Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Автоматизація тестування вбудованих систем

Спеціальність !!**—**”!!”

Науковий керівник:

канд. техн. наук, доцент

Мосійчук Віталій Сергійович

Київ**—**2019

Анотація

Дипломна робота на тему «Автоматизація тестування вбудованих систем» виконано на 55 сторінках, що включать 5 ілюстрацій, 1 таблиць, 0 додатки та 9 бібліографічних посилань.

З стрімким розвитком технологій та виробництва різної електроніки з використанням керуючого мікроконтролера, з’являється необхідність в швидкому та автоматизованому тестуванні пристроїв. Це дозволить значно зменшити час виходу готового продукту та збільшує надійність цих пристроїв.

Метою дипломного проекту є дослідження доцільності та ефективності використання автоматизованої системи для тестування програмного забезпечення використовуємого в мікроконтролерах.

Для вирішення поставленої мети були розглянуті існуючі платні рішення, а також безкоштовні відкриті проекти.

Ключові слова: автоматизація тестування, тестування вбудованих систем.

ANNOTATION

The degree project consists of an explanatory note on 55 pages, contains 5 illustrations, 1 tables, 0 link, 9 appendices.

With the rapid development of technology and the production of various electronics using a controlling microcontroller, there is a need for rapid and automated testing of devices. This will significantly reduce the time it takes to exit the finished product and increase the reliability of these devices.

The diploma project purpose is studying the feasibility and effectiveness of using an automated system for testing software used in microcontrollers.

For solving this goal, existing paid solutions were considered as well as free open projects.

Keywords: embedded systems testing automation, testing automation.

**ПОЯСНЮВАЛЬНА ЗАПИСКА**

**до дипломної роботи**

на тему: Автоматизація тестування вбудованих систем.

Київ **—** 2019

Зміст

[Перелік скорочень 7](#_Toc11323953)

[ВСТУП 8](#_Toc11323954)

[Розділ 1. Огляд рішень тестування вбудованих систем 11](#_Toc11323955)

[1.1 Підходи щодо тестування 11](#_Toc11323956)

[1.2 Емулятори 11](#_Toc11323957)

[1.2.1 Jumper virtual lab 12](#_Toc11323958)

[1.2.2 Matlab & Simulink 12](#_Toc11323959)

[1.2.3 Zephyr 13](#_Toc11323960)

[1.2.4 Proteus 13](#_Toc11323961)

[1.2.5 Вибір платформи для автоматичного тестування 14](#_Toc11323962)

[1.3 Більш детально про QEMU-Eclipse 16](#_Toc11323963)

[1.3.1 Чому QEMU? 16](#_Toc11323964)

[1.3.2 Чому GNU MCU Eclipse QEMU? 16](#_Toc11323965)

[1.3.3 Переваги GNU MCU Eclipse QEMU 16](#_Toc11323966)

[1.3.4 Підтримувані плати і мікроконтролери 17](#_Toc11323967)

[1.3.5 Сумісність та периферійні пристрої 19](#_Toc11323968)

[1.3.1 SysTick 19](#_Toc11323969)

[1.3.2 Обмеження 19](#_Toc11323970)

[1.3.3 Документація 19](#_Toc11323971)

[1.3.4 Майбутній розвиток 20](#_Toc11323972)

[1.3.5 Оновлення 20](#_Toc11323973)

[Розділ 2. Особливості налаштування платформи емуляції 21](#_Toc11323974)

[2.1 Компіляція та збірка. Встановлення 21](#_Toc11323975)

[2.2 Запуск емуляції та огляд налаштувань 23](#_Toc11323976)

[2.3 Додавання нового мікроконтролера до QEMU 25](#_Toc11323977)

[2.4 Додавання нової плати 36](#_Toc11323978)

[2.5 Керування процесом емуляції через QMP 41](#_Toc11323979)

[Розділ 3. Навчальні приклади реалізації автоматизованого тестування 43](#_Toc11323980)

[3.1 Постановка задачі для демонстрації 43](#_Toc11323981)

[3.2 Емуляція інтерфейсів вводу/виводу даних 44](#_Toc11323982)

[3.3 Скрипти для автоматизації 44](#_Toc11323983)

[Розділ 4. Охорона праці 47](#_Toc11323984)

[4.1 Визначення основних потенційно шкідливих та небезпечних виробничих факторів 47](#_Toc11323985)

[4.2 Технічне рішення та організаційні заходи з безпеки та гігієни праці 48](#_Toc11323986)

[4.2.1 Охорона праці та виробничої санітаріїї при експлуатації ПЕОМ 48](#_Toc11323987)

[4.3 Вплив ЕМВ 49](#_Toc11323988)

[4.3.1 Виробничий шум 50](#_Toc11323989)

[4.3.2 Електробезпека 51](#_Toc11323990)

[4.3.3 Розрахунок захисного відключення електромережі при аварійному режимі роботи електрообладнання 52](#_Toc11323991)

[Висновки 54](#_Toc11323992)

[Перелік посилань 55](#_Toc11323993)

# Перелік скорочень

ПЗ – програмне забезпечення.

ПК – персональний комп’ютер.

WSL – windows subsystem Linux

FPU – Floating Point Unit

TCP – Transmission Control Protocol (Протокол керування передачею)

# ВСТУП

Вбудовані системи подібні повітрю - вони всюди навколо нас, хоча ми їх не помічаємо. І хоча вони складають основу програмного і апаратного забезпечення, більшість користувачів поняття не має, наскільки вони важливі. Вбудовані системи згадують лише тоді, коли вони перестають правильно функціонувати. Уявіть, що ваш улюблений фітнес-браслет втратив зв'язок зі смартфоном. Паніка! А що якщо перестає працювати монітор пацієнта, або інфузійна система, або система діалізу в лікарні? Упевнений, ви не хотіли б опинитися в літаку зі зламаною панеллю приладів і вже точно не сіли б в несправний безпілотний автомобіль.

Сьогодні багато виробників, натхненні ідеєю «автоматизації всього», відмовляються від механіки на користь електронних пристроїв. В результаті складність рішень значно зростає, а значить, потрібно набагато більше тестів перед випуском кінцевих продуктів на ринок.

Говорячи про тестування вбудованого ПЗ, не можна не згадати, що тестування - це всього лише інструмент більш широкого процесу верифікації та валідації. Якщо ми хочемо поставити на ринок надійне і безпечне рішення, ми повинні взяти до уваги не тільки самі дефекти, але і повний спектр вимог до продукту. А їх може бути безліч - від вимог до «заліза», енергоспоживанню і часу відгуку до стабільності і надійності програмних компонентів. У підсумку нам потрібно охопити всі ці вимоги, і ви здивуєтеся тому, скільки незначних дефектів ми часом тестуємо і виправляємо (або не виправляємо, якщо власник продукту вирішує не витрачати на це час і сили).

Незважаючи на те, що технології розвиваються дуже швидко, автоматизація тестування вбудованого ПЗ все ще відстає. Звичайно, на те є численні об'єктивні причини. Розробити набір інструментів і тестів для програмного забезпечення, якого немає навіть в альфа-версії. Протестувати «залізо», що знаходиться в інженерному центрі замовника на іншому кінці світу. Виявити і виправити дефекти операційних систем, їх драйверів і самого ядра, підтримуючи при цьому всю екосистему з численними версіями, оновленнями, новими функціями і периферійними пристроями. Все це вже працює з середини 90-х років.

І кожен раз ми шукаємо можливість автоматизувати процес. Ручне тестування вимагає часу і ресурсів, а в деяких випадках воно практично неможливо через швидкість роботи вбудованих компонентів і пристроїв. Місячні ставки ручних тестувальників начебто не ростуть, але такі проекти як і раніше обходяться дорого. Автоматизувати тестування досить затратне і складне, особливо з огляду на брак на ринку засобів автоматизації тестування вбудованого і системного ПЗ. Однак це не означає, що нічого не можна зробити. Досвідченому професійному провайдеру, який знає специфіку бізнесу і галузі, не важко буде розробити кілька наборів тестів, що відповідають вашим вимогам і унікальним особливостям продукту. Гарним прикладом є команда що створила цілого робота для тестування складного медичного обладнання відділення реанімації та інтенсивної терапії. Робот фізично імітував поведінку користувача пристрою, тому ніяких додаткових симуляцій або модифікацій не треба було, а тривалість тестового прогону скоротилася з трьох тижнів до декількох годин.

Не варто забувати, що ми увійшли в так звану еру інтернету речей, штучного інтелекту, віртуальної реальності, машинного навчання - все ви чули ці модні слова. Я розглядаю штучний інтелект як новий рубіж в тестуванні вбудованого ПЗ, але не в плані написання або виконання тестів. Ні, я не настільки оптимістичний. І все ж штучний інтелект представляється мені потенційно дуже потужним інструментом, що діє як всеосяжна експертна система. Використовуючи вхідні дані про вбудованій системі і апаратних компонентах, він міг би попередити тестувальників про наявність «вузьких місць» і невідповідностей і навіть запропонувати найбільш підходящий набір тестів для забезпечення повного покриття вимог до продукту.

