

1. přímé programování mikrokontroléru pomocí ICSP[2],
2. externím hardwarovým programátorem s využitím přednahraného bootloaderu,
3. pomocí softwaru Arduino IDE přes emulovanou sériovou linku.

V navrženém simulačním prostředí je využíván výhradně třetí způsob, jelikož je mezičlánek vždy k PC připojen. Je to pro uživatele tedy nejsnazší cesta.

2.3.2 Hardwarové rozhraní

Arduino UNO má 14 pinů, které mohou být využity pro vstup a výstup digitálního napěťového signálu. V simulačním prostředí HIL bude k pinům připojen testovaný hardware pomocí spojovacích kabelů. Celkově je tedy teoreticky možné připojit maximálně 14 různých napěťových signálů.

Některé piny mají kromě přenosu digitálního napěťového signálu tyto speciální funkce [1]:

- pin 0 (RX) resp. 1 (TX) přijímá resp. vysílá data po sériové lince. Tyto piny jsou propojeny s korespondujícími piny na čipu ATmega16U2;
- piny externího přerušení 2 a 3, mohou být konfigurovány k detekci příchozí vzestupné, sestupné hrany nebo obojího;
- piny 3, 5, 6, 9, 10 a 11 umožňují PWM výstup;
- piny 10, 11, 12, 13 se vyžívají při SPI komunikaci;
- k pinu 13 je připojena LED;
- piny A0 až A5 se dají využít jako analogové vstupy s 10 bitovým A/D převodníkem.

[1]Výrobce Atmel Corporation
[2]In-Circuit Serial Programming

Zapisování a čtení digitálního signálu lze provádět funkcemi `digitalRead()`, `digitalWrite()`. Vždy je třeba specifikovat pin, na který je zápis prováděn, a hodnotu signálu.

Čtení analogového signálu lze na pinech A0 až A5 provádět funkcí `analogRead()`. Zápis analogového signálu ale možný není, jelikož Arduino UNO nemá vestavěný D/A převodník.

Čtení PWM signálu lze provádět pomocí hardwarového přerušení a měření doby pulsu. K tomu slouží funkce `pulseIn()`, která vrací délku pulsu v milisekundách. Implementace této metody však využívá funkce `delay()`, která na určitou dobu může blokovat vykonávání programu. To je v mezičlánku nežádoucí. Místo této funkce čtení PWM signálu probíhá pomocí tzv. Pin Change Interruptu, který reaguje na změnu hodnoty daného portu. Tato událost se zaznamená do externích proměnných a změří se v mikrosekundách délka pulsu. Zápis PWM signálu se dá provést funkcí `analogWrite()` v rozmezí 0 - 255 na pinech, které to umožňují.

2.3.3 Přerušení

Přerušení (angl. *interrupt*) je speciální stav, do kterého se mikrokontrolér dostane, pokud se naplní jedna z předem definovaných podmínek. Slouží k obsluze asynchronních událostí, aby nebylo nutno neustále zjišťovat stav událostí v bloku `loop()`.

Pokud přerušení nastane, procesor dokončí právě prováděnou atomickou operaci a předá řízení do tzv. obsluhy přerušení (angl. *Interrupt Service Routine*). Po obsloužení přerušení se naváže vykonávání operací tam, kde naposled skončilo.

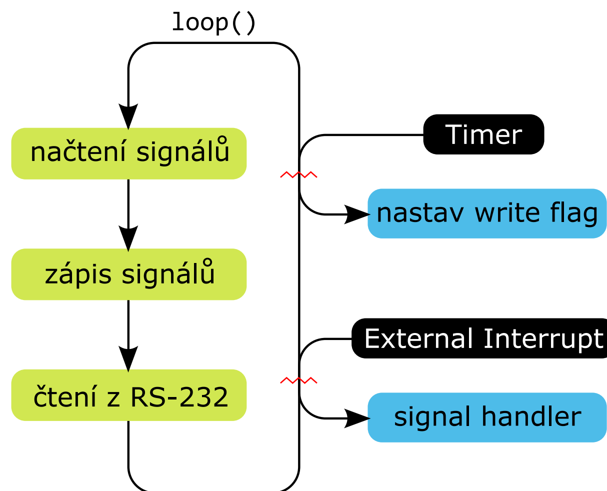
Obsluha přerušení je funkce obsahující sekvenci příkazů, které má mikrokontrolér v případě přerušení provést. V simulačním prostředí se přerušení využívá k implementaci mnoha funkcí. Jednotlivé obsluhy přerušení musí vykonávat co nejméně instrukcí, protože v době obsluhy přerušení nemůže pokračovat vykonávání hlavní smyčky programu a navíc nemůže nastat žádné další přerušení stejného ani jiného druhu. Optimalizace kódu v obsluhách přerušení je pro správnou funkci mezičlánku klíčová. Jednotlivé druhy přerušení a jejich použití je popsáno v sekcích 2.4.3 a 2.4.4.

2.4 Průběh komunikace

Komunikace mezi PC a mezičlánkem probíhá přes standardní USB spojení. Na Arduinu je kromě hlavního mikroprocesoru ještě čip ATmega16U2, který mimo jiné zprostředkovává emulaci sériové linky RS-232 přes USB. Arduino se tedy v operačním systému připojuje na porty typu COM a může přes ně probíhat obousměrná datová komunikace.

Jelikož jediným úkolem mezičlánku je zprostředkovávat překlad signálů z datové

komunikace na napěťové signály, zabírá tato činnost většinu procesorového času. Mezičlánek opakovaně vykonává činnosti dle diagramu na obrázku 2.3.



Obrázek 2.3: Diagram činnosti mezičlánku

2.4.1 Hlavní smyčka programu

V hlavní smyčce, tedy v bloku `loop()`, vykonává program opakovaně čtení ze specifikovaných pinů. Přečtené hodnoty ukládá do proměnných, které jsou v kontextu této kapitoly označovány „signál“. Poté na konkrétní piny zapíše jednotlivé signály obdržené přes sériovou linku z PC.

Používané piny jsou deklarované pomocí globálních proměnných a mód čtení nebo zápisu je pro každý pin určen v bloku `setup()`. Pro různé situace a různé roboty je tato konfigurace odlišná, což je hlavní důvod, proč musí být programy pro mezičlánek automaticky generovány při každé změně konfigurace pinů. Více o tom v části o generátoru kódu pro Arduino 3.3.6.

2.4.2 Obsluha sériové linky

Událost SerialEvent nastává, pokud Arduino zaznamená příchozí data na sériové lince. Obsluha není zavolána pomocí přerušení, ale voláním metody `serialEvent()` po každém jednotlivém průchodu hlavní smyčkou programu.

Vstupní bajty jsou ukládány do vyrovnávací paměti a bajtů k přečtení tedy může být v danou chvíli více. V mezičlánku je tato metoda použita k načtení signálů vycházejících ze simulace. Signály jsou uloženy do proměnných a v hlavní smyčce odeslány na příslušné piny. O časování se v tomto případě musí starat protějšek komunikační vrstvy, tedy proces běžící v PC.

2.4.3 Přerušení časovačem

Timer se v Arduinu používá k opakovanému spuštění bloku kódu s konstantní periodou. V komunikační vrstvě slouží k odesílání dat přes sériovou linku. Timer je nastaven na periodu 5 ms (200 Hz), což je nejvyšší rychlost, které se podařilo stabilně dosáhnout. Každých 5 ms jsou tedy všechny hodnoty signálů na výstupu testovaného hardwaru odeslány do simulovaného systému v PC.

V obsluze přerušení však odesílání dat neprobíhá, pouze se nastaví pomocná proměnná (tzv. *flag*) na hodnotu `true` a samotné odesílání dat probíhá v hlavní smyčce programu. Tím pádem může být odesílání dat kdykoliv přerušeno např. externím přerušením.

2.4.4 Externí přerušení

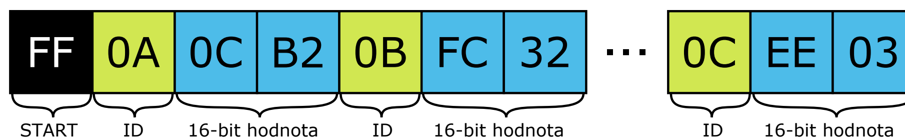
Běh programu lze přerušit i externím signálem, např. vzestupnou hranou. Toho je využito ke čtení PWM signálu nebo emulování senzorů, které komunikují pomocí triggerů či délky pulzů. Pokud žádný z prvků simulace nevyužívá externí přerušení, tento blok v programu není.

2.5 Komunikační protokol

Po sériové lince se dá komunikovat standardně pomocí kódování ASCII v textové podobě. V takovém případě však každý znak potřebuje k přenosu celý bajt. S ohledem na optimalizaci rychlosti přenosu informace, která je v HIL simulaci důležitým faktorem, bylo nutno navrhnout úspornější řešení.

PC si s mezičlánkem vyměňuje zprávy zakódované ve speciálním komunikačním protokolu, který byl pro tyto účely vytvořen. Každá zpráva obsahuje vždy všechny signály přítomné v simulaci.

2.5.1 Zpráva protokolu



Obrázek 2.4: Příklad jedné zprávy komunikačního protokolu

Každá zpráva (Obrázek 2.4) začíná bajtem `0xFF`, za kterým může následovat libovolný, ale předem známý počet hodnot signálu.

Každá z těchto hodnot je uvozena unikátním identifikačním bajtem[3], který spojuje hodnotu ze zprávy s konkrétní signálovou proměnnou. Různých signálů může být tedy teoreticky až 255, což daleko přesahuje počet vstupů a výstupů na Arduinu.

Samotná hodnota je vždy dvoubajtová, má tedy rozlišení 16-bitů, což představuje 65 535 úrovní signálu.

2.5.2 Dekódování protokolu

Protokol byl vytvořen s ohledem na co největší datovou úspornost a snadné dekódování. Umožňuje navázat komunikaci i uprostřed zprávy nebo po krátkodobé ztrátě spojení. Implementace tohoto algoritmu musí udržovat svůj stav ve vedlejších (statických) proměnných tak, aby mohla zpracovávat příchozí bajty po jednom. Tento algoritmus je v téměř nezměněné podobě implementován i na straně PC, kde dekóduje zprávy odeslané z mezičlánku.

S každým příchozím bajtem je nutno zjistit, o jaký druh se jedná s ohledem na stav dekódovacího algoritmu. Pokud dekódování zprávy ještě nezačalo (tj. platí `started == false`) a zároveň se příchozí bajt rovná bajtu `0xFF` (`1111 1111`), nastavíme proměnnou `started` na `true`, což znamená, že dekódování zprávy bylo započato. Tento mechanismus zajistí, že v samotných hodnotách signálu se hodnota `0xFF` může nacházet, jelikož proměnná `started` již není `false`.

