**М. О. Притула, І. А. Дудатьєв, Б. О. Пінаєв**

# ПРОГРАМУВАННЯ МІКРОПРОЦЕСОРНИХ СИСТЕМ

Міністерство освіти і науки України
Вінницький національний технічний університет

**М. О. Притула, І. А. Дудатьєв, Б. О. Пінаєв**

# ПРОГРАМУВАННЯ МІКРОПРОЦЕСОРНИХ СИСТЕМ

*Електронний лабораторний практикум*

Вінниця
ВНТУ
2026

УДК 004.272
П75

Рекомендовано до видання Вченою Радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 3 від 30.09.2025 р.)

Рецензенти:
**О. М. Возняк,** кандидат технічних наук, доцент
**Д. В. Михалевський,** доктор технічних наук, професор
**Д. Х. Штофель,** кандидат технічних наук, доцент

**Притула, М. О.**

П75 Програмування мікропроцесорних систем : лабораторний практикум [Електронний ресурс] / Притула М.О., Дудатьєв І. А., Пінаєв Б. О. – Вінниця : ВНТУ, 2026. – (PDF, 138 с.)

Посібник присвячено матеріалам курсу з дисципліни «Програмування мікропроцесорних систем» для здобувачів, що навчаються за спеціальністю «Інформаційно-вимірювальні технології» денної та заочної форм навчання.

Мета посібника – надати здобувачам можливість більш детально вивчити аудиторний матеріал і підготуватися до іспиту, а також застосувати отримані знання для подальшої фахової роботи.

Перелік та зміст тем відповідає програмі вказаної вище дисципліни.

© ВНТУ, 2026

**Зміст**

Лабораторна робота 1
Налаштування Arduino IDE та платформи wokwi ..................4
Лабораторна робота 2
Робота з цифровими входами. Зчитування стану кнопки ......22
Лабораторна робота 3
Робота з аналоговими сигналами .............................................46
Лабораторан робота 4
Серійна комунікація...................................................................68
Лабораторна робота 5
Передача даних по I2C на LCD1602 ........................................89
Лабораторна робота 6
Відображення інформації на OLED дисплеї ............................115
ПЕРЕЛІК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ.............................137

# Лабораторна робота 1
## Налаштування Arduino IDE та платформи wokwi

**Мета**. Ознайомитися з процесом підключення ESP32 до Arduino IDE та перевірити правильність роботи середовища. Також здобувачі мають навчитися:

- встановлювати Arduino IDE;
- додавати підтримку нових плат через менеджер плат;
- підключати мікроконтролер ESP32 до комп'ютера;
- компілювати та завантажувати код у мікроконтролер;
- тестувати роботу скетча Blink для перевірки працездатності мікроконтролера.

### Основні теоретичні відомості

***Arduino IDE*** – це вільно доступне середовище для розробки програмного забезпечення з відкритим кодом, призначене для написання, компіляції та завантаження програм на мікроконтролери серії Arduino і сумісні з ними плати, серед яких ESP32, STM32, ATtiny тощо. Програма сумісна з основними операційними системами – Windows, macOS та Linux, і завдяки своїй простоті та доступності є чудовим вибором для здобувачів і початківців. Вона активно розвивається спільнотою розробників та вважається стандартним інструментом для швидкого створення електронних прототипів.

Arduino IDE існує у двох основних варіантах: версія 1.x – класична і стабільна, та версія 2.x, яка базується на платформі Eclipse Theia та містить низку покращень: зручний редактор коду з автодоповненням, інтегрований відлагоджувач, організація файлів у вкладках, а також вбудовані інструменти, такі як серійний монітор і термінал.

Структура проєкту в середовищі максимально спрощена – кожен проєкт (скетч) розміщується в окремій директорії, де головний файл має розширення .ino. У разі потреби додаються інші файли: .h, .cpp тощо. Компіляція виконується кнопкою «Verify», а завантаження на мікроконтролер – через «Upload». У разі помилки IDE виводить опис проблеми у відповідному вікні.

Arduino IDE також забезпечує доступ до таких функцій:

- перегляд даних через Serial Monitor;
- візуалізація значень за допомогою Serial Plotter;
- встановлення бібліотек із ZIP-архівів або GitHub;
- створення власних плат через опис у форматі JSON для Board Manager;
- робота з кількома підключеними пристроями одночасно.

Для плат ESP32 середовище дозволяє змінювати параметри конфігурації, включно тактову частоту процесора, обсяг пам'яті,

налаштування Wi-Fi, активацію BLE та режимів енергозбереження. Також підтримується оновлення прошивки «по повітрю» (OTA).

Завдяки інтуїтивно зрозумілому інтерфейсу та низькому порогу входу, Arduino IDE ідеально підходить для навчання, хакатонів, перших спроб роботи з мікроконтролерами, створення MVP-прототипів або автоматизації в побуті.

Графічний інтерфейс містить:

- вікно редактора коду;
- кнопки компіляції та завантаження;
- інструменти перегляду серійного порту;
- виведення повідомлень компілятора.

Програми в середовищі називаються скетчами і базуються на мові програмування C/C++. Кожен скетч має базову структуру з функціями setup() (для ініціалізації) та loop() (безперервне виконання коду).

Бібліотеки для роботи з різними пристроями можна встановлювати через Library Manager. Для підтримки нових плат, таких як ESP32, потрібно додати відповідне джерело в Board Manager, після чого доступні пакети з компілятором, завантажувачем і прикладами коду.

Основні переваги Arduino IDE:

- проста конфігурація та використання;
- багатий вибір бібліотек і прикладів;
- сумісність із широким спектром плат;
- підтримка роботи з COM-портами, серійним виводом і базовим налагодженням.

Обмеження:

- не підходить для великих, масштабних проєктів;
- не має повноцінного вбудованого налагоджувача;
- менш гнучка структура проєкту порівняно з середовищами на зразок PlatformIO або Visual Studio Code.

Для мікроконтролерів ESP32 Arduino IDE є дуже зручним інструментом: дозволяє швидко створювати прошивки, тестувати периферію (GPIO, ADC, PWM, Wi-Fi тощо) і користуватися всіма можливостями бездротового оновлення.

Отже, Arduino IDE – це базове, але функціонально потужне середовище для розробки, яке є чудовою точкою старту у світі мікропроцесорної техніки.

***ESP32*** оснащено апаратними засобами безпеки та криптографічними прискорювачами, серед яких:

- Secure Boot – гарантує запуск лише цифрово підписаних та перевірених прошивок;

- Flash Encryption – забезпечує шифрування вмісту пам’яті Flash для захисту даних;

- HMAC і генератор випадкових чисел (RNG) – підтримка апаратного генерування унікальних ключів і автентифікації;

- Anti-rollback захист – запобігає поверненню до старих, потенційно уразливих версій прошивки.

Функціональну блок-діаграму ESP32 наведено на рис. 1.1.

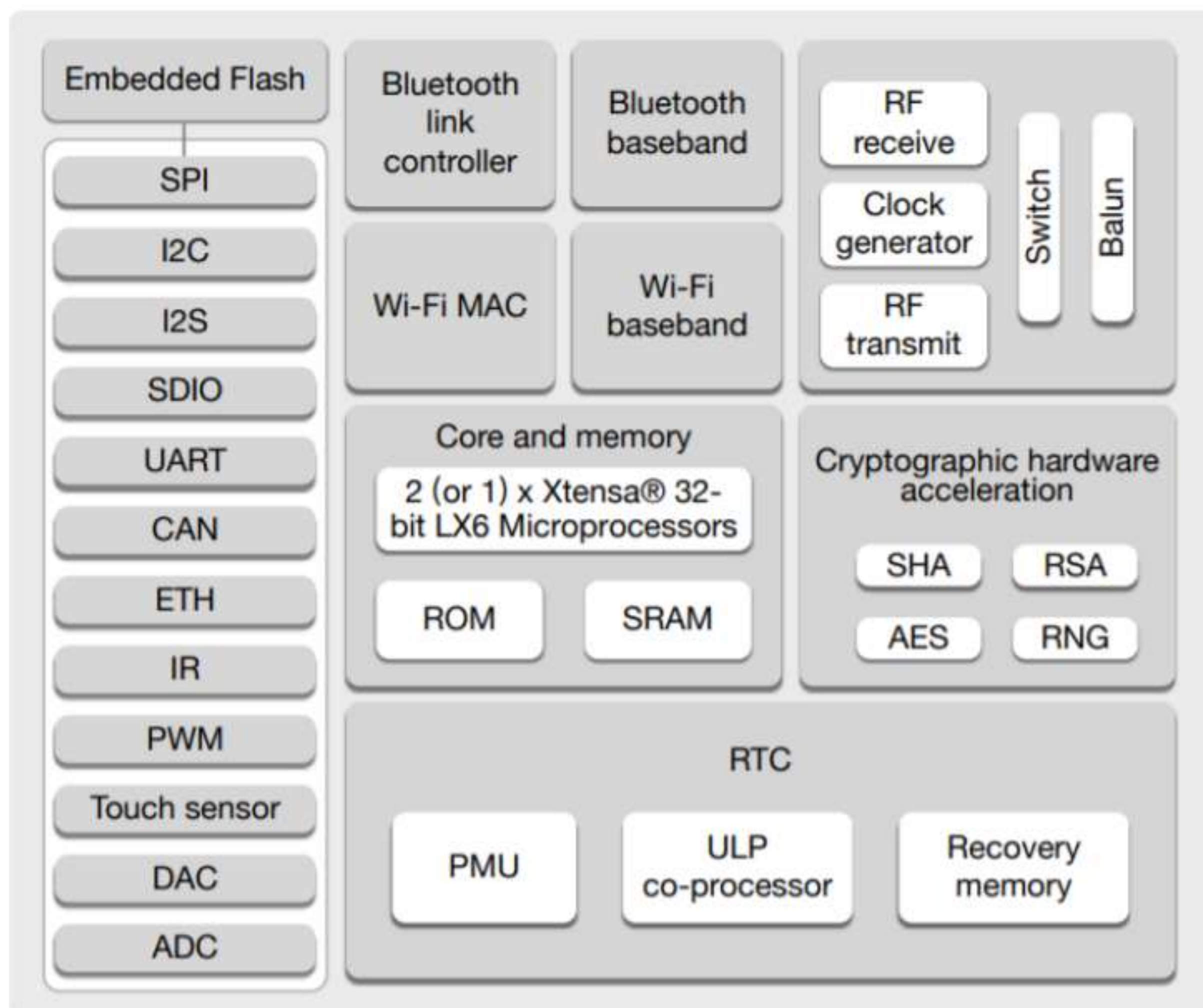



Рисунок 1.1 – Функціональна блок-діаграма ESP32

Серед апаратних особливостей ESP32 – підтримка Ethernet, SDIO, RMT (дистанційне керування), а також вбудованого інтерфейсу для підключення камери (наприклад, у версії ESP32-CAM), що дозволяє використовувати модуль у сферах відеоспостереження, потокової передачі відео та автоматизації.

ESP32 також сумісний із FreeRTOS у середовищі Arduino IDE, що відкриває можливість створювати багатозадачні застосунки з незалежними потоками. Наприклад, окремі завдання можуть відповідати за зв’язок,

опитування датчиків і керування пристроями. Це критично важливо для систем реального часу та підвищеної надійності.

Найбільш поширені варіанти плат ESP32:

- ESP32 DevKitC – базова плата з доступом до більшості виводів GPIO;
- ESP32-WROOM-32 – модуль для інтеграції в інші системи;
- ESP32-CAM – модуль з камерою, підтримкою microSD та оновленням прошивки по Wi-Fi;
- NodeMCU-32S – компактна плата з USB-UART та кнопками Reset і Boot.

Для розробників доступні:

- популярні бібліотеки (наприклад, AsyncWebServer, ESPAsyncTCP, MQTT-клієнти);
- інтеграція з хмарними платформами (Firebase, Blynk, ThingsBoard);
- підтримка зовнішніх компонентів: OLED, TFT-дисплеїв, GPS, сенсорів температури, вологості, руху тощо.

Окремо варто зазначити, що ESP32 поєднує високу функціональність і доступну ціну, що робить його вигідним вибором як для навчання, так і для розробки серійних пристроїв.

Серед доступних периферійних модулів:

- 12-бітний АЦП – до 18 аналогових каналів;
- 8-бітні ЦАПи – 2 канали аналогового виходу;
- таймери – 4 апаратні таймери по 64 біти або до 64 по 16 біт;
- SPI, I2C, UART, CAN – для гнучкої інтеграції з іншими пристроями;
- 10 сенсорних каналів – для ємнісного керування без додаткових чипів;
- 16 каналів PWM – керування двигунами, світлодіодами тощо.

Мікроконтролер може працювати в різних режимах енергозбереження:

- Active mode – повнофункціональний режим;
- Modem-sleep – знижене енергоспоживання за рахунок відключення модуля Wi-Fi;
- Light-sleep – зупинка ядра за збереження роботи частини периферії;
- Deep-sleep – мінімальне споживання з можливістю пробудження за таймером або сигналом з GPIO.

ESP32 також підтримує файлові системи SPIFFS і LittleFS, що дозволяє зберігати налаштування, логи та веб-інтерфейси прямо у Flash-пам’яті. Розпіновку ESP32 наведено на рис. 1.2.

Для зв’язку з мобільними пристроями доступні BLE GATT-сервери, Wi-Fi Direct, що дозволяє реалізовувати бездротові інтерфейси, зокрема для віддаленого керування та оновлення OTA.

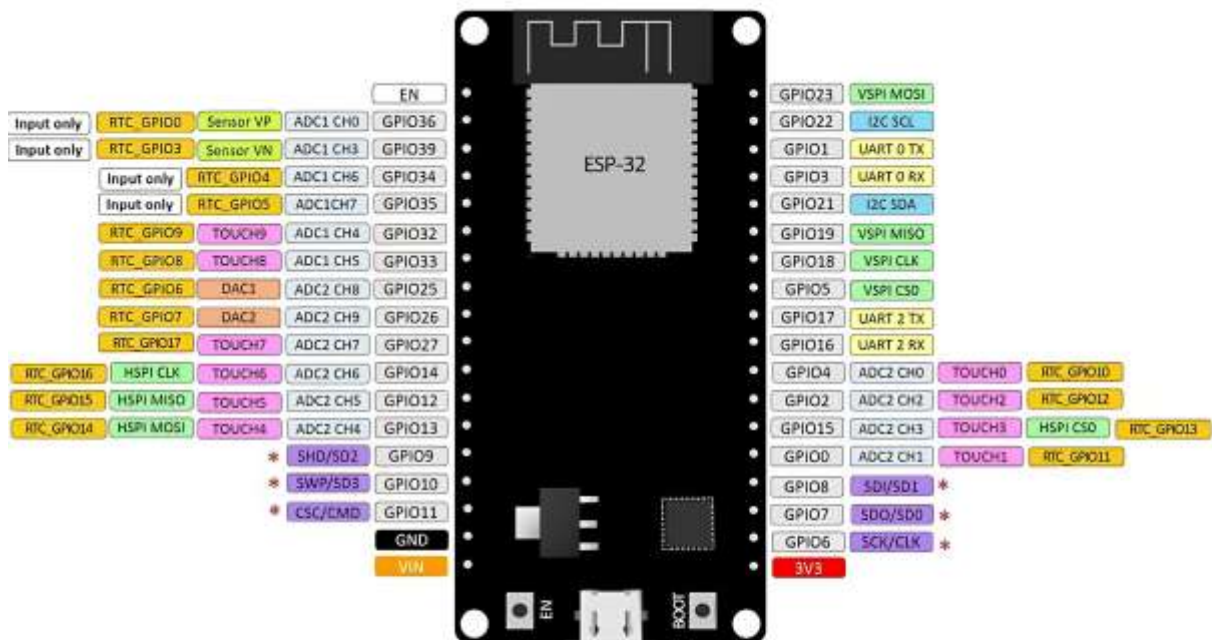



Рисунок 1.2 – Розпіновка ESP32

Можливість OTA-оновлення дозволяє змінювати прошивку ESP32 через Wi-Fi без підключення до комп'ютера – важлива перевага у разі використання в польових умовах.

Таким чином, ESP32 – це потужна платформа для створення багатофункціональних вбудованих систем із підтримкою бездротових технологій, сенсорного керування та реального часу.

Окремо варто згадати RTC-модуль, що дозволяє зберігати точний час навіть у режимах енергозбереження – актуально для автономних пристроїв і систем моніторингу.

ESP32 підтримує кілька бездротових протоколів: Bluetooth Classic – для підключення до звичайних пристроїв (телефони, аудіо), BLE – для енергоефективного зв'язку (маячки, фітнес-трекери), Wi-Fi – у режимах клієнта та точки доступу, включно можливість роботи як веб-сервера без роутера.

У плані розробки:

- Arduino IDE забезпечує простий старт завдяки графічному інтерфейсу, автооновленням бібліотек та прикладам;
- ESP-IDF – офіційне середовище Espressif для професійної розробки з доступом до низькорівневого функціоналу;
- альтернативи: MicroPython, Lua – для швидкого прототипування.

Популярність ESP32 зумовлена широкими можливостями, відкритою екосистемою, підтримкою спільноти та регулярним оновленням документації. Це один із найкращих варіантів для проєктів у сфері IoT, автоматизації, навчання та розробки готових рішень.

У технічному плані:

- два ядра Xtensa LX6 (32-бітні), що можуть працювати паралельно;

- частота до 240 МГц;
- до 520 кБ оперативної пам’яті;
- підтримка зовнішньої Flash (типово – 4 МБ);
- до 34 програмованих GPIO-виводів;
- вбудовані інтерфейси UART, SPI, I2C, CAN, PWM, ADC, DAC, SD-карта тощо;
- криптографічні модулі (AES, SHA, RSA, ECC);
- режими енергозбереження (Light Sleep, Deep Sleep).

Завдяки цим характеристикам ESP32 ефективно застосовується у розумних будинках, переносних пристроях, телеметрії, сенсорних системах та автоматизованому керуванні.

Таблиця 1.1 – Призначення пінів ESP32

| GPIO | Вхід | Вихід | Примітки |
|---|---|---|---|
| 0 | pulled up | OK | виводить ШІМ-сигнал під час завантаження |
| 1 | TX-контакт | OK | вивід налагодження під час завантаження |
| 2 | OK | OK | підключено до вбудованого світлодіода |
| 3 | OK | RX-контакт | ВИСОКИИ за завантаження |
| 4 | OK | OK | |
| 5 | OK | OK | виводить ШІМ-сигнал під час завантаження |
| 6-11 | x | x | підключено до інтегрованої SPI флеш-пам'яті |
| 12 | OK | OK | завантаження не вдається, якщо його підняти високо |
| 13 | OK | OK | |
| 14 | OK | OK | виводить ШІМ-сигнал під час завантаження |
| 15 | OK | OK | виводить ШІМ-сигнал під час завантаження |
| 16-19 | OK | OK | |
| 21-23 | OK | OK | |
| 25-27 | OK | OK | |
| 32-33 | OK | OK | |
| 34 | OK | | лише вхід |
| 35 | OK | | лише вхід |
| 36 | OK | | лише вхід |
| 39 | OK | | лише вхід |

*Піни тільки для входу*

GPIO з 34 до 39 є GPI – тільки вхідні. Ці контакти не мають внутрішніх резисторів, що підтягують або знижують. Вони не можуть бути використані як виходи, тому використовуйте ці контакти лише як входи:

- GPIO 34
- GPIO 35
- GPIO 36
- GPIO 39

*SPI-flash вбудована у ESP-WROOM-32*

Від GPIO 6 до GPIO 11 подано деякі плати ESP32. Однак ці піни підключені до вбудованої флеш-пам'яті SPI на мікросхемі ESP-WROOM-32, їх рекомендується використовувати для інших цілей. Призначення цих пінів:

- GPIO 6 (SCK / CLK)
- GPIO 7 (SDO / SD0)
- GPIO 8 (SDI / SD1)
- GPIO 9 (SHD / SD2)
- GPIO 10 (SWP / SD3)
- GPIO 11 (CSC / CMD)

*Ємнісні сенсорні GPIO*

ESP32 має 10 внутрішніх ємнісних сенсорних датчиків. Вони можуть відстежувати все, що містить електричний заряд, наприклад, вони можуть виявляти зміни, які виникають у разі торканні пальцями GPIO. Ці контакти можуть бути легко вбудовані у датчики торкання та замінювати механічні кнопки. Ємнісні сенсорні контакти можуть бути використані для пробудження ESP32 від глибокого сну.

*Внутрішні сенсорні датчики підключені до цих GPIO:*

- T0 (GPIO 4)
- T1 (GPIO 0)
- T2 (GPIO 2)
- T3 (GPIO 15)
- T4 (GPIO 13)
- T5 (GPIO 12)
- T6 (GPIO 14)
- T7 (GPIO 27)
- T8 (GPIO 33)
- T9 (GPIO 32)

*Аналого-цифровий перетворювач (АЦП)*

ESP32 має вхідні канали АЦП 18 x 12 біт. Це GPIO, які можна використовувати як АЦП:

- АЦП1_CH0 (GPIO 36)
- АЦП1_CH1 (GPIO 37)
- АЦП1_CH2 (GPIO 38)
- АЦП1_CH3 (GPIO 39)

- АЦП1_CH4 (GPIO 32)
- АЦП1_CH5 (GPIO 33)
- АЦП1_CH6 (GPIO 34)
- АЦП1_CH7 (GPIO 35)
- Груповий полімер ADC2_12 (0)
- АЦП2_CH2 (GPIO 2)
- АЦП2_CH3 (GPIO 15)
- АЦП2_CH4 (GPIO 13)
- АЦП2_CH5 (GPIO 12)
- АЦП2_CH6 (GPIO 14)
- АЦП2_CH7 (GPIO 27)
- АЦП2_CH8 (GPIO 25)
- АЦП2_CH9 (GPIO 26)

Примітка: контакти ADC2 не можна використовувати під час використання Wi-Fi. Тому, якщо ви використовуєте Wi-Fi і у вас виникають проблеми з отриманням значення від GPIO ADC2, ви можете натомість розглянути можливість використання GPIO ADC1, що має вирішити вашу проблему.

Вхідні канали АЦП мають роздільну здатність 12 біт. Це означає, що ви можете отримати аналогові показання в діапазоні від 0 до 4095, в яких відповідає 0 – 0 В, а 4095 – 3,3 В. У вас також є можливість встановити дозвіл каналів в коді, а також діапазон АЦП.

*Цифро-аналоговий перетворювач (ЦАП)*

На ESP32 є два 8-бітових канали ЦАП для перетворення цифрових сигналів на аналогові вихідні сигнали напруги.

- CAP1 (GPIO25)
- CAP2 (GPIO26)

*GPIO реального часу*

ESP32 має підтримку RTC GPIO. GPIO, що маршрутизуються в підсистему з низьким енергоспоживанням RTC, можна використовувати, коли ESP32 перебуває у стані глибокого сну. Ці RTC GPIO можуть використовуватися для виходу ESP32 із глибокого сну, коли працює співпроцесор Ultra Low Power (ULP). Наступні GPIO можуть бути використані як зовнішні джерела пробудження.

- RTC_GPIO0 (GPIO36)
- RTC_GPIO3 (GPIO39)
- RTC_GPIO4 (GPIO34)
- RTC_GPIO5 (GPIO35)
- RTC_GPIO6 (GPIO25)
- RTC_GPIO7 (GPIO26)
- RTC_GPIO8 (GPIO33)
- RTC_GPIO9 (GPIO32)
- RTC_GPIO10 (GPIO4)

- RTC_GPIO11 (GPIO0)
- RTC_GPIO12 (GPIO2)
- RTC_GPIO13 (GPIO15)
- RTC_GPIO14 (GPIO13)
- RTC_GPIO15 (GPIO12)
- RTC_GPIO16 (GPIO14)
- RTC_GPIO17 (GPIO27)

*ШІМ*

ШИМ-контролер ESP32 має 16 незалежних каналів, які можна налаштувати для генерації ШІМ-сигналів із різними властивостями. Всі висновки, які можуть виступати як виходи, можуть використовуватися як висновки ШІМ (GPIO з 34 по 39 не можуть генерувати ШІМ).

Щоб встановити сигнал ШІМ, необхідно визначити ці параметри в коді:

- Частота сигналу;
- Робочий цикл;
- ШІМ-канал;
- GPIO, на яких ви бажаєте вивести сигнал.

*I2C*

У випадку використання ESP32 з Arduino IDE потрібно використовувати виводи ESP32 I2C за замовчуванням (підтримуються бібліотекою Wire):

- GPIO 21 (SDA)
- GPIO 22 (SCL)

*SPI* контакти наведено в табл. 1.2

Таблиця 1.2 – Піни для SPI за замовчуванням:

| SPI | MOSI | MISO | CLK | CS |
|---|---|---|---|---|
| VSPI | GPIO 23 | GPIO 19 | GPIO 18 | GPIO 5 |
| HSPI | GPIO 13 | GPIO 12 | GPIO 14 | GPIO 15 |

*Переривання*

Всі GPIO можуть бути налаштовані для обробки переривань.

*Strapping pins*

- GPIO 0
- GPIO 2
- GPIO 4
- GPIO 5
- GPIO 12
- GPIO 15

Вони використовуються для переведення ESP32 у режим завантажувача або режим перепрошивки. На більшості плат розробки із вбудованим USB/Serial вам не потрібно турбуватися про стан цих контактів. Плата переводить контакти у правильний стан для перепрошивки або режиму завантаження.

Однак, якщо до цих контактів підключено периферійні пристрої, у вас можуть виникнути проблеми при спробі завантажити новий код, перепрошувати ESP32 або перезавантажити плату.

*Піни зі зміною сигналу під час завантаження*

Деякі GPIO змінюють стан на Hight або виводять ШИМ-сигнали під час завантаження або скидання. Це означає, що якщо у вас є виходи, підключені до цих GPIO, ви можете отримати несподівані результати під час перезавантаження або завантаження ESP32.

- GPIO 1
- GPIO 3
- GPIO 5
- GPIO 6 – GPIO 11 (підключені до вбудованої флеш-пам'яті SPI ESP32 – використовувати не рекомендується).
- GPIO 14
- GPIO 15

*Пін (EN)*

Enable (EN) – Цей контакт можна підключити до кнопки, наприклад, для перезапуску ESP32.

*Допустимий струм у GPIO*

Абсолютний максимальний струм, який споживається GPIO, становить 40 мА.

*ESP32 Вбудований датчик Холла*

ESP32 також має вбудований датчик Холла, який виявляє зміни у магнітному полі в його оточенні.

Arduino IDE – це зручне середовище розробки з відкритим кодом. Для початку роботи з ESP32 необхідно:

1. Додати URL до менеджера плат у налаштуваннях IDE для завантаження підтримки ESP32.
2. Вибрати модель плати (наприклад, ESP32 Dev Module).
3. Встановити драйвер для чипа USB-UART (CP2102 або CH340).
4. Вибрати COM-порт і налаштувати швидкість (зазвичай 115200 бод).

Класичний скетч Blink демонструє основи роботи з GPIO: він керує вбудованим світлодіодом, вмикаючи і вимикаючи його з інтервалом у 1 секунду. Це дозволяє перевірити коректність підключення, конфігурацію середовища та роботоздатність мікроконтролера.

## Порядок виконання роботи

***1. Встановлення Arduino IDE***

1.1. Завантаження Arduino IDE

Перейдіть на офіційний сайт Arduino:

https://www.arduino.cc/en/software

Виберіть версію, що відповідає вашій операційній системі:

- Windows Installer (рекомендується)
- macOS (zip або App Store версія)
- Linux (64-bit, ARM, Snap тощо)

1.2. Установлення Arduino IDE

Для Windows:

- Запустіть інсталятор `.exe`.
- Підтвердьте угоду (Accept License Agreement).
- Встановіть усі компоненти (включно з драйверами).
- Завершіть інсталяцію.

Для macOS:

- Відкрийте `.zip` або `.dmg` файл.
- Перетягніть Arduino IDE у папку Applications.

Для Linux:

- Завантажте `.tar.xz` архів.
- Розпакуйте його в зручне місце.
- Відкрийте термінал у папці та виконайте:

***2. Додавання підтримки ESP32***

Arduino IDE не підтримує ESP32 «з коробки», тому потрібно додати пакет вручну:

- Відкрийте Arduino IDE.
- Перейдіть до меню:

`Файл (File) → Налаштування (Preferences)`

- У полі «Додаткові URL для менеджера плат» вставте:

`https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json`

- Натисніть «ОК».

***3. Встановлення плати ESP32***

- Перейдіть до меню:

`Інструменти (Tools) → Плата (Board) → Менеджер плат (Boards Manager)`

- У полі пошуку введіть `ESP32`.
- Знайдіть **esp32**.

***4. Встановлення драйверів (за необхідності)***

Для підключення ESP32 через USB можуть знадобитися драйвери:

- CP2102: https://www.silabs.com/developers/usb-to-uart-bridge-vcp-drivers
- CH340G: https://sparks.gogo.co.nz/ch340.html

Встановіть відповідний драйвер залежно від чипа на вашій платі.

***5 Перевірка встановлення***

- Підключіть ESP32 до комп'ютера через USB.
- Виберіть порт:

`Інструменти (Tools)` → `Порт (Port)` → COMX / ttyUSBX

- Виберіть плату:

`Інструменти (Tools)` → `Плата (Board)` → `ESP32 Dev Module` (або вашу модель)

- Завантажуємо код:

```
// Замість LED_BUILTIN використовуємо конкретний пін GPIO 2
const int ledPin = 2;

void setup() {
  pinMode(ledPin, OUTPUT);  // Налаштування піна як вихід
}

void loop() {
  digitalWrite(ledPin, HIGH);  // Ввімкнути світлодіод
  delay(1000);                 // Затримка 1 секунда
  digitalWrite(ledPin, LOW);   // Вимкнути світлодіод
  delay(1000);                 // Затримка 1 секунда
}
```

Результат в Arduino IDE показано на рис. 1.3.

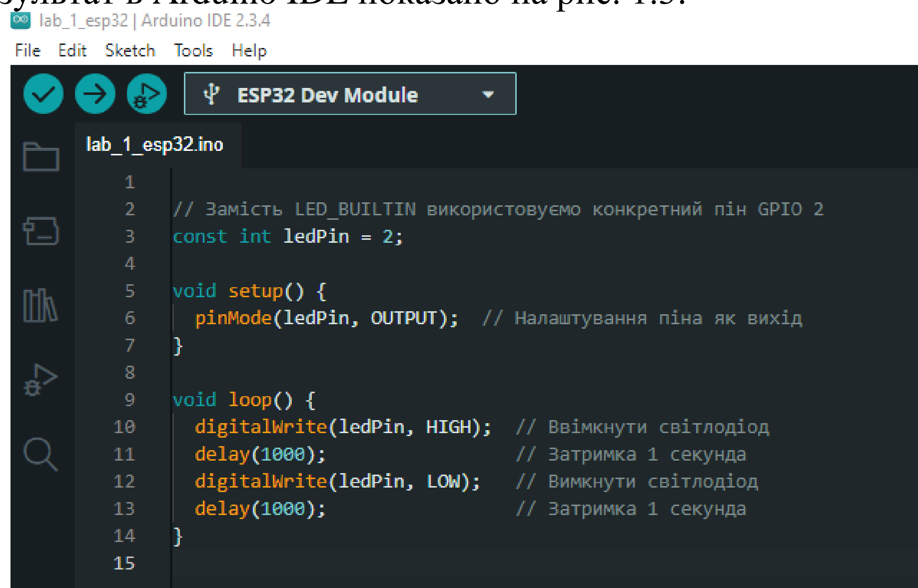



Рисунок 1.3 – Приклад коду в середовищі Arduino IDE

- Натисніть кнопку «Завантажити (Upload)».

Світлодіод має почати блимати як показано на рис. 1.4 – це підтверджує, що все встановлено правильно.

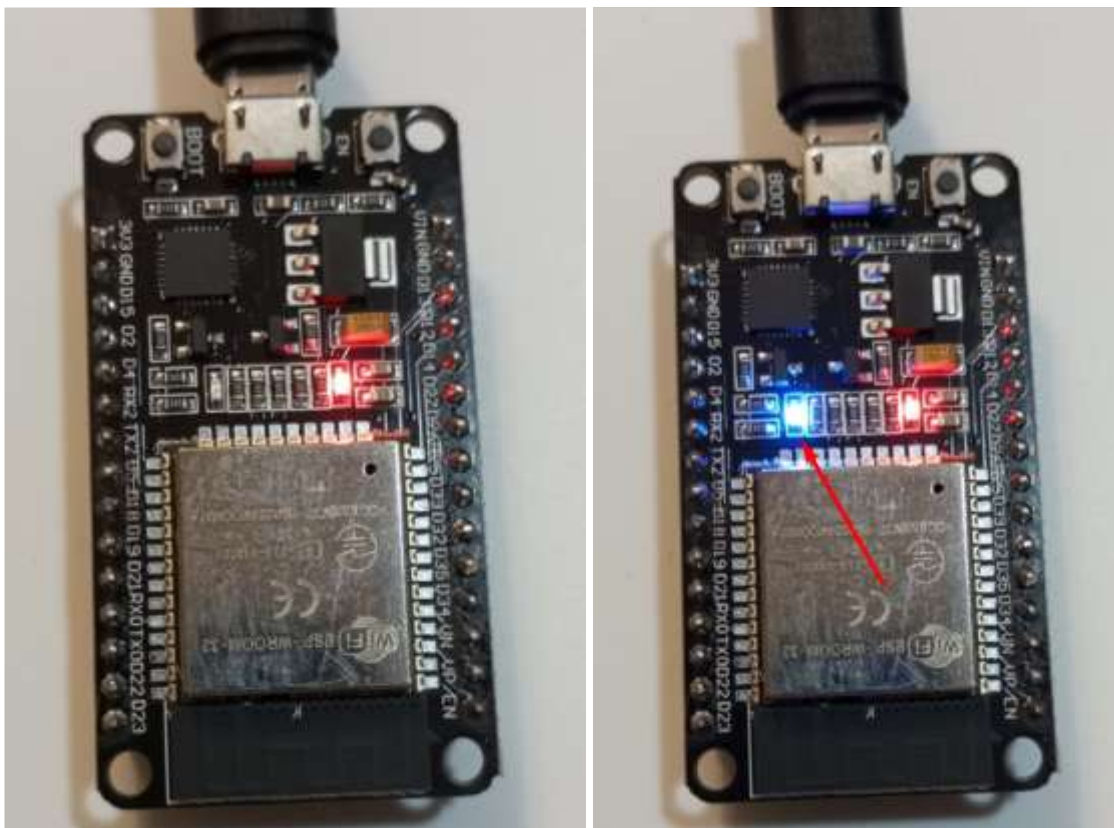
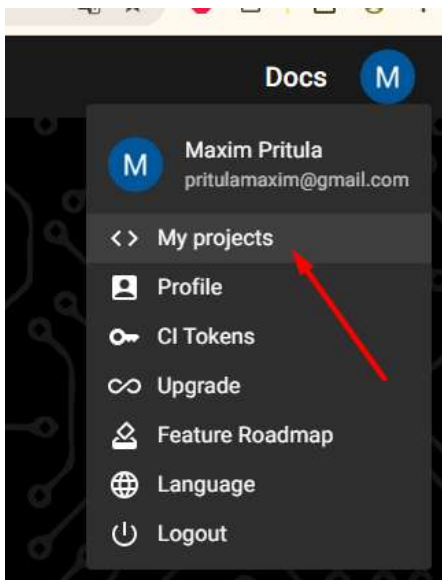


Рисунок 1.4 – Результат виконання програми на ESP32

### *6. Моделювання на платформі wokwi*

Кожен програмний пакет або платформа для моделювання ESP32 має свої обмеження. Ми для моделювання або перевірки коду будемо використовувати платформу wokwi https://wokwi.com/. Wokwi-онлайн – платформа для моделювання електроніки, яка дозволяє створювати та тестувати проєкти з мікроконтролерами прямо в браузері. Вона підтримує популярні плати, такі як Arduino Uno, Mega, Nano, ESP32, ESP8266 та Raspberry Pi Pico. Інтерфейс Wokwi інтуїтивно зрозумілий: праворуч знаходиться схема з компонентами, а ліворуч – редактор коду. Платформа дозволяє легко з'єднувати електронні компоненти, як-от світлодіоди, кнопки, дисплеї, сенсори тощо. Писати код можна мовою Arduino (C/C++) або MicroPython, залежно від вибраної плати. Wokwi підтримує емуляцію GPIO, UART, I2C, SPI, PWM та інших інтерфейсів. Завдяки симуляції в реальному часі користувач може миттєво бачити результат змін у коді. Платформа не потребує встановлення програм – усе працює в браузері.

Першим кроком на ній є реєстрація. Далі, коли ми залогінилися, необхідно створити свій проєкт, послідовність дій наведено на рис. 1.5.



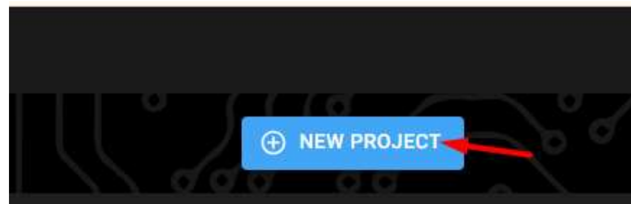



Рисунок 1.5 – Послідовність дій для створення власного проєкту на ESP32

Далі в лівому вікні вводимо наш код так, як показано на рис. 1.6. А в правому вікні складаємо нашу схему як показано на рис. 1.7.

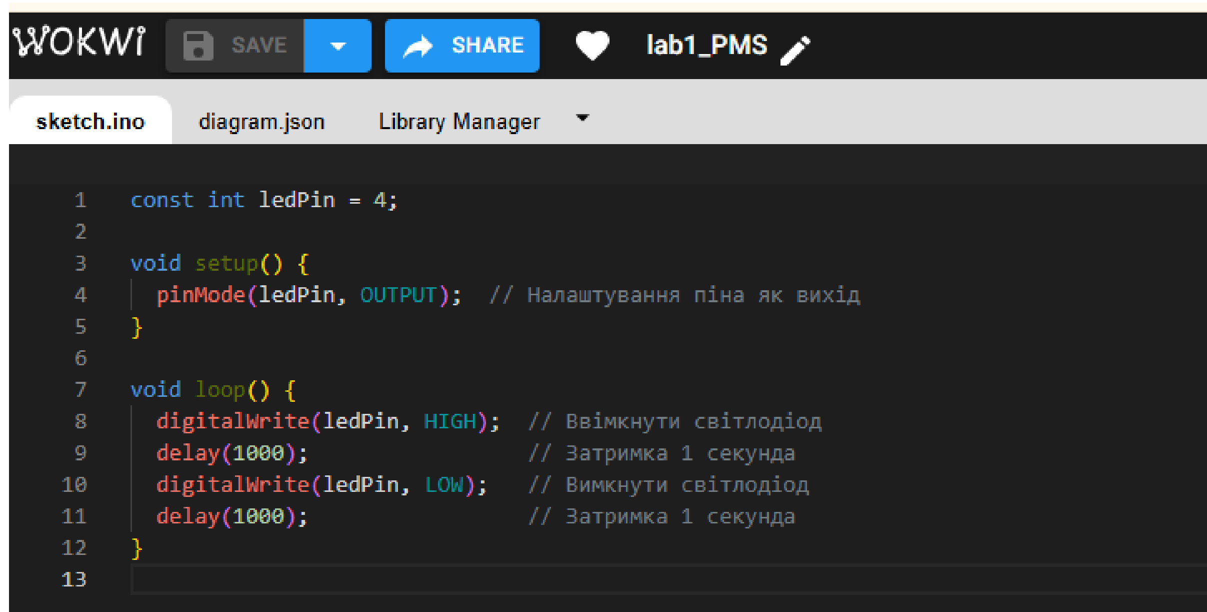



Рисунок 1.6 – Введений код на платформі wokwi для ESP32

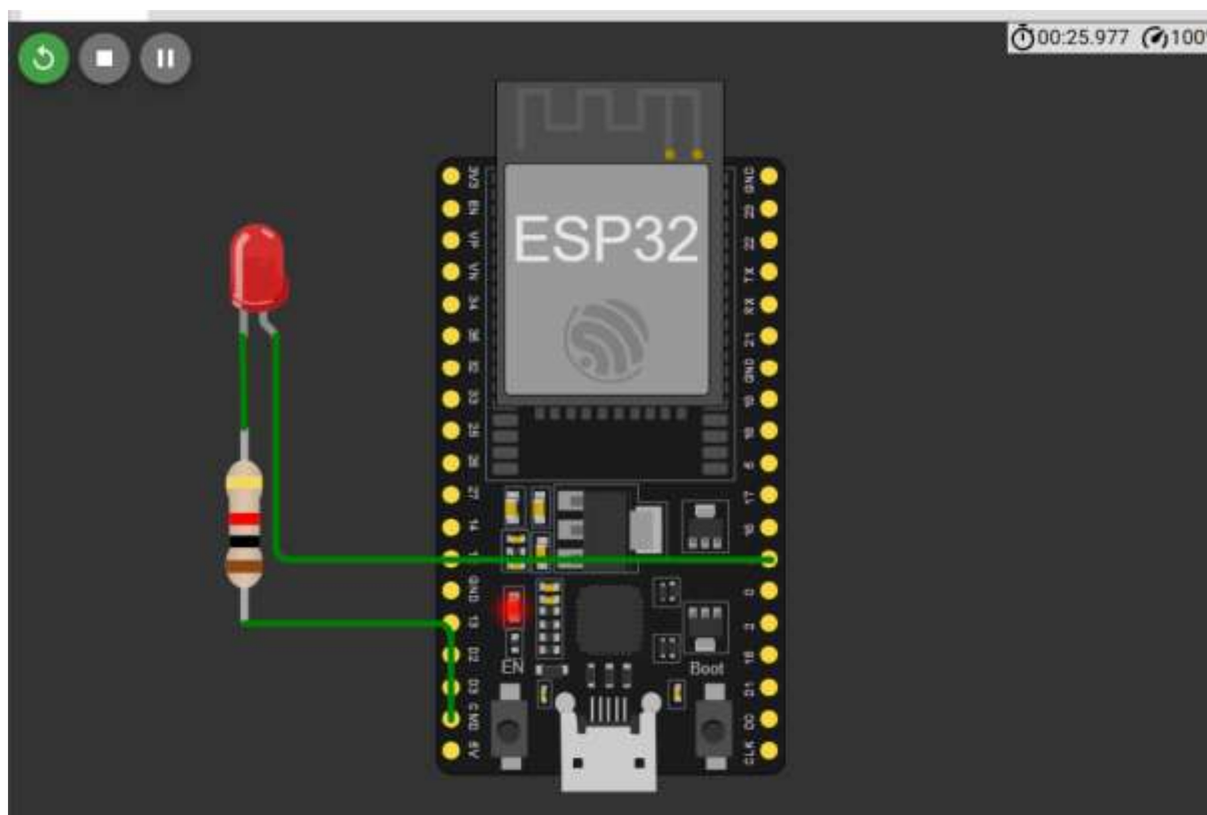
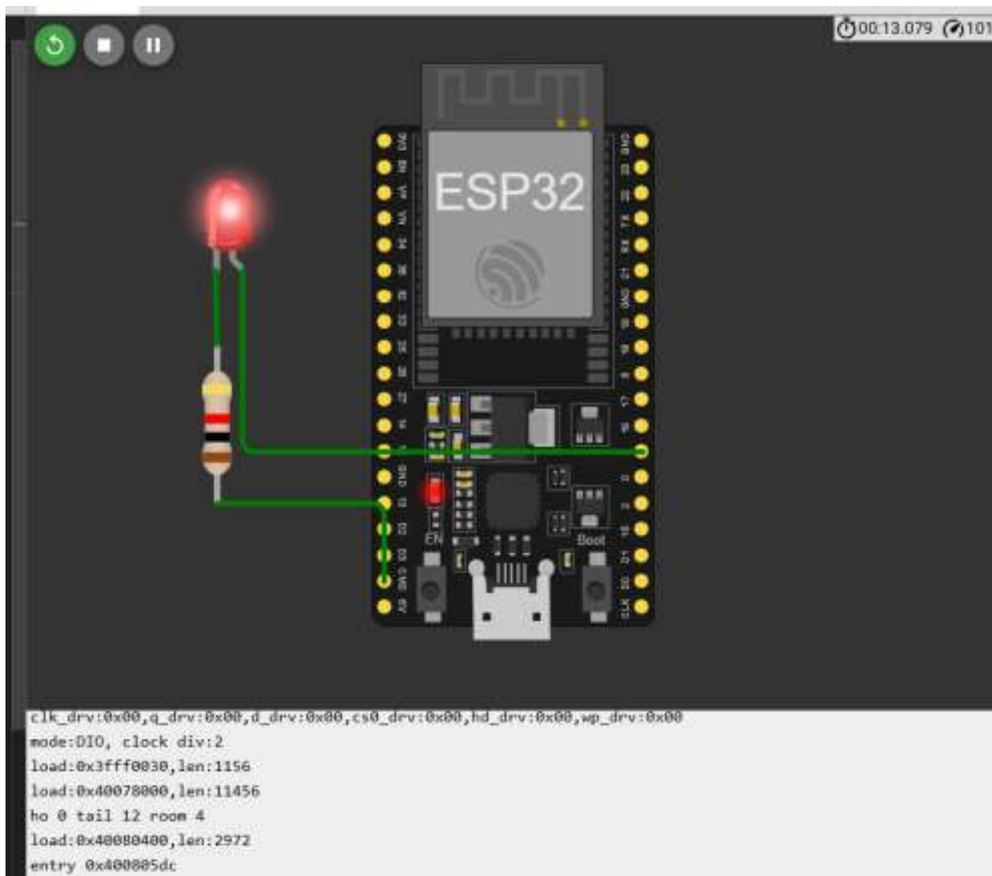



Рисунок 1.7 – Нарисована схема на платформі wokwi для ESP32

Результати дослідження показано на рис. 1.8.



