КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА
Факультет комп'ютерних наук та кібернетики
Кафедра інтелектуальних програмних систем

Кваліфікаційна робота
на здобуття освітнього рівня магістра

за спеціальністю 121 Інженерія програмного забезпечення
на тему:

РОЗРОБКА КООПЕРАТИВНОЇ ОПЕРАЦІЙНОЇ СИСТЕМИ ДЛЯ МІКРОКОНТРОЛЕРА STM32 НА МОВІ RUST

Виконав студент 2-го курсу магістратури
Морозюк Антон

(підпис)

Науковий керівник:
доцент, кандидат технічних наук
Демківський Євген Олександрович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.

Студент

(підпис)

Роботу розглянуто й допущено до захисту на засіданні кафедри інтелектуальних програмних систем
«10» травня 2023 р.,
протокол №9
Завідувач кафедри

О. ПРОВОТАР

(підпис)

Київ – 2023

РЕФЕРАТ

Обсяг роботи 57 сторінок, 1 ілюстрація, 15 джерел посилань.

КООПЕРАТИВНА ОПЕРАЦІЙНА СИСТЕМА, МОВА ПРОГРАМУВАННЯ RUST, ПЛАНУВАЛЬНИК ЗАДАЧ, ПОРІВНЯННЯ МОВ ПРОГРАМУВАННЯ, ПРОГРАМУВАННЯ МІКРОКОНТРОЛЕРІВ.

Об’єктом розробки є кооперативна операційна система для мікроконтролера STM32F091RCT6 (далі в тексті просто STM32), написана на мові програмування Rust.

Метою роботи є дослідження мови програмування Rust та порівняння її з мовою програмування C в контексті розробки програмних продуктів під так звані bare metal платформи, а також власне розробка кооперативної операційної системи під мікроконтролер STM32.

Методи розробки: програмування мовою програмування Rust.

Результатами роботи є кооперативна операційна система, написана мовою програмування Rust під мікроконтролер STM32, а також порівняння мов програмування Rust та C. Новизна отриманих результатів полягає в тому, що розроблена операційна система простіша за наявні аналоги, що позитивно впливає на її швидкодію, тоді як більшість операційних систем під мікроконтролери є досить складними, що впливає на їх швидкодію, їхня ж функціональність часто може бути надлишковою.

Програмний продукт, отриманий в результаті виконання даної роботи може бути використаний у подальшому як платформа для розробки під мікроконтролер STM32.

Також цей програмний продукт був уже використаний автором дослідження для створення датчика вологи та температури з екраном на базі цього ж мікроконтролера.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

Крос-компіляція – компіляція програмного коду під процесор, відмінний від того, на якому відбувається крос-компіляція.

Дебагер (також налагоджувач) – програма, що використовується для тестування та виправлення вад інших програм [1].

ВСТУП

Оцінка сучасного стану об’єкта розробки. В сучасному світі мікроконтролери використовуються в широкому спектрі пристроїв, починаючи від медичних та інструментальних пристроїв і закінчуючи великими системами управління автомобілями та промисловими установками. Для розробки таких систем необхідна ефективна та безпечна операційна система, яка забезпечує швидкість, надійність та безпеку виконання різноманітних завдань. Розробку IoT проектів зазвичай починаються з вибору конкретних пристроїв та плат за допомогою яких буде створюватись кінцевий продукт. Важливим критерієм для вибору плати від конкретного виробника є не тільки її технічні характеристики, а й підтримка зі сторони програмного забезпечення – чи підтримують її популярні інтегровані середовища розробки, чи є для неї бібліотека на потрібній мові програмування, чи можна на ній запустити операційну систему(якщо вона потрібна).

Питання необхідності операційної системи також досить спірне – часто необхідні лише мінімальні її атрибути(менеджмент пам’яті і запуск декількох задач) і не хочеться заради цього “тягнути” у свій проект якусь величезну операційну систему.

З іншого боку, написання власної операційної системи це дуже цікавий, складний та корисний проект, яким займалося чимало програмістів. Варто сказати, що є чимало різних варіантів операційних систем, які можна використати для власних проектів, але в даній роботі буде розглядатись створення мінімалістичної операційної системи, що дозволяє запускати її на пристроях з дуже обмеженими ресурсами.

Актуальність роботи та підстави для її виконання. Робота є актуальною, оскільки існує не так багато альтернативних операційних систем на мові Rust, а наявні або занадто складні (як RTOS), або не мають всього необхідного функціоналу, або не зрозуміло на якому етапі розробки вони знаходяться і чи будуть колись завершені.

Мета й завдання роботи. Метою роботи є створення власної кооперативної операційної системи для мікроконтролера STM32F091RCT6. Завдання роботи полягає у розробці структури операційної системи, яка забезпечує безпеку виконання задач, має низьку вартість та споживання ресурсів, а також може бути використана для широкого спектру проектів.

Об'єкт, методи й засоби розробки. Об’єктом розробки є операційна система для мікроконтролера STM32. Методом розробки є програмування на мові Rust. В процесі розробки було використано rustup та cargo для крос-компіляції вихідного коду під процесор ARM, дебагер openocd, а також термінал minicom для перегляду UART повідомлень від мікроконтролера. Програмний код було написано мовою Rust, також невелика частина написана на асемблері.

Можливі сфери застосування. Загалом, ця операційна система розробляється для використання у власних проектах (наприклад, з використанням цієї операційної системи було зроблено датчик температури та вологості з екраном), але не виключено і застосування в інших цілях для прискорення написання якогось проекту під мікроконтролер STM32, оскільки в даній операційній системі вирішено ряд проблем, з якими доведеться зіткнутися у випадку відмови від використання інших, більш громіздких операційних систем, як от RTOS.

РОЗДІЛ 1. ЗАГАЛЬНИЙ ОПИС МОВИ ПРОГРАМУВАННЯ RUST

1.1. Загальні концепти мови програмування Rust

За замовчуванням змінні в мові програмування Rust є константами та оголошуються наступним чином:

let x = 5;

Для того, щоб значення цієї змінної можна було поміняти далі, її слід визначити як mutable:

let mut x = 5;

Також є константи, які позначаються ключовим словом const. На відміну від змінних, з константами неможливо використати ключове слово mut.

Мова програмування Rust має різноманітні типи даних, які можуть бути класифіковані в декілька груп в залежності від їх характеристик:

- Числові типи

Rust підтримує числові типи, такі як цілочисельні типи (i8, i16, i32, i64, i128, isize) та беззнакові типи (u8, u16, u32, u64, u128, usize). Ці типи можна використовувати для зберігання цілих чисел зі знаком та без.

Наприклад, щоб створити змінну типу i32, необхідно використати наступний синтаксис:

let x: i32 = 42;

- Типи з плаваючою крапкою

Rust також підтримує типи з плаваючою крапкою, такі як f32 та f64. Ці типи даних використовуються для зберігання дійсних чисел з плаваючою крапкою.

let y: f32 = 3.14;

- Булевий тип

У Rust є тип даних bool, який може мати значення true або false. Цей тип даних використовується для представлення логічних значень.

let is_rust_awesome: bool = true;

- Символьний тип

Rust також має тип даних char, який використовується для зберігання одного символу Unicode.

let letter: char = 'a';

- Кортежі та масиви

Типи даних, що дозволяють зберігати колекції об’єктів. Кортежі та масиви мають фіксовану довжину, але кортежі можуть зберігати в собі одночасно елементи кількох типів, тоді як масив можуть зберігати лише елементи одного типу.

Створення та доступ до елементу кортежу:

let x: (i32, f64, u8) = (500, 6.4, 1);

let five_hundred = x.0;

Створення та доступ до елементу масиву:

let a = [1, 2, 3, 4, 5];

let first = a[0];

- Структури

У мові програмування Rust структури є схожими до структур в C та інших мовах програмування. Структури – це спосіб об'єднання декількох пов'язаних даних в одній конструкції, яка може бути передана функціям та методам.

Структури в Rust оголошуються за допомогою ключового слова struct та мають ім'я, поля та можуть мати реалізації методів. Наприклад, так виглядає оголошення структури, яка представляє точку на двовимірній площині:

```
struct Point {
    x: i32,
```

```
    y: i32,
}
```

У цьому оголошенні ми визначаємо структуру з ім'ям Point, яка має два поля типу i32: x та y.

Структури в Rust також можуть мати методи, що дозволяють змінювати їх стан. Наприклад, метод translate для структури Point, який зміщує точку на вказану кількість пікселів:

```
impl Point {
    fn translate(&mut self, dx: i32, dy: i32) {
        self.x += dx;
        self.y += dy;
    }
}
```

У цьому прикладі ми використовуємо ключове слово impl, щоб оголосити методи для структури Point. Метод translate приймає вказівник на структуру self та два параметри dx та dy для зміщення точки на відповідну кількість пікселів.