А поки ми чекаємо появи такої «розумної» системи, я спробую описати існуючі способи для автоматизації тестування вбудованого ПЗ.

# Огляд рішень тестування вбудованих систем

На сьогодні вже існують готові рішення для емуляцію вбудованих систем. Деякі з них вже використовуються на великих підприємствах для масового тестування. Ці системи мають широкий функціонал що дозволяє значно розширити межі тестування пристроїв. Деякі з них знаходяться на стадії розробки що дозволяє тестувати лише деякі периферійні модулі тих чи інших пристроїв.

## Підходи щодо тестування

На сьогодні всі виробники будь яких рішень, перед запуском виробу у виробництво мають цей виріб перевірити на стійкість та коректність роботи при тих чи інших умовах.

У випадку вбудованих систем це найбільше актуально так як вся логіка роботі пристрою створюється людиною, і деякі не прозорі сценарії можуть бути не передбаченні. Тому виробники самих мікроконтролерів додали вбудований у МК відладчик. Він дозволяє подивитися роботу програми поетапно та більш детально перевірити окремі периферійні блоки МК. Але такий спосіб дуже часу затратний.

Також довгий час всі вироби тестувалися вручну, такий спосіб також займає велику кількість часу, та потребує мати в наявності готовій прототип для тестування що іноді неможливо у випадку коли над проектом працює декілька людей у різних країнах.

Тому деякі виробники почали створювати програми які будуть це робити але без допомоги людини у автоматичному режимі. Що дозволяє бути впевненим у стресо стійкості виробу та зекономити час.

## Емулятори

Нижче показані декілька найпопулярніших інструментів для емуляції та тестування вбудованих систем.

### Jumper virtual lab



Рисунок 1.2.1 — Логотип програми Jumper virtual lab

Дозволяє запустити прошивку у віртуальній лабораторії за лічені хвилини, навіть не маючи спеціальної програми, а одразу на сайті.

Має досить не поганий опис та декілька прикладів для початку. Є безкоштовною, але на даний момент має не великий функціонал, тому не підходить для реалізації великих проектів.

Також це закритий проект тому його не можна вдосконалювати та оновлювати у майбутньому.

### Matlab & Simulink



Рисунок 1.2.2 — Логотип програми Matlab & Simulink

Має дуже великий функціонал, і вже використовується великими компаніями для автоматизації тестування виробів.

Це повністю закритий проект для отримання можливості використовувати його необхідно придбати ліцензію. Що не для всіх компаній це допустимо.

### Zephyr



Рисунок 1.2.3 — Логотип програми Zephyr

Має досить не поганий функціонал але дуже високий поріг входження, через недостатню кількість документації. Як і попередники цей проект не має відкритий сирцевий код

### Proteus

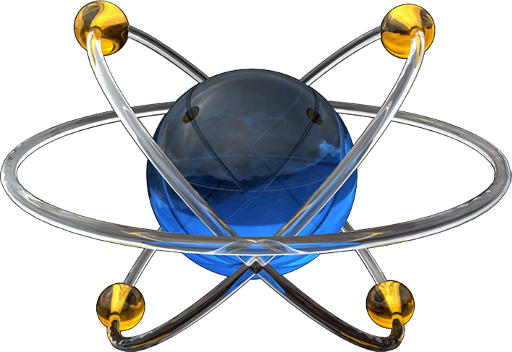


Рисунок 1.2.4 — Логотип програми Proteus

Пакет являє собою систему схемотехнічного моделювання, що базується на основі моделей електронних компонентів, прийнятих в [PSpice](https://uk.wikipedia.org/wiki/PSpice). Відмінною рисою пакету Proteus Design є можливість моделювання роботи програмованих пристроїв: [мікроконтролерів](https://uk.wikipedia.org/wiki/%D0%9C%D1%96%D0%BA%D1%80%D0%BE%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%B5%D1%80), [мікропроцесорних систем](https://uk.wikipedia.org/wiki/%D0%9C%D1%96%D0%BA%D1%80%D0%BE%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D0%BE%D1%80%D0%BD%D0%B0_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), DSP і ін. Proteus Design включає в себе більше 6000 електронних компонентів з усіма довідковими даними, а також демонстраційні ознайомчі проекти.

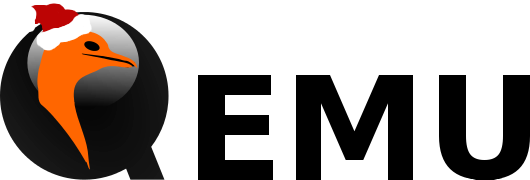


Рисунок 1.2.5 — Логотип програми QEMU

**QEMU** — [вільна програма](https://uk.wikipedia.org/wiki/%D0%92%D1%96%D0%BB%D1%8C%D0%BD%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) з відкритим [сирцевим кодом](https://uk.wikipedia.org/wiki/%D0%A1%D0%B8%D1%80%D1%86%D0%B5%D0%B2%D0%B8%D0%B9_%D0%BA%D0%BE%D0%B4) для [емуляції](https://uk.wikipedia.org/wiki/%D0%95%D0%BC%D1%83%D0%BB%D1%8F%D1%86%D1%96%D1%8F) [апаратного забезпечення](https://uk.wikipedia.org/wiki/%D0%90%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%B5_%D0%B7%D0%B0%D0%B1%D0%B5%D0%B7%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%BD%D1%8F) різних платформ. QEMU дозволяє запустити програму, зібрану для однієї апаратної платформи, на системі із зовсім іншою архітектурою, наприклад, виконати [застосунок](https://uk.wikipedia.org/wiki/%D0%97%D0%B0%D1%81%D1%82%D0%BE%D1%81%D1%83%D0%BD%D0%BE%D0%BA) для [ARM](https://uk.wikipedia.org/wiki/ARM) на [x86](https://uk.wikipedia.org/wiki/X86)-сумісному [ПК](https://uk.wikipedia.org/wiki/%D0%9F%D0%9A). У режимі [віртуалізації](https://uk.wikipedia.org/wiki/%D0%92%D1%96%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%96%D0%B7%D0%B0%D1%86%D1%96%D1%8F) в QEMU досягається продуктивність виконання коду в ізольованому оточенні близька до нативної системи.

### Вибір платформи для автоматичного тестування

Нижче приведена таблиця з приведеними перевагами та недоліками для використання в автоматизації тестування вбудованих систем на підприємствах:

Таблиця 1 – Рівні звукового тиску від різних джерел.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | З відкритим сирцевим кодом | Має широкий функціонал | Є можливість оновлювати та вносити новий функціонал | Є одразу готовий продукт |
| Jumper virtual lab | X | X | X | ✓ |
| Matlab & Simulink | X | ✓ | X | ✓ |
| Zephyr | X | ✓ | X | ✓ |
| Proteus | X | ✓ | X | ✓ |
| QEMU Eclipse | ✓ | ✓ | ✓ | X |

Для багатьох компаній є досить важливо мати можливість дане програмне рішення підстроювати під свої конкретні вимоги. Тому підходящим рішенням є QEMU Eclipse що має повністю відкритий сирцевий код та підтримується та оновлюється розробниками до сьогодні. А також вона має дуже високий функціонал та велику кількість документації.

Багато розробників вбудованих систем використовують цю платформу для емуляції та тестування. Вона має можливість емулювати хоч і багато платформ та не у повній мірі, що означає що розробнику що випускає свій продукт буде необхідно створити відсутній функціонал для подальшого тестування.

## Більш детально про QEMU-Eclipse

GNU MCU Eclipse QEMU - це розгалуження публічного проекту QEMU з відкритим вихідним кодом, налаштований для більшої підтримки ядер Cortex-M, а також краща інтеграція з плагіном GNU ARM QEMU Debugging.

### Чому QEMU?

Додавання підтримки емулятора до GNU MCU Eclipse вимагало достатньо часу для реалізації. До кінця 2014 року було прийнято рішення використовувати QEMU, емулятор з відкритим вихідним кодом. Він єготовий, він забезпечує хорошу основу для емуляції, він реалізує сервер GDB, він забезпечує підтримку semihosting, і хоча він не має наміру точно імітувати поведінку процесорного часу, він зберігає досить точний відлік часу, коли емулює переривання таймера , тому його можна також використовувати для емуляції програм, що використовують RTOS.