Další bajt po startovním bajtu by měl být identifikační bajt hodnoty signálu. Pokusíme se ho najít v předem známém poli identifikačních bajtů. Pokud nebyl nalezen, znamená to, že čtení příchozích bajtů bylo započato na špatném místě (např. uprostřed zprávy), a proto musí být proces dekódování resetován. Naopak pokud byl identifikační bajt nalezen, můžeme pokračovat v dekódování nastavením proměnné `key`.

Následující dva bajty obsahují samotnou 16-bitovou hodnotu signálu. Po jejich přečtení je nejprve daný signál použit, tj. je namapován na správný rozsah, a příslušná signálová proměnná je nastavena na novou hodnotu. Dále je zvýšeno počítadlo příchozích signálů.

Tento proces se opakuje, dokud signálové počítadlo nedosáhne předem známé hodnoty počtu signálů ve zprávě. Poté je dekódovací mechanismus resetován a připraven na čtení další zprávy.

Implementace výše popsaného algoritmu je ve zdrojovém kódu 2.1.

[3] Tento bajt NESMÍ mít hodnotu `0xFF`

```
int signalCounter = 0;      // počet již přečtených signálů
boolean started = false;    // začalo čtení zprávy?
boolean high = false;       // byl přečten první bajt hodnoty?
byte key = START;           // načtené ID hodnoty
byte hi = 0;                // načtený první bajt hodnoty
byte lo = 0;                // načtený druhý bajt hodnoty

void processByte(byte b) {
  if (!started && b == START) {
    started = true;
  } else if (key != START) {    // už bylo načteno ID hodnoty
    if (!high) {
      hi = b;                   // načti první bajt hodnoty
      high = true;
    } else {
      lo = b;                   // načti druhý bajt hodnoty
      high = false;
      useSignal();              // signál přečten -> použit
      signalCounter++;
      key = START;              // resetuj ID signálu na 0xFF
    }
  } else if (started) {         // čtení zprávy začalo
    if (findSignal(b)) {        // je to známé ID?
      key = b;                  // nové ID signálu
    } else {
      started = b == START;     // chyba, resetuj čtení zprávy
    }
  }

  if (signalCounter == packetSize) {
    signalCounter = 0;          // přečteny všechny hodnoty
    started = false;            // připrav na další zprávu
  }
}
```

Zdrojový kód 2.1: Dekódování protokolu

2.5.3 Kódování protokolu

O poznání jednodušší je kódování do protokolu při odesílání zprávy z mezičlánku do PC. Napřed je inicializován výstupní buffer, což je pole o třech bajtech. Poté je na sériovou linku zapsán `START` bajt. Následně se po jednom odešlou všechny výstupní signály.

Nejprve je vždy do výstupního bufferu zapsán identifikační bajt daného signálu. Hodnota signálu je poté namapována ze svého skutečného rozsahu na 16-bitovou hodnotu protokolu. Následně musí být rozložena na dva bajty pomocí bitových

operací. Výsledné dva bajty jsou nahrány do výstupního bufferu a celý buffer je najednou zapsán na sériovou linku. Takto algoritmus pokračuje, dokud nezapíše všechny hodnoty.

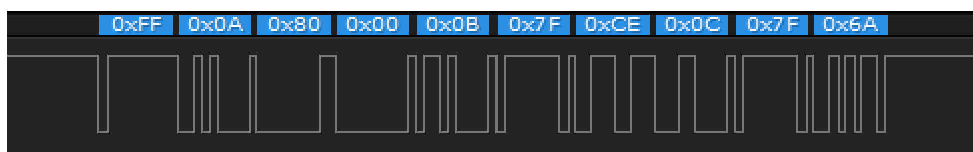
Ukázka kódování jedné zprávy obsahující jeden digitální a jeden analogový signál je ve zdrojovém kódu 2.2.

```
byte buf[3];                        // výstupní buffer
byte lo, hi;                        // první a druhý bajt hodnoty
short v;                            // namapovaná hodnota signálu

Serial.write(0xFF);                 // zápis START bajtu

v = (short) map(LED_IN_on_off, 0, 1, SHORT_MIN, SHORT_MAX);
buf[0] = 12;                            // identifikační bajt
buf[1] = (byte) ((v >> 8) & 0xFF);      // rozložení na dva bajty
buf[2] = (byte) (v & 0xFF);
Serial.write(buf, 3);                   // zápis bufferu na výstup

v = (short) map(MOTOR_0_IN_enable, 0, 255, SHORT_MIN, SHORT_MAX);
buf[0] = 13;
buf[1] = (byte) ((v >> 8) & 0xFF);
buf[2] = (byte) (v & 0xFF);
Serial.write(buf, 3);
```

Zdrojový kód 2.2: Kódování protokolu

2.5.4 Analýza komunikace

Pomocí logického analyzátoru byl zachycen průběh komunikace na sériové lince (Obrázek 2.5). V ukázce je vidět startovní bajt `0xFF` následovaný třemi signály (tj. tři trojice bajtů). Signály mají identifikační bajty po řadě `0x0A`, `0x0B`, `0x0C`.



Obrázek 2.5: Jedna zpráva zachycené komunikace

Kapitola 3

Simulace prostředí

Tato kapitola se podrobně věnuje další dílčí části simulačního prostředí, která představuje simulovaný systém. Simulace systému probíhá téměř kompletně na PC z důvodu výpočetní složitosti, především kvůli nutnosti vizualizovat systém ve 3D prostoru.

Jedna z možností, jak simulovat reálný svět na PC, je pomocí numerických metod. Systémy jsou specifikovány exaktními matematickými modely a v každém kroku simulace je určitou metodou (např. Newton-Eulerovou) spočítán přírůstek dané veličiny. Taková simulace tedy spočívá v numerickém řešení diferenciálních rovnic. Nevýhodou tohoto přístupu je především nutnost znát přesné matematické modely vyskytující se v systému a vysoká časová náročnost. To je v případě simulace prostředí pro robota nevhodné.

V posledních letech byla vyvinuta celá řada tzv. fyzikálních enginů, především pro potřeby počítačových her. Ty slouží k simulaci běžných jevů, jako je gravitace, tření, valivý odpor, dynamika tekutin a částic apod., v reálném čase. Fyzikální engine využívá fyzikálních zákonů k výpočtu poloh a rychlostí 3D objektů ve scéně, jejichž hranice uživatel specifikuje 3D polygony, a řadou parametrů stanoví jejich počáteční podmínky. Pro tuto metodu není potřeba znát přesné matematické modely, a proto je fyzikální engine využit i pro simulaci systémů v této práci.

3.1 Volba simulátoru

Tuto metodu využívá většina programů pro simulaci robotů. Jedním z nejpoužívanějších je simulátor Webots od firmy Cyberbotics, který umožňuje volbu mezi proprietárním fyzikálním enginem a ODE[1] a pro vizualizaci využívá OGRE[2]. Propojení s Arduinem by ale bylo poměrně komplikované, protože Webots nenabízí mnoho možností programovacích technik a kontroléry robotů nejsou přenositelné.

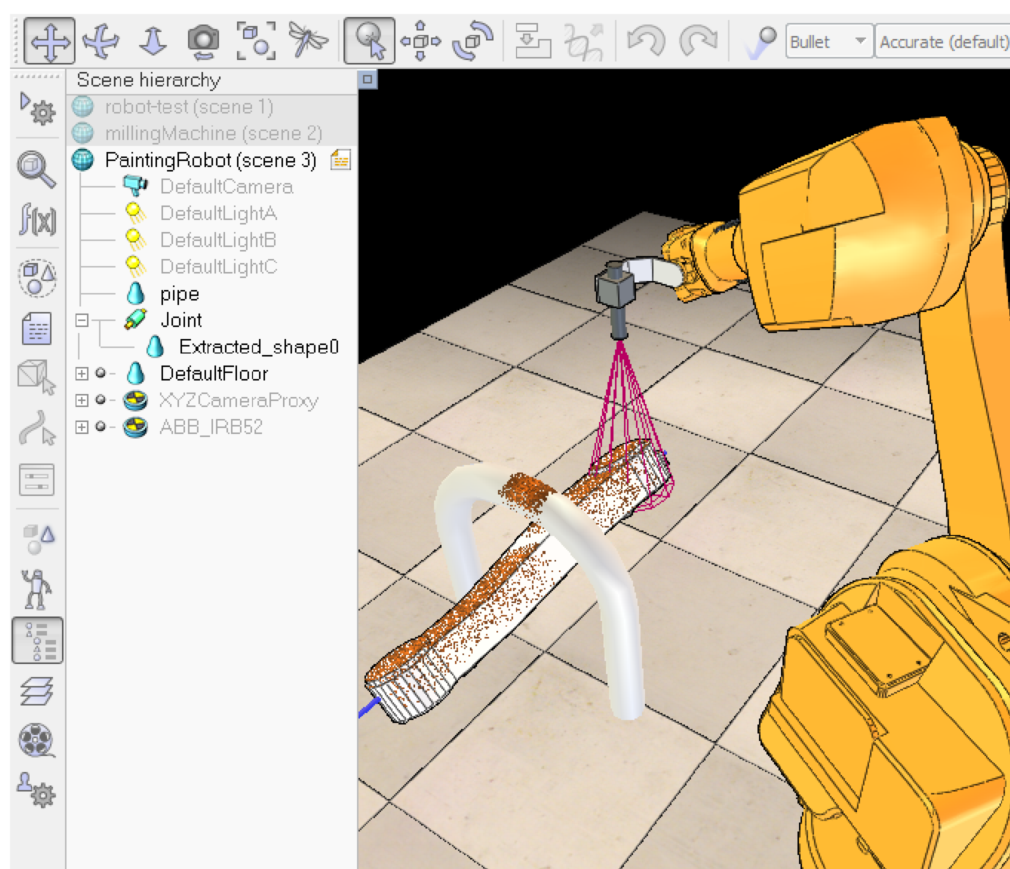
[1] Open Dynamics Engine

[2] Object-Oriented Graphics Rendering Engine

Dále lze zmínit třeba ARS, Gazebo nebo SimSpark, které z různých důvodů tomuto projektu rovněž nevyhovovaly. Jako nejlepší robotický simulátor se jevil V-REP, který je stručně popsán v následující sekci.

3.2 Simulační software V-REP

V-REP (**V**irtual **R**obot **E**xperimentation **P**latform) je produktem firmy Coppelia Robotics vydávaným zdarma pro výukové účely (Obrázek 3.1). Hlavním cílem tohoto simulátoru je nabídnout co nejvíce možností programování jak samotného simulačního prostředí, tak jednotlivých objektů ve scéně (např. robotů). V-REP je navržen přímo tak, aby jeho zabudování do HIL simulačních prostředí bylo co možná nejjednodušší. Coppelia Robotics navíc k V-REP simulátoru poskytuje online obsáhlou dokumentaci a celou řadu návodů, což je vzhledem k zaměření této práce velmi důležité.