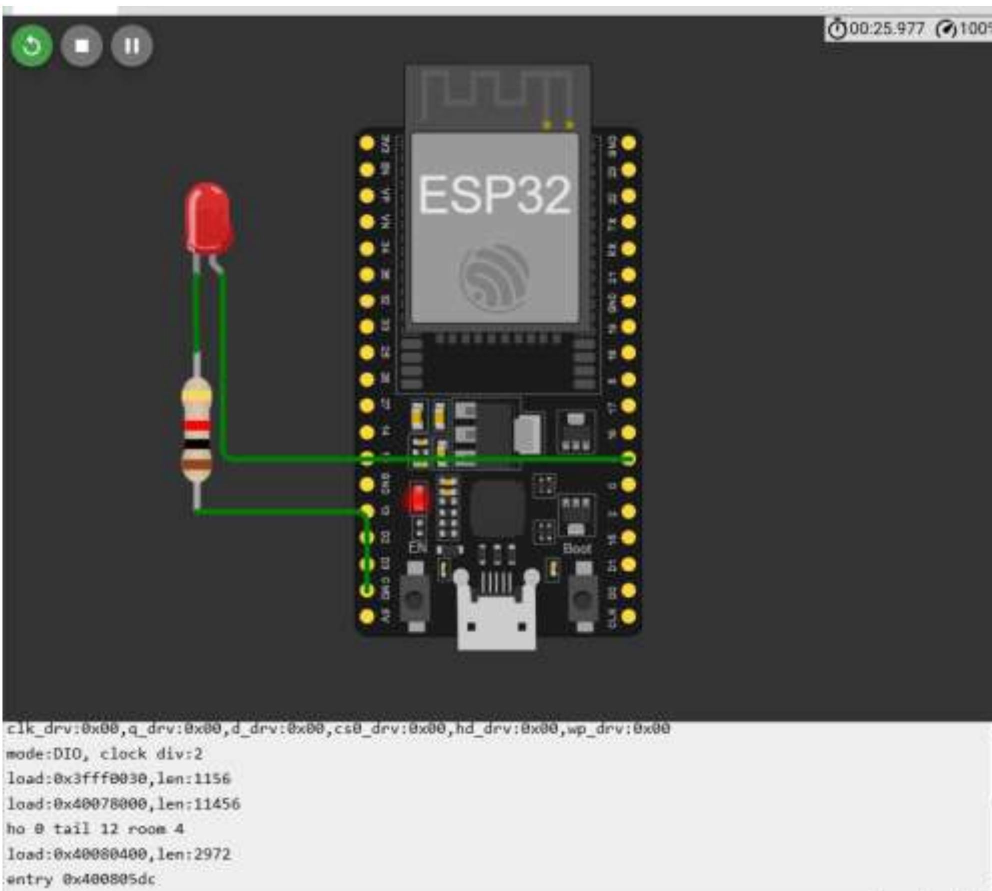




Рисунок 1.8 – Результати дослідження

Таким чином, було проведено інсталяцію програми Arduiono IDE, встановлено додаткові драйвери для модуля ESP32, написано код прошивки для нашої тестової схеми, проведено програмування та тестування модуля, перевірено його функціональність та проведено еквівалентне моделювання на платформі wokwi.

## Варіанти завдань

1. Змусити світлодіод блимати з інтервалом 1 секунда на піні GPIO2.
2. Змінити частоту блимання світлодіода на 0.5 секунди (500 мс).
3. Підключити зовнішній світлодіод до GPIO4 і реалізувати блимання.
4. Змінити пін у коді з GPIO2 на GPIO5 і перевірити роботу.
5. Організувати ввімкнення світлодіода на 2 секунди, вимкнення – на 1 секунду.
6. Додати другий світлодіод і змусити їх блимати почергово.
7. Реалізувати плавне вмикання і вимикання світлодіода з використанням analogWrite() (PWM).
8. Додати кнопку, яка вмикає світлодіод за натискання.
9. Реалізувати затримку вмикання світлодіода після натискання кнопки.
10. Виводити повідомлення в Serial Monitor під час блимання («LED ON», «LED OFF»).
11. Зробити блимання, яке змінює інтервал кожні 10 секунд.
12. Додати кнопку, яка перемикає світлодіод між ручним та автоматичним режимами.
13. Додати ще два світлодіоди, що блиматимуть із різними затримками.
14. Реалізувати логіку: 3 короткі блимання, 1 довге (як сигнал SOS).
15. Реалізувати ввімкнення LED тільки за першого натискання кнопки і вимкнення – за другого.
16. Використати millis() замість delay() для реалізації безблокувального блимання.
17. Додати повільне миготіння одного світлодіода і швидке другого.
18. Зробити миготіння залежним від змінної (наприклад, int speed = 100;).
19. Реалізувати циклічне перемикання між трьома світлодіодами (ефект «біжучого вогню»).
20. Реалізувати PWM-затемнення світлодіода кнопкою: за кожного натискання яскравість збільшується.

## Зміст звіту

Звіт має містити:
1. Завдання.
2. Обгрунтування алгоритму програми.
3. Текст коду програми.
4. Електричні схеми реалізованих рішень.
5. Висновки за результатами проведених досліджень.

## Контрольні запитання

1. Що таке ESP32 і які його основні особливості?
2. Яке призначення GPIO-пінів у ESP32?
3. Чим Arduino IDE відрізняється від ESP-IDF?
4. Що означає pinMode() в Arduino IDE?
5. Яка функція використовується для виводу логічного рівня на пін?
6. Для чого використовується функція delay()?
7. Який діапазон значень може мати пін ESP32 у разі використання з digitalWrite()?
8. Що означає HIGH та LOW в Arduino-коді?
9. Чому важливо вибирати «безпечні» GPIO під час керування виходами ESP32?
10. Яке призначення вбудованого світлодіода на ESP32?
11. До якого GPIO-піна було підключено світлодіод у вашій лабораторній роботі?
12. Чому у випаду використання GPIO2 світлодіод блимає лише один раз?
13. Який резистор був використаний у схемі і для чого він потрібен?
14. Як зміниться робота програми, якщо замість delay(1000) використати delay(200)?
15. Як змінити пін для підключення світлодіода у коді?
16. Що станеться, якщо підключити світлодіод без резистора?
17. Чи можна підключити кілька світлодіодів до різних GPIO? Яким чином?
18. Як відбувається завантаження скетчу з Arduino IDE на ESP32?
19. Як виглядає структура базового скетчу в Arduino IDE?
20. Як перевірити, що ESP32 успішно виконує ваш код?

# Лабораторна робота 2
## Робота з цифровими входами. Зчитування стану кнопки

**Мета**. Ознайомитися з принципами роботи цифрових входів ESP32 та навчитися зчитувати стан кнопки. Закріпити навички написання скетчів в Arduino IDE, обробки подій від кнопок і керування пристроями (наприклад, світлодіодом) на основі стану вхідного сигналу.

### Основні теоретичні відомості

Мікроконтролери, такі як ESP32, мають багато універсальних портів введення-виведення (GPIO – General Purpose Input/Output), які можуть бути налаштовані як входи або виходи.

***Цифрові входи (GPIO) у ESP32***

ESP32 має до 34 універсальних цифрових входів/виходів (GPIO), які можна програмно налаштовувати як:

- входи – для зчитування сигналів із кнопок, датчиків, зовнішніх мікросхем;
- виходи – для керування світлодіодами, реле, двигунами тощо.

GPIO-піни мають високу гнучкість і можуть виконувати функції введення, виведення PWM, UART, I2C, SPI, сенсорного вводу, а також бути джерелом переривань.

Цифровий вхід ESP32 дозволяє зчитувати логічний стан зовнішніх сигналів, які подаються на GPIO-пін:

- LOW (логічний 0) – напруга близько до 0 В;
- HIGH (логічна 1) – напруга близько до 3.3 В.

Вхід використовується для:

- обробки натискання кнопок і перемикачів;
- зчитування сигналів із датчиків руху, світла, магнітних герконів;
- реагування на цифрові сигнали з інших пристроїв або мікроконтролерів.

GPIO-пін налаштовується як вхід за допомогою функції:

```
pinMode(pin, INPUT); // без підтягування
pinMode(pin, INPUT_PULLUP); // із внутрішнім підтягувальним резистором
```

- INPUT – пін читає сигнал, але не має стабільного стану без зовнішнього резистора (floating).
- INPUT_PULLUP – активується внутрішній резистор (~30–50 кОм), який тримає пін у HIGH, доки він не з'єднаний із землею (GND).

ESP32 також підтримує `INPUT_PULLDOWN`, але в Arduino IDE його можуть не підтримувати всі бібліотеки. У ESP-IDF доступні всі режими.

Стан піну зчитується функцією:

```
int value = digitalRead(pin);
```

- Якщо на піні LOW (наприклад, кнопка натиснута) – повертається 0.
- Якщо HIGH (наприклад, кнопка відпущена або підтягнута) – повертається 1.

Це дозволяє легко організовувати умови:

```
if (digitalRead(buttonPin) == LOW) {
  // Кнопка натиснута
}
```

Для ESP32 важливо дотримуватись безпечного діапазону напруг на вході:

- LOW: 0 – 0.8 В
- HIGH: 2.0 – 3.3 В
- Максимум: 3.3 В (не можна подавати 5 В без дільника напруги або резисторів!)

Подавання 5 В без захисту може пошкодити GPIO.

Цифровий вхід має таку логіку:

- Цифровий буфер для зчитування логічного рівня.
- Вхідний фільтр для захисту від завад.
- Вбудовані підтягувальні резистори, які можна активувати або вимкнути програмно.

GPIO-входи ESP32 можуть використовуватися для переривань (interrupts):

```
attachInterrupt(digitalPinToInterrupt(pin),
ISR_function, CHANGE);
```

Це дозволяє реагувати миттєво на натискання кнопки без постійного опитування у `loop()`.

Не всі GPIO однаково придатні для введення, в табл. 2.1 наведено порівняння різних пінів.

Приклади застосування

- Кнопки керування: запуск функцій, зміна режимів.
- Детектори: геркони, PIR-сенсори, кінцеві вимикачі.
- Зв’язок: прийом цифрових сигналів від інших пристроїв.

Таблиця 2.1 – Порівняння різних пінів ESP32

| GPIO | Статус | Примітки |
|---|---|---|
| 6–11 | ❌ | використовуються для SPI Flash, не використовувати |
| 0, 2, 12, 15 | ⚠ | важливі під час завантаження (boot), використовуйте обережно |
| 34–39 | ✅ (вхід) | лише вхід, не підтримують `digitalWrite()` |

Помилки та рекомендації

- Не залишайте вхідний пін «плаваючим» – використовуйте `INPUT_PULLUP` або зовнішній резистор.
- Уникайте GPIO, пов'язаних з завантаженням (BOOT), особливо GPIO12 та GPIO15.
- Не подавайте напругу вище 3.3 В.
- Для стабільної роботи кнопок використовуйте затримку (debounce) або переривання.

***Кнопка як джерело сигналу***

Кнопка – це простий електромеханічний перемикач, що дозволяє створити або розімкнути електричний контакт між двома точками. Вона використовується як джерело цифрового сигналу, який мікроконтролер може зчитати через GPIO-пін у режимі входу.

У звичайному стані (коли кнопка не натиснута) контакт розімкнутий, і електричний сигнал не проходить. Під час натискання контакти замикаються, створюючи провідне з'єднання – мікроконтролер зчитує зміну логічного рівня.

Найчастіше використовується схема з підключенням одного виводу кнопки до GPIO, а другого – до землі (GND). GPIO налаштовується як `INPUT_PULLUP`, щоб внутрішній резистор підтягнув лінію до логічної 1 (HIGH). Тобто є відповідність між фізичним станом кнопки та логічним рівнем сигналу на GPIO-піні, коли використовується схема з підтягуванням до VCC (через внутрішній INPUT_PULLUP або зовнішній резистор).

Якщо кнопка відпущена (контакти розімкнуті), GPIO не з’єднаний із землею, тож завдяки підтягувальному резистору на піні присутній логічний рівень HIGH (1). Якщо кнопка натиснута (контакти замкнені), пін замикається на GND, і логічний рівень стає LOW (0).

Ця інверсія типова для схем із підтягуванням до живлення, і програмна логіка має враховувати, що натиснута кнопка = LOW, а не HIGH.

Якщо кнопка не натиснута і резистора немає, пін перебуває в «плаваючому стані» (floating) – він вловлює електричні наводки, і `digitalRead()` може повертати випадкові значення.

Підтягувальний резистор гарантує, що пін буде в стабільному стані HIGH, доки кнопка не замкне його на GND.

В ESP32 можна активувати вбудований резистор через:

```
pinMode(buttonPin, INPUT_PULLUP);
```

Під час роботи з кнопками необхідно також враховувати проблему «деренчання» (bounce), коли кнопка натискається або відпускається, контакт не замикається миттєво – протягом 5–50 мс може виникати деренчання контактів, що спричиняє кілька швидких змін стану. Це може призвести до хибного зчитування багатьох натискань замість одного.

Рішення:

- апаратна фільтрація (RC-коло);
- програмна затримка (`delay(50)`);
- логіка debounce з `millis()` або програмним фільтром.

Приклад базової схеми:

- Один контакт кнопки → GND
- Інший контакт кнопки → GPIO14
- `pinMode(14, INPUT_PULLUP)` — GPIO підтягнутий до HIGH.

В проєктах кнопки є базовим засобом введення в мікроконтролерні системи. Вони можуть керувати:

- вмиканням/вимиканням пристроїв;
- переключенням режимів роботи;
- генерацією подій або запуском функцій;
- введенням даних (у комбінації з дисплеєм або індикаторами).

***Підтягувальні резистори (Pull-up / Pull-down)***

Підтягувальні резистори – це електричні елементи, які під'єднують вхід мікроконтролера до фіксованого рівня напруги (VCC або GND) через резистор високого номіналу (зазвичай 10 кОм). Їх мета – забезпечити стабільний логічний рівень, коли вхід не з’єднаний безпосередньо з якимось джерелом сигналу.

Коли цифровий вхід мікроконтролера не підключено до певного рівня напруги, він знаходиться в так званому «плаваючому стані» (floating). У такому стані:

- значення `digitalRead()` буде невизначеним (або HIGH, або LOW);
- вхід вразливий до електромагнітних наводок;
- можуть виникати хибні спрацьовування (наприклад, під час натискання кнопки).

Підтягувальний резистор гарантує, що логічний рівень буде стабільним незалежно від того натиснута кнопка чи ні. Види підтягувальних резисторів наведено в табл. 2.2.

Таблиця 2.2 – Види підтягувальних резисторів

| Тип | Підключення | Забезпечує логічний рівень |
|---|---|---|
| Pull-up | GPIO ← резистор ← VCC | HIGH (1), коли кнопка не натиснута |
| Pull-down | GPIO ← резистор ← GND | LOW (0), коли кнопка не натиснута |

В Arduino і ESP32 найчастіше застосовується pull-up-схема, де кнопка замикається на землю, а GPIO підтягнутий до HIGH.

Рядок коду `pinMode(buttonPin, INPUT_PULLUP`:

- активує вбудований підтягувальний резистор;
- дозволяє уникнути використання зовнішнього резистора;
- дозволяє зчитувати кнопку за схемою: натиснуто = LOW, не натиснуто = HIGH.

В таблиці 2.3 наведено порівняння використання вбудованих та зовнішніх резисторів.

Таблиця 2.3 – Порівняння вбудованих і зовнішніх резисторів

| Властивість | Вбудовані резистори | Зовнішні резистори |
|---|---|---|
| Зручно використовувати | ✓ | ✗ (потрібні на схемі) |
| Типовий номінал | 30–50 кОм | 4.7–10 кОм |
| Регулювання опором | ✗ | ✓ |
| Надійність у шумних середовищах | ✗ (чутливіші) | ✓ |

Приклад схеми з зовнішнім pull-up:

```
pinMode(buttonPin, INPUT);
```

Тоді в схемі потрібно підключити резистор 10 кОм між GPIO і VCC, а кнопку — між GPIO і GND.

Особливості ESP32:

- ESP32 підтримує вбудовані pull-up і pull-down резистори;
- Активуються за допомогою:
  - `INPUT_PULLUP` — підтягує до VCC;
  - `INPUT_PULLDOWN` — (може не підтримуватись в Arduino IDE, але є в ESP-IDF).

Приклад логіки:

```
void setup() {
  pinMode(14, INPUT_PULLUP); // GPIO14
}

void loop() {
  if (digitalRead(14) == LOW) {
    // Кнопка натиснута
  } else {
    // Кнопка не натиснута
  }
}
```

Таким чином, підтягувальні резистори – важливий елемент схем вводу. Вони:

- забезпечують коректне зчитування сигналів;
- запобігають помилковим станам;
- дозволяють мікроконтролеру правильно інтерпретувати логіку зовнішніх кнопок і сигналів.

***Обробка натискання кнопки та деренчання контактів***

У цифрових схемах одна сторона кнопки зазвичай підключена до GPIO-піну мікроконтролера, інша – до землі (GND).

GPIO налаштовується як вхід з підтягуванням (INPUT_PULLUP), що дозволяє мікроконтролеру розпізнати два стани:

- HIGH (1) – кнопка не натиснута;
- LOW (0) – кнопка натиснута, пін замкнуто на землю.

Зчитування кнопки виконується через:

```
int state = digitalRead(buttonPin);
```

Проблема деренчання (англ. *contact bounce*)

Кнопка – механічний пристрій, і її контакти не замикаються миттєво. У перші мілісекунди після натискання (або відпускання) контакти можуть кілька разів швидко відкриватися і замикаються. Це називається деренчанням контактів.

Внаслідок цього:

- програма може неправильно сприймати одне натискання як кілька;
- виникають хибні спрацьовування у логіці, особливо якщо кнопка запускає події або зміни станів.

В таблиці 2.4 наведено приклад деренчання контакту. Це небажане деренчання може тривати 5–50 мс.

Таблиця 2.4 – Приклад деренчання контакту

| Час (мс) | Стан піну |
|---|---|
| 0 | HIGH |
| 2 | LOW |
| 4 | HIGH |
| 5 | LOW |
| 6 | HIGH |
| 10 | стабільно LOW (натиснута) |

Для обробки деренчання програмним шляхом наведемо два методи.

Метод 1. Просте затримування (delay)

Найпростіший спосіб – додати невелику затримку після зчитування стану кнопки:

```
if (digitalRead(buttonPin) == LOW) {
  delay(50); // Зачекати, поки деренчання вщухне
  if (digitalRead(buttonPin) == LOW) {
    // Кнопка точно натиснута
  }
}
```

Цей метод простий, але блокує виконання інших частин коду.

Метод 2. Використання `millis()` (неблокувальний debounce)

```
const int buttonPin = 14;
int lastState = HIGH;
unsigned long lastDebounceTime = 0;
const unsigned long debounceDelay = 50;

void loop() {
  int reading = digitalRead(buttonPin);

  if (reading != lastState) {
    lastDebounceTime = millis();
  }

  if ((millis() - lastDebounceTime) > debounceDelay) {
    if (reading == LOW) {
      // Стабільне натискання кнопки
    }
```

```
  }

  lastState = reading;
}
```

Такий підхід не блокує виконання програми, і підходить для складніших проєктів.

Апаратне debounce-рішення

Можна також реалізувати апаратне debounce-рішення – через RC-фільтр:

- резистор (наприклад, 10 кОм) у серії з кнопкою;
- конденсатор (0.1 мкФ) між GPIO і GND.

Це ефективно приглушує високочастотне деренчання.

Debounce необхідно враховувати завжди, коли:

- кнопка використовується для запуску функцій;
- реалізується перемикання станів (toggle);
- враховується кожне окреме натискання (наприклад, підрахунок кліків).

Таким чином:

- Деренчання – це звичайне явище в механічних кнопках, яке може спричинити логічні помилки.
- Його потрібно фільтрувати програмно або апаратно, особливо у відповідальних системах.
- Найкращі практики – це неблокувальна фільтрація через `millis()` або апаратний RC-фільтр.

***Альтернативна схема з зовнішнім резистором***

Альтернативна схема з зовнішнім підтягувальним резистором (pull-up) використовується тоді, коли з певних причин не застосовують внутрішній резистор або коли потрібно точніше контролювати значення номіналу опору.

У такій схемі кнопка з'єднується між GPIO і GND, а резистор – між GPIO і VCC (3.3 В). Це дозволяє тримати пін у стані HIGH, доки кнопка не натиснута, після чого пін замикається на землю і переходить у стан LOW. Результати функціонування схеми з зовнішнім резистором наведено в таблиці 2.5.

Таблиця 2.5 – Результати функціонування схеми з зовнішнім резистором

| Стан кнопки | Опір між GPIO і GND | Напруга на GPIO | `digitalRead()` |
|---|---|---|---|
| Відпущена | Відсутній (розімкн.) | 3.3 В (через резистор) | HIGH |
| Натиснута | Нульовий | ~0 В (замикається на землю) | LOW |

*Переваги використання зовнішнього резистора*

- Дає контроль над номіналом резистора (наприклад, можна використати 1 кОм, 4.7 кОм або 10 кОм).
- Зовнішній резистор краще працює в умовах електричних шумів або за довгих дротів.
- Сумісність із багатьма мікроконтролерами, які можуть не мати вбудованого `INPUT_PULLUP`.

*Недоліки використання зовнішнього резистора*

- Потребує додаткових компонентів на схемі.
- Займає трохи більше місця на макетній платі або друкованій платі (PCB).

Коли використовується зовнішній резистор, у коді не можна вмикати внутрішній pull-up:

```
pinMode(buttonPin, INPUT); // НЕ INPUT_PULLUP
```

Інакше обидва резистори (внутрішній і зовнішній) створять дільник напруги й можуть спричинити нестабільність.

В процесі вибору номіналу резистора використовують типові значення:

- 10 кОм – стандартний номінал;
- 4.7 кОм – для зменшення чутливості до шуму;
- 1 кОм – якщо потрібна висока швидкодія сигналу;
- 20 кОм – не рекомендовано, оскільки сигнал може бути нечітким, особливо у випадку довгих проводів.

В таблиці 2.6 наведено порівняння схем з зовнішнім та внутрішнім pull-up.

Таблиця 2.6 – Порівняння схем з зовнішнім та внутрішнім pull-up

| Параметр | Внутрішній pull-up | Зовнішній pull-up |
|---|---|---|
| Простота реалізації | ✓ | ✗ (потрібен компонент) |
| Гнучкість (вибір номіналу) | ✗ | ✓ |
| Електромагнітна стійкість | Середня | Висока |
| Зайнятість простору на платі | Немає | Потрібен резистор |
| Рекомендовано для довгих дротів | ✗ | ✓ |

Таким чином, зовнішній підтягувальний резистор – це більш контрольоване і надійне рішення, яке доцільно застосовувати у реальних проєктах, особливо за використання довгих з’єднань, в промислових умовах або під час роботи з кількома мікроконтролерами. Водночас для

освітніх задач або макетування внутрішній `INPUT_PULLUP` зазвичай достатній.

***Робота з кількома кнопками***

У реальних вбудованих системах часто потрібно працювати з двома і більше кнопками – наприклад, для вибору режиму, навігації в меню, зміни параметрів або запуску різних функцій. Мікроконтролер ESP32 дозволяє підключати до десятків цифрових входів, кожен з яких можна використовувати для окремої кнопки.

У разі підключення декількох кнопок, кожна кнопка має бути підключена до окремого GPIO-піна. Типова схема – кнопка між GPIO та GND, а пін налаштовується як `INPUT_PULLUP`.

Схема для 3 кнопок:

```
GPIO14 --- кнопка1 --- GND
GPIO27 --- кнопка2 --- GND
GPIO32 --- кнопка3 --- GND
```

У коді кожну кнопку можна зчитувати окремо:

```
pinMode(14, INPUT_PULLUP);
pinMode(27, INPUT_PULLUP);
pinMode(32, INPUT_PULLUP);
```

Зчитування кількох кнопок у коді

```
int button1 = digitalRead(14);
int button2 = digitalRead(27);
int button3 = digitalRead(32);

if (button1 == LOW) {
  // Натиснута кнопка 1
}
if (button2 == LOW) {
  // Натиснута кнопка 2
}
if (button3 == LOW) {
  // Натиснута кнопка 3
}
```

У випадку значної кількості кнопок краще використовувати масиви пінів і цикли:

```
const int buttonPins[] = {14, 27, 32};
const int numButtons = 3;

void setup() {
  for (int i = 0; i < numButtons; i++) {
    pinMode(buttonPins[i], INPUT_PULLUP);
  }
}

void loop() {
  for (int i = 0; i < numButtons; i++) {
    if (digitalRead(buttonPins[i]) == LOW) {
      Serial.print("Натиснута кнопка №");
      Serial.println(i + 1);
    }
  }
}
```

Кожна кнопка може мати своє власне деренчання-фільтрування, особливо якщо вони виконують незалежні дії. Наприклад, можна створити масиви для зберігання `lastDebounceTime[]`, `lastState[]` тощо.

В таблиці 2.7 наведено застосування кількох кнопок.

Таблиця 2.7 – Застосування кількох кнопок

| Кількість кнопок | Приклади застосування |
|---|---|
| 2 | Ввімкнення/вимкнення, ліво/право |
| 3–4 | Меню навігація: вгору, вниз, вибір, назад |
| 5+ | Ігрові пульти, числові панелі, керування пристроями |

Альтернативи: матриці кнопок

За великої кількості кнопок (більше 10) замість використання окремого GPIO для кожної кнопки, застосовують матричну клавіатуру (наприклад, 4×4). Вона дозволяє зчитувати до 16 кнопок, використовуючи лише 8 пінів (4 рядки + 4 стовпці).

*Помилки, яких потрібно уникати*

- Всі кнопки мають бути або з внутрішніми `INPUT_PULLUP`, або з зовнішніми резисторами – не змішуй способи підтягування.
- Уникай використання GPIO6–GPIO11, оскільки вони зарезервовані для SPI Flash і не призначені для введення.
- Перевір, щоб усі кнопки не використовували GPIO, які важливі для boot-завантаження (наприклад, GPIO0, GPIO2, GPIO12).

Під час взаємодії з іншими пристроями зчитування стану кнопки часто використовується для керування:

- світлодіодами;
- дисплеями;
- реле;
- сервоприводами;
- запуском таймерів, функцій, меню тощо.

Отже, робота з кількома кнопками – це базовий, але надзвичайно важливий етап у розробці інтерактивних пристроїв. Завдяки великій кількості GPIO, ESP32 дозволяє реалізовувати багатокнопкові інтерфейси без додаткових мікросхем. Важливо зважати на підтягування, обробку деренчання та структурування коду для масштабування проєкту.

***Виведення у Serial Monitor***

Serial Monitor – це вбудований інструмент у середовищі Arduino IDE, який дозволяє встановлювати двосторонній текстовий зв’язок між мікроконтролером (наприклад, ESP32) та комп’ютером через USB. Він використовується для діагностики, відлагодження, виведення значень змінних, повідомлень, а також зчитування даних від користувача.

ESP32 (як і Arduino) має апаратний UART (Universal Asynchronous Receiver-Transmitter), який дозволяє передавати й приймати дані послідовно, біт за бітом. Підключення до ПК здійснюється через USB-UART перехідник (наприклад, CP2102 або CH340).

Щоб розпочати передачу даних, потрібно ініціалізувати серійний порт у функції `setup()`:

```
Serial.begin(115200);
```

- 115200 – це швидкість обміну (baud rate), яку потрібно обов’язково виставити також у самому Serial Monitor.
- Стандартні значення: 9600, 38400, 115200 – вибери те, що підтримує твоя плата.

*Практичне застосування Serial Monitor*

- Виведення стану кнопки (натиснута/відпущена).
- Моніторинг значень з аналогових або цифрових датчиків.
- Тестування логіки роботи програми.
- Виведення поточних параметрів або повідомлень про помилки.
- Передача команд у зворотному напрямку (від користувача до ESP32 – через `Serial.read()`).

Перш ніж відкривати монітор порту, потрібно вибрати порт та швидкості використання Serial Monitor:

- вибрати правильний COM-порт (інструменти → порт);
- встановити відповідну швидкість (наприклад, 115200).

Інакше ви побачите «сміття» в консолі або не побачите нічого.

В таблиці 2.8 наведено основні функції для виведення.

Таблиця 2.8 – Основні функції Serial Monitor

| Функція | Призначення |
|---|---|
| `Serial.print("Текст")` | Виводить текст без переходу на новий рядок |
| `Serial.println("Текст")` | Виводить текст і додає перехід на новий рядок |
| `Serial.print(змінна)` | Виводить значення змінної |
| `Serial.println(число, BIN)` | Виводить число у бінарному вигляді |
| `Serial.println(число, HEX)` | Виводить число у шістнадцятковому вигляді |

Приклад використання Serial Monitor

```
void setup() {
  Serial.begin(115200);
  Serial.println("Система ініціалізована.");
}

void loop() {
  int sensorValue = analogRead(34);
  Serial.print("Значення з датчика: ");
  Serial.println(sensorValue);
  delay(1000);
}
```

Альтернатива Serial Monitor – Serial Plotter

Arduino IDE також має Serial Plotter – інструмент для графічного виведення числових значень у реальному часі. Це зручно для відображення графіків температури, освітленості, напруги тощо.

*Обмеження щодо використання Serial Monitor*

- Serial Monitor блокує UART – якщо ваша програма використовує UART для іншого пристрою (наприклад, модуля GPS), це може викликати конфлікти.
- Не можна використовувати `Serial` до виклику `Serial.begin()` – це призведе до нічого або помилки.

*Рекомендації щодо використання Serial Monitor*

- Вставляй `Serial.print()` на різних етапах коду – це допоможе «побачити», що відбувається в програмі.
- Не перевантажуй Serial великим обсягом даних без затримок – це може зупинити програму або перевантажити буфер.

• Завжди перевіряй, чи вибрана та сама швидкість у коді й у моніторі порту.

З вищенаведеної інформації витікає, що Serial Monitor – це найпростіший і найпотужніший інструмент для відлагодження в Arduino IDE і не тільки. Він дозволяє розробнику зрозуміти внутрішній стан програми, побачити значення змінних у реальному часі, відстежити натискання кнопок та ефективно контролювати виконання коду на ESP32.

## Порядок виконання роботи

1. Складаємо в wokwi схему для дослідження кнопок.

В нас є три кнопки і RGB світлодіод, один контакт якого підключений до +5 В, а інші виводи через резистор до виводів ESP. Кнопки використовують внутрішні підтягувальні резистори, і тому замикають контакт просто на землю, як показано на рис. 2.1.

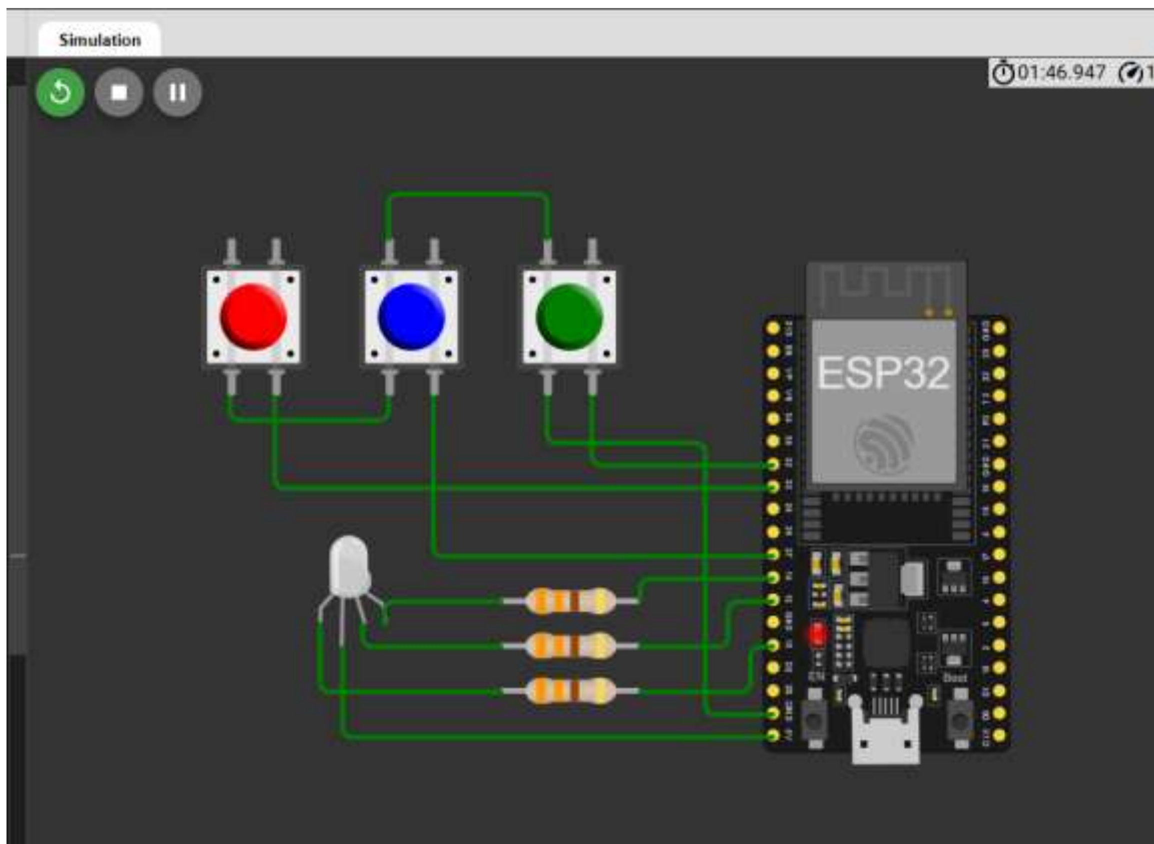



Рисунок 2.1 – Досліджувана схема кнопок в wokwi

2. Пишемо код нашої програми, яка працюватиме таким чином: за натискання кнопки певного кольору загорятиметься діод відповідного кольору. Одне натискання вмикає діод, наступне натискання вимикає діод.

```
const int buttonPins[] = {27, 32, 33};      // GPIO кнопок (B, R, G)
const int ledPins[] = {14, 12, 13};         // GPIO світлодіодів (B, R, G)
const int numButtons = sizeof(buttonPins) / sizeof(buttonPins[0]);

bool ledStates[numButtons] = {false, false, false};      // Стан кожного
світлодіода
bool lastButtonStates[numButtons] = {HIGH, HIGH, HIGH};  // Стан кнопок для
debounce

void setup() {
  Serial.begin(115200);
  for (int i = 0; i < numButtons; i++) {
    pinMode(buttonPins[i], INPUT_PULLUP);   // Кнопки з підтягуванням
    pinMode(ledPins[i], OUTPUT);            // Світлодіоди як виходи
    digitalWrite(ledPins[i], HIGH);         // Вимкнені (анод — HIGH)
  }
}

void loop() {
  for (int i = 0; i < numButtons; i++) {
    bool currentButtonState = digitalRead(buttonPins[i]);

    // Якщо кнопка перейшла з HIGH в LOW (натиснення)
    if (lastButtonStates[i] == HIGH && currentButtonState == LOW) {
      ledStates[i] = !ledStates[i];  // Перемикаємо логіку
      // Якщо світлодіод має бути ВМИКНЕНИЙ → вивід LOW (для спільного анода)
      digitalWrite(ledPins[i], ledStates[i] ? LOW : HIGH);

      Serial.print("Кнопка №");
      Serial.print(i + 1);
      Serial.print(" — світлодіод ");
      Serial.println(ledStates[i] ? "УВІМК." : "ВИМК.");
    }

    lastButtonStates[i] = currentButtonState;
  }

  delay(30);  // Простий debounce
}
```

Пояснення нашого коду.

Масиви налаштувань

```
const int buttonPins[] = {27, 32, 33};  // GPIO
кнопок
const int ledPins[] = {14, 12, 13};      // GPIO
для R, G, B каналів
```

Зберігає номери пінів ESP32, до яких підключено кнопки та світлодіоди відповідних кольорів.

Змінні станів

```
bool ledStates[numButtons] = {false, false, 
false};
bool lastButtonStates[numButtons] = {HIGH, HIGH, 
HIGH};
```

- `ledStates[]` – запам'ятовує, чи ввімкнено кожен із трьох кольорів.
- `lastButtonStates[]` – зберігає попередній стан кожної кнопки (для обробки натискання, а не утримання).

Ініціалізація в `setup()`

```
for (int i = 0; i < numButtons; i++) {
  pinMode(buttonPins[i], INPUT_PULLUP); // кнопки
  pinMode(ledPins[i], OUTPUT);          // 
світлодіоди
  digitalWrite(ledPins[i], HIGH);       // 
вимкнення (для анода HIGH = викл)
}
```

- INPUT_PULLUP: активує внутрішній підтягувальний резистор – кнопка підключена до GND.
- HIGH на світлодіоді → вимкнений (анод +, катод теж + → нема струму).

Основний цикл `loop()`

```
bool currentButtonState = 
digitalRead(buttonPins[i]);
```

Зчитуємо поточний стан кнопки:

- LOW – кнопка натиснута;
- HIGH – кнопка відпущена.

Обробка натискання (зміна HIGH → LOW)

```
if (lastButtonStates[i] == HIGH && 
currentButtonState == LOW) {
  ledStates[i] = !ledStates[i];  // Перемикаємо 
стан
```

```
        digitalWrite(ledPins[i], ledStates[i] ? LOW :
HIGH);  // LOW = вмикання
```

- Цей блок виконується тільки за натискання (не за утримання).
- `ledStates[i] = !ledStates[i];` – перемикає стан.
- Якщо `true` → виводимо LOW, щоб увімкнути світлодіод (оскільки в тебе спільний анод).
- Якщо `false` → подаємо HIGH (вимкнення).

Виведення в Serial Monitor

```
Serial.print("Кнопка №...");
Serial.println(" УВІМК. / ВИМК.");
```

Пише в монітор стан натиснутої кнопки та чи увімкнувся світлодіод.

Оновлення попереднього стану кнопки

```
lastButtonStates[i] = currentButtonState;
```

Без цього кнопка оброблялась би кожен цикл, а не тільки в момент натискання.

Debounce

```
delay(30);
```

Проста затримка для фільтрації «деренчання контактів», щоб уникнути багаторазових спрацювань від одного натискання.

3. Досліджуємо результати роботи нашого коду та схеми. У цій лабораторній роботі ми досліджуватимемо принцип роботи трьох кнопок, підключених до ESP32, для керування RGB-світлодіодом зі спільним анодом. Кожна кнопка відповідає за окремий колір: червоний, зелений або синій. За натискання кнопки відповідний світлодіодний канал перемикається між станами вмикання та вимикання. Ми спостерігатимемо зміни кольору світлодіода, що виникають внаслідок комбінацій натискань. Для індикації стану кожного каналу використовується серійний монітор. Таким чином, ми отримаємо уявлення про роботу цифрових входів та логіки перемикачів у мікроконтролерних системах. Результати наведено на рис. 2.2 – 2.5.

