Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Курсовая работа

Колесная платформа
по дисциплине «Микропроцессорные системы»

Выполнили: Ферапонтов М.В.
Савчук А.А.
Дорошин Д.А.
Луцай П.П.
Артеев Д.Д.
Яровой В.Д.
Группа: гр. 3530904/00104

Проверил: Круглов С.К.

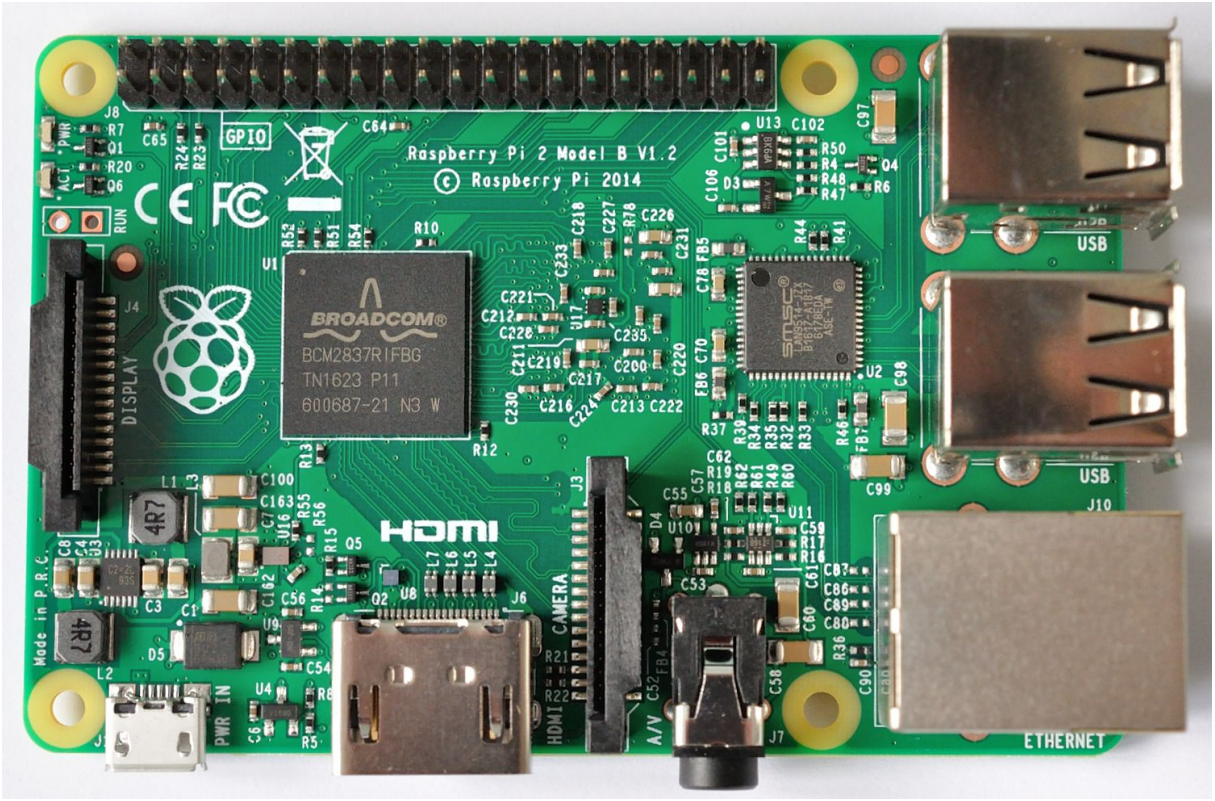
Санкт-Петербург
2022

Содержание

1 Аппаратная реализация

1.1 Raspberry pi 2b+

При разработке был использован Raspberry Pi 2 model B+.