Obrázek 3.1: Hlavní okno simulátoru V-REP

Fyzikální engine je možno volit mezi Bullet, ODE či Vortex. Vizualizace probíhá pomocí interních knihoven. Větší část tohoto softwaru je vyvinuta v jazyce C++

a lze nainstalovat na platformy Windows, Linux i MacOS X.

Pro programování robotů i simulace poskytuje V-REP interní API[3], které je možno volat ze skriptů běžících přímo v programu, a vzdálené API, které je možno volat ze vzdáleného klienta (nemusí běžet na stejném PC). Možnosti programování kontrolérů pro objekty ve V-REP lze shrnout do následující tabulky:

Vlastnost	Interní skript	Add-on	Klient vzdáleného API
Druh API	Standardní API	Standardní API	Vzdálené API
Podporované jazyky	Lua	Lua	C/C++, Python, Java, Matlab, Urbi
Funkce	>280 funkcí	>270 funkcí	>100 funkcí
Synchronní operace	Ano, bez zpoždění	Ano, bez zpoždění	Ano, se zpožděním způsobeným komunikací
Asynchronní operace	Ano	Ano	Ano

Tabulka 3.1: Programovací techniky [3]

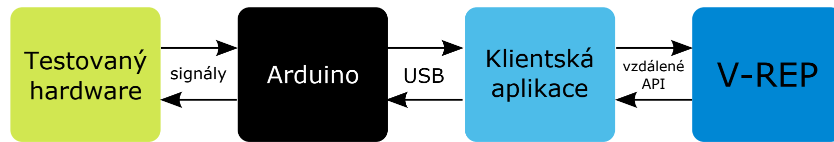
V této práci jsou interní skripty využity k zjišťování parametrů za běhu simulace a jejich odesílání mezičlánku nebo naopak přijímání signálů a jejich aplikování na simulační prostředí. Propojení a komunikaci s mezičlánkem obstarává samostatně běžící klientská aplikace, jejíž návrh a implementace jsou hlavní částí této práce. S V-REP simulátorem tato aplikace komunikuje pomocí vzdáleného API. K její inicializaci a spuštění je využit jednoduchý add-on[4] vytvořený v jazyce Lua, aby bylo možné klientskou aplikaci spouštět přímo z prostředí simulátoru V-REP.

3.3 Klientská aplikace

Poslední chybějící částí je zprostředkování komunikace mezi simulačním prostředím (simulátor V-REP) a mezičlánkem (Obrázek 3.2). Tyto funkce jsou zajištěny klientskou aplikací HIL Simulation Plugin, které se věnuje zbytek této kapitoly. Kromě komunikace rovněž klientská aplikace umožňuje připojení k mezičlánku pomocí sériové linky a hlavně generování programu pro mezičlánek podle objektů vyskytujících se ve scéně simulátoru.

Hardwarové nároky a návod k instalaci jsou v příloze A.

[3] Rozhraní pro programování aplikací (angl. *Application Programming Interface*)
[4] česky *doplněk*



Obrázek 3.2: Klientská aplikace

3.3.1 Volba programovacího jazyka

Klientská aplikace je napsána v jazyce Java[5]. Tento jazyk byl zvolen především proto, že je s ním autor obeznámen. Jedná se o jeden z nejpoužívanějších objektově orientovaných programovacích jazyků a V-REP simulátor pro něj poskytuje vzdálené API. Aplikace využívá nejnovější verzi (Java 8) a nové programovací techniky jako jsou lambda výrazy a zpracování kolekcí pomocí tzv. streamů.

Kromě knihovny vzdáleného API ze simulátoru V-REP (`remoteApiJava.dll`) využívá aplikace knihovnu JSSC[6] pro komunikaci po sériové lince, knihovnu Jackson pro serializaci dat ve formátu JSON[7] a knihovnu JavaFX pro implementaci grafického uživatelského rozhraní.

3.3.2 Komunikace s V-REP

Komunikace se simulátorem V-REP probíhá pomocí zmíněného vzdáleného API. Stará se o ni třída `SimulatorHandler`. Ze vzdáleného API je využito především funkcí `simxGetFloatSignal()`, `simxSetFloatSignal()` pro čtení signálů příslušného datového typu a `simxGetIntegerSignal()`, `simxSetIntegerSignal()` pro zápis signálů.

K přečtení či zápisu signálu pomocí vzdáleného API je nutno znát přesný název signálu. Název je složen ze jména zařízení, jeho pořadového čísla ve scéně, podtržítka a názvu daného signálu[8].

Na straně V-REP simulátoru má každý objekt (aktuátor či senzor), který potřebuje přijímat či odesílat signály, připojený asynchronní interní skript (tzv. *child script*) v jazyce Lua. Úkolem tohoto skriptu je buď číst signál s konkrétním jménem a novou hodnotu signálu aplikovat na aktuátor (např. rozsvítit LED či nastavit rychlost motoru), nebo vypočítat novou hodnotu senzoru a odeslat tento signál do klientské aplikace (např. ultrasonický senzor vzdálenosti). Příklad skriptu pro rozsvícení LED je ve zdrojovém kódu 3.1.

[5] Vyvinut firmou Sun Microsystems
[6] Java Simple Serial Connector
[7] JavaScript Object Notation
[8] např. název signálu určujícího rychlost jednoho z motorů: `electricMotor6V#2_enable`

```
if (sim_call_type==sim_childscriptcall_actuation) then
  -- načtení aktuálních parametrů LED
  local state,z,d,s=simGetLightParameters(handle)

  -- prečtení signálu z klientské aplikace
  local signal=simGetIntegerSignal(name..suffix..onOffSignal)

  -- nastavení nové hodnoty LED zapnuto/vypnuto
  simSetLightParameters(handle, signal, nil, d, s)
end
```

Zdrojový kód 3.1: Lua skript pro LED aktuátor

Výše zmíněný skript, který musí mít každý aktuátor i senzor používaný v HIL simulaci, je ukládán spolu s grafickým 3D modelem do jednoho souboru s příponou `ttm`.

Ke každému zařízení je rovněž nutno přiložit konfigurační soubor ve formátu JSON, který klientské aplikaci mimo jiné specifikuje, které V-REP objekty má sledovat, a které signály zařízení odesílá nebo přijímá. Obsah konfiguračního souboru popisuje sekce 3.3.5 o datovém modelu.

V rámci této práce bylo vytvořeno několik základních aktuátorů a senzorů, které jsou podrobně popsány v příloze A.4.

3.3.3 Komunikace s mezičlánkem

Komunikace s mezičlánkem je zajišťována instancí třídy `BridgeHandler`, která má za úkol inicializovat specifikovaný port a přihlásit se k příjímání dat. Přijímání dat z mezičlánku probíhá asynchronně dle návrhového vzoru objektově orientovaného programování Pozorovatel[9]. Po zachycení události na sériové lince jsou bajty na vstupu dekódovány velmi podobným mechanismem jako při přijímání dat mezičlánkem (viz sekci o dekódování protokolu 2.5.2).

Tento proces probíhá v samostatném vlákně, je proto nutno příchozí zprávy synchronizovat s hlavním komunikačním vláknem. Na to je využita synchronizační kolekce `BlockingQueue` z balíku `java.util.concurrent`.

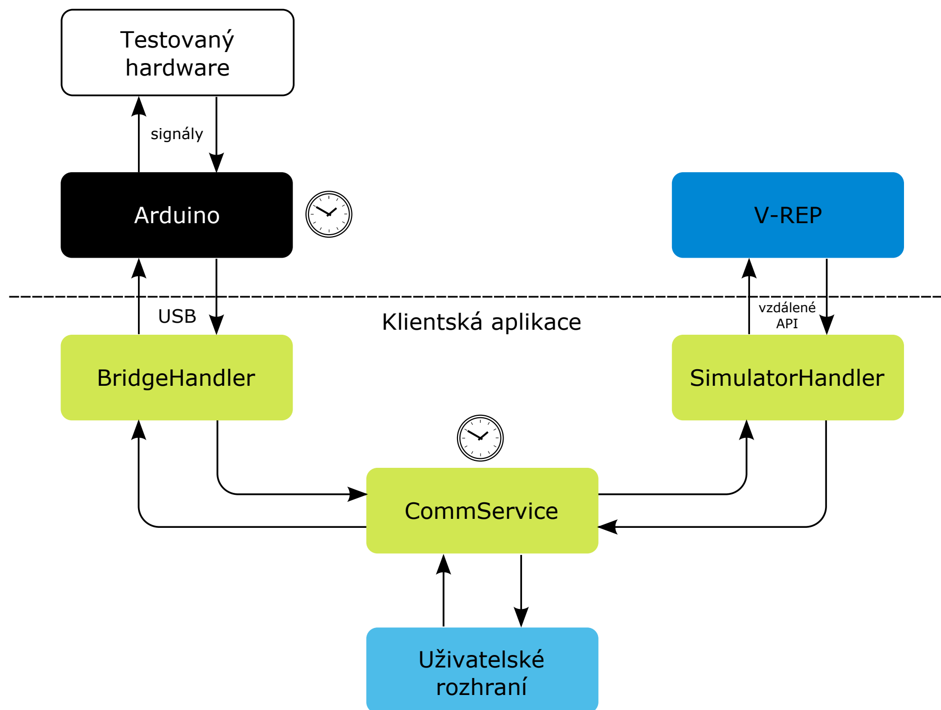
Pokud je nutno data naopak odeslat na sériovou linku, jsou nejprve zakódována do protokolu a poté odeslána.

3.3.4 Synchronizace komunikace

Jelikož hlavní vlákno klientské aplikace musí neustále reagovat na uživatelské vstupy (např. stisk tlačítka), musí komunikace probíhat na pozadí, tedy v jiném

[9]angl. *Listener*

vlákně. Instance třídy `CommService` má na starost spuštění tohoto komunikačního vlákna a předávání zpráv mezi V-REP simulátorem a Arduinem. Tuto činnost vykonává opakovaně se stanovenou periodou. Pro potřeby přesného časování je využita třída `java.util.concurrent.ScheduledThreadPoolExecutor`.



Obrázek 3.3: Diagram komunikace HIL simulační smyčky

Zde je nutno poznamenat, že operační systém sám přiděluje procesorový čas mezi všechny běžící procesy a perioda zasílání zpráv proto nemůže být deterministická. Při komunikaci dochází k určitému rozkolísání komunikační frekvence (angl. *jitter*). Jak toto rozkolísání ovlivňuje řídící smyčku HIL simulace je popsáno v ukázce programování robota v kapitole 4.