Однією з важливих можливостей структур в Rust є можливість використання derive для автоматичної генерації інших методів та функцій. Наприклад, можна автоматично згенерувати методи для порівняння структур за допомогою PartialEq:

```
#[derive(PartialEq)]
struct Point {
    x: i32,
    y: i32,
}
```

Це дозволяє порівнювати дві структури Point за допомогою оператора ==.

Структури в Rust також можуть мати різні види полів, такі як поля з власністю, поля з вказівниками на інші об'єкти та навіть поля з функціональними типами даних.

Структури в Rust мають кілька переваг порівняно з C, зокрема, безпечний доступ до пам'яті, захист від переповнень буферів та гарантію безпеки в паралельних середовищах. Однак, Rust також має більш виразну систему типів та набір функціональних можливостей, таких як дженерики та замикання, що роблять його більш потужним і ефективним засобом для розробки різних типів програм.

- Вказівники та нульові вказівники

У Rust є вказівники та нульові вказівники, які можна використовувати для маніпулювання пам'яттю.

У Rust, вказівники використовуються для вказівки на певні об'єкти в пам'яті. Існує два типи вказівників: & (посилання) та * (вказівник).

```
// Оголошуємо змінну і типу i32
let i = 42;
// Створюємо посилання на змінну i
let i_ref = &i;
// Створюємо вказівник на змінну i
let i_ptr = &i as *const i32;
```

8. Зрізи

Зрізи використовуються для зберігання змінної кількості значень одного типу. Зрізи можуть бути створені з використанням індексів масиву або з використанням іншого зрізу.

```
let array: [i32; 3] = [1, 2, 3];
let slice: &[i32] = &array[1..];
```

9. Рядки

У Rust, рядки представлені як вектори символів Unicode. Ці рядки можна створити за допомогою літералів рядків або за допомогою векторів символів.

```
let string_literal = "Hello, world!";
let string_vector: Vec<char> = vec!['H', 'e', 'l', 'l', 'o', ', ', 'w', 'o', 'r', 'l', 'd', '!'];
```

- Опціональні типи

У Rust є опціональні типи, які використовуються для представлення значень, які можуть бути null або не null. Опціональні типи позначені за допомогою ключового слова Option<T>, де T - це тип, який може бути опціональним.

```
let optional_string: Option<String> = Some("Hello, world!".to_string());
let empty_string: Option<String> = None;
```

Основна ідея полягає в тому, що замість того, щоб повертати null, ми можемо повернути Option, який має два варіанти: Some (значення присутнє) та None (значення відсутнє).

Option може бути використаний для запобігання помилок виконання програми, що можуть виникати в результаті звернення до null-значень, забезпечуючи більш безпечну та просту обробку цих випадків. Зокрема, використання Option сприяє зменшенню кількості помилок виконання, що можуть бути важкими для виявлення та усунення.

Один з основних способів роботи з типом Option - використання match-виразу. Match-вираз дозволяє перевіряти різні варіанти значень, які можуть бути повернуті Option, та виконувати різні дії залежно від результату перевірки.

Наприклад, припустимо, що ми маємо функцію, яка повертає значення типу Option<i32>:

```
fn get_value(x: i32) -> Option<i32> {
    if x > 0 {
        Some(x * 2)
    } else {
```

```
        None
    }
}
```

Ця функція приймає цілочисельний параметр x та повертає значення типу Option<i32>, яке містить подвоєне значення x, якщо воно більше 0, та None у протилежному випадку.

Ми можемо використовувати match-вираз для обробки значень типу Option<i32>:

```
let x = get_value(5);

match x {
    Some(n) => println!("The value is {}", n),
    None => println!("The value is absent"),
}
```

У цьому прикладі ми використовуємо match-вираз для перевірки значення x. Якщо значення x є Some, то ми можемо отримати його значення та виконати дії з ним. У протилежному випадку, якщо значення x є None, ми можемо виконати інші дії, які будуть відповідати випадку відсутності значення.

Оскільки Option може бути використаний в будь-якому контексті, де можуть бути відсутні значення, він зазвичай використовується в функціях, які повертають результати, які можуть бути null.

Наприклад, можлива ситуація, коли ми хочемо отримати значення з рядка та конвертувати його у ціле число. Однак, якщо рядок не може бути конвертований у ціле число, то функція поверне None. Використовуючи тип Option, ми можемо забезпечити більш просту та безпечну обробку цього випадку:

```
fn string_to_int(input: &str) -> Option<i32> {
```

```
    match input.parse::<i32>() {
        Ok(n) => Some(n),
        Err(_) => None,
    }
}
```

У цьому прикладі функція string_to_int отримує рядок input та повертає значення типу Option<i32>. Якщо рядок може бути конвертований у ціле число, функція поверне значення Some з цим числом. Якщо рядок не може бути конвертований у ціле число, функція поверне значення None.

У випадку використання типу Option, match-вираз дозволяє просто та безпечно обробляти випадки, коли значення можуть бути відсутніми. Оскільки це один з ключових механізмів мови Rust для обробки помилок та виключень, знання та розуміння типу Option та match-виразу є важливим для ефективної розробки програм на Rust.

1.2. Володіння даними як механізм безпеки

Одним з ключових елементів мови Rust, який суттєво відрізняє її від більшості інших мов, а також є однією з важливих складових безпеки цієї мови є концепт володіння.

Концепція володіння (ownership) в Rust – це механізм, який контролює доступ до пам'яті та забезпечує безпеку виконання коду. У Rust, кожне значення має свого власника (owner), який відповідає за звільнення пам'яті, яку воно використовує, коли воно вже не потрібне.

Коли значення присвоюється змінній, змінна стає власником цього значення. Якщо змінна втрачає свою власність, пам'ять, на яку посилається це значення, буде звільнена автоматично.

Розглянемо декілька прикладів використання володіння в Rust:

1. Присвоєння значення змінній

```
let x = 42;
```

У цьому прикладі змінна x стає власником значення 42. Якщо змінна втрачає свою власність (наприклад, виходить за межі блоку коду, в якому була оголошена), пам'ять, на яку посилається це значення, буде автоматично звільнена.

2. Передача значення у функцію

```
fn print_number(x: i32) {
    println!("The number is {}", x);
}

let x = 42;
print_number(x);
```

У цьому прикладі змінна x передається у функцію print_number як аргумент. Коли функція закінчує свою роботу, змінна x втрачає свою власність, і пам'ять, на яку посилається це значення, автоматично звільняється.

3. Повернення значення з функції

```
fn create_string() -> String {
    let s = String::from("Hello, world!");
    s
}
```

```
let my_string = create_string();
println!("{}", my_string);
```

У цьому прикладі функція create_string створює новий об'єкт String та повертає його. Коли функція повертає значення, воно передається у змінну my_string, яка стає власником цього значення. Коли змінна my_string втрачає свою власність (наприклад, виходить за межі блоку коду, в якому була оголошена), пам'ять, на яку посилається це значення, буде автоматично звільнена.

Володіння в Rust дозволяє запобігти багатьом типам помилок, пов'язаних з керуванням пам'яті, таким як витік пам'яті, подвійне звільнення пам'яті та інші. Однак, цей механізм може призвести до деяких викликів, зокрема, до помилок позичання (borrowing errors), які можуть з'являтися при роботі з посиланнями на значення.

Помилки позичання виникають тоді, коли змінна, яка містить посилання на значення, спробує взяти його власність або виконати інші дії, які призведуть до втрати володіння. Ці помилки можна вирішити, використовуючи правила позичання та власності в Rust.

Правила позичання та власності в Rust:

1. Змінні можуть мати тільки один власник. Якщо змінна передається у функцію як аргумент, її власником стає функція.

```
fn print_number(x: i32) {
   println!("The number is {}", x);
}
let x = 42;
print_number(x);
// Змінна x не має власника після цього блоку коду
```

2. Змінні можуть мати декілька незмінних позичальників (immutable borrows) або один змінний позичальник (mutable borrow).

```
let mut x = 42;
let y = &x; // незмінних позичальник
let z = &x; // незмінних позичальник
let w = &mut x; // змінний позичальник
println!("{}", *y);
```

3. Позичальники повинні жити не довше, ніж їх власник.

```
{
   let x = 42;
   let y = &x;
   // Змінна x втрачає свою власність після цього блоку коду, тому
   // y більше не буде дійсним
}
{
   let mut x = 42;
   let y = &mut x;
   // Змінна x втрачає свою власність після цього блоку коду, тому
   // y більше не буде дійсним
}

{
   let mut x = 42;
   let y = &mut x;
   *y = 24;
   // Змінна x все ще має свого власника, тому y дійсний
}
```

```
{
    let mut x = 42;
    let y = &x;
    let z = &mut x; // Помилка компіляції
}
```

Використання володіння та позичання в Rust може бути незвичайним для програмістів, які звикли до інших мов програмування, проте цей механізм забезпечує безпеку виконання коду та допомагає уникнути багатьох типів помилок, пов'язаних з керуванням пам'яті.