Характеристики:

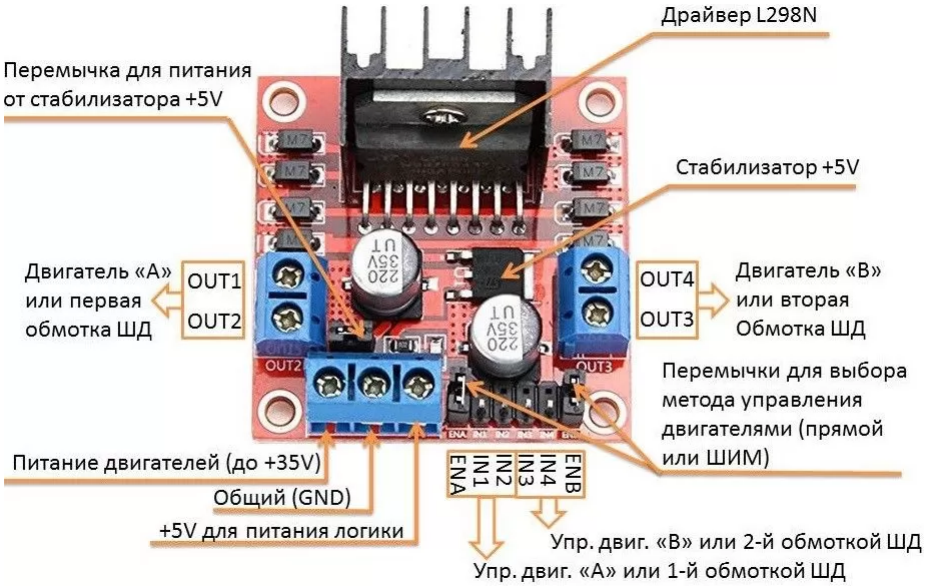
- Процессор - ARM Cortex-A7 CPU 900MHz
- Количество ядер - 4
- Оперативная память - 1Gb

1.2 Используемые детали и модули

Были использованы детали из набора Arduino

1. Драйвер управления движения моторов L289N
2. Колеса 4 шт.
3. Кнопка для управления движения 4шт.
4. Резистр 22Ω 4 шт
5. Корпус
6. Плата расширения

Драйвер L298N используется для многофункционального управления двигателями постоянного тока. Схема модуля, состоящая из двух H-мостов, позволяет подключать к нему один биполярный шаговый двигатель или одновременно два щёточных двигателя постоянного тока. При этом есть возможность изменять скорость и направление вращения моторов. Управление осуществляется путём подачи соответствующих сигналов на командные входы, выполненные в виде штыревых контактов. На рисунке показан внешний вид модуля с кратким описанием всех его составляющих.



1.3 Широтно-импульсная модуляция

В драйвере L289N *ENA*, *ENB* – контакты для активации/деактивации первого и второго двигателей или соответствующих обмоток ШД. Подача логической единицы на эти контакты разрешает вращение двигателей, а логический ноль – запрещает. Для изменения скорости вращения щёточных моторов на эти контакты подаётся ШИМ-сигнал.