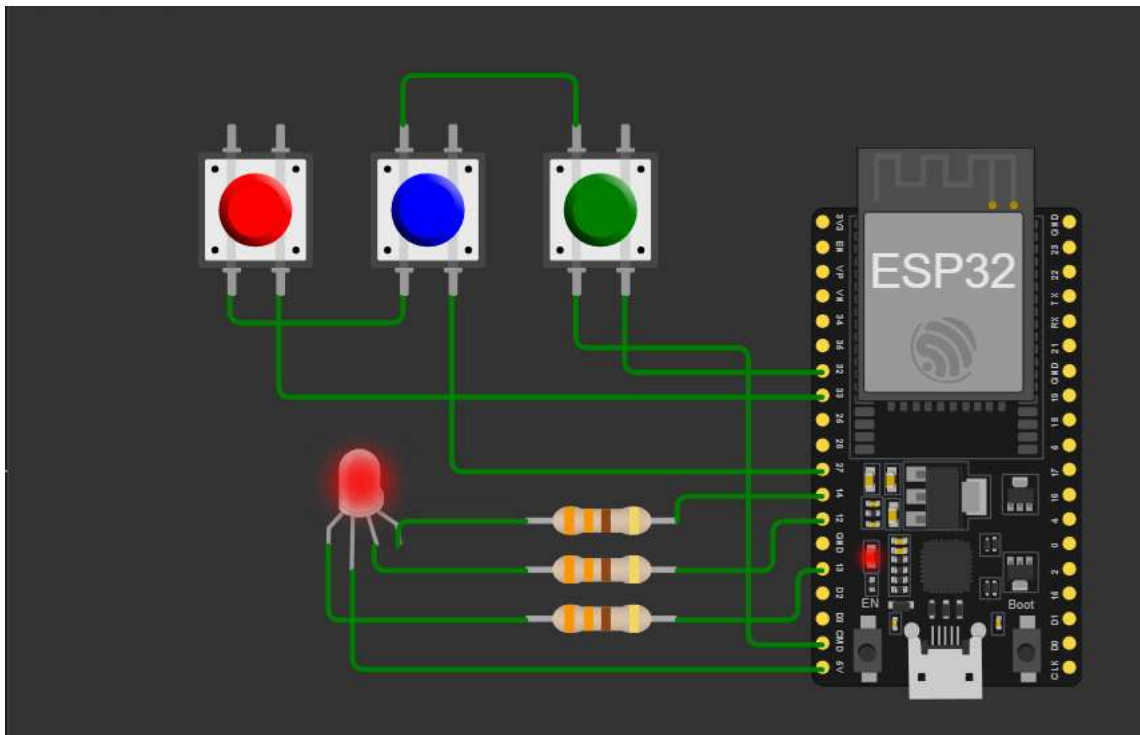



Рисунок 2.2 – Результат моделювання за натискання червоної кнопки

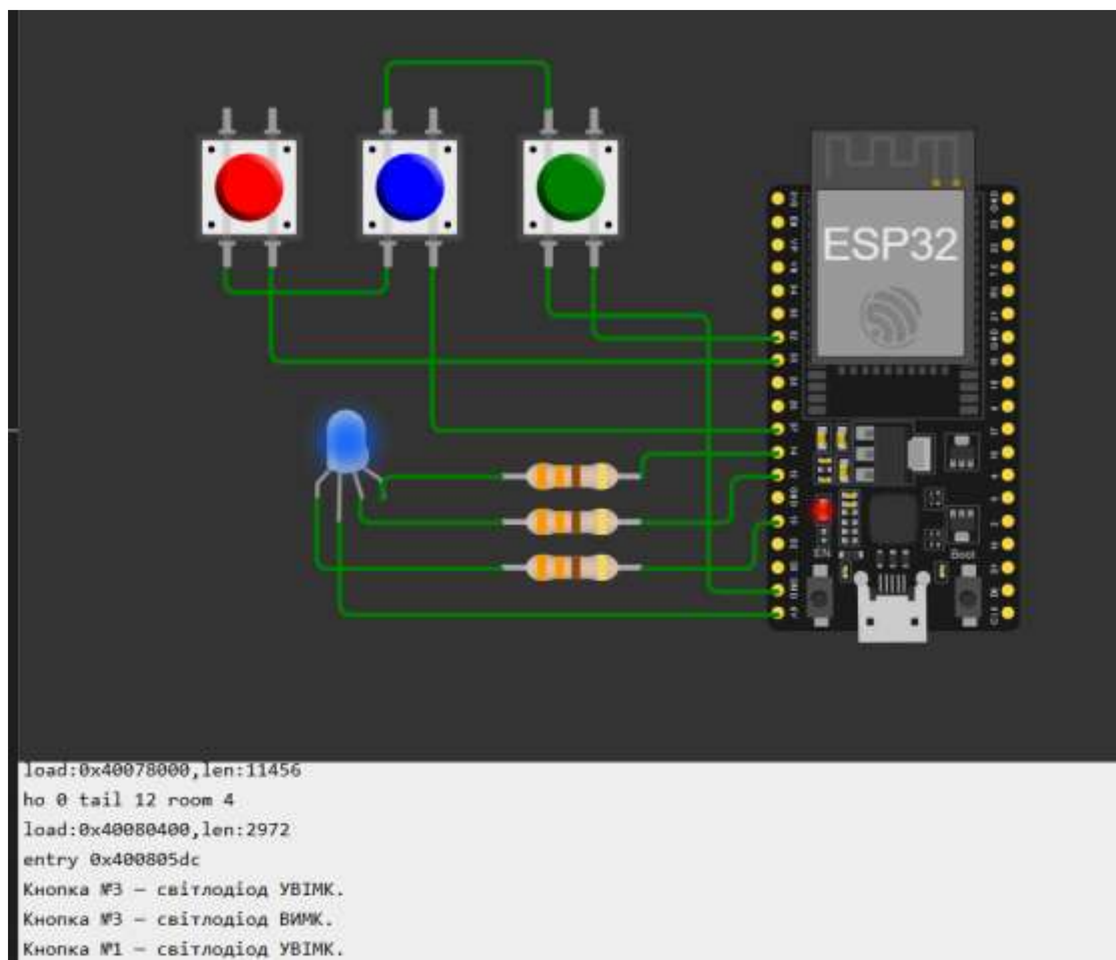



Рисунок 2.3 – Результат моделювання за натискання синьої кнопки

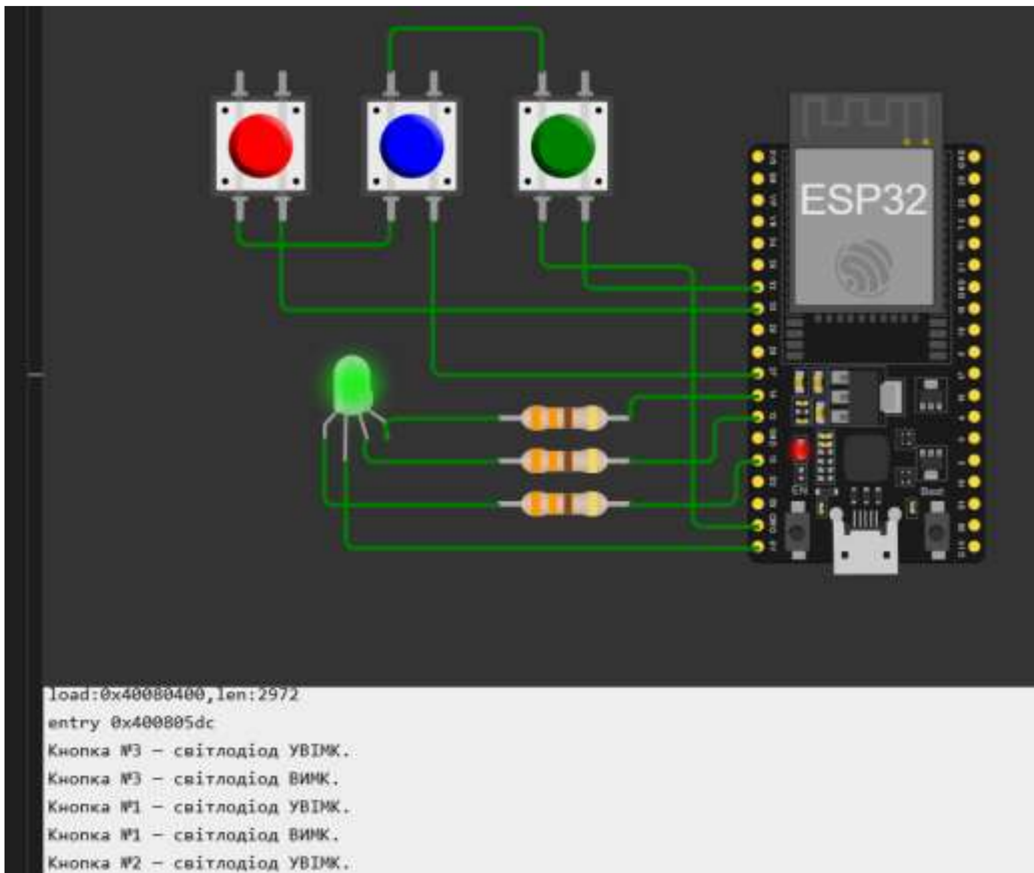



Рисунок 2.4 – Результат моделювання за натискання зеленої кнопки

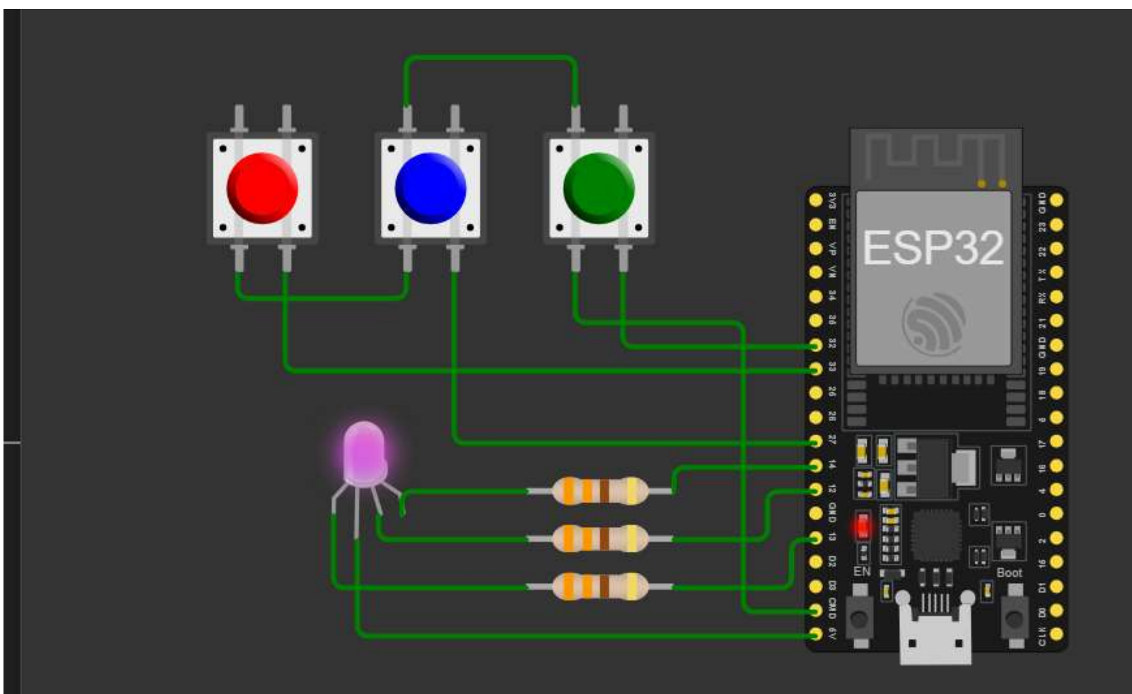



Рисунок 2.5 – Результат моделювання за натискання червоної та синьої кнопок

3. Складаємо аналогічну схему на макетній платі з ESP32 (рис. 2.6) та проводитимемо її дослідження (рис. 2.7 – 2.9).

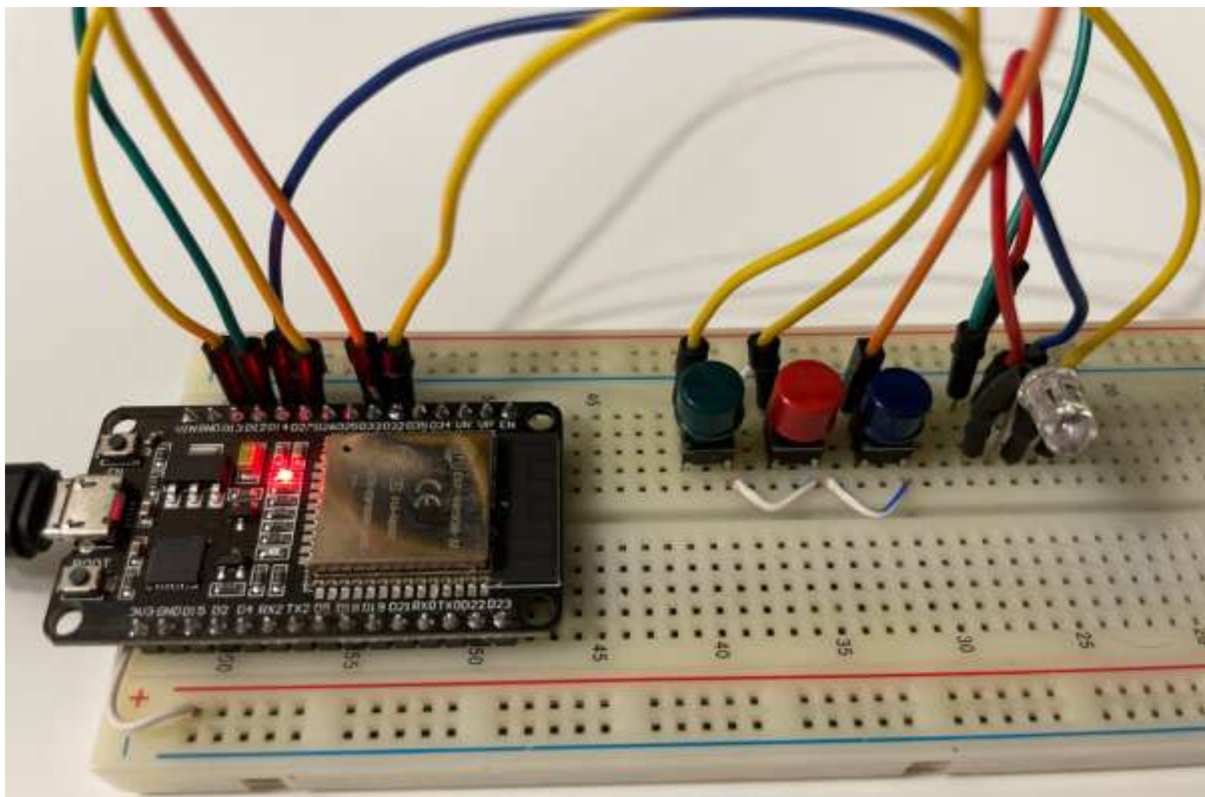


Рисунок 2.6 – Досліджувана схема, складена на макетній платі

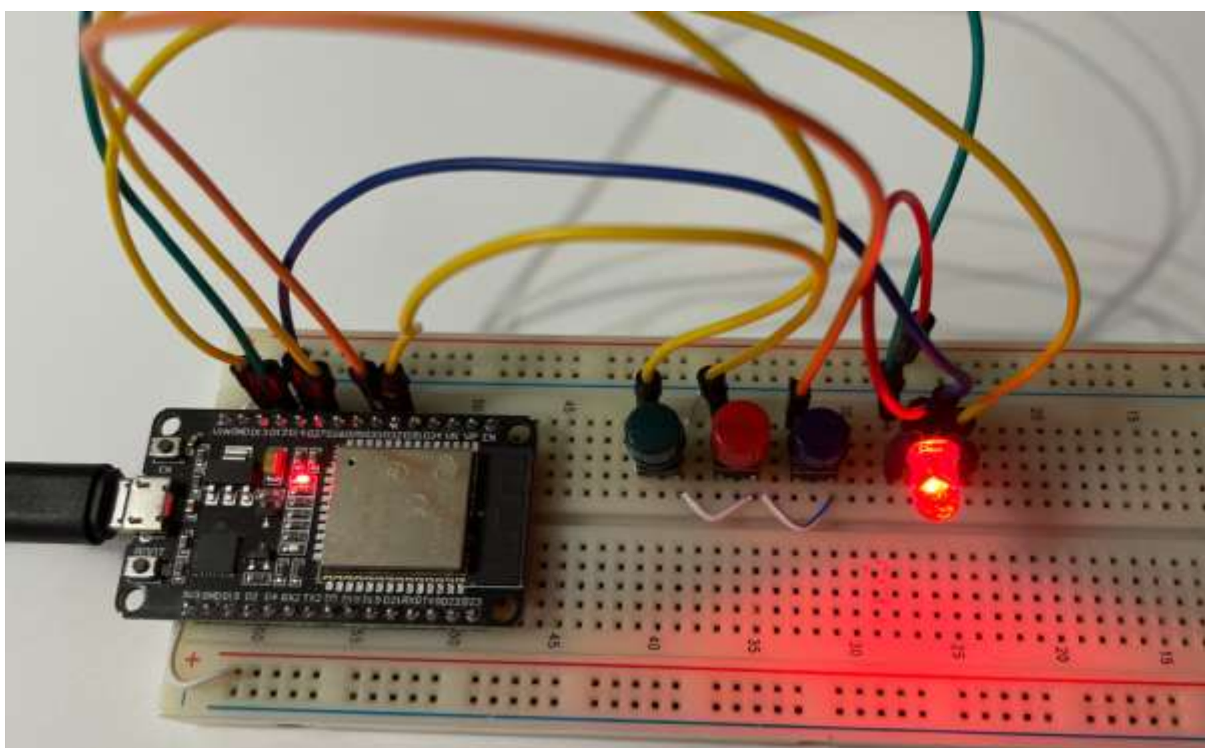


Рисунок 2.7 – Результат дослідження макета за натискання червоної кнопки

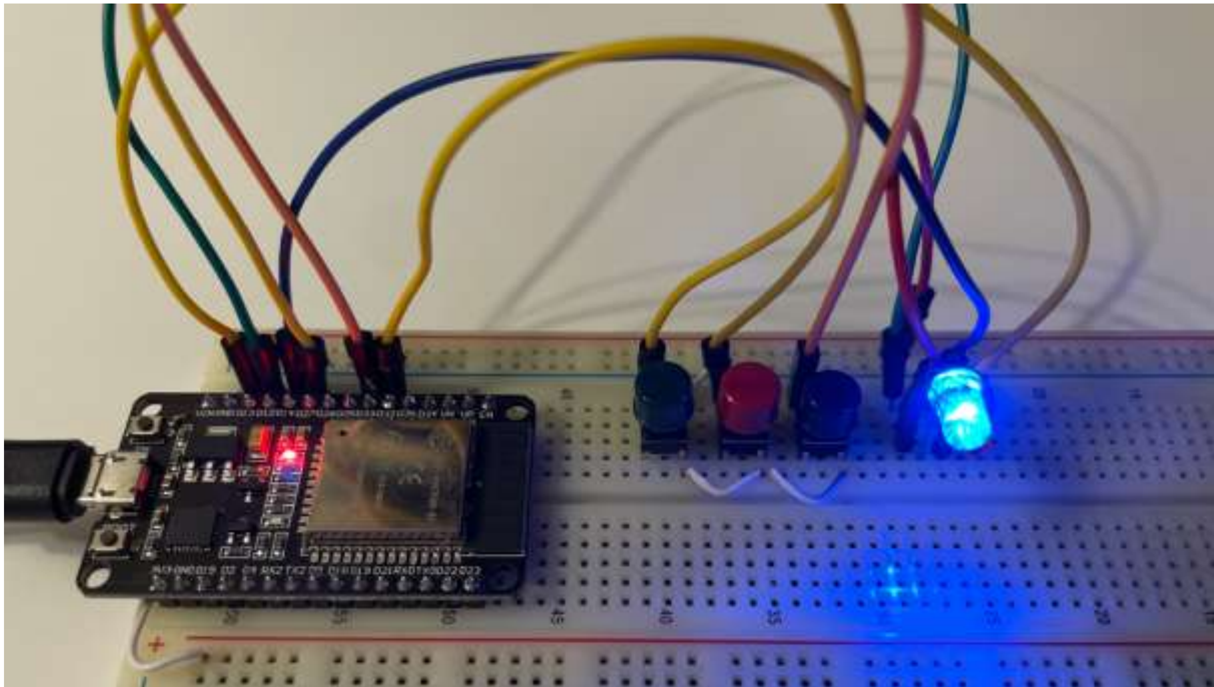


Рисунок 2.8 – Результат дослідження макета
за натискання синьої кнопки

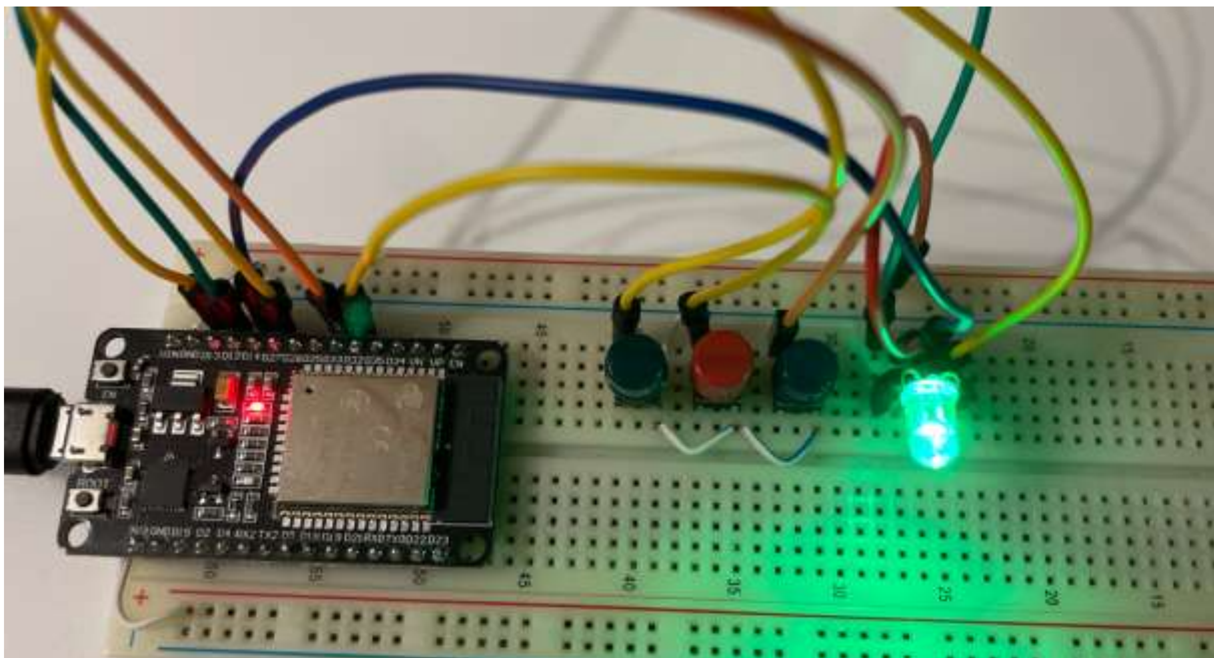


Рисунок 2.9 – Результат дослідження макета
за натискання зеленої кнопки

4. Таким чином, внаслідок виконання цієї роботи було успішно реалізовано повноцінну систему керування RGB-світлодіодом за допомогою трьох незалежних кнопок, підключених до мікроконтролера ESP32. Проєкт охоплює як апаратну, так і програмну складові, що

дозволяє глибше зрозуміти принципи роботи цифрових входів/виходів, обробки сигналів користувача, а також ефективної взаємодії з виконавчими елементами (світлодіодами).

Особливу увагу приділено оптимізації коду: використання масивів і циклів дозволило зменшити дублювання, підвищити масштабованість і покращити читабельність програми. Застосування програмної обробки деренчання контактів (debounce) забезпечило стабільну та коректну реакцію системи на натискання кнопок, що є важливою практикою під час роботи з фізичними перемикачами.

Застосована схема з RGB-світлодіодом спільного анода дала змогу закріпити розуміння полярності LED та логіки керування: активний стан досягається подачею LOW на вивід мікроконтролера. Це стало гарним прикладом практичного використання відкритих колекторів або логіки з інверсією.

Отже, виконана робота є якісною основою для створення інтерфейсів користувача у вбудованих системах та інтелектуального освітлення.

## Варіанти завдань

1. Зчитати стан однієї кнопки та виводити «Натиснута»/«Відпущена» у Serial Monitor.
2. Реалізувати схему: за натискання кнопки засвічується один світлодіод.
3. Увімкнути світлодіод у разі натиснутої кнопки та вимкнути – у разі відпущеної.
4. Вмикати світлодіод на 1 секунду за кожного натискання кнопки (не утримання).
5. Реалізувати миготіння світлодіода, яке вмикається кнопкою № 1 і вимикається кнопкою № 2.
6. Реалізувати підрахунок кількості натискань кнопки з виведенням у Serial Monitor.
7. Реалізувати логіку «Toggle» – кожне натискання перемикає стан світлодіода.
8. Підключити дві кнопки: одна вмикає світло, інша – вимикає.
9. Реалізувати затримку вимкнення: у разі натискання світлодіод світиться ще 3 секунди.
10. Реалізувати схему, де утримання кнопки вмикає миготіння, а відпускання – вимикає.
11. Підключити кнопку та керувати яскравістю світлодіода за принципом: кожне натискання – збільшення ШІМ.
12. Реалізувати затримку реагування на кнопку (не реагувати, поки кнопка не натиснута довше 1 с).
13. Створити логіку, за якої світлодіод засвічується тільки після трьох натискань поспіль.

14. Реалізувати схему: поки кнопка утримується – світлодіод блимає кожні 0.5 секунди.

15. Створити просту охоронну логіку: якщо кнопку не натиснули протягом більше 10 с – вмикається сигналізація (світлодіод).

16. Створити логіку блокування кнопки: після одного натискання вона ігнорується протягом 5 секунд.

17. Вивести повідомлення «Довге натискання», якщо кнопку тримають довше 2 секунд.

18. Підключити дві кнопки і реалізувати логіку AND (світлодіод вмикається, тільки коли натиснуті обидві).

19. Змінювати колір RGB-світлодіода залежно від того, яку кнопку натиснуто (3 кнопки, 3 кольори).

20. Реалізувати лічильник: натискаючи кнопку – додається +1, і число виводиться у Serial Monitor.

## Зміст звіту

Звіт має містити:

1. Завдання.
2. Обгрунтування алгоритму програми.
3. Текст коду програми.
4. Електричні схеми реалізованих рішень (фото, скріни і под.).
5. Висновки за результатами проведених досліджень.

## Контрольні запитання

1. Що таке цифровий вхід у мікроконтролері?
2. Яке логічне значення (HIGH чи LOW) зчитується на піні, коли кнопка підключена до GND і не натиснута під час використання INPUT_PULLUP?
3. Яке призначення внутрішніх підтягувальних резисторів (INPUT_PULLUP)?
4. Які наслідки може спричинити відсутність підтягувального резистора на вході?
5. Як визначити, що кнопка була натиснута, у коді на Arduino?
6. Що таке «деренчання контактів» і як воно проявляється в роботі програми?
7. Назвіть один простий спосіб боротьби з деренчанням кнопки у коді.
8. Як працює команда digitalRead(pin)?
9. Яка різниця між INPUT і INPUT_PULLUP в pinMode()?
10. Для чого використовується delay() у програмі з кнопкою?
11. Чому не рекомендується використовувати лише delay() для debounce у складних проєктах?

12. Як реалізувати реагування лише на момент натискання, а не на утримання кнопки?

13. Яке логічне значення буде на піні, коли кнопка натиснута за схеми з INPUT_PULLUP?

14. Чим відрізняється активний високий рівень кнопки від активного низького?

15. Як можна реалізувати перемикач стану світлодіода за допомогою однієї кнопки?

16. Чому корисно зберігати попередній стан кнопки у змінній?

17. Як можна реалізувати обробку довгого натискання кнопки?

18. Як підключити кілька кнопок до ESP32 без використання зовнішніх резисторів?

19. Які GPIO ESP32 не рекомендується використовувати для кнопок? (пояснити чому)

20. Для чого використовується Serial Monitor під час налагодження кнопок?

# Лабораторна робота 3
## Робота з аналоговими сигналами

**Мета**. Ознайомитися з роботою аналогових входів (ADC) на ESP32. Навчитися зчитувати аналогові значення з датчиків або потенціометра. Зрозуміти принцип роботи широтно-імпульсної модуляції (PWM) та її генерації. Реалізувати зв'язок між аналоговим входом і PWM-виходом (керування яскравістю LED або швидкістю двигуна).

### Основні теоретичні відомості

*Поняття аналогового сигналу та його цифрової інтерпретації*

Аналоговий сигнал – це фізичний сигнал, який змінюється безперервно в часі та може набувати нескінченного числа значень у певному діапазоні. Найпоширенішими прикладами аналогових сигналів є температура повітря, освітленість, сила струму, звукові хвилі, положення об'єктів, тиск, вологість, швидкість вітру тощо. У технічних системах аналоговий сигнал зазвичай подається як напруга, яка змінюється плавно в межах певного діапазону, наприклад від 0 до 3.3 В.

На відміну від цифрових сигналів, які можуть набувати лише дискретних значень (наприклад, «1» або «0»), аналогові сигнали є безперервними та мають необмежену кількість можливих рівнів. Ця властивість робить їх надзвичайно корисними для подання реальних фізичних величин. Проте, через особливості сучасної цифрової електроніки, такі сигнали часто потрібно перетворити у цифрову форму для обробки мікроконтролерами.

***Переваги та недоліки аналогових сигналів***

*Переваги*

- Висока точність відображення природних процесів.
- Відсутність потреби в перетворенні у випадку роботи з аналоговими пристроями.
- Простота формування сигналу через аналогові сенсори (фоторезистори, терморезистори тощо).

*Недоліки*

- Піддаються впливу шумів та завад.
- Складність точного вимірювання без спеціалізованої техніки.
- Незручність у зберіганні, передаванні та обробці без перетворення в цифрову форму.

Щоб мікроконтролер міг опрацювати аналоговий сигнал, його потрібно перетворити у цифровий. Це завдання виконує аналого-цифровий перетворювач (ADC – Analog-to-Digital Converter). ADC вимірює значення вхідної напруги та подає його як число, яке відповідає пропорції цієї напруги до максимально можливої (еталонної).

Наприклад, якщо ADC має 10-бітну розрядність, він видає значення в діапазоні від 0 до 1023. Якщо вхідна напруга становить половину від максимальної (наприклад, 1.65 В за 3.3 В максимальних), ADC поверне значення, близьке до 512.

Розрядність ADC (bit depth) – це кількість бітів, яку використовує перетворювач для подання значення напруги. Вища розрядність означає більшу точність вимірювання. Наприклад:

- 8 біт – 256 рівнів (0–255);
- 10 біт – 1024 рівні (0–1023);
- 12 біт – 4096 рівнів (0–4095);
- 16 біт – 65536 рівнів (0–65535).

Формула для обчислення реальної напруги за значенням ADC:

$$V = (ADC/(2^n - 1)) * V_{ref}$$

де $n$ – кількість бітів розрядності,

$V_{ref}$ – опорна напруга,

ADC – цифрове значення.

Частота вибірки визначає, як часто ADC зчитує аналоговий сигнал. Для більшості застосувань у мікроконтролерах достатньо частоти 1–10 тис. вимірювань на секунду. Важливо враховувати теорему Найквіста: для точного відображення сигналу потрібно вибірку здійснювати з частотою не менше ніж удвічі вищою за максимальну частоту змін сигналу.

*Приклади аналогових сенсорів*

- Потенціометр – дозволяє змінювати напругу вручну.
- Фоторезистор – змінює опір залежно від освітлення.
- Терморезистор – реагує на зміну температури.
- Мікрофон – перетворює звук на аналоговий сигнал.
- Датчики тиску, глибини, вологості та інші фізичні сенсори.

Мікроконтролери, зокрема ESP32, обробляють лише цифрову інформацію. Їх обчислювальні ядра, пам'ять і логіка працюють з дискретними значеннями. Для того щоб ці пристрої могли реагувати на реальні процеси, аналогові дані потрібно «перекласти» у цифрову форму, зрозумілу мікропроцесору.

Оскільки аналогові сигнали чутливі до завад, у схемах часто використовуються RC-фільтри, екранування, конденсатори згладжування. Програмно дані можна фільтрувати за допомогою ковзного середнього (moving average), експоненціального згладжування або медіанних фільтрів. Це дозволяє зменшити похибки під час зчитування даних та зробити керування більш стабільним.

У лабораторній роботі аналоговий сигнал зчитується з потенціометра. Його положення впливає на значення напруги, яке надходить на ADC мікроконтролера ESP32. Після оцифрування отримане

значення використовується для управління широтно-імпульсною модуляцією (PWM), яка змінює яскравість світлодіода. Таким чином, обертаючи потенціометр, ми можемо керувати світловим потоком LED у режимі реального часу.

Це яскраво демонструє як аналогові сигнали перетворюються, аналізуються і керують фізичними процесами через цифрову електроніку.

Таким чином, поняття аналогового сигналу та його цифрової інтерпретації є ключовими для розуміння принципів взаємодії мікроконтролерів з навколишнім світом. Аналогові сигнали дозволяють зчитувати природні явища, а цифрова обробка – керувати пристроями точно, ефективно та програмно. Саме завдяки цьому симбіозу сучасна автоматика, робототехніка та IoT стали настільки гнучкими й потужними.

*Аналогові входи ESP32 (ADC)*

Мікроконтролер ESP32 містить два аналогово-цифрові перетворювачі (ADC), що мають 12-бітну розрядність. Це дозволяє зчитувати аналогові значення з точністю від 0 до 4095 (тобто $2^{12} = 4096$ рівнів).

ESP32 має два незалежних ADC-модулі:

- ADC1 – обслуговує входи GPIO 32–39;
- ADC2 – обслуговує входи GPIO 0, 2, 4, 12–15, 25–27.

В документації зазначено, що ADC2 не рекомендується використовувати одночасно з Wi-Fi, оскільки обидві підсистеми використовують спільні ресурси. У практичних задачах зазвичай застосовують ADC1.

*Характеристики ADC ESP32*

- Розрядність: до 12 біт.
- Діапазон напруги: зазвичай 0–3.3 В (може змінюватися залежно від конфігурації).
- Кількість каналів: до 18 (у різних варіантах ESP32).
- Підтримка калібрування: можливе калібрування за температурою, напругою живлення тощо.
- Програмна підтримка: Arduino IDE, ESP-IDF, бібліотеки для Python/MicroPython.

ADC ESP32 не має вбудованого згладжування чи фільтрації, тому для якісного зчитування аналогових сигналів потрібно:

- додавати фільтри низьких частот на вході (резистор і конденсатор);
- усереднювати показники програмно (медіанне, середнє або експоненційне згладжування);
- забезпечувати стабільне джерело живлення (мінімізувати шум у ланцюгах живлення);
- використовувати внутрішню бібліотеку `esp_adc_cal` (для ESP-IDF) або Arduino-аналог - `analogReadResolution()`.

Рекомендується використовувати GPIO з ADC1: 32–39. GPIO36 (VP) і GPIO39 (VN) особливо зручні, оскільки мають стабільні характеристики та фізично позначені на багатьох платах DevKit.

У середовищі Arduino IDE зчитування аналогового значення виконується просто:

```
int value = analogRead(34);   // Зчитування з GPIO34 (ADC1 канал)
```

Повертає значення в діапазоні від 0 до 4095. Значення залежить від вхідної напруги.

За замовчуванням ADC працює з 12-бітною розрядністю. Можна змінити розрядність за допомогою:

```
analogReadResolution(12);   // Встановлення 12-бітної розрядності
```

Зверніть увагу, що Arduino IDE для ESP32 обмежує API для розширених налаштувань. Повна конфігурація доступна в ESP-IDF через `adc1_config_width()`.

У ESP32 можна змінювати коефіцієнт ослаблення вхідного сигналу (attenuation) для розширення діапазону вимірювання:

- `ADC_0db` – до 1.1 В;
- `ADC_2_5db` – до 1.5 В;
- `ADC_6db` – до 2.2 В;
- `ADC_11db` – до 3.3 В (типово).

```
analogSetAttenuation(ADC_11db);
```

Це дозволяє підключати датчики з більш високою напругою без перевантаження ADC.

ESP32 підтримує калібрування ADC за напругою та температурою. У ESP-IDF це реалізується через `esp_adc_cal_characterize()` – вона дозволяє отримати точне значення вольт, а не просто «сирі» дані.

Приклад 1: Зчитування значення з потенціометра

```
const int potPin = 34;  // підключено до ADC1
int analogValue = 0;

void setup() {
  Serial.begin(115200);
}
```

```
void loop() {
  analogValue = analogRead(potPin);
  Serial.println(analogValue);
  delay(200);
}
```

Приклад 2: Зчитування напруги та виведення у вольтах

```
const int sensorPin = 35;

void setup() {
  Serial.begin(115200);
}

void loop() {
  int raw = analogRead(sensorPin);
  float voltage = raw * (3.3 / 4095.0);
  Serial.print("Voltage: ");
  Serial.println(voltage);
  delay(500);
}
```

Усереднення значень

```
const int analogPin = 36;

int averageAnalog(int pin, int samples) {
  long total = 0;
  for (int i = 0; i < samples; i++) {
    total += analogRead(pin);
    delay(5);
  }
  return total / samples;
}

void setup() {
  Serial.begin(115200);
}

void loop() {
  int value = averageAnalog(analogPin, 20);
  Serial.println(value);
  delay(300);
}
```

Перетворення в PWM (комбінований приклад)

```
const int analogPin = 34;
const int ledPin = 18;

void setup() {
  ledcSetup(0, 5000, 8);      // канал 0, частота 5 кГц, розрядність 8 біт
  ledcAttachPin(ledPin, 0);   // прив'язка GPIO18 до каналу 0
}

void loop() {
  int val = analogRead(analogPin);
  int pwm = map(val, 0, 4095, 0, 255);
  ledcWrite(0, pwm);
  delay(100);
}
```

Додаткові можливості:

- `analogSetAttenuation()` – зміна ослаблення для розширення діапазону;
- `analogSetWidth()` – встановлення розрядності (ESP32 SDK) ;
- можливість використання DMA та таймерів;
- програмна обробка шумів і дрейфу значень;
- підтримка апаратних тригерів (в ESP-IDF).

*Типові помилки*

- Зчитування з ADC2 за активного Wi-Fi → дає `0` або некоректні значення.
- GPIO12 не можна тримати високим під час старту – він впливає на режим завантаження.
- Недостатня фільтрація сигналу → «стрибки» значень.
- Підключення джерела сигналу з вищою напругою, ніж 3.3 В – може пошкодити ADC.

*Застосування*

- Зчитування показників з аналогових сенсорів (температури, вологості, тиску).
- Побудова аналогових інтерфейсів (потенціометри, джойстики).
- Аналіз вхідних сигналів у часовій області.
- Вимірювання напруги живлення та споживання.
- Сенсорні панелі з аналоговими датчиками.

Ця інформація та її розуміння є фундаментом для розуміння аналогових можливостей ESP32.

*Широтно-імпульсна модуляція (PWM) на ESP32*

PWM (Pulse Width Modulation) – це метод цифрової модуляції, за якого передається інформація, змінюючи співвідношення тривалості високого сигналу до повного періоду. Цей метод дозволяє ефективно керувати аналоговими навантаженнями за допомогою цифрових виходів. У практиці застосовується для керування яскравістю світлодіодів, потужністю двигунів, температурою у нагрівачах, а також для цифро-аналогового перетворення.

PWM забезпечує гнучкий контроль за переданою енергією, і його імпульсна природа дозволяє високоефективне перемикання у силових схемах. У багатьох сучасних мікроконтролерах (зокрема в ESP32) цей процес апаратно підтримується і може бути налаштований з високою точністю (рис. 3.1).

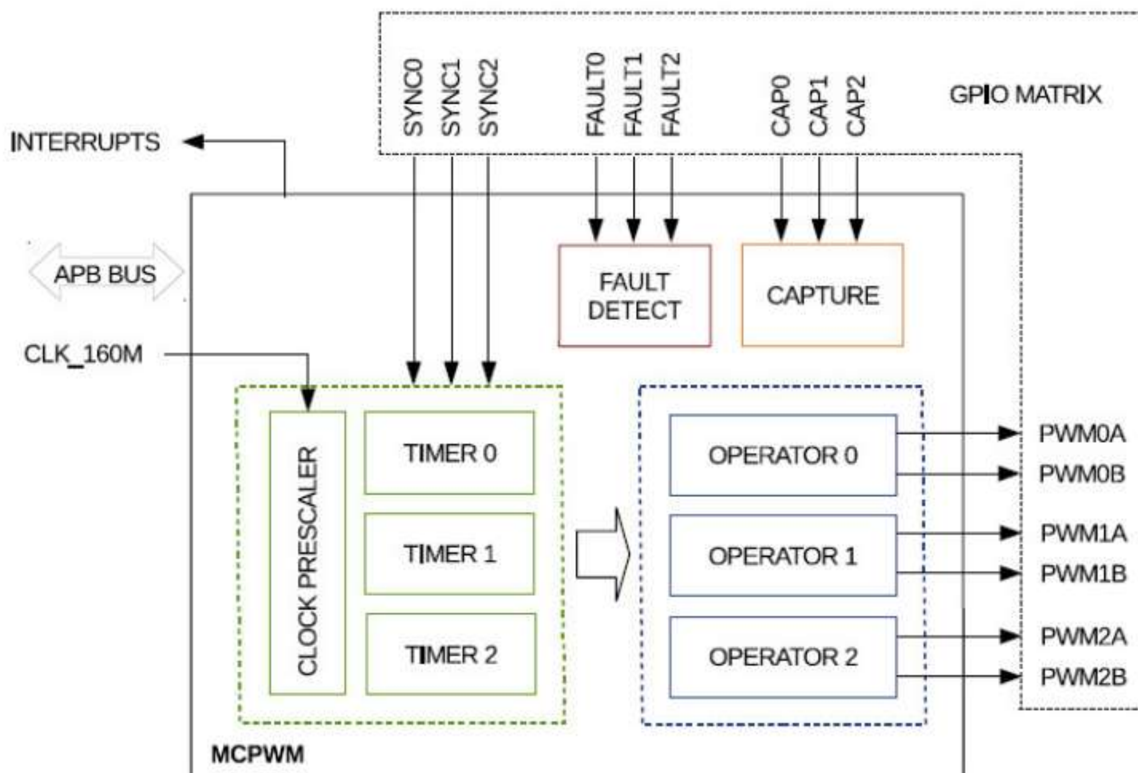



Рисунок 3.1 – Структура MCPWM модуля ESP32

PWM-сигнал – це прямокутна хвиля з фіксованою частотою, в якій змінюється тривалість сигналу HIGH у межах одного циклу. Середнє значення напруги визначається шириною імпульсу. Наприклад, за напруги живлення 3.3 В і duty cycle 25 % середня напруга становить 0.825 В.

Формально, середнє значення PWM можна обчислити за формулою:

$$U_{ср} = U_{постійне} \times (\text{duty cycle} / 100)$$

Візуально:

- duty cycle 10% – вузький імпульс → слабке світіння LED;
- duty cycle 90% – широкий імпульс → майже повна яскравість.

PWM-сигнали часто проходять через RC-фільтри, які згладжують сигнал і створюють більш плавну аналогову напругу, придатну для DAC-застосувань. Приклади ШІМ сигналу з різним коефіцієнтом заповнення зображено на рис. 3.2.

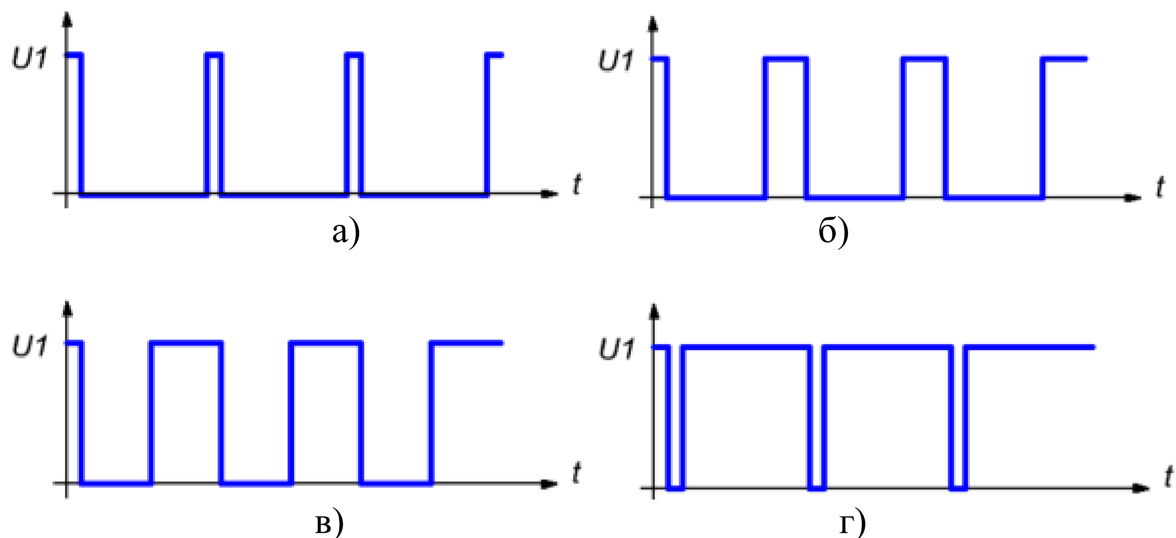



Рисунок 3.2 – ШІМ сигнал з коефіцієнтом заповнення
а) 10%; б) 30%; в) 50%; г) 100%

ESP32 має 16 апаратних PWM-каналів, які можна використовувати для керування різними пристроями. Кожен канал прив'язується до GPIO через LEDC (LED Controller). PWM поділяється на високошвидкісні (HS) та низькошвидкісні (LS) канали. Кожен таймер може мати власну частоту, розрядність та прив'язані канали.

Кількість каналів дозволяє ESP32 одночасно:

- керувати RGB-стрічками,
- керувати кількома сервоприводами,
- подавати аналогову напругу через PWM на різні пристрої.