1.3. Робота з помилками у мові Rust

У Rust, обробка помилок відрізняється від того, як це робиться в більшості інших мов програмування. Rust не має виключень, але використовує механізм обробки помилок за допомогою типу Result.

Тип Result має два варіанти значень: Ok та Err. Ok повідомляє про успішне виконання операції та передає результат виконання, а Err повідомляє про помилку та передає об'єкт Error, що містить інформацію про помилку. Користувач може використовувати механізм match, щоб перевірити, чи є результат Ok чи Err.

```
use std::fs::File;
use std::io::prelude::*;

fn read_file_contents(path: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(path)?;
    let mut contents = String::new();
```

```
    file.read_to_string(&mut contents)?;
    Ok(contents)
}

fn main() {
    let contents = read_file_contents("example.txt");
    match contents {
        Ok(contents) => println!("Contents of file: {}", contents),
        Err(error) => println!("Error reading file: {}", error),
    }
}
```

У прикладі вище, функція read_file_contents відкриває файл за допомогою File::open та зчитує його вміст за допомогою file.read_to_string. Ці методи можуть повернути помилки, тому використовується оператор “?”, який автоматично розпаковує значення типу Result. Якщо будь-який з цих методів поверне помилку, виконання функції припиниться, а значення типу Result буде містити об'єкт Error.

В коді вище, match використовується для перевірки, чи є значення типу Result Ok чи Err. Якщо значення є Ok, ми виводимо його вміст. Якщо значення є Err, ми виводимо повідомлення про помилку.

Оператор “?” є дуже потужним механізмом обробки помилок в Rust, оскільки дозволяє автоматично перевіряти результати функцій, що повертають об'єкти типу Result та Option та забезпечує більш зручну і зрозумілу обробку помилок.

Оператор “?” можна використовувати тільки у функціях, що повертають тип Result або Option. Якщо результатом функції є значення, що не є об'єктом типу Result або Option, оператор “?” не може бути використаний.

Якщо результатом виконання функції є об'єкт типу Result або Option, то оператор “?” перевіряє, чи є результатом успішне виконання функції. Якщо результатом є помилка, оператор “?” автоматично повертає цю помилку з поточної функції, тим самим передаючи її до вищої рівня обробки помилок.

Оператор “?” також дозволяє виконувати послідовні функції, що повертають об'єкти типу Result або Option, за умови, що вони повертають однаковий тип помилок.

```
use std::fs::File;
use std::io::{self, prelude::*, BufReader};

fn read_file(path: &str) -> io::Result<String> {
    let file = File::open(path)?;
    let mut buf_reader = BufReader::new(file);
    let mut contents = String::new();
    buf_reader.read_to_string(&mut contents)?;
    Ok(contents)
}

fn main() -> io::Result<()> {
    let contents = read_file("example.txt")?;
    println!("File contents: {}", contents);
    Ok(())
}
```

У прикладі вище, функція read_file використовує оператор “?” для автоматичної обробки помилок під час виконання функцій File::open та buf_reader.read_to_string. Крім того, використання оператора “?” дозволяє нам більш чисто та зрозуміло написати код. Оператор “?” допомагає виконувати більш

ефективну та читабельну обробку помилок в Rust, а також допомагає зменшити кількість непотрібного коду. Однак, слід бути обережним при використанні оператора “?”, оскільки він автоматично перевіряє результат функції та може приховати деякі важливі помилки. Також слід пам'ятати, що при використанні оператора “?” виконується раннє повернення з поточної функції у разі виникнення помилки, тому поточна функція не буде виконуватись до кінця.

У цілому, оператор `?` є корисним механізмом обробки помилок в Rust, який забезпечує зручну та ефективну обробку помилок в більшості випадків. Однак, слід пам'ятати про його обмеження та ризики, пов'язані з автоматичною обробкою помилок. Для більш складних випадків можна використовувати більш детальну обробку помилок з використанням оператора `match` та інших механізмів.

У Rust є також функція unwrap(), яка дозволяє отримати значення типу Ok з об'єкту типу Result. Однак, якщо значення типу Result є Err, функція unwrap() викличе помилку та завершить виконання програми з повідомленням про помилку та стеком викликів. Тому, використання unwrap() рекомендується лише тоді, коли ми впевнені, що значення типу Result не буде містити помилок.

```
fn divide(x: f64, y: f64) -> Result<f64, &'static str> {
    if y == 0.0 {
        return Err("division by zero");
    }
    Ok(x / y)
}

fn main() {
    let result = divide(10.0, 0.0).unwrap(); // Помилка: division by zero
    println!("Result: {}", result);
}
```

У прикладі вище, функція divide ділить число x на число y та повертає результат в об'єкті типу Result. Якщо значення y дорівнює 0.0, функція повертає Err з повідомленням про помилку. У main() функції ми використовуємо unwrap() для отримання результату виконання функції divide. Оскільки другий аргумент функції є 0.0, що призведе до помилки, виконання програми буде припинено, а на екран буде виведено повідомлення про помилку та стек викликів.

Замість використання unwrap(), можна використовувати функцію expect(), яка дозволяє вказати власне повідомлення про помилку, що дозволяє краще розуміти, що саме призвело до помилки.

```
fn main() {
    let result = divide(10.0, 0.0).expect("Cannot divide by zero");
    println!("Result: {}", result);
}
```

У прикладі вище, замість unwrap() використовується expect() з власним повідомленням про помилку.

Якщо у програмі виникає критична помилка, яка не може бути оброблена, Rust може викликати помилку паніки (panic). Помилка паніки зазвичай означає, що виникла непередбачувана ситуація, яка не може бути оброблена програмно. У такому випадку виконання програми припиняється, а на екран виводиться повідомлення про помилку та стек викликів.

```
fn divide(x: f64, y: f64) -> f64 {
    if y == 0.0 {
        panic!("division by zero");
    }
    x / y
}
```

```
fn main() {
    let result = divide(10.0, 0.0);
    println!("Result: {}", result);
}
```

У прикладі вище, функція divide викликає помилку паніки, якщо значення y дорівнює 0.0. У main() функції ми викликаємо функцію divide, передаючи їй 0.0 в якості другого аргументу. Це призведе до помилки паніки, програма буде припинена, а на екран виведеться повідомлення про помилку та стек викликів.

У Rust є можливість створювати власні типи помилок, які повинні бути описані у вигляді структури або перерахування. Це дозволяє створювати більш детальні повідомлення про помилки та додаткові методи для обробки помилок.

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;

#[derive(Debug)]
enum MyError {
    FileNotFound,
    IOError(std::io::Error),
}

impl Error for MyError {}

impl std::fmt::Display for MyError {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match *self {
            MyError::FileNotFound => write!(f, "file not found"),
```