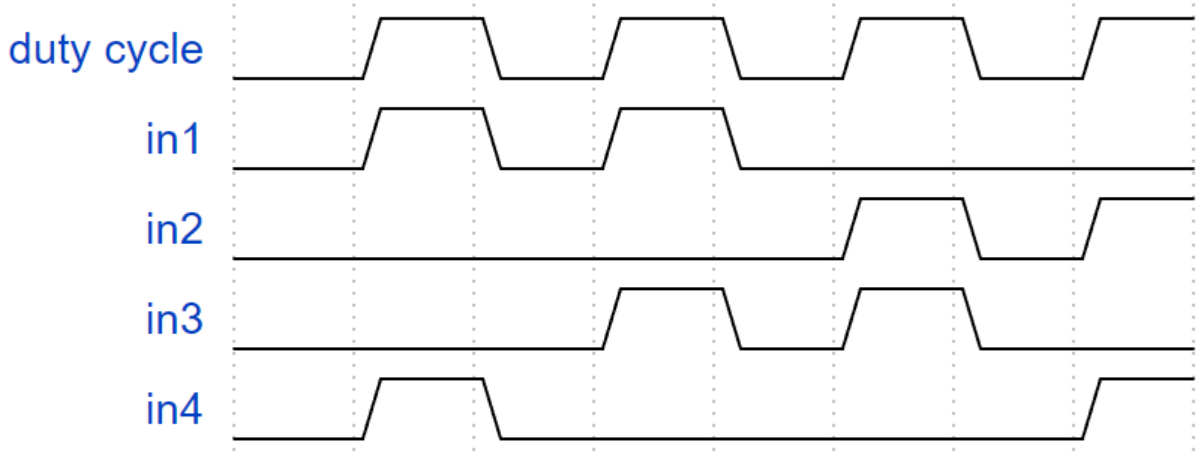
ШИМ - метод уменьшения средней мощности, передаваемой электрическим сигналом, путем эффективного разделения его на отдельные части. Среднее значение напряжения (и тока), подаваемого на нагрузку, регулируется быстрым включением и выключением переключателя между питанием и нагрузкой. Чем дольше переключатель включен по сравнению с периодами выключения, тем выше общая мощность, подаваемая на нагрузку.

```
1   pub fn set_frequency(&self, frequency: f64, duty_cycle: f64) -> Result<()>
```

- frequency - частота
- duty cycle - скважность, задается числом с плавающей точкой в промежутке между 0.0 и 1.0.

В нашей программе изначально значение скважности установлено на 0, что означает что на двигатели будет подаваться в среднем 0V. Никакого движения не происходит.

При нажатии на конпку мы подаем сигналы на левые и правые двигатели и устанавливаем скважность на 1, что означает что на двигатели будет подаваться в среднем 5V.