### Чому GNU MCU Eclipse QEMU?

QEMU - відмінний проект, але його оригінальний фокус полягав у тому, щоб емулювати плати з великими ядрами, як правило, класом додатків, здатними працювати з ядрами Unix / Linux. Підтримка голих плат на базі Cortex-M була доступна тільки для дуже обмеженого діапазону ядер Cortex-M3, тому мало використовуються в GNU MCU Eclipse.

Більш того, підтримка семіхостінгу в загальнодоступній версії QEMU була порушена, а деталізація, необхідна для інтеграції з модулем QEMU, відсутня, тому вона не може бути використана з модулями GNU MCU Eclipse.

### Переваги GNU MCU Eclipse QEMU

Основними перевагами використання GNU MCU Eclipse QEMU є:

користувачі можуть запускати blinky проекти, створені шаблонами GNU MCU Eclipse, негайно, без будь-яких специфічних апаратних засобів, ні плати розробки, ні програміста JTAG; підтримуються всі функції налагодження, такі як одиночний степінг, перевірка змінних, регістри тощо.

Емулятор забезпечує дуже зручну платформу для запуску модульних тестів, повністю підтримуючи необхідні для цього функції.

### Підтримувані плати і мікроконтролери

Плати які зараз підтримуються GNU MCU Eclipse QEMU, такі:

* **Maple**
* **NUCLEO-F103RB**
* **NetduinoGo**
* **NetduinoPlus2**
* **STM32-E407**
* **STM32-H103**
* **STM32-P103**
* **STM32-P107**
* **STM32F4-Discovery**
* **STM32F429I-Discovery**

Підтримуються наступні мікроконтролери Cortex-M3 і Cortex-M4:

* **STM32F103RB**
* **STM32F107VC**
* **STM32F405RG**
* **STM32F407VG**
* **STM32F407ZG**
* **STM32F429ZI**
* **STM32L152RE**

Триває робота з підтримки наступних плат, але дата випуску не встановлена:

* **EK-TM4C123GXL**
* **FRDM-K20D50M**
* **FRDM-**
* **FRDM-K64F**
* **FRDM-KL25Z**
* **FRDM-KL26Z**
* **FRDM-KL43Z**
* **KL[12]7 MCUs**
* **FRDM-KL46Z**
* **MCUs**
* **LPCXpresso**
* **NUCLEO-F334R8**
* **NUCLEO-F411RE**
* **NUCLEO-L152RE**
* **Netduino2**
* **OLIMEXINO-STM32**
* **SAM3-H256**
* **STM32F0-Discovery**
* **STM32F3-Discovery**
* **STM32VL-Discovery**
* **TWR-K60F120M**
* **XMC 2Go**
* **XMC1100 Boot Kit**
* **XMC1200 Boot Kit**
* **XMC1300 Boot Kit**
* **XMC4200 Enterprise**
* **XMC4400 Enterprise Kit**
* **XMC4500 Enterprise Kit**
* **XMC4500 Relax Kit**
* **XMC4500 Relax Lite Kit**

### Сумісність та периферійні пристрої

GNU MCU Eclipse QEMU підтримує певний ступінь сумісності з оригінальною qemu, але він не включає великі ARM-ядра, зосереджені лише на ядрах Cortex-M.

Основна вимога сумісності для емулятора полягала в тому, щоб підтримувати всі пристрої, які розглядалися під час стандартної ініціалізації CMSIS, які зазвичай відносяться до параметрів PLL & clock.

Наступна вимога полягала в тому, щоб повністю підтримати GPIO, на рівні, що дозволяє одиному або більше світлодіодів блимати.

### SysTick

Blinky проекти, створені шаблонами GNU MCU Eclipse (і всі проекти STM32 HAL), використовують SysTick для вимірювання часу, тому будь-яка реалістична емуляція повинна підтримувати SysTick. GNU MCU Eclipse QEMU не тільки робить це, але й забезпечує відносно точні часові інтервали, а виняток SysTick може використовуватися від простого вимірювання часу до керування планувальником реального часу з розумним числом перемикань завдань в секунду (1000 Гц в порядку) .

### Обмеження

* QEMU точно емулює інструкції ядра Cortex-M3 і M4; специфічні команди з плаваючою комою Cortex-M4 не підтримуються;
* Поточна реалізація NVIC використовує певний код контролера переривань з більших ядер і не дуже точний.

### Документація

Офіційна документація QEMU доступна в двох посібниках:

Емулятор QEMU - Документація користувача,

Внутрішні елементи QEMU розташовані в папці doc у місці інсталяції. Офіційний сайт QEMU також має сторінку з різними посиланнями на посібники QEMU.

Всі надають багато інформації про QEMU, але не вдаються до деталей, пов'язаних з Cortex-M.

### Майбутній розвиток

Після початкової версії доведено, що ідея визначення периферійного регістра через таблиці є функціональною, стало очевидно, що ці таблиці не для людей, а для машин. Ці визначення, так чи інакше, вже доступні з файлів SVD SVM (дуже великі XML). Таким чином, для використання QEMU, наступним кроком є обробка SVD-файлів і генерування JSON-файлів, з периферійними регістрами, точно так, як це потрібно для швидкого завантаження емулятором при запуску. Потім, з файлів JSON, можуть бути згенеровані об'єкти реєстру периферійних даних під час виконання.

В даний час не існує причин для цих змін, але якщо хтось зацікавлений підтримати розвиток, пріоритети можуть бути змінені.

### Оновлення

Як розгалуження публічного проекту з відкритим вихідним кодом QEMU.Ппланується, що GNU MCU Eclipse QEMU буде слідувати випускам QEMU, а також буде дотримуватися графіка випуску GNU MCU Eclipse для конкретних функцій.

# Особливості налаштування платформи емуляції

Так як QEMU - це проект з відкритим сирцевим кодом то для початку необхідно завантажити всі необхідні файли, встановити необхідні бібліотеки та “зібрати” проект у один виконуючий файл.

Це можна зробити як на Windows так і на Linux системі. У своєму прикладі я буду використовувати Windows 10 з встановленої на неї спеціальної програми яка дозволяє використовувати команди Linux - Windows Subsystem for Linux (WSL) . На далі всі команди будуть виконуватися у терміналі WSL.

## Компіляція та збірка. Встановлення

Для завантаження всіх файлів використаємо Git, тож запускаємо WSL:

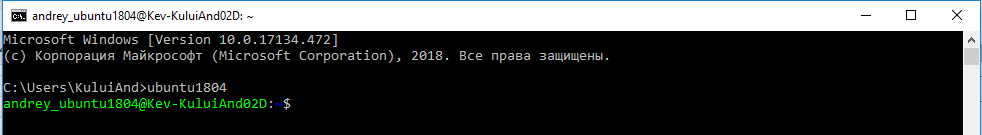
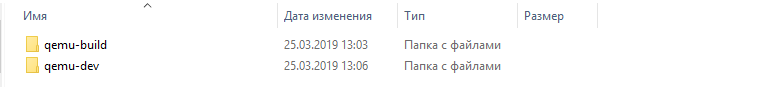


Рисунок 2.1.1 — Термінал WSL

$ git clone https://github.com/gnu-mcu-eclipse/qemu

Після чого в нашій директорії з’являться наступні папки.



В qemu-build знаходяться файли для збірки усього проекту, а в qemu-dev необхідні для цього бібліотеки.

Перед збіркою всього проекту у виконуючий файл необхідно ще встановити пакет програм. Одна з причин чому я використовую термінал Linux це можливість дуже швидко та зручно встановити ці самі необхідні пакети:

$ sudo apt-get update

$ sudo apt-get upgrade

$ sudo dpkg --configure –a

$ sudo apt-get -f install

$ sudo apt-get install the\_package - this step is optional

$ sudo apt install python3

$ sudo apt install gcc

$ sudo apt install -y pkg-config

$ sudo apt-get update --fix-missing

$ sudo apt install bison

$ sudo apt install flex

$ sudo apt-get install git libglib2.0-dev libfdt-dev libpixman-1-dev zlib1g-dev

$ sudo apt-get install git-email

$ sudo apt-get install libaio-dev libbluetooth-dev libbrlapi-dev libbz2-dev

$ sudo apt-get install libcap-dev libcap-ng-dev libcurl4-gnutls-dev libgtk-3-dev

$ sudo apt-get install libibverbs-dev libjpeg8-dev libncurses5-dev libnuma-dev

$ sudo apt-get install librbd-dev librdmacm-dev

$ sudo apt-get install libsasl2-dev libsdl1.2-dev libseccomp-dev libsnappy-dev libssh2-1-dev

$ sudo apt-get install libvde-dev libvdeplug-dev libxen-dev liblzo2-dev

$ sudo apt-get install valgrind xfslibs-dev

$ sudo apt-get install libnfs-dev libiscsi-dev

$ sudo apt-get install libsdl2-dev

$ sudo apt-get install libsdl2-image-2.0-0

$ sudo apt install npm

$ sudo npm install xsvd –global

На даний момент у нас є все необхідне для збірки. Сам проект Qemu являє собою велику кількість Сі фалів тому для їх збірки необхідно лише створити та запустити так званий Makefile, за це вже потурбувався розробник QEMU.