```
            MyError::IOError(ref err) => write!(f, "I/O error: {}", err),
        }
    }
}

fn read_file_contents(path: &str) -> Result<String, Box<dyn Error>> {
    let mut file = match File::open(path) {
        Ok(file) => file,
        Err(error) => return Err(Box::new(MyError::IOError(error))),
    };
    let mut contents = String::new();
    match file.read_to_string(&mut contents) {
        Ok(_) => Ok(contents),
        Err(error) => Err(Box::new(MyError::IOError(error))),
    }
}

fn main() -> Result<(), Box<dyn Error>> {
    let contents = read_file_contents("example.txt")?;
    println!("Contents of file: {}", contents);
    Ok(())
}
```

У прикладі вище, ми визначаємо власний тип помилок MyError, який містить два варіанти помилок: FileNotFound та IOError. Далі ми реалізуємо трейти Error та Display для нашого типу помилок. В функції read_file_contents, яка зчитує вміст файлу, ми перевіряємо результат виконання функції File::open. Якщо функція повертає помилку, ми повертаємо наш тип помилок MyError::IOError, який містить

об'єкт Error типу std::io::Error. Аналогічно, якщо функція file.read_to_string повертає помилку, ми повертаємо наш тип помилок MyError::IOError.

У main() функції ми викликаємо функцію read_file_contents, якій передаємо назву файлу. Якщо функція повертає помилку, ми передаємо її далі за допомогою оператора “?”. Якщо функція виконується успішно, ми виводимо вміст файлу на екран.

Загалом, в Rust є кілька різних механізмів обробки помилок, які дозволяють ефективно управляти помилками та забезпечувати більш точне та ефективне управління ресурсами. Ці механізми залежать від контексту використання та вимог програми.

1.4. Риси та загальні типи, їх використання

Trait (риса, іноді називається інтерфейсом) – це механізм, що дозволяє описувати загальні властивості та функціональність, які можуть бути реалізовані різними типами в Rust. Trait можна розглядати як договір між різними типами, що гарантує, що вони мають певну загальну поведінку.

У Rust, trait можна використовувати для опису загальної функціональності, яку можуть реалізовувати різні типи. Trait містить опис методів, які мають бути реалізовані в типах, що використовують цей trait. Також можна визначати додаткові методи та асоційовані константи у trait.

Один з прикладів trait в Rust – це Iterator, який описує поведінку об'єктів, що дозволяють ітеруватись через колекції даних. Цей trait містить опис методів, які дозволяють отримати наступний елемент послідовності (next()), перевірити, чи закінчилась послідовність (is_end()), та інші методи. Цей trait може бути реалізований різними типами даних, які мають підтримку ітерації, такими як вектори, списки, масиви та інші.

Щоб використовувати trait в Rust, необхідно спочатку описати його за допомогою ключового слова trait. Опис trait може включати в себе методи, асоційовані типи та константи. Далі, типи даних можуть реалізовувати цей trait, надаючи реалізації методів, що включені в trait.

```
trait Printable {
    fn print(&self);
}
struct Person {
    name: String,
    age: u32,
}

impl Printable for Person {
    fn print(&self) {
        println!("Name: {}, Age: {}", self.name, self.age);
    }
}

fn main() {
    let person = Person { name: String::from("John"), age: 30 };
    person.print();
}
```

У прикладі вище, ми визначаємо trait Printable, який містить один метод print(). Далі, ми визначаємо структуру Person, яка містить поля name та age. Ця структура реалізує trait Printable, надаючи реалізацію метода print(), який виводить на екран інформацію про особу. У функції main() ми створюємо об'єкт типу Person та викликаємо метод print(), що було визначено в trait Printable.

Trait є потужним інструментом в Rust, оскільки він дозволяє створювати загальні та переносимі рішення для роботи з різними типами даних. За допомогою trait можна створювати абстракції над даними та методами, що дозволяють працювати з ними, незалежно від конкретного типу даних. Trait також допомагає забезпечувати безпеку типів, оскільки дозволяє вказувати обмеження на типи даних, які можуть використовувати певні функції або методи.

У Rust є багато вбудованих trait, таких як Clone, Debug, PartialEq та інші, що дозволяють працювати з типами даних у стандартній бібліотеці та стандартному синтаксисі мови. Також можна визначати власні trait та використовувати їх для створення власних абстракцій та бібліотек.

Generic (загальні типи) - це механізм, що дозволяє описувати загальні типи та функції, які можуть працювати з будь-яким типом даних. Подібний механізм з такою ж назвою є в Java, а у C++ схожий механізм називається template. У Rust, generic дозволяє створювати функції та структури даних, які можуть працювати з будь-яким типом даних, що задовольняє певні обмеження.

У Rust, generic визначаються за допомогою параметрів типу. Параметр типу включає ім'я типу, яке може використовуватись у функції або структурі даних, та обмеження на цей тип. Обмеження може включати в себе trait, який повинен бути реалізований типом даних, або інші обмеження, які задаються за допомогою ключового слова where.

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
```

```
        Point { x, y }
    }

    fn get_x(&self) -> &T {
        &self.x
    }

    fn get_y(&self) -> &T {
        &self.y
    }
}

fn main() {
    let point = Point::new(10, 20);
    println!("x: {}, y: {}", point.get_x(), point.get_y());
}
```

У прикладі вище, ми визначаємо структуру Point<T>, яка містить поля x та y типу T. Тип T є параметром типу, який може бути замінений на будь-який тип даних. Ми використовуємо ключове слово impl для визначення методів, які працюють з будь-яким типом T. В функції main() ми створюємо об'єкт типу Point<i32> за допомогою методу new(), який приймає два аргументи типу i32. Ми викликаємо методи get_x() та get_y(), щоб отримати значення полів x та y об'єкта point.

Generic дозволяє писати більш зручний та переносимий код, оскільки дозволяє працювати з будь-яким типом даних, що задовольняє певні обмеження, без необхідності створювати окремі реалізації для кожного типу даних. Крім того,

generic допомагає забезпечувати безпеку типів, оскільки дозволяє вказувати обмеження на типи даних, які можуть використовувати певні функції або методи.

У Rust, generic можна використовувати не тільки для структур даних, але і для функцій та методів. Параметри типу можуть мати довільну кількість, та можуть використовувати будь-яке ім'я типу, що задовольняє певні обмеження.

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut max = list[0];
    for &item in list.iter() {
        if item > max {
            max = item;
        }
    }
    max
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];
    let result = largest(&numbers);

    println!("The largest number is {}", result);
    let chars = vec!['a', 'z', 'k', 'e', 'b'];
    let result = largest(&chars);
    println!("The largest char is {}", result);
}
```

У прикладі вище, ми визначаємо функцію largest, яка приймає посилання на масив типу T, де T є параметром типу з обмеженнями PartialOrd та Copy. Ми використовуємо метод iter() для отримання ітератора по масиву, та порівнюємо

кожен елемент з поточним максимумом, зберігаючи найбільший елемент у змінній max. У функції main() ми створюємо два масиви – numbers та chars, та викликаємо функцію largest() для обох масивів. Функція largest() працює з будь-яким типом даних, що задовольняє обмеженням на тип T.

У Rust, generic дозволяє писати більш переносимий та зручний код, що може працювати з різними типами даних. Generic допомагає забезпечувати безпеку типів та використовується в багатьох вбудованих типах та бібліотеках, які дозволяють працювати з різними типами даних, таких як вектори, хеш-таблиці, файлові операції та інші.

Одним з важливих принципів, пов'язаних з використанням generic в Rust, є необхідність задавати обмеження на типи даних, що використовуються в generic. Це дозволяє забезпечувати безпеку типів та допомагає компілятору Rust перевірити правильність використання generic та попередити можливі помилки в коді.

Загальні типи також дозволяють створювати власні типи даних, які можуть використовувати різні типи даних для зберігання та обробки інформації. Наприклад, власний тип даних може містити в собі декілька полів різних типів даних, які можуть зберігати різну інформацію. Такі типи даних можуть бути корисні при роботі зі структурованими даними, такими як таблиці баз даних, XML-файли та інші формати.

1.5. Час життя – одна з основних складових безпеки Rust

Lifetime (час життя) – це механізм, що використовується в Rust для контролювання часу життя посилань та уникнення небезпечних взаємодій між ними. У Rust, кожен блок коду має свій власний lifetime, який визначає, як довго

можуть існувати посилання на дані в цьому блоку коду. Цей механізм є достатньо унікальним і його важко знайти в схожому вигляді в інших мовах програмування. Наряду з ownership, lifetime є одним з найважливіших механізмів для забезпечення безпечності Rust, але при цьому і одним із найскладніших для розуміння, для людей, що звикли до інших мов програмування.

Lifetime зазвичай використовується в контексті посилань на дані. У Rust, посилання можуть бути двох типів: посилання з власником (owned references) та посилання без власника (borrowed references). Посилання з власником (owned references) володіють власником даних та контролюють їхній час життя. Посилання без власника (borrowed references) можуть бути створені на базі посилань з власником, та їх час життя залежить від часу життя посилання з власником.

```
fn main() {
    let mut s1 = String::from("hello");
    let r1 = &s1; // borrowed reference
    let r2 = &s1; // borrowed reference
    println!("r1: {}, r2: {}", r1, r2);
}
```

У прикладі вище, ми створюємо змінну s1 типу `String`, яка містить рядок "hello". Далі ми створюємо два посилання без власника (r1 та r2), які посилаються на s1. Ці посилання можуть бути створені тільки після створення посилання з власником, і їхній час життя залежить від часу життя посилання з власником s1.

У Rust, lifetime можна визначати як ім'я, яке починається з символу апострофа '. Lifetime може бути вказаний в типі даних, що передаються як аргументи функції, або використовуватись у декларації посилання. Наприклад:

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
```

```
        x
    } else {
        y
    }
}
```

Ця функція longest приймає два посилання на рядки (x та y), і повертає посилання на той рядок, яка має більшу довжину. Ми вказуємо lifetime 'a як параметр типу функції, що означає, що обидва посилання мають бути збережені на протязі життя 'a. Це допомагає Rust визначити, яке посилання може існувати довше, та дозволяє нам забезпечувати безпеку використання посилань.