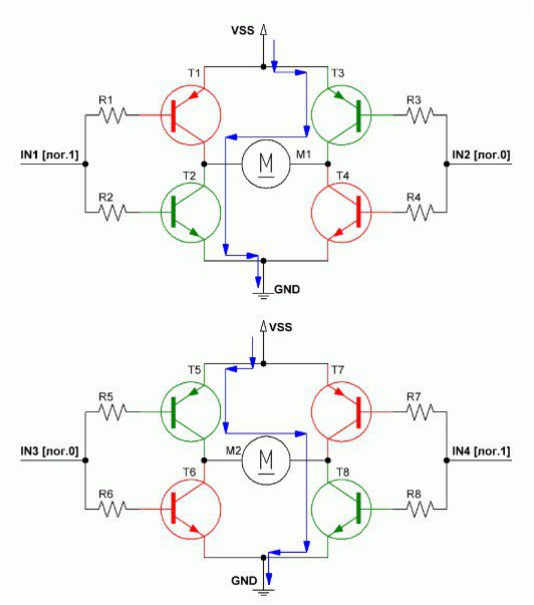


1.4 Питание

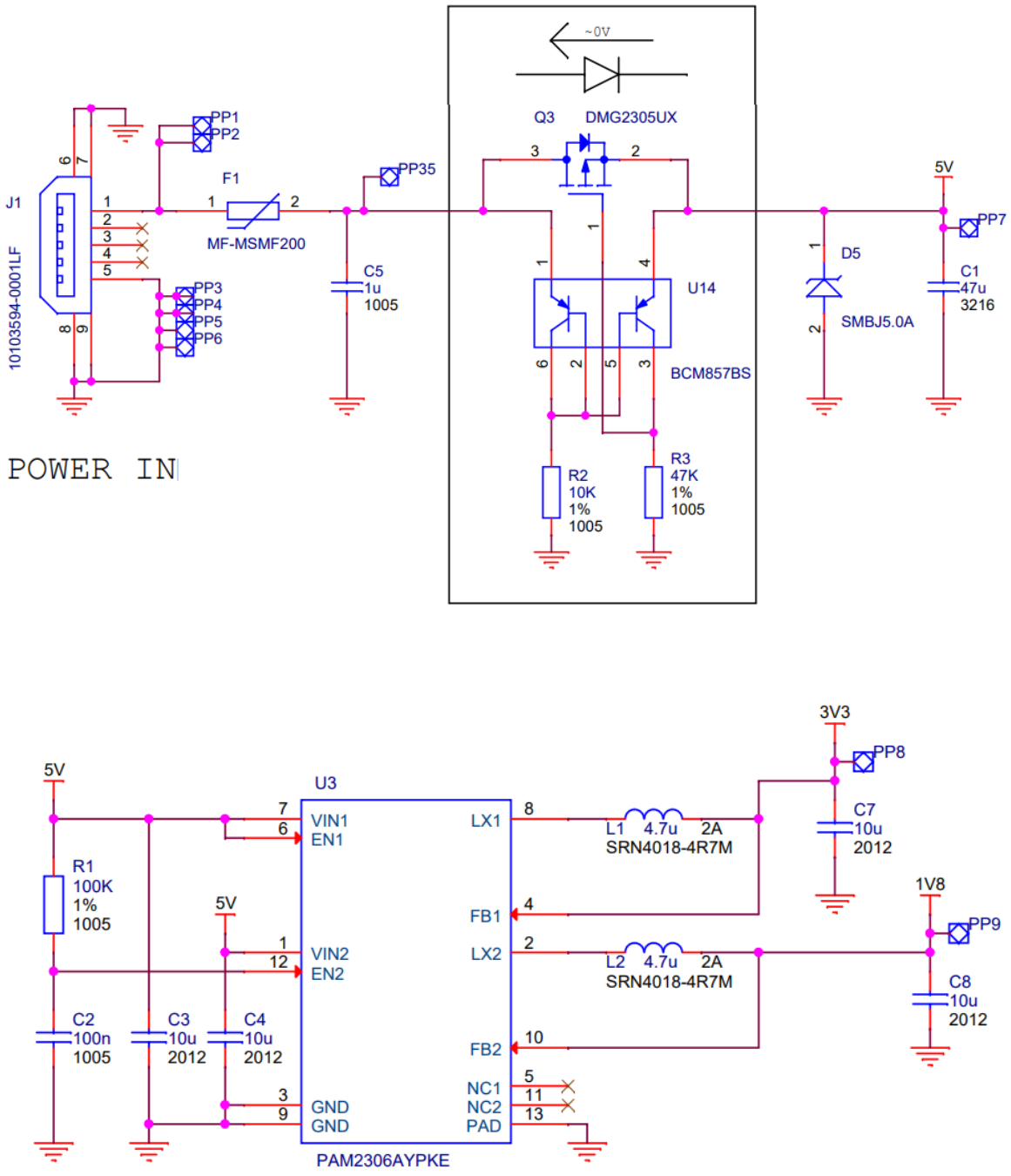
1. L289N - Логическое устройство - 5V
2. L289N - Моторы - 12V
3. Кнопка - 3.3V
4. Raspberry pi 2b+ - 5V