*Ключові характеристики*

- Частота: 1 Гц – 40 МГц (обмежується розрядністю).
- Розрядність: до 16 біт (макс. значення PWM – 65535).
- Таймери: 4 окремі таймери (по 2 для HS та LS)
- GPIO: PWM може бути прив'язаний майже до будь-якого цифрового виводу.

В Arduino IDE є бібліотечна підтримка PWM на ESP32, яка реалізується за допомогою `ledc`-функцій:

```
ledcSetup(channel,  freq,  resolution_bits);    //
Налаштування таймера
ledcAttachPin(gpio, channel);                    //
Прив'язка піна
ledcWrite(channel, duty);                        //
Встановлення значення
```

Також доступні функції `ledcWriteTone` та `ledcWriteNote`, які генерують звукові сигнали для дзвінків та аудіо.

Базовий приклад. Світлодіод зі сталою яскравістю

```
const int ledPin = 18; // GPIO18

void setup() {
  ledcSetup(0, 5000, 8);     // Канал 0, 5 кГц, 8
біт
  ledcAttachPin(ledPin, 0);
  ledcWrite(0, 128);        // 50% яскравість
}

void loop() {
  // Постійне світіння
}
```

Плавне затемнення (fade-in/fade-out)

```
void loop() {
  for (int i = 0; i <= 255; i++) {
    ledcWrite(0, i);
    delay(5);
  }
  for (int i = 255; i >= 0; i--) {
    ledcWrite(0, i);
    delay(5);
  }
}
```

PWM керування з аналогового входу (потенціометр)

```
const int potPin = 34;
const int ledPin = 18;
```

```
void setup() {
  ledcSetup(0, 5000, 8);
  ledcAttachPin(ledPin, 0);
}

void loop() {
  int value = analogRead(potPin);
  int brightness = map(value, 0, 4095, 0, 255);
  ledcWrite(0, brightness);
  delay(10);
}
```

Розрядність впливає на плавність сигналу. Для LED достатньо 8 біт, для сервоприводів – 10–12 біт. Висока розрядність потребує нижчої частоти.

Наприклад:

- 8 біт @ 5 кГц – гарна універсальна конфігурація;
- 12 біт @ 1 кГц – для моторів або сервоприводів.

PWM не потрібно застосовувати на GPIO 6–11 (використовуються для SPI Flash).

Щоб перетворити PWM на справжній аналоговий сигнал (для АЦП, підсилювачів тощо), застосовують RC-фільтр:

- R ≈ 1–10 кОм.
- C ≈ 0.1–1 мкФ.
- Fcutoff = 1 / (2πRC).

Фільтр згладжує сигнал, прибирає високочастотні гармоніки, робить його придатним для аналогових входів інших пристроїв.

ESP32 підтримує динамічне налаштування PWM у реальному часі. Можна створювати ефекти затухання, адаптивного освітлення, автоматичного регулювання на основі зворотного зв'язку. Також можливе керування через Wi-Fi або Bluetooth.

*Застосування PWM у практиці*

- RGB-підсвітка.
- Аудіогенератори.
- Цифрове керування аналоговими модулями.
- Енергозбережне освітлення.
- Позиціонування сервоприводів.
- Підтримка ШІМ-контролерів двигунів.
- Адаптивне керування потужністю.

*Зв'язок ADC та PWM для реалізації керування*

У мікроконтролерних системах, зокрема на базі ESP32, важливою є можливість побудови замкнених контурів керування – коли аналоговий вхід (ADC) використовується для зчитування фізичних параметрів, а вихід (PWM) – для керування пристроями у відповідь на ці параметри.

Зв’язок ADC та PWM дозволяє реалізувати адаптивні системи: наприклад, регулювання яскравості світлодіода залежно від освітленості, керування швидкістю вентилятора залежно від температури або потужності двигуна на основі положення потенціометра.

Алгоритм виглядає так:

1. Зчитати аналоговий сигнал (наприклад, напругу з потенціометра або датчика) за допомогою `analogRead()`.
2. Перетворити значення АЦП у бажаний діапазон PWM (зазвичай від 0 до 255 для 8-бітного каналу).
3. Передати це значення на PWM-канал через `ledcWrite()`.

*Переваги такого підходу*

- Реалізація замкнених контурів керування без необхідності використання зовнішніх ЦАПів.
- Можливість адаптації вихідного сигналу в режимі реального часу.
- Побудова автономних систем, що реагують на зміни середовища.

Приклад: Регулювання яскравості світлодіода потенціометром

```
const int potPin = 34;     // аналоговий вхід
const int ledPin = 18;     // PWM-вихід

void setup() {
  ledcSetup(0, 5000, 8);  // канал 0, 5кГц, 8біт
  ledcAttachPin(ledPin, 0);
}

void loop() {
  int      adcValue      =      analogRead(potPin);
//0-4095
  int pwmValue = map(adcValue, 0, 4095, 0, 255);
// масштабування до 8 біт
  ledcWrite(0, pwmValue);
  delay(10);
}
```

ESP32 дозволяє поєднувати кілька аналогових джерел (комбінування з сенсорами), наприклад:

- керування швидкістю обертання вентилятора за допомогою датчика температури;
- регулювання інтенсивності освітлення залежно від фоторезистора;

- адаптація гучності звуку від мікрофонного вхідного сигналу.

Аналогові сигнали можуть бути зашумлені. Щоб уникнути стрибків яскравості чи шумового мерехтіння:

- застосовуйте усереднення значень ADC (наприклад, по 10 вимірювань);
- використовуйте фільтрацію ПІД-алгоритмами або RC-фільтрами;
- уникайте використання шумних GPIO (6–11) для аналогових сигналів.

Можливо поєднувати вимірювання аналогових значень з глибоким сном (deep sleep), за якого ESP32 періодично прокидається, вимірює параметр і на його основі змінює вихідний сигнал. Це особливо корисно для IoT-пристроїв на батарейках.

*Застосування в проєктах*

- Смарт-освітлення: яскравість адаптується до освітлення в кімнаті.
- Розумні вентиляційні системи: швидкість залежить від температури або вологості.
- Аудіосистеми: гучність регулюється через потенціометр.
- Робототехніка: позиціонування сервоприводів або моторів на основі сенсорних зчитувань.

Таким чином, поєднання ADC та PWM на ESP32 дозволяє реалізовувати гнучке керування аналоговими пристроями без необхідності у складних схемах або зовнішніх модулях. Це відкриває широкі можливості для побудови адаптивних, енергоефективних і розумних вбудованих систем.

В цьому варіанті завдання зчитувати аналоговий сигнал (напруга з потенціометра) та відповідно керувати яскравістю LED через PWM дозволяє познайомитись з принципами конвертації даних, масштабування та контролю реальних пристроїв.

## Порядок виконання роботи

1. Складаємо в wokwi схему для дослідження кнопок.

В нас є змінний резистор і RGB світлодіод, один контакт якого підключений до +3.3 В, а інші виводи через резистор до виводів ESP.

Кінцеву схему зображено на рис. 3.3.

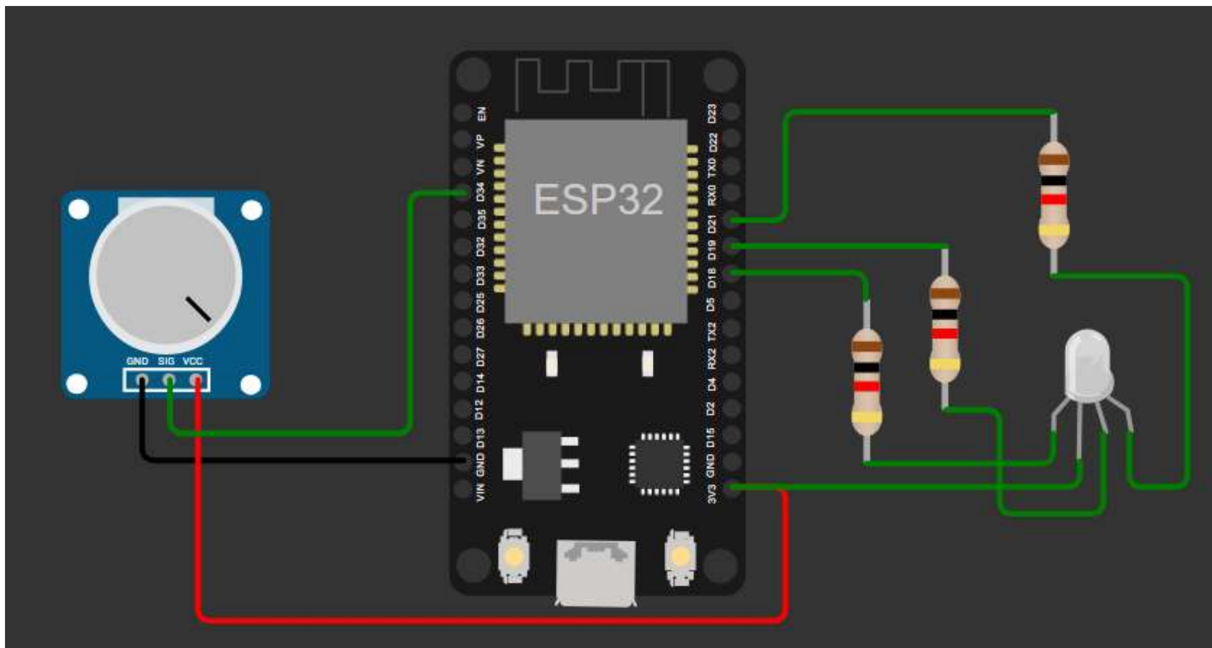



Рисунок 3.3 – Досліджувана схема в wokwi

2. Пишемо код нашої програми, яка працюватиме таким чином: за зміни положення регулятора резистора змінюватиметься колір світлодіода.

```
#include "driver/ledc.h"

const int potPin   = 34;  // ADC-вхід від потенціометра

// Піни RGB-LED (загальний анод — живлення на +, керування через GND)
const int redPin   = 18;
const int greenPin = 19;
const int bluePin  = 21;

// Загальні налаштування PWM
const ledc_mode_t      speedMode  = LEDC_LOW_SPEED_MODE;
const ledc_timer_t     timerNum   = LEDC_TIMER_0;
const uint32_t         freqHz     = 5000;             // 5 кГц
const ledc_timer_bit_t bitDepth   = LEDC_TIMER_8_BIT;  // 8 біт => 0–255

void setup() {
  Serial.begin(115200);
  while (!Serial) delay(1);

  // 1) Таймер PWM
  ledc_timer_config_t tcfg;
  tcfg.speed_mode       = speedMode;
  tcfg.timer_num        = timerNum;
  tcfg.duty_resolution  = bitDepth;
  tcfg.freq_hz          = freqHz;
  tcfg.clk_cfg          = LEDC_AUTO_CLK;
  ledc_timer_config(&tcfg);
```

```
  // 2) Канали PWM для R, G, B
  ledc_channel_config_t ccfg;
  // RED
  ccfg.speed_mode = speedMode;
  ccfg.channel    = LEDC_CHANNEL_0;
  ccfg.timer_sel  = timerNum;
  ccfg.intr_type  = LEDC_INTR_DISABLE;
  ccfg.gpio_num   = redPin;
  ccfg.duty       = 0;
  ccfg.hpoint     = 0;
  ledc_channel_config(&ccfg);
  // GREEN
  ccfg.channel    = LEDC_CHANNEL_1;
  ccfg.gpio_num   = greenPin;
  ledc_channel_config(&ccfg);
  // BLUE
  ccfg.channel    = LEDC_CHANNEL_2;
  ccfg.gpio_num   = bluePin;
  ledc_channel_config(&ccfg);
}

void loop() {
  int raw = analogRead(potPin);  // 0–4095

  // Зворотна шкала PWM для загального анода
  int dutyR = 255 - map(raw, 0, 4095, 0, 255);   // чим більше raw, тим
темніше R
  int dutyG = 255 - map(raw, 0, 4095, 255, 0);   // G — обернено
  int dutyB = 255 - map(raw, 0, 4095, 0, 255);   // B — напівдіапазон

  ledc_set_duty(speedMode, LEDC_CHANNEL_0, dutyR);
  ledc_update_duty(speedMode, LEDC_CHANNEL_0);

  ledc_set_duty(speedMode, LEDC_CHANNEL_1, dutyG);
  ledc_update_duty(speedMode, LEDC_CHANNEL_1);

  ledc_set_duty(speedMode, LEDC_CHANNEL_2, dutyB);
  ledc_update_duty(speedMode, LEDC_CHANNEL_2);

  // Вивід у консоль
  Serial.print("ADC=");  Serial.print(raw);
  Serial.print(" | R="); Serial.print(dutyR);
  Serial.print(" G=");   Serial.print(dutyG);
  Serial.print(" B=");   Serial.println(dutyB);

  delay(100);
}
```

Пояснення нашого коду.

Підключення бібліотеки

```
#include "driver/ledc.h"
```

Це спеціальна бібліотека для ESP32, яка дозволяє працювати з апаратними PWM-таймерами (`LEDC` – LED Control).

Визначення пінів

```
const int potPin   = 34;
const int redPin   = 18;
const int greenPin = 19;
const int bluePin  = 21;
```

- `potPin` – вхід, до якого підключений потенціометр.
- `redPin`, `greenPin`, `bluePin` – виводи ESP32, підключені до червоного, зеленого та синього каналів RGB-світлодіода.

Налаштування PWM

```
const ledc_mode_t       speedMode  =
LEDC_LOW_SPEED_MODE;
const ledc_timer_t      timerNum   = LEDC_TIMER_0;
const uint32_t          freqHz     = 5000;
const ledc_timer_bit_t bitDepth    =
LEDC_TIMER_8_BIT;
```

- PWM-модуль працює в низькошвидкісному режимі (`LOW_SPEED_MODE`).
- Таймер 0 використовується для всіх 3 каналів.
- Частота ШІМ – 5 кГц.
- Роздільна здатність – 8 біт, тобто `0-255` рівнів яскравості.

Функція `setup()`

```
Serial.begin(115200);
```

Запускає послідовну передачу даних на швидкості 115200 бод для виведення у Serial Monitor.

Конфігурація таймера

```
ledc_timer_config_t tcfg;
...
ledc_timer_config(&tcfg);
```

Створюється структура налаштувань таймера, яка визначає:

- частоту;
- розрядність PWM;
- який таймер і режим буде використовуватись.

Конфігурація кожного каналу PWM

Для кожного каналу (RED, GREEN, BLUE) виконується налаштування:

```
ledc_channel_config_t ccfg;
...
ledc_channel_config(&ccfg);
```

Параметри:

- канал: `LEDC_CHANNEL_0`, `1`, `2`;
- GPIO: відповідний до каналу;
- duty: стартова яскравість 0 (вимкнено);
- hpoint: hardware trigger point – встановлено в 0.

Функція `loop()`

Зчитування з потенціометра

```
int raw = analogRead(potPin);  // 0–4095
```

Зчитує аналогове значення з потенціометра.

Діапазон ESP32 ADC: 0–4095.

Генерація кольору через PWM

```
int dutyR = 255 - map(raw, 0, 4095, 0, 255);
int dutyG = 255 - map(raw, 0, 4095, 255, 0);
int dutyB = 255 - map(raw, 0, 4095, 0, 255);
```

- `map()` перетворює зчитане значення (`0–4095`) у діапазон `0–255`.
- `255 - ...` – тому що використовується світлодіод із загальним анодом (тобто яскравість вища за меншого сигналу).

Запис значення PWM

```
ledc_set_duty(speedMode, LEDC_CHANNEL_0, dutyR);
ledc_update_duty(speedMode, LEDC_CHANNEL_0);
...
```

Ці пари функцій оновлюють значення яскравості кожного каналу.

Виведення значень у Serial Monitor

```
Serial.print("ADC=");  Serial.print(raw);
```

```
Serial.print(" | R="); Serial.print(dutyR);
Serial.print(" G=");   Serial.print(dutyG);
Serial.print(" B=");   Serial.println(dutyB);
```

Це дозволяє бачити в реальному часі значення, які використовуються для кожного кольору, у вигляді:

```
ADC=2180 | R=119 G=136 B=119
```

Пауза

```
delay(100);
```

Коротка затримка (100 мс) перед наступною ітерацією.
Таким чином цей скетч:
- читає значення потенціометра;
- динамічно змінює колір RGB-світлодіода через PWM;
- показує зміну значень у консолі.

3. Досліджуємо результати роботи нашого коду та схеми.

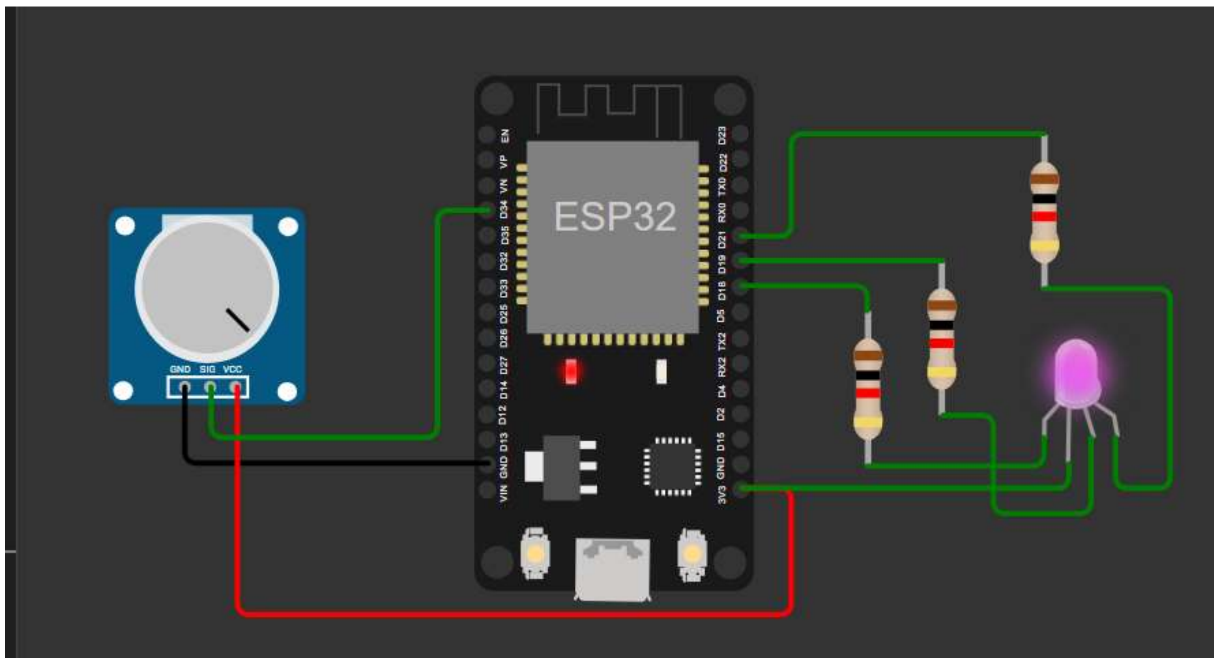



Рисунок 3.4 – Результат моделювання
у разі правого положення змінного резистора

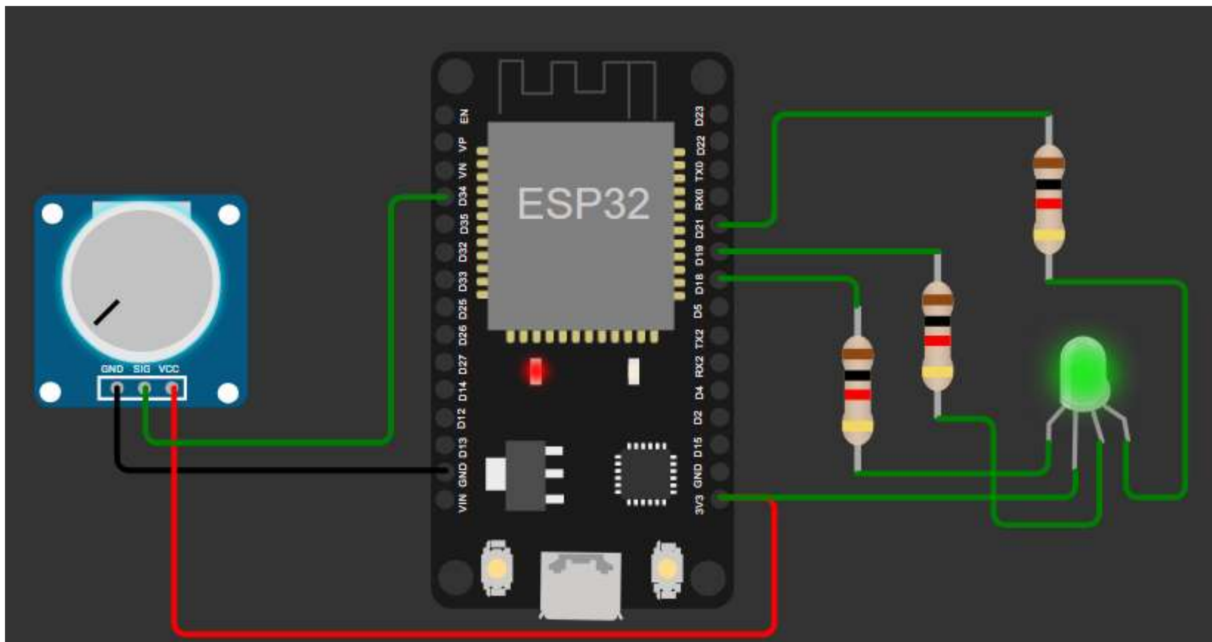



Рисунок 3.5 – Результат моделювання
у разі лівого положення змінного резистора

Таким чином, ми отримаємо уявлення про роботу аналогових входів та логіку роботи PWM у мікроконтролерних системах. Результати наведено на рис. 3.4 – 3.5.

3. Складаємо аналогічну схему на макетній платі з ESP32 (рис. 3.6) та проводитимемо її дослідження (рис. 3.7).

Код програми такий:

```
const int redPin   = 18;  // GPIO для червоного
const int greenPin = 19;  // GPIO для зеленого
const int bluePin  = 21;  // GPIO для синього
const int potPin   = 34;  // GPIO для потенціометра (ADC)
const int freq = 5000;
const int resolution = 8;
const int redChannel   = 0;
const int greenChannel = 1;
const int blueChannel  = 2;

void setup() {
  Serial.begin(9600);
ledcAttach(redPin, freq, resolution);
ledcAttach(greenPin, freq, resolution);
ledcAttach(bluePin, freq, resolution);
}

void loop() {
  int adcValue = analogRead(potPin); // 0-4095
  int range = map(adcValue, 0, 4095, 0, 765); // 0-765 для плавного переходу
```

```
  int red, green, blue;
  if (range <= 255) {
    red = 255 - range;
    green = range;
    blue = 0;
  } else if (range <= 510) {
    red = 0;
    green = 510 - range;
    blue = range - 255;
  } else {
    red = range - 510;
    green = 0;
    blue = 765 - range;
  }
  ledcWrite(redPin, red);
  ledcWrite(greenPin, green);
  ledcWrite(bluePin, blue);
  Serial.print("ADC: "); Serial.print(adcValue);
  Serial.print(" | R: "); Serial.print(red);
  Serial.print(" G: "); Serial.print(green);
  Serial.print(" B: "); Serial.println(blue);
  delay(50);
}
```

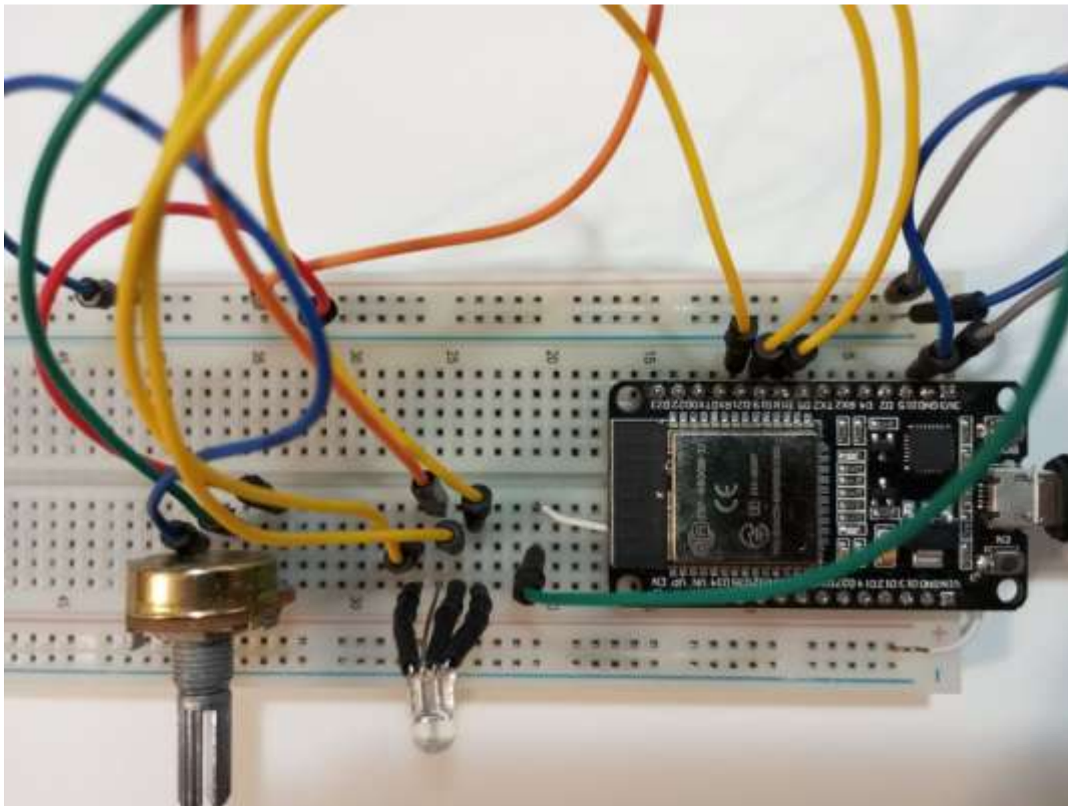


Рисунок 3.6 – Досліджувана схема, складена на макетній платі

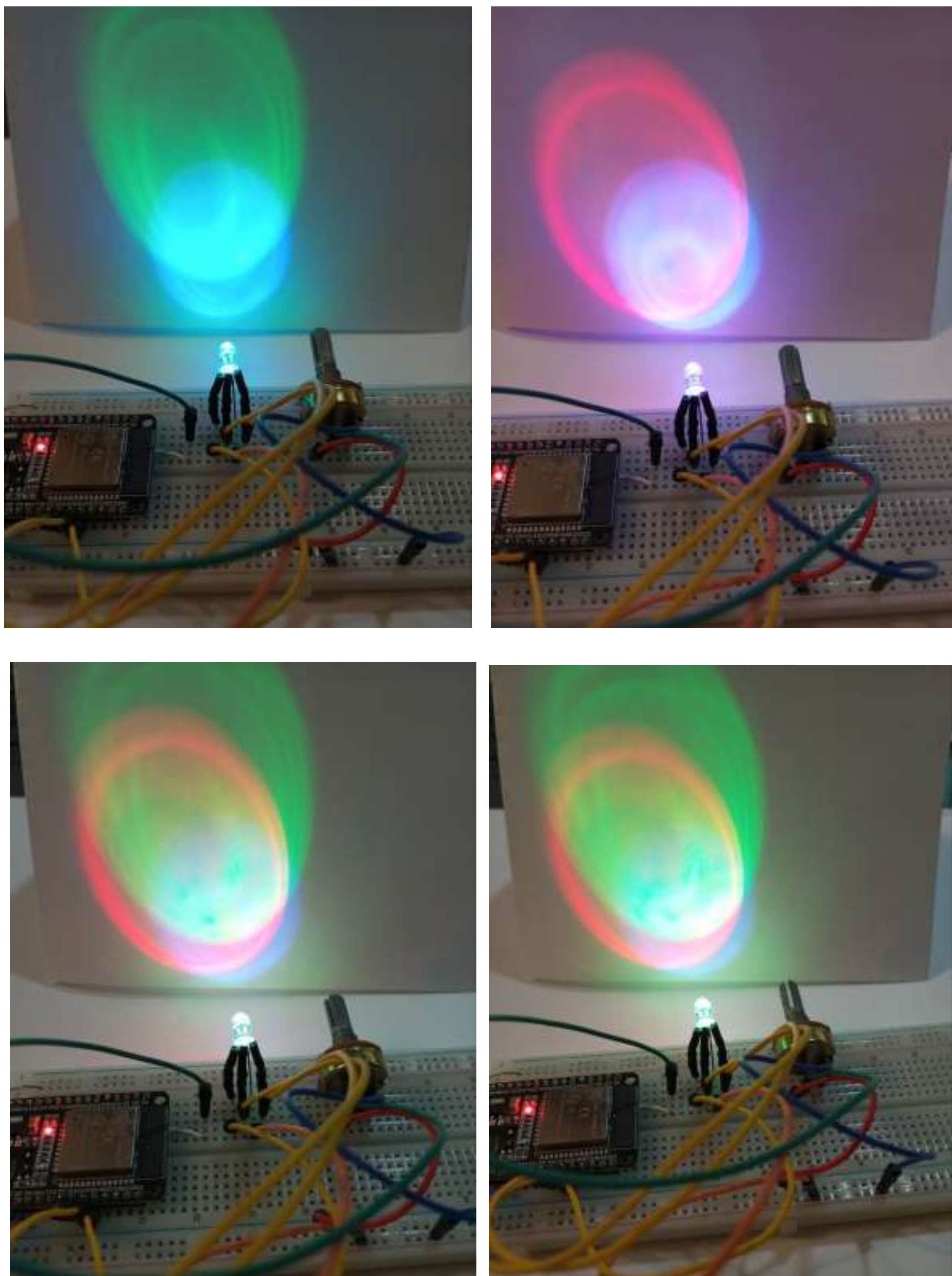


Рисунок 3.7 – Результат дослідження макета за різних положень змінного резистора

Таким чином, практична реалізація та дослідження підтвердили теоретичні основи роботи з аналоговими сигналами та логіку роботи PWM у мікроконтролерних системах.

**Варіанти завдань**

1. Реалізувати зміну яскравості одного світлодіода за допомогою потенціометра.
2. Реалізувати керування швидкістю обертання DC-мотора залежно від положення потенціометра.
3. Змінювати частоту звукового сигналу на п’єзоелементі на основі аналогового входу.
4. Створити плавне затемнення світлодіода (fade in/out) з керуванням інтенсивності через потенціометр.
5. Реалізувати керування інтенсивністю освітлення залежно від рівня зовнішнього освітлення (датчик освітленості).
6. Змінювати колір RGB-світлодіода залежно від положення одного потенціометра.
7. Реалізувати незалежне керування червоним, зеленим та синім каналами за допомогою трьох потенціометрів.
8. Створити ефект дихаючого RGB (плавна зміна кольору) з можливістю керування швидкістю через потенціометр.
9. Реалізувати зміну кольору на RGB-світлодіоді залежно від температури з термодатчика.
10. Програмно задавати переливання кольорів (color transition) із затримкою, керованою аналоговим сигналом.
11. Керувати напрямком і швидкістю обертання DC-мотора (PWM + логіка).
12. Керувати положенням сервомотора залежно від потенціометра.
13. Змінювати швидкість обертання вентилятора залежно від температури.
14. Реалізувати плавний запуск та зупинення двигуна із аналоговим керуванням.
15. Додати захист двигуна: у випадку перевищення певного значення на аналоговому вході – вимкнути обертання.
16. Зчитувати два аналогових сигнали та керувати двома незалежними світлодіодами (PWM).
17. Реалізувати ефект тіньового датчика: світлодіод яскравішає у разі наближення руки (на основі фоторезистора).
18. Керувати одночасно RGB-світлодіодом та швидкістю двигуна від різних потенціометрів.
19. Реалізувати світловий індикатор рівня: залежно від сигналу плавно засвічувати 3 світлодіоди.
20. Створити симуляцію зміни кольору RGB-світлодіода за натискання кнопки, а інтенсивність PWM – від потенціометра.

## Зміст звіту

Звіт має містити:

1. Завдання.
2. Обгрунтування алгоритму програми.
3. Текст коду програми.
4. Електричні схеми реалізованих рішень (фото, скріни і под.).
5. Висновки за результатами проведених досліджень.

## Контрольні запитання

1. Що таке аналоговий сигнал і чим він відрізняється від цифрового?
2. Яке призначення аналого-цифрового перетворювача (ADC) у мікроконтролері?
3. Який діапазон значень повертає функція analogRead() на ESP32?
4. Що таке широтно-імпульсна модуляція (PWM)?
5. Як впливає коефіцієнт заповнення (duty cycle) PWM на яскравість світлодіода?
6. Яку роль відіграє частота PWM-сигналу?
7. Чим відрізняються 8-бітна та 10-бітна розрядності PWM?
8. Які функції використовуються для створення PWM на ESP32?
9. Яка функція використовується для прив’язки PWM-каналу до певного GPIO-піна?
10. Для чого використовується ledc_update_duty()?
11. Який діапазон значень набуває функція map() і для чого вона використовується?
12. Який тип змінної краще використовувати для зберігання значень з analogRead()?
13. Що відбудеться, якщо не викликати ledc_update_duty() після ledc_set_duty()?
14. Як підключити RGB-світлодіод із загальним анодом до ESP32?
15. Як змінити частоту PWM-сигналу для підключеного світлодіода?
16. Як перевірити, чи дійсно змінюється яскравість світлодіода?
17. Що буде, якщо до одного GPIO підключити одразу кілька PWM-каналів?
18. Чи можна використовувати один таймер для кількох PWM-каналів? Як?
19. Яка бібліотека потрібна для використання ledc_timer_config_t?
20. Як підключити двигун до ESP32 для керування його швидкістю через PWM?

# Лабораторна робота 4
## Серійна комунікація

**Мета**. Ознайомитися з принципами роботи серійного порту UART у мікроконтролері ESP32, навчитися використовувати функції обміну даними через Serial Monitor, зчитувати та передавати дані між ESP32 і ПК, а також розробити базові приклади введення/виведення інформації.

### Основні теоретичні відомості

UART (Універсальний асинхронний приймач-передавач) – це спеціалізований апаратний блок, що забезпечує передачу інформації між двома пристроями без потреби у спільному тактовому сигналі. У цій системі обмін даними виконується асинхронно: приймач і передавач використовують стартові та стоп-біти для синхронізації. Інформація передається побітово, зазвичай по одному байту (8 біт), починаючи зі стартового біта, за яким ідуть інформаційні біти, додатково може передаватися біт парності та один або кілька стоп-бітів.

Структура інтерфейсу UART містить приймач (Rx) і передавач (Tx). Ці компоненти можуть бути частиною мікроконтролера або окремими мікросхемами, які забезпечують перетворення рівнів сигналу. Передача відбувається через дві лінії: одна для надсилання, інша – для прийому.

Підключення UART між двома пристроями зазвичай здійснюється шляхом перехресного з’єднання: вихід передавача одного пристрою з’єднується з входом приймача іншого. Для стабільної роботи також необхідне підключення GND обох пристроїв.

Для деяких випадків застосовуються спеціальні мікросхеми, такі як SC16C850, які забезпечують перетворення між паралельною та послідовною передачею даних.

Основним призначенням лінії передавача та приймача для кожного пристрою є передача та приймання послідовних даних, призначених для послідовного зв’язку.

Інтерфейс UART може використовуватись для обміну даними між двома мікроконтролерами як показано на рис. 4.1.

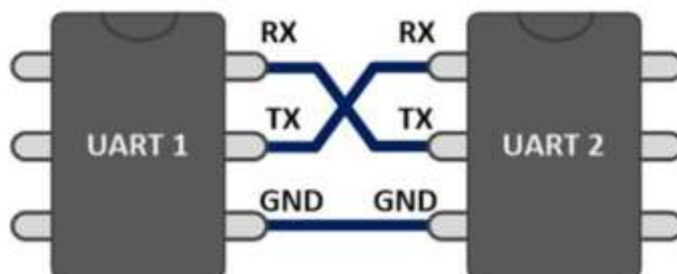



Рисунок 4.1 – Комунікація двох мікроконтрорлерів через UART

Для зв'язку двох пристроїв знадобляться два дроти, причому з'єднувати їх потрібно хрест-навхрест – RX першого в TX другого і навпаки. Провід землі (GND) необхідний для підтримання однакової напруги логічних рівнів на обох пристроях.

Також існують мікросхеми (наприклад SC16C850), які перетворюють паралельні дані в послідовний інтерфейс UART та навпаки.

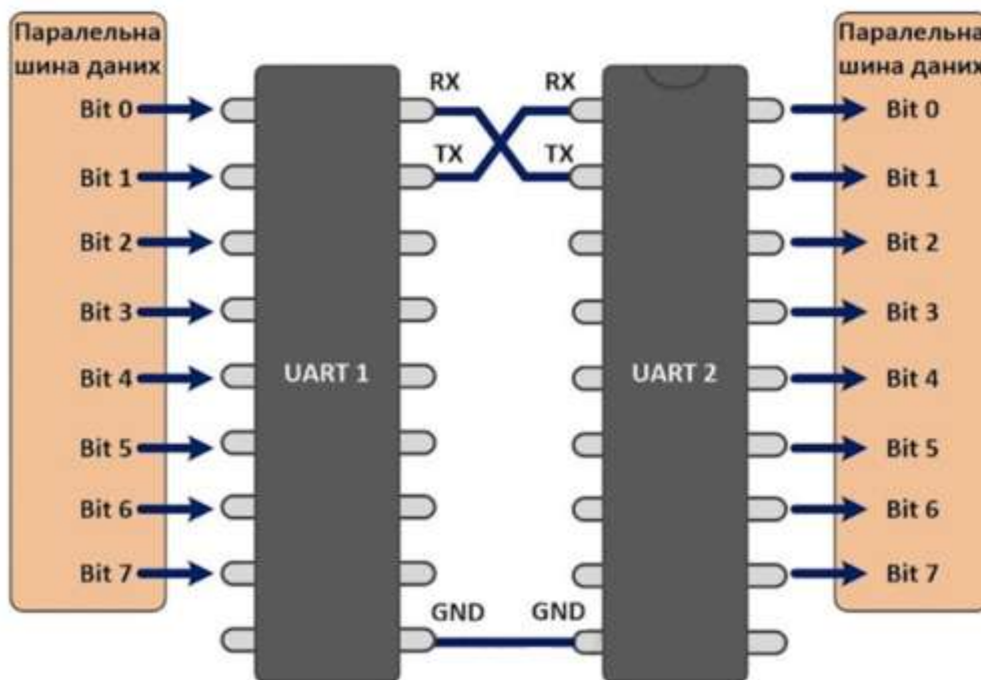



Рисунок 4.2 – Комунікація двох пристроїв
з використанням мікросхем перетворювачів

Також можна використовувати апаратне керування потоком з UART (рис. 4.3) за допомогою двох додаткових ніжок, які називаються RTS (англ. request to send) і CTS (англ. clear to send).

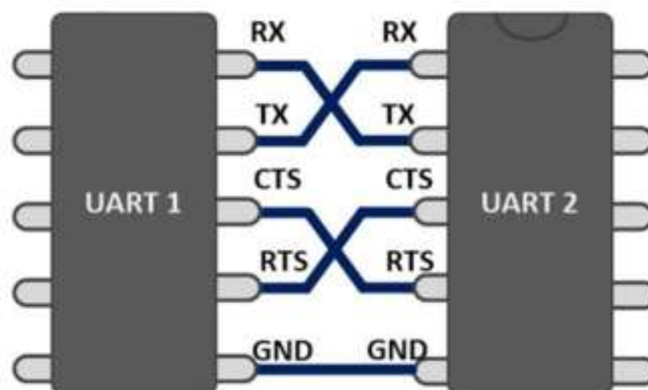



Рисунок 4.3 – Комунікація через UART
з апаратним керуванням потоком

Ці провідники перехресно з’єднані між двома пристроями. Якщо апаратне керування потоком увімкнено, кожна сторона використовуватиме свій RTS, щоб вказати, що вона готова надіслати нові дані, і прочитає свій CTS, щоб перевірити, чи дозволено їй надсилати дані на іншу сторону.

На практиці найбільш доступно підключення UART можна продемонструвати на двох платах Arduino UNO. Arduino UNO – це досить популярна дешева плата для розробки різних пристроїв на мікроконтролерах AVR. На сумісних з Arduino платах UART позначається символами RX та TX поруч із відповідними виводами. На Arduino UNO це 0 та 1 цифрові виводи як показано на рис. 4.4.

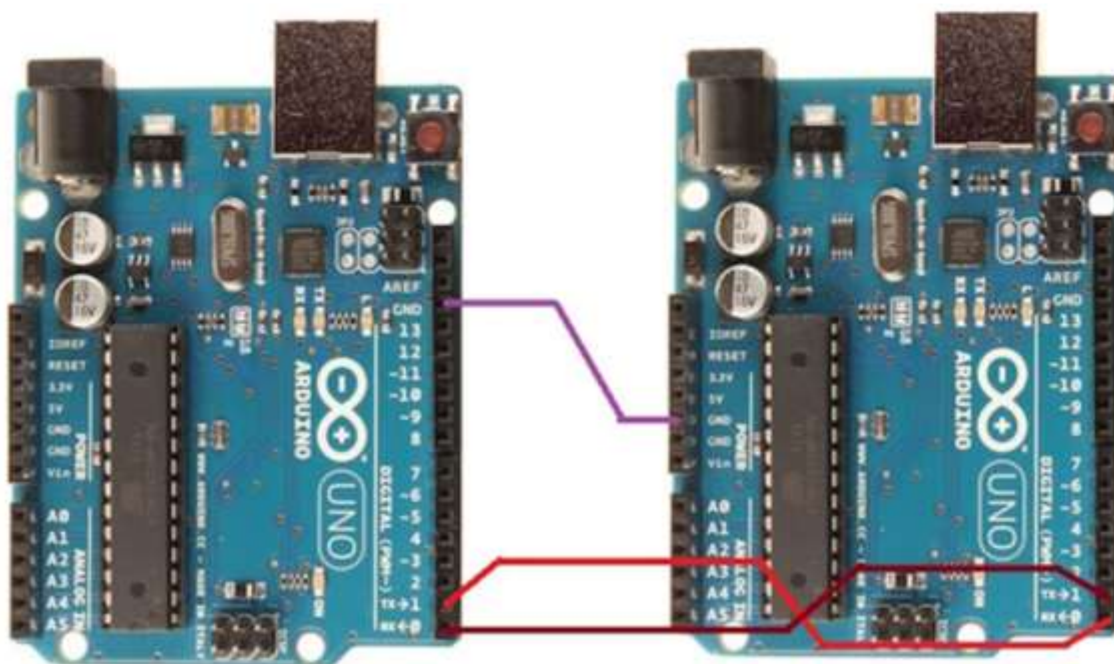



Рисунок 4.4 – Комунікація через UART двох плат Arduino UNO

Апаратна реалізація UART зазвичай працює з рівнями сигналів TTL, де логічна одиниця відповідає напрузі до 5 В. Такий варіант добре підходить для передачі даних на короткі відстані – наприклад, у межах однієї плати. Проте за необхідності зв’язку між пристроями на більшій відстані використовують спеціальні підсилювачі або перетворювачі рівнів, які забезпечують вищу напругу. Найчастіше це стандартизовані рішення, зокрема за протоколами RS-232 або RS-485 (рис. 4.5).

У протоколі RS-232 логічному нулю відповідає позитивна напруга в діапазоні від +5 В до +15 В, тоді як логічна одиниця визначається негативною напругою в межах від −5 В до −15 В. Цей інтерфейс зазвичай застосовується для з’єднання мікроконтролерів із ПК через COM-порт. Найпоширенішим перетворювачем TTL↔RS-232 є мікросхема MAX232.