Všechny úkony prováděné v jedné periodě komunikačního vlákna jsou popsány ve zdrojovém kódu 3.2. Proměnné `bridge` resp. `simulator` odkazují na instance třídy `BridgeHandler` resp. `SimulatorHandler` popsané výše.

```
// Mezičlánek --> V-REP
// odebere čekající zprávu z BlockingQueue
Map<Signal, Number> bridgeInput = bridge.receiveSignals();
// odešle signály do V-REP
simulator.sendSignals(bridgeInput);

// V-REP --> Mezičlánek
// požádá vzdálené API o hodnoty signálů
Map<Signal, Number> simulatorInput = simulator.receiveSignals();
// odešle signály do mezičlánku
bridge.sendSignals(simulatorInput);
```

Zdrojový kód 3.2: CommService

Diagram komunikace celé HIL simulační smyčky od mezičlánku po simulátor V-REP je na obrázku 3.3. V diagramu vystupují všechny tři výše zmíněné třídy a šipky naznačují toky informací. Obrázek hodin označuje třídy, které generují vlastní periodu komunikace v dedikovaném vlákně.

Uživatelské rozhraní je závislé pouze na třídě `CommService` a ta je nezávislá na implementaci tříd `BridgeHandler` a `SimulatorHandler`, což umožňuje v budoucnu implementovat i komunikaci s jinými simulátory a mezičlánky, než pro jaké je klientská aplikace v době vydání navržena.

3.3.5 Datový model

Jak již bylo zmíněno v kapitole 2 o programování mezičlánku, konfigurace pinů musí vždy reflektovat počet a druh zařízení ve scéně simulátoru. Pokud se např. k robotu připojí nový senzor, mezičlánek musí být přeprogramován tak, aby na zvolených pinech generoval očekávané signály, které testovaný hardware může zpracovat.

Pro tyto účely musel být navržen datový model zahrnující veškeré potřebné informace o použitých zařízení tak, aby bylo vždy možné generovat správný program pro mezičlánek.

Základem datového modelu je objekt `DeviceConfig`, který obsahuje veškeré informace o jednom zařízení (tj. aktuátor nebo senzor). Tento objekt je serializován do formátu JSON a ukládán jako textový soubor s příponou `json` do stejného adresáře a se stejným jménem jako soubor obsahující 3D model objektu ve scéně a připojené Lua skripty (přípona `ttm`). Ukázka konfiguračního souboru pro aktuátor LED je ve zdrojovém kódu 3.3.

```
{
  "name": "LED",
  "signals": [
    {
      "name": "active",
      "direction": "INPUT",
      "type": "INTEGER",
      "min": 0,
      "max": 1,
      "mode": "DIGITAL"
    }
  ],
  "file": "led.ttm"
}
```

Zdrojový kód 3.3: Ukázka konfiguračního souboru JSON

Soubor základní konfigurace zařízení

Každé zařízení musí mít v souboru uvedeno unikátní jméno (`name`) a seznam obsahující konfigurace nejméně jednoho vstupního nebo výstupního signálu (`signals`). Tento konfigurační soubor je do samotné aplikace při spuštění načten a uložen do tříd `DeviceConfig`, `SignalConfig`, případně `Handler`. Každá konfigurace signálu specifikuje tyto položky v libovolném pořadí:

"name"
: jméno daného signálu,

"direction"
: směr toku dat, kde `INPUT` znamená do simulátoru a `OUTPUT` ze simulátoru,

"type"
: datový typ signálu `INTEGER` nebo `FLOAT`,

"min"/"max"
: rozsah, který se používá na mapování hodnoty při kódování do komunikačního protokolu,

"mode"
: mód, ve kterém je signál přenášen, určuje jaké vlastnosti jsou vyžadovány od pinu na mezičlánku, hodnoty: `DIGITAL`, `ANALOG`, `PWM`, `INTERRUPT`,

Pokud signál vyžaduje speciální instrukce pro čtení a zápis (např. zařízení HC-S04 popsané v sekci A.4.4), jeho konfigurace musí obsahovat ještě objekt typu "handler". V tom je specifikováno jméno souboru speciálních instrukcí (viz níže) a seznam konfigurací pinů, které signál používá. Každá konfigurace pinu obsahuje tyto položky:

"name"
: jméno daného pinu,

"direction"
: směr toku dat, kde `INPUT` znamená do simulátoru a `OUTPUT` ze simulátoru,

"mode"
: určuje, jaké vlastnosti jsou vyžadovány od pinu na mezičlánku, hodnoty: `DIGITAL`, `ANALOG`, `PWM`, `INTERRUPT`.

Soubor speciálních instrukcí pro čtení a zápis

Pokud je nutné, aby signál komunikoval s více piny najednou nebo pokud má signál nároky na extrémně rychlou odezvu, musí mechanismus čtení a zápisu probíhat na mezičlánku. Zařízení s takovým signálem musí být kromě souborů `ttm` a `json` popsáno ještě v souboru speciálních instrukcí s příponou `hlr`. Obsah tohoto souboru je při generování programu pro mezičlánek připojen na specifikovaná místa. Ukázka konkrétního souboru speciálních instrukcí je součástí zdrojového kódu A.10 v příloze.

Jednotlivé sekce speciálních instrukcí musí být odděleny nadpisy, které určují, kam má generátor následující řádky kódu zařadit. Možné nadpisy bloků jsou:

##DECLARATIONS##
: tento blok umisťuje generátor na konec standardních deklarací,

##SETUP##
: tento blok umisťuje generátor na konec bloku `setup()`,

##LOOP##
: tento blok umisťuje generátor na konec bloku `loop()`,

##FUNCTIONS##
: tento blok je umisťován mimo ostatní bloky a mohou v něm být tedy definovány pomocné funkce.

Dále jsou pro všechny řetězce obklopené znakem procenta (%) vygenerovány unikátní identifikátory, které umožní přidávat několik zařízení daného typu najednou. Pokud se řetězec mezi procenty shoduje se jménem jednoho z pinů v konfiguraci signálu popsané v sekci 3.3.5, generátor nahradí tento řetězec proměnnou, která obsahuje identifikátor konkrétního pinu, který byl připojen uživatelem.

K přístupu k dalším možným způsobům identifikace pinu pro potřeby programování AVR lze za jméno pinu doplnit tyto speciální direktivy:

PORT
: jméno portu, ve kterém se daný pin nachází, vrátí např. „PORTD“

PIN
: AVR pojmenování dané skupiny pinů, vrátí např. „PINB“

PCINTE
: číslo skupiny pinů pro nastavení registru PCICR při využívání PinChange interruptů

PCMSK
: PCMSK daného pinu v číselné podobě, vrátí např. „`1 << 7`“

Takový řetězec je pak nahrazen příslušnou informací o daném pinu. Například řetězec `%triggerPCINTE%` je nahrazen číslem PCINTE skupiny reálného pinu, který byl při konfiguraci přiřazen k pinu `trigger`.

Pomocí těchto nástrojů lze vytvořit celou škálu různých zařízení, která vyžadují odezvu v řádu mikrosekund nebo potřebují využívat k přenosu jednoho signálu více pinů. Zde je nutno poznamenat, že speciální instrukce nesmí obsahovat žádná volání `delay()`, která by mohla blokovat komunikaci s mezičlánkem.

V době vydání klientské aplikace jsou v simulátoru V-REP vytvořena čtyři zařízení s přiloženými konfiguračními soubory (podrobně popsáno v příloze A.4). Jedná se o koncový spínač, ultrasonický dálkoměr, elektrický stejnosměrný motor a LED. Všechna tato zařízení budou využita v ukázce programování robota v kapitole 4. Aplikace je navržena tak, aby bylo možné kdykoliv doplnit libovolná další zařízení bez zásahu do zdrojového kódu.

3.3.6 Generátor programu pro mezičlánek

Před spuštěním simulace musí být do mezičlánku nahrán program[10], ve kterém je specifikována úloha každého použitého pinu a druhy signálů, které má mezičlánek přijímat a odesílat. O generování sketche se stará třída `Sketch`, která na vstupu dostává seznam objektů typu `Signal`. Tyto objekty kromě konfigurace `SignalConfig` obsahují navíc i konkrétní název signálu, který očekává simulátor V-REP, a pin, se kterým má být signál spojen. Pokud signál využívá více pinů, pak tento objekt obsahuje mapu pinů a jejich konfigurací.

S využitím všech těchto informací může třída `Sketch` vygenerovat potřebný program. Každý generovaný program se skládá z několika částí, které se postupně připojují k výslednému textovému řetězci. Příklad vygenerovaného programu pro mezičlánek pro přenos dvou signálů (stisknutí tlačítka a rozsvícení LED) je v příloze B.2.

[10] v případě Arduina se nazývá *sketch*

Makra

První částí jsou makra, která některé další části využívají. Makra jsou generována nezávisle na nastavení signálů a jsou vždy stejná. Jedním z použitých maker je například `FAST_WRITE` (3.4), které umožňuje zápis digitálního signálu na pin pomocí AVR instrukcí [2], které jsou mnohonásobně rychlejší, než zápis pomocí funkcí z knihovny Arduino.

```
#define FAST_WRITE(pin, port, value)   \
do {                                   \
  if (value)  port |= _BV(pin);        \
  else  port &= ~_BV(pin);             \
} while (0)
```

Zdrojový kód 3.4: Makro `FAST_WRITE`

Deklarace

Jako další se připojují deklarace proměnných. Nejprve se definují tyto konstanty: rozpětí datového typu `int`, hodnota `START` bitu pro komunikační protokol, velikost komunikačního paketu a pole identifikačních bajtů signálů v protokolu.

Následně se definují proměnné pinů. Každý název této proměnné se skládá z předpony `p_` a jména daného signálu. Poté se definují proměnné, které v průběhu simulace obsahují aktuální hodnoty signálů. Jejich jména jsou složena z předpony `s_`, ze jména signálu a jeho směru. Dále se připojují deklarační bloky z objektů `Handler`.

Setup

V bloku `setup()` se nastavují všechny piny na správné módy čtení či zápisu, připojí se případné hardwarové interrupty a nastaví se timer interrupt. Poté se případně přidají nastavovací instrukce z objektů `Handler`.

Loop

Do tohoto bloku se přidají instrukce ke čtení a zápisu signálů podle toho, o jaký druh signálu se jedná. Poté se případně přidají čtecí či zapisovací instrukce z objektů `Handler`.