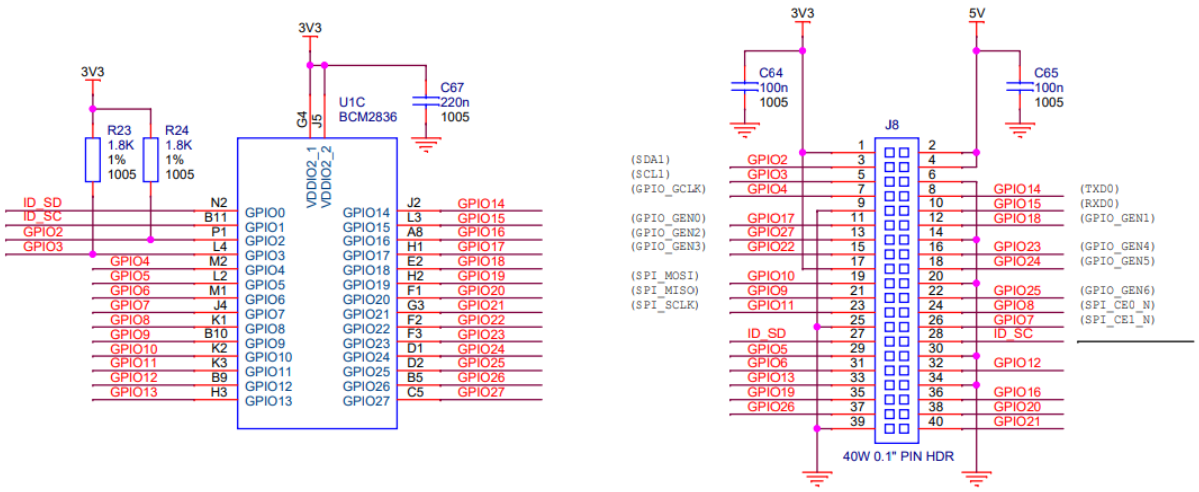
2 Схемы

2.1 Принципиальная схема драйвера L289N



2.2 Принципиальная схема Raspberry Pi

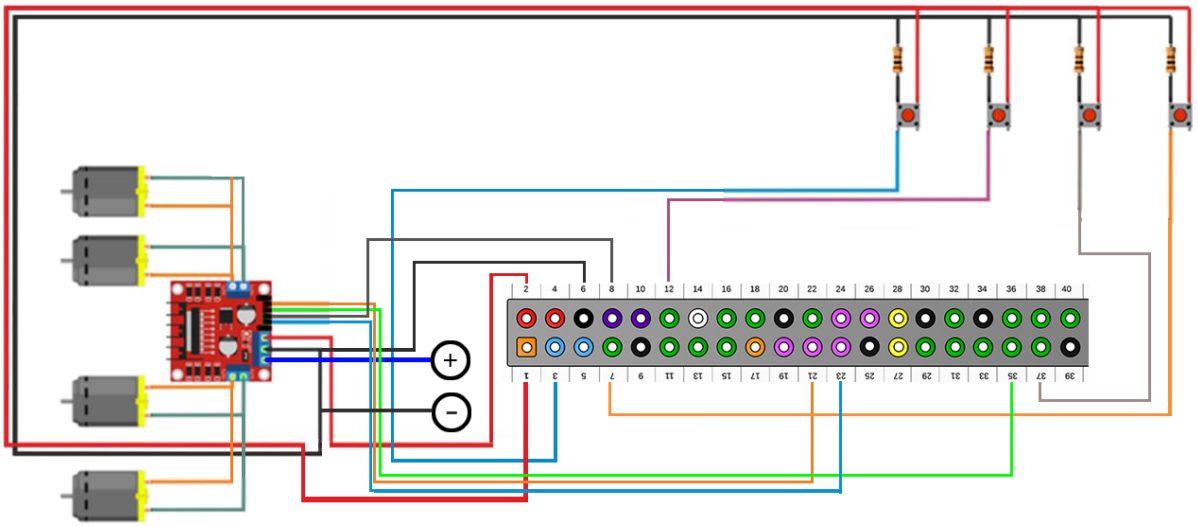


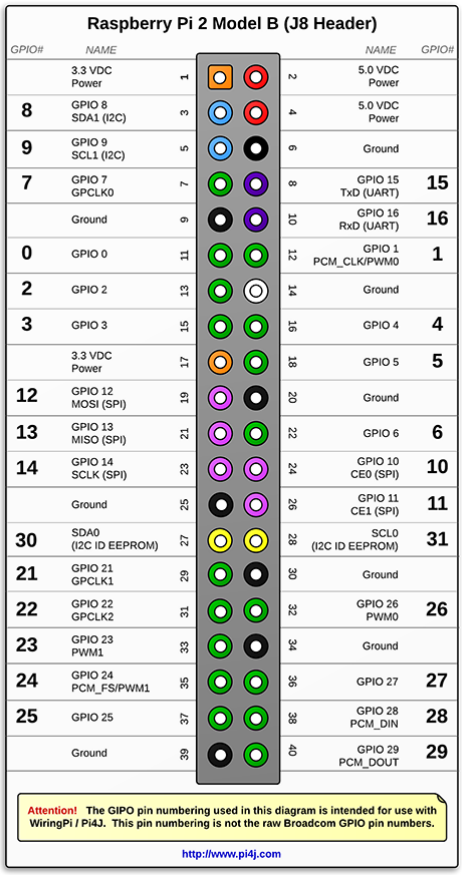


Полную принципиальную схему можно найти здесь.