Він створив так званий SHELL скрипт який робить все сам. Тож переходимо у директорію де знаходиться скрипт та запускаємо його:

$ cd /mnt/…/qemu-build/scripts/

$ ./build-native.sh

Після виконання цієї команди ми отримаємо виконуючий файл який далі будемо використовувати для емуляції.

## Запуск емуляції та огляд налаштувань

Тепер маючи виконуючий файл ми можемо запустити сам емулятор.

Qemu не має графічного інтерфейсу тож виконувати все будемо через термінал WSL.

Ось основні параметри які нам знадобляться для запуску емуляції.

**--board “ім’я плати”** – параметр який вказує для вибору плати

**--board ?** – дізнатися всі доступні для емуляції плати.

**--mcu “ім’я мікроконтролера”** – параметр який вказує мікроконтролер що буде емулюватися

**--mcu ?** – дізнатися всі доступні мікроконтролери

**--image** **“ім’я образу програми”** – В даний час емулятор не зберігає постійний стан флеш-пам'яті між прогонами; замість цього вона починається з порожньої флеш памяті (без прошивки) і попередньо завантажує її вмістом, доступним у файлі ELF. Для сеансів налагодження це можна зробити за допомогою протоколу GDB. Для автономних сеансів QEMU потрібно знати ім'я файлу ELF:

**--gdb tcp::4444** – При використанні з відладчиком QEMU повинен реалізовувати протокол GDB-сервера, щоб клієнт GDB міг підключитися до нього. Порт QEMU за замовчуванням - 1234; можливість запуску QEMU в режимі сервера і прослуховування по TCP-порту 1234

У режимі сервера GDB більше не потрібно вказувати зображення ELF, оскільки воно буде завантажено клієнтом GDB, як і в будь-якому звичайному сеансі налагодження, яке переписує флеш-пам'ять MCU.

**--nographic** - За замовчуванням, GNU MCU Eclipse QEMU спробує запустити в графічному режимі, показуючи зображення емуляції плати і одного або більше анімованих світлодіодів. Для запущених модульних тестів це взагалі не потрібно, і QEMU може бути запущений з відключеною підтримкою графіки

**--verbose** - За замовчуванням qemu дуже тиха, вона ледве скаржиться на помилки. Щоб зробити його мінімально соціальним, додайте одну -- verbose, і вона розповість, що відбувається

**-d** - За замовчуванням QEMU навіть якщо він виявляє проблеми під час емуляції, він не повідомляє про них.

Щоб зробити щоб QEMU інформував користувача про будь які проблеми, можна використовувати опції -d для включення деяких параметрів налагодження.

Є багато таких варіантів, але ті, які використовує плагін

**--semihosting-config** - QEMU повністю реалізує протокол ARM semihosting. Щоб увімкнути його, скористайтеся наступним складним варіантом

**--semihosting-cmdline** - Для повноцінного використання семіхостінга, також має бути можливість передавати параметри командного рядка до емульованого програми. Можна використовувати будь-яку кількість опцій, але, як правило, рекомендується звести це до мінімуму, оскільки більшість вбудованих програм виділятимуть лише невеликий буфер, щоб отримати опції напів витрат, і якщо цей буфер буде переповнений, жодних опцій не буде . Ця опція повинна бути останньою і всі наступні параметри будуть передані до програми

Перший параметр буде переданий як argv [0], тому він зазвичай повинен відображати ім'я програми. Наступні параметри будуть передані як argv [1]… argv [n-1], без спеціальної обробки.

**exit() (code)** - При запуску в режимі semihosting, якщо програма main () повертає 0, а програма є повністю семіхостованою, QEMU також повертає 0 до батьківського процесу. Це особливо корисно при запуску модульних тестів, оскільки 0 означає, що тест пройшов успішно. На жаль, напівперіодичні характеристики не дозволяють передати повний код виходу, тому для всіх інших повернутих значень QEMU поверне 1, який може бути використаний як індикатор невдалого тестування.

Це не є великим обмеженням, оскільки модульні тести можна налаштувати для запису XML-файлу з детальними результатами тестування, тому двійковий код виходу достатній для представлення пропуску / невдачі, а системи безперервної інтеграції зазвичай оброблятимуть файл XML деталі.

Тепер для запуску емуляції виконуємо наступну команду.

$ ./qemu-system-gnuarmeclipse /

--verbose /

--verbose /

--mcu STM32F446RE /

--image /mnt/d/Andrey/QEMU\_projects/qemuEclipse/trunk/Main/srs/OUT/Blink.out /

--nographic /

-d func

Після цієї команди запуститься емуляція мікроконтролера STM32F446RE з програмою Blink.out.

## Додавання нового мікроконтролера до QEMU

Після встановлення QEMU ми маємо лише декілька підтримуваних мікроконтролерів, в своєму прикладі я використовую STM32F446RE мікроконтролер, і його не має у списках підтримуваних. Тож першо-черговою задачею є створення нового мікроконтролера для QEMU.

В QEMU реалізований спеціальний механізм по додаванню нових мікроконтролерів. Для цього він використовую спеціальній файл у форматі .svd цей файл зазвичай носить ім’я того мікроконтролера для якого він був згенерований. В моєму випадку мені необхідно завантажити STM32F446.svd файл з сайту виробника цього мікроконтролера.

SVD файл має наступну структуру:

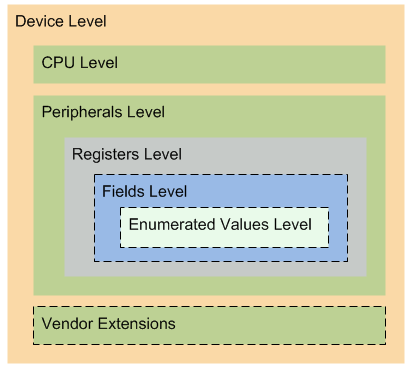


Рисунок 2.3.1 — Структура SVD файлу

Цей файл має повний опис всього мікроконтролера, всіх його регістрів, всіх параметрів, а також детальний опис у сього адресного простору.

Але сам по собі він не має цінності, цей файл необхідний для генерування набору Сі файлів які будуть виконувати роль периферійних блоків даного мікроконтролера.

Цим займається три SHELL скрипта:

- convert.sh

- patch.sh

- code.sh

З початку необхідно виконати convert.sh дописавши йому шлях до завантаженого STM32F446.svd файлу

На приклад так:

echo

xsvd convert \

--file "../../../../../SVD/STM32F446.svd" \

--output "STM32F446-xsvd.json" \

--verbose

Після чого виконуємо цей скрипт convert.sh:

$ ./convert.sh

Та на виході отримуємо новий файл STM32F446-xsvd.json

Наступний крок це запуск patch.sh скрипт, але він використовує два файла один з яких ми щойно створили а другий це STM32F446-patch.json

Його необхідно створити власноруч відштовхуючись від реальних параметрів мікроконтролера:

{

"comment" "Patch for QEMU use.",

"device" {

"name" "STM32F446",

"cpu" {

"name" "CM4",

"revision" "r1p0",

"endian" "little",

"mpuPresent" "false",

"fpuPresent" "false",

"nvicPrioBits" "3",

"deviceNumInterrupts" "96",

"vendorSystickConfig" "false",

"qemuItmPresent" "true"

},

"access" "read-write",

"qemuAlignment" "any",

"peripherals" [

{

"name" "GPIOA",

"qemuGroupName" "GPIO"

},

{

"name" "GPIOB",

"qemuGroupName" "GPIO"

},

{

"name" "GPIOC",

"qemuGroupName" "GPIO"

},

{

"name" "GPIOD",

"qemuGroupName" "GPIO"

},

{

"name" "GPIOE",

"qemuGroupName" "GPIO"

},

{

"name" "GPIOF",

"qemuGroupName" "GPIO"

},

{

"name" "GPIOG",

"qemuGroupName" "GPIO"

},

{

"name" "GPIOH",

"qemuGroupName" "GPIO"

},

{

"name" "USART1",

"qemuGroupName" "USART",

"qemuAlignment" "word-halfWord"

},

{

"name" "USART2",

"qemuGroupName" "USART",

"qemuAlignment" "word-halfWord"

},

{

"name" "USART3",

"qemuGroupName" "USART",

"qemuAlignment" "word-halfWord"

},

{

"name" "USART6",

"qemuGroupName" "USART",

"qemuAlignment" "word-halfWord"

},

{

"name" "UART4",

"qemuGroupName" "UART",

"qemuAlignment" "word-halfWord"