Pomocné funkce

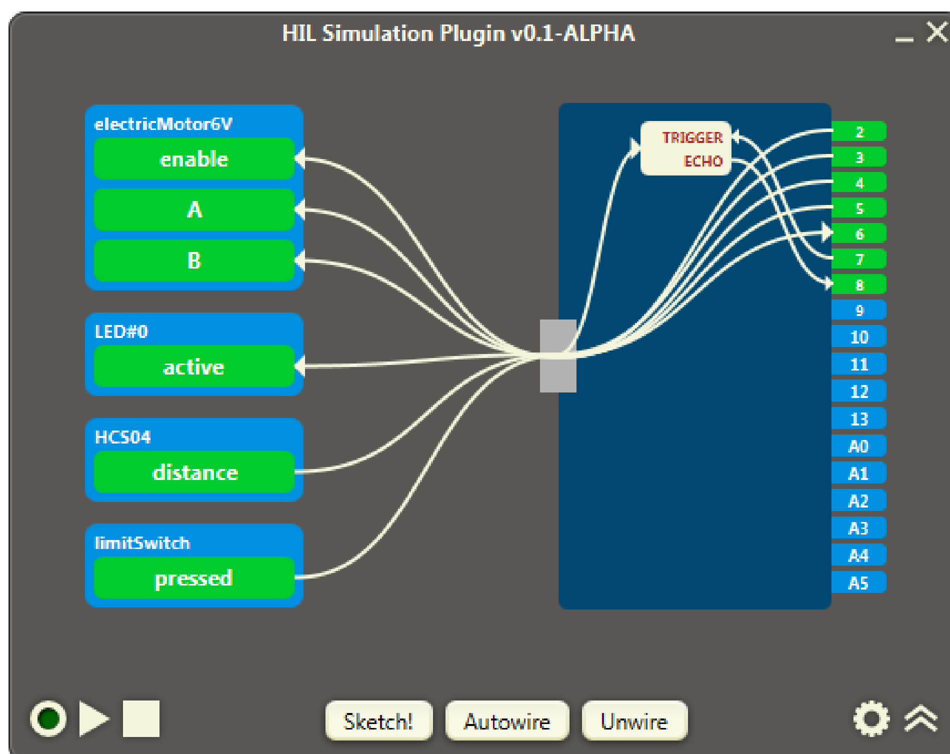
Pokud to konfigurace signálů vyžaduje, nyní se připojí pomocné funkce, jako je např. obsluha hardwarového přerušení, nebo funkce speciálního čtení či zápisu z objektů `Handler`.

Komunikace

Na konec programu se připojí funkce zpracovávající příchozí data po sériové lince a funkce pro zasílání dat opačným směrem. Tyto funkce byly podrobně popsány v části o komunikačním protokolu (2.5) v kapitole 2.

3.3.7 Grafické uživatelské rozhraní

Uživatelské rozhraní umožňuje uživateli specifikovat parametry simulace a zároveň je přehledně prezentuje. Při vývoji byl kladen důraz nejen na jednoduchost použití, ale i na dobrou vizuální stránku.



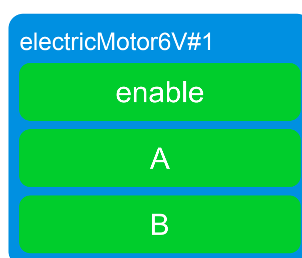
Obrázek 3.4: Hlavní okno klientské aplikace

Od uživatele je nutno získávat několik vstupů, např. jaké piny mají být propojeny s jakými signály či kdy spustit a zastavit simulaci. V následujícím textu budou popsány všechny části grafického uživatelského rozhraní a způsob, jak jsou výše uvedené vstupy získávány.

Seznam zařízení

Levou polovinu hlavního okna aplikace zabírá seznam všech objektů ve scéně, které aplikace rozpozná. Tento seznam se automaticky aktualizuje jednou za 300 ms, pokud uživatel toto chování v nastavení nedeaktivuje. Rozpoznávání probíhá pouze na základě jména zařízení, které je definované v jednom z konfiguračních souborů. Objekty, které mají neznámé názvy, klientská aplikace ignoruje.

Každé zařízení je v uživatelském rozhraní reprezentováno modrým obdélníkem, který obsahuje signály reprezentované zelenými obdélníky. Na obrázku 3.5 je například reprezentace elektromotoru se třemi signály (enable, A a B).



Obrázek 3.5: Zařízení v GUI

Spojování signálů

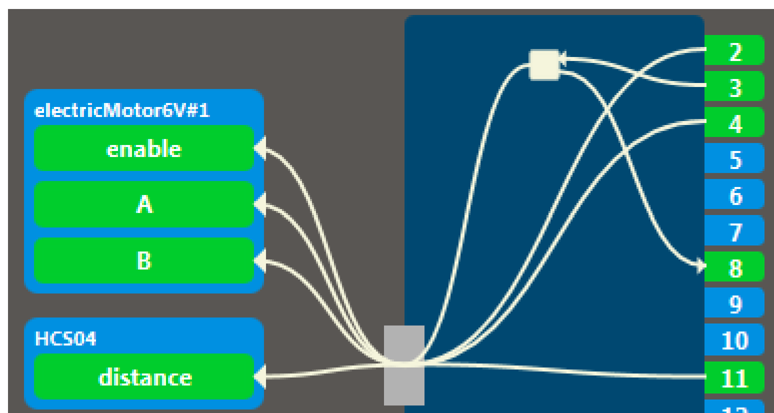
V pravé polovině okna uživatelského rozhraní se nachází vyobrazení mezičlánku, s naznačeným USB portem v levé části, skrz který všechna spojení prochází (tak jako ve skutečnosti), a všemi existujícími piny vpravo. K propojení signálu s pinem stačí táhnout myší ze zeleného obdélníku. Všechny dostupné piny, které mají požadované vlastnosti, se rozsvítí, a pokud uživatel pustí myš nad vyobrazeným pinem, dojde k propojení pinu se signálem.

Pokud daný signál potřebuje komunikovat s více piny najednou (může nastat u zařízení se speciálními instrukcemi pro čtení či zápis), nejprve se rozsvítí v mezičlánku blok, ke kterému lze následně připojit požadovaný počet pinů. Šipky vždy indikují směr toku dat. Ukázku propojení signálů s piny lze vidět na obrázku 3.6. Je vidět, že zařízení HCS04 má jediný signál, který ale využívá dva piny.

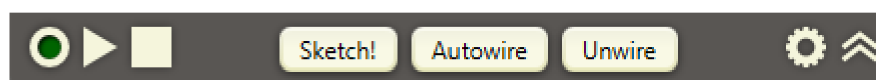
Ovládací panel

Ve spodní části okna se nachází skupina tlačítek (Obrázek 3.7). První skupina tlačítek se vztahuje ke spouštění a zastavování simulace. Po stisku tlačítka *Play* se naváže spojení s mezičlánkem a poté se spustí simulace v simulátoru V-REP pomocí vzdáleného API. Kruh v levé části indikuje běžící simulaci rozsvícením.

Tlačítko *Sketch!* nejprve vygeneruje sketch (popsáno v části 3.3.6), poté zobrazí okno s vygenerovaným programem k nahrání do mezičlánku. Tlačítko *Autowire*



Obrázek 3.6: Ukázka propojení signálů s piny



Obrázek 3.7: Ovládací prvky

umožňuje automatické propojení signálů s volnými piny. Tlačítko *Unwire* odebere všechna spojení mezi piny a signály a umožní tak znovu konfigurovat propojení.

Tlačítko *Options* (ozubené kolo) zobrazí dialog nastavení, kde lze změnit COM port, ke kterému je mezičlánek připojen, chování a vzhled okna aplikace apod. Poslední tlačítko umožňuje schovat seznam zařízení i mezičlánek a aplikace pak zobrazuje pouze tlačítka k ovládání simulace. To uživatel může využít, pokud se konfigurace zařízení ve scéně dlouhodobě nemění a okno klientské aplikace by pak zabíralo příliš místa na obrazovce.

Kapitola 4

Ukázková simulace robota

Poslední kapitola této práce si klade za cíl sjednotit všechny shromážděné poznatky v ukázce programování reálného robota s využitím HIL simulačního prostředí popsaného v předchozích kapitolách.

V následujícím textu jsou podrobně rozepsány všechny kroky, které uživatel klientské aplikace a simulačního softwaru V-REP musí učinit před spuštěním HIL simulace při implementování úlohy pro konkrétního robota. Úloha byla navíc zvolena tak, aby využila všechny aktuátory a senzory, jejichž konfigurační soubory a modely byly vytvořeny jako součást této práce. Celá tato kapitola může rovněž sloužit jako návod k použití HIL simulačního prostředí.

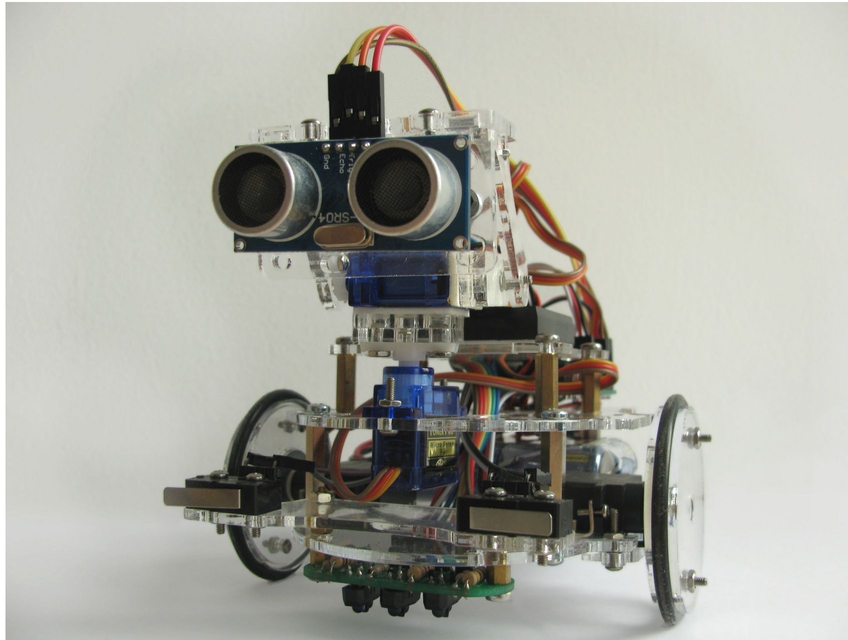
4.1 Popis robota

Robot, který má být simulován, je kolový robot s diferenciálním pohonem dvou kol s možností umístění senzorů na přední nárazník a na horní část robota (Obrázek 4.1). Robot je osazen mikrokontrolérem Arduino Nano, který může být naprogramován pomocí PC.

K pohonu slouží dva stejnosměrné motory. Každý z nich je řízen pomocí dvou signálů určujících směr otáčení a jednoho určujícího rychlost otáčení. Na přední část robota byla dále umístěna žlutá LED, která může sloužit k indikaci stavu řídícího algoritmu. V dané konfiguraci má robot na předním nárazníku namontované dva koncové spínače, které slouží jako taktilní senzory. Na vrchní části robota je dále umístěn ultrasonický senzor vzdálenosti HC-S04.

4.2 Popis úlohy

Úkolem robota je vyhledat a srazit šest překážek, umístěných do kruhu a tvořených obdélníkovými deskami postavenými na kratší hranu. K orientaci v prostoru robot může využít ultrasonického senzoru vzdálenosti i taktilních senzorů.



Obrázek 4.1: Fotografie popisovaného robota

Startovní pozice robota je kdekoliv uvnitř kruhu překážek. Za úspěšně splněnou úlohu je považován stav, kdy všechny desky leží na zemi.