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";
    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

У прикладі вище, ми створюємо два рядки – string1 та string2. Ми викликаємо функцію longest та передаємо до неї посилання на обидва рядки. Оскільки обидва посилання мають однаковий час життя 'a, функція longest може повернути посилання на один із рядків, яке може бути використане пізніше в коді.

Важливою особливістю lifetime в Rust є те, що вони не додають ніяких додаткових витрат на час виконання програми. Rust використовує систему lifetime для генерування безпечного та ефективного коду, який дозволяє забезпечити безпеку типів та уникнути небезпечних взаємодій між посиланнями на дані.

1.6. Замикання або лямбда-функції в Rust

Closure – це функція, яка може звернутися до змінних, що знаходяться в зовнішньому контексті (lexical closure). В Rust, closure є анонімною функцією, яка може бути передана як аргумент в іншу функцію, або бути повернута з функції.

Синтаксис closure у Rust досить простий. Ось приклад:

```
let plus_one = |x| x + 1;
let result = plus_one(10);
println!("{}", result); // виведе 11
```

У прикладі вище ми створюємо closure, яке додає до свого аргументу число 1. Ми передаємо значення 10 до цього closure та отримуємо результат 11.

Один зі способів використання closure в Rust - це заміна функцій з виразами lambda, які можуть бути передані в якості аргументів до інших функцій. Наприклад, ми можемо використовувати closure для сортування масиву за спаданням:

```
let mut numbers = vec![5, 2, 4, 1, 3];
numbers.sort_by(|a, b| b.cmp(a));
println!("{:?}", numbers); // виведе [5, 4, 3, 2, 1]
```

У прикладі вище, ми створюємо масив чисел та викликаємо метод sort_by, який приймає closure як аргумент. Цей closure визначає порядок сортування – в нашому випадку, ми використовуємо cmp для порівняння двох чисел та повернення значення, яке вказує, яке число має бути розташоване перед іншим. Зверніть увагу, що використання closure дозволяє нам визначати порядок сортування у залежності від поточного контексту.

Ще один спосіб використання closure – це зберігання їх у змінних та передача в інші функції як звичайні аргументи. Це дозволяє нам створювати більш гнучкий та динамічний код. Наприклад, ми можемо використовувати closure для фільтрування елементів в масиві:

```
let numbers = vec![1, 2, 3, 4, 5];
let even_numbers: Vec<_> = numbers.into_iter().filter(|x| x % 2 == 0).collect();
println!("{:?}", even_numbers); // виведе [2, 4]
```

У прикладі вище ми створюємо масив чисел та викликаємо метод into_iter, який повертає ітератор для нашого масиву. Ми потім викликаємо метод filter, який приймає closure як аргумент та повертає новий ітератор, який містить тільки ті елементи, що задовольняють умову в closure. Нарешті, ми викликаємо метод collect, щоб зібрати всі відфільтровані елементи в новий масив even_numbers.

Однією з важливих особливостей closure у Rust є те, що вони можуть захоплювати значення зі зовнішнього контексту та зберігати їх для майбутнього використання. Наприклад:

```
fn main() {
    let x = 4;
    let add_x = |y| x + y;
    let result = add_x(10);
    println!("{}", result); // виведе 14
}
```

У прикладі вище, ми створюємо closure add_x, яке додає до свого аргументу число 4 (значення зі зовнішнього контексту) та повертає результат. Ми передаємо число 10 до цього closure, та отримуємо результат 14. Це дозволяє нам створювати більш гнучкі та динамічні функції, які можуть звертатися до значень зі зовнішнього контексту.

Загалом, closure в Rust дозволяють писати більш гнучкий та динамічний код, що може працювати з різними типами даних та виконувати різноманітні операції. Це може бути особливо корисним в додатках, які працюють з динамічними даними та потребують різноманітних алгоритмів та операцій.

1.7. Розумні вказівники для роботи з динамічним виділенням пам’яті

Smart pointers – це спеціальні типи даних, які забезпечують додатковий контроль над ресурсами та дозволяють зменшити ризики небезпечних операцій, таких як витік пам'яті. В Rust є кілька типів smart pointers, таких як Rc, Box, Ref, RefCell та RefMut.

1. Rc (reference counting) – це smart pointer, який дозволяє створювати декілька власників (owners) для одного та ж ресурсу, причому кількість власників збільшується та зменшується автоматично за допомогою системи рахунку посилань. Загалом це досить розповсюджений принцип роботи розумного вказівника, який є, наприклад у C++, а також використовується як частина механізму garbage collection в інших мовах. Rc корисний, коли потрібно створювати кілька копій об'єкту, але не хочете виконувати повну копіювання даних.

```
use std::rc::Rc;

struct Person {
    name: String,
    age: u8,
}

fn main() {
    let person = Person {
        name: String::from("John"),
        age: 30,
    };
```

```
    let rc_person = Rc::new(person);

    let rc_person1 = Rc::clone(&rc_person);
    let rc_person2 = Rc::clone(&rc_person);
}
```

У прикладі вище ми створюємо об'єкт типу Person. Ми створюємо Rc з цього об'єкту та створюємо два посилання (rc_person1 та rc_person2) на цей об'єкт за допомогою методу clone. Rc дозволяє нам створювати багато посилань на один та самий об'єкт, та автоматично видаляти об'єкт після того, як останнє посилання на нього буде знищене.

2. Box – це smart pointer, який дозволяє створювати об'єкти на стеку та керувати життєвим циклом об'єкту на кучі. Box дозволяє нам передавати об'єкти як аргументи до функцій та повертати об'єкти з функцій.

```
fn main() {
    let number = 5;
    let boxed_number = Box::new(number);
    println!("Boxed number: {}", boxed_number);
}
```

У прикладі вище ми створюємо змінну number та передаємо її до Box::new, щоб створити новий Box з цим числом. Ми можемо передавати цей Box до функцій та повертати його з функцій як звичайний об'єкт.

3. Ref та RefMut – це smart pointers, які дозволяють отримувати доступ до даних за допомогою запозичення. Ref та RefMut використовуються зі структурами, які підтримують систему рахунку посилань, такими як Rc. Ref дозволяє отримувати доступ до даних за запозичення для читання, тоді як RefMut дозволяє отримувати доступ до даних за запозичення для запису.

```
use std::cell::RefCell;

fn main() {
    let x = RefCell::new(42);

    let y = x.borrow();
    assert_eq!(*y, 42);

    let mut z = x.borrow_mut();
    *z = 13;
    assert_eq!(*z, 13);
}
```

У прикладі вище ми створюємо змінну x типу RefCell, яка дозволяє нам отримувати доступ до даних за запозиченням. Ми отримуємо позичення за допомогою методу borrow, який повертає immutable reference до даних. Ми можемо читати дані за допомогою отриманого посилання. Ми також можемо отримати mutable reference до даних за допомогою методу borrow_mut, щоб записати дані. Важливо звернути увагу, що може бути отримано тільки одне mutable посилання на даний об'єкт в один момент часу.

4. RefCell – це smart pointer, який дозволяє отримувати доступ до даних за пзапозиченняозиченням та змінювати їх, не використовуючи систему рахунку посилань. RefCell корисний, коли потрібно змінювати дані в immutable контексті, наприклад, в методі структури, який позначений як &self.

Усі ці smart pointers дозволяють нам підвищити безпеку та зручність коду, зменшуючи ризики витоку пам'яті та небезпечних операцій з пам'яттю. Кожен з цих типів smart pointers має свої власні особливості та використовується в різних

ситуаціях. Розуміння того, коли та який smart pointer слід використовувати, допоможе написати безпечні та ефективні програми в мові Rust.

1.8. Небезпечний Rust або як використовувати низькорівневі операції

Останньою з важливих деталей Rust ми розглянемо ключове слово unsafe, оскільки в контексті цієї роботи воно є дуже важливим і без нього було б неможливо її виконати.

unsafe – це ключове слово мови Rust, яке дозволяє виконувати небезпечні операції, які неможливо виконати за допомогою безпечних засобів мови Rust. Це дозволяє нам виконувати низькорівневі операції, такі як робота з низькорівневими системними інтерфейсами або створення власних реалізацій типів даних. Однак, використання unsafe також збільшує ризики збоїв та знижує безпеку програми.