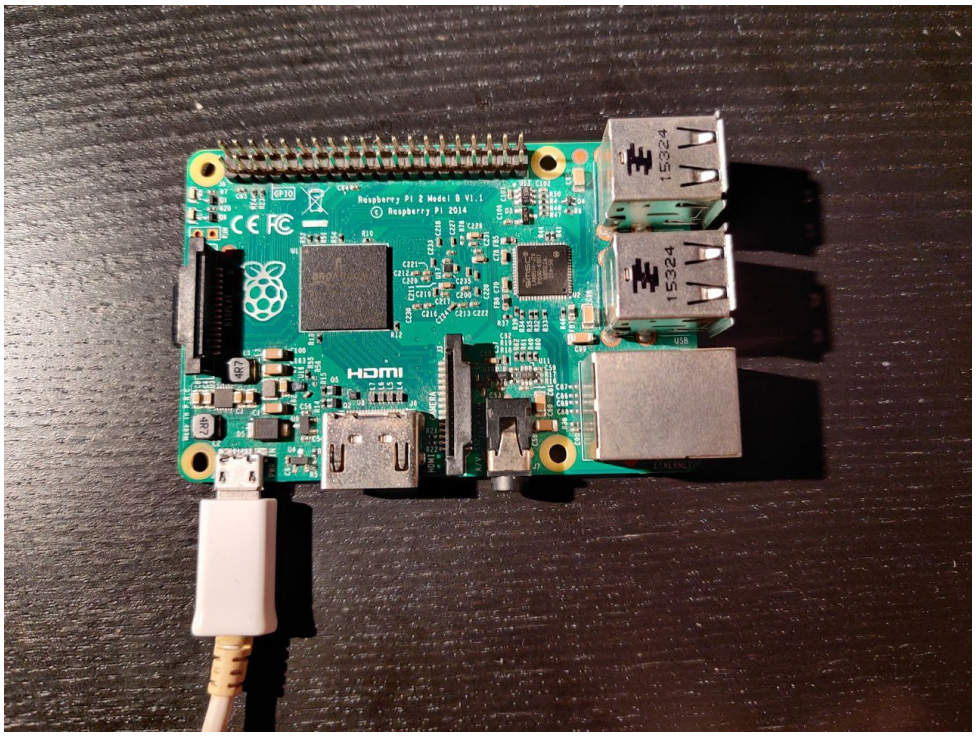
2.3 Схема подключений

Пин №1 подает 3.3 V на кнопки. Пин №2 подает 5 V питания на логическое устройство. Моторы питаются из внешнего зарядного устройства.





Raspberry Pi питает от micro-USB, который подает 5V.



3 Программное обеспечение

3.1 Используемые инструменты

1. Rust - 1.25.1 - Язык программирования
2. rppal - 0.14.0 - Raspberry Pi Peripheral Access Library. Предоставляет доступ к GPIO у Raspberry Pi.