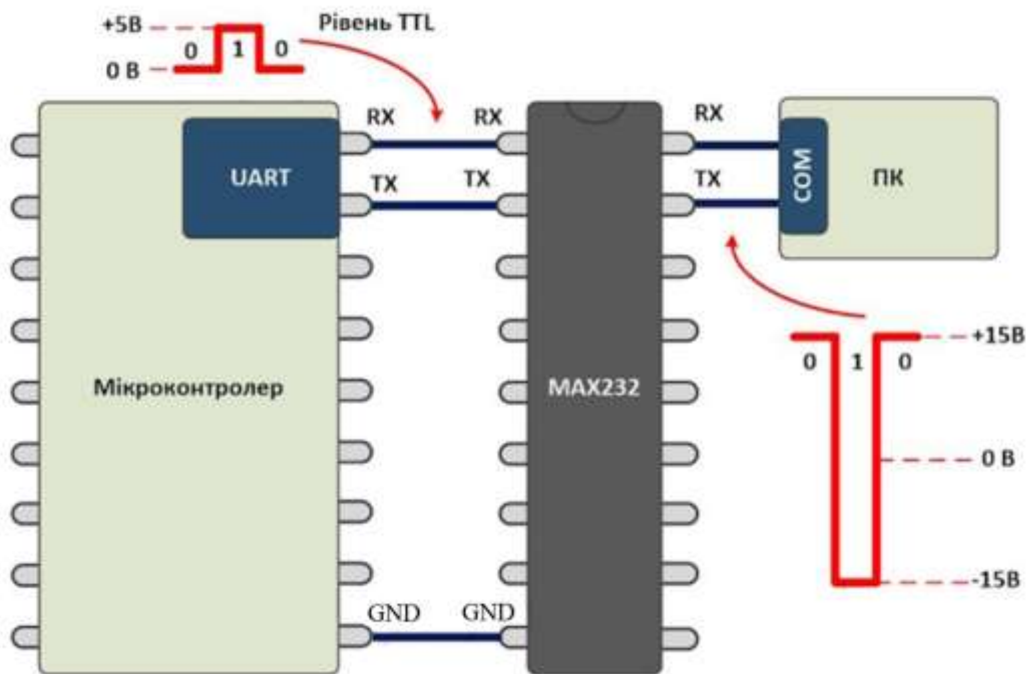



Рисунок 4.5 – Передачі даних через UART з перетворювачем рівнів

Прикладом використання UART з мікросхемою MAX232 є схема підключення мікроконтролера ESP8266 до ПК як показано на рис. 4.6:

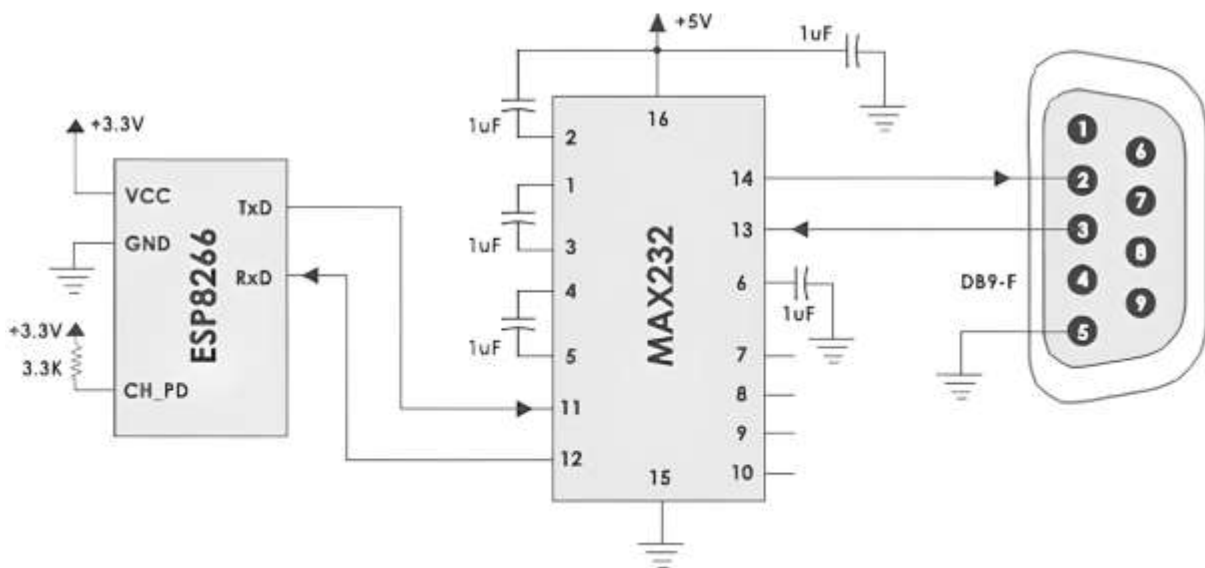



Рисунок 4.6 – Підключення мікроконтролера ESP8266 до ПК 9

Існують також мікросхеми сполучення USB з UART (рис. 4.7), наприклад мікросхема FT232RL або CH340G. Для системи Windows необхідно для них встановити спеціальний драйвер.

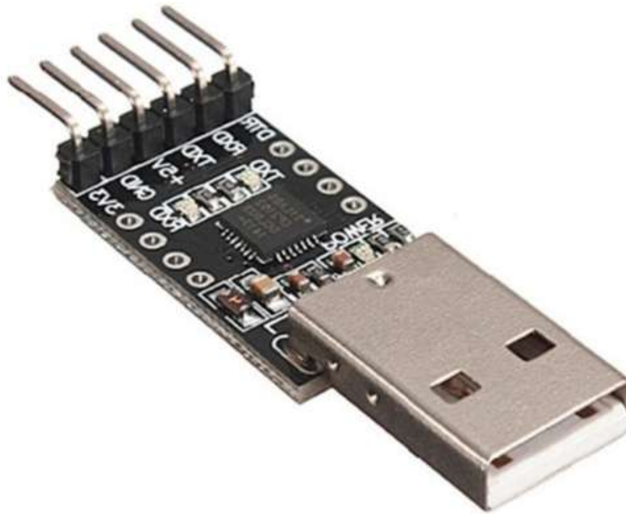


Рисунок 4.7 – Використання мікросхем в перехідниках USB

Зв'язок UART здійснюється в так званих кадрах (рис. 4.8). Кожен кадр складається з бітів – фрагментів інформації, які послідовно надсилаються по лінії. Ці біти підряд містять початковий біт, біти даних, додатковий біт парності та один або два стоп-біти. Кожен біт може мати або високу (вказує на логічну 1), або низьку напругу (вказує на логічний 0). Кожен біт кожного байта передається на рівний відведений проміжок часу (фактично, тайм-слот).

Рисунок 4.8 – Використання мікросхем в перехідниках USB

Поки інформація не передається, на лінії зберігається високий рівень. Спад сигналу – це команда приймачу, що почнеться передача байта у вигляді ряду нулів та/або одиниць з дотриманням заздалегідь обумовлених часових відрізків. Після восьмого біта йде стоп-сигнал високого рівня – чекаємо передачі наступного байта.

Додатковий (необов'язковий) біт парності дозволяє приймачу перевірити, чи правильно отримано дані. Цей біт знаходиться між останнім

бітом даних і стоп-бітом. Значення біта залежатиме від використовуваного типу паритету: парного чи непарного.

Апаратно контролер UART реалізований з допомогою зсувних регістрів як показано на рис. 4.9. Кожен UART містить регістр зсуву, який є основним методом перетворення між послідовною та паралельною формами.

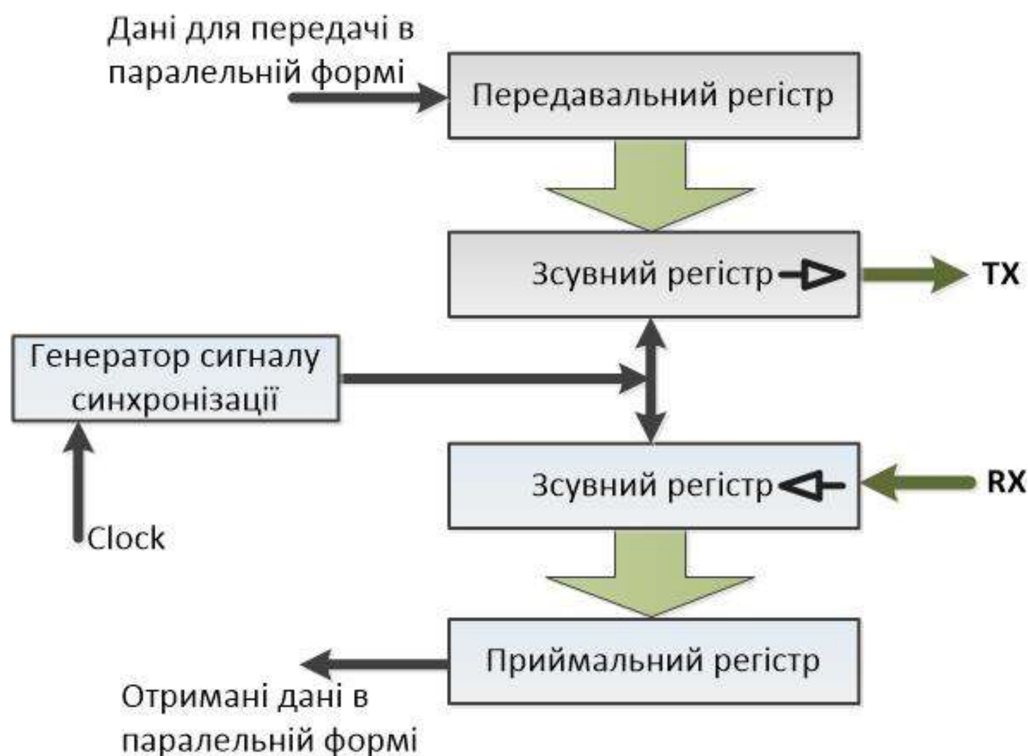



Рисунок 4.9 – Апаратна реалізація контролера UART

Оскільки протокол UART є асинхронним, то передавач і приймач не мають спільного сигналу синхронізації. Тому потрібно, щоб приймач та передавач мали однакову швидкість передачі даних, яка заздалегідь обумовлена. Інакше повідомлення буде спотворено та неправильно витлумачено. Стандарт передбачає можливу похибку у швидкості до 5 %

Швидкість вимірюється в бітах за секунду, або коротко – у бодах. Найпоширеніші швидкості передачі даних UART на сьогодні становлять 4800, 9600, 19200, 57600 і 115200. Для високої швидкості передачі даних рекомендується використовувати коротші кабелі.

Дані передаються байт за байтом, кожен байт містить:

- старт-біт (1 біт) – початок передачі;
- біти даних (звичайно 8);
- необов’язковий біт парності – для перевірки помилок;
- стоп-біт(и) – кінець передачі (1 або 2 біти).

*Параметри налаштування UART*

Щоб забезпечити правильну комунікацію між пристроями, обидві сторони мають бути налаштовані однаково.

- Швидкість передачі (baud rate) – наприклад, 9600, 115200 біт/с.
- Кількість бітів даних – 8 біт є найпоширенішим варіантом.
- Парність – odd, even або none.
- Кількість стоп-бітів – 1 або 2.

Швидкість передачі (baud rate) – це кількість бітів, які передаються за секунду.

Стандартні значення:

`9600`, `19200`, `38400`, `57600`, `115200`, `230400`, `460800` біт/с.

Для коректного передавання важливо, щоб обидва пристрої (наприклад, ESP32 і ПК) працювали з однаковою швидкістю.

UART може використовуватись у таких конфігураціях:

- ESP32 ⇄ ПК (через USB-UART),
- ESP32 ⇄ GPS / GSM / Bluetooth модуль,
- ESP32 ⇄ ESP32 (через хрестове з'єднання TX-RX).

Для виявлення помилок UART підтримує:

- біт парності (even/odd);
- перевірку переповнення буфера;
- визначення framing error (відсутність стоп-біта) ;
- контроль збоїв лінії (break condition).

У системах реального часу важливо враховувати затримки. Для мінімізації:

- використовують FIFO-буфери;
- налаштовують переривання;
- застосовують DMA для безперервного потоку.

UART застосовується у багатозадачних системах (FreeRTOS і Zephyr). Потоки можуть читати/записувати UART з використанням черг. Забезпечується неблокувальна обробка введення/виведення. Це дозволяє паралельну роботу UART і логіки контролера.

Якщо проаналізувати мікроконтролер ESP32, то він підтримує широкий набір інтерфейсів для зв'язку з периферійними пристроями. Завдяки цьому він ідеально підходить для проєктів Інтернету речей (IoT), вбудованих систем, сенсорних мереж тощо. Розглянемо основні інтерфейси ESP32 та проведемо їх порівняння.

До *основних інтерфейсів ESP32* належать:

1. UART (Universal Asynchronous Receiver-Transmitter)

- Тип: асинхронний послідовний інтерфейс.
- Кількість портів: 3 апаратні (UART0, UART1, UART2).
- Швидкість: до 5 Мбіт/с.
- Застосування: налагодження (Serial Monitor), підключення Bluetooth, GPS, GSM, датчиків.
- Переваги: проста реалізація, 2 дроти (TX, RX).
- Недоліки: тільки точка-точка, немає синхронізації.

2. SPI (Serial Peripheral Interface)
- Тип: синхронний послідовний інтерфейс.
- Кількість: 4 SPI-контролери (включно з HSPI, VSPI).
- Швидкість: до 80 МГц.
- Застосування: дисплеї (TFT, OLED), карти пам’яті, швидкі датчики.
- Переваги: висока швидкість, підтримка кількох slave-пристроїв.
- Недоліки: більше дротів (MISO, MOSI, SCK, SS), складніше конфігурування.

3. I2C (Inter-Integrated Circuit)
- Тип: синхронний послідовний шинний інтерфейс.
- Кількість: 2 порти I2C (програмно можна реалізувати більше).
- Швидкість: до 400 кГц (Fast Mode), до 3.4 МГц (High-Speed Mode).
- Застосування: підключення датчиків (температури, освітлення, тиску), RTC, EEPROM.
- Переваги: лише 2 дроти (SDA, SCL), можливість підключити до 127 пристроїв.
- Недоліки: нижча швидкість, потенційна нестабільність на довгих шинах.

4. CAN (Controller Area Network)
- Тип: диференціальний послідовний інтерфейс.
- Наявність: є один апаратний CAN-контролер (через зовнішній трансивер).
- Швидкість: до 1 Мбіт/с.
- Застосування: автомобільна електроніка, промислові контролери.
- Переваги: надійність, захист від завад, робота в мережі до 110 вузлів.
- Недоліки: складна реалізація, потрібні драйвери (наприклад, MCP2551).

5. USB (Universal Serial Bus)
- Тип: швидкісний цифровий інтерфейс.
- Наявність: у базових ESP32 немає вбудованого USB (окрім ESP32-S2/S3).

- Застосування: прошивка, емуляція HID/CDC, USB-OTG.
- Переваги: універсальність, живлення через USB.
- Недоліки: складна реалізація, потрібно вибирати відповідні SoC.

6. Wi-Fi / Bluetooth
- Тип: бездротові інтерфейси.
- Наявність: у всіх ESP32.
- Застосування: мережі, IoT, зв'язок між пристроями.
- Переваги: бездротовість, підтримка BLE.
- Недоліки: споживання енергії, залежність від прошивки.

Результати порівняння різних інтерфейсів наведено в таблиці 4.1.

Таблиця 4.1 – Результати порівняння різних інтерфейсів в ESP32

| Інтерфейс | Тип | Швидкість | Дроти | Пристрої на шині | Переваги | Недоліки |
|---|---|---|---|---|---|---|
| UART | Асинхронний | ~5 Мбіт/с | 2 | 1:1 | Простий, налагодження | Без синхронізації |
| SPI | Синхронний | до 80 МГц | 4 | 1:множина | Дуже швидкий | Більше дротів |
| I2C | Синхронний | до 3.4 МГц | 2 | до 127 | Економія пінів | Нижча швидкість |
| CAN | Диференціальний | до 1 Мбіт/с | 2 + трансивер | ~110 | Надійність, стійкість до завад | Складність |
| USB | Швидкісний | до 480 Мбіт/с | 2 (D+, D−) | точка-точка або хаб | Стандартний для ПК | Не на всіх ESP32 |
| Wi-Fi | Бездротовий | до 150 Мбіт/с | – | IP-мережа | Інтернет, бездротовий обмін | Споживання енергії |
| Bluetooth | Бездротовий | до 2 Мбіт/с | – | кілька пристроїв | BLE, мобільна інтеграція | Обмежена дальність |

Таким чином, можна зробити висновок, що мікроконтролер ESP32 підтримує широкий набір інтерфейсів для обміну даними, кожен з яких має свої особливості. Вибір інтерфейсу залежить від вимог до швидкості, кількості пристроїв, способу підключення та умов експлуатації.

Наприклад:
- UART – для налагодження та простих датчиків.
- SPI – для швидкісних периферійних пристроїв.
- I2C – коли потрібно підключити багато простих пристроїв.
- CAN – у промисловості або автомобілях.
- USB – для прямої взаємодії з ПК.
- Wi-Fi/Bluetooth – для бездротових рішень.

Перейдемо тепер до детального аналізу можливостей UART в ESP32.

Функції UART в Arduino для ESP32 наведено в табл. 4.2.

ESP32 має три апаратні UART-порти:

- UART0 – використовується Arduino IDE для моніторингу (`Serial`);
- UART1, UART2 – доступні для підключення зовнішніх пристроїв (GPS, Bluetooth, датчики).

У більшості плат ESP32 за замовчуванням:

- TX0 = GPIO1, RX0 = GPIO3;
- TX1 = GPIO10, RX1 = GPIO9;
- TX2 = GPIO17, RX2 = GPIO16.

Таблиця 4.2 – Функції UART в ESP32

| Функція | Призначення |
|---|---|
| `Serial.begin(115200)` | Ініціалізація UART із заданою швидкістю |
| `Serial.print(val)` | Вивід значення `val` у UART без переносу рядка |
| `Serial.println(val)` | Вивід `val` з переходом на новий рядок |
| `Serial.available()` | Перевірка, чи є доступні вхідні байти |
| `Serial.read()` | Зчитування одного байта з UART |
| `Serial.readString()` | Зчитування рядка, поки не надійде символ завершення |
| `Serial.flush()` | Очікування завершення передачі |

Приклад роботи з UART:

```
void setup() {
  Serial.begin(9600); // Старт UART з baud rate 9600
}

void loop() {
  if (Serial.available()) {
    char c = Serial.read(); // Зчитуємо введений символ
    Serial.print("You typed: ");
    Serial.println(c);
  }
}
```

*Застосування UART*

- Підключення датчиків (GPS, RFID, Bluetooth-модулі).
- Комунікація між мікроконтролерами.
- Налагодження програм через Serial Monitor.
- Передача даних між ESP32 та комп’ютером.
- Модемне з’єднання: ESP32 через SIM800 для SMS / GPRS.
- IoT-платформи: обмін через UART з LoRa-модулями.
- Термінальні інтерфейси CLI для налаштування пристрою в польових умовах.
- Програмне оновлення прошивки (UART Bootloader).

*Переваги UART*

- Проста реалізація.
- Лише 2 дроти.
- Широко підтримується.

*Недоліки UART*

- Асинхронність може спричинити помилки.
- Лише точка-точка (1 передавач – 1 приймач).
- Максимальна швидкість нижча, ніж у SPI.

*Обмеження UART*

- Відсутність тактового сигналу обмежує довжину кабелів.
- Неможливість підключення більше двох пристроїв у стандартному режимі.
- Чутливість до завад за високих швидкостей або довгих кабелів.

Таким чином, UART – один з базових інтерфейсів для передачі даних в мікроконтролерних системах. Його підтримка в ESP32 дозволяє просто з'єднувати плату з комп'ютером, налагоджувати програми, зчитувати дані від сенсорів або передавати команди. UART є незамінним інструментом на початковому етапі вивчення вбудованих систем.

**Порядок виконання роботи**

1. Складаємо в wokwi схему для дослідження.

В нас є резистор і світлодіод, один контакт якого підключений до виводів ESP. Кінцеву схему зображено на рис. 4.10. В цій схемі ми подаватимемо текстові команди «LED ON» та «LED OFF» для загоряння та згасання діоду відповідно через UART.

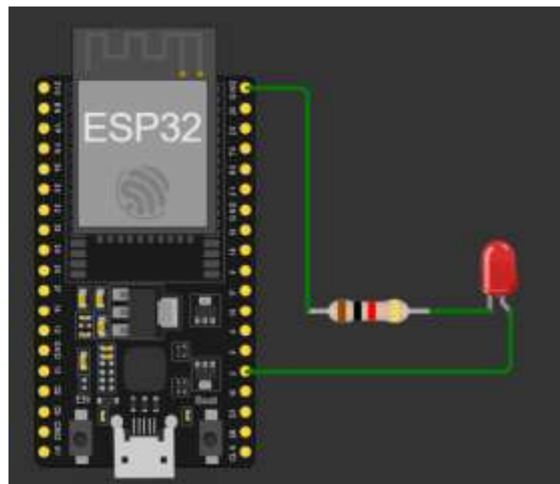



Рисунок 4.10 – Досліджувана схема в wokwi

2. Пишемо код нашої програми, яка працюватиме таким чином: за команди «LED ON» світлодіод загорятиметься, за команди «LED OFF» вимикатиметься, у разі іншої команди виводитиметься інформаційне повідомлення.

```
#define LED_PIN 2

void setup() {
  Serial.begin(115200);
  pinMode(LED_PIN, OUTPUT);
  Serial.println("ESP32 UART тест");
  Serial.println("Введіть команду: LED ON або LED OFF");
}

void loop() {
  if (Serial.available()) {
    String input = Serial.readStringUntil('\n');

    input.trim();

    if (input == "LED ON") {
      digitalWrite(LED_PIN, HIGH);
      Serial.println("Світлодіод увімкнено.");
    }
    else if (input == "LED OFF") {
      digitalWrite(LED_PIN, LOW);
      Serial.println("Світлодіод вимкнено.");
    }
    else {
      Serial.println("Невідома команда. Спробуйте: LED ON або LED OFF");
    }
  }
}
```

*Пояснення нашого коду*

1. Оголошення константи

`#define LED_PIN 2`

- Це директива препроцесора, яка задає ім’я LED_PIN для порту GPIO2.
- У ESP32 на платі Wokwi або DevKit цей пін зазвичай пов'язаний із вбудованим світлодіодом, який можна вмикати/вимикати.

2. Функція `setup()` – виконується один раз під час старту

`void setup() {`

```
    Serial.begin(115200);
    pinMode(LED_PIN, OUTPUT);
    Serial.println("ESP32 UART тест");
    Serial.println("Введіть команду: LED ON або LED
OFF");
  }
```

◆ `Serial.begin(115200);`

- Ініціалізує серійну комунікацію зі швидкістю 115200 біт/с.
- Це дозволяє обмінюватися даними між ESP32 та комп'ютером через USB або UART.

◆ `pinMode(LED_PIN, OUTPUT);`

- Встановлює пін 2 як вихідний – для керування світлодіодом.

◆ `Serial.println(...)`

- Виводить текст у `Serial Monitor`, по одному рядку.
- Це повідомлення про старт і підказка для користувача.

3. Функція `loop()` – основний цикл програми

```
void loop() {
  if (Serial.available()) {
    String input = Serial.readStringUntil('\n');

    input.trim();

    if (input == "LED ON") {
      digitalWrite(LED_PIN, HIGH);
      Serial.println("Світлодіод увімкнено.");
    }
    else if (input == "LED OFF") {
      digitalWrite(LED_PIN, LOW);
      Serial.println("Світлодіод вимкнено.");
    }
    else {
      Serial.println("Невідома                команда.
Спробуйте: LED ON або LED OFF");
    }
  }
}
```

◆ `Serial.available()`

- Перевіряє, чи є вхідні дані у буфері UART. Якщо так, програма продовжує зчитування.

◆ `Serial.readStringUntil('\n')`

- Зчитує рядок тексту до символу нового рядка (\n), тобто до натискання Enter.

- Повертає результат як `String`.

❖ `input.trim()`

- Видаляє всі пробіли на початку та в кінці рядка, а також `\r`, `\n`.
- Це важливо, бо інакше порівняння рядків не буде працювати правильно.

❖ `if (input == "LED ON") ...`

- Порівнює отриманий текст із відомими командами:
  - Якщо "LED ON" – увімкнути світлодіод через `digitalWrite(HIGH)` і вивести відповідь.
  - Якщо "LED OFF" – вимкнути через `digitalWrite(LOW)`.

❖ `else`

- Якщо рядок не збігається з відомими командами – вивести повідомлення про помилку.

3. Досліджуємо результати роботи нашого коду та схеми.

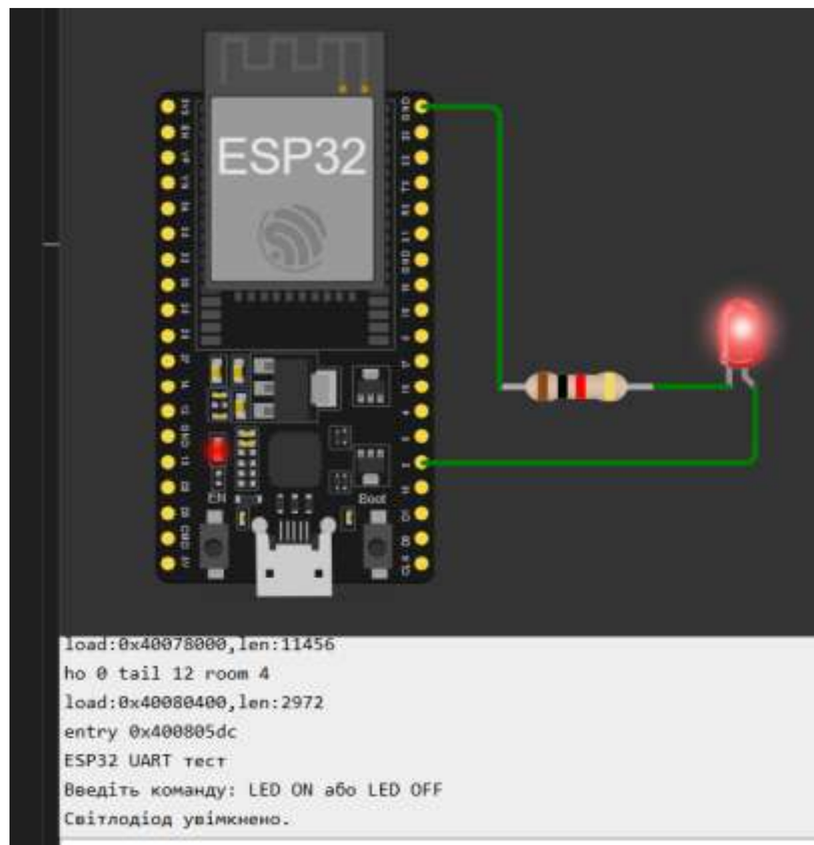



Рисунок 4.11 – Результат моделювання ввімкнення діода

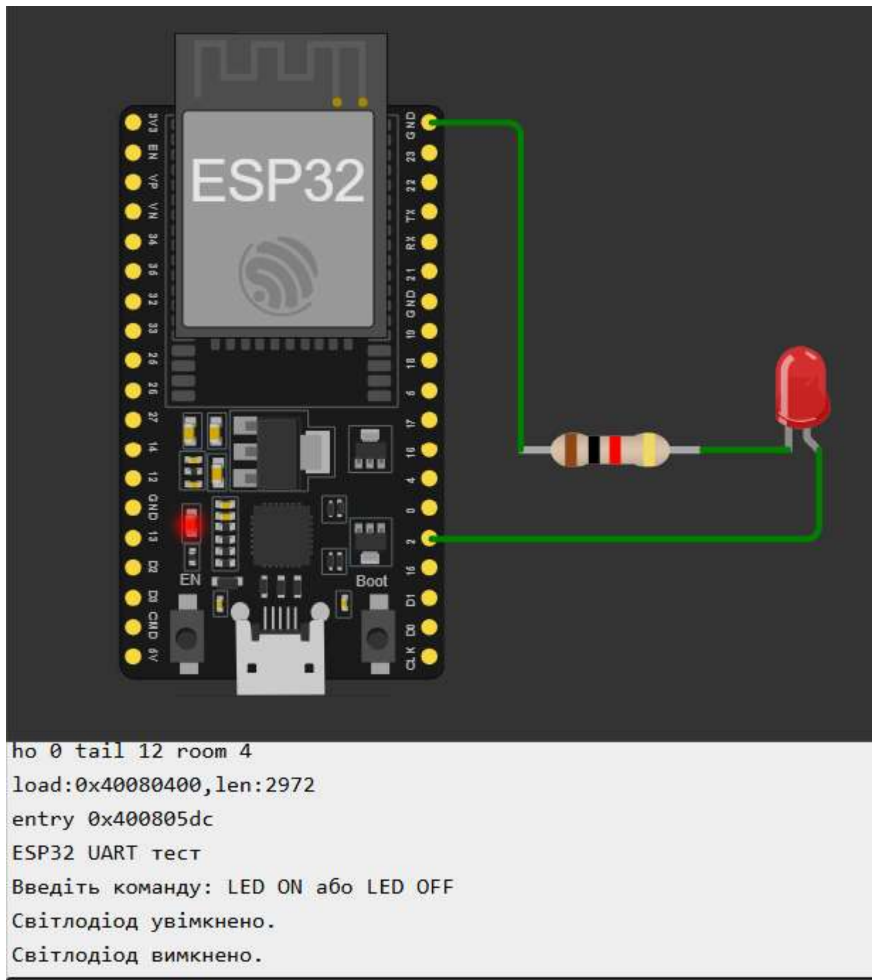



Рисунок 4.12 – Результат моделювання вимкнення діода

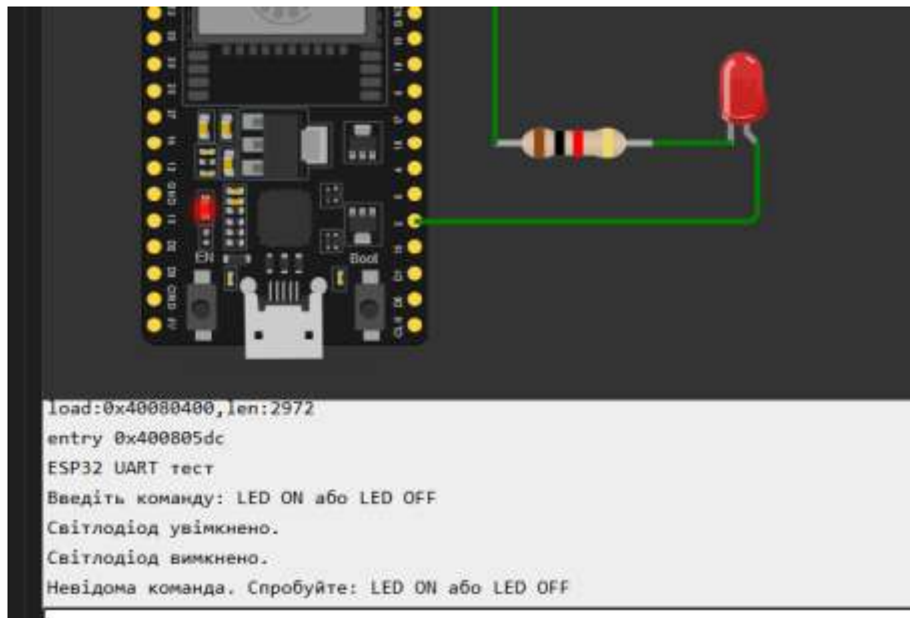



Рисунок 4.13 – Результат моделювання введення неправильної команди

3. Складаємо аналогічну схему на макетній платі з ESP32 (рис. 4.14) та проводитимемо її дослідження (рис. 4.15 – 4.17).

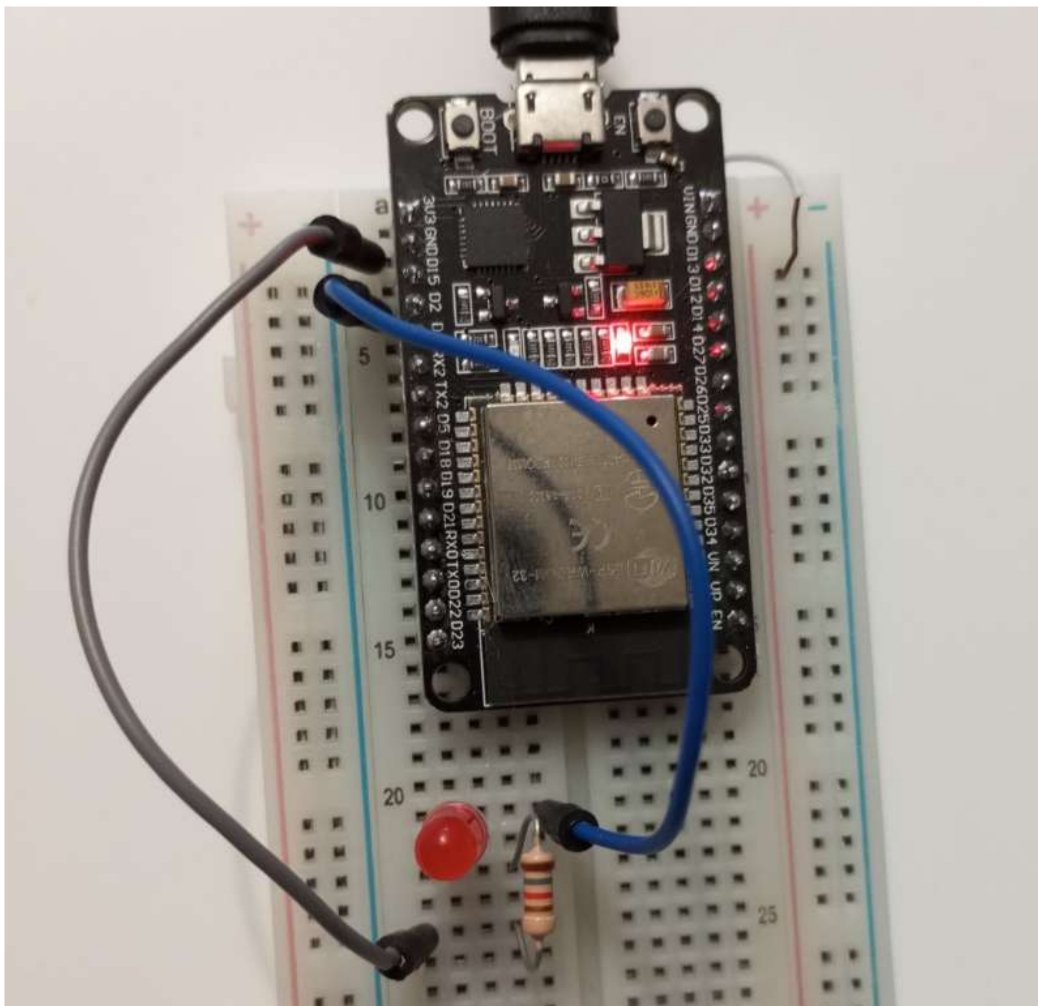


Рисунок 4.14 – Досліджувана схема на макетній платі

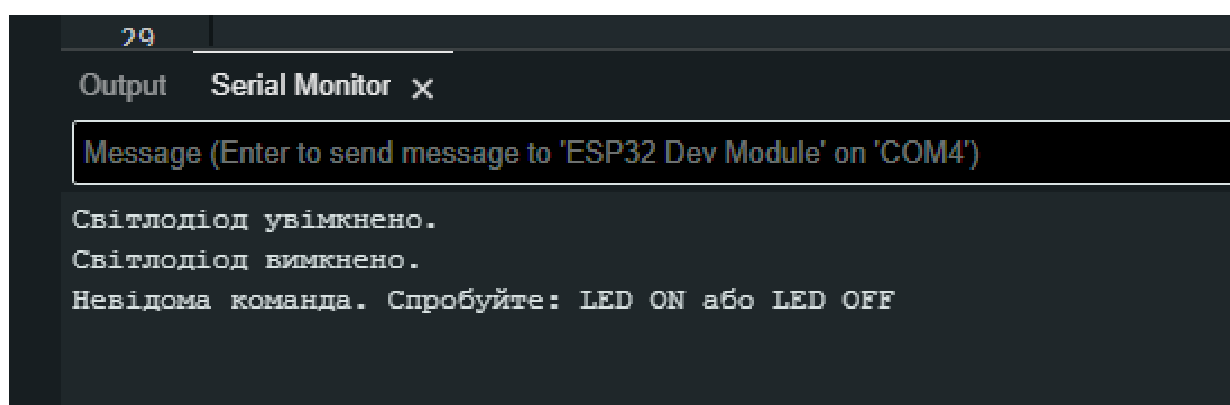
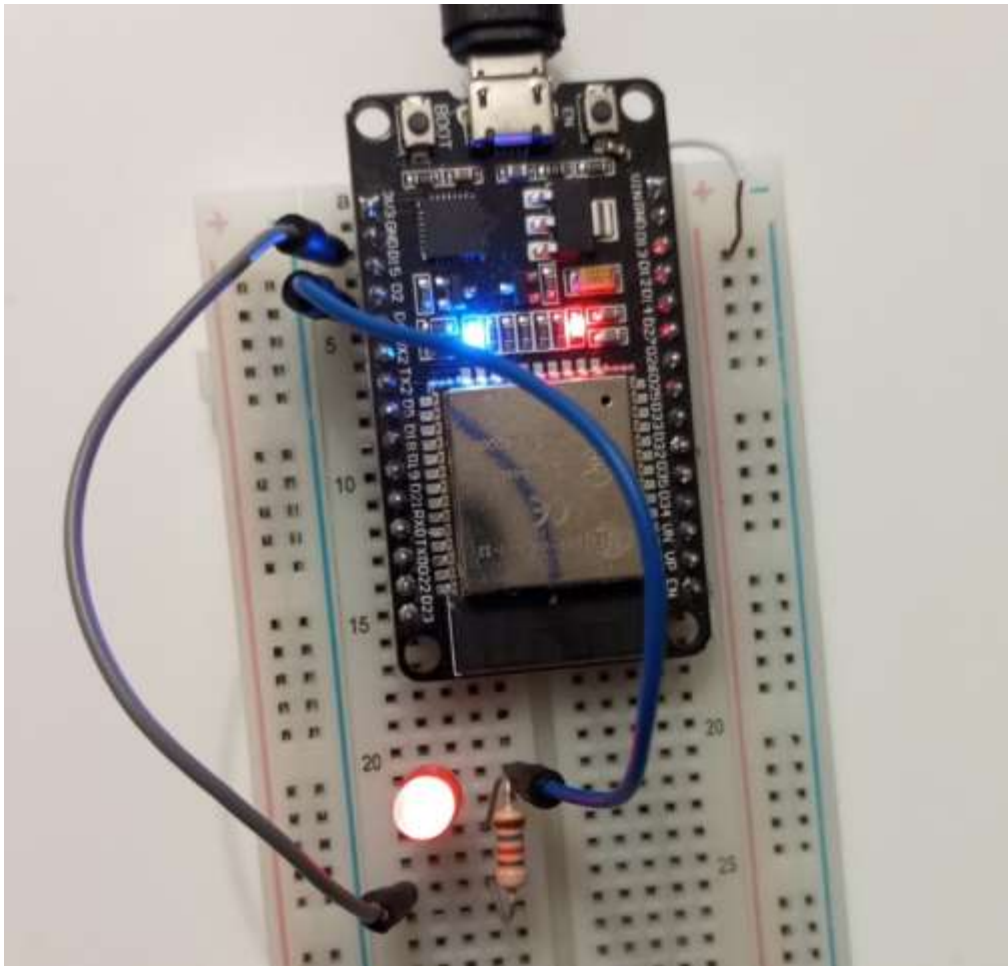



Рисунок 4.15 – Результат моделювання введення неправильної команди



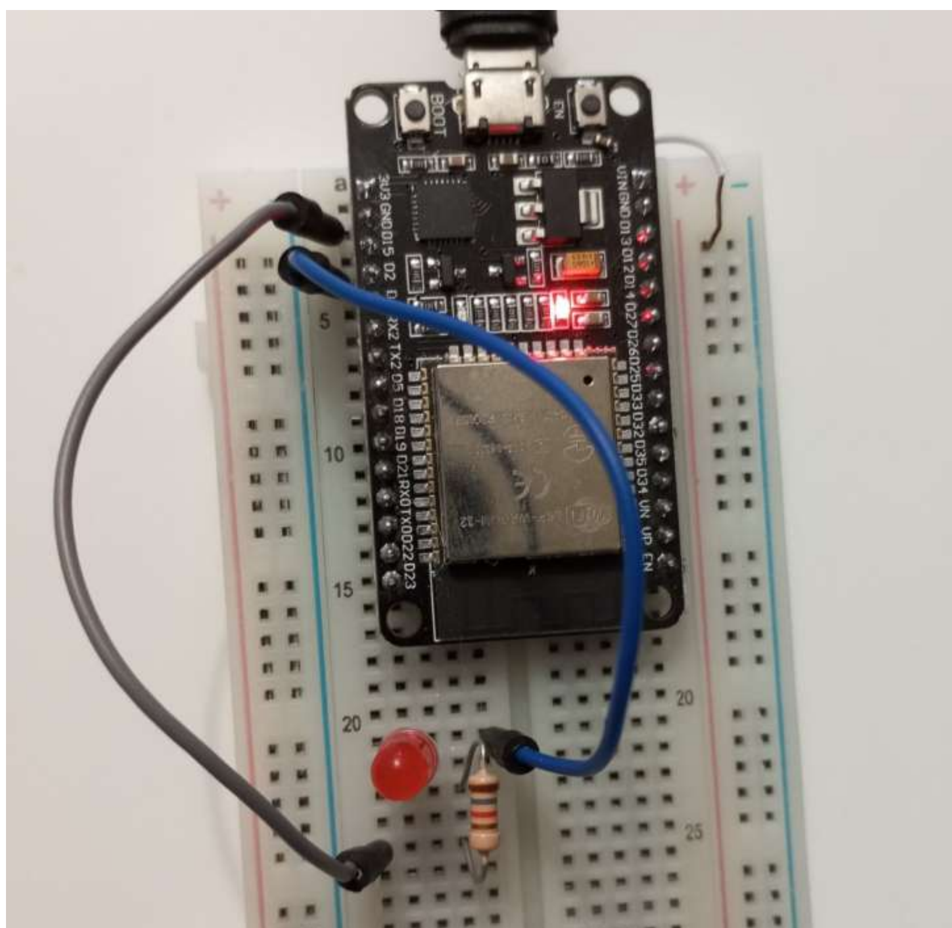
```
24      else {
25        Serial.println("Невідома команда
26      }
27    }
28  }
29
```

Output    Serial Monitor ×

Message (Enter to send message to 'ESP32 Dev Module' on

Світлодіод увімкнено.

Рисунок 4.16 – Результат моделювання ввімкнення діода



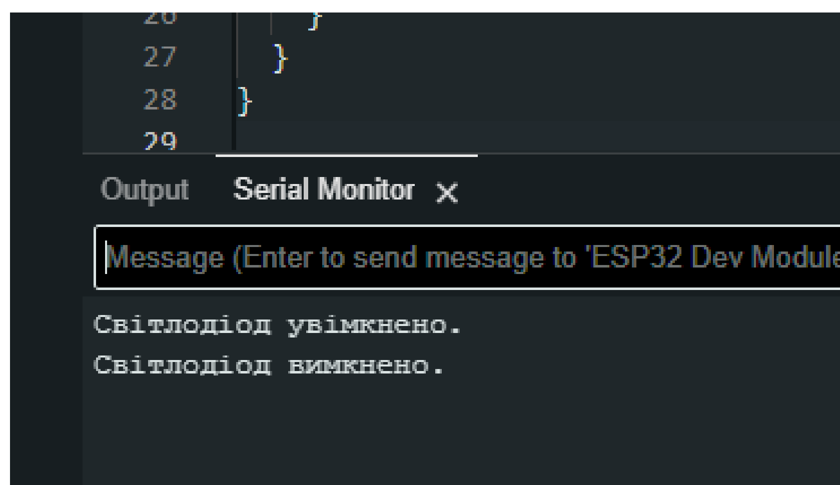



Рисунок 4.17 – Результат моделювання вимкнення діода

4. Таким чином, у ході лабораторної роботи було розглянуто принципи роботи UART (Universal Asynchronous Receiver/Transmitter) — одного з базових інтерфейсів асинхронної серійної передачі даних.