У безпечній мові Rust існує кілька правил, які забезпечують безпеку виконання коду. Ці правила дозволяють гарантувати, що програма не виконає небезпечних операцій, таких як доступ до неіснуючої пам'яті чи виклик неправильних функцій. Проте іноді, щоб виконати певні операції, потрібно обійти ці правила та використати unsafe.

Основні приклади використання unsafe в Rust:

- Робота з низькорівневими системними інтерфейсами. Наприклад, якщо потрібно взаємодіяти з операційною системою, потрібно використовувати unsafe, оскільки це може призвести до виконання небезпечних операцій.

```
unsafe {
   libc::signal(libc::SIGTERM, libc::SigHandler::SigIgn);
}
```

- Робота з C-функціями та сторонніми бібліотеками. Rust дозволяє використовувати C-функції та сторонні бібліотеки, однак це може бути небезпечно та призвести до виконання неправильних операцій.

```
extern "C" {
    fn c_function(x: i32) -> i32;
}
unsafe {
    let result = c_function(42);
}
```

- Створення власних типів даних. unsafe може використовуватися для створення власних типів даних, які не можуть бути використані за допомогою безпечних засобів мови Rust. Наприклад, можна створити власний тип даних, який прямо взаємодіє з пам'яттю.

```
struct RawVec<T> {
    ptr: *mut T,
    cap: usize,
}
impl<T> RawVec<T> {
    unsafe fn new(cap: usize) -> Self {
        let size = mem::size_of::<T>() * cap;
        let align = mem::align_of::<T>();
        let ptr = malloc(size) as *mut T;
        Self { ptr, cap }
    }
}
impl<T> Drop for RawVec<T> {
    fn drop(&mut self) {
```

```
        unsafe {
            for i in 0..self.cap {
                ptr::drop_in_place(self.ptr.add(i));
            }
            free(self.ptr as *mut u8);
        }
    }
}
```

У прикладі вище ми створюємо власний тип даних RawVec, який забезпечує безпечний доступ до масиву елементів. Оскільки RawVec взаємодіє з пам'яттю напряму, ми використовуємо unsafe для взаємодії з пам'яттю та створення власних типів даних.

Використання unsafe необхідно мінімізувати та використовувати тільки в тих випадках, коли без нього неможливо виконати певну операцію. Код, який використовує unsafe, повинен бути докладно протестований та перевірений на наявність небезпечних операцій. Використання unsafe вимагає більш глибокого розуміння мови Rust та системного програмування, тому його використання має бути обмеженим та обґрунтованим.

РОЗДІЛ 2. НАПИСАННЯ КООПЕРАТИВНОЇ ОПЕРАЦІЙНОЇ СИСТЕМИ

2.1. Запуск функції main на STM32

Перше, що потрібно зробити це створити виконуваний файл, який би не опирався на жодні можливості, які надає ОС. Для цього потрібно відключити стандартну бібліотеку Rust, створити хендлер для panic, відключити stack unwinding у випадку помилки(для нього використовуються системні бібліотеки, яких у нас не буде), а також створити функцію _start(це перша функція, яка викликається при початку виконання програми, а не main).

Крім того, потрібно зробити так, що програма взагалі запустилася, оскільки мікроконтролер не знає, де саме знаходиться наш _start. Для цього потрібно визначити reset handler, який викликається після перезавантаження мікроконтролера і розмістити його у відповідному місці в пам’яті.

Оскільки стандартна бібліотека мови Rust не реалізована для нашої операційної системи, то ми не можемо опиратися на її функції і її потрібно відключити за допомогою директиви #![no_std] на початку файлу. Крім цього, потрібно використовувати директиву #[no_mangle] для того щоб Rust не змінював назви функцій.

2.2. Скрипт компонувальника

Компонувальник (також редактор зв'язків, лінкер – від англ. Link editor, linker) – програма, яка виконує компонування (англ. linking) – приймає на вхід один або кілька об'єктних модулів (та/або бібліотек) і збирає їх в один виконуваний модуль.[2] Крім цього, компонувальник буде відповідальним за те, щоб код

Reset_Handler був там, де його очікує знайти мікроконтролер, а також за значення змінних, імена яких починалися з нижніх підкреслень у попередньому коді.

З документації процесора знаходимо, що очікуваний адрес Reset_Handler це 0x00000004. З документації мікроконтролера [3] отримуємо інформацію про розмір та початковий адрес оперативної та флеш пам’яті.

Далі крок за кроком будемо наводити код скрипта компонувальника з поясненнями. Спочатку вказуємо entry point – тобто функцію, з якої починається виконання програми. Далі вказуємо параметри пам’яті і розмір стеку:

```
ENTRY(Reset_Handler)
MEMORY
{
rom      (rx)  : ORIGIN = 0x08000000, LENGTH = 0x00040000
ram      (rwx) : ORIGIN = 0x20000000, LENGTH = 0x00008000
}
STACK_SIZE = 0x2000;
```

Після цього буде йти список секцій - частину програмного коду, які будуть розміщені послідовно у виконуваному файлі. Фактично, можна не ділити виконуваний файл на секції. Це як з функціями у звичайних програмах - можна написати всю програму в одній функції, але для збільшення читабельності їх краще розбивати. Всі секції описуються всередині блоку SECTIONS.

Першою секцією є .text – загальноприйнята назва секції, що містить константи і код програми. В цю секцію ми хочемо помістити вміст усіх скомпільованих файлів, а також таблицю виключень – масив із вказівниками на хендлери виняткових ситуацій (серед яких головним для нас є Reset_Handler). Ця секція має бути завантажена у флеш пам’ять.

```
.text :
  {
```

```
    KEEP(*(.vectors .vectors.*))
    *(.text.*)
    *(.rodata.*)
    _etext = . ;
  } > rom
```

Далі йде секція .bss - вона зберігає неініціалізовані дані, які потрібно буде занулити перед виконанням програми.

```
.bss (NOLOAD) :
 {
    _sbss = . ;
    *(.bss)
    *(.bss*)
    *(COMMON)
    _ebss = . ;
  } > ram
```

Секція .data зберігає константи, а тому має зберігатися у флеш пам’яті, а потім завантажуватись в оперативну пам’ять.

```
.data :
  {
    _sdata = .;
    *(.data*);
    _edata = .;
  } > ram AT >rom
```

Також потрібно виділити пам’ять на стек і зарезервувати її:

```
.stack (NOLOAD):
  {
    . = ALIGN(8);
```

```
    _sstack = .;
    . = . + STACK_SIZE;
    . = ALIGN(8);
    _estack = .;
  } > ram
```

Після компілювання цього коду і компонування з даним скриптом ми нарешті можемо завантажити його на мікроконтролер і потрапити у вічний цикл – для того щоб упевнитися в цьому використовуємо дебагер.

2.3. Запуск програми “мигаючий LED” на STM32

IoT варіантом hello world є програма blinking LED – мигаючий діод на мікроконтролері. В нашому випадку будемо використовувати вбудований світлодіод для цього.

З тієї ж документації знаходимо, що за вбудований LED відповідає GPIO пін PA5. Для керування GPIO в даному мікроконтролері використовуються memory mapped регістри (вони не є регістрами у прямо розумінні цього слова, але є адресами у фізичній пам’яті записом в які можна керувати різними функціями мікроконтролера). Для початку потрібно буде увімкнути GPIO периферію за допомогою RCC регістру (за замовчуванням більшість периферії вимкнена). Далі за допомогою GPIO регістрів встановлюємо режим роботи піна PA5 у вихід, відключаємо pull up і pull down. Далі можемо керувати включенням і виключенням діоду за допомогою регістру BSRR (bit set reset register) – що є складовою GPIO регістрів; достатньо просто поставити відповідний біт цього регістру в значення 0 або 1 (оскільки ми користуємось піном PA5, то треба міняти значення 5-го біта в цьому регістрі).

2.4 Налаштування UART

UART (universal asynchronous receiver-transmitter) – це розповсюджений спосіб асинхронної комунікації. За допомогою перехідника з UART на USB можна буде відправляти та отримувати повідомлення з мікроконтролера. Це буде значно зручніше ніж перевіряти усе через дебагер. Але і налаштування буде складніше, ніж у випадку з GPIO.

Аналогічно з GPIO потрібно включити необхідну периферію за допомогою RCC регістру, але це ще не все – UART передає сигнали фреймами, чергуючи високе та низьке значення струму. Для цього йому потрібен якийсь осциллятор, який би задавав “ритм”. На кожен тік осциллятору UART буде передавати логічний 0 або 1. Кожен фрейм складається з:

1. Стану спокою (логічне 1).
2. Початковий біт (логічне 0).
3. Дані (зазвичай по 8 біт в одному фреймі).
4. Біт парності (не обов’язково, залежить від налаштувань).
5. Стоп (логічне 1).