3.2 Код программы

```
/* main.rs */
use std::thread;
use std::time::Duration;
use color_eyre::Result;

use rppal::gpio::Gpio;
use rppal::pwm::{Pwm, Channel};
use rppal::system::DeviceInfo;

mod engine;
mod input;

fn main() -> Result<()> {
    println!("Working␣on␣a␣{}.", DeviceInfo::new()?.model());

    let mut engine_left = engine::Engine{
        forward_pin: Gpio::new()?.get(14)?.into_output(),
        backward_pin: Gpio::new()?.get(15)?.into_output(),
        pwm: Pwm::new(Channel::Pwm0)?,
    };

    let mut engine_right = engine::Engine{
        forward_pin: Gpio::new()?.get(26)?.into_output(),
        backward_pin: Gpio::new()?.get(13)?.into_output(),
        pwm: Pwm::new(Channel::Pwm1)?,
    };
    let mut ddc = engine::DigitalDirectionController{
        left: engine_left,
        right: engine_right,
    };
    let mut input = input::ButtonInput{
        forward: Gpio::new()?.get(8)?.into_input(),
        backward: Gpio::new()?.get(1)?.into_input(),
        left: Gpio::new()?.get(25)?.into_input(),
        right: Gpio::new()?.get(7)?.into_input(),
    };
```

```
37
38      loop {
39          let direction = input.get_direction();
40          println!("Direction: {:?}", direction);
41          ddc.direction(direction)?;
42          thread::sleep(Duration::from_millis(50));
43      }
44  }
```

```
1   /* input.rs */
2   use color_eyre::Result;
3   use rppal::gpio::InputPin;
4
5   use super::engine::Direction;
6
7   pub struct ButtonInput {
8       pub forward: InputPin,
9       pub backward: InputPin,
10      pub left: InputPin,
11      pub right: InputPin,
12  }
13
14  impl ButtonInput {
15      pub fn get_direction(&mut self) -> Direction {
16          if self.forward.is_high() {
17              return Direction::Forward;
18          };
19          if self.left.is_high() {
20              return Direction::Left;
21          };
22          if self.right.is_high() {
23              return Direction::Right;
24          };
25          if self.backward.is_high() {
26              return Direction::Backward;
27          };
28          Direction::Stop
29      }
30  }
```

```
1   /* engine.rs */
2   use color_eyre::Result;
3   use rppal::gpio::OutputPin;
4   use rppal::pwm::Pwm;
5
6   pub struct Engine {
7       pub forward_pin: OutputPin,
```

```
    pub backward_pin: OutputPin,
    pub pwm: Pwm,
}

impl Engine {
    pub fn set(&mut self, direction: f64) -> Result<()> {
        if !self.pwm.is_enabled()? {

            self.pwm.set_frequency(20.0, 0.0)?;
            self.pwm.enable()?;
        }
        let direction = direction.clamp(-1.0, 1.0);
        if direction >= 0.0 {
            self.forward_pin.set_high();
            self.backward_pin.set_low();
        } else {
            self.forward_pin.set_low();
            self.backward_pin.set_high();
        }
        self.pwm.set_duty_cycle(direction.abs())?;
        Ok(())
    }
}

impl Drop for Engine {
    fn drop(&mut self) {
        self.forward_pin.set_low();
        self.backward_pin.set_low();
    }
}

pub struct AnalogDirectionController {
    pub left: Engine,
    pub right: Engine,
}

impl AnalogDirectionController {
    pub fn direction(&mut self, gradient: f64, radial: f64) -> Result<()> {
        let gradient = gradient.clamp(-1.0, 1.0);
        let radial = radial.clamp(-1.0, 1.0);
        Ok(())
    }
}

#[derive(Debug)]
pub enum Direction {
```

```
    Stop,
    Forward,
    Backward,
    Left,
    Right
}

pub struct DigitalDirectionController {
    pub left: Engine,
    pub right: Engine,
}

impl DigitalDirectionController {
    pub fn direction(&mut self, direction: Direction) -> Result<()> {
        match direction {
            Direction::Stop => {
                self.left.set(0.0)?;
                self.right.set(0.0)
            },
            Direction::Forward => {
                self.left.set(1.0)?;
                self.right.set(1.0)
            },
            Direction::Backward => {
                self.left.set(-1.0)?;
                self.right.set(-1.0)
            },
            Direction::Left => {
                self.left.set(-1.0)?;
                self.right.set(1.0)
            },
            Direction::Right => {
                self.left.set(1.0)?;
                self.right.set(-1.0)
            },
        }
    }
}
```