Було вивчено, як ініціалізувати серійний порт на мікроконтролері ESP32 за допомогою функції Serial.begin() та як відправляти і приймати дані через Serial Monitor у середовищі Arduino IDE або платформі Wokwi.

Реалізовано базову систему обміну командами між ESP32 та користувачем, що дозволяє керувати вбудованим світлодіодом на основі введених команд (LED ON, LED OFF).

Засвоєно роботу з функціями:

- Serial.available() – для перевірки наявності вхідних даних;
- Serial.readStringUntil() – для зчитування рядків до символу нового рядка;
- Serial.println() – для виведення інформації у термінал.

Було закріплено практичні навички перетворення текстової інформації на керівні дії в мікроконтролері, що є важливим у проєктах із віддаленим чи автоматизованим керуванням пристроями.

Робота є основою для подальшого використання UART у взаємодії з іншими пристроями (GPS-модулі, Bluetooth, GSM, комп'ютери) та побудови більш складних систем введення/виведення.

## Варіанти завдань

1. Виводити у Serial Monitor фразу «Лабораторна робота 4. UART» кожні 2 секунди.
2. Реалізувати лічильник секунд з моменту запуску мікроконтролера.
3. Створити програму, яка очікує введення числа з клавіатури Serial Monitor і виводить його квадрат.
4. Реалізувати обробку текстової команди "ON"/"OFF" – за "ON" вмикається світлодіод, за "OFF" – вимикається.
5. Реалізувати команду, яка виводить допомогу: у разі введення "HELP" – список доступних команд.
6. Написати програму, яка зчитує число та повертає повідомлення: «Парне» або «Непарне».
7. Створити простий калькулятор, який приймає два числа та оператор (+, -, *, /) і виводить результат.
8. Реалізувати введення з клавіатури і виведення на OLED дисплей введеного рядка.
9. Створити програму, яка підраховує кількість введених символів до натискання Enter.
10. Зчитувати з потенціометра значення напруги і виводити його у Serial Monitor.
11. Виводити повідомлення «Система активна» кожні 10 секунд.
12. Реалізувати зміну частоти моргання світлодіода за значенням, введеним через Serial Monitor.
13. Створити меню у Serial Monitor для вибору одного з трьох режимів роботи.
14. Реалізувати введення логіну/пароля через серійний порт і виведення повідомлення про успішний/невдалий вхід.

15. Виводити температуру з датчика (наприклад, TMP36 або симуляція) у градусах Цельсія.

16. Реалізувати фіксацію моменту натискання кнопки та надсилати повідомлення через UART.

17. Зчитувати координати з джойстика та виводити їх у реальному часі у консоль.

18. Реалізувати прийом команд типу «SET:100» для встановлення значення змінної.

19. Надсилати у Serial Monitor дані про кількість натискань кнопки.

20. Реалізувати логування – зберігати введені значення в масив і виводити їх за командою «SHOW».

## Зміст звіту

Звіт має містити:

1. Завдання.
2. Обгрунтування алгоритму програми.
3. Текст коду програми.
4. Електричні схеми реалізованих рішень (фото, скріни і под.).
5. Висновки за результатами проведених досліджень.

## Контрольні запитання

1. Що таке UART і яка його основна функція?
2. Яка різниця між UART, SPI та I2C?
3. Які виводи ESP32 за замовчуванням використовуються для UART (TX і RX)?
4. Що таке Serial Monitor і для чого він використовується?
5. Яка команда в Arduino використовується для ініціалізації UART?
6. Що означає параметр Serial.begin(115200)?
7. Якою командою надсилають текст у Serial Monitor?
8. Яка команда використовується для зчитування даних із UART?
9. У чому різниця між Serial.print() та Serial.println()?
10. Як організувати прийом числового значення з клавіатури Serial Monitor?
11. Що буде, якщо в Serial.read() буфер порожній?
12. Як організувати очікування команди від користувача через UART?
13. Яка максимальна швидкість UART-порту на ESP32?
14. Як можна змінити швидкість передачі UART у програмі?
15. Яка функція дозволяє перевірити, чи є доступні дані в буфері UART?
16. Що таке буфер UART і як уникнути його переповнення?

17. Яким чином можна організувати спілкування між двома ESP32 через UART?

18. Чому важливо виставляти однакову швидкість передачі (baud rate) на обох пристроях UART?

19. Як підключити зовнішній UART-пристрій до ESP32?

20. Які є типові помилки під час використання серійної комунікації і як їх уникати?

# Лабораторна робота 5
## Передача даних по I2C на LCD1602

**Мета**. Ознайомитися з принципами роботи інтерфейсу I2C на мікроконтролері ESP32. Навчитися підключати LCD-дисплей (модель 1602 з модулем I2C) та передавати на нього значення, зчитані з аналогового входу. Реалізувати виведення напруги на екран у вольтах з використанням потенціометра.

### Основні теоретичні відомості

Рідкокристалічний дисплей (РК-дисплей) – це плоский дисплей, у якому для формування зображення використовуються властивості рідких кристалів. На відміну від традиційних електронно-променевих трубок (ЕПТ), РК-дисплеї не випромінюють світло самі по собі, а працюють завдяки підсвітці або відбитому світлу.

РК-дисплей складається з двох прозорих панелей, між якими знаходиться шар рідких кристалів. Ці кристали змінюють свою орієнтацію під впливом електричного поля, змінюючи тим самим поляризацію світла. Це дозволяє контролювати проходження світла через дисплей і формувати зображення.

Основні компоненти:

- підсвітка (Backlight) – забезпечує освітлення для дисплея;
- поляризатори – два фільтри, які поляризують світло;
- скляні пластини – між ними розміщені рідкі кристали;
- електроди – створюють електричне поле для керування кристалами.

Кольорові фільтри (для кольорових РК-дисплеїв) – дозволяють формувати кольорове зображення.

*Переваги*

- Низьке енергоспоживання.
- Тонкий та легкий форм-фактор.
- Висока роздільна здатність.
- Відносно недороге виробництво.

*Недоліки*

- Обмежені кути огляду (у деяких типах).
- Менша контрастність порівняно з OLED.
- Залежність від підсвітки – не повністю «чорний» колір.

Одним із поширених прикладів РК-дисплея є LCD1602 – це рідкокристалічний дисплей із 2 рядками по 16 символів. Щоб зменшити кількість підключень, до нього часто додають модуль I2C (на основі мікросхеми PCF8574), який дозволяє керувати дисплеєм через дві лінії – SDA та SCL. Дисплей з I2C модулем зображено на рис. 5.1.

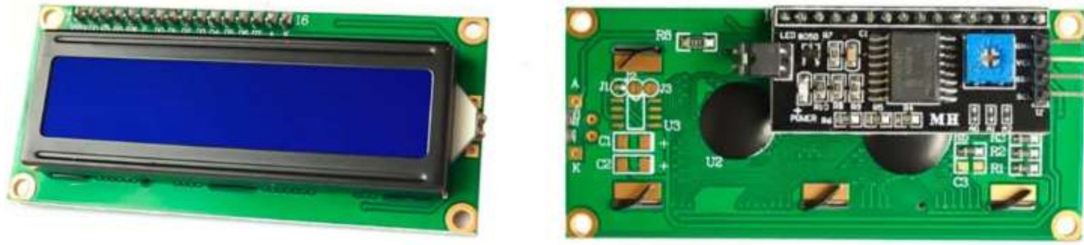


Рисунок 5.1 – Дисплей з I2C модулем.

Технічні характеристики дисплея LCD1602:
- символьний тип відображення;
- світлодіодна підсвітка;
- контролер HD44780;
- напруга живлення 5В;
- формат 16×2 символів;
- діапазон робочих температур від -20 °C до +70 °C,
- діапазон температур зберігання від -30 °C до +80 °C;
- кут огляду – 180 градусів.

Кожен із виводів LCD1602 має своє призначення (рис. 5.2):
1 – Земля GND;
2 – Живлення 5В;
3 – Встановлення контрастності дисплея;
4 – Команда, дані;
5 – Записування та читання даних;
6 – Увімкнути;
7…14 – Лінії даних;
15 – Плюс підсвічування;
16 – Мінус підсвічування.

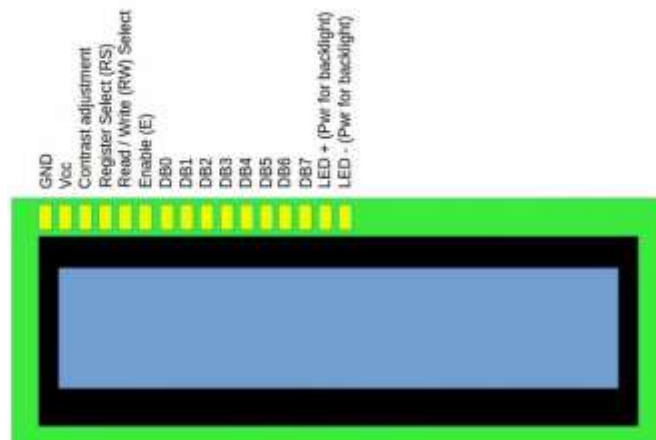



Рисунок 5.2 – Виводи дисплея LCD1602.

Для роботи з LCD1602 через паралельний інтерфейс використовується бібліотека LiquidCrystal.h, яка містить набір методів для керування символьними ЖК-дисплеями з контролером HD44780 або сумісним. Основні методи цієї бібліотеки наведено в таблиці 5.1 та

додаткові внутрішні методи – в таблиці 5.2. В таблиці 5.3 наведено згруповані методи бібліотеки LiquidCrystal.h.

*Ініціалізація об'єкта*

```
LiquidCrystal lcd(rs, enable, d4, d5, d6, d7);
```

*або для 8-бітного режиму:*

```
LiquidCrystal lcd(rs, enable, d0, d1, d2, d3, d4,
d5, d6, d7);
```

Таблиця 5.1 – Основні методи бібліотеки LiquidCrystal.h

| Метод | Опис |
|---|---|
| `begin(cols, rows)` | Ініціалізація дисплея з вказаною кількістю стовпців і рядків. |
| `clear()` | Очищення екрана та повернення курсору в позицію (0,0). |
| `home()` | Повернення курсору в позицію (0,0) без очищення. |
| `setCursor(col, row)` | Встановлення курсору в позицію `col`, `row`. |
| `print(data)` | Вивід тексту, числа або символу на екран. |
| `write(data)` | Виведення одного символу або байта (ASCII). |
| `createChar(index, data[])` | Створення власного символу у CGRAM (індекси 0–7). |
| `cursor()` | Увімкнення видимого курсору (підкреслення). |
| `noCursor()` | Приховання курсору. |
| `blink()` | Увімкнення миготіння курсору. |
| `noBlink()` | Вимкнення миготіння. |
| `display()` | Увімкнення дисплея (після `noDisplay()`). |
| `noDisplay()` | Вимкнення дисплея (без втрати даних). |
| `scrollDisplayLeft()` | Прокрутка всього дисплея вліво. |
| `scrollDisplayRight()` | Прокрутка всього дисплея вправо. |
| `leftToRight()` | Текст зліва направо (за замовчуванням). |
| `rightToLeft()` | Текст справа наліво. |
| `autoscroll()` | Автоматична прокрутка тексту у разі досягнення межі. |
| `noAutoscroll()` | Вимкнення автопрокрутки. |
| `setRowOffsets(row0, row1, row2, row3)` | Встановлення зсувів для дисплеїв із нестандартною пам’яттю. |

Таблиця 5.2 – Додаткові внутрішні методи бібліотеки LiquidCrystal.h

| Метод | Призначення |
|---|---|
| `command(uint8_t value)` | Відправлення команди дисплею. |
| `send(uint8_t, uint8_t)` | Передача даних або команд. |
| `write4bits(uint8_t)` | Передача 4-бітних даних (4-бітний режим). |
| `write8bits(uint8_t)` | Передача 8-бітних даних (8-бітний режим). |
| `pulseEnable()` | Генерація імпульсу enable для підтвердження передачі. |

Таблиця 5.3 – Таблиця методів бібліотеки LiquidCrystal_PCF8574

| Категорія | Методи |
|---|---|
| Ініціалізація | `begin()`, `setRowOffsets()` |
| Екран | `clear()`, `home()`, `setCursor()` |
| Текст | `print()`, `write()`, `createChar()` |
| Курсор/відображення | `cursor()`, `noCursor()`, `blink()`, `noBlink()`, `display()`, `noDisplay()` |
| Прокрутка/напрям | `scrollDisplayLeft()`, `scrollDisplayRight()`, `leftToRight()`, `rightToLeft()`, `autoscroll()`, `noAutoscroll()` |

Приклад використання:

```
#include <LiquidCrystal.h>

// RS, EN, D4, D5, D6, D7
LiquidCrystal lcd(12, 11, 5, 4, 3, 2);

void setup() {
  lcd.begin(16, 2);
  lcd.print("Привіт, Arduino!");
}

void loop() {
  lcd.setCursor(0, 1);
  lcd.print(millis() / 1000);
}
```

Бібліотека LiquidCrystal.h – ідеальний вибір для дисплеїв без I2C модуля або якщо потрібне повне ручне керування пінами.

Перш ніж обговорювати підключення дисплея через I2C-перехідник, коротко поговоримо про сам протокол I2C.

I2C / IIC (Inter-Integrated Circuit) – це протокол, що спочатку створювався для зв'язку інтегральних мікросхем всередині електронного пристрою. Розробка належить фірмі Philips. В основі I2C протоколу є використання 8-бітної шини, яка потрібна для зв'язку блоків в електроніці, що управляє, і системі адресації, завдяки якій можна спілкуватися по одних і тих самих проводах з декількома пристроями. Ми просто надаємо дані то одному, то іншому пристрою, додаючи до пакетів даних ідентифікатор потрібного елемента.

Як показано на рис. 5.3, на шині можуть бути присутні головний пристрій, який називається master (ведучий, контролер) та декілька підлеглих пристроїв, які називаються slave (ведений, периферійний пристрій).

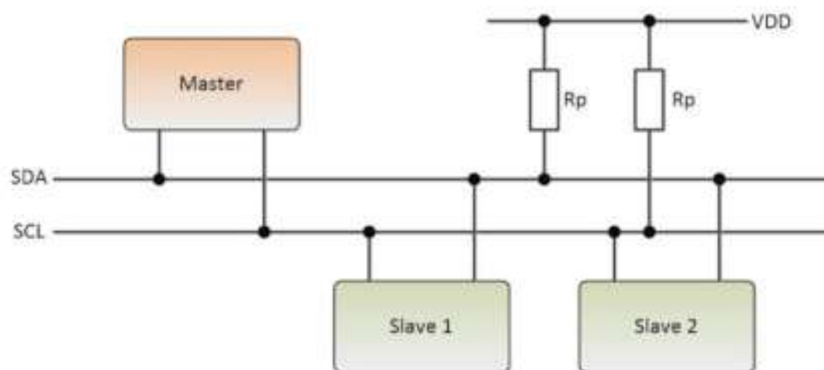



Рисунок 5.3 – Схема з’єднання пристроїв

Повідомлення в I2C протоколі (рис. 5.4) розбиваються на два типи кадрів: кадр адреси, де контролер (ведучий) надсилає один або кілька кадрів даних периферійному (веденому) пристрою, на який надсилається повідомлення. Ці кадри являють собою 8-бітні повідомлення даних, що передаються від контролера до периферійного пристрою або навпаки.

Повідомлення поділяються на два типи: читання або запису. Вони складаються з таких бітів:

1. Умова старту;
2. 7 або 8 бітів адреси;
3. Біт R/W вказує, чи ведений пристрій має тепер отримувати (0), чи передавати (1);
4. Біти даних;
5. Біти підтвердження/непідтвердження (ASK/NASK);
6. Умова зупинення.

Рисунок 5.4 – Структура I2C повідомлення

Розглянемо часові діаграми I2C протоколу (рис. 5.5 – 5.7).

Дані розміщуються на лінії даних SDA тоді, коли на SCL є низький рівень, і дані вичитуються, коли SCL стає високим.

Ведучий (Master) пристрій починає зв'язок, видаючи умову старту і потім надсилає унікальну 7-бітну адресу веденого (slave) пристрою, де старший біт (MSB) є першим. Восьмий біт після початку R/W вказує, чи ведений пристрій має тепер приймати (0) чи передавати (1). За цим йде біт ACK, надісланий веденим, підтверджуючи отримання попереднього байта. Потім передавач (ведучий або ведений, залежно від біта R/W) передає байт даних, починаючи з MSB. У кінці байта приймач (незалежно від того, ведучий або ведений) видає новий біт ACK. Цей 9-бітовий шаблон повторюється, якщо потрібно передати більше байтів.

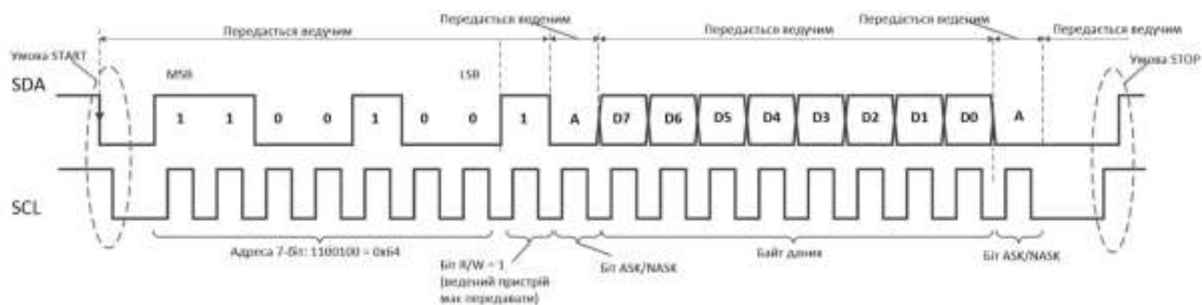



Рисунок 5.5 – Успішна передача байта запису I2C

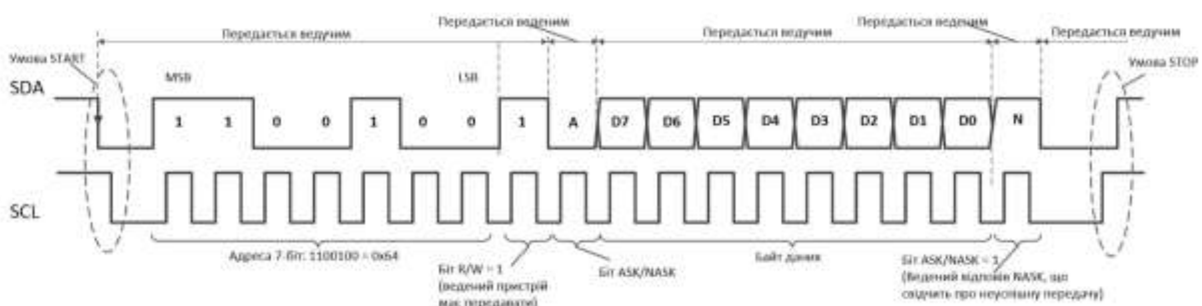



Рисунок 5.6 – Неуспішна передача байта запису I2C

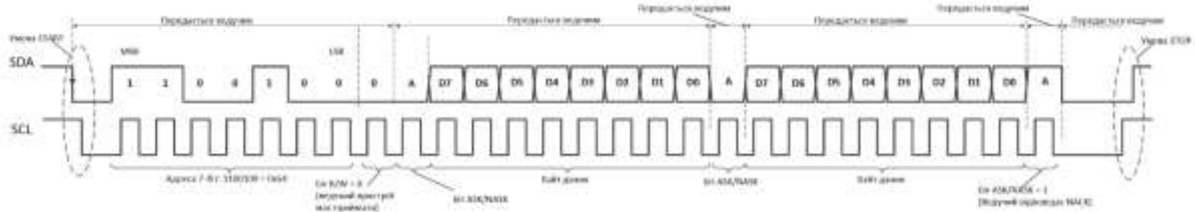



Рисунок 5.7 – Успішне читання I2C

У транзакції *запису* (ведений отримує дані), коли ведучий пристрій закінчив передачу всіх байтів даних, він відстежує останній біт ACK, а потім видає умову зупинення. У транзакції *читання* (ведений передає дані) ведучий пристрій не підтверджує останній отриманий байт (відповідає з NACK). Потім ведучий видає умову зупинення.

Іноді важливо, щоб контролеру було дозволено обмінюватися кількома повідомленнями за один раз, не дозволяючи втручатися іншим контролерам на шині. В цьому випадку умову початку можна повторити під час передачі без необхідності попереднього завершення умовою зупинення. Це особливий випадок, який називається повторним стартом, що показано на рис. 5.8.

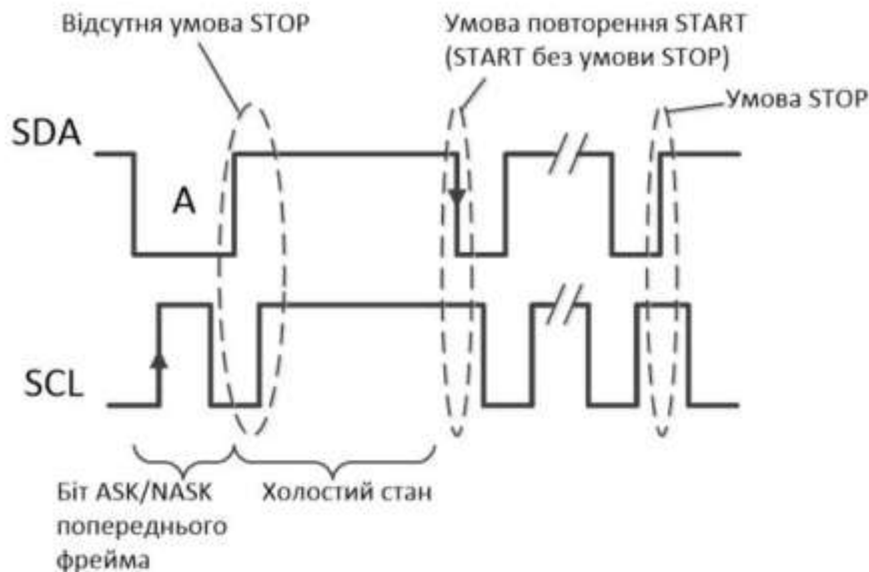



Рисунок 5.8 – Повторний старт

На апаратному рівні всі пристрої I2C мають сигнальні виводи з «відкритим колектором» або «відкритим стоком» (залежно від реалізації) і формування сигналів відбувається через «монтажне І».

Передача та приймання сигналів здійснюється притисканням лінії в логічний 0, а в логічну 1 сигнал встановлюється сам шляхом підтягувальних резисторів Rp (рис. 5.9). Вони обов'язкові завжди!

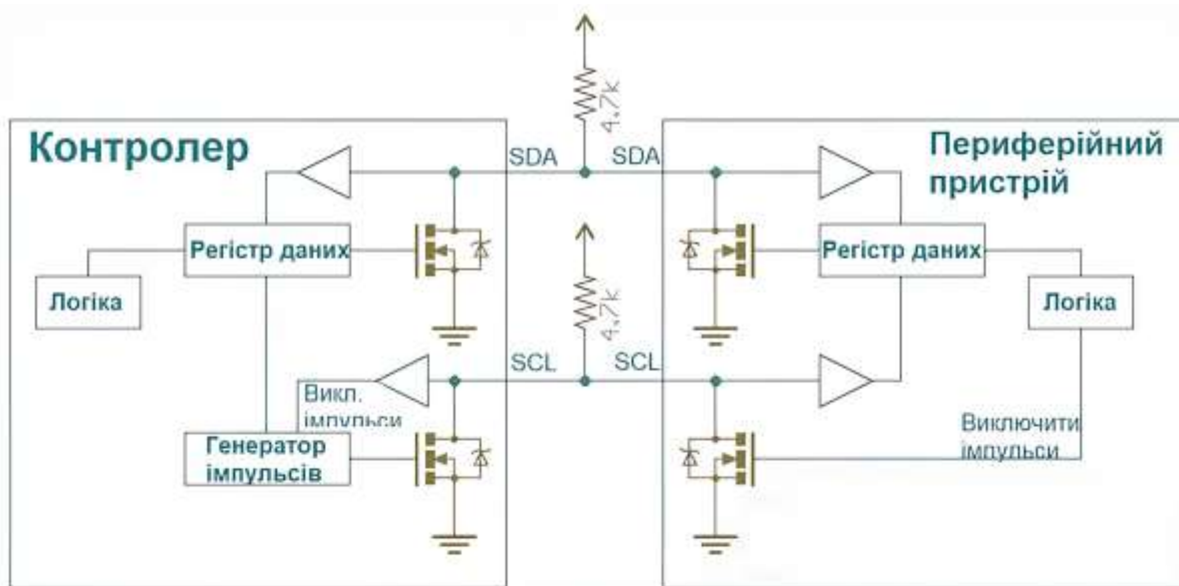



Рисунок 5.9 – Апаратна реалізація I2C

Біти синхронізуються на спадних фронтах тактувального сигналу SCL. Стандартна швидкість передачі даних становить 100 кбіт/с, у швидкому режимі – 400 кбіт/с.

Оптимальним номіналом підтягувального резистора є опір в діапазоні від 1 до 10 кОм. Переважно ставлять опір 4,7 кОм.

Чим більший опір резистора, тим довше лінія відновлюється в одиницю (іде перезаряд паразитної ємності між проводами) і тим сильніше завалюються фронти імпульсів, а значить швидкість передачі спадає. Також високий опір резистора приводить до зменшення завадостійкості. Саме тому в I2C швидкість передачі набагато нижча, ніж у SPI.

Чим більша відстань передачі і чим більше пристроїв на лінії, тим має бути менший опір резистора.

Також під час вибору резистора необхідно враховувати паразитну ємність між лініями шини. Чи більша ємність, тим менший має бути опір резистора.

Найпростіша схема I2C може містити лише один «відомий» пристрій (найчастіше це мікроконтролер) та кілька «ведених» (наприклад, дисплей LCD). Кожен пристрій має адресу в діапазоні від 7 до 127. Двох пристроїв з однаковою адресою в одній схемі не може бути.

Плата ESP32 підтримує I2C на апаратному рівні.

У роботі I2C можна виділити кілька переваг:

1. Для роботи потрібно лише 2 лінії – SDA (лінія даних) та SCL (лінія синхронізації).

2. Підключення великої кількості провідних приладів.

3. Зменшення часу розробки.

4. Для керування всім набором пристроїв потрібен лише один мікроконтролер.

5. Можливе число мікросхем, що підключаються до однієї шини, обмежується тільки граничною ємністю.

6. Висока надійність передавання даних завдяки спеціальному фільтру подавлення імпульсних завад, вбудованому в мікросхему.

7. Проста процедура діагностики збоїв, що виникають, швидке налагодження несправностей.

8. Шина вже інтегрована в багатьох розробницьких платах, тому не потрібно додатково розробляти шинний інтерфейс.

*Недоліки*

1. Існує ємнісне обмеження на лінії – 400 пФ.

2. Важке програмування контролера I2C, якщо на шині є кілька різних пристроїв.

3. У випадку великої кількості пристроїв виникають труднощі локалізації збою, якщо один з них помилково встановлює стан низького рівня.

I2C (Inter-Integrated Circuit) – послідовний протокол із двома лініями:

SDA – передача даних;

SCL – синхронізація (тактування).

ESP32 має декілька I2C-шин. За замовчуванням:

GPIO21 – SDA

GPIO22 – SCL

*Схема підключення*

LCD1602 (з I2C модулем):

VCC → 5V (або 3.3V);

GND → GND;

SDA → GPIO21 (ESP32);

SCL → GPIO22 (ESP32).

Потенціометр:

Один кінець до 3.3V;

Інший кінець до GND;

Центральний (вихід) до GPIO34.

Для підключення РК-екрана через I2C інтерфейс необхідний спеціальний окремий I2C перехідник, який зображено на рис. 5.10.

У випадку використання цього перехідника, рідкокристалічний монітор з підтримкою I2C підключається до плати за допомогою чотирьох провідників – два провідники для даних, два провідники – для живлення.

Вивід GND підключається до GND на платі.

Вивід VCC – на 5V.

SCL, SDA – передача даних через I2C.

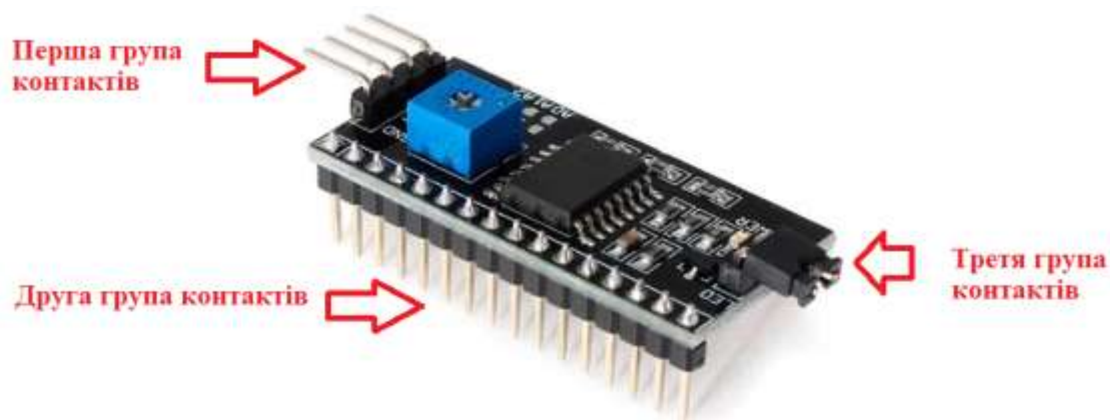



Рисунок 5.10 – I2C перехідник для РК-екрана LCD1602

Для роботи з таким перехідником необхідно використовувати бібліотеку LiquidCrystal_PCF8574.

Основні методи бібліотеки LiquidCrystal_PCF8574 аналогічні методам бібліотеки LiquidCrystal, але з додатковими функціями для роботи з розширювачем портів PCF8574.

Бібліотека LiquidCrystal_PCF8574 – це розширення бібліотеки LiquidCrystal для роботи з різними модулями з ЖК-дисплеями, які використовують розширювач портів PCF8574 або аналогічні мікросхеми. Ця бібліотека дозволяє контролювати ЖК-дисплей через шину I2C з використанням лише двох пінів (SDA та SCL).

Наведемо повний перелік методів бібліотеки LiquidCrystal_PCF8574 (версія 2.2.0) з коротким описом для кожного:

*Конструктор*

- `LiquidCrystal_PCF8574(TwoWire &wire = Wire, uint8_t address = 0x27, uint8_t cols = 16, uint8_t rows = 2, uint8_t charsize = LCD_5x8DOTS);`

Ініціалізує об’єкт з I²C-шиною, адресою та розміром дисплею.

*Налаштування екрана*

- `begin(uint8_t cols, uint8_t rows, uint8_t charsize = LCD_5x8DOTS);`

Ініціалізація пам’яті дисплея.

- `isConnected();`

Перевіряє зв’язок з PCF8574, повертає `true/false`.

*Виведення тексту та курсор*

- `clear();` – очищення екрана (курсор в позиції 0,0).
- `home();` – пересування курсору в 0,0 без очищення.
- `setCursor(uint8_t col, uint8_t row);` – встановлення курсору на задану позицію.
- `print(const char *str);` – виведення тексту.

• `print(char c); print(unsigned long, int base = DEC);` – виведення символу або числа в заданій основі.

• `write(uint8_t byte);` – виведення байта (ASCII або символу).

*Кастомні символи*

• `createChar(uint8_t index, uint8_t data[8]);` – формування власного символу за масивом.

• `createChar(uint8_t index, const uint8_t data[8] PROGMEM);` – версія для AVR з PROGMEM.

*Параметри відображення*

- `display();` – увімкнути відображення.
- `noDisplay();` – вимкнути (фон підсвітки лишається).
- `cursor();` – показати курсор.
- `noCursor();` – сховати курсор.
- `blink();` – курсор блимає.
- `noBlink();` – без миготіння.

*Підсвітка*

- `backlight();` – увімкнути підсвітку.
- `noBacklight();` – вимкнути підсвітку.
- `setBacklight(bool value);` – вмикає або вимикає (True/False).

*Робота з I²C*

• `begin();` – альтернативний початок з раніше вказаним розміром дисплею.

• `init();` – старий варіант ініціалізації (застосовується у ключах).

• `write4bits(uint8_t value);` – внутрішній метод передачі 4 біт.

• `expanderWrite(uint8_t data);` – низькорівнева передача через PCF8574.

• `pulseEnable(uint8_t data);` – створює імпульс "enable" для LCD.

*Додаткові внутрішні методи* (не призначені для користування прямо)

• `command(uint8_t value);` – надсилання команди дисплею.

• `send(uint8_t value, uint8_t mode);` – передача даних/команди.

• Обробка `Wire` (I²C): `wire->beginTransmission(address)...wire->endTransmission()` іменовані у `expanderWrite()`, `init()`.

В таблиці 5.4 наведено згруповані методи бібліотеки LiquidCrystal_PCF8574.

Таблиця 5.4 – Таблиця методів бібліотеки LiquidCrystal_PCF8574

| Група | Методи |
|---|---|
| Ініціалізація | `LiquidCrystal_PCF8574(...)`, `begin()`, `init()`, `isConnected()` |
| Екран | `clear()`, `home()`, `setCursor()` |
| Вивід | `print()`, `write()` |
| Користув. символи | `createChar()` |
| Курсор/подв. ефекти | `display()`, `noDisplay()`, `cursor()`, `noCursor()`, `blink()`, `noBlink()` |
| Підсвітка | `backlight()`, `noBacklight()`, `setBacklight()` |
| Низ. рівень I²C | `write4bits()`, `expanderWrite()`, `pulseEnable()`, `send()`, `command()` |

Приклад використання:

```
#include <Wire.h>
#include <LiquidCrystal_PCF8574.h>

LiquidCrystal_PCF8574 lcd(0x27); // Адреса модуля I2C

void setup() {
  lcd.begin(16, 2);        // Ініціалізація дисплея 16x2
  lcd.setBacklight(true); // Вмикаємо підсвітку
  lcd.setCursor(0, 0);
  lcd.print("Привіт, світ!");
}

void loop() {
  // Нічого не виконується
}
```

## Порядок виконання роботи

1. Складаємо в wokwi схему годинника реального часу з відображенням інформації на LCD1602.

Нам необхідно отримати сигнал з модуля реального часу та відобразити дату та час на LCD1602. Дисплей та модуль реального часу підключені через I2C інтерфейс.

Схему дослідження на платформі wokwi показано на рис. 5.11.

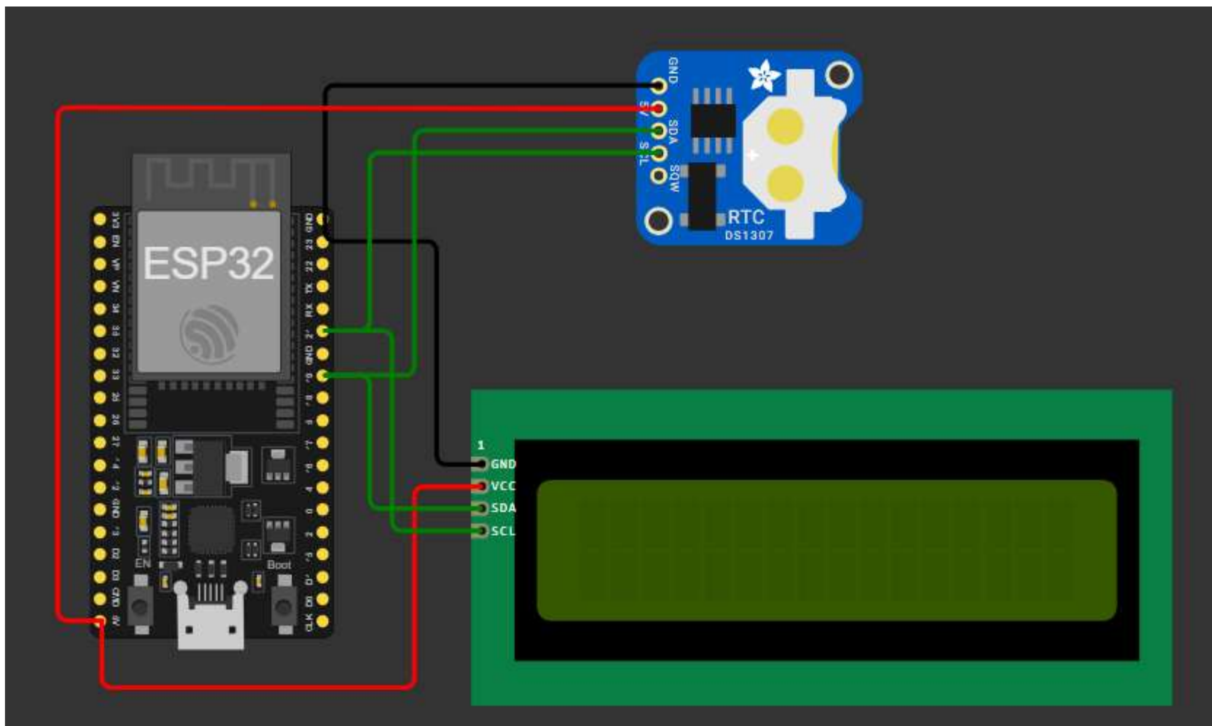



Рисунок 5.11 – Досліджувана схема годинника реально часу в wokwi

2. Пишемо код нашої програми.

```
#include <Wire.h>
#include <LiquidCrystal_PCF8574.h>
#include <RTClib.h>

// Ініціалізація LCD (адреса може бути 0x27 або 0x3F)
LiquidCrystal_PCF8574 lcd(0x27);
RTC_DS3231 rtc;

void setup() {
  Wire.begin(19, 21); // SDA, SCL для ESP32

  lcd.begin(16, 2);    // LCD 16x2
  lcd.setBacklight(true);
  lcd.clear();

  if (!rtc.begin()) {
```

```
    lcd.setCursor(0, 0);
    lcd.print("RTC isn't pres!");
    while (1); // Зупинити виконання
  }

  if (rtc.lostPower()) {
    lcd.setCursor(0, 0);
    lcd.print("RTC clear");
    rtc.adjust(DateTime(F(__DATE__), F(__TIME__))); // Автоматично з
компіляції
  }

  lcd.setCursor(0, 0);
  lcd.print("Clock is ready");
  delay(2000);
  lcd.clear();
}

void loop() {
  DateTime now = rtc.now();

  lcd.setCursor(0, 0);
  lcd.print("Time: ");
  printTwoDigits(now.hour());
  lcd.print(":");
  printTwoDigits(now.minute());
  lcd.print(":");
  printTwoDigits(now.second());

  lcd.setCursor(0, 1);
  lcd.print("Date: ");
  printTwoDigits(now.day());
  lcd.print(".");
  printTwoDigits(now.month());
  lcd.print(".");
  lcd.print(now.year());

  delay(1000);
}

void printTwoDigits(int number) {
  if (number < 10) lcd.print("0");
  lcd.print(number);
}
```

Пояснення нашого коду.

1. Підключення бібліотек

```
#include <Wire.h>                         // Для роботи з I2C
#include <LiquidCrystal_PCF8574.h>    // Для LCD1602 через I2C (PCF8574)
#include <RTClib.h>                    // Для модуля реального часу DS3231
```

2. Ініціалізація об'єктів

```
LiquidCrystal_PCF8574 lcd(0x27);      // LCD дисплей з адресою I2C 0x27
RTC_DS3231 rtc;                        // RTC об'єкт для роботи з модулем DS3231
```

3. Функція `setup()`

```
Wire.begin(19, 21); // Ініціалізує I2C-шину з пінів SDA=21, SCL=19 (для ESP32)
```

На ESP32 можна вказувати кастомні SDA/SCL, тому пін SDA – GPIO 21, а SCL — GPIO 19.

```
lcd.begin(16, 2);          // Ініціалізація LCD розміром 16 символів × 2 рядки
lcd.setBacklight(true);   // Включити підсвітку
lcd.clear();               // Очистити екран
```

Перевірка підключення RTC:

```
if (!rtc.begin()) {
  lcd.setCursor(0, 0);
  lcd.print("RTC isn't pres!");
  while (1); // Зупиняє виконання програми
}
```

Якщо модуль RTC не відповідає – виводиться повідомлення та програма зупиняється.

Перевірка втрати живлення (скидання):

```
if (rtc.lostPower()) {
  lcd.setCursor(0, 0);
  lcd.print("RTC clear");
  rtc.adjust(DateTime(F(__DATE__), F(__TIME__)));
}
```

- `rtc.lostPower()` – перевіряє, чи був скинутий час (наприклад, під час першого підключення).

• `rtc.adjust(...)` – встановлює час компіляції як поточний час (`__DATE__`, `__TIME__`).

Готовність дисплея:

```
lcd.setCursor(0, 0);
lcd.print("Clock is ready");
delay(2000);
lcd.clear();
```

Користувачеві показується повідомлення про готовність, а потім екран очищається.

4. Функція `loop()`

```
DateTime now = rtc.now(); // Отримуємо поточну
дату та час із модуля RTC
```

Вивід часу (перший рядок LCD):

```
lcd.setCursor(0, 0);
lcd.print("Time: ");
printTwoDigits(now.hour());
lcd.print(":");
printTwoDigits(now.minute());
lcd.print(":");
printTwoDigits(now.second());
```

Вивід дати (другий рядок LCD):

```
lcd.setCursor(0, 1);
lcd.print("Date: ");
printTwoDigits(now.day());
lcd.print(".");
printTwoDigits(now.month());
lcd.print(".");
lcd.print(now.year());
```

Дані виводяться у форматах `HH:MM:SS` і `DD.MM.YYYY`.

5. Функція `printTwoDigits()`

```
void printTwoDigits(int number) {
  if (number < 10) lcd.print("0");
  lcd.print(number);
}
```

Ця допоміжна функція додає ведучий нуль до чисел менших за 10. Наприклад, `9` буде виведено як `09`.

Затримка на оновлення

```
delay(1000);
```

Оновлення дисплея щосекунди.

Таким чином було використано такі елементи:

| Компонент | Призначення |
|---|---|
| `Wire` | Шина I2C між ESP32 ↔ LCD та RTC |
| `LiquidCrystal_PCF8574` | Підключення LCD1602 по I2C |
| `RTClib` | Бібліотека для модуля часу DS3231 |
| `rtc.begin()` | Перевірка доступності RTC |
| `rtc.now()` | Отримання поточного часу |
| `lcd.setCursor() / lcd.print()` | Виведення тексту на LCD |
| `printTwoDigits()` | Форматування чисел з ведучим нулем |

3. Досліджуємо результати роботи нашого коду та схеми. Результат наведено на рис. 5.12 та рис. 5.13.