На рисунку 2.1 схематично показано один фрейм.

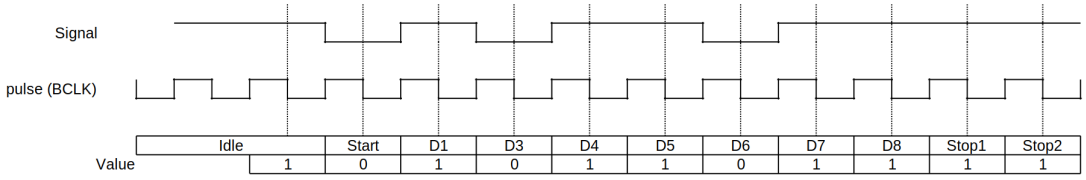

Рисунок 2.1 – Фрейм UART

З вище написаного випливає що крім власне налаштування UART нам потрібно ще налаштувати осцилятор. Для цього через той же RCC вмикаємо HSI(high speed internal) осциллятор, налаштовуємо його частоту та встановлюємо його як осциллятор для UART. Далі потрібно лише вибрати правильні альтернативні функції для GPIO пінів – один UART_TX (відправник) і один

UART_RX (отримувач) та налаштувати параметри фреймів(кількість біт даних, стоп біт та біт паритету). Після цього вже можна користуватись UART.

В плані використання все доволі просто – дані записуються в TDR(transmit data register) і читаються з RDR(receive data register). Щоб зрозуміти чи встиг отримувач прочитати попередній фрейм і чи готовий він до наступного потрібно чекати поки не обнулиться біт в ISR (interrupt status register) регістрі що відповідає за запис, аналогічно з читанням.

2.5. Менеджер пам’яті

Дуже важливим елементом операційної системи є менеджер пам’яті – у випадку складної системи майже неможливо буде розподіляти пам’ять просто запам’ятовуючи який процес в який діапазон фізичної пам’яті повинен писати. Для цього і потрібен менеджер пам’яті – видавати процесу пам’ять з якою він може працювати не заважаючи іншим процесам.

Для даної операційної системи недоступна віртуальна пам’ять на рівні заліза, тому вся реалізація менеджера пам’яті, по суті, зводиться до реалізації стандартного) API для роботи з пам’яттю – функцій alloc(виділення пам’яті) і dealloc(звільнення пам’ятті).

Отже, наразі вся доступна для операційної системи пам’ять це діапазон 0x20000000 - 0x20008000. Для її розподілу виберем досить простий варіант – будемо ділити цю пам’ять на блоки, які будуть утворювати зв’язний список. Для цього використаємо таку структуру даних:

```
struct BlockHeader {
    size: usize,
    next: *mut BlockHeader,
    prev: *mut BlockHeader,
    free: bool,
```

```
    data: [u8; 0],
}
```

Таким чином ми має зв’язний список хедерів, після яких будуть іти власне дані. Ці хедери будуть використовуватися для виділення та звільнення пам’яті менеджером пам’яті, а інші функції матимуть доступ лише до даних.

Коротко пройдемося по полям цієї структури: size це просто розмір блоку, далі йдуть вказівники на наступний і попередній блоки, потім змінна, яка буде вказувати на те чи вільний зараз цей блок; і нарешті data - на власне вміст цього блоку пам’яті.

Таким чином робота з пам’яттю виглядає наступним чином – починаємо з одного блока, який містить всю доступну пам’ять. При першому виклику alloc цей блок ділиться на два: у перший матиме розмір, який потрібно повернути з alloc, його змінна free буде 0, а решта пам’яті буде в другому блоці, для якого free = 1. Аналогічним чином продовжуватимемо далі поки є пам’ять. Якщо буде викликано dealloc - просто змінюємо значення free відповідного блоку.

Із запропонованим підходом є одна проблема – фрагментація пам’яті. Уявимо, що в нас є 100 байт пам’яті, а хедери її взагалі не займають(для простоти ілюстрації проблеми). Розглянемо наступну послідовність викликів:

```
a = alloc(20);
b = alloc(20);
c = alloc(20);
d = alloc(20);
f = alloc(20);

dealloc(b);
dealloc(c);
x = alloc(40);
```

На останньому рядку ми отримаємо помилку, бо в нас немає вільного блоку, розмір якого був би 40 або більше, при цьому є 2 блоки по 20 байтів, які розміщені поряд – варто було б їх трактувати як 1.

Зазначимо також, що це не вирішує проблеми, якби було звільнено не блоки d і c, а блоки b і d - рішення цієї проблеми виходить за рамки цієї роботи.

2.6 Кооперативна багатозадачність

Тепер час перейти до основної функції нашої мінімалістичної операційної системи – створення і менеджмент задач, оскільки саме для цього зазвичай і використовують операційні системи в IoT проектах. В рамках цієї роботи ми будемо створювати кооперативну операційну систему – тобто задачі повинні будуть добровільно передавати право на виконання іншим задачам.

Є два основних підходи до реалізації багатозадачності в операційній системі: кооперативна багатозадачність (cooperative multitasking) і перериваюча багатозадачність (preemptive multitasking). Основна різниця між ними полягає у способі розподілу часу і керуванні процесами.

Кооперативна багатозадачність:

- кожен процес або задача повинні свідомо передавати управління іншим процесам;
- процеси виконуються почергово відразу після завершення попереднього процесу або після виклику певної системної функції для передачі управління іншому процесу;
- виконання процесів у кооперативній багатозадачності залежить від коректності і чесності програмного коду. Якщо один процес зациклюється або блокується, то це може призвести до затримок у роботі інших процесів;

- цей підхід зазвичай вимагає меншої кількості системних ресурсів і має меншу накладну вартість, але може бути менш стійким до помилок та зламів програмного коду.

Перериваюча багатозадачність:

- операційна система призначає пріоритети процесам та здійснює переривання для передачі управління від одного процесу до іншого;
- операційна система має розпорядження переривати виконання процесу після певного інтервалу часу або за певних умов (наприклад, надходження події або вимоги іншого процесу);
- цей підхід дозволяє краще керувати ресурсами та пріоритетами, а також підвищує стійкість системи до помилок і зламів програмного коду;
- кожен процес має встановлений пріоритет, і операційна система розподіляє часові і ресурсні кванти процесора відповідно до цих пріоритетів;
- якщо один процес блокується або займає багато часу, операційна система може призначити управління іншому процесу з вищим пріоритетом;
- цей підхід може бути вимогливішим щодо ресурсів, оскільки операційна система потребує додаткових механізмів для керування перериваннями та розподілу часу.

Як видно, кожен підхід має свої плюси і мінуси, але в даній роботі було вирішено реалізовувати кооперативну багатозадачність, так як вона менш вимоглива до ресурсів, а також значно простіша в реалізації. Крім того, такий тип багатозадачності краще підходить під реалізацію мінімалістичної операційної системи, оскільки при подальшій розробці на цій операційній системі і створенні

задач, що на ній виконуватимуться, не потрібно буде перейматися щодо переривань.

Для того щоб це реалізувати нам потрібно мати процедуру зміни контексту (context switch). Ця процедура буде відповідати за те, щоб зберегти значення регістрів та вказівники на стек при переході між різними задачами. Крім того, воно буде виконувати ще деякі дії, про які мова буде йти нижче.

Почнемо з стеків – кожній задачі потрібен буде власний стек для роботи з локальними змінами. На щастя, STM32 підтримує можливість зберігати вказівники відразу на два стеки - він має регістри sp (stack pointer) і psp (process stack pointer). Це дає можливість в одному з них зберігати вказівник на стек менеджера задач, а в іншому - на стек конкретної задачі. Для того, щоб вибрати який із вказівників на стек буде використовуватися потрібно просто змінити значення другого з кінця біту в control регістрі. З приводу менеджера – будемо використовувати простий та досить ефективний підхід round robin – всі задачі будуть запускатися по черзі (перша задача додана в чергу піде на виконання першою).