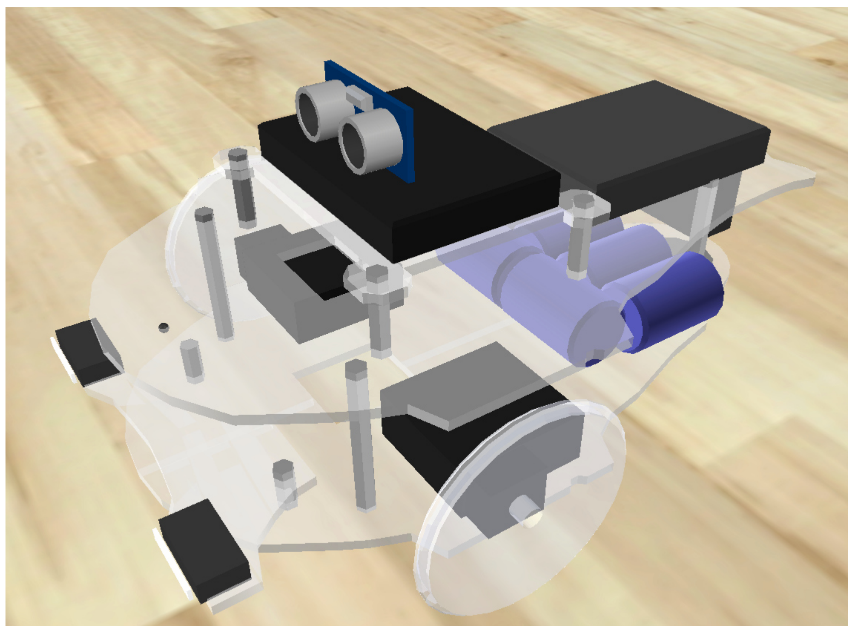
4.3 Zpracování

Pro snadné programování této úlohy využijeme HIL simulační prostředí popsané v předchozích kapitolách. Simulace bude probíhat v simulačním nástroji V-REP a ke komunikaci s řídící jednotkou robota využijeme mezičlánek a klientskou aplikaci HIL Simulation Plugin.

Nejprve je potřeba ve V-REP simulátoru vytvořit model robota a osadit ho virtuálními senzory a aktuátory. Poté stačí propojit signály s příslušnými piny v HIL Simulation Pluginu a odpovídající piny na mezičlánku se skutečným mikrokontrolérem na robotu a simulace může být spuštěna.

4.3.1 Model robota

V simulátoru V-REP se každý robot sestává z modelu pro vizualizaci, který je obvykle složitější a obsahuje mnoho detailů (Obrázek 4.2), a modelu pro fyzikální simulaci, který skutečně reaguje na fyzikální impulsy a jsou pro něj detekovány kolize. Ten je naopak jen velmi hrubým obrysem skutečného robota, čímž simulaci výpočetně výrazně zjednoduší. Oba tyto modely byly vytvořeny pomocí 3D modelovacího



Obrázek 4.2: Model robota

nástroje[1]. Po dokončení modelu robota bylo potřeba nastavit jeho hmotnost a barvu všech materiálů pro vizualizaci.

Model robota byl následně osazen přípojnými body, na které lze přidávat libovolné další objekty, jako jsou např. senzory. Pohon robota je uskutečněn pomocí kloubů `revoluteJoint`, které umožňují otáčení kolem jedné osy a nastavení otáček.

K motorům byla připevněna kola konfigurovaná tak, aby měla relativně velké tření, což eliminuje jejich prokluzování při simulaci. Zadní kolo, které je tvořeno malou kovovou koulí, naopak musí prokluzovat s minimálním třením, a proto bylo konfigurováno jako hladké. Sestavování objektů v rámci simulátoru V-REP je popsáno v jeho dokumentaci.

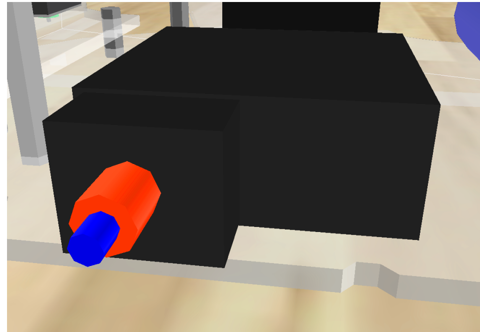
Simulace motorů

Motory (Obrázek 4.3) lze simulovat pomocí objektu `electricMotor6V` z knihovny HIL modelů (viz sekci A.4.2). Je třeba pamatovat na to, že se každý motor otáčí na jinou stranu a to reflektovat při programování řídící jednotky.

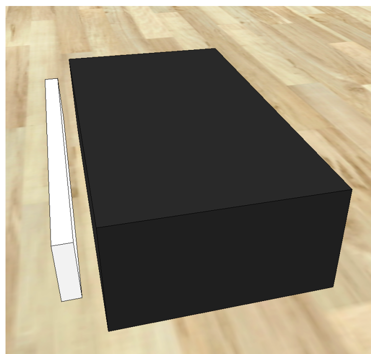
Simulace taktilních senzorů

Taktilní senzory (Obrázek 4.4) jsou rovněž součástí základní knihovny HIL modelů (viz sekci A.4.3). Umístíme je na přední nárazník tak, aby přesahovaly přes podvozek.

[1] Popis vytváření 3D modelů není obsahem této práce.



Obrázek 4.3: Model elektrického motoru



Obrázek 4.4: Model taktilního senzoru

Simulace dálkového senzoru

Dálkový senzor (Obrázek 4.5) z knihovny HIL modelů (viz sekci A.4.4) umístíme na střed horní části modelu a to tak, aby z něj byl nezakrytý výhled vpřed.

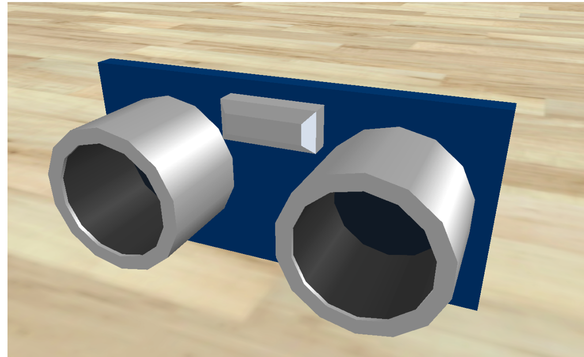
V mikrokontroléru lze zjistit vzdálenost překážky pomocí následujícího vztahu:

$$d = v_s \cdot t_e$$

kde v_s je rychlost šíření zvuku ve vzduchu v metrech za sekundu a t_e je detekované zpoždění ultrazvukových vln v sekundách.

4.3.2 Programování robota

Všechny potřebné části simulačního prostředí byly popsány. Nyní stačí pouze naprogramovat řídící jednotku robota tak, aby splnil požadovaný úkol. Řídící jednotku lze programovat pomocí Arduino IDE. Algoritmus, který povede ke splnění úkolu, je velmi jednoduchý:



Obrázek 4.5: Model ultrasonického senzoru

1. Pokud je v dosahu překážka, jeď vpřed.
2. Pokud byl zaznamenán kontakt s překážkou, jeď vzad několik vteřin.
3. Pokud není v dosahu překážka, otáčej se na místě vlevo.

Funkční kód, který implementuje výše uvedený algoritmus, je v příloze B.1.

4.3.3 HIL simulace

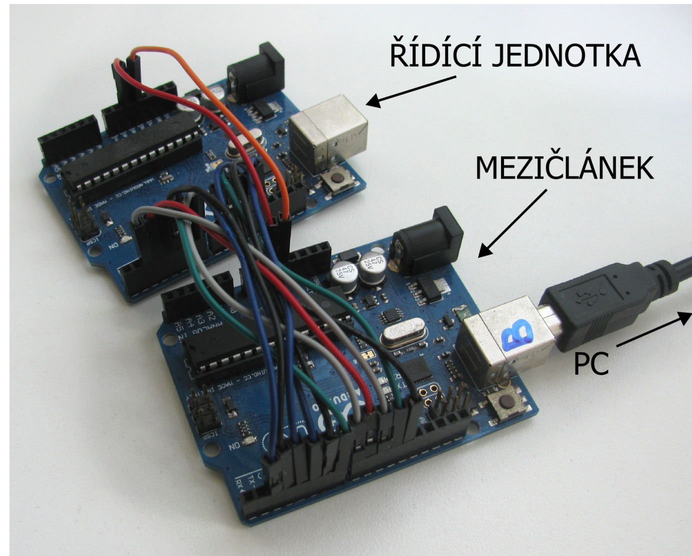
Nejprve byla vytvořena vhodná scéna v simulátoru V-REP. Ta se skládá z podlahy, po které robot může volně jezdit, a šesti výše specifikovaných překážek. Hotová scéna je součástí distribuce klientské aplikace.

Pro úspěšné spuštění simulace je nutno v HIL Simulation Plugin propojit signály s dostupnými piny, vygenerovat program pro mezičlánek a ten do mezičlánku nahrát. Dále je potřeba propojit mezičlánek s testovaným mikrokontrolérem přesně tak, jak je propojen v pluginu (příklad propojení je na obrázku 4.6). Nyní je již možné simulaci spustit pomocí tlačítka *Play* v klientské aplikaci.

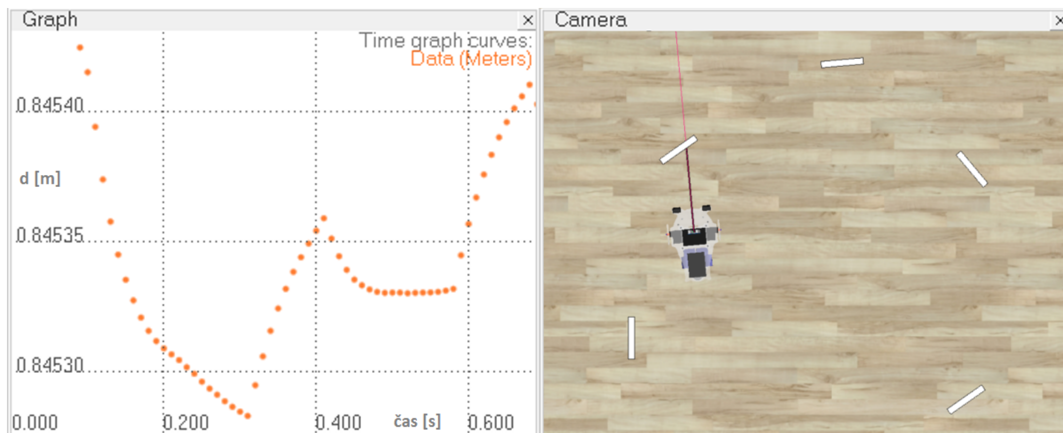
Při simulaci bylo dosaženo velmi dobrých výsledků. Na obrázku 4.7 je vidět grafický výstup ze simulátoru během simulace. Graf v horní části reprezentuje aktuální hodnotu signálu ze senzoru vzdálenosti. Robot zpravidla bez chyby srazil všechny překážky.