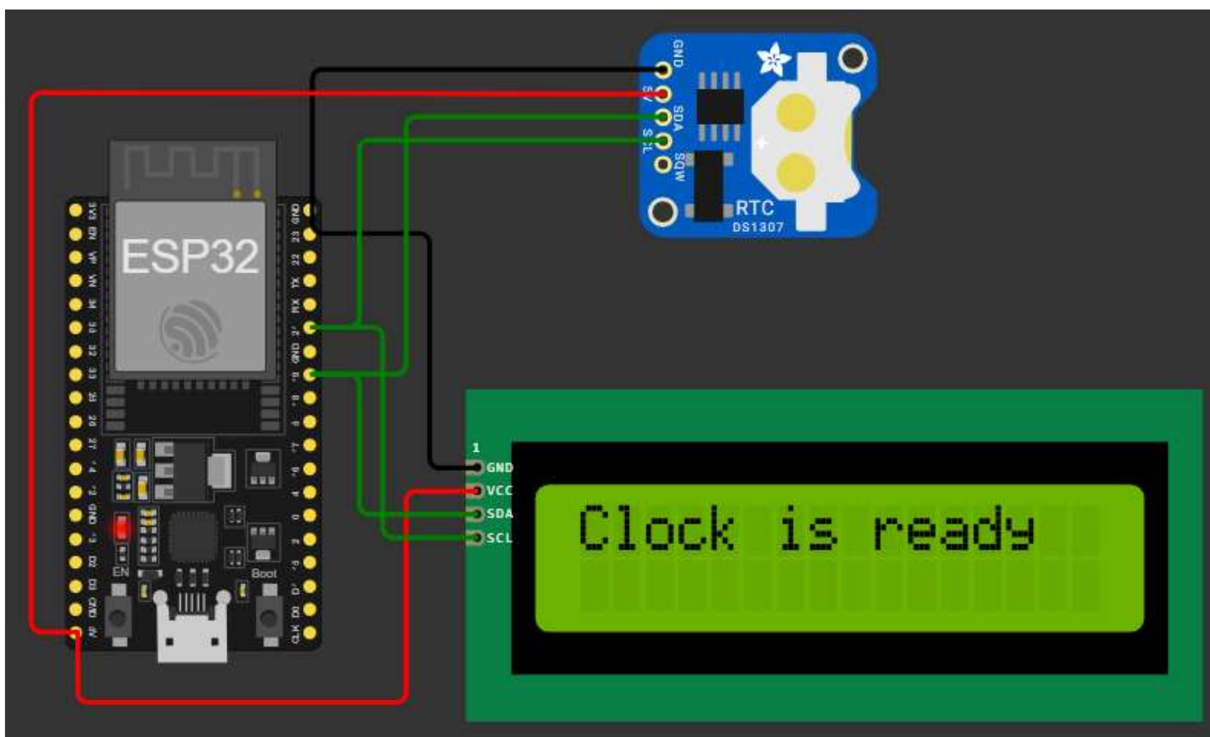



Рисунок 5.12 – Результат моделювання годинника

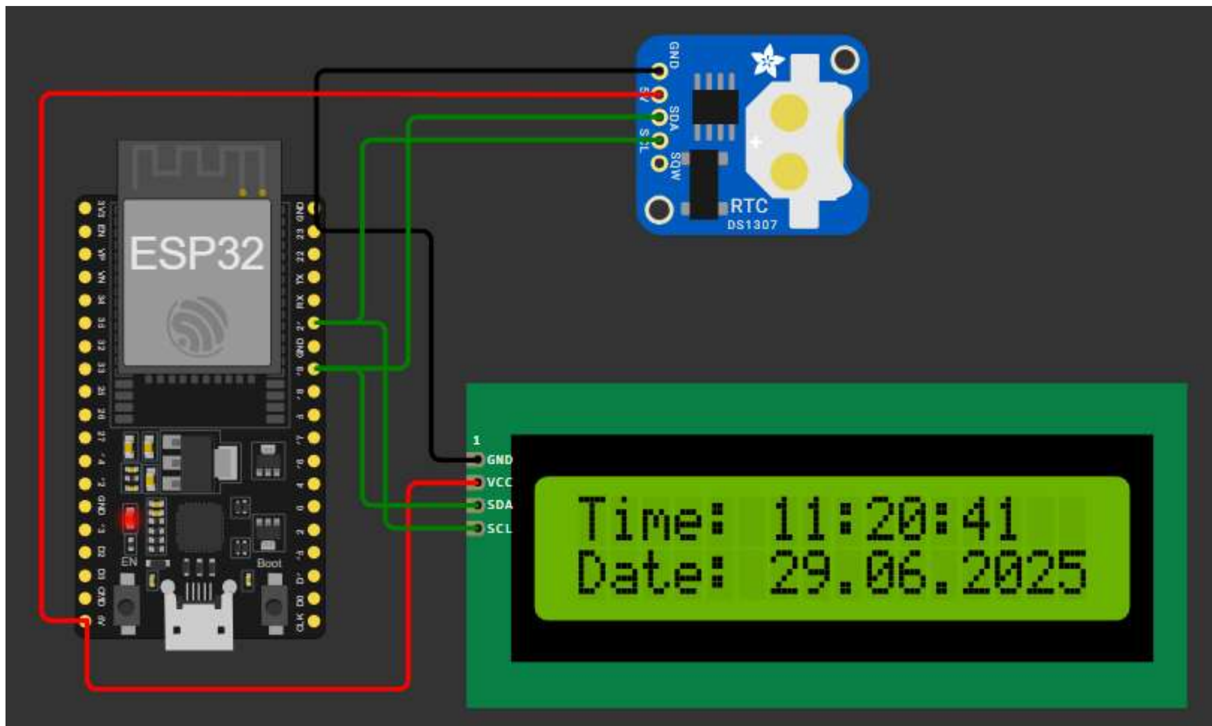



Рисунок 5.13 – Результат виведення інформації

4. Складаємо аналогічну схему на макетній платі з ESP32 (рис. 5.14). В нас лише зміниться модуль реального часу на DS1302, оскільки він наявний в лабораторії.

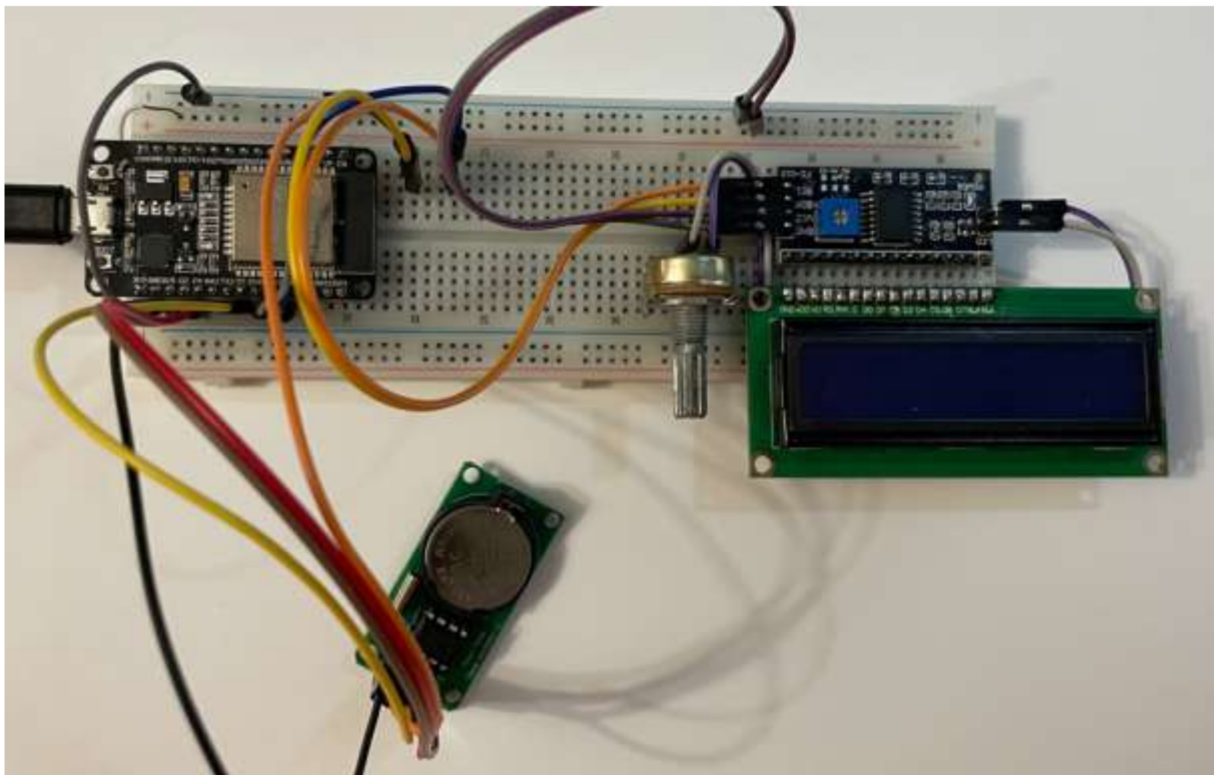


Рисунок 5.14 – Досліджувана схема, складена на макетній платі

5. Оскільки ми замінили модуль реального часу, то трішки і змінився код на такий:

```
// CONNECTIONS:
// DS1302 CLK/SCLK --> 5
// DS1302 DAT/IO   --> 4
// DS1302 RST/CE   --> 2
// DS1302 VCC --> 3.3v - 5v
// DS1302 GND --> GND

#include <Wire.h>
#include <RtcDS1302.h>
#include <LiquidCrystal_PCF8574.h>  // LCD через I2C

// RTC
ThreeWire myWire(4, 5, 2);  // IO, SCLK, CE
RtcDS1302<ThreeWire> Rtc(myWire);

// LCD1602 (I2C address — 0x27 або 0x3F, перевір вручну)
LiquidCrystal_PCF8574 lcd(0x27); // Якщо не працює — спробуй 0x3F

void setup() {
  Serial.begin(57600);

  // --- LCD INIT ---
  Wire.begin(19, 21); // Для ESP32 — SDA, SCL
  lcd.begin(16, 2);
  lcd.setBacklight(255);
  lcd.clear();
  lcd.setCursor(0, 0);
  lcd.print("Initializing...");
  delay(1000);
  lcd.clear();

  // --- RTC INIT ---
  Serial.print("compiled: ");
  Serial.print(__DATE__);
  Serial.println(__TIME__);

  Rtc.Begin();

  RtcDateTime compiled = RtcDateTime(__DATE__, __TIME__);
  printDateTime(compiled);
  Serial.println();

  if (!Rtc.IsDateTimeValid()) {
    Serial.println("RTC lost confidence in the DateTime!");
    Rtc.SetDateTime(compiled);
  }
```

```
  if (Rtc.GetIsWriteProtected()) {
    Serial.println("RTC was write protected, enabling writing now");
    Rtc.SetIsWriteProtected(false);
  }

  if (!Rtc.GetIsRunning()) {
    Serial.println("RTC was not actively running, starting now");
    Rtc.SetIsRunning(true);
  }

  RtcDateTime now = Rtc.GetDateTime();
  if (now < compiled) {
    Serial.println("RTC is older than compile time!  (Updating DateTime)");
    Rtc.SetDateTime(compiled);
  } else if (now > compiled) {
    Serial.println("RTC is newer than compile time. (this is expected)");
  } else if (now == compiled) {
    Serial.println("RTC is the same as compile time! (not expected but all is
fine)");
  }
}

void loop() {
  RtcDateTime now = Rtc.GetDateTime();

  printDateTime(now);
  Serial.println();

  if (!now.IsValid()) {
    Serial.println("RTC lost confidence in the DateTime!");
  }

  // --- LCD update ---
  lcd.setCursor(0, 0);
  lcd.print("Time ");
  print2digits(now.Hour()); lcd.print(":");
  print2digits(now.Minute()); lcd.print(":");
  print2digits(now.Second()); lcd.print(" ");

  lcd.setCursor(0, 1);
  lcd.print("Date ");
  print2digits(now.Day()); lcd.print(".");
  print2digits(now.Month()); lcd.print(".");
  lcd.print(now.Year());

  delay(1000);
}

// ------------------------------
// Допоміжні функції
```

```
#define countof(a) (sizeof(a) / sizeof(a[0]))

void printDateTime(const RtcDateTime& dt) {
  char datestring[26];

  snprintf_P(datestring,
             countof(datestring),
             PSTR("%02u/%02u/%04u %02u:%02u:%02u"),
             dt.Month(),
             dt.Day(),
             dt.Year(),
             dt.Hour(),
             dt.Minute(),
             dt.Second());
  Serial.print(datestring);
}

void print2digits(int number) {
  if (number < 10) lcd.print("0");
  lcd.print(number);
}
```

Давайте детально розглянемо написаний код.

Цей скетч реалізує годинник реального часу (RTC) на базі модуля DS1302, виводить поточну дату і час на LCD1602 через інтерфейс I²C, а також дублює їх у Serial Monitor. Нижче – докладний опис усіх частин коду.

🔌 *Підключення:*

```
// DS1302 CLK/SCLK --> 5
// DS1302 DAT/IO   --> 4
// DS1302 RST/CE   --> 2
// DS1302 VCC      --> 3.3V або 5V
// DS1302 GND      --> GND
```

📦 *Підключення бібліотек:*

```
#include <Wire.h>      // для I2C
#include <RtcDS1302.h>//бібліотека RTC від Makuna
#include <LiquidCrystal_PCF8574.h>     //
бібліотека для LCD1602 через I2C
```

🔧 *Ініціалізація пристроїв*

📟 LCD:

```
LiquidCrystal_PCF8574 lcd(0x27); // адреса I2C
LCD дисплея (може бути 0x3F)
```

🕒 RTC DS1302:

```
ThreeWire myWire(4, 5, 2);  // IO=4, SCLK=5, CE=2
```

```
RtcDS1302<ThreeWire> Rtc(myWire);
```

ThreeWire – це тип підключення DS1302, не через I2C, а власним протоколом з 3-ма лініями: IO, CLK, CE.

*`setup()`: налаштування при старті*

```
Wire.begin(19, 21); // Для ESP32: SDA=19, SCL=21
lcd.begin(16, 2);   // 16 символів × 2 рядки
lcd.setBacklight(255); // Увімкнення підсвітки
```

🕓 RTC ініціалізація

```
Rtc.Begin();
RtcDateTime compiled = RtcDateTime(__DATE__, __TIME__);
```

`__DATE__` та `__TIME__` – це макроси компілятора, що зберігають дату і час компіляції. Ці значення використовуються для початкового налаштування модуля RTC, якщо дані в ньому некоректні.

```
if (!Rtc.IsDateTimeValid()) {
    Rtc.SetDateTime(compiled);
}
```

□□ Якщо RTC не має дійсної дати – встановлюється дата компіляції.

```
if (Rtc.GetIsWriteProtected()) {
    Rtc.SetIsWriteProtected(false);
}
```

🔓 RTC може бути захищений від запису – знімаємо цей захист.

```
if (!Rtc.GetIsRunning()) {
    Rtc.SetIsRunning(true);
}
```

□□ Якщо RTC не працює – запускаємо його.

*`loop()`: оновлення LCD і Serial монітора щосекунди*

Отримання поточного часу:

```
RtcDateTime now = Rtc.GetDateTime();
```

Виведення на Serial:

```
printDateTime(now);
```

Виведення на LCD:

```
lcd.setCursor(0, 0);
lcd.print("Time ");
print2digits(now.Hour()); lcd.print(":");
print2digits(now.Minute()); lcd.print(":");
print2digits(now.Second());
lcd.setCursor(0, 1);
lcd.print("Date ");
print2digits(now.Day()); lcd.print(".");
print2digits(now.Month()); lcd.print(".");
```

```
lcd.print(now.Year());
```
Рядок 1: час, рядок 2: дата.

🔧*Допоміжні функції:*

➤ `printDateTime()` — для Serial
```
snprintf_P(datestring,
           sizeof(datestring),
           PSTR("%02u/%02u/%04u %02u:%02u:%02u"),
           dt.Month(), dt.Day(), dt.Year(),
           dt.Hour(), dt.Minute(), dt.Second());
Serial.print(datestring);
```
Виводить дату та час у форматі `MM/DD/YYYY HH:MM:SS`

➤ `print2digits()` — для LCD
```
if (number < 10) lcd.print("0");
lcd.print(number);
```
Гарантує, що числа (наприклад, 7) виводяться як `07` – важливо для формату часу.

6. Проводимо дослідження функціонування макета рис. 5.15 та рис. 5.1.

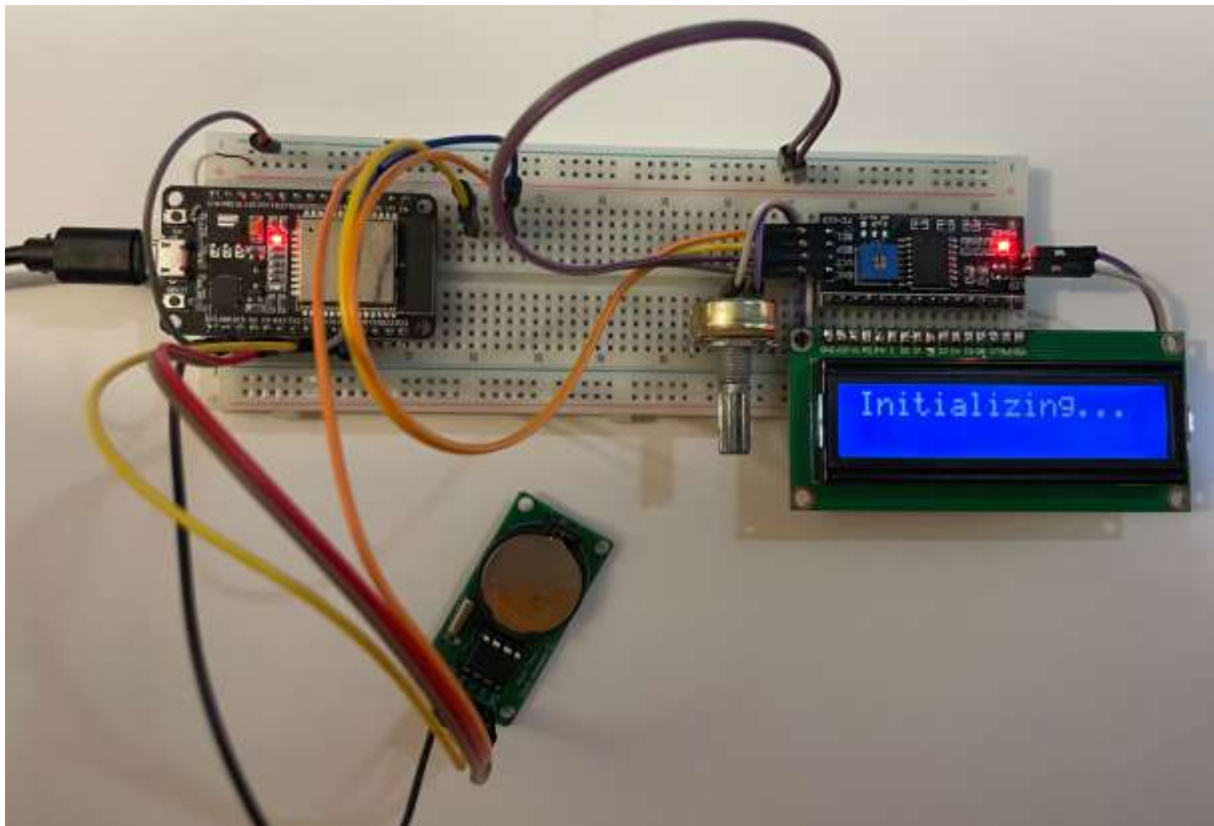



Рисунок 5.15 – Стартовий процес ініціалізації макета

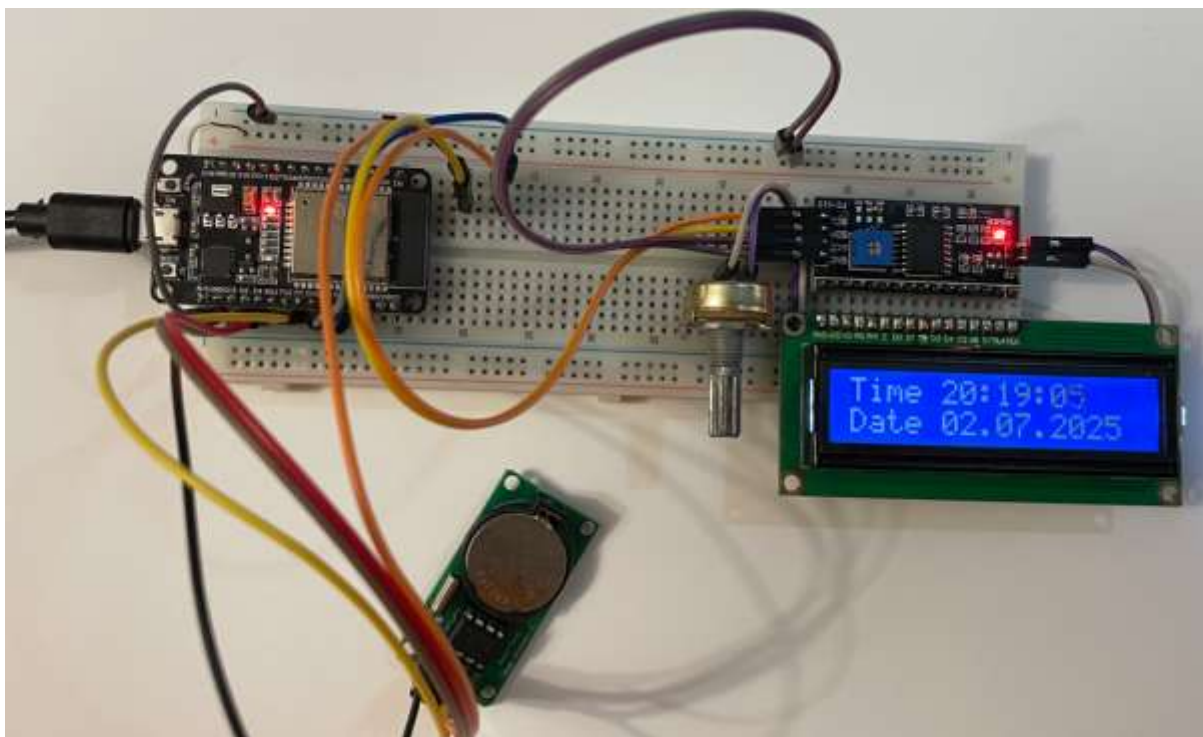



Рисунок 5.16 – Поточний режим функціонування макета

7. У ході виконання роботи було успішно реалізовано програмний модуль, що забезпечує зчитування реального часу з модуля DS1302 та виведення актуальної дати й часу на LCD1602 з інтерфейсом I2C. Було проведено детальне дослідження різних бібліотек для RTC, зокрема Rtc by Makuna та DS1302 від Rafa Couto, що дозволило вибрати стабільне рішення для ESP32.

Під час налагодження виникли типові помилки компіляції, пов’язані з неправильним використанням функцій setTime(), setDate() та getTime(), які не підтримувались конкретною реалізацією бібліотеки. Ці проблеми було усунуто шляхом аналізу вихідного коду бібліотек та заміни функціоналу на правильний (Rtc.SetDateTime(), Rtc.GetDateTime()).

Було також діагностовано та виправлено помилки на шині I2C (I2C hardware NACK detected), пов'язані з неправильним зазначенням пінів SDA і SCL для ESP32. Завдяки точному визначенню пінів Wire.begin(19, 21) вдалося стабілізувати зв’язок з LCD.

Загалом розроблений код повністю виконує поставлене завдання: дозволяє точно зчитувати та відображати реальний час і дату, що зберігаються у DS1302, та виводити їх на дисплей у зручному для користувача форматі. Отримані навички є корисними для подальшої розробки мікропроцесорних систем, що використовують модуляцію часу та інтерфейси I2C.

## Варіанти завдань

1. Виводити на дисплей напругу у форматі "U = X.XX В" з оновленням кожні 0,5 секунди.
2. Виводити не лише напругу, а й відсоткове значення (наприклад, "75 %").
3. Реалізувати індикатор рівня напруги у вигляді смуги з символів (барграф).
4. Виводити повідомлення "LOW", якщо напруга менше 1 В, "HIGH" – якщо більше 2.5 В.
5. Додати кнопку-перемикач між двома режимами відображення (наприклад: напруга/відсотки).
6. Виводити максимум і мінімум значень напруги за останні 30 секунд.
7. Реалізувати оновлення дисплея лише у разі зміни значення більш ніж на 0.05 В.
8. Додати годинник мілісекунд (таймер), який показує час з початку запуску.
9. Показувати на другому рядку текстову шкалу (наприклад: [### ]).
10. Виводити цифрове значення ADC (0–4095) поряд із напругою.
11. Створити «аналоговий» покажчик: залежно від напруги курсор переміщується по екрану.
12. Замість LCD1602 використати OLED (через ту саму бібліотеку Wire).
13. Створити умовне підсвічування символів: якщо напруга зростає – показати ↑.
14. Вивести дані у вигляді двох рядків: "ADC = xxxx" та "U = x.xx В".
15. Реалізувати можливість «паузи» на екрані за кнопкою – фіксація значення.
16. Виводити напругу, переведену в шкалу температури (наприклад: "t = x °C").
17. Створити систему попередження – якщо напруга менше 0.5 В, блимає напис "ALARM".
18. Використати кілька потенціометрів і виводити значення кожного на окремому рядку.
19. Додати автокалібровку: визначити і вивести середнє значення напруги за 5 секунд.
20. Реалізувати логіку, яка під час натискання кнопки скидає максимум/мінімум напруги.

## Зміст звіту

Звіт має містити:

1. Завдання.
2. Обгрунтування алгоритму програми.
3. Текст коду програми.
4. Електричні схеми реалізованих рішень (фото, скріни і под.).
5. Висновки за результатами проведених досліджень.

## Контрольні запитання

1. Що таке інтерфейс I2C і для чого він використовується?
2. Які переваги має I2C порівняно з SPI або UART?
3. Які сигнальні лінії використовуються в I2C?
4. Яка стандартна адреса I2C-дисплея LCD1602?
5. Що означає параметр SDA і SCL в ESP32, і які виводи зазвичай використовуються?
6. Як ініціалізується дисплей LCD1602 через I2C в Arduino?
7. Яку бібліотеку потрібно підключити для роботи з LCD1602 по I2C?
8. Яка функція відповідає за виведення тексту на дисплей?
9. Як очистити екран LCD1602 у коді?
10. Що таке аналоговий сигнал?
11. Яке значення має analogRead() для ESP32?
12. В якому діапазоні працює вбудований ADC ESP32?
13. Як перетворити значення з analogRead() у напругу?
14. Який тип змінної краще використовувати для відображення напруги з десятковою частиною?
15. Як вивести змінну типу float на LCD1602?
16. Чому за живлення 3.3 В максимальна напруга на потенціометрі не може бути більшою за цю межу?
17. Які типові помилки у разі підключення I2C-дисплея до ESP32?
18. Що буде, якщо адреса дисплея в коді не збігається з реальною?
19. Чому важливо встановлювати затримку delay() у циклі loop()?
20. Як протестувати підключення дисплея в середовищі Wokwi?

## Лабораторна робота 6
### Відображення інформації на OLED дисплеї

**Мета**. Ознайомитись із принципом роботи OLED-дисплея, підключенням його до мікроконтролера ESP32 через інтерфейс I2C та навчитися виводити текстову інформацію на дисплей.

### Основні теоретичні відомості

OLED-дисплеї (Organic Light-Emitting Diode) – це дисплеї, які складаються з органічних світлодіодів і можуть виводити текст, графіку, символи. Структуру органічного світлодіода зображено на рис. 6.1.

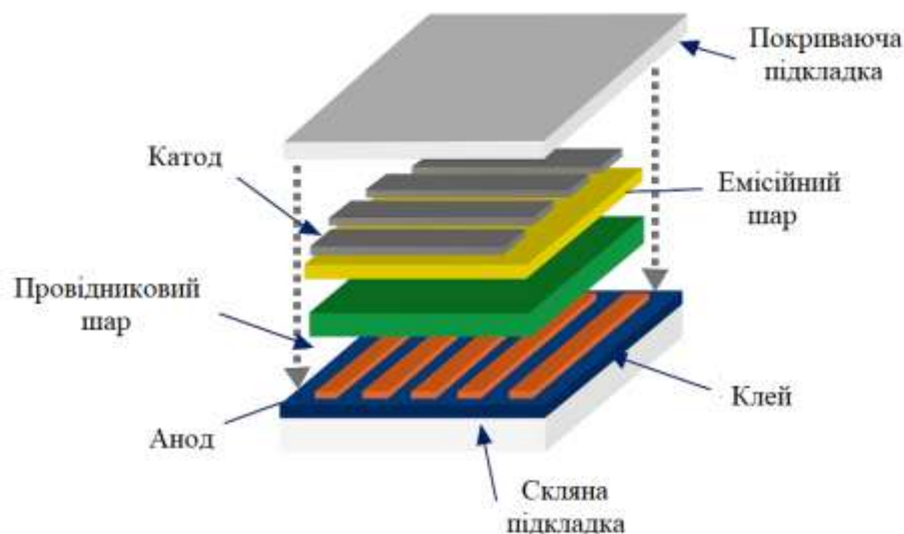



Рисунок 6.1 – Структура органічного світлодіода

Органічний світлодіод (OLED) – це тип світлодіода, в якому випромінювальний шар складається з тонкої плівки органічних речовин, здатних до електролюмінесценції. Органічний світлодіод має таку будову:

- *підкладка*, що може бути виконана зі скла, гнучкої плівки або пластику;
- *анод*, виготовлений із прозорого струмопровідного матеріалу (наприклад, оксиду індію та олова – ITO), через який в активний шар вводяться позитивні носії заряду (дірки);
- *шар органічних матеріалів*, який містить два функціональні підшари: один транспортує дірки (може містити поліанілін або PEDOT – поліетилендіокситіофен), інший – електрони (з боку катода). У зоні між цими підшарами відбувається рекомбінація носіїв заряду з випромінюванням світла. Як емісійний шар часто використовують поліфеніленвініл або поліфлуорен;
- *катод*, що подає електрони в органічний шар. Він може бути як прозорим, так і металевим (непрозорим).

Внаслідок такої будови електрони та дірки з протилежних сторін мігрують до емісійного шару, де їх рекомбінація спричиняє випромінювання світла.

Товщина одного OLED-пікселя у 200 разів менша за діаметр людського волосся, що свідчить про надзвичайно мініатюрну структуру елемента.

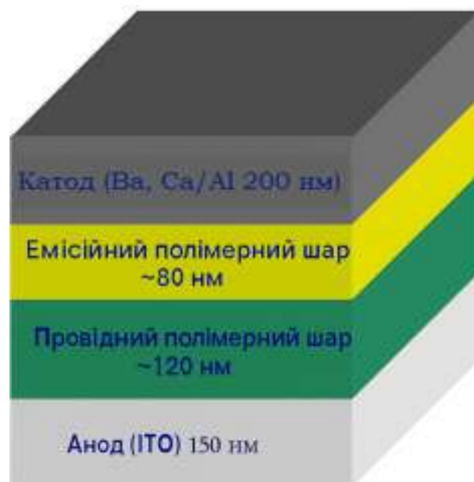



Рисунок 6.2 – Структура одного піксела

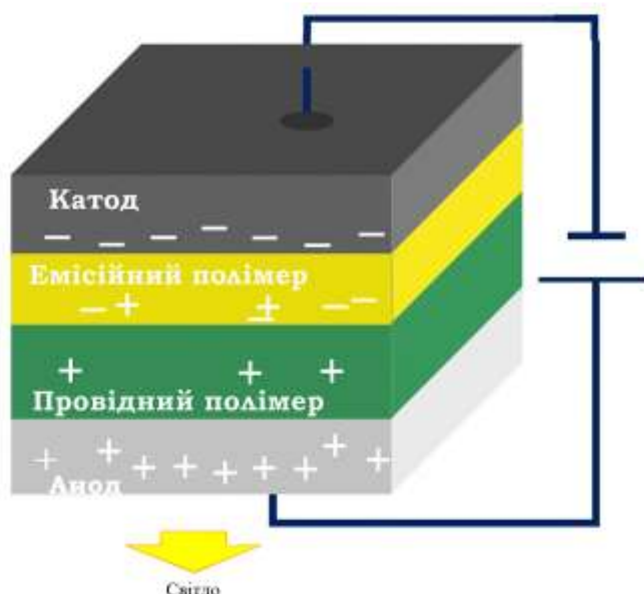



Рисунок 6.3 – Схема випромінювальної рекомбінації
під час проходження прямого струму

У структурі OLED електрони подаються до структури через катод, тоді як дірки (позитивні носії заряду) – через анод. Під впливом прикладеної зовнішньої напруги ці носії заряду рухаються у бік емісійного шару з органічного полімеру. У цьому шарі відбувається їх взаємна рекомбінація – злиття електрона і дірки, що супроводжується випромінюванням фотона, тобто утворенням світлового кванта.

OLED-дисплей містить дві основні складові: сам екран, який випромінює світло, та блок керування. Система управління зазвичай складається з комп’ютера або мікроконтролера, плати керування, а також пристроїв, які забезпечують надходження контенту – це можуть бути інтернет-з'єднання, відеоплеєри, медіасервери тощо.

Основою зображення на OLED-екрані є органічний світлодіод – він і є джерелом світла. Щоб формувати кольорову картинку, кожен піксел складається з трьох світлодіодів, що випромінюють світло червоного, зеленого та синього кольору. Комбінація інтенсивностей цих трьох складових дозволяє створювати всі кольори спектра.

Існує два основні різновиди OLED-дисплеїв:

- PMOLED (з пасивною матрицею);
- AMOLED (з активною матрицею).

Ключова відмінність між ними полягає у методі керування пікселами:

У PMOLED дисплеях для формування зображення використовуються лінії та стовпці, які активуються почергово (рис. 6.4). Щоб засвітити конкретний піксел, необхідно подати сигнал на відповідний рядок і стовпець. В один момент часу активним може бути лише один піксел, тому для відображення повного зображення система має швидко перемикати сигнали між усіма пікселами у матриці, здійснюючи послідовну розгортку.

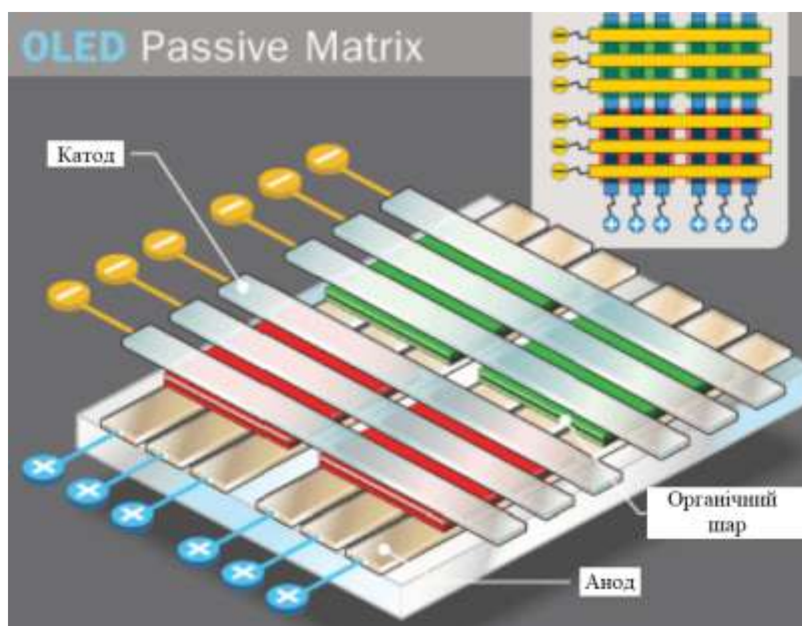



Рисунок 6.4 – Структура пасивно-матричного OLED

У AMOLED дисплеях застосовується активна матриця з транзисторами, які керують кожним пікселем окремо, дозволяючи

одночасне управління багатьма пікселами (рис. 6.5). Це забезпечує вищу яскравість, контрастність та енергоефективність.

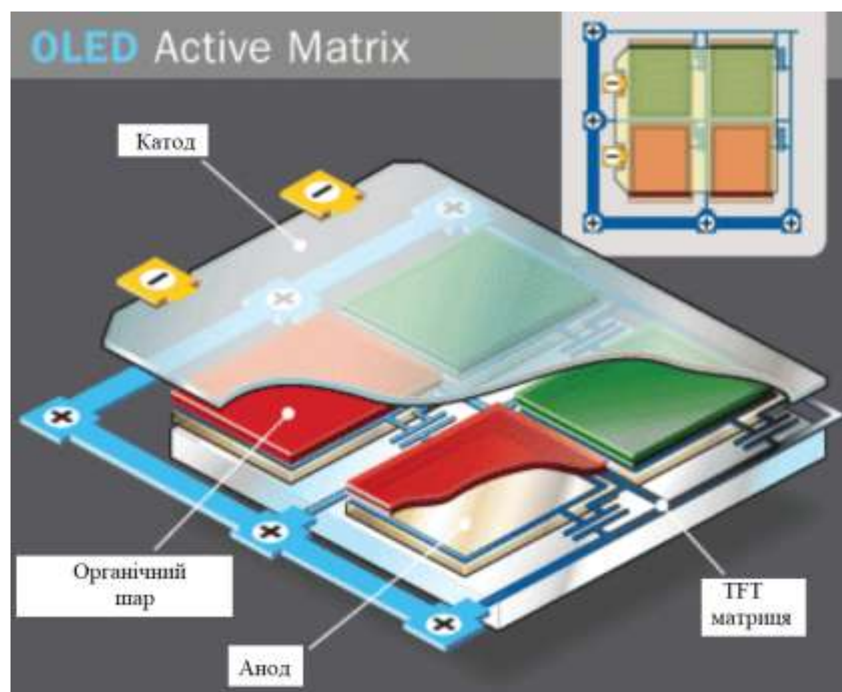



Рисунок 6.5 – Структура активно-матричного OLED

Пасивна матриця OLED (PMOLED) працює за принципом почергового керування пікселами, подібно до того, як це реалізовано в застарілих електронно-променевих трубках (ЕПТ). Тобто кожен піксел активується лише на короткий час, протягом якого через нього проходить електричний струм. Інтенсивність світіння безпосередньо залежить від сили струму, що подається на відповідну комірку.

Дисплеї з пасивною матрицею мають просту конструкцію, а їх виготовлення є технічно нескладним. Проте такі екрани споживають більше електроенергії порівняно з іншими типами OLED-дисплеїв. Це зумовлено необхідністю зовнішньої схеми керування для кожного рядка і стовпця.

Втім, навіть з урахуванням цього, енергоспоживання PMOLED все ж менше, ніж у рідкокристалічних дисплеїв (LCD). Через обмеження технології (високу складність рядкової розгортки) вони не підходять для створення екранів великого формату. Натомість їх доцільно застосовувати в компактних пристроях із діагоналлю 2–3 дюйми (5–7,5 см), таких як мобільні телефони, портативні плеєри чи кишенькові комп’ютери. Основна перевага – низька вартість виготовлення.

В дисплеях з активною матрицею (AMOLED) кожен окремий піксел керується індивідуально. Це дозволяє досягти високої швидкості оновлення зображення, а також забезпечує кращу яскравість і контрастність. Завдяки цій архітектурі можливе виробництво екранів

великих розмірів – вже сьогодні випускаються панелі діагоналлю до 40 дюймів (100 см).

Управління пікселами в AMOLED здійснюється за допомогою тонкоплівкових транзисторів (TFT), які формують сітку керування під анодним шаром. Хоча така конструкція є більш складною та дорожчою у виробництві, ніж у PMOLED, вона має низку переваг: менше енергоспоживання, вища якість зображення та можливість відображення відео завдяки високій частоті оновлення.

Сьогодні AMOLED-дисплеї активно використовуються у смартфонах, ноутбуках, розумних годинниках і планшетах. У перспективі їх застосування розширюється на великі телевізори, інформаційні панелі, електронні вивіски та рекламні табло.

Серед альтернативних технологій органічних світлодіодних екранів виділяється TOLED (від англ. Transparent and Top-emitting OLED), що перекладається як прозорий та фронтально-випромінювальний OLED-дисплей. Ця технологія дозволяє створювати екрани, які є частково або повністю прозорими, завдяки чому можуть застосовуватися, наприклад, у вітринах, головних дисплеях на автомобілях або носимій електроніці (рис. 6.6).

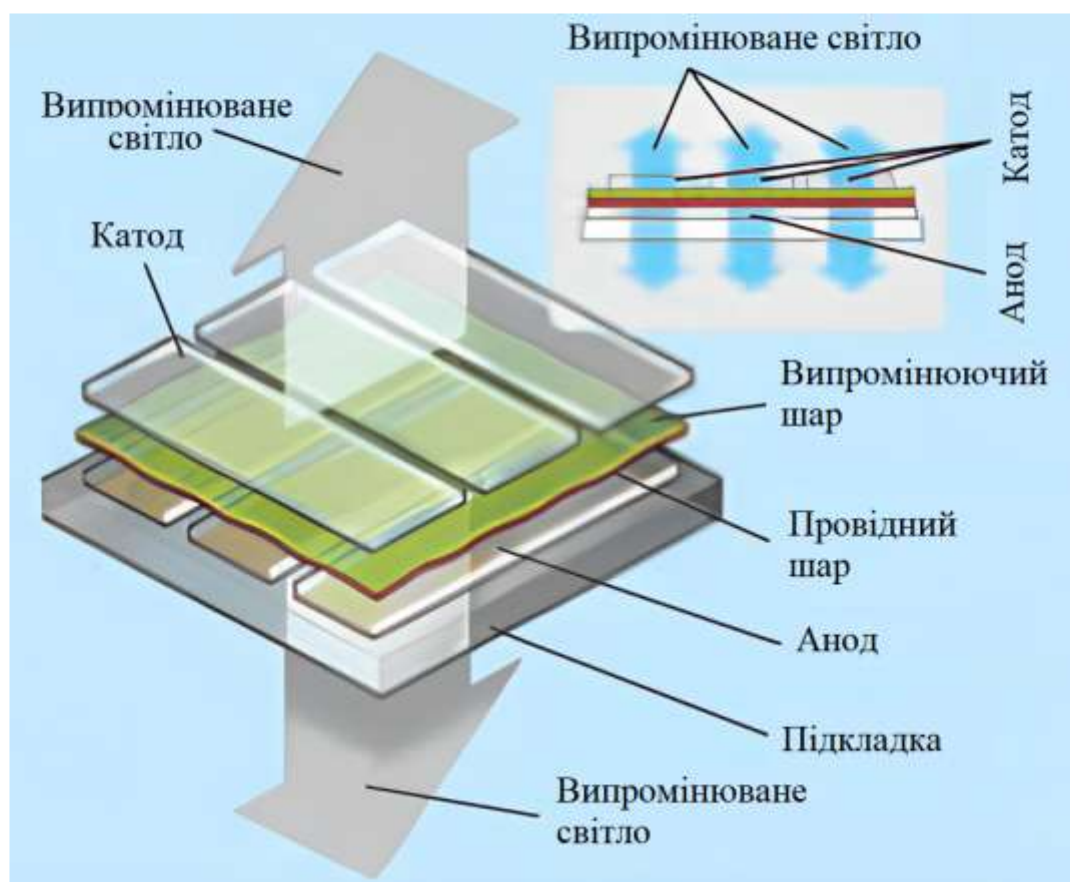



Рисунок 6.6 – Структура TOLED-дисплея

Крім прозорості, TOLED також забезпечує високу контрастність зображення, оскільки світловипромінювання здійснюється з верхнього

шару, що мінімізує втрати світла та покращує яскравість і глибину чорного кольору. Це робить технологію перспективною для використання в умовах з підвищеним рівнем зовнішнього освітлення.

*Прозорі OLED-дисплеї TOLED*

TOLED-дисплеї (Transparent and Top-emitting OLED) – це особливий тип органічних світлодіодних екранів, у яких світло може випромінюватися як тільки вгору, так і вниз, або ж в обох напрямках одночасно. Така конструкція забезпечує підвищену контрастність, що покращує сприйняття інформації на дисплеї навіть у випадках сильного зовнішнього освітлення, наприклад, на сонці (рис. 6.7).