},

{

"name" "UART5",

"qemuGroupName" "UART",

"qemuAlignment" "word-halfWord"

},

{

"name" "ADC1",

"qemuGroupName" "ADC"

},

{

"name" "ADC2",

"qemuGroupName" "ADC"

},

{

"name" "ADC3",

"qemuGroupName" "ADC"

},

{

"name" "I2C1",

"qemuGroupName" "I2C"

},

{

"name" "I2C2",

"qemuGroupName" "I2C"

},

{

"name" "I2C3",

"qemuGroupName" "I2C"

},

{

"name" "CAN1",

"qemuGroupName" "CAN"

},

{

"name" "CAN2",

"qemuGroupName" "CAN"

},

{

"name" "DMA1",

"qemuGroupName" "DMA"

},

{

"name" "DMA2",

"qemuGroupName" "DMA"

},

{

"name" "SPI1",

"qemuGroupName" "SPI"

},

{

"name" "SPI2",

"qemuGroupName" "SPI"

},

{

"name" "SPI3",

"qemuGroupName" "SPI"

}

]

}

}

А також необхідно дописати сам скрипт:

echo

xsvd patch \

--file "STM32F446-xsvd.json" \

--patch "STM32F446-patch.json" \

--output "../STM32F446-qemu.json" \

--remove "NVIC" \

--group-bitfield "RCC/PLLCFGR/PLLQ" \

--group-bitfield "RCC/PLLCFGR/PLLP" \

--group-bitfield "RCC/PLLCFGR/PLLN" \

--group-bitfield "RCC/PLLCFGR/PLLM" \

--group-bitfield "RCC/CFGR/SWS" \

--group-bitfield "RCC/CFGR/SW" \

--verbose

Після чого ми готові виконати цей скрипт:

$ ./patch.sh

На виході цього скрипта ми отримаємо STM32F446-qemu.json файл

Тепер ми дописуємо шлях до нього в останньому скрипту і виконуємо його:

echo

xsvd code \

--file "../STM32F446-qemu.json" \

--verbose

$ ./code.sh

Буде згенеровано велика кількість Сі файлів для кожного периферійного блоку МК окремо які на даний момент ніяк не повязанні з QEMU але вони будуть активно використовуватися якщо ми захочимо використовувати якийсь новий периферійний блок (USART, I2C, SPI…).

Ну і щоб тепер qemu побачило новий мікроконтролер необхідно прив’язати деякі файли що ми згенерували щойно.

Шукаємо mcu.c файл. Це файл де проходить ініціалізація усіх МК та їх периферія. Тож зараз нам необхідно в цьому файлі створити нову структуру та зробити посилання на згенеровані файли:

...

static const STM32Capabilities stm32f446xx = {

.family = STM32\_FAMILY\_F4,

.f4 = {

.is\_446x = true

/\*\*/

},

.hsi\_freq\_hz = 18000000,

.lsi\_freq\_hz = 32000,

#if 0

.has\_rcc = true,

.has\_pwr = true,

.has\_exti = true,

.has\_syscfg = true,

#endif

.num\_exti = 23,

.has\_rtc = true,

.num\_back\_bytes = 80,

.has\_periph\_bitband = true,

.ccm\_size\_kb = 64,

.back\_sram\_size\_kb = 4,

.has\_crc = true,

.has\_dma1 = true,

.num\_dma1 = 8,

.has\_dma2 = true,

.num\_dma2 = 8,

.has\_fsmc = true,

.has\_ac\_tim1 = true,

.has\_ac\_tim8 = true,

.has\_gp\_tim2 = true,

.has\_gp\_tim3 = true,

.has\_gp\_tim4 = true,

.has\_gp\_tim5 = true,

.has\_gp\_tim9 = true,

.has\_gp\_tim10 = true,

.has\_gp\_tim11 = true,

.has\_gp\_tim12 = true,

.has\_gp\_tim13 = true,

.has\_gp\_tim14 = true,

.has\_bc\_tim6 = true,

.has\_bc\_tim7 = true,

.has\_iwdg = true,

.has\_wwdg = true,

.has\_i2c1 = true,

.has\_i2c2 = true,

.has\_i2c3 = true,

.has\_usart1 = true,

.has\_usart2 = true,

.has\_usart3 = true,

.has\_usart6 = true,

.has\_uart4 = true,

.has\_uart5 = true,

.has\_spi1 = true,

.has\_spi2 = true,

.has\_spi3 = true,

.has\_i2s1 = true,

.has\_i2s2 = true,

.has\_plli2s = true,

.has\_sdio = true,

.has\_eth = true,

.has\_bx\_can1 = true,

.has\_bx\_can2 = true,

.has\_usb\_otg\_fs = true,

.has\_usb\_otg\_hs = true,

.has\_dcmi = true, /\* Only 407, not 405 \*/

.has\_rng = true,

.has\_gpioa = true,

.has\_gpiob = true,

.has\_gpioc = true,

.has\_gpiod = true,

.has\_gpioe = true,

.has\_gpiof = true,

.has\_gpiog = true,

.has\_gpioh = true,

.has\_gpioi = true,

.has\_adc1 = true, /\* 12-bit, 16 channels \*/

.has\_adc2 = true,

.has\_adc3 = true,

.has\_ts = true, /\* ADC1\_IN16 \*/

.has\_dac1 = true, /\* 12-bit \*/

.has\_dac2 = true,

/\*\*/

};

...

{

.name = TYPE\_STM32F446RE,

.cortexm = {

.flash\_size\_kb = 512,

.sram\_size\_kb = 128, /\* 64K CCM not counted \*/

.svd\_file\_name = "STM32F446-qemu.json",

.svd\_device\_name = "STM32F446", },

.stm32 = &stm32f446xx,

/\*\*/

},

...

Ми закінчили створення усіх необхідних пакетів для додавання їх до QEMU.

Тепер необхідно у /qemu-dev/qemu.git/hw/cortexm/stm32/**mcus.c** файлдодати нову структуру

static const STM32Capabilities stm32fXXXxx = {};

Ця структура описує, які периферійні пристрої будуть використовуватися для мікроконтролера, і вказується частота синхронізації і кількість переривань MК.  
  
Тепер необхідно додати структуру, яка визначає, скільки і яка пам'ять знаходиться в нашому мікроконтролері.

static const STM32PartInfo stm32\_mcus[] = {};

Також в /qemu-dev/qemu.git/include/hw/cortexm/stm32/mcus.h ми додаємо визначення з нашим MC.

#define TYPE\_STM32FXXXXX "STM32FXXXXX

Далі необхідно налаштувати трактування мікроконтролера у /qemu-dev/qemu.git/hw/cortexm/stm32/**rcc.c** файлі.

case STM32\_FAMILY\_F4:

Також якщо скомпілювати проект то ми можемо отримати помилки пов’язані з відсутніми полями в регістрах щойно доданих компонентів. Для запобігання цього необхідно звернутися до з генерованої папки support та скопіювати з відповідних периферійних блоків відповідні регістри.

По аналогії ми можемо додати всю необхідну периферію, при необхідності, наприклад:

**- exit.c**

**- flash.c**

**- gpio.c**

**- rcc.c**

**- usart.c**

Та багото іншої, і процес додавання буде точно такий як і в попередньому прикладі.

На даний момент QEMU має невелике обмеження пов’язане з відсутності підтримки FPU (Floating Point Unit) або операція з плавучою комою.

FPU рахується також як периферія мікроконтролера та дозволяє виконувати дії з числами набагато швидше якщо б це виконувалося без FPU.

Тому це треба враховувати при створені проекту для мікроконтролера інакше QEMU при компіляції видасть помилку та повідомить про нездатність емулювати FPU.

Враховуючи все сказане ми можемо перекомпілювати проект та виконати команду “mcu ?” щоб побачити що у списку підтримуваних контролерів тепер присутній STM32F446RE.

На далі ми зможемо проводити емуляцію як тільки з одним мікроконтролером так і з платою яка буде в себе включати МК та якусь кількість елементів таких як світло діоди, кнопки, зовнішня FLASH пам'ять та інше.

## Додавання нової плати

Плата це сукупність мікроконтролера та різних елементів які під’єднані до МК. Такі як світлодіоди, кнопки, та ін.

В більшості випадків сам по собі мікроконтролер не тестують а тестують пристрій у цілому. Тому зараз я покажу як створити плату з периферією та потрібним мікроконтролером.

Для того щоб створити нову плату в першу чергу необхідно створити Сі файл в якому ми опишемо її функціонал.