Rozkolísání komunikační frekvence (jitter), ke kterému dochází v důsledku nedeterministického plánování úloh v operačním systému Windows, nezpůsobuje v tomto jednoduchém algoritmu žádné problémy. Ty by se pravděpodobně objevily při řízení rychlých procesů nebo při návrhu řízení pomocí identifikace diskrétních systémů, kde je pevná vzorkovací frekvence pro správné fungování důležitá.

Zvýšenou pozornost je také třeba věnovat průběhu simulace, která nemusí nutně běžet v reálném čase, pokud není hardware v PC dostatečně výkonný. Pokud se



Obrázek 4.6: Propojení mezičlánku s řídící jednotkou



Obrázek 4.7: Graf vzdálenosti od překážky a pohled na robota shora

začne simulace za reálným časem opožďovat, přestávají správně fungovat algoritmy založené na přesném časování[2]. V takovém případě je nutno navrhnout řídící algoritmy bez časových prodlev nebo zajistit dostatečně dobrý hardware, který zpoždění simulace nezpůsobí.

[2] např. když obsahují příkaz `delay()`

Závěr

Hlavním úkolem této práce bylo navrhnout a implementovat HIL simulační prostředí pro jednoduché robotické platformy. Jednotlivé kapitoly postupně popsaly jakým způsobem bylo toto prostředí vytvořeno, jak může být konfigurováno a jak ho správně použít.

V práci byly specifikovány všechny vzniklé problémy a implementováno jejich řešení. Výrazným omezením byla nutnost přenést informační smyčku co nejblíže testovanému hardwaru, aby byla zajištěna dostatečně rychlá odezva.

Výsledkem práce je Java aplikace, která plní úlohu pluginu v simulátoru V-REP a zajišťuje komunikaci simulátoru s hardwarovým mezičlánkem, jež zprostředkovává komunikaci mezi PC a testovaným hardwarem. Dále je součástí této práce základní knihovna modelů aktuátorů a senzorů i jejich konfigurací a jeden kompletně sestavený kolový robot v ukázkové scéně.

Vytvořené simulační prostředí může být s úspěchem využito ke zjednodušení procesu programování jednoduchých robotů ať už studenty nebo hobby amatéry. Zdrojový kód a všechny použité soubory práce jsou navíc distribuovány jako open-source a mohou být dále vyvíjeny autorem nebo dobrovolníky.

Ve stávajícím řešení existuje mnoho prostoru ke zlepšení. Především by měla být rozšířena knihovna aktuátorů a senzorů, aby obsahovala většinu nejčastěji používaných zařízení. Dále by měla být implementována možnost zvolit konkrétní druh mezičlánku tak, aby kromě Arduino UNO mohly být využity i výkonnější vývojové desky s více I/O piny. Pro zlepšení uživatelského rozhraní by mohla být aplikace přeložena do více jazyků a umožňovat nastavení více konfiguračních parametrů. Zároveň by aplikace měla být otestována na více operačních systémech a hardwarových konfiguracích. Všechny části simulačního prostředí navíc nabízejí prostor pro rychlostní optimalizaci kódu.

I ve stávající podobě má však podle autora toto řešení jistě velký přínos pro potenciální uživatele.

Příloha A

HIL Simulation Plugin

A.1 Minimální hardwarové nároky

Operační systém: Microsoft Windows 7 (64-bit)
Procesor: Intel Core 2 Duo, 2.8GHz
RAM: 4 GB

A.2 Pokyny k instalaci

Pro správný běh aplikace musí být v systému nainstalováno:

- Java SE Runtime Environment 8 (`http://www.oracle.com`),
- V-REP PRO EDU V3.2.1 (`http://www.coppeliarobotics.com`)[1].

Postup instalace aplikace je následující:

1. Spusťte instalační soubor `hil-plugin_1.0.exe`.
2. Klikněte na *Run Archive*.
3. Vyberte cestu k umístění simulátoru V-REP, standardně: `C:\Program Files (x86)\V-REP3\V-REP_PRO_EDU`.
4. Klikněte na *Unzip*.
5. Pokud simulátor V-REP nebyl před instalací ukončen, je ho nyní třeba restartovat.

[1]Kompatibilita s novějšími verzemi není zaručena

A.3 Pokyny k použití

Klientská aplikace se spouští pomocí položky *HIL Simulation Plugin* v menu *Add-ons* přímo se simulátoru V-REP.

Záznam chyb aplikace (tzv. *log*) se ukládá do souboru `.HILSimPlugin.log` do kořene umístění V-REP. Nastavení se ukládá do textového souboru do adresáře `.HILSimPlugin` v domovské složce uživatele. Tento soubor se automaticky vytváří při prvním spuštění aplikace.

Aplikace HIL Simulation Plugin se připojuje na vzdálené API simulátoru V-REP na portu 19997. Tento port je ve standardní instalaci V-REP automaticky nastaven v souboru `remoteApiConnections.txt`.

A.4 Základní knihovna aktuátorů a senzorů

Součástí klientské aplikace je knihovna čtyř zařízení, která lze použít při HIL simulaci robotů. Tato knihovna může být rozšířena bez zásahu do kódu pouhým přidáním souborů `ttm` a `json` do adresáře `models/hil-plugin/` v kořenovém adresáři instalace V-REP.

Každý skript, který je připojen k zařízení z této knihovny mimo výkonné části obsahuje také část inicializační (zdrojový kód A.1), která je pro všechna zařízení podobná. Definují se v ní jména signálů daného zařízení a také se zjišťují číselné přípony daného objektu, což zajistí, že ve scéně může být více objektů stejného druhu. Ve výkonné části skriptu je pak nutno celé jméno signálu poskládat takto: `name..suffix.."_"..signal`.

Další informace o vytváření skriptů pro simulaci jsou součástí online dokumentace simulátoru V-REP.

```
if (sim_call_type==sim_childscriptcall_initialization) then

   --zde musi byt jmeno daneho signalu prirazeno do promenne
  signalVariable="signalName"

  handle=simGetObjectAssociatedWithScript(sim_handle_self)
  suffix, name=simGetNameSuffix(simGetObjectName(handle))

  if (suffix<0) then
    suffix=''
  else
    suffix='#'..suffix
  end
end
```

Zdrojový kód A.1: Inicializační část Lua skriptů

A.4.1 LED

LED je tvořena objektem typu `Light`. Rozsah signálu je v rozmezí 0-1, kde 0 znamená zhasnuto a 1 znamená rozsvíceno.

```
if (sim_call_type==sim_childscriptcall_actuation) then

  local state,zero,df,sp=simGetLightParameters(handle)

  local signal=simGetIntegerSignal(name..suffix.."_"..active)

  if (signal) then
    simSetLightParameters(handle, signal, nil, df, sp)
  end
end
```

Zdrojový kód A.2: Lua skript pro LED aktuátor

```
{
  "name": "LED",
  "signals": [
    {
      "name": "active",
      "direction": "INPUT",
      "type": "INTEGER",
      "min": 0,
      "max": 1,
      "mode": "DIGITAL"
    }
  ]
}
```

Zdrojový kód A.3: Konfigurační soubor `led.json`

A.4.2 Stejnosměrný elektrický motor

Elektrický motor je tvořen kloubovým spojem, který dovoluje otáčení dle jedné osy `revoluteJoint`. Využívá ke svému běhu tři vstupní signály: `enable`, `A`, `B`. Signál `enable` je typu `PWM` a má rozsah 0 - 1024. Čím vyšší je hodnota tohoto signálu, tím rychleji se hřídel motoru otáčí.

Signály `A` a `B` jsou typu `DIGITAL` a řídí směr otáčení dle tabulky A.1. Toto chování odpovídá stejnosměrnému motoru připojenému k motorovému řadiči, jako je např. L298B[2].

[2] výrobce STMicroelectronics

chování	enable	A	B
otáčení vpřed	>0	HIGH	LOW
otáčení vzad	>0	LOW	HIGH
motorová brzda	>0	HIGH	HIGH
	>0	LOW	LOW
volnoběh	0	HIGH/LOW	HIGH/LOW

Tabulka A.1: Chování motoru v závislosti na vstupních signálech

```
if (sim_call_type==sim_childscriptcall_actuation) then

  local enable=simGetFloatSignal(name..suffix.."_"..enable)
  local a=simGetIntegerSignal(name..suffix.."_"..a)
  local b=simGetIntegerSignal(name..suffix.."_"..b)

  if (enable and a and b) then
    if (a == b) then
      direction = 0
    elseif (a > b) then
      direction = 1
    elseif (a < b) then
      direction = -1
    end

    simSetJointTargetVelocity(handle, 0.02*enable*direction)
  end
end
```

Zdrojový kód A.4: Lua skript pro elektrický motor

```
{
  "name": "electricMotor6V",
  "description": "electric motor with motor driver L293B",
  "signals": [
    {
      "name": "enable",
      "direction": "INPUT",
      "type": "FLOAT",
      "min": 0,
      "max": 1024,
      "mode": "PWM"
    },
```

```
    {
      "name": "A",
      "direction": "INPUT",
      "type": "INTEGER",
      "min": 0,
      "max": 1,
      "mode": "DIGITAL"
    },
    {
      "name": "B",
      "direction": "INPUT",
      "type": "INTEGER",
      "min": 0,
      "max": 1,
      "mode": "DIGITAL"
    }
  ]
}
```

Zdrojový kód A.5: Konfigurační soubor `electricMotor6V.json`

A.4.3 Koncový spínač

Taktilní senzory využívají speciální vazbu (tzv. `forceSensor`), která dynamicky sváže pohyblivou a nepohyblivou část spínače. V každém kroku simulace je měřena vzdálenost obou částí spínače. Pokud je nižší než stanovený práh, je spínač považován za sepnutý.

```
if (sim_call_type==sim_childscriptcall_sensing) then

  --relativni vzdalenost pohyblive a nepohyblive casti
  local p=simGetObjectPosition(surface, handle)

  --velikost vektoru
  local d=math.sqrt(p[1]*p[1]+p[2]*p[2]+p[3]*p[3])

  if (d < 0.006) then
    signal=1
  else
    signal=0
  end

  simSetIntegerSignal(name..suffix.."_"..onOffSignal, signal)
end
```

Zdrojový kód A.6: Lua skript pro koncový spínač

```
{
  "name" : "limitSwitch",
  "signals" : [ {
    "name" : "pressed",
    "direction" : "OUTPUT",
    "type" : "INTEGER",
    "min": 0,
    "max": 1,
    "mode" : "DIGITAL"
  } ]
}
```

Zdrojový kód A.7: Konfigurační soubor `limitSwitch.json`

A.4.4 Dálkový senzor HC-S04

Dálkový senzor je v simulátoru zprostředkován pomocí jednoho z vestavěných senzorů vzdálenosti s paprskovým snímacím prostorem[3]. Parametry byly nastaveny tak, aby co nejpřesněji odpovídaly skutečnému senzoru HC-S04. Signál vzdálenosti je odesílán na mezičlánek, kde však musí být převeden speciálními instrukcemi tak, aby mezičlánek simuloval skutečné chování HC-S04.