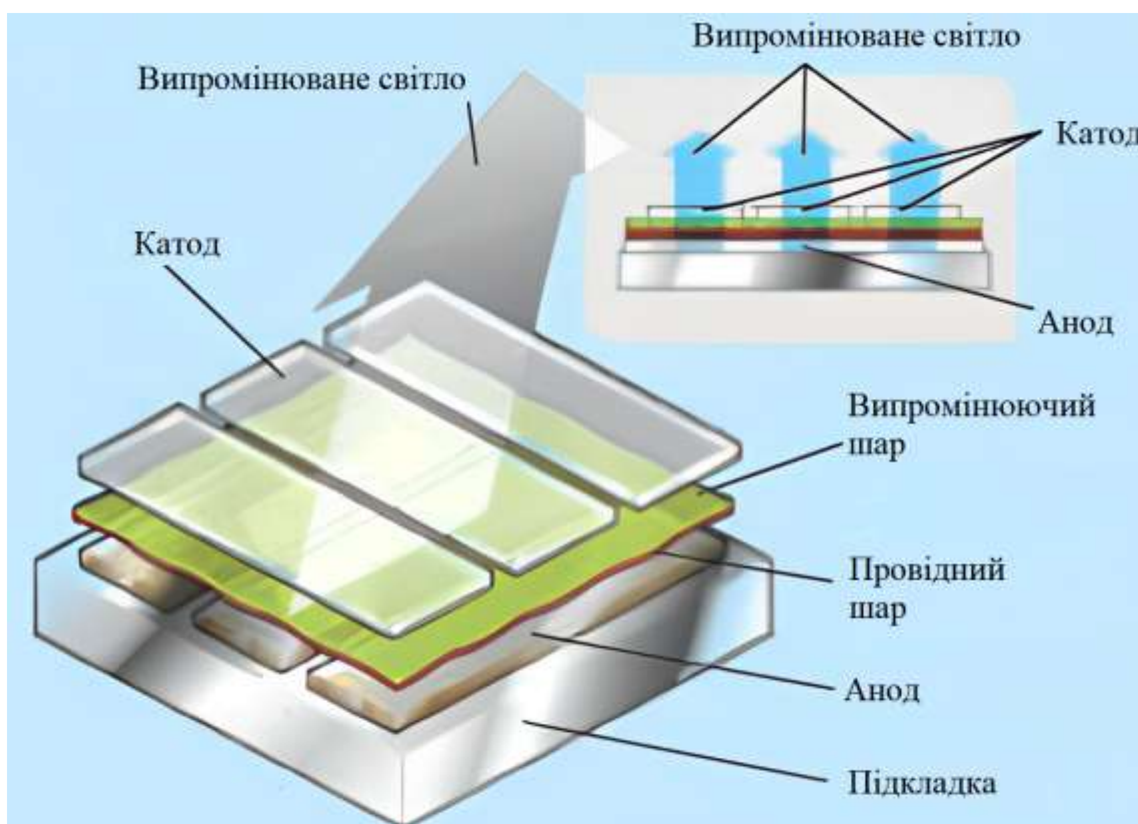



Рисунок 6.7 – Структура TEOLED

TOLED-технологія реалізується як у вигляді пасивної, так і активної матриці, залежно від способу керування пікселами. Вимкнені TOLED-дисплеї залишаються до 70 % прозорими, що відкриває широкі можливості для інтеграції в інтер'єр чи транспорт – їх можна безпосередньо встановлювати на лобові скла автомобілів, торговельні вітрини або елементи розумного простору.

Завдяки високій прозорості ці дисплеї також можна поєднувати з непрозорими матеріалами, зокрема металом, кремнієвими підкладками, фольгою тощо. Це дає змогу створювати екрани з фронтальним виведенням зображення, які можуть знайти застосування, наприклад, у

майбутніх динамічних банківських картках із цифровим відображенням інформації.

Важливим фактором прозорості TOLED є використання прозорих органічних шарів та електродів, що дозволяє досягти максимальної світлопроникності.

Крім того, завдяки застосуванню спеціальних підкладок з низьким коефіцієнтом відбиття світла, такі дисплеї можуть демонструвати контрастність, яка значно перевищує показники LCD-екранів. Саме тому TOLED-технології все активніше використовуються в індустріях, де потрібна висока видимість – наприклад, у дисплеях для шоломів пілотів, військовій авіації або портативній електроніці.

*TOLED → SOLED та гібридні матриці*

- На базі прозорих TOLED-структур можна створювати багатошарові екрани (наприклад, SOLED) або гібридні двосторонні матриці.
- У двонаправлених TOLED-панелях світло випромінюється з обох боків, тож за тієї самої діагоналі площу відображення фактично подвоюють. Це корисно для пристроїв, де потрібно показати більше контенту, ніж дозволяє класичний екран.

SOLED (Stacked OLED) — «штабельні» OLED

- У SOLED підпіксели R, G та B розташовані вертикально один над одним, а не поруч, як у РК- або ЕПТ-матрицях.
- Кожен підпіксел керується окремо.
  - Колір задають, змінюючи струм через три кольорові шари.
  - Яскравість регулюють амплітудою струму (у монохромних версіях – широтно-імпульсною модуляцією).
- *Переваги*: дуже висока щільність заповнення активної області, відмінна роздільна здатність і, за оцінками розробників, утричі краща якість зображення, ніж у РК- та кінескопних панелей.

*FOLED (Flexible OLED) – гнучкі екрани*

- Головна риса – пластична підкладка (полімерна чи тонка металева фольга) з одного боку та OLED-шари, захищені тонкою герметичною плівкою, – з іншого.
- *Переваги FOLED*: ультратонкий профіль, мінімальна маса, ударостійкість, довговічність і, головне, гнучкість, що дає змогу розміщувати екран у найнесподіваніших формах.
- Перше покоління FOLED уже придатне для «комфорт-пристроїв»: смартфонів, що повторюють вигин долоні, чи портативних програвачів із зігнутим дисплеєм.

*Піксел у світлодіодному екрані*

- Фізичний піксел складається з трьох світлодіодів базових кольорів – червоного, зеленого та синього.
- Розміри піксела залежать від габаритів та кількості цих світлодіодів.
- Незалежно від конфігурації, піксел відтворює одну мінімальну точку зображення.
- Для характеристики екрана використовують:
  - Розмір піксела (фізична довжина сторони/діаметр),
  - Крок піксела (відстань між центрами сусідніх пікселів), що визначає деталізацію й якість картинки.

Перші OLED-дисплеї були створені на основі мікромолекулярних матеріалів, але їх виробництво виявилося надто дорогим через необхідність використання вакуумного напилення.

Початок розвитку полімерних OLED-дисплеїв відбувся у 1989 році, коли дослідники з Кембриджського університету синтезували новий полімер – поліфениленвініл. Такі дисплеї виготовляються шляхом нанесення полімерного шару на підкладку за допомогою спеціального струменевого друку. Подібні дисплеї іноді називають LEP (Light-Emitting Polymer – полімер, що випромінює світло). Основа таких дисплеїв може бути гнучкою, з радіусом вигину до 1 см і менше.

Втім, за такими параметрами, як тривалість служби та світловіддача, мікромолекулярні OLED-пристрої наразі перевершують полімерні аналоги LEP. Порівняльні характеристики цих двох технологій щодо довговічності та ефективності світіння наведено нижче.

Сьогодні застосовуються три основні архітектурні підходи до побудови кольорових OLED-дисплеїв:

- із використанням окремих випромінювачів для кожного кольору;
- за схемою WOLED + CF (білі випромінювачі у поєднанні з кольоровими фільтрами);
- із застосуванням перетворення короткохвильового випромінювання для створення кольорового зображення.

Найпростішим і найвідомішим варіантом побудови OLED-дисплеїв є трикольорова схема, яку ще називають моделлю з окремими емісійними елементами. У такій конфігурації використовуються три органічні речовини, кожна з яких випромінює світло одного з базових кольорів: червоного (R), зеленого (G) або синього (B). Цей підхід вважається найефективнішим з погляду енергоспоживання. Проте на практиці виникають труднощі з підбором органічних матеріалів, які б одночасно відповідали необхідним довжинам хвиль та мали однакову яскравість.

Другий спосіб технічно реалізується простіше. У ньому застосовуються три однакові білі випромінювачі, світло яких проходить крізь кольорові фільтри. Такий підхід суттєво поступається в

енергоефективності першому варіанту, оскільки значна частина світлового потоку втрачається під час фільтрації.

Третій варіант, відомий як CCM (Color Changing Media — середовище з перетворенням кольору), базується на використанні синіх випромінювачів та спеціально підібраних люмінесцентних речовин. Ці речовини перетворюють короткохвильове синє світло на випромінювання з довшими хвилями – зелене та червоне. У цьому випадку синій компонент створюється безпосередньо, без перетворення.

*Порівняння дисплейних технологій: переваги та недоліки OLED*

Основними альтернативами органічним світлодіодним дисплеям (OLED) сьогодні залишаються рідкокристалічні панелі (LCD) та світлодіодні індикатори (LED). LCD застосовують у всьому – від наручних ґаджетів до великих телевізорів, тоді як класичні (нематричні) LED найчастіше бачимо на циферблатах електронних годинників. Нижче – стислий огляд того, чим OLED виграє та поступається своїм конкурентам.

Таблиця 6.1 – Порівняльна таблиця

| Фактор | OLED | LCD | Плазма / класичний LED |
|---|---|---|---|
| Товщина й вага | Найтонші, найлегші | Товстіші через підсвічування | Плазма – найбільш масивна |
| Підкладка | Гнучкі полімери чи тонке скло | Скло (жорстке) | Скло |
| Підсвічування | Не потрібне – піксел світиться сам | Обов'язкове LED-підсвічування | Вбудовані комірки плазми |
| Кут огляду | ≈ 170° без спотворень | Залежить від типу матриці (TN найгірше) | Добрий, але відблиски |
| Енергоспоживання | Низьке (без підсвічування) | Вище | Середнє/вище |
| Яскравість регульована | 1 – 100 000 кд/м² | Зазвичай нижче | Висока |
| Контрастність | До 1 000 000 : 1 | До 2 000 : 1 | ≈ 5 000 : 1 |

*Переваги OLED порівняно з плазмою*

1. Значно менша товщина та маса панелі.
2. Така сама або більша яскравість за менших витрат електроенергії.
3. Можливість створювати гнучкі або навіть згортувані екрани.
4. Відсутність ефекту «вигорання» під час тривалого показу статичного зображення.

*Переваги OLED над LCD*

1. Компактність і менша вага завдяки відсутності підсвічування.

2. Немає жорстких обмежень щодо кута огляду – картинка залишається чіткою під будь-яким ракурсом.

3. Миттєва реакція пікселів (на кілька порядків швидша, ніж у LCD), тобто мінімальна інерційність.

4. Достовірніша передача кольорів і надвисокий контраст.

5. Робота у широкому температурному діапазоні (-40 … +70 °C).

6. Перспектива масового випуску гнучких або прозорих дисплеїв.

*Параметри зображення*

- *Яскравість*. OLED-матриці можуть світити як тьмяно (кілька кд/м² для нічного режиму), так і надяскраво (>100 000 кд/м²). В експлуатації зазвичай обмежуються ~1 000 кд/м², щоб не скорочувати ресурс.
- *Контрастність*. Типові значення ≈ 1 000 000 : 1 – на порядок вищі за LCD та CRT.
- *Кут огляду*. Якість картинки практично не змінюється під час огляду з боку; сучасні IPS- та VA-матриці LCD наблизилися, але все ще поступаються OLED.

*Вразливі місця та поточні обмеження*

1. Термін служби синього субпіксела. Органічні сполуки, що випромінюють синій колір, деградують швидше (≈ 17-18 тис. год безперервної роботи), ніж червоні та зелені, які витримують 30-40 тис. год. Через це з часом баланс кольорів порушується.

2. Складність масштабного виробництва великих матриць. На сьогодні технологія still дорожча та менш відпрацьована, ніж масове виготовлення LCD.

3. Чутливість до вологи та кисню. Необхідна надійна герметизація; потрапляння води миттєво пошкоджує органічний шар.

OLED-технологія забезпечує неперевершену контрастність, глибокий чорний колір, гнучкість і тонкий профіль, але поки що стикається з проблемою довговічності синього органічного шару та високою собівартістю великих екранів. Попри це, для портативної електроніки й преміальних дисплейних рішень її переваги переважують недоліки, і розвиток матеріалів поступово розширює сферу застосувань.

OLED-дисплеї можуть використовувати два інтерфейси:

- I²C (двопровідний інтерфейс) – SDA (дані) і SCL (тактовий сигнал),
- SPI (швидкісний інтерфейс) – використовується рідше через більшу кількість необхідних ліній.

Для варіанта з I²C:

- OLED має 4 виводи: VCC, GND, SCL, SDA,
- VCC зазвичай підключається до 3.3В або 5В (залежно від дисплея),
- GND – до «землі»,

- SCL та SDA – до відповідних I²C-виводів ESP32 (типові — GPIO 22 і GPIO 21).

**Увага**! ESP32 має кілька I²C-шин, але за замовчуванням використовуються GPIO22 (SCL) і GPIO21 (SDA). Їх можна змінити в коді.

Найзручнішими бібліотеками для OLED-дисплеїв SSD1306 у середовищі Arduino IDE є:

- `Adafruit_SSD1306`;
- `Adafruit_GFX`.

Бібліотека Adafruit_SSD1306 є конкретною реалізацією для OLED-дисплеїв з контролером SSD1306. Вона успадковує всі методи Adafruit_GFX, але додає низку специфічних для SSD1306 функцій, включно з ініціалізацією дисплея, керуванням буфером і виведенням на екран. Її основні методи наведено в таблиці 6.2.

Таблиця 6.2 – Основні методи Adafruit_SSD1306

| **Метод** | **Опис** |
|---|---|
| `begin(VccType, address)` | Ініціалізація дисплея. Наприклад: `begin(SSD1306_SWITCHCAPVCC, 0x3C)`. |
| `clearDisplay()` | Очищає внутрішній буфер (екран ще не оновлюється). |
| `display()` | Виводить буфер на екран. Без цього всі рисунки залишаються в пам'яті. |
| `invertDisplay(true/false)` | Інвертує кольори (біле стає чорним і навпаки). |
| `dim(true/false)` | Зменшує яскравість. |
| `ssd1306_command(cmd)` | Надсилає команду безпосередньо на дисплей (низькорівневий доступ). |
| `startscrollright(...)`, `startscrollleft(...)`, `stopscroll()` | Програмна прокрутка тексту/графіки на екрані. |

Бібліотека Adafruit_GFX – це графічна бібліотека, яка забезпечує базову функціональність рисування: текст, лінії, прямокутники, кола, трикутники, піксели, зображення. Вона є «основою», яку використовують багато інших бібліотек дисплеїв (зокрема Adafruit_SSD1306). Її основні методи наведено в таблиці 6.3.

=

Таблиця 6.3 – Основні методи Adafruit_GFX

| **Метод** | **Опис** |
|---|---|
| `setCursor(x, y)` | Встановлює позицію курсора для тексту. |
| `setTextSize(size)` | Задає масштаб тексту (1 – стандартний розмір). |
| `setTextColor(color)` | Встановлює колір тексту (наприклад, `SSD1306_WHITE`). |
| `setTextColor(color, bg)` | Текст і фон (рідко використовується в OLED). |
| `println("Text")` | Виводить текст у поточній позиції та переводить курсор на новий рядок. |
| `print("Text")` | Виводить текст без переходу на новий рядок. |
| `drawPixel(x, y, color)` | Рисує піксел у вказаній позиції. |
| `drawLine(x0, y0, x1, y1, color)` | Рисує пряму лінію. |
| `drawRect(x, y, w, h, color)` | Рисує порожній прямокутник. |
| `fillRect(x, y, w, h, color)` | Рисує заповнений прямокутник. |
| `drawCircle(x, y, r, color)` | Рисує порожнє коло. |
| `fillCircle(x, y, r, color)` | Рисує заповнене коло. |
| `drawTriangle(x0, y0, x1, y1, x2, y2, color)` | Рисує трикутник. |
| `fillTriangle(...)` | Рисує заповнений трикутник. |
| `drawRoundRect(x, y, w, h, r, color)` | Рисує прямокутник із заокругленими кутами. |
| `fillRoundRect(...)` | Рисує заповнений прямокутник із заокругленням. |

Бібліотека Adafruit_GFX не працює без дисплейної бібліотеки (як Adafruit_SSD1306), оскільки не відповідає за фізичний зв’язок із дисплеєм – вона лише рисує у буфері. В таблиці 6.4 наведено типові проблеми, які можуть виникати під час роботи з OLED-дисплеєм у разі використання вищезазначених бібліотек.

Таблиця 6.4 – Типові проблеми

| **Проблема** | **Можливе рішення** |
|---|---|
| Екран не виводить нічого | Перевірити адресу I2C (0×3C або 0×3D), правильне підключення, живлення. |
| Помилка компіляції | Переконайтесь, що обидві бібліотеки встановлено через Library Manager. |
| Миготіння/артефакти | Завжди очищайте дисплей (`clearDisplay()`) перед рисуванням нового кадру. |

Програмна ініціалізація OLED-дисплея

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64

// Створення об'єкта дисплея для інтерфейсу I2C
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

void setup() {
  Wire.begin();  // Ініціалізація I2C
  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) { // 0x3C — типова I2C-адреса OLED
    Serial.println(F("Не вдалося знайти OLED дисплей"));
    for (;;);
  }

  display.clearDisplay(); // Очистити дисплей
  display.setTextSize(1); // Розмір тексту
  display.setTextColor(SSD1306_WHITE); // Колір тексту
  display.setCursor(0, 0); // Позиція початку тексту
  display.println(F("Привіт від ESP32!"));
  display.display(); // Вивести на дисплей
}

void loop() {
  // Тут можна динамічно змінювати зображення
}
```

### Порядок виконання роботи

1. Складаємо в wokwi схему для відображення сигналу синусоїди.

Нам необхідно згенерувати сигнал синусоїди та відобразити його на OLED-дисплеї. Дисплей підключений через двопровідний інтерфейс.

Схему дослідження на платформі wokwi показано на рис. 6.8.

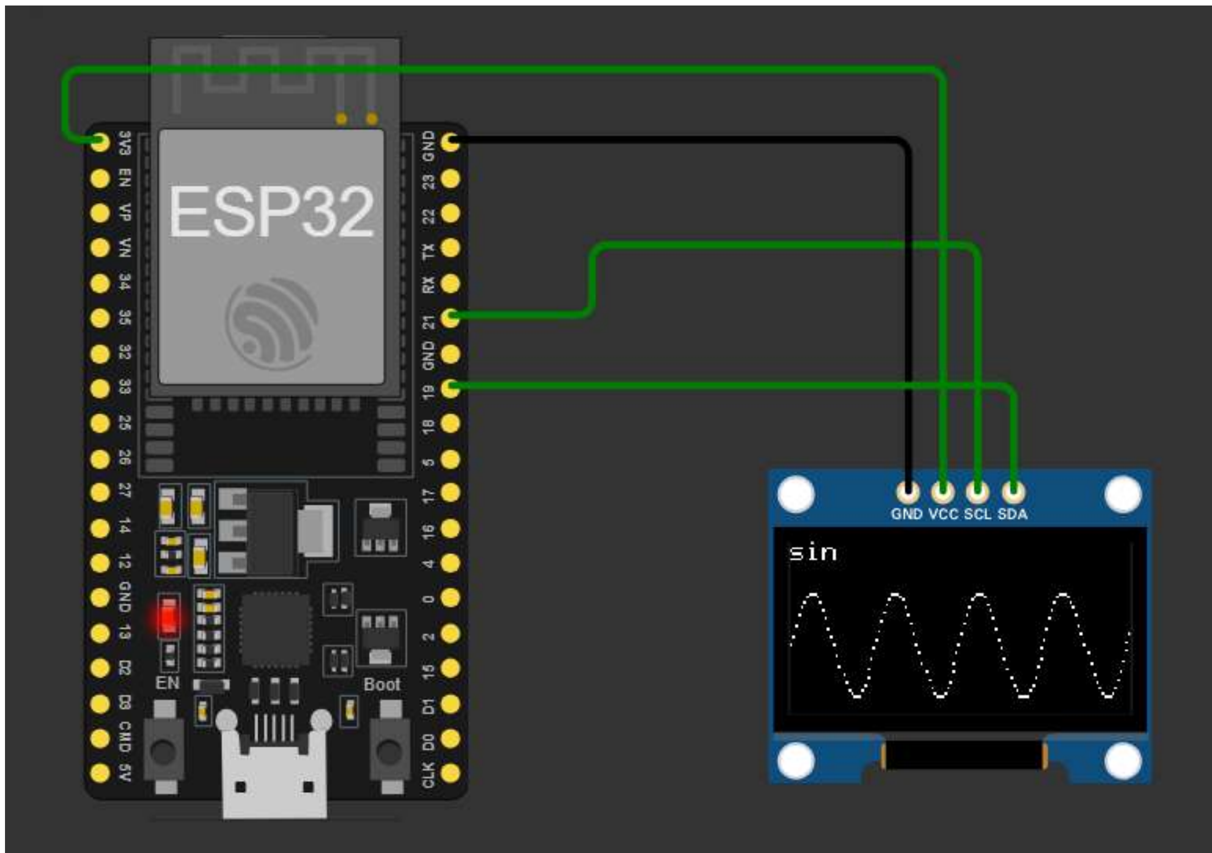



Рисунок 6.8 – Досліджувана схема OLED-дисплея в wokwi

2. Пишемо код нашої програми, яка працюватиме таким чином: спочатку буде надпис типу сигналу «sin», а нижче буде нарисовано синусоїду, яка буде рухатись в режимі реального часу.

```
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <math.h>

#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define OLED_RESET    -1
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

// Параметри синусоїди
#define AMPLITUDE 20
#define FREQUENCY 0.2
```

```
#define OFFSET_Y 38  // Висота синусоїди (зверху)
int x = 0;

void setup() {
  Serial.begin(115200);
  Wire.begin(19, 21);  // SDA, SCL

  if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
    Serial.println("SSD1306 allocation failed");
    for (;;);
  }
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 0);
  display.println("sin");
  display.display();
  delay(1000);
}

void loop() {
  display.clearDisplay();

  // Текст вгорі
  display.setCursor(0, 0);
  display.println("sin");

  // Малюємо синусоїду
  for (int i = 0; i < SCREEN_WIDTH; i++) {
    float y = sin((i + x) * FREQUENCY) * AMPLITUDE;
    display.drawPixel(i, OFFSET_Y - (int)y, SSD1306_WHITE);
  }

  display.display();
  x++;  // Зсув по фазі
  delay(30); // Швидкість анімації
}
```

*Пояснення нашого коду*

1. Підключення бібліотек

```
#include <Wire.h>    // Шина I²C (ESP32 ↔ OLED)
#include <Adafruit_GFX.h>      // Універсальний графічний «рушій»
#include <Adafruit_SSD1306.h> // Драйвер саме для контролера SSD1306
#include <math.h>              // Математичні функції (тут потрібен sin)
```

- Wire.h ініціалізує апаратні або програмні лінії I²C.
- Adafruit_GFX дає високорівневі методи (текст, лінії, кола, …).
- Adafruit_SSD1306 реалізує низькорівневий доступ до RAM дисплея.
- math.h потрібен, щоб викликати `sin()`.

2. Константи та глобальні змінні

```
#define SCREEN_WIDTH  128   // Ширина матриці OLED у пікселях
#define SCREEN_HEIGHT  64   // Висота
#define OLED_RESET     -1   // Немає окремого дроту reset → -1

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);
```

`display` — глобальний об'єкт-драйвер. Розмір і вказівник на `Wire` (I²C) потрібні конструктору.

```
#define AMPLITUDE 20   // «Висота хвилі» у пікселях (±20)
#define FREQUENCY 0.2  // Крок аргументу sin у рад/піксел
#define OFFSET_Y 38    // Вертикальне зміщення хвилі (базова лінія)
int x = 0;             // Поточний фазовий зсув (крок анімації)
```

- AMPLITUDE – масштаб по осі Y.
- FREQUENCY – фактично 2π / період; тут 0,2 рад ≈ 57 пікселів на повну хвилю (2π / 0,2 ≈ 31,4 / 0,2 = ~31).
- OFFSET_Y – базова лінія, від якої рисуємо синусоїду (рахунок з верхнього краю екрана).
- x – лічильник кадрів; кожен кадр зсуває увесь графік на 1 піксел уперед.

3. setup()

```
Serial.begin(115200);
Wire.begin(19, 21);     // SDA = GPIO19, SCL = GPIO21
```

- Вмикаємо UART для відлагодження (115 кбіт/с).
- Примусово вибираємо нестандартні виводи I²C: GPIO19 → SDA, GPIO21 → SCL (типово ESP32 використовує 21/22).

```
if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C)) {
  Serial.println("SSD1306 allocation failed");
```

```
    for (;;);              // Безкінечний цикл, якщо
не ініціалізувався
  }
```

- display.begin() подає живлення на панель (SSD1306_SWITCHCAPVCC вмикає внутрішній charge-pump) і адресує її на I²C-адресі 0x3C.
- Якщо повернеться false – модуль не відповів→ зупиняємо все.

```
  display.clearDisplay();      // Обнуляємо буфер
кадру
  display.setTextSize(1);      // Базовий шрифт 6×8
пікселів
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0, 0);     // Початок екрана
  display.println("sin");      // Пише у RAM (ще не
видно)
  display.display();           // Передає RAM →
відеопам'ять контролера
  delay(1000);                 // Показуємо
заставку 1 с
```

display.display() обов'язковий: поки ви його не викличете, усе рисується лише у буфері SRAM мікроконтролера.

4. loop() – анімаційний кадр

4.1 Очищення кадру

```
  display.clearDisplay();   // Скидає лише
локальний буфер
```

4.2 Повторно друкуємо текст

```
  display.setCursor(0, 0);
  display.println("sin");
```

Чому знову? Тому що ми щойно стерли увесь буфер, включно з попереднім текстом.

4.3 Рендеринг синусоїди

```
  for (int i = 0; i < SCREEN_WIDTH; i++) {
    float y = sin((i + x) * FREQUENCY) * AMPLITUDE;
    display.drawPixel(i, OFFSET_Y - (int)y,
SSD1306_WHITE);
  }
```

1. i – поточна колонка (0…127).
2. (i + x) – додаємо фазовий зсув, щоб хвиля «котилася».
3. Множимо на FREQUENCY → отримуємо аргумент у радіанах.
4. sin(...) дає значення від –1 до +1.
5. Множимо на AMPLITUDE, щоб розтягнути до ±20 пікселів.

6. Перекидаємо знак координати `OFFSET_Y - y`, оскільки на екрані вісь Y йде згори вниз.

7. `drawPixel()` ставить білу точку у буфері. Виконуємо для кожного стовпця → отримуємо безперервну «нитку».

4.4 Відправка кадру на екран

```
display.display();  // I²C передає 1 кБ RAM у відеопам'ять
```

4.5 Підготовка до наступного кадру

```
x++;        // Фаза → хвиля зміщується на 1 піксел праворуч
delay(30); // ≈33 кадри/с (30 мс пауза)
```

Зсув через `x++` створює ілюзію руху: у наступному циклі кожна колонка читає «сусіднє» значення синуса.

5. Налаштування параметрів наведено в табл. 6.5.

Таблиця 6.5 – Налаштування параметрів

| Параметр | Ефект | Як змінити |
|---|---|---|
| AMPLITUDE | Висота хвилі (0 → пряма лінія, 32 → майже до краю) | Зменшіть, якщо стикається з текстом |
| FREQUENCY | Період синуса (менше число → довша хвиля) | 0.1 → двічі плавніше, 0.4 → удвічі «щільніше» |
| OFFSET_Y | Вертикальна позиція середньої лінії | 0 … 63; для тексту зверху ≥16 не чіпати |
| delay() | Швидкість анімації | 10 мс → швидше; 100 мс → повільніше |

6. Що ще можна покращити

1. Товстіша лінія

```
display.drawPixel(i, OFFSET_Y - (int)y + 1, SSD1306_WHITE);
display.drawPixel(i, OFFSET_Y - (int)y - 1, SSD1306_WHITE);
```

або `display.drawLine(prevX, prevY, newX, newY, SSD1306_WHITE)`.

2. Ось X або маркери – `drawFastHLine()` під хвилею.

3. Оптимізація енергоспоживання – зменшити частоту оновлення, вмикати/вимикати дисплей `display.ssd1306_command(SSD1306_DISPLAYOFF)`.

Досліджуємо результати роботи нашого коду та схеми. Результат наведено на рис. 6.9.

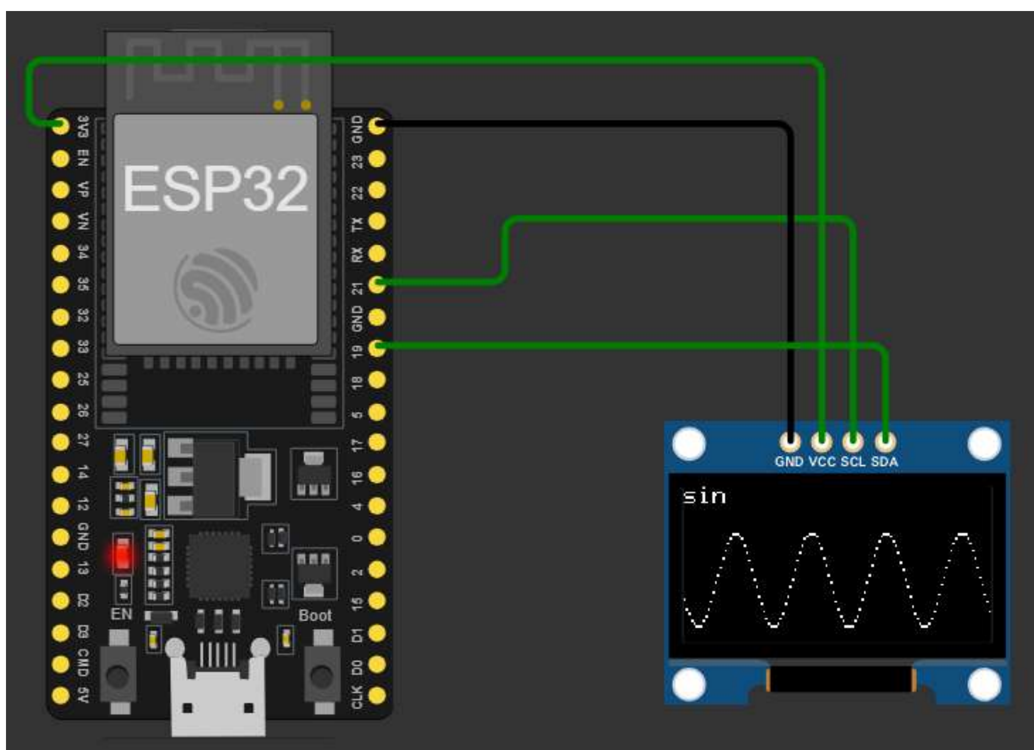


Рисунок 6.9 – Результат моделювання OLED-дисплея

3. Складаємо аналогічну схему на макетній платі з ESP32 (рис. 6.10) та проводитимемо її дослідження рис. 6.11.

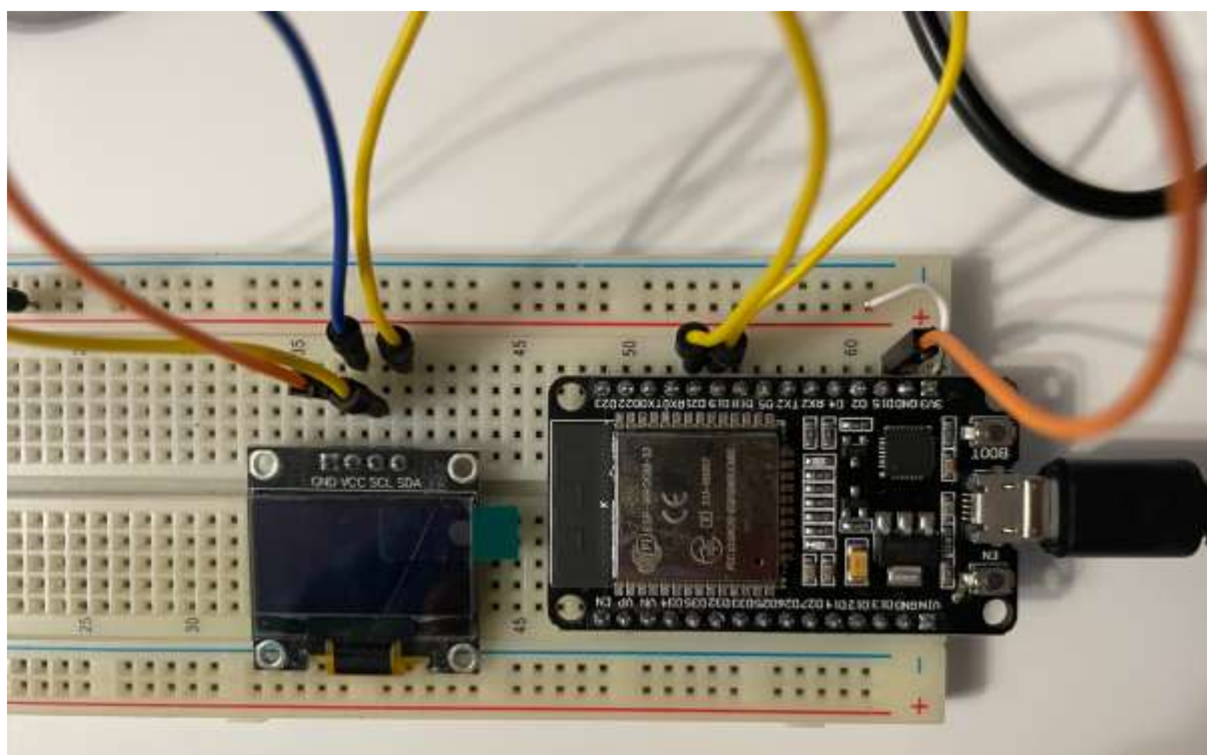


Рисунок 6.10 – Досліджувана схема, складена на макетній платі

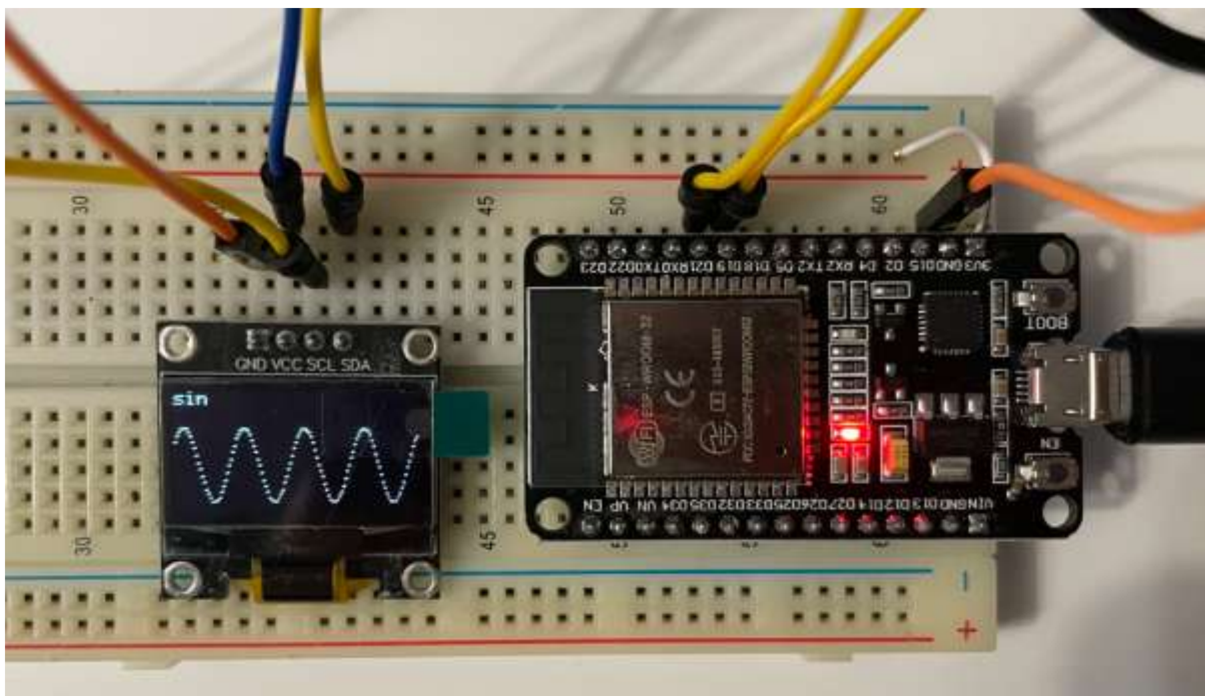



Рисунок 6.11 – Результат дослідження макета

4. У ході роботи було успішно реалізовано взаємодію між мікроконтролером ESP32 та графічним дисплеєм OLED SSD1306 по інтерфейсу I²C. Результатом є виведення на екран динамічного графіка синусоїди, що імітує хвилю, яка рухається, та текстового заголовку "sin".

*Досягнуті цілі*

- Ініціалізовано OLED-дисплей з використанням бібліотек Adafruit_GFX та Adafruit_SSD1306.
- Налаштовано шину I²C з індивідуальними виводами: GPIO19 (SDA) і GPIO21 (SCL).
- Реалізовано графічне зображення синусоїди, яка динамічно оновлюється у циклі loop(), створюючи ефект руху.
- Текстовий заголовок "sin" виведено окремо у верхній частині дисплея.
- Програма стабільно працює в режимі реального часу з частотою оновлення ~33 кадри/с.

*Використані технічні рішення*

- Алгоритм побудови хвилі базується на функції sin() з поступовим фазовим зсувом через інкремент змінної x.
- Частота і амплітуда хвилі налаштовуються за допомогою макросів FREQUENCY та AMPLITUDE, що дозволяє легко адаптувати параметри графіка.
- Затримка між кадрами визначена як 30 мс, що забезпечує плавну анімацію без перевантаження процесора.

## Варіанти завдань

1. Вивести на OLED текст «Hello, World!» по центру екрана.
2. Відображати температуру, отриману з аналогового датчика (наприклад, TMP36).
3. Вивести на дисплей поточне значення таймера (millis).
4. Реалізувати зміну кольору тексту залежно від напруги (для екранів з кольором або симуляції).
5. Відображати напругу з потенціометра у вигляді горизонтального барграфа.
6. Вивести змінне повідомлення залежно від напруги: «LOW», «NORMAL», «HIGH».
7. Виводити дані з кнопки: «Pressed» або «Released».
8. Додати обробку двох аналогових входів та виводити обидва значення на OLED.
9. Відображати індикатор заряду батареї (імітація за значенням напруги).
10. Реалізувати анімацію рухомої стрілки по колу або квадрату.
11. Виводити поточну дату/час із модуля реального часу (наприклад, DS3231).
12. Створити екранну заставку, яка прокручується кожні 5 секунд.
13. Виводити лічильник, який збільшується натисканням кнопки.
14. Додати режим відображення графіка напруги в реальному часі.
15. Реалізувати текстовий інтерфейс для вибору режиму (через кнопки).
16. Виводити почергово символи з таблиці ASCII.
17. Додати меню з 3 пунктами та навігацію між ними.
18. Відобразити логотип навчального закладу у вигляді монохромного зображення.
19. Реалізувати обертання тексту на 90° (за допомогою setRotation()).
20. Виводити повідомлення «Занадто висока температура!», якщо значення перевищує поріг.

## Зміст звіту

Звіт має містити:

1. Завдання.
2. Обгрунтування алгоритму програми.
3. Текст коду програми.
4. Електричні схеми реалізованих рішень (фото, скріни і под.).
5. Висновки за результатами проведених досліджень.

## Контрольні запитання

1. Що таке OLED-дисплей?
2. У чому полягає принцип дії OLED-технології?
3. Які переваги OLED-дисплеїв над LCD-дисплеями?
4. Яка роздільна здатність найбільш поширеного OLED-дисплея на базі SSD1306?
5. Який інтерфейс зв'язку використовується для підключення OLED-дисплея до ESP32?
6. Які бібліотеки потрібні для роботи з OLED-дисплеєм у середовищі Arduino IDE?
7. Яка функція використовується для ініціалізації дисплея?
8. Для чого використовується функція display.clearDisplay()?
9. Як встановити позицію курсора на дисплеї?
10. Яка функція використовується для оновлення вмісту екрана після зміни даних?
11. Як встановити розмір тексту на дисплеї?
12. Як задати колір тексту на монохромному OLED-дисплеї?
13. Що відбудеться, якщо не викликати display.display() після виведення тексту?
14. Яким чином OLED-дисплей підключається до ESP32?
15. Яка стандартна I2C-адреса дисплея SSD1306?
16. Як дізнатися точну I2C-адресу OLED-дисплея у випадку сумнівів?
17. Що таке Adafruit_GFX і яку роль відіграє ця бібліотека?
18. Як на дисплеї виводиться число з плаваючою точкою (наприклад, 3.14)?
19. Як реалізувати виведення графічних елементів (лінії, прямокутники) на OLED-дисплей?
20. У чому полягає перевага використання OLED-дисплеїв у вбудованих системах?

## ПЕРЕЛІК РЕКОМЕНДОВАНИХ ДЖЕРЕЛ


1. Tertulien Ndjountche. Digital Electronics 1: Combinational Logic Circuits. Wiley-ISTE, 2021. 276 p.
2. Thompson Carter. Embedded Systems with C: Programming Microcontrollers for Real-World Application. Independently Published, 2025. 286 p.
3. Гулий В. Д., Жуйков В. Я., Рябенький В. М. Цифрова схемотехніка : навчальний посібник для ВНЗ / за ред. В. М. Рябенького. Львів : Новий світ-2000, 2020. 736 с.
4. Матвієнко М. П. Пристрої цифрової електроніки : навчальний посібник. Суми : СУМДУ, 2021. 392 с.
5. Godse A. P., Godse Dr. D. A. Digital Electronics. Vendors: Technical Publi-cations, 2021. 220 p.
6. Білинський Й. Й., Книш Б. П. Цифрова схемотехніка. Електронно-обчислювальні пристрої : навч. посібник. Вінниця : ВНТУ, 2021. 66 с.
7. Alexander Axelevitch. Digital Electronic Circuits - The Comprehensive View. World Scientific, 2020. 436 p.
8. Georgios Papanikolaou. Bare-Metal Embedded C Programming: Develop High-Performance Embedded Systems. Packt Publishing, 2024. 448 p.
9. Єсаулов С. М., Бабічева О. Ф. Аналіз, синтез і проектування цифрових систем керування : навчальний посібник. Харків : ХНУМГ ім. О. М. Бекетова, 2021. 187 с.
10. Jack Ganssle. The Firmware Handbook (Embedded Technology). Elsevier Science & Technology, 2024. 365 p.
11. Anthony Massa, Michael Barr. Programming Embedded Systems: With C and GNU Development Tools, 2nd Edition. O'reilly Media, Incorporated, 2022. 336 p.

*Електронне навчальне видання*

**Максим Олександрович Притула**
**Ігор Андрійович Дудатьєв**
**Богдан Олегович Пінаєв**

# ПРОГРАМУВАННЯ МІКРОПРОЦЕСОРНИХ СИСТЕМ

**Лабораторний практикум**

Рукопис оформив *М. Притула*

Редактор *Т. Старічек*

Оригінал-макет виготовила *Т. Старічек*

Підписано до видання 20.02.2026 р.
Гарнітура Times New Roman.
Зам. № Р2026-021.

Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95,
м. Вінниця, 21021.
press.vntu.edu.ua;
Email: irvc.vntu@gmail.com
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.