#include <hw/cortexm/board.h>

#include <hw/cortexm/stm32/mcus.h>

#include <hw/cortexm/gpio-led.h>

#include <hw/cortexm/button-gpio.h>

#include <hw/cortexm/gpio-pwm.h>

#include <hw/cortexm/board-elements/tim-pwm.h>

#include <hw/cortexm/board-elements/pre\_stack.h>

#include <hw/cortexm/board-elements/analog-adc.h>

#include <hw/cortexm/board-elements/motor.h>

#include <hw/cortexm/button-reset.h>

#include <hw/cortexm/board\_physical\_elements.h>

#include <hw/cortexm/helper.h>

// ----- ST STM32F4-Discovery -------------------------------------------------

// http://www.st.com/content/st\_com/en/products/evaluation-tools/product-evaluation-tools/mcu-eval-tools/stm32-mcu-eval-tools/stm32-mcu-discovery-kits/stm32f4discovery.html

static GPIOLEDInfo stm32f4\_discovery\_leds\_info[] = {

{

.name = "led:green",

.active\_low = false,

.colour\_name = "green",

.x = 258,

.y = 218,

.w = 8,

.h = 10,

.gpio\_path = DEVICE\_PATH\_STM32\_GPIO\_D,

.irq\_name = STM32\_IRQ\_GPIO\_ODR\_OUT,

.gpio\_bit = 12,

/\*\*/

},

{

.name = "led:orange",

.active\_low = false,

.colour\_name = "orange",

.x = 287,

.y = 246,

.w = 8,

.h = 10,

.gpio\_path = DEVICE\_PATH\_STM32\_GPIO\_D,

.irq\_name = STM32\_IRQ\_GPIO\_ODR\_OUT,

.gpio\_bit = 13,

/\*\*/

},

{

.name = "led:red",

.active\_low = false,

.colour\_name = "Red",

.x = 258,

.y = 274,

.w = 8,

.h = 10,

.gpio\_path = DEVICE\_PATH\_STM32\_GPIO\_D,

.irq\_name = STM32\_IRQ\_GPIO\_ODR\_OUT,

.gpio\_bit = 14,

/\*\*/

},

{

.name = "led:blue",

.active\_low = false,

.colour\_name = "blue",

.x = 230,

.y = 246,

.w = 8,

.h = 10,

.gpio\_path = DEVICE\_PATH\_STM32\_GPIO\_D,

.irq\_name = STM32\_IRQ\_GPIO\_ODR\_OUT,

.gpio\_bit = 15,

/\*\*/

},

{ },

/\*\*/

};

static ButtonGPIOInfo stm32f4\_discovery\_buttons\_user\_info[] = {

{

.name = "button:user",

.active\_low = false,

.gpio\_path = DEVICE\_PATH\_STM32\_GPIO\_A,

.irq\_name = STM32\_IRQ\_GPIO\_IDR\_IN,

.gpio\_bit = 0,

.pull\_up\_down = BUTTON\_PULL\_STATE\_DOWN,

/\*\*/

},

{ },

/\*\*/

};

static ButtonResetInfo stm32f4\_discovery\_button\_reset\_info = {

.x = 262,

.y = 324,

.w = 40,

.h = 40,

/\*\*/

};

static void stm32f4\_discovery\_board\_init\_callback(MachineState \*machine)

{

CortexMBoardState \*board = CORTEXM\_BOARD\_STATE(machine);

cortexm\_board\_greeting(board);

#ifdef ENABLE\_GRAPHIC

BoardGraphicContext \*board\_graphic\_context = cortexm\_board\_init\_graphic\_image(board, "STM32F4-Discovery.jpg");

#else

BoardGraphicContext \*board\_graphic\_context = NULL;

#endif

BoardPhysicalContext \*board\_physical\_context = g\_malloc0\_n(1, sizeof(BoardPhysicalContext)); // выделяем память под нашу структуру где будут хранится кнопки

{

// Create the MCU.

Object \*mcu = cm\_object\_new\_mcu(machine, TYPE\_STM32F407VG);

// Set the board specific oscillator frequencies.

cm\_object\_property\_set\_int(mcu, 0, "hse-freq-hz"); // 8.0 MHz

cm\_object\_property\_set\_int(mcu, 32768, "lse-freq-hz"); // 32 kHz

cm\_object\_realize(mcu);

}

Object \*peripheral = cm\_container\_get\_peripheral();

// Create board buttons.

button\_reset\_create\_from\_info(peripheral, &stm32f4\_discovery\_button\_reset\_info, board\_graphic\_context);

#ifdef ENABLE\_GRAPHIC

button\_gpio\_create\_from\_info(peripheral, stm32f4\_discovery\_buttons\_user\_info, board\_graphic\_context);

//gpio\_led\_create\_from\_info(peripheral, develop\_f446re\_leds\_info, board\_graphic\_context);

#else

bpe\_create\_element\_from\_info(BPE\_ELEMENT\_TYPE\_BUTTON, peripheral, stm32f4\_discovery\_buttons\_user\_info, board\_physical\_context);

bpe\_create\_element\_from\_info(BPE\_ELEMENT\_TYPE\_LED, peripheral, stm32f4\_discovery\_leds\_info, board\_physical\_context);

#endif

}

static void stm32f4\_discovery\_board\_class\_init\_callback(ObjectClass \*oc,

void \*data)

{

MachineClass \*mc = MACHINE\_CLASS(oc);

mc->desc = "ST Discovery kit for STM32F407/417 lines";

mc->init = stm32f4\_discovery\_board\_init\_callback;

}

static const TypeInfo stm32f4\_discovery\_machine = {

.name = BOARD\_TYPE\_NAME("STM32F4-Discovery-University"),

.parent = TYPE\_CORTEXM\_BOARD,

.class\_init = stm32f4\_discovery\_board\_class\_init\_callback,

/\*\*/

};

static void stm32\_machines\_init(void)

{

type\_register\_static(&stm32f4\_discovery\_machine);

}

type\_init(stm32\_machines\_init);

У відповідних структурах ми описуємо периферійні блоки які збираємося використовувати.

Наприклад структура stm32f4\_discovery\_buttons\_user\_info описує кнопки які будуть присутні на створеній платі, а також до якого виводу мікроконтролера вона буде підключена, є чи не має підтяжки кнопки і т.д.

Для того щоб цю плату можна було використовувати спочатку необхідно додати цей Сі файл до Makefile який знаходиться в тій самій директорії що і Сі файл плати. Це необхідно зробити для того щоб при компіляції проекту компілятор знав що треба також скомпілювати новий файл з створеною платою.

## Керування процесом емуляції через QMP

QEMU Machine (QMP) - це протокол на основі JSON, який дозволяє програмам керувати екземпляром QEMU.

Особливості:

* Легкий, текстовий, легкий для аналізу формат даних
* Підтримка асинхронних повідомлень (події)
* Обговорення можливостей
* Гарантії стабільності API / ABI

Для того щоб мати можливість керувати емуляцією віддалено. Спочатку необхідно підключитися до відповідного порту через TCP протокол:

$ telnet localhost 4444

Після чого завжди слідує перша привітальна команда:

$ {'execute':'qmp\_capabilities'}

Після чого вже можна відправляти будь які інші команди і навіть створювати свої власні.

Наприклад для того щоб створити qmp команду для натискання кнопки необхідно лише додати відповідну функцію до qmp.c файлу:

void qmp\_qmp\_button(int64\_t num\_button, bool state, Error \*\*errp)

{

    button\_click\_handler(num\_button, state);

}

А також опис цієї функції у json форматі:

# Since: 2.7

##

{ 'command': 'qmp\_button', 'data': {'num\_button': 'int', 'state': 'bool'} }

Тепер якщо ми відправимо через TSP з’єднання команду

$ { "execute" "qmp-button", "arguments" { "num\_button" 0, "state" true } }

То ми натиснемо на кнопку, а якщо цю команду:

$ { "execute" "qmp-button", "arguments" { "num\_button" 0, "state" false } }

То кнопка буде віджата.

# Навчальні приклади реалізації автоматизованого тестування

В даному розділі я продемонструю приклад використання автоматизованого тестування вбудованих систем на платформі QEMU.

## Постановка задачі для демонстрації

Для демонстрації буде емулюватися існуюча платформа STM32F446RE-NUCLEO.

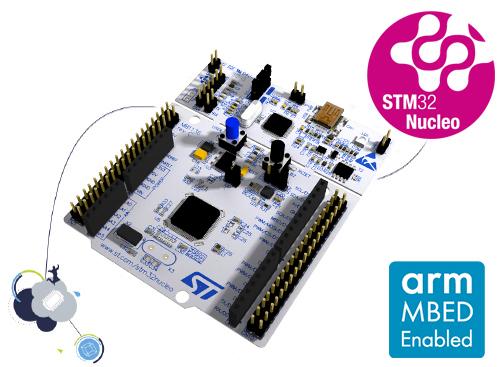


Рисунок 3.1.1 —плата STM32F446RE-NUCLEO

На даній платі присутні кнопки та світлодіоди.