Ще одна річ, яку треба зробити під час зміни контексту - запам’ятати поточні значення деяких регістрів, а потім записати в регістри ті значення, які вони мали на момент виходу із задачі. З документації знаходимо, що зберігати треба регістри r0-r7, а крім того нам потрібно буде зберегти регістр lr (він зберігає адрес функції, в яку потрібно повернутись після закінчення поточної) регістр. Але коли ми вже перейшли до задачі їй потрібні дещо інші регістри - lr менеджера задач їй не потрібен бо задачі після свого завершення повинні звільняти ресурси. Натомість при переході в задачу треба зчитати pc регістр(він зберігає адрес команди, яку процесор буде виконувати) – це забезпечить можливість перервати задачу і повернутись саме на те місце, з якого було виконано переривання. В результаті код context switch буде виглядати наступним чином:

```
.global activate
```

```
activate:
    ldr r1, [r0]
    msr psp, r1
    push {r0-r7, lr}

    ldr r1, =task_delete
    mov lr, r1

    /* Use process stack pointer */
    mrs r1, control
    add r1, #2
    msr control, r1

    pop {r0-r7, pc}
```

Крім цього потрібна ще функція yield – щоб задача могла повернути керування менеджеру задач. Вона написана по аналогії:

```
.global yield
yield:
    push {r0-r7, lr}

    /* Use main stack pointer */
    mrs r0, control
    sub r0, #2
    msr control, r0

    pop {r0}
    mrs r1, psp
    str r1, [r0]
```

```
pop {r1-r7, pc}
```

Останнє що потрібно зробити – реалізувати task_delete, яку використовує activate. В ній потрібно просто звільнити пам’ять задачі й перейти до менеджера задач.

РОЗДІЛ 3. ПОРІВНЯННЯ МОВ RUST ТА C

3.1. Загальне порівняння мов Rust та C

Мови програмування Rust та C часто використовуються для схожих задач(хоч сфера використання першої трохи ширша), в основному для системного програмування. Оскільки Rust є новішою мовою, то вона змогла врахувати помилки попередників та взяти корисні ідеї з інших мов. Пройдемося по основним пунктам, в яких ці мови відрізняються:

1. Безпека та контроль помилок: Rust розроблено з урахуванням питань безпеки та контролю помилок. Rust використовує систему власності для управління пам'яттю та забезпечення безпеки програми. Це дозволяє попередити помилки, такі як виток пам'яті, переповнення буферу та неправильні вказівники, що можуть бути небезпечними в embedded розробці. У C програміст повинен вручну керувати пам'яттю та буферами, що може бути складним та приводити до помилок, особливо коли такі помилки виникають не при кожному запуску програми.

2. Багатопоточність та паралелізм: Rust має вбудовану підтримку багатопоточності та паралелізму, що дозволяє розробникам створювати безпечні та ефективні багатопотокові програми. У C для роботи з багатопоточністю потрібно використовувати сторонні бібліотеки або API, що може бути складним та небезпечним.

3. Стиль програмування та розробка: Rust пропонує сучасні та продумані підходи до програмування, такі як власність та типи-структури даних. Rust має вбудовану підтримку пакетного менеджера та системи збірки, що дозволяє легко

управляти залежностями та збиранням проектів. У C потрібно вручну керувати залежностями та збиранням проекту.

4. Швидкість виконання: Rust та C є мовами з високою швидкістю виконання. Rust має вбудовану підтримку низькорівневих операцій та оптимізацій, що дозволяє отримати дуже швидкі програми. Загалом, у дуже багатьох ситуаціях зкомпільований код на Rust і C мало відрізняється, що говорить про високу ефективність компілятора Rust.

5. Спільнота та екосистема: Rust має швидко зростаючу та активну спільноту, що дозволяє розробникам отримувати швидку та професійну підтримку. Rust також має багату екосистему, що включає в себе бібліотеки, фреймворки та інструменти розробки, при цьому мова вже не перший рік займає перше місце в рейтингу StackOverflow і стрімко розвивається. У C дуже велика спільнота та екосистема, яка значно більше за спільноту та екосистему Rust, проте вже не дуже швидко розвивається.

В контексті розробки операційної системи Rust був значно зручнішим завдяки системі збірки cargo, завдяки які не потрібно самостійно збирати всі залежності як в C. Крім того структури Rust та можливість визначати функції для них допомагають набагато краще ділити код на модулі, а завдяки великій кількості перевірок під час компіляції частину вад вдається знайти ще до прошивки мікроконтролера. При цьому, Rust дозволяє виконувати всі ті ж низькорівневі операції за допомогою ключового слова unsafe.

ВИСНОВКИ

Дослідження реалізації операційних систем на мікроконтролерах є актуальною проблемою в сучасному світі Інтернету речей та вбудованих систем. У даній дипломній роботі було розроблено кооперативну операційну систему для мікроконтролера STM32 на мові Rust.

В ході даної роботи було детально розглянуто процедуру розробки кооперативної операційної системи для мікроконтролера STM32 від першого запуску програмного коду на мікроконтролері до створення менеджерів пам’яті та задач. Було детально описано процес налаштування GPIO та UART використовуючи memory mapped регістри, а також описано процедуру зміни контексту та написано асемблерний код її реалізації.

В результаті роботи було розроблено мінімалістичну кооперацііну операційну систему для мікроконтролера STM32F091RCT6. В проектах на її основі можна буде легко організувати паралельне виконання декількох задач, обмін повідомленнями через UART та контроль периферії за допомогою GPIO. Ця операційна система є кращою за інші доступні аналоги, тому, що надає достатньо функціоналу для реалізації проектів, для яких достатньо простої кооперативної операційної системи, при цьому займаючи досить мало місця в пам’яті мікроконтролера, на відміну від більш складних операційних систем як, наприклад, FreeRTOS. Крім того, більшість альтернатив реалізовують (або намагаються реалізувати) операційну систему з перериваннями, а не кооперативну.

Розроблена операційна система вже була використана для проекту – через GPIO до STM32 було підключено датчик вологи і температури, дані з якого потім виводились на LCD дисплей. Власне, для подібних задач дана операційна система підходить чи не найкраще – займає мінімум пам’яті, надає необхідний функціонал

для створення задач (наприклад окремі задачі для зчитування датчиків та запису отриманих значень).

Крім того, дана робота є дуже цінною в плані знань, отриманих під час її виконання: відомості про компонувальник, деталі роботи з процесором та регістрами, знання ассемблеру, принципи роботи менеджерів пам’яті та задач.

Результати дослідження показали, що мова Rust є потенційно ефективним інструментом для розробки операційних систем на мікроконтролерах, зокрема, завдяки своїм унікальним властивостям, що дозволяють забезпечити високу стійкість до збоїв та безпеку пам'яті.

Весь вихідний код можна знайти за посиланням https://github.com/AntonMoroziuk/Rust-OS.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Налагоджувач [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Налагоджувач.
2. Технічна документація для ARMv6-M [Електронний ресурс] – Режим доступу до ресурсу: https://developer.arm.com/documentation/ddi0419/latest.
3. Компонувальник [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Компонувальник.
4. Технічна документація на STM32F091RCT6 [Електронний ресурс] – Режим доступу до ресурсу: https://www.st.com/en/microcontrollers-microprocessors/stm32f103c8.html .
5. The Rust language [Електронний ресурс] – Режим доступу до ресурсу: https://doc.rust-lang.org/book/title-page.html.
6. Writing OS in Rust [Електронний ресурс] – Режим доступу до ресурсу: https://os.phil-opp.com/freestanding-rust-binary/.
7. From zero to main [Електронний ресурс] – Режим доступу до ресурсу: https://interrupt.memfault.com/blog/zero-to-main-rust-1.
8. Крос-компілятор [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Крос-компілятор.
9. Огляд наявних операційних систем реального часу на мові Rust [Електронний ресурс] – Режим доступу до ресурсу: https://arewertosyet.com.
10. Rust on an STM32 microcontroller [Електронний ресурс] – Режим доступу до ресурсу:

https://medium.com/digitalfrontiers/rust-on-a-stm32-microcontroller-90fac16f6342.

11. Rust on STM32 [Електронний ресурс] – Режим доступу до ресурсу: https://jonathanklimt.de/electronics/programming/embedded-rust/rust-on-stm32-2/.
12. Linux kernel development process [Електронний ресурс] – Режим доступу до ресурсу: https://docs.kernel.org/process/development-process.html.
13. Cooperative and preemptive scheduling [Електронний ресурс] – Режим доступу до ресурсу: https://www.rapitasystems.com/blog/what-are-co-operative-and-pre-emptive-scheduling-algorithms.
14. Master Memory Management: Create Your Own malloc Library from Scratch [Електронний ресурс] – Режим доступу до ресурсу: Cooperative and preemptive scheduling [Електронний ресурс] – Режим доступу до ресурсу: https://www.rapitasystems.com/blog/what-are-co-operative-and-pre-emptive-scheduling-algorithms..
15. Alloc source code [Електронний ресурс] – Режим доступу до ресурсу: https://crates.io/crates/alloc-stdlib.