Reálný senzor zachycuje vzestupné hrany napěťového signálu na pinu označeném `TRIGGER` a poté vysílá ultrazvukové vlny. Ty se odrážejí od překážek a senzor je po návratu zachytí. Délka zpoždění mezi odesláním vln a jejich detekcí určuje délku pulzu, který lze číst na pinu `ECHO`. Pro simulaci tohoto senzoru je nutná velmi rychlá odezva, a proto výše popsaný mechanismus zajišťuje mezičlánek. Kromě konfiguračního souboru je součástí tohoto senzoru také soubor speciálních instrukcí pro čtení a zápis s příponou `hlr`.

```
if (sim_call_type==sim_childscriptcall_sensing) then

  local result,distance=simReadProximitySensor(handle)

  -- saturace
  if (result == 0 or distance > 2) then
    distance = 2
  end

  noiseFactor=0.001
  distance=noiseFactor*math.random() + distance -- prida sum
  simSetFloatSignal(name..suffix.."_"..distanceSignal, 100*distance)
end
```

Zdrojový kód A.8: Lua skript pro HC-S04

[3]angl. *Ray proximity sensor*

```json
{
  "name": "HCS04",
  "signals": [
    {
      "name": "distance",
      "direction": "OUTPUT",
      "type": "FLOAT",
      "min": 0,
      "max": 200,
      "handler": {
        "name": "HCS04_distance",
        "pins": [
          {
            "name": "trigger",
            "mode": "INTERRUPT",
            "direction": "INPUT"
          },
          {
            "name": "echo",
            "mode": "DIGITAL",
            "direction": "OUTPUT"
          }
        ]
      }
    }
  ]
}
```

Zdrojový kód A.9: Konfigurační soubor HSC04.json

```
##DECLARATIONS##
volatile boolean %b0% = false;
boolean %b1% = false;
boolean %b2% = false;

##SETUP##
pinMode(%echo%, OUTPUT);
pinMode(%trigger%, INPUT);
cli();
PCICR |= 1 << %triggerPCINTE%;
PCMSK%triggerPCINTE% |= %triggerPCMSK%;
sei();

##LOOP##
static unsigned long %m1% = 0;
static unsigned long %m2% = 0;
static int %d2% = 0;
```

```
if (%b0%) {
  %b1% = true;
  %m1% = micros();
  %b0% = false;
}

if (%b1% && (micros() - %m1%) >= 5000) {  // cekej 5 ms
  digitalWrite(%echo%, HIGH);
  %m2% = micros();
  %d2% = (int) (%distance% * 58.2);  // vypocitani delka pulzu
  %b2% = true;
  %b1% = false;
}

if (%b2% && (micros() - %m2%) >= %d2%) {  // cekej delku pulzu
  digitalWrite(%echo%, LOW);
  %b2% = false;
}

##FUNCTIONS##
ISR(PCINT%triggerPCINTE%_vect) {
  if (digitalRead(%trigger%)==1 && !%b1% && !%b2%) {
    %b0% = true;  // byl zaznamenan trigger, nastav flag
  }
}
```

Zdrojový kód A.10: Speciální instrukce pro ultrasonický senzor

Příloha B

Zdrojový kód

B.1 Program řídící jednotky robota

```
#define echo 9
#define trigger 10

#define switchL 2
#define switchR 3

#define motorL 5
#define motorL_A 11
#define motorL_B 12

#define motorR 6
#define motorR_A 4
#define motorR_B 7

#define led 13

long distance = 0;
long prev = 0;
int spd = 100;

boolean b = false;
boolean blinkb = false;
boolean stopped = false;
boolean volatile hit = false;

void setup() {
  Serial.begin(115200);
  pinMode(trigger, OUTPUT);
  pinMode(echo, INPUT);
  pinMode(motorL_A, OUTPUT);
  pinMode(motorL_B, OUTPUT);
  pinMode(motorR_A, OUTPUT);
```

```
  pinMode(motorR_B, OUTPUT);
  pinMode(led, OUTPUT);

  attachInterrupt(switchL-2, obstacleHit, RISING);
  attachInterrupt(switchR-2, obstacleHit, RISING);
}

void loop() {
  digitalWrite(trigger, LOW);
  delayMicroseconds(2);
  digitalWrite(trigger, HIGH);
  delayMicroseconds(30);
  digitalWrite(trigger, LOW);

  long duration = pulseIn(echo, HIGH);
  distance = duration / 58.2;

  if (b && millis() - prev > 2000) {   // robot ma prestat couvat
    b = false;
    hit = false;
  }

  if (!b) {                          // robot prave necouva
    if (hit) {                       // pokud byl detekovan naraz
      back();                        // zacni couvat
      b = true;
      prev = millis();
      hit = false;
      blinkb = false;
    } else if (distance < 150) {     // pokud je pred robotem prekazka
      if (!stopped) {                // nejprve zastav otaceni
        stopf();
        stopped = true;
        delay(200);                  // po zastaveni cekej 200 ms
      } else {
        forward();                   // jed vpred
        blinkb = false;
      }
    } else {
      left();                        // jinak se na miste otacej doleva
      stopped = false;
      blinkb = true;
    }
  }

  delay(50);                         // delay kvuli senzoru vzdalenosti

  if (blinkb) {                      // blikani
    digitalWrite(led, !digitalRead(led));
```

```
  }
}

void obstacleHit() {
  Serial.println(millis());
  hit = true;
}

void stopf() {
  digitalWrite(motorL_A, HIGH);
  digitalWrite(motorL_B, HIGH);
  digitalWrite(motorR_A, HIGH);
  digitalWrite(motorR_B, HIGH);
}

void forward() {
  analogWrite(motorL, spd);
  digitalWrite(motorL_A, HIGH);
  digitalWrite(motorL_B, LOW);

  analogWrite(motorR, spd);
  digitalWrite(motorR_A, LOW);
  digitalWrite(motorR_B, HIGH);
}

void back() {
  analogWrite(motorL, spd);
  digitalWrite(motorL_A, LOW);
  digitalWrite(motorL_B, HIGH);

  analogWrite(motorR, spd);
  digitalWrite(motorR_A, HIGH);
  digitalWrite(motorR_B, LOW);
}

void left() {
  analogWrite(motorL, 0.25*spd);
  digitalWrite(motorL_A, LOW);
  digitalWrite(motorL_B, HIGH);

  analogWrite(motorR, 0.25*spd);
  digitalWrite(motorR_A, LOW);
  digitalWrite(motorR_B, HIGH);
}
```

Zdrojový kód B.1: Program řídící jednotky robota

B.2 Ukázka vygenerovaného programu pro mezičlánek

```
//====================================================+
//                  GENERATED MACROS                  |
//====================================================+
#define FAST_WRITE(pin, port, value) \
  do { \
    if (value) port |= _BV(pin); \
    else port &= ~_BV(pin); \
  } while (0)
#define FAST_READ(pin, port) ((port & (1 << pin)) == (1 << pin))

//====================================================+
//                  GENERATED DECLARATIONS            |
//====================================================+
int INT_MIN = -32768;
int INT_MAX = 32767;
byte START = 0xFF;
byte packetSize = 1;
byte SIGNALS[] = {11};

boolean volatile write = false;

// PIN VARIABLES
byte p_LED_activeIN = 2;
byte p_limitSwitch_pressedOUT = 3;

// SIGNAL VARIABLES
int s_limitSwitch_pressedOUT = 0;
int s_LED_activeIN = 0;

//====================================================+
//                  GENERATED SETUP                   |
//====================================================+
void setup() {
  Serial.begin(115200);
  pinMode(p_limitSwitch_pressedOUT, OUTPUT);
  pinMode(p_LED_activeIN, INPUT);

  setupInterrupt();
}

//====================================================+
//                  GENERATED LOOP                    |
//====================================================+
void loop() {
  s_LED_activeIN = FAST_READ(PD2, PIND);
  FAST_WRITE(PD3, PORTD, s_limitSwitch_pressedOUT);
```

```
  if (write) {
    write = false;
    writeToSerial();
  }
}

//=======================================================+
//                    GENERATED SERIAL WRITING           |
//=======================================================+
void writeToSerial() {
  byte buf[3];
  byte lo, hi;
  int v;
  Serial.write(0xFF);

  v = (int) map(s_LED_activeIN, 0, 1, INT_MIN, INT_MAX);
  buf[0] = 10;
  buf[1] = (byte) ((v >> 8) & 0xFF);
  buf[2] = (byte) (v & 0xFF);
  Serial.write(buf, 3);
}

ISR(TIMER1_COMPA_vect) {
  write = true;
}

void setupInterrupt() {
  noInterrupts();
  TCCR1A = 0;
  TCCR1B = 0;
  TCNT1  = 0;
  OCR1A = 313;
  TCCR1B |= (1 << WGM12);
  TCCR1B |= (1 << CS12);
  TIMSK1 |= (1 << OCIE1A);
  interrupts();
}

//=======================================================+
//                    GENERATED SERIAL READING           |
//=======================================================+
void serialEvent() {
  while (Serial.available() > 0)
    processByte(Serial.read());
}

int signalCounter = 0;
boolean started = false;
boolean high = false;
```

```
byte key = START;
byte hi = 0;
byte lo = 0;

void processByte(byte b) {
  if (!started && b == START) {
    started = true;
  } else if (key != START) {
    if (!high) {
      hi = b;
      high = true;
    } else {
      lo = b;
      high = false;
      useSignal();
      signalCounter++;
      key = START;
    }
  } else if (started) {
    if (findSignal(b)) {
      key = b;
    } else {
      started = b == START;
    }
  }

  if (signalCounter == packetSize) {
    signalCounter = 0;
    started = false;
  }
}

boolean findSignal(byte b) {
  for (int i = 0; i < packetSize; i++)
    if (SIGNALS[i] == b) return true;

  return false;
}

void useSignal() {
  int value = ((hi & 0xFF) << 8) | (lo & 0xFF);

  switch (key) {
    case 11 :
      s_limitSwitch_pressedOUT = map(value, INT_MIN, INT_MAX, 0, 1);
      break;
  }
}
```

Zdrojový kód B.2: Příklad programu pro mezičlánek

Literatura

[1] arduino.cc. *Arduino UNO Reference*. 2015. `http://arduino.cc/en/main/arduinoBoardUno` [Online; cit. 4.4.2015].

[2] *8-bit Atmel Microcontroller with In-System Programmable Flash*. Atmel, 2011.

[3] M. F. E. Rohmer, S. P. N. Singh. V-rep: a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[4] W. Ren and T. F. S. University. *Accuracy Evaluation of Power Hardware-in-the-loop (PHIL) Simulation*. Florida State University, 2007. ISBN 9780549467694.