Технічне завдання:

1. Створити програму для STM32F446RE-NUCLEO яка при натисканні кнопки буде запускати блимання світлодіодів. Період блимання 1 секунда. Кількість миготінь 5 раз.
2. Впевнитися в тому що створена програма працює на платі.
3. Створити об’єкт плати в qemu та навчитися натискати на кнопку
4. Створити тест на Python для перевірки коректності роботи емулятора.

## Емуляція інтерфейсів вводу/виводу даних

Інтерфейсами вводу і виводу можуть виступати такі блоки як I2C, SPI, DMA, GPIO…

Ці блоки вже були створенні на етапі коли ми додавали новий МК до складу QEMU. Але вони ще не доступні для використання, їх також потрібно підключити до необхідного мікроконтролера, задати джерело тактування та описати повну логіку їх роботи.

Так як деякі процеси в середині мікроконтролера проходять на чисто апаратному рівні(такі як запис у регістр певного значення після спрацювання переривання), то для коректної емуляції треба це реалізувати в QEMU щоб емуляція була максимально наближена до реальної.

## Скрипти для автоматизації

Скрипт для тестування буде написаний на Python та буде досить простий. Його завданням буде натиснути на кнопку та прослідкувати коректну реакцію на це мікроконтролера, якщо реакція буде очікувана то він відобразить це у консолі.

Таким чином ми зможемо протестувати нашу програму на емуляторі QEMU.

Код скрипту для тестування приведений нижче:

import socket

import sys

import json

import time

from qemu\_class.qmp\_tcp\_socket\_class import QmpTcpSocket

# Main program

HOST, PORT = "localhost", 4444

BUTTON\_NUMBER = 0

qmp\_cmd\_capabilities ='{ "execute": "qmp\_capabilities" }'

template\_qmp\_cmd\_Button\_switch\_on ='{ "execute": "qmp\_button", "arguments": { "num\_button": "none", "state": true } }'

template\_qmp\_cmd\_Button\_switch\_off ='{ "execute": "qmp\_button", "arguments": { "num\_button": "none", "state": false } }'

# Set qemu fram name to settings request

jsonData = json.loads(template\_qmp\_cmd\_Button\_switch\_on)

jsonData['arguments']['num\_button'] = BUTTON\_NUMBER

qmp\_cmd\_Button\_switch\_on = json.dumps(jsonData)

# Set qemu fram name to read request

jsonData = json.loads(template\_qmp\_cmd\_Button\_switch\_off)

jsonData['arguments']['num\_button'] = BUTTON\_NUMBER

qmp\_cmd\_Button\_switch\_off = json.dumps(jsonData)

# Create qmp object and connect to qemu

qmp = QmpTcpSocket(HOST, PORT)

qmp.open\_socket\_connection()

# Get QEMU version data json

jsonData = qmp.send\_receive\_json(None)

print ("Sent: {}".format(None))

print ("Received : ", jsonData)

# Get capabilities

jsonData = qmp.send\_receive\_json(qmp\_cmd\_capabilities.encode())

print ("Sent: {}".format(qmp\_cmd\_capabilities))

print ("Received : ", jsonData)

# Get switch on

jsonData = qmp.send\_receive\_json(qmp\_cmd\_Button\_switch\_on.encode())

print ("Sent: {}".format(qmp\_cmd\_Button\_switch\_on))

print ("Received : ", jsonData)

# switch off

jsonData = qmp.send\_receive\_json(qmp\_cmd\_Button\_switch\_on.encode())

print ("Sent: {}".format(qmp\_cmd\_Button\_switch\_off))

print ("Received : ", jsonData)

# close connection

qmp.close\_socket\_connetcion()

В даному коді використовується бібліотека для парсингу json команд.

Даний код створює з’єднання з QEMU через TCP з’єднання по наперед визначеному порту, і відправляє команди після чого прослуховує порт на можливу відповідь. Що дозволяє створити двох направлений заявок для обміну даних між тестом та виконуваною програмою на стороні емулятора.

Даний підхід має свої недоліки у вигляді невеликих затримок через обмеження швидкості, але підібравши більш гнучкий підхід цей недолік можна нівелювати.

# Охорона праці

В даному розділі розглядаємо питання з безпеки і гігієни праці, які можуть мати місце при проектуванні, виготовленні макета та експлуатації виготовленого виробу.

Основну увагу в цьому розділі присвячено питанням захисту персоналу від ЕМВ, електробезпеки, а також аналізу небезпечних та шкідливих факторів і при використанні обчислювальної техніки, а також відповідності умов праці в робочому приміщенні санітарним нормам та виконанню умов пожежної безпеки.

В даному розділі ( Визначення основних потенційно шкідливих та небезпечних виробничих факторів) запропоновані (Технічне рішення та організаційні заходи з безпеки та гігієни праці).

## Визначення основних потенційно шкідливих та небезпечних виробничих факторів

Процеси розробки та дослідження вентиля на основі Y-циркулятора виконуються в науково-дослідницькій лабораторії та за допомогою персонального комп’ютера (ПК). При роботі на ПК працівник піддається впливу іонізуючого, інфрачервоного й ультрафіолетового випромінювань екрана монітора, рівні яких повинні відповідати вимогам ДСанПіН 3.3.2.007-98 «Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин ЕОМ». Відповідно до ДСанПіН 3.3.2.007-98 максимальний рівень рентгенівського випромінювання 100 мкР/ч. Гранично припустимий рівень ультрафіолетового випромінювання становить 10 Вт/м2 відповідно до ДСанПіН 3.3.2.007-98. Відповідно до ДСанПіН 3.3.2.007-98електромагнітні випромінювання радіочастотного діапазону не повинні перевищувати встановленої норми 10 В/м.

Шкідливими факторами, що знижують працездатність і викликають зміни в організмі людини, є також метеорологічні (температура, вологість, швидкість руху повітря). Газовий склад у виробничому приміщенні часто відрізняється від атмосферного підвищеною концентрацією вуглекислого газу і наявністю пилу.

Штучне освітлення, а саме відсутність у спектрі ламп денного світла й ламп накалювання біологічно активної ультрафіолетової складової при тривалому впливі може призвести до ультрафіолетової недостатності, при якій знижуються бактерицидні властивості шкіри та імунітет. При експлуатації вимірювальних приладів завжди існує небезпека ураження струмом. Крім того, існує небезпека виникнення пожежі, насамперед з причин електричного характеру. При проведенні повного циклу робіт з проектування потенційно небезпечними та шкідливими факторами є:

• можливість ураження електричним струмом;

• підвищений рівень шуму;

• невідповідність освітлення санітарним нормам;

• несприятливі мікрокліматичні умови;

## Технічне рішення та організаційні заходи з безпеки та гігієни праці

### Охорона праці та виробничої санітаріїї при експлуатації ПЕОМ

Відповідно до ДСанПіН 3.3.2.007 - 98 основними шкідливими факторами, зв’язаними з роботою на ПЕОМ є:

* напруга зорових органів і пов’язане з ним стомлення, захворювання і побічні ефекти;
* значне навантаження на пальці і кисті рук, що при відсутності профілактики і медичного контролю, може викликати професійні захворювання;
* тривале перебування в тому самому положенні, що викликає застійні явища в організмі і може привести до різних захворювань;
* механічні шуми, зв’язані з роботою принтера і вентиляційної системи відводу тепла комп’ютера;
* можливість ураження електричним струмом.

Праця за ПЕОМ для користувачів пов'язана з певними особоливостми. Їх основу складає виконання різноманітних за змістом та тривалості робіт, а це в свою чергу викликає високу напругу функцій зору, розумової діяльності.

Основні проблеми, пов’язані з використанням моніторів, відносяться до надзвичайної напруги очей і стомленню користувача. Симптоми напруги очей, як правило, є результат надмірної активності очних м’язів. В умовах роботи користувача комп’ютерів є цілий ряд факторів, що заважає роботі ока — самосвітний об’єкт при позитивному контрасті з фоном, нерівномірний розподіл яскравості, необхідність частого переводу погляду на об’єкти, розміщених на різних рівнях, що вимагає значної напруги зорових функцій і адаптаційних механізмів ока. Здатність очних м’язів відновлюється після достатнього відпочинку і відповідних вправ.

## Вплив ЕМВ

При налагоджувані даного виробу в робочому приміщені необхідно дотримуватися правил техніки безпеки, щодо захисту від ЕМВ відповідно до вимог ДСНіП3”№476 та ДСНіП3”№239. По перше, забороняються будь які роботи з відкритими кінцями хвилеводів. По друге, з метою зниження рівня ЕМВ, обов’язково слід застосовувати еквівалентні поглинаючі хвилеводні навантажувачі. В першу чергу, це стосується використання еквіваленту антени.

Оскільки вентиль працює на частоті 8 - 12 ГГц, то при його налагоджувані людина попадає в дальню зону джерела ЕМВ. В цьому випадку нормується допустима величина густини потоку енергії.

Рівень потужності модуля дорівнює Рімп=20 Вт.). Мінімальна відстань між людиною та модулем при його налагоджувані буде складати 0,5м.

Розрахуємо рівень потужності ГПЕ на робочому місці.

Вт/м2

- середня потужність ( Рср =Римп\*tімп/Т)

Рімп=20 Вт

tімп = 100 нсек

Т = 1/f=50 мсек

G – коефіцієнт направленості (приймаємо для хвилеводного фланця G=1)

Враховуючи те, що згідно з вимогами ГОСТ 12.1.006-84



К = 1; Т = максимум 4 год за робочу зміну.



Таким чином фактичний рівень ГПЕ, яка впливає на людину в робочій зоні нижче припустимих норм і в додаткових заходах по зменшеню рівня ЕМВ в робочій зоні нема необхідності.

### Виробничий шум

Відповідно до ДСН 3.3.6.037-99 рівень шуму в приміщенні для програмістів не повинен перевищувати 50 дБА для режиму налагодження і 65 дБА для режиму введення інформації.

Приміщення розташоване вікнами у двір і знаходиться далеко від проїздної частини вулиці. Основними джерелами шуму в приміщенні є устаткування і люди. Розглянута кімната не призначена для прийому відвідувачів і тому в ній не спостерігається великого скупчення людей. Тому основним джерелом шуму є комп’ютерна техніка.

Джерелами шуму при роботі ЕОМ є механічні частини принтера, що рухаються, і вентилятори. При роботі вентиляційної системи, що забезпечує оптимальний температурний режим електронних блоків ЕОМ і вмонтована в задню панель, створюється аеродинамічний шум. Шум, створюваний працюючим комп’ютером, може бути охарактеризований як широко смужний, постійний з аперіодичним посиленням при роботі принтера.

Механічні пристрої ЕОМ знаходяться всередині корпуса і захищенні кожухом, тому шуму практично не створюють. Крім того, у приміщенні встановлена сучасна техніка, що характеризується низьким рівнем шуму. Лазерні принтери, на відміну від матричних, також характеризуються низьким рівнем шуму.

Таким чином, еквівалентний рівень шуму в приміщення за робочий день не перевищує норму.

### Електробезпека

Згідно ДСТУ ІЕС 61140.2015 та ПУЕ науково-дослідницька лабораторія відноситься до приміщень без підвищеного ризику. Електроустаткування належить до приладів до 1000 В. Устаткування, що використовується, відповідно до ДСТУ ІЕС 61140.2015 належить до устаткування 0І, І та ІІ класів за електрозахистом.

У процесі експлуатації електронно-обчислювального обладнання людина може доторкнутися до частин електроустаткування, які перебувають під напругою. Оцінка небезпеки дотику до струмоведучих частин відноситься до визначення сили струму, що протікає через тіло людини, і порівняння його із допустимим значенням відповідно до ГОСТ 12.1.038-88. У загальному випадку допустима величина струму, що протікає через тіло людини, залежить від схеми підключення електроустаткування до електромережі, роду й величини напруги живлення, схеми включення.

При виконанні розрахунків для дипломного проекту використовувався блок живлення ПК, ПК відповідно ІI і III класів за електрозахистом. Для правильного визначення необхідних засобів та заходів захисту від ураження електричним струмом необхідно знати допустимі значення напруг доторкання та струмів, що проходять через тіло людини.

Гранично допустимі значення напруги дотику та сили струму для нормального (безаварійного) та аварійного режимів електроустановок при проходженні струму через тіло людини по шляху «рука – рука» чи «рука – ноги» регламентуються ПУЕ-2017 (табл.Х.1).

Таблиця Х.1. Граничнодопустимі значення напруги доторкання , що проходить через тіло людини при нормальному режимі електроустановки.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| t,(сек) | До 0,1 | 0,2 | 0,5 | 0,7 | 0,9 |  |
| ,(В) | 500 | 400 | 200 | 130 | 100 | 65 |

Основними технічними засобами, що забезпечують безпеку робіт (згідно ПУЕ-2017) є: надійна ізоляція, захисне заземлення, занулення, захисне відключення, засоби індивідуального захисту. У системі трифазних мереж із глухо заземленою нейтраллю, яка використовується у науково-дослідницькій лабораторії, найкращими засобами захисту є: надійна ізоляція струмоведучих частин електроустаткування відповідно до ПУЕ-2017 і занулення відповідно до ПУЕ (з'єднання елементів, що перебувають під напругою, із глухо заземленою нейтраллю). Крім того, для заземлення переносних частин обладнання застосовують спеціальне з'єднання.

### Розрахунок захисного відключення електромережі при аварійному режимі роботи електрообладнання

Виконаємо розрахунок ланцюга захисного відключення фазного проводу при короткому замиканні (КЗ). Струм КЗ можна обчислити за формулою:

де = 220 В -напруга фазного проводу;

= 3 Ом -опір нульового проводу;

= 7 Ом -опір фазного проводу;

0,1 Ом -еквівалентний опір трансформатора.

Для постійної роботи автоматів захисту з електромагнітним розпилювачем струм його спрацьовування повинен бути в 1,4 рази менше струму короткого замикання при струмі до 100 А.

Таким чином, струм спрацьовування автомату повинен бути менше 15,6А.

Автомати максимально струмового захисту, встановлені у науково-дослідницькій лабораторії задовольняють цій умові (< 15,6 А,< 0,8 с.).

Розрахуємо напругу дотику до корпусів електрообладнання при короткому замиканні:

Відповідно до ПУЕ-2017, щоб ця напруга була безпечна для людини, необхідно використовувати автомати максимально струмового захисту у яких час спрацьовування менше 0,8с.

Автомати максимально струмового захисту, встановлені у науково-дослідницькій лабораторії задовольняють цим умовам (< 15,6 А,< 0,8 с.).

Із проведених розрахунків видно, що у науково-дослідницькій лабораторії основним захистом від поразки електричним струмом є занулення та застосування пристроїв максимального струмового захисту.

# Висновки

Автоматизоване тестування передбачає використання спеціального програмного забезпечення (крім тестованого) для контролю виконання тестів і порівняння очікуваного фактичного результату роботи програми. Цей тип тестування допомагає автоматизувати часто повторювані, але необхідні для максимізації тестового покриття завдання.

Деякі завдання тестування, такі як низькорівневе регресійне тестування, можуть бути трудомісткою і вимагають багато часу якщо виконувати їх вручну. Крім того, мануальное тестування може недостатньо ефективно знаходити деякі класи помилок. У таких випадках автоматизація може допомогти заощадити час і зусилля проектної команди.

Після створення автоматизованих тестів, їх можна в будь-який момент запустити знову, причому запускаються і виконуються вони швидко і точно. Таким чином, якщо є необхідність частого повторного прогону тестів, значення автоматизації для спрощення супроводу проекту і зниження його вартості важко переоцінити. Адже навіть мінімальні патчі і зміни коду можуть стати причиною появи нових багів.

Тож для великих довго тривалих проектів цей підхід підходить, але для проектів які не розраховані на масове виробництво написання та створення таких тестів буде не вигідним і буде займати багато часу та коштів.

# Перелік посилань

1. Biological sensors for solar ultraviolet radiation. Yagura T, Makita K, Yamamoto H, Menck CF, Schuch AP. Sensors (Basel). 2011;11(4):4277-94. Epub 2011 Apr 12.
2. Journal of Photochemistry and Photobiology B: Biology Volume 53, Issues 1–3, 1 November 1999, Pp. 1–6.
3. G. Ronto,S. Gaspar,A. Berces. Phage T7 in biological UV dose measurement.Journal of Photochemistry and Photobiology B Biology. DOI:10.1016/1011-1344(92)85030-X. 03/1992; 12(3), pp. 285-94.
4. E. B. Podgorsak. Radiation oncology physics: a handbook for teachers and students. Vienna: International Atomic Energy Agency, 2005. ISBN 92–0–107304–6. 696p.
5. Полупроводниковые фотоприемники: Ультрафиолетовый, видимый и ближний инфракрасный диапазоны спектра / И.Д. Анисимова, И.М. Викулин, Ф.А. Заитов, Ш.Д. Курмашев; Под ред. В.И.Стафеева. M.: Радио и связь, 1984. 216с.
6. http://ru.wikipedia.org/wiki/Радиометр
7. http://ru.wikipedia.org/wiki/Дозиметр
8. http://jp.hamamatsu.com/products/sensor-etd/pd011/C9536/index\_en.html
9. http://www.hamamatsu.com/jp/en/product/category/3100/4001/4103/S1226-18BU/index.html